12-31-2006

An architecture for intelligent health assessment enabled IEEE 1451 compliant smart sensors

Donald Albert Nickles
Rowan University

An Architecture for Intelligent Health Assessment Enabled IEEE 1451 Compliant Smart Sensors

by

Donald Albert Nickles

A Thesis Submitted to the

Graduate Faculty in Partial Fulfillment of the

Requirements for the Degree of

MASTER OF SCIENCE

Department: Electrical and Computer Engineering
Major: Engineering (Electrical Engineering)

Approved:

In Charge of Major Work

For the Major Department

For the College

Members of the Committee:

Rowan University
Glassboro, New Jersey

ABSTRACT

Donald A. Nickles
An Architecture for Intelligent Health Assessment Enabled
IEEE 1451 Compliant Smart Sensors
2006/07
Dr. John L. Schmalzel
Master of Science in Engineering (Electrical)

As systems become increasingly complex and costly, potential failure mechanisms and indicators are numerous and difficult to identify, while the cost of loss is very expensive - human lives, replacement units, and impacts to national security. In order to ensure the safety and long-term reliability of vehicles, structures, and devices attention must be directed toward the assessment and management of system health. System health is the key component that links data, information, and knowledge to action. Integrated Systems Health Management (ISHM) doctrine calls for comprehensive real-time health assessment and management of systems where the distillation of raw data into information takes place within sensors and actuators. This thesis develops novel field programmable health assessment capability for sensors and actuators in ISHM. Health assessment and feature extraction algorithms are implemented on a sensor or actuator through the Embedded Routine Manager (ERM) API. Algorithms are described using Health Electronic Datasheets (HEDS) to provide more flexible run-time operation. Interfacing is accomplished through IEEE Standard 1451 for Smart Sensors and Actuators, connecting ISHM with the instrumentation network of the future. These key elements are validated using exemplar algorithms to detect noise, spike, and flat-line events onboard the ISHM enabled Methane Thruster Testbed Project (MTTP) at NASA Stennis Space Center in Mississippi.

ACKNOWLEDGEMENTS

First and foremost I acknowledge Dr. John Schmalzel for providing the opportunity to participate in this unique and cutting edge research endeavor and for his unwavering support as my graduate advisor. Special thanks are also in order for Dr. Shreekanth Mandayam and Dr. Anthony Marchese for serving on my thesis committee.

Any of the work on this project would not have been possible without the support of the National Aeronautics and Space Administration (NASA). I am grateful for the opportunity to conduct the majority of the research and development contained in this thesis onsite at NASA's John C. Stennis Space Center in Mississippi, working alongside accomplished engineers and scientists. Special thanks to Dr. William St. Cyr, Dr. Ramona Travis, Dr. Fernando Figueroa, Lester Langford, Chuck Thurman, and Randy Holland for providing access to facilities, equipment, and support during my residence.

There are also many individuals on the home front who deserve acknowledgement for fulfilling important, though less visible, roles. I acknowledge my mother for supporting me in my decision to continue my education and for encouraging me by showing interest in my work. I acknowledge my father for his guidance and direction that helps me to make the best decisions when faced with difficult choices. I acknowledge my loving girlfriend, Rebecca Vanderslice, for her love and support though I worked long hours and spent time far away from home. My thanks also go to the rest of my family and friends that have provided encouragement, support, and guidance throughout.

I also recognize my colleague Jon Morris, who has filled many shoes as friend, hurricane Katrina survivor, fellow graduate student, and point of contact at Stennis Space Center. I give my best wishes to Jon and my other comrades who also strove to complete their own graduate studies alongside of me: Daniel, David, Hector, Hussein, Justin, Lucas, Mark, Nate, and Rob.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1: INTRODUCTION

There is a rapid and observable acceleration in complexity in numerous modern engineering systems. This may be readily seen in integrated circuits, such as the latest Intel Pentium 4 processor that contains 1.3 billion transistors or digital cameras that contain as many as 10 million picture elements and rival the performance of their professional film-based counterparts. The same progression is also true for space vehicles, which--including launch, tracking, and support systems--contain millions, if not billions, of elements. The motivation for the focus on health assessment in Intelligent Sensors is in response to the need for an integrated approach to managing increasingly complex systems. The long-term goal is to provide continuous assessment of system state with accompanying advisory outputs suitable for automatic, real-time responses as well as slower, human-in-the-loop decisions. Integrated Systems Health Management (ISHM) offers further impetus for Intelligent Sensors by distributing sensor-related processing and health assessment of sensor signals to the sensor domain. The approach of ISHM is to add sufficient intelligence to all levels of the data acquisition and system control hierarchy to detect potentially harmful deviation from the operational norm and empower the system to take appropriate action to prevent catastrophic system failures. This need, coupled with significant advances in technology over the last few years (reliable high-speed networks, radiation hardened components, advanced microcontroller architectures, to name a few) provides the necessary building blocks to localize health assessment at the individual sensing nodes and supply useful data, information, and knowledge to system processes. Before addressing the design and

development of an intelligent, health-enabled Smart Sensor architecture, it is important to first review the objectives and architecture of ISHM and relevant technologies.

1.1 Integrated Systems Health Management

Successful ISHM requires smart sensors, smart processes, and smart subsystems that are fully integrated [1, 2, 3]. ISHM may also retain some form of high-level reasoning to improve the performance and functionality of the entire system by making effective use of the data, information, and knowledge (DIaK) made available by the smart components. The topology of that top level controller may include model-based reasoning, rule-based inference, statistical analysis, and empirical or expert knowledge [4, 5, 6, 7, 8] Fig. 1.1 illustrates the ISHM architecture proposed by Figueroa [3], which provides the motivation for the work within this thesis. This ISHM model has the potential for high performance because it is specifically designed to take advantage of the distributed computing potential of smart sensors, potentially offloading ISHM related processing from higher layers. There are important cost and reliability functions associated with moving a certain number of MIPS from a higher to a lower level of the architecture, even if the lower layer may have significantly reduced computational bandwidth.

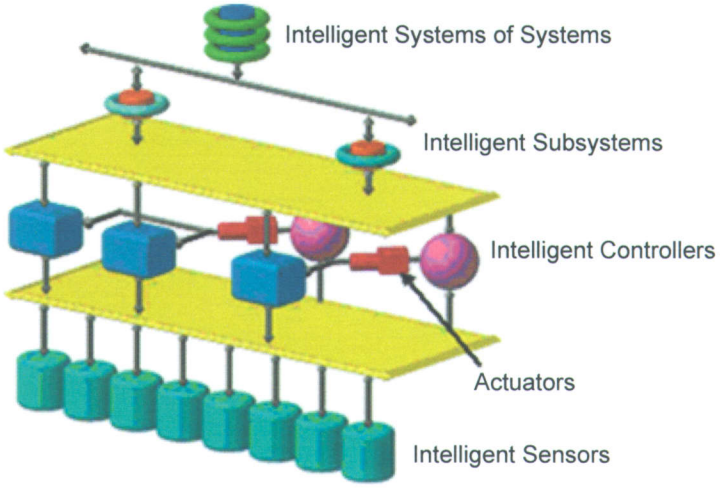


Figure 1.1: Overview of Integrated System Health Management showing critical components and hierarchy as adapted from [3].

Suppose a non-ISHM hypothetical system requires n CPU cycles in order to convert all analog transducer data into engineering values, and present the results to an operator in graphical and numerical form. For a small system, this would be contained within a single computer, either a general purpose machine (i.e., Intel based CPU), or possibly a data acquisition module with a virtual instrument interface running on a workstation computer. Now add the most basic elements of ISHM such as computing statistics, $2(f_s)$ DFT spectra, and applying one or more digital filters on each of the sensor feeds. In this case, the net complexity of each individual sensor running these algorithms is only $O(N) + O(N \log N) + O(N^c)$. It is assumed that the basic statistical functions (mean, variance, etc) have linear complexity, the DFT is computed using a radix-2 FFT approximation, and the variable c denotes the order of the digital filter polynomial. Thus, each individual sensor (assuming all sensors use the same configuration) requires a fixed amount of computational power. (Note this is the ideal case, as there typically is a mixture of low bandwidth and high bandwidth sensing nodes.) Nevertheless, the individual computing need summed across all sensors adds a multiplier of m to the equation developed above. Performing these operations in a modest real-time environment for even a dozen sensors gives rise to a dominating data processing load. While the network capable smart sensor is much less powerful than even a low-end desktop computer, it possesses enough horsepower to offload the processing within its scope using its onboard microcontroller. This diverted energy can then be invested in tools to merge those features, derive inferences, make decisions, and respond to events.

A fundamental assertion of this work is that the sensors and actuators participating in ISHM intercommunicate with other elements using *IEEE Standard 1451 for A Smart Transducer Interface for Sensors and Actuators* [9]. For clarity, a "Smart Sensor" is defined to be a sensor

that conforms to elements of the IEEE 1451.X family of standards, and an "Intelligent Sensor" is a Smart Sensor with added ISHM capabilities.

IEEE 1451 is unique in that it defines a unified and object-oriented model for encapsulating commercial off the shelf (COTS) networked transducers and actuators. Objects conforming to the standard may be utilized on different underlying networks or with different types or classes of transducers by simply changing the respective hardware abstraction layer and physical interface. These attributes make it easy to scale sensor networks and dramatically lower maintenance costs associated with sensor maintenance, replacement, and deployment.

ISHM with Intelligent Sensors is necessary to achieve the most effective use of DIaK, and as we will see in the next section, ISHM makes uses of lessons learned from other established health management schemes to do this.

1.2 Survey of Alternative Health Management Approaches

Health Management is a recent phrase that is an extension to the more familiar Condition Based Maintenance (CBM) or Reliability Centered Maintenance (RCM). In all cases, it expresses the intent of improving On Condition Maintenance (OCM). However, the method by which ISHM functions and provides useful services is novel. A few implementations of health management systems currently in use include the US Navy's Integrated Bridge System (IBS) for missile destroyers, NASA's Integrated Vehicle Health Management (IVHM) for spacecraft, and Open System Architecture for Condition Based Maintenance (OSA-CBM) intended for a broad suite of military, industrial, and aerospace systems. While all of these schemes present an approach to system integration and health management, they were developed to support specific vehicles, devices, or platforms. A brief overview of IBS, IVHM, and OSA-CBM is provided below.

Integrated Bridge Systems integrate navigational, steering, and propulsion systems to improve the efficiency and effectiveness of the bridge [10]. This includes everything from automatically updating paperless charts to detail information and control of the engine and machinery rooms to ergonomic and user friendly touch screen displays, all interconnected over a digital network. While this system provides a limited amount of health management, its primary purpose is to integrate systems to provide better and more effective command and control. IBS works in conjunction with Voyage Management System (VMS) and Steering Control Console (SCC) to accomplish this. IBS is at a maturity level sufficient to allow deployment on Nimitz class carriers (CVN-74) and Arleigh-Burke missile destroyers (DDG-51) in the US Naval Fleet.

Integrated Vehicle Health Management (IVHM) is jointly developed by NASA Ames Research Center (ARC) and NASA Kennedy Space Center (KSC). IVHM has been used in a variety of missions and programs, including Space Shuttle, Deep Space-1, X-33, X-34, and X-37 [11, 12]. IVHM seeks to minimize human in the loop interactions by making more operations autonomous. IVHM makes use of Vehicle Management Software (VMS) to gather sensor data and vehicle operating state. Livingstone [13] is used for decision making and diagnosis in certain IVHM implementations. Both Livingstone and VMS run as separate processes within the Vehicle Management Computer under VxWorks [12].

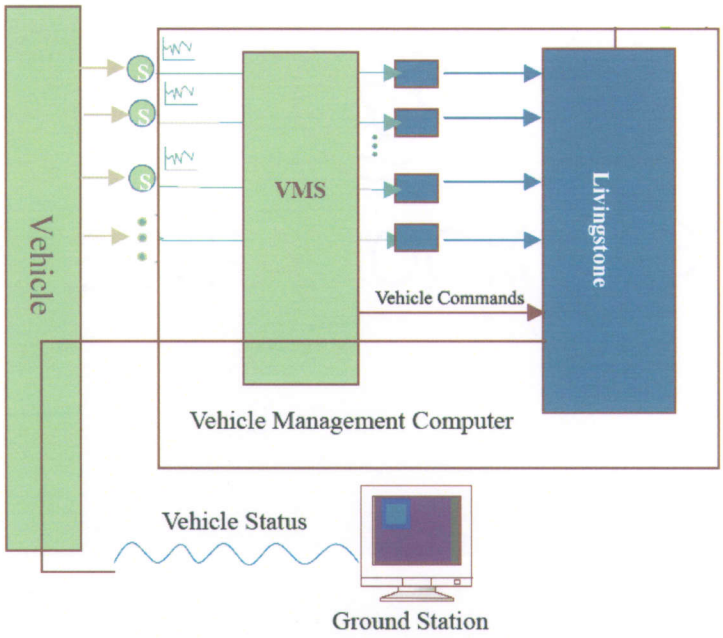


Figure 1.2: IVHM information flow and processing diagram [12].

IVHM implements health management by distilling raw data into *health summary information* accessible to Livingstone, that in turn makes decisions based on expert vehicle knowledge, vehicle system models, and vehicle state information [11]. The architecture for formulating health summary information consists of distributed data acquisition elements, and special high density solid state *health nodes* that intercommunicate with sensors and other health nodes [11]. The sensors report data over a digital bus to the health nodes, which then distill it into information that is communicated to Livingstone. This information may also be intercepted by other health nodes. One advantage of the ISHM approach described in this thesis is that health-related processing is distributed and can take place everywhere in the system instead of in functionally separate units.

OSA-CBM is a recent integrated health management architecture, which has adopted an open architecture concept. Open architecture is a systems engineering approach that aids the integration and interchangeability of components from multifarious sources and contains

publicly available interface and compliance specifications. This approach promises work that is nonproprietary with lowered access barriers to interested developers or users. This is evident in the OSA-CBM consortium as membership includes major players from defense, aerospace, industrial, and research organizations. OSA-CBM is still under active development [14, 15], and as such represents the closest counterpart to ISHM. The hierarchical view of OSA-CBM is presented in Fig. 1.3 as depicted by Thurston and Lebold [16].

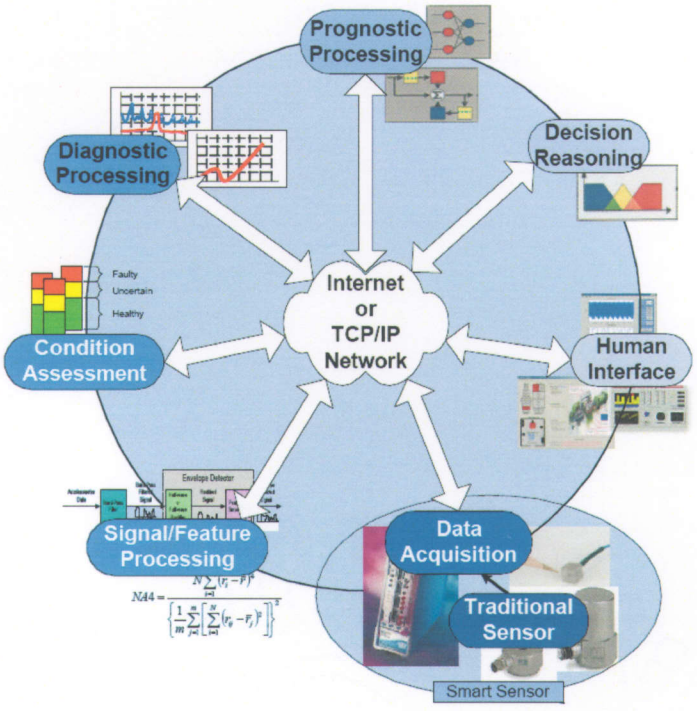


Figure 1.3: OSA-CBM hierarchy and interconnectivity diagram [16].

The goal of OSA-CBM is to develop a hardware and software technology independent architecture that specifies interfaces and transactions between objects in a condition based maintenance system. The data model is object oriented, and conforms to UML specifications.

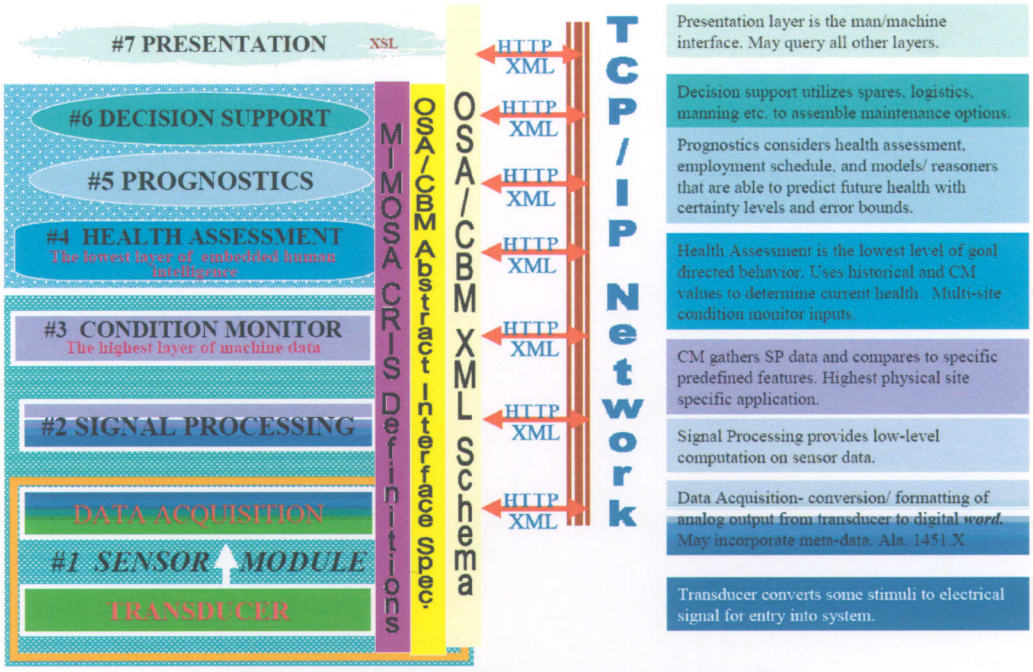


Figure 1.4: OSA-CBM structural overview.

It is also important to note OSA-CBM accommodates distributed data acquisition architectures such as that afforded by IEEE 1451. However, the primary differences between OSA-CBM and ISHM lie in the application of the health assessment, monitor, signal processing, and network communication schema. OSA-CBM uses CORBA as the common communication link; thus every OSA-CBM element will have a CORBA interface. While this may facilitate integration with some already existing elements, CORBA is notoriously inefficient in real-time operations. In contrast, ISHM takes a simplified approach, and uses IEEE 1451 as the common message service for all intelligent elements.

The Manufacturers and Information Management Open Systems Alliance (MIMOSA) has developed a protocol for data exchange between asset management systems [17]. This protocol is the Common Relational Information Schema (CRIS), which is essentially a relational database that supports interoperable CBM technology systems within OSA-CBM. The interaction between the levels of the hierarchy shown in Fig. 1.3, and the underlying structures

are shown in Fig. 1.4, per the OSA-CBM Consortium's Milestone 4 Review [18]. This database (left side of Fig. 1.4) is mapped to the operational behavior (right side of Fig. 1.4). The Common Object Request Broker Alliance (CORBA) model is used as middleware in between the functional levels, and provides a standardized interface to the underlying communication infrastructure.

1.3 Advantages of Integrated System Health Management

Now that the predominant health management and system integration architectures have been briefly discussed, it is important to ask what ISHM offers that is not currently available from existing implementation of IBS, IVHM, or OSA-CBM. Part of the answer considers the qualifications of the Stennis Space Center (SSC) ISHM development team partners and their incorporation of the latest sensor technologies and integration tools. The SSC ISHM design and development alliance includes autonomous systems expertise (PI, Figueroa) in the context of the NASA SSC rocket engine test stands, combined with hardware experts from NASA KSC, event detection algorithm experts from NASA Glenn Research Center (GRC), seasoned IVHM veterans from NASA ARC, as well as academic teams from Southern Illinois University at Carbondale (SIU), and the instrumentation and sensor expertise of Rowan University. These strategic alliances give the SSC ISHM development effort a distinct and definite advantage over the competition; many years of practical experience with health management, event detection, and system development will help ISHM evolve to become the next generation of health management. Another significant advantage is the efficient and streamlined ISHM architecture (refer to Fig. 1.1) that incorporates the substantial benefits of Smart Sensors and Actuators to actively participate in the health management process. Distributed and networked "smart" data

acquisition is the future of instrumentation technology, and distributed "intelligent" instruments enhanced with health assessment capability represent a critical milestone in the future of health management.

1.4 Rocket Engine Test Stand (RETS) ISHM Development at NASA's John C. Stennis Space Center

Current ISHM development is centered at NASA Stennis Space Center in Hancock County, Mississippi. SSC has a very decorated history, beginning in 1961 as Mississippi Test Operations (MTO). The establishment of MTO is a direct result of President John F. Kennedy's historic commitment to sending man to the moon within a decade. The original mission of MTO was to support the Apollo program as NASA's only national facility for rocket engine testing. On July 1st, 1965, MTO was renamed the Mississippi Test Facility (MTF), and just short of a year later in April of 1966 the first Saturn V 1st- and 2nd-stages were successfully tested. The first stage is powered by five F-1 engines each capable of producing 1.5 million pounds of thrust using RP1-LOX. The second stage rocket is powered by five J-2 engines[1] and is capable of producing over one million pounds of thrust using LH2-LOX. This milestone led to the successful launch of Apollo 11 on July 16th, 1969, only three years after the first engine test at MTF. MTF continued supporting the NASA mission and in 1971 was designated the testing facility for the Space Shuttle Main Engine (SSME). However, by this time the facility had grown to include NASA's Earth Resources Laboratory, so the facility was renamed the National Space Technology Laboratories (NSTL) in 1974 to reflect this expanded mission. Soon after, other offices including the Naval Oceanographic Office (NOO) moved to NSTL. This part of NSTL history is

[1] It is interesting to note that with President Bush's recommitment to a lunar return, an updated version of the venerable J-2 engine, code named J-2X, is being actively considered for the new Crew Exploration and Cargo Launch Vehicles (CEV, CaLV).

commemorated by the first launch of the Space Shuttle Columbia (STS-1) on April 12th 1981. In February of 1988, NSTL celebrated the 1000th SSME test, and soon after in May the NSTL was renamed John C. Stennis Space Center by Executive Order of Ronald Reagan to honor long-time Mississippi Senator John C. Stennis [19]. At this time there are two dedicated test facilities for testing SSME (A1, A2), as well as the original test facility (B1, B2) from the Apollo program. Over the next decade, additional test facilities were constructed to test components (turbopumps, SSME blocks, igniters, etc) for the SSME and other emerging engine technologies. Current test programs include SSME flight certification, flight certification for the expendable RS-68 used in Boeing's Delta Launch System, and development test programs including hybrid components and the Integrated Powerhead Demonstrator (IPD).

Stennis Space Center is a unique facility that performs all testing and flight certification for engines used in manned space flight, as well as providing services to private industry. Currently there are two complexes that support testing of large scale engines and assemblies, denoted as Rocket Engine Test Stand (RETS) A and RETS B. Their capabilities are listed in Table 1.1. Each RETS contain thousands of sensing nodes, which measure temperatures, pressures, strains, gas spectrophotometry, forces, vibration, acoustic emissions, and also records video of both the test articles and the RETS infrastructure.

Table 1.1: Test Complex A & B specifications.

Designation	Thrust	Dynamic Loading	Cooling	Oxidizer	Propellant	Features
A1	1.5M-lb	1.1M-lb Vert / 700k-lb Horiz	220k-gal/min	LOX	LH2	Gimbal
A2	1.5M-lb	1.1M-lb Vert / 700k-lb Horiz	220k-gal/min	LOX	LH2	Diffuser
B1	13M-lb	11M-lb Vert / 6M-lb Horiz	330k-gal/min	LOX	LH2/JP	Reconfigurable
B2	13M-lb	11M-lb Vert / 6M-lb Horiz	330k-gal/min	LOX	LH2/JP	Reconfigurable

Currently both positions of RETS A are used in SSME flight certification. During a test of an SSME on A1, the engine runs for a full 8 minutes and 40 seconds – the same run profile required for the Space Shuttle to reach low Earth orbit. Extra propellant is supplied to the Test Complex

during this event to allow a full uninterrupted burn. The engine is capable of 330,000lbs of thrust in atmospheric conditions. Test Stand B is the largest of the RETS, standing over 350ft tall and having the highest load capacity. B2 is currently occupied by the RS-68 engine test program. The RS-68 uses LH2 for fuel and LOX for oxidizer, similar to the SSME, but with a much greater rated output of 600,000lbs. Besides the heavy-duty, earth-shaking, rain-making engines in RETS A and B, there is also the RETS E Complex that houses several bays for testing hybrid engines and components. The capabilities of RETS E are provided in Table 1.2.

Table 1.2: E Complex capabilities chart.

Cell	Thrust	Angle	LOX (gal @ psig)	LH2 (gal @ psig)	LN2 (gal @ psig)	GH2 (cu. ft @ psig)	GN2 (cu. ft @ psig)	He (cu. ft @ psig)	
1-1	750k-lb	Horizontal	48,240 @ 165 - 28k	75,653 @ 33 - 8.5k	28,000 @ 165	1,875 @ 15k	2,750 @ 4.5k - 15k	1,515 @ 4.5k	
1-2	60k-lb	Horizontal +10°	48,240 @ 165 - 28k	75,653 @ 33 - 8.5k	28,000 @ 165	1,875 @ 15k	2,750 @ 4.5k - 15k	1,515 @ 4.5k	
1-3	60k-lb	Horizontal +10°	48,240 @ 165 - 28k	75,653 @ 33 - 8.5k	28,000 @ 165	1,875 @ 15k	2,750 @ 4.5k - 15k	1,515 @ 4.5k	
					RP-1 (gal @ psig)			H20 (gal @ psig)	GOX (gal @ psig)
2-1	100k-lb	Horizontal +10°	13.500 @ 150 - 9.3k	19,500 @ 400 - 4k	3,490 @ 1.8k-9k	1,500 @ 6.6k	3,122 @ 5.6k - 15k	145 @ 6k	1,375 @ 4.5k
				JP (gal @ psig)	H2O2 (gal @ psig)	DI H20 (gal @ psig)			
3-1	60k-lb	Horizontal	700 @ 60 - 2k	1500 @ 3.5k	2,500 @ 3.5k - 4.5k	2,800 @ ATM	151.9 @ 6k		
3-2	25k-lb	Vertical	700 @ 60 - 2k	1500 @ 3.5k	2,500 @ 3.5k - 4.5k	2,800 @ ATM	151.9 @ 6k		

It is apparent that RETS E is very versatile and is able to handle a large amount of fuels, oxidizers, and thrust capacities with infrastructure for testing high-pressure turbo pumps, injectors, igniters, etc. Another reason the E Complex is important is that its role for propulsion development is most compatible with ISHM development. In conjunction with the Data Acquisition and Control Systems (DACS) Laboratory, new systems such as ISHM may be connected in tandem with RETS E instrumentation and control systems to perform validation, verification, and ultimately gain confidence before a site wide or RETS wide deployment. The Stennis Space Center test suite does not end here, however. For small engines and boosters, there is the Portable Rocket Engine Test Stand (PRETS), which is a trailer mounted RETS that can test horizontally mounted engines with up to 1,000lbs thrust output. The PRETS is ideal for demonstration of engines or instrumentation/control systems, as well as small device trials.

Rocket engine testing is a large responsibility of SSC, and there is an obligation to provide theses services in a cost effective and efficient manner. As SSC looks to the future,

there is a need to enhance reliability, provide high-quality data with quicker turn around between tests, and to integrate some measures of autonomy into the test environment to minimize loss or waste in the event of a failed test due to infrastructure or test article fault. ISHM is a viable answer to this need, and will help deliver the Test Stand of the Future at SSC. The proposed technology maturation cycle is given in Fig. 1.5, and shows the eventual extension beyond the test facilities and into launch vehicles, exploration vehicles, and the International Space Station.

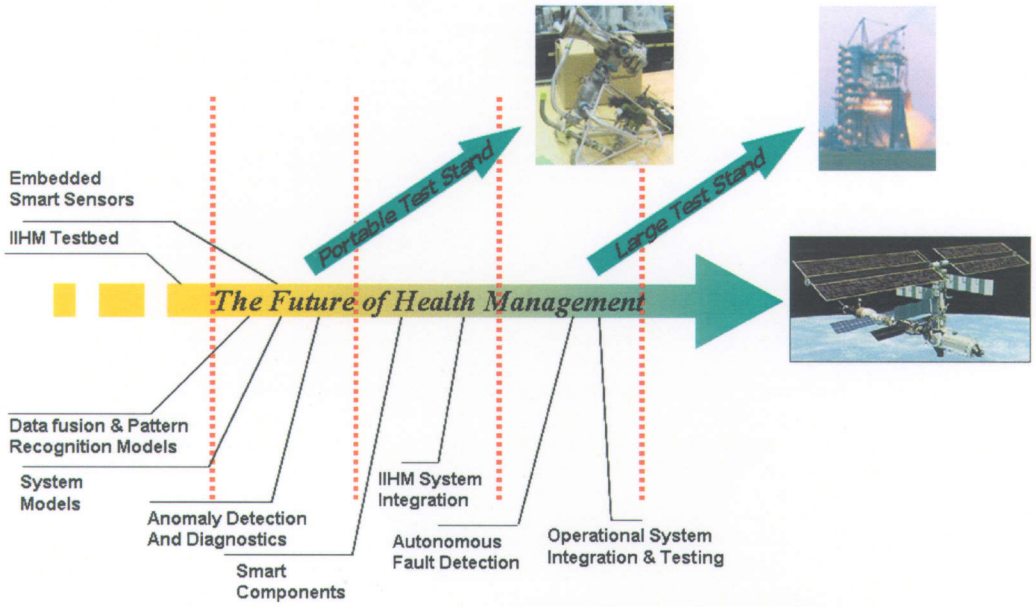


Figure 1.5: ISHM Technology Maturation Roadmap.

The starting point for ISHM is the PRETS. The reasons for starting on PRETS include utilization of a platform that is not currently part of a major test operation, support of rapid development due to its relative simplicity (limited number of sensors, processes, subsystems), and mobility for offsite demonstrations. Simultaneous with PRETS integration is the implementation of an ISHM on RETS E3 operating in shadow mode to fulfill validation and verification requirements. Operation on the other RETS can be simulated with the wealth of historical data that is available. A piping and instrumentation diagram (PID) of the PRETS is included in Appendix A for reference.

1.5 ISHM Core Competency: Smart Networked Sensors

A key opportunity for a state-of-the-art ISHM is the utilization of Smart Sensors. Smart Sensors differ from traditional legacy sensors in that they are sensors that are combined with local data acquisition and adhere to some variant of the IEEE 1451.X standard family. A Smart Sensor contains an embedded microcontroller and communicates over a digital network. The Smart Sensors discussed in this thesis conform to the IEEE 1451.1 Standard for Smart Sensors and Actuators. New opportunities arise when local computing power is available at the sensor. Some have proposed to use a generic smart sensor's computing power to do anything from multi-sensor data fusion [20] to ubiquitous computing and ambient intelligence [8]. In this work, the additional computing capacity is used to provide health-enabled functions to allow the smart sensor to perform as an *intelligent* sensor in an ISHM architecture. Fig. 1.6 shows the functional comparison between Intelligent Sensors, Smart Sensors and their traditional counterparts.

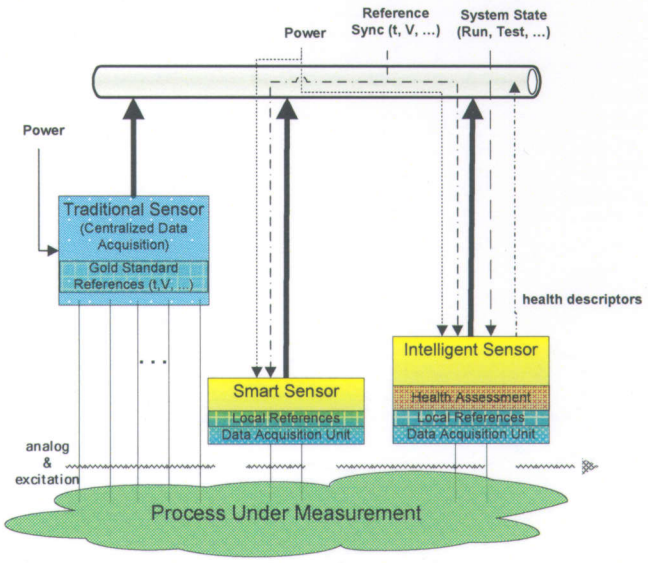


Figure 1.6: Fundamental comparison of Intelligent, Smart, and traditional sensors.

The most important difference between Smart Sensors and traditional sensor configurations is in the key operations of data acquisition, data conversion, and communications. Intelligent Sensors

are Smart Sensors with information generation capability, providing value-added descriptors about the measured signal(s). The discussion will begin with the more common traditional sensor, and then turn towards the Smart Sensor/Intelligent Sensor alternatives.

1.5.1 Traditional Sensor Systems

As shown in Fig. 1.6, traditional sensors rely on centralized voltage and time references. Multiple channels are captured and held simultaneously by a sample-and-hold circuit, and then converted to digital code values using an analog-to-digital converter. Infrastructure requirements usually require that this equipment is housed at some central location that is not near any one particular sensor location. Long runs of analog signals require careful conditioning and a large amount of shielded cabling. This corresponds to an expensive increase in the analog amplification and filtering requirements to extract minute sensor signals. Time and voltage references are located at this central facility, and provide the timestamps and references for all measurement channels. The sensing configuration is static, as the addition or modification of sensors requires available channels in the processing facility and independent wire runs from the facility to the terminating location in the field. In addition, other features such as sampling rate and post processing options are limited to the capabilities of the central facility. The goal of Smart Sensor technology is to turn this static instrumentation environment dynamic.

1.5.2 The Smart Sensor/Intelligent Sensor Advantage

The Smart Sensor is an entirely distributed approach to data acquisition and measurement, where each Smart Sensor node contains the ability to measure one or more attached transducers, convert the reading into a digital value, and transmit that digital value across a digital bus or network. On one hand the Smart Sensor solves an entire set of problems existing with the traditional sensor systems. Cabling is reduced to a minimum, and using a combined power and

signal technology, such as power-over-Ethernet (POE), each sensor merely needs a loop of Cat5 cabling to be fully operational. Faults in cabling due to environmental damage are easier to detect on the digital network, and network switches may be strategically placed to minimize the amount of wiring, while providing redundant connectivity. Furthermore, the Smart Sensor Standard is object oriented; allowing the sensor network to be dynamic as new sensing nodes can be provisioned very easily. The Smart Sensor Standard allows for multiple transducers per sensor node, and there is no requirement that each transducer be of the same type. This can allow redundant physical connections with minimal infrastructure duplication. In order to provide the basic instrumentation functions of conversion and time alignment, each Smart Sensor element has its own highly precise measurement and time reference. Smart Sensors are manufactured using inexpensive off-the-shelf components that only result in an incremental increase in costs. A few examples are high speed and high accuracy data converters [21], low drift clock oscillators [22], and voltage references with variation on the order of 2ppm [23]. The only complication is in the out of box calibration of the voltage references and clock oscillators. Even though the performance characteristics of the individual components are outstanding, a problem arises with the overall component variability among a distributed acquisition system. Standards such as IEEE 1588 [24] address the timing issue, and implementations reportedly have achieved nanosecond accuracy between nodes, with hardware implementations available from Intel [25]. Converter reference, onboard temperature sensor calibration, and ADC characterization take place during initial calibration and commissioning, and the Smart Sensor checks periodically to determine if the calibration needs to be further adjusted and to ensure the analog components are within specification. Smart Sensors can be interfaced using a variety of network standards and architectures including Ethernet, CAN, RS-485, RS-422, 802.11a,b,g,

Bluetooth, and more. Proper choice of network architecture can minimize wiring requirements, increase reliability, and support a large array of Smart Sensors with high data throughput. Intelligent Sensors embody Smart Sensor functionality, with the added function task of performing real-time health assessment.

1.5.3 IEEE Standard 1451: Standard for Smart Sensors and Actuators

The IEEE Instrumentation and Measurement Society has taken the initiative to develop standards describing key components of Smart Sensor technology. To date, the IEEE 1451.X family of standards is the only published standard for interfacing Smart Sensors and Smart Actuators. This represents an important advance over previous uses of the term *smart,* which simply indicated the presence of a microcontroller and the mapping of analog transducers and actuators to a digital communications bus. The IEEE 1451.X family of standards describes a Smart Sensor or Actuator as a device that consists of a number of key elements. Those elements are a transducer interface module (TIM), network capable application processor (NCAP), transducer independent interface (TII), and transducer electronic datasheets (TEDS). Fig. 1.7 graphically shows the relationship between these elements.

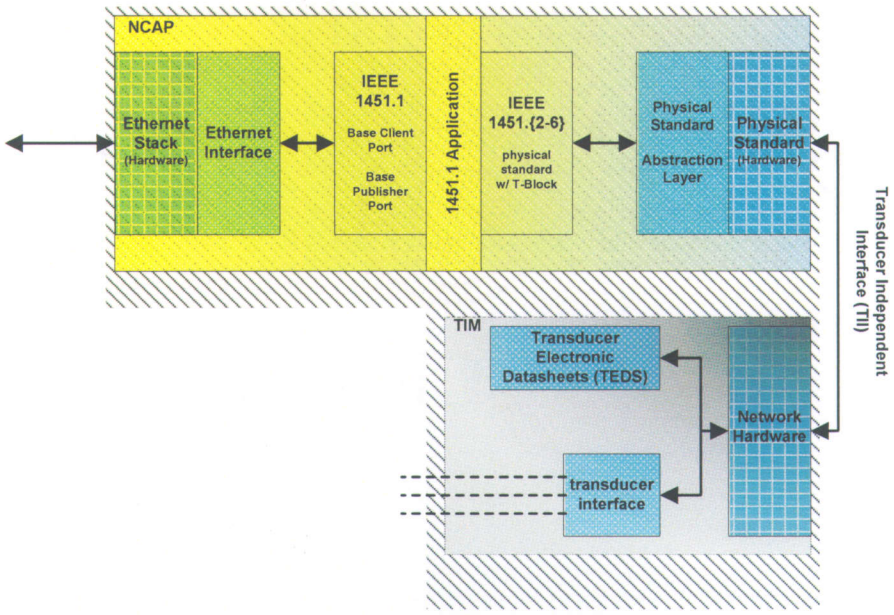


Figure 1.7: Smart Sensor component framework for IEEE Standard 1451.

The NCAP is the component that connects an object to a 1451 compliant network. The object may be a 1451 compliant sensor (consisting of one or more TIMS), actuator, data archiving tool, or model-based reasoning tool. The boundary between object and NCAP can be software or hardware, depending on what is most suitable for the application. In the case of a hardware boundary it is the TII that connects NCAP to TIM. IEEE 1451.2 thru 1451.6 provides physical standards for establishing and interfacing the TII. See Table 1.1 for a listing and description of these standards. Currently there is support for TII using serial peripheral interface (SPI), CANopen, Bluetooth, and Zigbee. There is also support for a mixed mode interface, where the TII contains a mixed mode interface (MMI) supporting both digital and analog signals. The MMI provides for TEDS, connected via a digital bus, and measurement and control interface over analog channels.

The TIM contains the necessary instrumentation elements to process and convert raw analog transducer signals. This includes amplification, filtering, digitization, and linearization. In a functional sense, the TIM embodies the legacy sensor. Any given Smart Sensor may contain

up to 255 TIMs. The TIM also contains the TEDS, which contain basic TIM information (version, manufacturer, revision, etc) and advanced TIM information for the data acquisition and interpretation of TIM output. As such, the TIM is self describing. Each physical standard also establishes applicable and relevant TEDS. There are TIM-specific TEDS that include minimum and maximum sensing capability, accuracy, and the calibration curve for converting to engineering units. The NCAP also has a TEDS block that contains information about the sensor hardware, its location, and manufacturing information. There are extensions to TEDS that allow users to add relevant information that is not part of an existing TEDS template. Chapter 2 discusses TEDS further. Additional details on the TEDS data structure definition and example TEDS can be found in [26] and [27].

Table 1.3: List of 1451 physical layers. These are used to define the connection between NCAP and TIM, along with TIM specific TEDS and NCAP visible classes.

Phys Std	Description
1451.2	TIM←→NCAP + TEDS for enhanced SPI w/ revision for UART
1451.3	Multidrop TIM←→NCAP Network + TEDS
1451.4	TIM←→NCAP interface with analog and digital modalities for current loops, microphones, thermocouples, etc.
1451.5	Wireless TIM←→NCAP using 802.11, 802.15.1, or 802.15.4
1451.6	TIM←→NCAP using open source CAN

IEEE 1451.1 [9] is the portion of the standard that defines the object model for the Smart Sensor. As such, it is minimally required to have an object (sensor, actuator, etc) participate on an IEEE 1451 sensor network. IEEE 1451.2 [28] specifies the details of interfacing TIMS over a TII that consists of SPI or UART. IEEE 1451.3 [29] defines TIM/NCAP interaction over a TII that is a multidrop serial (RS-422) interface. IEEE 1451.4 [30] defines a mixed mode interface, as mentioned earlier, which is ideal for thermocouples, microphones, and other voltage or current loop devices and represents the simplest and least costly way to add a simplified version of IEEE

1451 functionality to a sensor. However, capabilities of such 1451.4 sensors are limited and mean they are unlikely to possess sufficient computing power to meaningfully contribute to the ISHM architecture described here. Wireless TII is covered in IEEE 1451.5 [31], and contains specifications for popular wireless networks such as wireless Ethernet, Bluetooth, and Zigbee. IEEE 1451.6 [32] is the physical standard that supports a TII consisting of an openCAN bus.

1.6 Statement of the Problem

The next generation of ISHM will be greatly enhanced through the use of Smart Sensors in order to create Intelligent Sensors. Utilizing Intelligent Sensors minimizes the computational bandwidth at central nodes and increases reliability across the system due to their distributed nature. Intelligent Sensors, loaded with health assessment capability, increase the benefit substantially by determining measurement confidence and signal behavior characteristics at the earliest possible point in the instrumentation and measurement process. This translates to improved real-time performance and reliability.

This thesis presents a method for embedding and dynamically updating feature extraction routines and health assessment algorithms (herein collectively referred to as health assessment routines) used in health evaluation aboard a networked IEEE 1451 Smart Sensor. Some of the routines used in this work are adapted from previous researchers, but all routines are implemented in novel ways as part of embedded sensor-level health assessment to support ISHM's doctrine for pervasive intelligent elements. The ultimate goal of this work is to develop a method to dynamically distribute health assessment capabilities into Smart Sensors and Smart Actuators. Additionally, this groundwork can be evidence in support of an extension to IEEE 1451 setting baseline provisions for future intelligent elements.

The specific aims are to:

- Survey current implementations of sensor-level health management.
- Survey tools and techniques for implementing a complex real-time embedded microcontroller based, multi-channel measurement instrument.
- Develop an IEEE 1451 compliant Smart Sensor.
- Enhance the Smart Sensor with dynamic health assessment algorithms that operate on signals in real-time.
- Provide a machine readable health electronic data sheet (HEDS) that specifies algorithm parameters including limits, thresholds, reporting frequency, and interpretation of results.
- Provide capability for changing HEDS based on system operating state or based on the occurrence of events.
- Validate the Intelligent Sensor against canonical data to assess computational accuracy and document the performance of the Intelligent Sensor.

1.7 Scope and Organization

In order to develop a viable method for embedding health assessment routines into IEEE 1451 Smart Sensors, this thesis includes work on the following:

- Identification of RETS real-time requirements that an Intelligent Sensor running under ISHM must support.
- Develop an NCAP object on the sensor that communicates via Ethernet (IEEE Standard 802.3) to other NCAPs. Use Open1451 specifications [33] as guidelines.

- Develop APIs that allow firmware developers to interface health assessment routines to Intelligent Sensors with minimal effort. This includes unified methods for handling data streams, memory, time slices, and other operating system resources.
- Define Health Electronic Data Sheets (HEDS) that identify and characterize the operational details of health assessment routines running on an Intelligent Sensor.
- Development of remote download tools to update Smart Sensor firmware over the network.

The second chapter explores pertinent background, providing in-depth discussion of previous work with Smart Sensors, anomaly detection algorithms, and pertinent feature extraction routines. The third chapter details the approaches taken to meet the Intelligent Sensor development objectives. The fourth chapter is dedicated to the benchmarking and performance results of the completed intelligent sensor. Finally, the fifth chapter discusses the outcomes of this research, draws conclusions, and suggests areas of future work.

CHAPTER 2: BACKGROUND

This chapter begins by discussing prior technology that is pertinent to this work. Those topics include the role of sensors, actuators, and health nodes in IVHM, current development with IEEE 1451-compliant Smart Sensors, and a detailed review of the multipurpose Intelligent Sensor developed at Rowan University. The second part of the chapter discusses technology and techniques that have been researched and are implemented as part of this work, pursuant to the problem statement of Section 1.6. Specifically, this includes Smart Sensor implementation, Intelligent Sensor distributed health assessment routines, real-time embedded system performance, and Open1451 compatibility.

2.1 Prior Pertinent Technology

This section reviews in detail the three most important premises of this work. The first is the substantial contributions of IVHM, with a quick reflection on the necessity of health management. The next section discusses current advances in Smart Sensor technology. The first clause of the standard was published in 1997, and even though today few instruments are equipped with Smart Sensor connection points, there have been critical advances in the development and deployment of Smart Sensors. The final subsection then focuses on the multipurpose Intelligent Sensor development efforts at Rowan University in support of ISHM.

2.1.1 Integrated Vehicle Health Management

The most primitive method of health management is scenario planning. This type of health management is typified by the response to the Apollo XIII in-flight accident on April 13th, 1970. The events leading up to the flight accident were intermittent anomalous sensor readings, along

with unexplained variations in temperatures, pressures, and voltage levels triggered when the cryogenic circulation fans were activated [34]. Afterward, the reconstructed chain of events pointed to faulty valve electrical wiring that caused the #2 oxygen tank to explode. Without vigilant and predictive health management, anomalous situations had to progress until a component or system failure took place. Then, when the failure occurred, the only response was to run scenarios on a test bed in an attempt to determine likely causes and ways to recover from the failure. While the crew of Apollo 11 were able to bring the situation under control and return home, this method of manual diagnosis and recovery is insufficient for applications where systems are located at such distances that there are considerable communication delays, involve events that occur faster than humans can possibly respond to, or involve complex predictions of potentially destructive or seemingly unrelated trends occurring over long periods. The difficulty of trending is signified by an SSC test operator monitoring a routine oxidizer tank filling process, but given only his or her knowledge of the normal response, is unable to identify a subtle trend in the decay of a valve's response over a period of many months.

The Integrated Vehicle Health Management (IVHM) approach of the 90's had the goal of reducing—or even eliminating—manual screening of post-flight data to determine maintenance schedules. The first iteration of a Shuttle IVHM system was flown on STS-95 in October of 1998, which coincidentally also included the studies of space health effects on Senator John Glenn, who returned to space as the oldest astronaut. IVHM program implementations also include Thermal Expert System (TEXSYS) [35], Ground Processing Scheduling System (GPSS) [36], and Remote Agent Experiment (RAX), which flew onboard Deep Space-1. RAX is the most recent IVHM experiment; using Livingstone as the inference engine to provide among

other capabilities, an autonomous navigation and mission operations package known as *AutoNav* for Deep Space-1 [37].

The IVHM effort for the X-34 is of special interest. The X-34 was intended to be a pilotless aircraft, although it was never tested beyond towed or captive flight tests. It made use of IVHM technology and was loaded with diagnostic algorithms to detect component degradation and system level health monitoring. A highlight of this system is the capability for an operator to have access to high-level health information and inferences, along with the raw sensor data and justification for those inferences directly from the diagnostic algorithms [38].

To date very limited work has been done with Smart Sensors as a major component of IVHM, although vehicle impact IVHM has suggested the use of a non-IEEE 1451 Smart Sensor [39] that has onboard networking, distributed data acquisition, and signal processing capabilities. These developments are part of a concept demonstrator program between NASA and the Australian Commonwealth Scientific and Industrial Research Organisation (CSIRO) for the NASA Robust Aerospace Vehicle Program (RAV). The specific goal of the concept demonstrator is a system to detect damage and approximate collision energy to a vehicle's skin from impact by space debris. This is an interesting application, as it consists of an array of sensors numbering in the thousands, distributed processing, and system intelligence. The blueprint for in-flight IVHM consists of low-weight and low-power sensors, in a matrix of distributed processing Smart Sensors interconnected with a digital bus providing regenerative, auto calibration, and cross-check of data [11]. The distributed data processing is to occur in real-time. Special *health nodes* have software to perform trending and fault prediction. Based on the results of the trending and fault prediction analysis, health nodes are also responsible for fault isolation and adaptive mission planning/scheduling. This model differs significantly from that

proposed in ISHM. The principal difference is that ISHM with Intelligent Sensors establishes the health management doctrine in every node in the system, whereas IVHM only provides select nodes to perform health analysis. Another significant difference is in the ability of ISHM to adapt health management models (at the system, process, and sensor/actuator level) to system state, a capability that is not part of IVHM, which operates on the same model from launch to landing [12]. In IVHM, the diagnostic algorithms embedded in the health nodes provide fault isolation, health prognostics, pattern recognition tools, and sensor validation capabilities. Nodes also contain a model linked to the Livingstone reasoning engine. Self healing and self calibration are addressed through installation of sensors that contain multiple sensing transducers or parametric sensing capability. The fabric for sensor communications uses either Fiber Data Distributed Interface (FDDI) or MIL-STD 1773.

Although each of the IVHM applications described is different, the primary role of sensors in IVHM is typically that of a traditional sensor. Some of the later developments show progress towards sensors that resemble Smart Sensors, though not necessarily aware or compliant to the IEEE 1451 standard. Functionality resembling Intelligent Sensors is located in nodes separate from sensing/actuating functions and are fewer in number. The approach for ISHM based on Intelligent Sensors, which is the focus of this work, improves upon the existing technology by integrating the capabilities of the IVHM *health node* into the Smart Sensor and Smart Actuator, providing localized health management that contributes to system-wide health management.

2.1.2 Smart Sensors

Conventionally, the modifier *smart* is used to describe any device that contains an embedded microcontroller. For example, some dishwashers have features called *smartwash* where sensors

are used to determine the level of soil on dishes and apply the optimum wash cycle [40, 41]. *Smart Cards* contain the capacity to store, change, and present personal information in a small credit card form factor. Automobile airbags that adjust their deployment based on vehicle speed, and road intersections that adjust signal behavior based on traffic patterns are examples of smart systems. In contrast, this work tightens the definition of a *Smart Sensor* or *Smart Actuator* to be those devices that comply with the IEEE 1451.X Smart Sensor standards. The goal of IEEE 1451.X is to be implementation neutral, but at the same time provide a framework for transducer modules that can be easily interconnected and provide an automatic identification and self description capability for smart elements communicating over a network.

Smart Sensors are emerging in the market place. National Instruments has several product lines of IEEE 1451 compliant Smart Sensors that interface with the LabView environment as virtual instruments [42]. Endevco [43] also has a number of Smart Sensor products. The latest developments from Honeywell include a line of equipment health monitors for detecting mechanical wear, fluid leaks, angle drift, fluid flow, and temperature change [44]. This product line is different in that these devices are intended to be installed as after market devices (as opposed to part of the standard installed instrumentation base) that evaluates a specific steady state health parameter. Sensitivity is adjusted manually at the unit, and a single digital channel is required for interfacing. In addition, independent research groups [45, 46] are developing Smart Sensor technology, which offer further insights into the flexibility and adaptability of IEEE 1451. Most of these efforts focus on IEEE 1451.4 TII/TIM interfaces due to their relative simplicity and pervasiveness in the marketplace, though other interesting instrumentation projects have been successfully accomplished with the other physical standards [48]. These developments all represent steps forward for Smart Sensors, however the existing

product suite do not meet the needs of an Intelligent Sensor—i.e., a health enabled Smart Sensor to support ISHM.

While there are significant efforts at creating and establishing a market base for Smart Sensors products, there is also a move to establish Smart Sensor networks in unique application areas. KDNet is in the process of developing the Telemetrics Testbed in Busan City, Korea. The Telemetrics Testbed [47] applies IEEE 1451 for monitoring and control of national infrastructure and surveillance applications. Highlights include real-time sensor systems that interact with pipelines, assess environmental pollution, and monitor structural integrity. The network is unique in that it is a city-wide WAN that supports many diverse sensing and actuating nodes, over both wired and wireless links. The network is centered around IEEE 1451.1, 1451.3, and 1451.5.

Yet another unique application is attributed to Engineering Development Corporation of Columbia, Maryland, who is developing Smart Sensors to evaluate oil casing health through miles of piping drilled into the ground. Their solution is based on IEEE 1451.2, where the individual sensors are inline and connected in series with the cabling, reporting the health of the casing as it penetrates the Earth.

The US Department of Homeland Security, in conjunction with the National Oceanic and Atmospheric Administration (NOAA), the Department of Defense (DoD), and Oak Ridge National Laboratory (ORNL) are in the process of testing an IEEE 1451 compliant sensor network at Fort Bragg, North Carolina [49]. The network contains sensor nodes that detect chemical, biological, radiological, nuclear, and explosive (CBRNE) threats. Within the near future it is likely that this, or a network similar to it, will be deployed on a national scale.

With all of this development effort, IEEE 1451 will soon have enough installed base to be recognized as a major player in instrumentation and measurement circles worldwide. In concert with this effort, a multipurpose Smart Sensor development platform has been developed at Rowan University. The purpose of this development platform is to sustain the development of Intelligent Sensors in support of the Stennis Space Center vision for ISHM.

2.1.3 Multipurpose Smart Sensor Development Platform

The development goals for the Smart Sensor development efforts at Rowan University are to create a device that contains the hardware necessary to support an IEEE 1451.1 interface, time synchronization, and measurement acquisition in a variety of configurations, and to incorporate components of health assessment in support of ISHM. That is, the first part of the effort seeks to develop a Smart Sensor; the second part is to develop an Intelligent Sensor. As such, it is possible for both technologies to mature in parallel.

The Smart Sensor development platform features a Rabbit 3000 main microcontroller running at 45MHz, and a secondary 8051F300 microprocessor running at 4MHz. The main CPU contains 512kB of parallel FLASH, 512kB of program memory, 256kB of battery-backed data RAM, and 8MB of serial FLASH. The RAM backup battery is estimated to last 30 years. The main CPU is interfaced to a Standard Micro-Systems Corporation (SMSC) Ethernet semiconductor with MAC and integrated PHY, supporting both Ethernet 10BaseT and 100BaseT over twisted pairs. Initial programming of the main CPU is accomplished through an onboard JTAG interface. With appropriate firmware, subsequent programming may be performed over the Ethernet network. The Smart Sensor is also compatible with IEEE 802.11af power over Ethernet (PoE) [50]. The use of PoE avoids additional wiring for power to the Smart Sensor. The secondary CPU has 16kB of memory, and is programmable via local connection to the

onboard Silicon Labs C2I interface or through the main CPU. The main CPU is also connected to the secondary CPU via serial peripheral interface (SPI).

The Smart Sensor core contains a versatile analog acquisition system based on the Analog Devices AD7794 Sigma-Delta analog to digital converter (ADC) [21]. The ADC can sample at speeds up to 470Hz, with a maximum effective resolution of 23bits. The analog front end consists of an input buffer with input range of GND+100mV to AVDD-100mV, and is most often used for devices such as strain gauges, thermocouples, or resistive temperature detectors to avoid loading the transducer. Alternatively, the input buffer may be bypassed to achieve another 60mV in the upper and lower range; however, this increased range is obtained at the expense of adding a dynamic load to the circuit. Internal gains of 1 thru 128, mod 2 are software configurable. The gain is achieved using an instrumentation amp which has excellent noise and linearity performance[2]. A precision 1.17V voltage reference (4 ppm/°C temperature coefficient) is integrated on the chip with the option to connect an external reference. Another valuable feature is a 100nA current source and current sink for detecting continuity of a transducer. As with the gain control, the burn-out currents are software configurable. The Smart Sensor is also capable of generating excitation currents of 10μA, 210μA, or 1mA, as well as bias voltages of half the analog supply voltage. Three analog inputs are provided for user connection. Fig. 2.2 shows the completed Smart Sensor, designated SNTS/ROME B.1.

[2] AD7794 datasheet specifications show 40nV RMS noise and 21 bits effective resolution when the part is set to a gain of 64. It is interesting to note that gains of 1 and 2 bypass the instrumentation amp and are performed digitally.

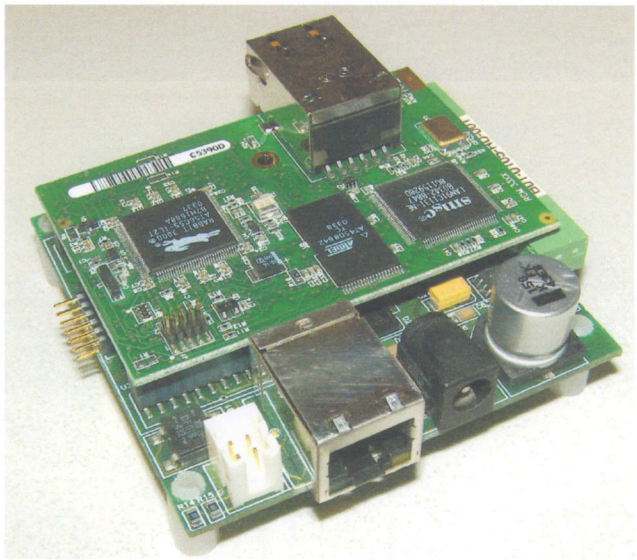


Figure 2.1: Smart Sensor Development Platform; overall size: 2.77'' x 2.86''.

The first application of the Smart Sensor development platform is for temperature sensing based on a thermocouple (TC) of type K. To measure temperature, a TC is connected to analog channel 1, and a second temperature transducer (typically, a semiconductor proportional-to-absolute-temperature (PTAT) device) to channel 2 for cold junction compensation. The details can be found in the application note "A Smart Networked Temperature Sensor for ISHM Applications" [51]. Table 2.1 summarizes the Smart Sensor's capabilities.

Table 2.1: Smart Sensor Capability Chart

Operating System	Micro C/OS-II
Bus width	8 bit
CPU clock	45MHz
Basic floating point operations (+, -, x)	350 ticks
Advanced floating point operations (÷, √)	900 ticks
FFT (1024-point)	33ms
Power consumption	130mW
Analog burnout current	100nA source / 100nA sink
Excitation current	10μA, 210μA, 1mA
Analog-to-digital converter	8-channel multiplexer, 24 bit ∑Δ, 470Hz sampling frequency, 1.17V reference
On-board health status	Monitor RAM battery voltage, onboard ambient temperature, analog voltage, reference voltage

2.2 Core Technology Objectives

The remaining sections of this chapter delve into the technologies that are to be adapted, refitted, and integrated into the Smart Sensor to make it an Intelligent Sensor in accordance with the goals of Section 1.6. In particular the health assessment/event detection routines provided by NASA GRC are reviewed and analyzed for integration into the Intelligent Sensor, followed by an object-oriented overview of IEEE 1451. Chapter 3 focuses on the actual adaptation, modification, and integration efforts; including the development of supporting technologies.

2.2.1 Health Assessment Event Detection Routines

The health assessment algorithms used in this work are tools that generate health estimates based on the detection of characterized signal events, and are a function of system, measurement modality, measurand, and system state. The algorithms in this section were developed by NASA GRC at Lewis Field in Ohio. GRC has conducted extensive research in event detection; the original motivation for the algorithms included here was a 1993 program for the ATLAS Centaur electric and pneumatic subsystems. The success of these algorithms resulted in further modification for integration into the Propulsion Checkout and Control System (PCCS), as well as for use in a system designed for air aspirated engine diagnostics developed by Arnold Engineering Development Center (AEDC). These diverse applications have resulted in mature algorithms that are valuable assets as tried-and-true health assessment tools in ISHM. Three event attributes were selected for integration into the ISHM development work described in this work:

- Noise,
- Spike,
- Flatline

The original GRC interface to these algorithms is either through a command file or direct user input, as shown in Fig. 2.3. Once the input commands have been read and parsed, the event signal processing (ESP) routines are invoked, pulling in the source data via the data access routine (DAR) interface. Once the raw data are preprocessed (smoothed, fitted, etc), the event detection routines specified by the user commands are executed. When analysis of the dataset is complete, any detected events are tabulated and reported to the user through the computer terminal and written to a log file for future reference.

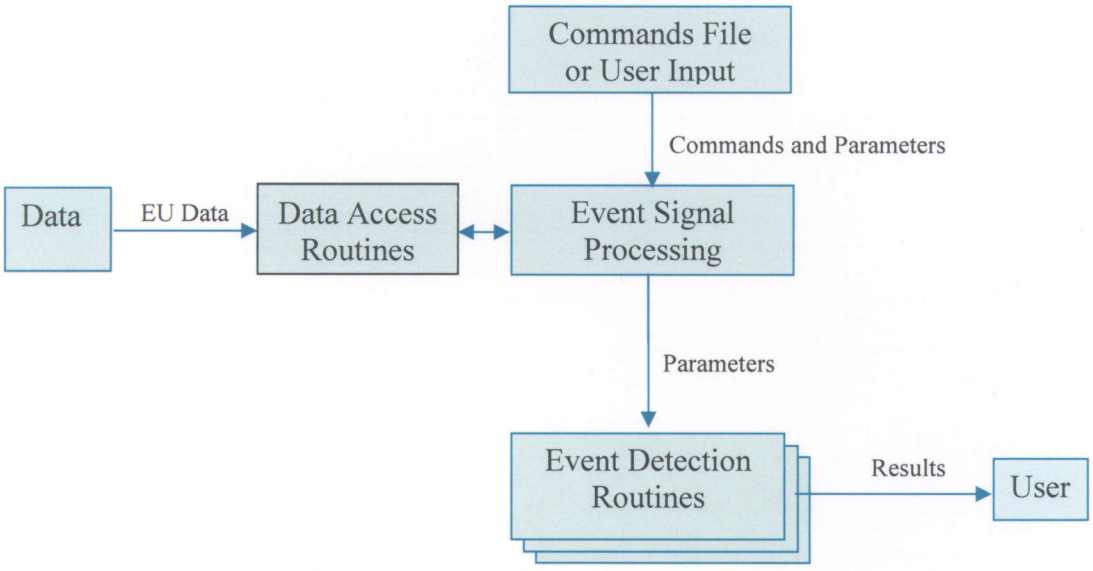


Figure 2.2: Top-level interaction for GRC event detection routines.

These routines are optimized for post processing of engine test data. Datasets are tracked by test ID and sensor ID. This is fundamentally different from the goal of this work (detailed in the Approach of Chapter 3) which is the need for algorithms in a real-time embedded Intelligent Sensor environment, where events are reported as soon as they are recognized rather than through offline batch processing. This realization is key to providing near real-time health assessment of systems, the ultimate goal of ISHM. To complicate matters, in a real-time application processing takes place with an emerging dataset, requiring substantial changes to the

data access methods, signal processing routines, and event detection routines in Fig 2.3. The following subsections cover the GRC signal processing and event detection routines in detail to foster an understanding of what it is they accomplish. This understanding will minimize the challenges of adapting them to the real-time Intelligent Sensor environment.

2.2.1.1 Smoothing

The smoothing function prepares the raw data prior to processing. Smoothing is used by the drift/level shift routines when performing the linear curve fit in Section 2.2.1.3. The flowchart for the GFC smoothing algorithm, *GF Smooth*, is shown in Fig. 2.4 and works in the following manner:

1. Define smoothing window (default 25 points) and divide the window into two equal halves. If the number of points is even, add an extra point. If there are not 25 data points, processing is aborted (recall that since the dataset is static, this will only happen once, at the end of the file).
2. Sum the upper half of the raw data window into a single variable
3. Update the upper half of the smoothed data array with the previously computed sum divided by the half window plus current position
4. Update the sum to include the next raw value
5. Repeat steps 3 and 4 for the points in the lower half of the smoothing window
6. Sum up the contents of the smooth window
7. For each data point, starting at the center of the smoothing window, and extending to the upper bound of the smoothing window, update each smoothed data element with the sum divided by the size of the smoothing window.
8. Update the sum to include the next data point

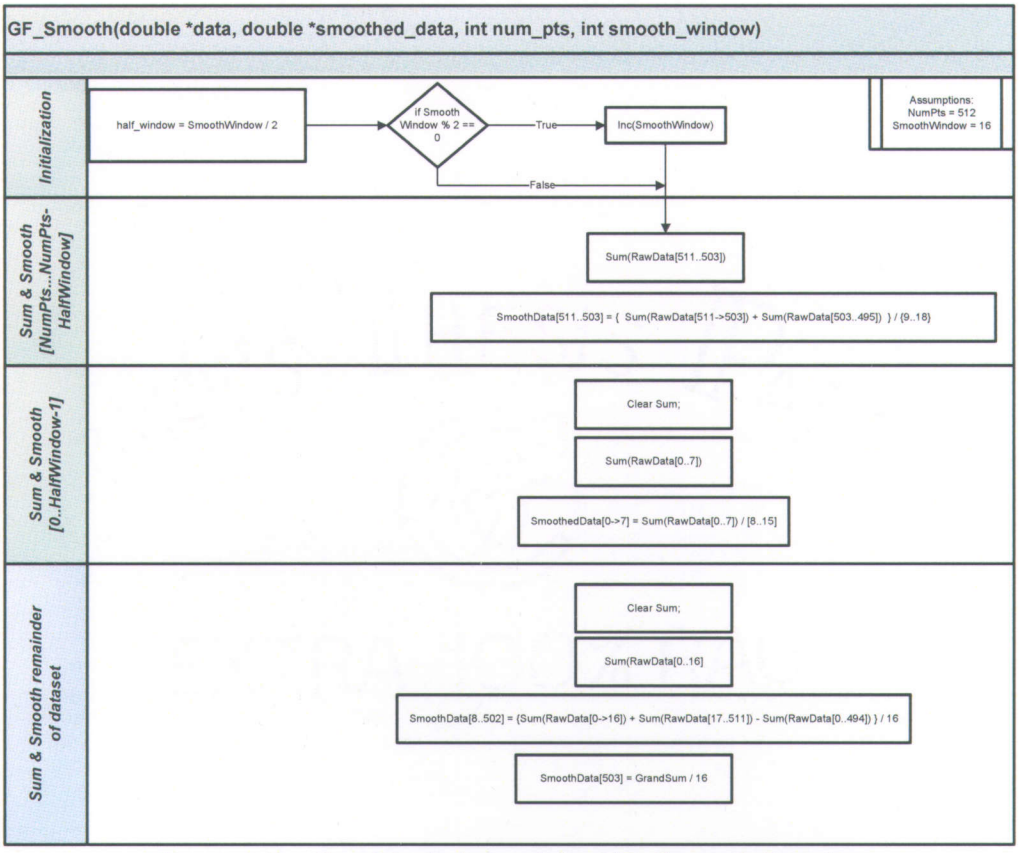


Figure 2.3: Graphical view of the GF_Smooth function.

While this smoothing mechanism is used for specific source data, similar results may be obtained by using an averaging filter.

2.2.1.2 Linear Periods

A provided dataset is broken down into linear periods by analyzing inflection points. Inflection points are detected by using the second derivative and are useful for identifying linear segments of a dataset. This approach is advantageous in an embedded Intelligent Sensor application because it involves less processing than other methods. In contrast, determining a least squares optimum fit by trying polynomial, exponential, and linear fits to match diverse datasets would be both time consuming to execute and require more code. The least squares approach also has drawbacks for datasets that change fast enough to become difficult to fit even with a continuous

high order polynomial. Thus, the solution is to find individual linear periods using techniques that are a compromise between generality and accuracy. Using the derivative response for finding inflection points is not without limitation. The derivative is affected by source signal quality (SNR) and step size (for discrete signals, such as those from an Intelligent Sensor, this is the sampling rate). To put this in perspective, consider a forcing function that consists of a single frequency sinusoid given as $\sin(2\pi ft)$. The derivative response $[2\pi f]\cos(2\pi ft)$ is 90° out of phase from the source and amplified by $2\pi f$. One would casually reason that a smaller time step not only allows larger source frequencies, but also increases performance in the presence of noise. This is not necessarily the case, and for the example in Fig. 2.4 shows the forcing function mentioned earlier combined with noise to an SNR of 35dB. These controls being constant, the first derivative is then computed for time steps of 10ms, 1ms, and 100s for frequencies from 1Hz to the Nyquist frequency for the respective time step.

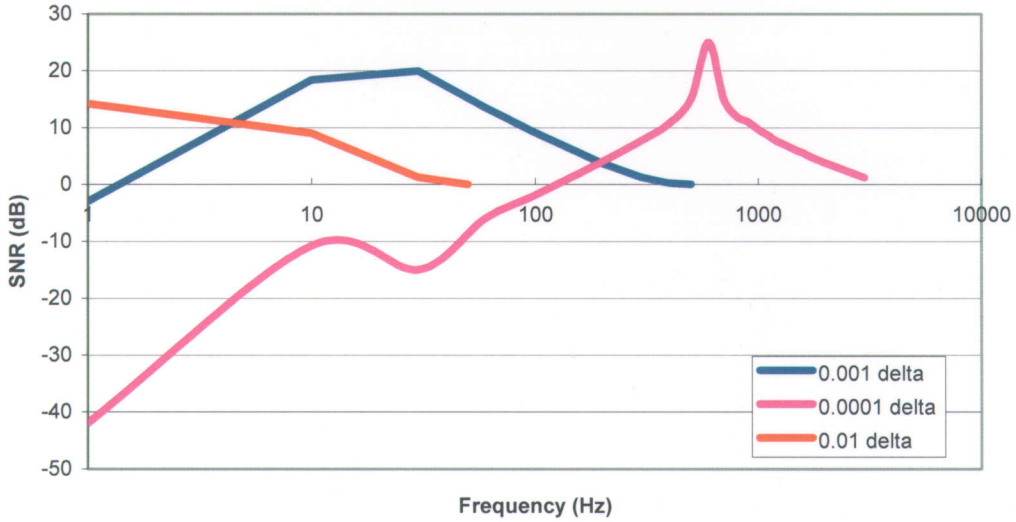


Figure 2.4: Derivative SNR vs. frequency performance for several time deltas.

The usable range is dependent on both the time step and frequency; in this example smaller deltas correspond to better higher frequency response, while low frequency response is

exceptionally poor. Larger time steps provide good low frequency response, but are fundamentally limited in maximum frequency. Due to these attributes, GRC determined inflection points and the associated linear periods using a different method.

GRC's segmentation of linear periods begins with a check to see whether the difference in time from the last to first data point is less than a user defined minimum period of linear behavior (MPLB). If the dataset is less than this value, but still contains at least two points, the only linear period in the dataset is recorded and execution is returned to the calling function. If the entire dataset is not linear as described in the prior step, then the next action is to remove linear trends from the data. This is done by taking the difference between the N^{th} *data* point and *the* 0^{th} data point in the dataset and dividing it by the corresponding difference in time to determine a slope constant, m, as shown in Eqn. 2.1.

$$m_{const} = \frac{x_N - x_0}{t_N - t_0} \tag{2.1}$$

After calculating the slope constant, the algorithm iterates through each data point as shown in Eqn. 2.2 to estimate what the value at the current index n would be if only the slope constant and initial point were used to define the linear segment. The value of η is the estimate of what the point at discrete time n should be, based on the computed slope constant.

$$\eta = (t_n - t_0)m_{const} + x_0 \tag{2.2}$$

The difference between the estimate and the real value is then compared to a register that maintains the value and location corresponding to the maximum difference found thus far. If the new difference is greater, the value and location of the data point replaces the contents of the register. After evaluating each data point in the set, the next step compares the largest recorded difference to see if it is 2.8 times greater than the user set linearity threshold. If this condition is true, then the process described above is performed on all points to the left and right of this data

point. This procedure continues recursively until the only differences encountered are less than the threshold. When this happens, the segment represents a contiguous linear period, and is recorded. A flowchart of this process is included in Appendix B.

2.2.1.3 Curve Fitting

Once the dataset has been smoothed and tagged with its linear sections, a curve fitting strategy is introduced. The underlying purpose for performing this curve fitting is to average the linear sections approximate the signal in the presence of noise. Since the dataset is already divided into linear sections, it is logical to accomplish the fit using a first order linear equation applied to each linear section. Eqn. 2.3 is the simple model of a linear equation that includes a slope, m, and intercept, b.

$$y_n = mx_n + b \tag{2.3}$$

While more complex approximations may be used, a sufficiently tight difference threshold in the previous section is sufficient for GRC's applications. The actual first order fit is performed by LU decomposition. This routine is borrowed from a numerical recipes text, and can be used to fit higher order curves, if need be. Least squares are used here, as the signal is already severely constrained, and unlike working with the entire dataset, should converge quickly.

2.2.1.4 Recording Signal Maximum and Minimum

The *MaxMin* function finds the maximum and minimum magnitudes in an array of data. It loops through all the data checking each point against the prior minimum and maximum values and updating the running maximum and minimum values as required. Note that this function uses the smoothed data produced from Section 2.2.1.1. *True* is returned if the operation completes without any problems. *False* is returned if there is a problem with the input parameters or the dataset.

2.2.1.5 Generic Features – Mean and Standard Deviation

The *GetStats* function computes a windowed standard deviation and mean for the dataset. This is the final preprocessing step before the GRC event detection routines are executed. Processing in this step is performed on the raw data with the results for each window stored in a standard deviation and average array. Deviations less than 0.005 are assigned to 0.005 to avoid a sparse condition. The event detection routines take as input parameters pointers to the respective array, while single value variables are passed by value unless it is expected that the value is to be changed by the routine.

2.2.1.6 Noisy PID Routine

Noise is defined for the purpose of this algorithm as the presence of a variance in a given raw signal that is larger than a given threshold, and that persists continuously for a user-provided minimum amount of time. Noise is classified as *fine* and *extreme*. Fine noise is a signal variance that is less than extreme, but larger than a nominal expected variance for the given signal. Extreme noise is assigned when the variance exceeds the Extreme Noise Threshold (ENT), which must be larger than the Fine Noise Threshold (FNT). This routine is very simple in that it performs a point-by-point analysis of the entire dataset for a given sensor channel and tests against the limits of ENT and FNT. If either is exceeded, the time index and type of noise is recorded. An ENT event supersedes an FNT event, and therefore only the ENT is reported. Once an event has been found, subsequent checks determine if the condition persists beyond the user-defined Minimum Event Time (MET), which is the smallest amount of time the event must continually exist before being declared an actual event. If a noise event is less than MET—i.e., the signal returns to a "quiet" behavior before MET lapses, the absence of noise is registered as the start of a quiet period. Analogous to the MET, the signal is now analyzed for a Minimum Quiet Time (MQT) threshold. If MQT is exceeded before ENT or FNT noise returns, a

suspected event is ignored and all counts are reset. If the noise event returns before MQT is exceeded, then the MET count resumes. The time index for this routine is in seconds. An example of noise detection is shown in Fig. 2.5.

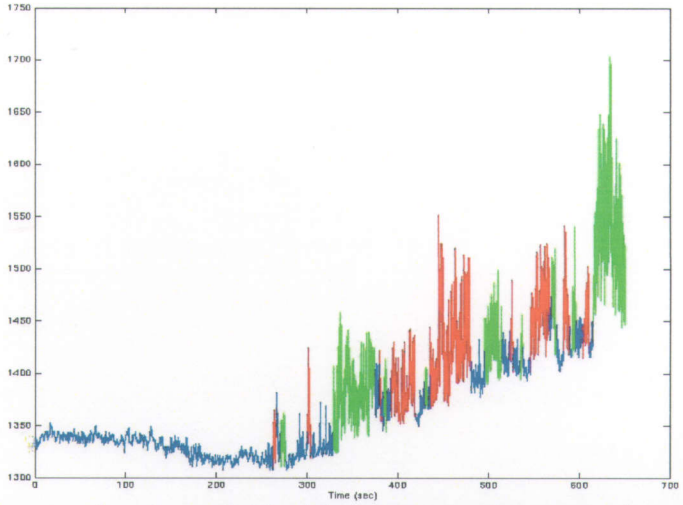


Figure 2.5: Noisy raw signal overlaid with extreme noise events in green and fine noise events in red obtained from GRC [52].

2.2.1.7 Is Flat Routine

Outputs of typical transducers normally exhibit some amount of deviation, even for otherwise nearly-constant measurands. Besides small fluctuations in the measurement, these variations can be due to transducer drift processes and signal conditioning errors. The intention of the *IsFlat* routine is to determine if a sensor signal is too quiet, indicating a potential fault in the transducer, or coupling to the measurand such as might occur if a TC became disbanded from an element. The routine operates on the premise that the raw signal is assumed to be flat and must be proven to be otherwise. To do this, the data set is broken into subsets of programmable length and subjected to three tests:

- Test of the slope of the mean values of the dataset.
- Test if the deviation of the signal exceeds a set maximum deviation.
- Test for how much the mean of the signal exceeds a set maximum.

This algorithm first requires the *GF_Fit(.)* routine to perform the least squares curve fit to obtain the slope for each linear period, as introduced in Section 2.2.1.2 and 2.2.1.3. The slope is used in conjunction with the total elapsed time of the dataset to determine the delta change over the dataset. If the delta exceeds a user defined threshold, the signal is considered *not flat.*

Testing the signal deviation is performed by finding the mean deviation over the entire data trace. Each data point in the trace is then subtracted from an anchor point (usually the first point in the dataset). If the difference is found to be greater than the mean deviation by a user defined number of instances from the mean deviation for a percentage of the signal exceeding the Maximum Sigma Jitter (MSJ), the signal is considered *not flat.*

The final method for evaluating the presence of sufficient signal energy is to take the mean of the incremental data set averages that have been computed using Section 2.2.1.5. From this the total data points exceeding a user-defined distance from that mean are determined. The percentage of data points exceeding this measure determines if the signal is *not flat.* The percentage is a user definable input parameter. A flowchart diagram is included in Appendix C to visually describe the operation of this algorithm.

2.2.1.8 FindSpike Routine

Another potentially useful piece of information bearing on the health of the Intelligent Sensor is the presence of impulsive behavior in the signal. *Spike* behavior is often indicative of bearing wear in mechanical systems, intermittent contacts, damaged thermocouples, or the presence of high-energy noise coupled into a system. The *FindSpike* routine uses a simple approach to operate on the raw dataset. If the difference between the value of the current data point and any of the three preceding data points exceeds a user-defined spike height parameter, a potential spike has been detected and the start time and position of the first data point is recorded. At this

point, there is a possibility that the flagged behavior may persist as another phenomenon, so it is too soon to make a determination. The routine continues to monitor the signal to determine whether it returns to values within the permitted spike height. If the duration of the spike behavior is less than the user set maximum spike width, a spike is reported. If the duration exceeds the maximum spike width, it is assumed to be a pulse and no report is generated. Defaults are 280ms for the maximum spike width, and 0.4 (normalized amplitude) for the spike height; these are highly dependent on the nature of the measurement system and the application. If the signal returns to a non-spike condition the algorithm continues as though no spike classification took place, repeating the process for the next set of four points, until the end of the data set is reached.

2.2.2 IEEE 1451 Smart Sensor Object Model

Thus far, the discussion of IEEE 1451 has been limited to examining the benefits that are afforded to both ISHM and non-ISHM systems. The rest of this chapter explores IEEE 1451 from an object oriented perspective in preparation to the implementation covered in Chapter 3. IEEE 1451 covers many facets of Smart Sensor/Actuator behavior; IEEE 1451.1 defines the overall object and communications structure for a Smart Sensor or Actuator. The physical standards 1451.2 thru 1451.6 introduced earlier are primarily geared toward implementations that have TIM(s) separate from the Network Capable Application Processor (NCAP), connected via an analog, digital, or mixed-mode TII and arranged according to Fig. 2.6.

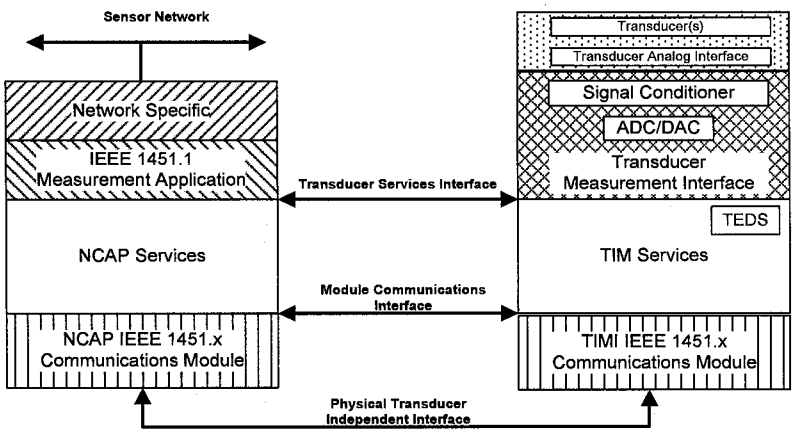


Figure 2.6: Compact view of 1451 application.

It is important to note the differences in the use of the term NCAP. NCAP, used as a noun, refers to the physical embodiment that has an IEEE 1451.1 measurement application running on it. In contrast, an NCAP object is the software instantiation of an object that inherits from the NCAP Block abstract class. Since 1451.1 is the complete object model that encompasses the entire Smart Sensor (including the physical standards), it deserves the most attention. IEEE 1451.1 is completely object oriented, and as such, provides a hierarchy of abstract classes which encompass the primary functionality of the Smart Sensor through inheritance down to the actual

base classes, from which the user can create interface classes. The top-level classes are shown in the UML diagram of Fig. 2.7. Only the methods mandated as required by the standard, as well as any optional methods necessary to our specific application, are shown for the sake of simplicity [53]. All classes inherit from the *Entity* and *Root* abstract classes. These two classes provide internal mechanisms and network visible functionality required to represent and perform an operation on a network object. The Class ID, Class Name, Object Tag, Object ID, and Object Dispatch Address are all object identification properties. The Object ID is a unique identifier that identifies an instance of an object. It must be unique with a system, and is generated by the local NCAP Block. The Class Name describes the purpose of the class. The Object Tag is designed to be used as a network-neutral binding point for client/server communications. It is intended to be assigned as part of system configuration, and is ideal for dynamic sensor networks. The Object ID is to be interpreted as an array of octets, of which there are two fields. The first field is a single octet, referenced by Table 2.2. The next field is the actual Object ID, which on an Ethernet network is the MAC address of the NCAP suffixed by two bytes that contain the Class ID.

Table 2.2: Valid values for the first byte of the Object ID.

Enumeration	Value	Definition
AIF_CLOSED	0	Closed System
AIF_ETHER_DCE	1	Ethernet using the DCE algoirthm per Object
AIF_ETHER_DCE_NCAP	2	Ethernet using the DCE algorithm per NCAP
AIF_ETHER_CUSTOM	3	Ethernet using the custom algorithm
AIF_FFBUS	4	FOUNDATION™ fieldbus
AIF_PROFIBUS	5	Profibus™
AIF_LON	6	LonTalk™
AIF_DNET	7	DEVICENET™
AIF_SDS	8	Smart Distributed System
AIF_CONTROLNET	9	CONTROLNET™
AIF_CANOPEN	10	CANopen™
AIF_1451_2	11	IEEE 1451.2
	12-255	Reserved for issuance by IEEE

The Object Dispatch Address is a network specific binding for client/server communications. In an Ethernet implementation it contains the IP address, Port, and Object ID. In this way, all of the necessary information to locate the sensor and a specific object on that sensor is available. The method *Perform* provides a way to execute an object.

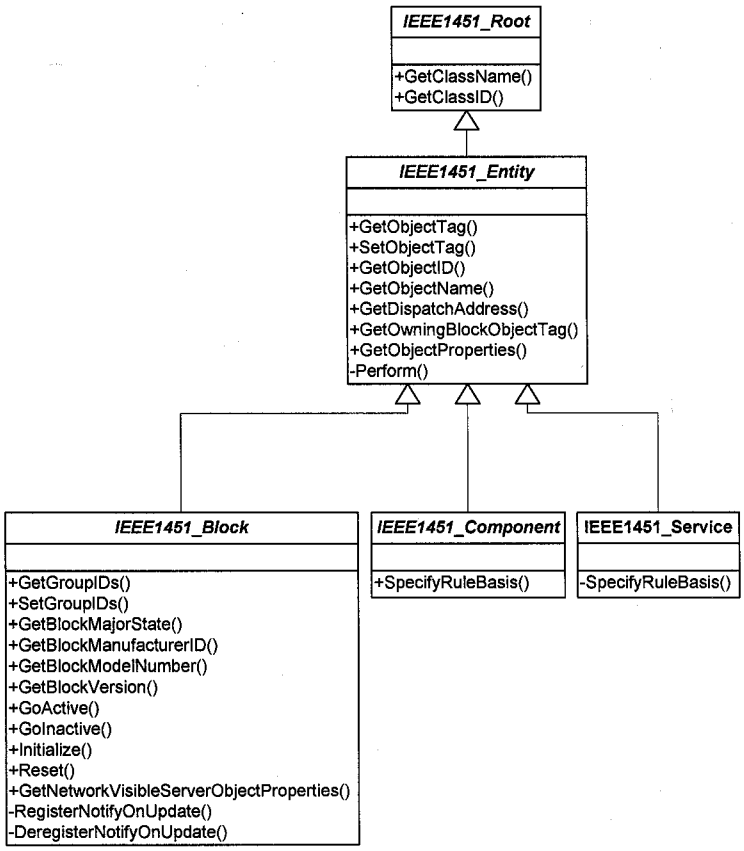


Figure 2.7: UML class diagram for the major 1451.1 Abstract Classes.

There are three primary abstract classes that constitute the core of 1451.1, and inherit from the root and entity abstract classes. They are the IEEE1451_*Block,* IEEE1451_*Component,* and IEEE1451_*Service* abstract classes. Each of these addresses specific areas of Smart Sensor functionality: objects, data model and representation, and resource interfacing. Each of these classes contains additional subclasses that are not shown in Fig. 2.7, but that will be explained in detail in the respective sections below.

2.2.2.1 IEEE 1451.1 Block Abstract Class

The Block abstract class is directly above the NCAP Block, Function Block, and BaseTransducer Block abstract classes in the class hierarchy. The UML class diagram for the Block abstract class is shown in Fig. 2.8. All of the network visible objects are contained within one of these three subclasses. In order for an object to be network visible, it must be owned by the NCAP Block since the NCAP is the central control object in the Smart Sensor. All network communications between Smart Sensor objects occur through an NCAP. Before a Smart Sensor can communicate on the sensor network, the NCAP must first be registered by executing *RespondToNCAPBlockAnnouncement* and setting the NCAP Block active using the inherited *GoActive* method. The other methods are effective on dynamically configured networks, where the number of sensors, their configuration, and capabilities are not known at runtime.

Figure 2.8: The NCAP Block abstract class.

The Function Block abstract class is the hook where application specific functionality may be linked onto the object model. This gives individual functions the ability to go active, be reset, perform actions, and respond to specific messages and remote objects similar to an NCAP Block. IEEE 1451 does not develop any required Function Blocks that are necessary for basic Smart Sensor operation. This is left entirely to application specific functionality, such as health assessment routines.

The last subclass under the Block abstract class is the Base Transducer Block abstract class. This class provides the highest level interface to the actual transducer by defining read, write, and status methods. Additional parameters are formulated in the Transducer Block abstract class, which inherits from the Base Transducer Block abstract class. It is here that bridging between 1451.1, and the respective physical standard (1451.2 thru P1451.6) takes place. It is possible to have transducers of different physical standards operating under a single NCAP, each with its own instantiated Transducer Block objects.

2.2.2.2 IEEE 1451.1 Component Abstract Class

The Component Abstract Class is responsible for providing data and time representation, as well as performing associated functions. In the standard, every data type corresponds to a TypeCode enumeration, which uniquely identifies each and every data type so that there is no ambiguity when interpreting data types. There are no publications or subscriptions associated with this class, as it does not have any network visible objects. The local operation *SpecifyRuleBasis* is used to specify rules that govern the objects behavior, such as startup or shutdown operations. The Parameter Class, which inherits from this class, models network visible variables and controls access. It has two network visible functions, *Read* and *Write*. These methods take an argument *data,* which may be a single number, time/value pairs, or array of numbers based on

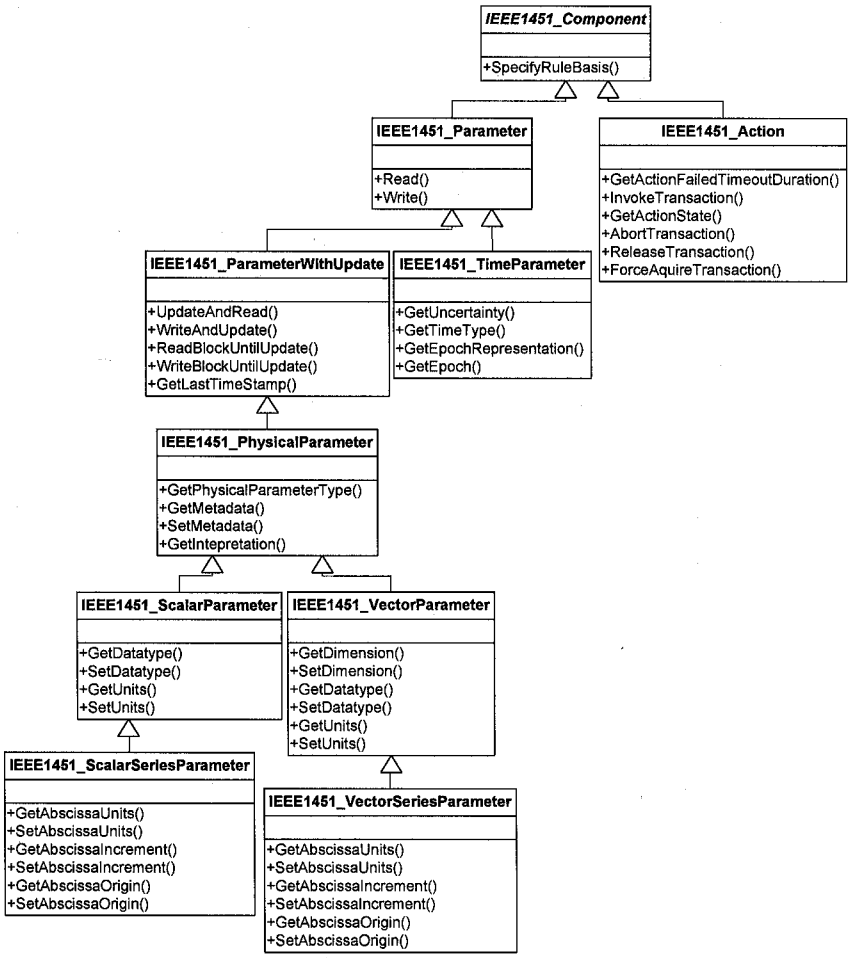


Figure 2.9: IEEE 1451 Component abstract class.

the parameter's class encoded as an Argument Array. An Argument Array is the standard represents data in the Smart Sensor and on the wire as a series of bytes consisting of paired TypeCodes and arguments. For example, an Argument Array to represent a 32 bit floating point value would be encoded into an Argument Array of five bytes; the first containing 0x1A to indicate FLOAT32_TC, and the remaining four containing the actual value, MSB first. In the case of an array of floating point values (perhaps in the case of an FFT spectrum), the first byte would consist of the TypeCode FLOAT32_ARRAY_TC, the next two would contain an integer count of the number of array elements (the length of each individual element is implied by the TypeCode) and the remaining bytes would contain each of the individual floating point values.

2.2.2.3 IEEE 1451.1 Service Abstract Class

The Service abstract class is the interface between the operating system, hardware resources, and the application software. Services that are provided through this abstract class include client/server communications, publication/subscription communications, and operating system control (mutual exclusion semaphores and flags, object locks). The class hierarchy for the Service Abstract Class is shown in Fig. 2.10.

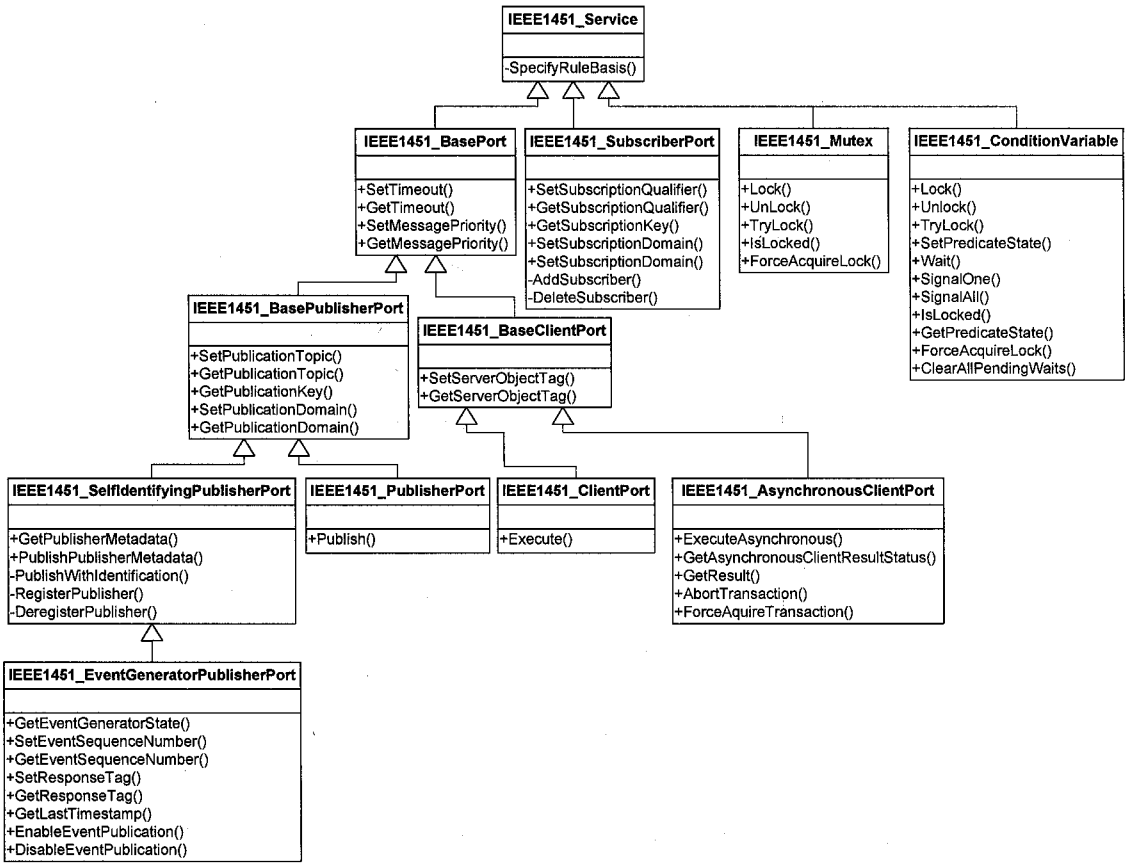


Figure 2.10: The IEEE 1451 Service Class.

2.2.2.4 IEEE 1451.4 and Transducer Electronic Datasheets

The Transducer Electronic Data Sheet (TEDS) is one of the primary components that make a smart device a *Smart Sensor*. The TEDS is a collection of machine readable attributes describing the manufacturer, capabilities, and conversion functions for the transducers connected to a Smart Sensor. Each transducer contains its own TEDS, which may be attached to the transducer, or

may be virtually associated using a *virtual TEDS.* There are several versions of TEDS, with 1451.2-1997 TEDS being the oldest and 1451.0-2005 TEDS the newest. This work focuses on 1451.4-2004 mixed mode TEDS due to relevance with thermocouples and because this is the IEEE 1451 version most commonly encountered in practice. IEEE 1451.4 defines a *mixed-mode* NCAP to TIM transducer independent interface (TII) that consists of a digital and analog interface. The digital interface is compliant with the Dallas One-Wire [54] specification. The nature of the analog signal lines are described within the TEDS. An example of this connectivity is shown in Fig. 2.11.

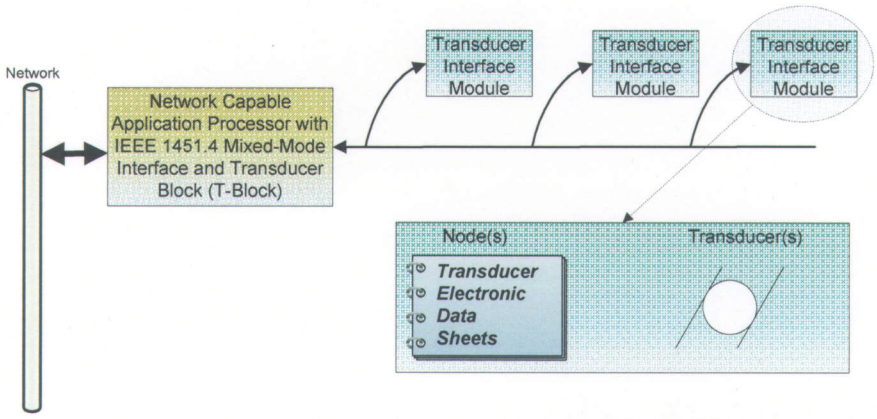


Figure 2.11: Role of 1451.4 and TEDS in a Smart Sensor solution.

In a Dot4 transducer block, there exists a unique registration number (URN) containing a family code (1 byte), serial number (6 bytes), and CRC (1 byte). This URN is assigned by IEEE and controls access to the TEDS memory. The computation of the CRC, 1-wire master device configuration, and 1-wire communications is further explored in Annex E and G of IEEE 1451.4 and "DS2430A 256-bit 1-wire EEPROM" [55]. The Smart Sensor implementation described here will not use Dallas 1-Wire or the MMI to achieve TEDS functionality, but the storage, retrieval, and parsing of TEDS and use of TEDS Templates will be according to the standard.

The Dot4 TEDS are stored in memory as a series of continuous bits. In the event of fragmentation between multiple nodes, each fragment is linked to adjacent nodes. TEDS memory starts with a 64-bit entity called Basic TEDS containing information about the transducer that is common to all transducers including:

Bits 0-13: Manufacturer ID (unsigned integer 17-16381)

14-29: Model Number (unsigned integer 0-32767)

30-34: Version Letter (1451 Chr5 'A'-'Z')

35-40: Version Number (unsigned integer 0-63)

41-64: Serial Number (unsigned long integer 0-16,777,215)

Basic TEDS is always the first 64 bits stored in the TEDS memory area. TEDS that follow Basic TEDS are referred to by the standard as either *IEEE, Manufacturer,* or *User* TEDS. TEDS of these types are either defined by IEEE in the standard, created by a specific transducer manufacturer, or are user defined, respectively. The major similarity is in how they are represented through the use of TEDS Templates and the Template Description Language (TDL). The Template is simply an overlay for the data, providing a direct and unambiguous interpretation of the data, as shown in Fig. 2.12.

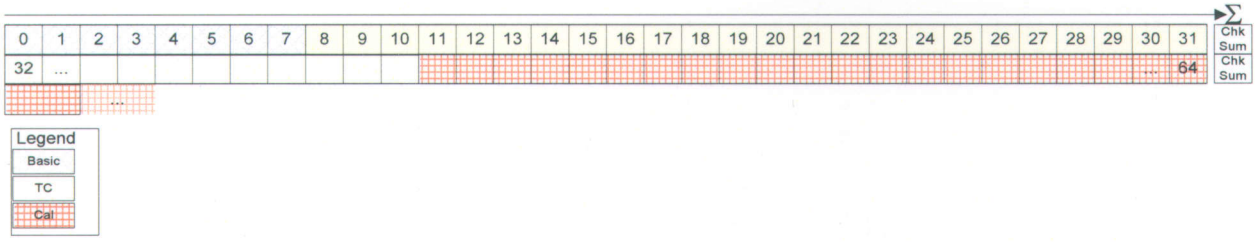


Figure 2.12: Typical TEDS storage and mapping for Basic, TC, and Calibration templates.

```
TEMPLATE 0,8,36,"Thermocouple"
  //The first 0 in the Template field indicates IEEE defined template, the 8 is the number of bits to read from
  //the sensor to get the template ID, the 36 is the decimal value of this template ID.
TDL_VERSION_NUMBER 2  //Version 2 refers to the final IEEE 1451.4 version 1.0 TDL specification
ABSTRACT IEEE 1451.4 Default Thermocouple Template
ABSTRACT For thermocouples and temperature sensors that provide a voltage output conforming to a standard
ABSTRACT thermocouple curve
SPACING

//Physical Base Units: (ratio, radian, steradian, meter, kg, sec, Ampere, kelvin, mole, candela, scaling, offset)
PHYSICAL_UNIT "°C",       (0, 0,0, 0,0, 0, 0,1,0,0,1,-273.15)  // Celsius is (kelvin - 273.15 K)
PHYSICAL_UNIT "V",        (0, 0,0, 2,1,-3,-1,0,0,0,1,0)        // Voltage: Volts equals m²okg/(sec³oA)
PHYSICAL_UNIT "sec",      (0, 0,0, 0,0, 1, 0,0,0,0,1,0)        // Time: base SI unit is seconds

ENUMERATE ElecSigTypeEnum,"Voltage Sensor","Current Sensor","Resistance Sensor","Bridge Sensor","LVDT
Sensor","Potentiometric Voltage Divider Sensor","Pulse Sensor","Voltage Actuator","Current Actuator","Pulse Actuator"
%ElecSigType, "Transducer Electrical Signal Type", ID, 0, ElecSigTypeEnum,"e","" = "Voltage Sensor"

%MinPhysVal, "Minimum Temperature", CAL, 11, ConRes, -273, 1, "0", "°C"
%MaxPhysVal, "Maximum Temperature", CAL, 11, ConRes, -273, 1, "0", "°C"
%MinElecVal, "Minimum Electrical Value", CAL, 7, ConRes, -25e-3, 1e-3, "0p", "V"
%MaxElecVal, "Maximum Electrical Value", CAL, 7, ConRes, -25e-3, 1e-3, "0p", "V"

ENUMERATE MapMethEnum, "Linear","Inverse m/(x+b)","Inverse (b+m/x)", "Inverse
1/(b+m/x)","Thermocouple","Thermistor","RTD","Bridge"
%MapMeth, "Mapping Method", ID, 0, MapMethEnum,"e","" = "Thermocouple"

ENUMERATE TCTypeEnum, "B","E","J","K","N","R","S","T","non-standard"
%TCType, "Thermocouple Type", ID, 4, TCTypeEnum, "e", ""

ENUMERATE CJSourceEnum, "CJC not provided by sensor","Sensor compensated for 0°C cold junction"
%CJSource, "Cold Junction Compensation", ID, 1, CJSourceEnum,"e",""

%SensorImped, "Output impedance of the sensor", ID, 12, ConRelRes, 1, 0.00155, "rp", "Ohm"

%RespTime, "Response Time", ID, 6, ConRelRes, 1E-6, 0.146, "rp", "sec"

%CalDate, "Calibration Date", CAL, 16, DATE,"d-mmm-yyyy",""
%CalInitials, "Calibration Initials", CAL, 15, CHR5,"s",""
%CalPeriod, "Calibration Period (Days)", CAL, 12, UNINT, "0", "days"

%MeasID, "Measurement location ID", USR, 11, UNINT,"0", ""

ENDTEMPLATE
```

Figure 2.13: Typical TEDS storage and mapping for Basic, TC, and Calibration

Templates are written in (TDL), which is a tagged markup language similar in form to the common Internet hypertext markup language (HTML). A sample TEDS Template is shown in Fig. 2.13 for thermocouples. Note the use of simple TDL syntax, punctuation, and keywords. Thus, with a TDL parser and the appropriate TEDS Template, a user is assured that the TEDS data is readable and properly interpreted. Multiple TEDS may be concatenated to fulfill the requirements of a particular transducer. In this case multiple templates need to be applied to the

TEDS. The keywords, or TDL commands, are grouped according to function, and are segmented into the categories of identification, control, and property. Character strings in TEDS are limited to A-Z, 0-9, ^,$,_,!,@,#, or &, provided that the first character is not a number. It appears that several manufacturers published provisional 1451.4 TEDS, referred to as "Revision 0.9 TEDS." An accelerometer template is shown in the 0.9 format for reference in Fig. 2.14.

```
Template Accelerometer, 2 gains, transfer function
abstract IEEE P1451.4 template version 0.9

crc8
"Manufacturer", "RO", 12, "Manufacturer"
"Model number", "RO", 16, "UNINT", "0000"
"Version letter", "RO", 5, "Chr5"
"Version number", "RO", 6, "UNINT", "0"
"Serial no.", "RO", 25, "UNINT", "0000000"
spacing

selector 2, 0, Selector of Template Descriptor = IEEE P1451.4
selector 8, 3, Template ID for IEEE 1451.P templates

"Calibration date", "RO", 16, "DATE"
"Supports multiplexer via no gain selected", "RO", 1, "UNINT", ""
"Default gain 00: no, 1:low, 2: high (PIO-A is lsb)", "RO", 2, "UNINT", ""
"Low gain Sensitivity @ Fref", "RO", 16, "ConRelRes", 100E-6, 0.0001, 0, "0.000p", "V/(m/s²)"
"High gain Sensitivity @ Fref", "RO", 16, "ConRelRes", 100E-6, 0.0001, 0, "0.000p", "V/(m/s²)"
"Fref", "RO", 8, "ConRelRes", 10.17501895022, 0.015, 0, "0p", "Hz"
"Low Gain F hp electrical", "RO", 12, "ConRelRes", 0.01, 0.001,0,"0.000p", "Hz"
"High Gain F hp electrical", "RO", 12, "ConRelRes", 0.01, 0.001,0,"0.000p", "Hz"


"Phase invertion (0: 0°, 1: 180°)", "RO", 1, "UNINT", "0"
"F lp electrical", "RO", 12, "ConRelRes", 100, 0.0015, 0, "0.000p", "Hz"
"F mounted resonance", "RO", 9, "ConRelRes", 100, 0.01, 0,"0.000p", "Hz"
"Mounted Q", "RO", 8, "ConRelRes", 0.3, 0.03, 0, "0.00p", ""
"Amplitude slope", "RO", 7, "ConRelRes", 0.852279961333371, 0.001, 0,"0.00p", ""
"Phase correction @ Fref.", "RO", 6, "ConRes", -3.2, 0.1, "0.0", ""°"
"Temperature coefficient", "RO", 9, "ConRelRes", 1E-6, 0.01, 0,"0.00p", "/°C"
"Sensitivity direction (x,y,z)", "RO", 2, "Direction"
spacing
"Meas. position ID", "RW", 9, "UNINT", ""
selector 2, 0, Selector of template descriptor = IEEE P1451.4
align 8
"User data (ascii)", "RW", 0, "ASCII"
EndTemplate
```

Figure 2.14: 1451.4 Rev 0.9 TEDS Template for an accelerometer.

There are three types of commands or keywords in a TEDS Template. Identification commands identify the template and associated TEDS. Control commands direct the flow through the template. Property commands provide the interpretation of the TEDS. A full list of commands is available by consulting IEEE 1451.4 [29]. The TEDS file system also provides for error

checking by performing a 32-bit checksum on the TEDS. The routine for computing the checksum involves the following steps:

1. Create a 4 byte integer and initialize to zero.
2. For each line, not including the Validation_KeyCode field, add the value of each octet to the 4 byte sum. Note that each octet is to be treated as an unsigned 8-bit integer. ASCII characters are to be treated no differently in checksum calculation.

TEDS is limited by the amount of memory available on the Smart Sensor. TEDS is designed to have very little memory overhead, while providing a robust and reliable mechanism for identification and configuration of TIMs.

This concludes the examination of IEEE 1451 and its key components. With a foundation developed detailing the essential components of the Intelligent Sensor (namely development of health assessment capability for noisy, flat, spiky signals, and an IEEE 1451 interface), focus is now directed toward Chapter 3, where these objectives are implemented.

CHAPTER 3: APPROACH

This chapter formulates the methods used to address and answer the challenges posed by the problem statement. To summarize the approach, there are a few major concepts that must be addressed in this chapter to realize an ISHM ready Intelligent Sensor:

- An investigation and selection of exemplar algorithms to be used in the Intelligent Sensor adapted from GRC algorithms for historical data.
- Memory management techniques to allow seamless buffering of large datasets accessible by multiple dependent and independent embedded algorithms.
- Providing a mechanism for easily adding, removing, and scheduling health algorithms through an API in C code, with Health Electronic Data Sheet (HEDS) self-descriptive capability.
- Developing methods for flashing the sensor operating firmware over Ethernet.
- Loading multiple sets of HEDS and switching between them at runtime for maximum health algorithm flexibility.
- An investigation of real-time operating systems for embedded processors.

3.1 Health Routine Adaptation for Real-time Sensor Applications

One of the most important aspects of this work is the development of embeddable health management in order to make a Smart Sensor into an Intelligent Sensor. Rather than using the Smart Sensor solely as a data source, it is modeled as an information source, providing valuable health feedback in addition to standard engineering measurements. This development has two implications; the first is that an Intelligent Sensor must be able to assess its own performance and

verify that it is operating within acceptable limits. Second, the Intelligent Sensor must assist with system level health assessment by extracting useful features from the data that it collects and communicating these with the ISHM environment. These features may either be examined in the Intelligent Sensor, or passed on to another processing tool for detailed analysis. An overview of the ISHM information and data flow is given in Fig. 3.1. Other analytical or numerical tools, such as MATLAB, may also be connected to the sensor network to assist in ISHM computations and analysis.

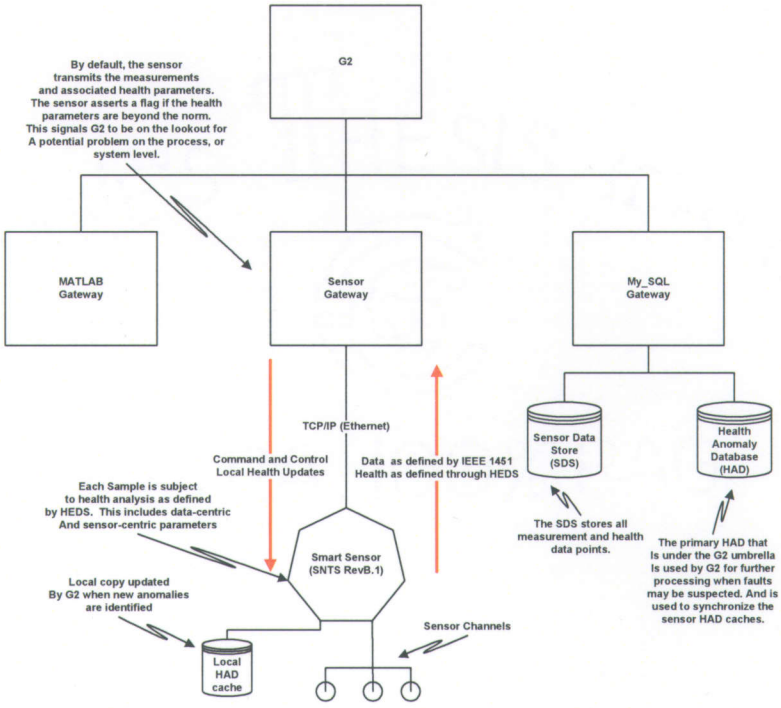


Figure 3.1: Component interactions in ISHM with health enabled Smart Sensors.

As previously addressed, this objective is incompatible with the previously described event detection algorithms of Chapter 2 because the GRC routines operate on a static buffer that contains a segment of raw measurement data. Recall that the first preprocessing operations such as filtering, fitting, and statistical computations are performed on the data sets. Next, event detection routines examine the data sets looking for signatures or patterns that hint of potentially

anomalous behaviors. Finally, a report is provided that details any events, the duration, and time of occurrence. Also recall that in a real-time environment preprocessing and analysis need to occur on an emerging data set. Similarly, the reporting mechanism needs to be activated when events occur, as a summary at the end of the run. In the embedded real-time environment, there is also the demand to consume a minimal amount of computational and memory resources. A major focus of this chapter is developing alternatives to the GRC routines of Chapter 2 that meet the above criteria. Furthermore, algorithms that have specific performance capabilities are not developed in a vacuum but rather are based on experience with historical data and operational experience. Over time, such algorithms are expected to be updated as more experience is gained. This understanding helps leverage the event detection expertise of GRC with advances in real-time signal processing, while also demonstrating the versatility of the Intelligent Sensor.

As a result, there are two preprocessing routines, four feature extraction routines, and three event detection routines provided as exemplar health assessment routines for evaluating sensor health in real-time on the Intelligent Sensor. The preprocessing routines are addressed first, which include the implementation of an 8 point sliding mean digital filter for smoothing the data used for computing derivatives and a high pass digital filter that only passes signal behavior indicative of noise. The feature extraction routines consist of a windowed mean, windowed standard deviation, 1^{st} and 2^{nd} derivatives, and discrete Fourier transform. Lastly, the event detection routines are used to detect noise, spikes, and flat-line behavior using the raw and preprocessed data streams.

3.1.1 Smoothing

While curve fitting may be an important technique because many process models follow relatively well-behaved polynomial or exponential responses, GRC's primary motivation behind

curve fitting was to approximate the true signal in the presence of noise. GRC's method was to fit to a first-order polynomial and in essence limit the bandwidth of the signal, and start new first line approximations whenever the signal's concavity changes. While at first glance this appears sound, the presence of noise (see Fig. 3.2) results in many unexpected concavity changes that ends up causing the curve fit to become more like a curve trace. Nevertheless, there are many effective curve fitting techniques, such as fitting to a power series, exponential series, logarithmic series, least squares approximation (loess, or non-linear Savitzky-Golay techniques), linear predictive adaptive filters, and radial basis functions. While likely one of these methods will be beneficial to ISHM and be implemented on the Intelligent Sensor in the future, the task at this point is to address the underlying objective of GRC: smoothing of signal data. Recall the entire purpose for using GRC's curve fitting in the first place is to constrain the raw signal to improve the performance of numerical analysis tools (in particular, recall the derivative discussion in Section 2.2.1.2). The alternative strategy used here takes into account the characteristically white noise present at the analog end of the Intelligent Sensor, and implements a moving average digital filter to smooth the signal.

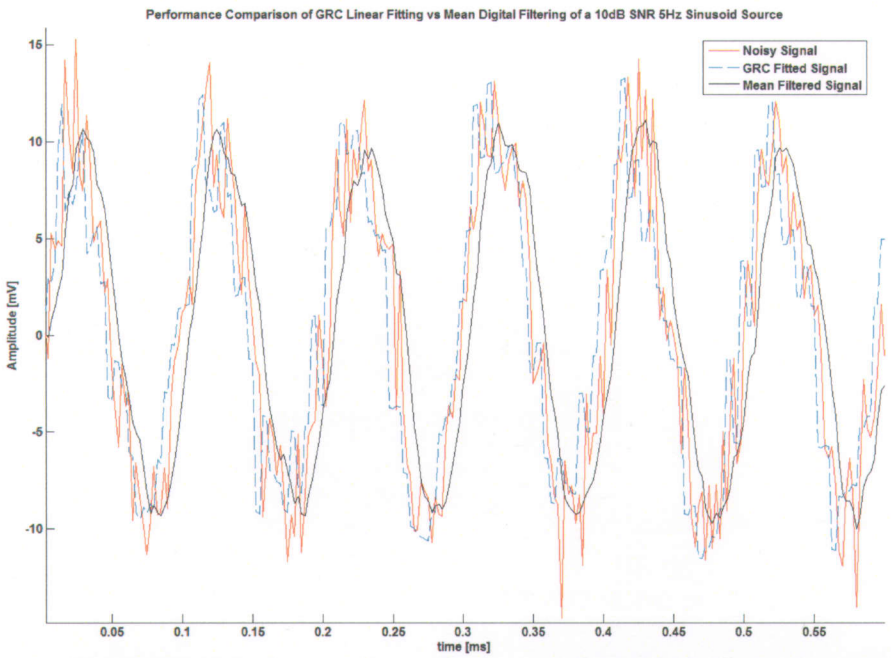


Figure 3.2: 10dB SNR sensor data against GRC curve fit and moving average digital filter.

The digital filter is implemented as a difference equation as depicted in Eqn. 3.1, with eight numerator coefficients and a denominator coefficient of unity.

$$y(n) = b_0 x(n) + b_1 x(n-1) + \cdots + b_M x(n-M) - a_1 y(n-1) - \cdots - a_N y(n-N) \tag{3.1}$$

Because of its effectiveness, this smoothing routine is used before the derivative routines discussed below, however the other routines in this thesis use the raw data directly.

3.1.2 Highpass Digital Filter

Part of the flexibility of the Intelligent Sensor is in its ability to contain algorithms of great diversity; from statistical functions to event detection routines to digital filters, and possibly even neural networks. As proof of concept, a typical high pass IIR Butterworth digital filter was chosen. The choice of high pass filter (HPF) is to better support noise detection presented in a later section. However, the actual filter response is not critical to the issue of embedding a filter structure in the Intelligent Sensor. The bandwidth requirements of a thermocouple are relatively low since they are typically attached (bonded) to thermal masses that are significantly greater

than the thermocouple. The HPF filter was designed for a 50Hz sampling rate, a stop band with minimum attenuation of -20dB from DC to 10Hz and a pass band of 15 Hz and higher—i.e., there should be -3dB of attenuation at 15Hz. These design choices for filter parameters were chosen based on observed dominant temperature transmitter signals of the MTTP data, which are found to generally be 5 Hz or less. The MTTP is discussed in the results of Chapter 4, with a piping and instrumentation diagram (PID) in Appendix A. The realized filter is of fourth order to accommodate the desired transition band, with the specifications shown in Fig. 3.3.

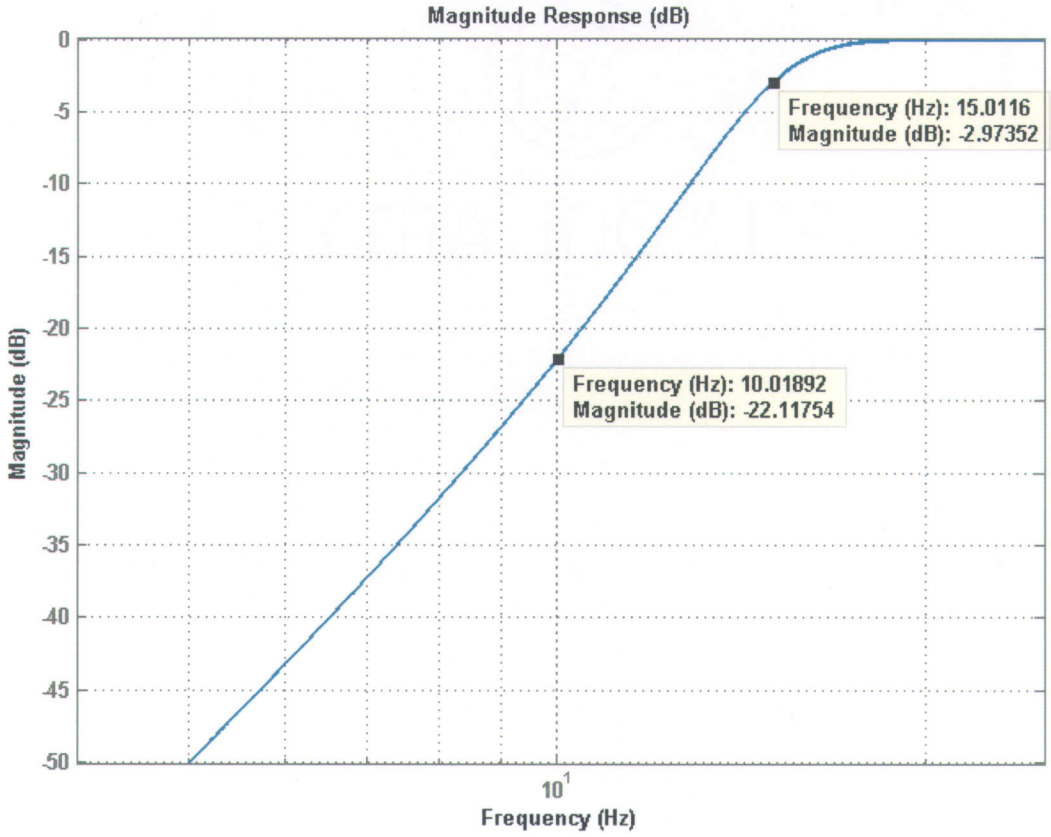


Figure 3.3: Magnitude Response for a 4th order HPF with Butterworth transition.

This filter is implemented using a difference equation, shown in Eqn. 3.1. The respective coefficients are obtained from the design in Fig. 3.3. As a health assessment feature of the Intelligent Sensor, it is supplied with a sliding window of four data values, and operates such that

when the window is not full, each new point results in a multiply/accumulate operation for the feedforward and feedback sections of the filter. When the window fills, the first filtered output is recorded for later observation by the event detection routines to follow. Now full, the window will slide for each new data point, resulting in recompilation of all of the coefficients and the evaluation of each subsequent filtered data point. The coefficients of the filter are provided for reference in Equation 3.2.

$$\begin{aligned} b0 &= 0.0466491 & a0 &= 1.0000000 \\ b1 &= -0.1865962 & a1 &= 0.7806856 \\ b2 &= 0.2798944 & a2 &= 0.6791779 \\ b3 &= -0.1865962 & a3 &= 0.1822788 \\ b4 &= 0.0466491 & a4 &= 0.0300717 \end{aligned} \tag{3.2}$$

The performance of the filter has been examined in both MATLAB and the Intelligent Sensor. The filter meets the design requirements in both environments, providing a minimum of -20dB at 10Hz and below, -3dB at 15Hz, and full 0dB at 20Hz and above as shown in the time domain graphs of Fig. 3.4.

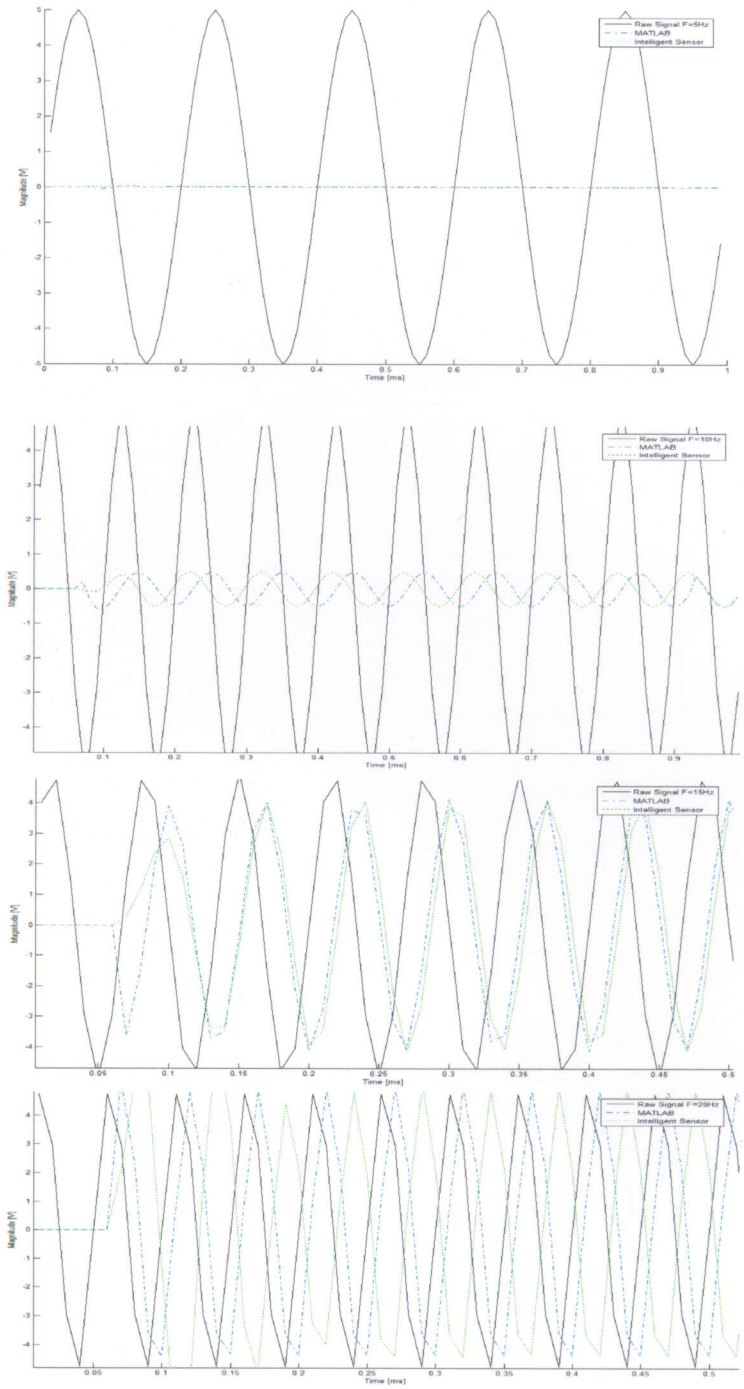


Figure 3.4: HPF performance for MATLAB and Intelligent Sensor implementations

While the effort in processing a digital filter is a function of the filter taps, it is still nothing more than addition and multiplication operations. The Intelligent Sensor development platform is based on a simple, fixed-point microcontroller that lacks a specific multiply-accumulate (MAC)

instruction. However, a MAC instruction is universally found in special-purpose DSP processors. Kennedy Space Center (KSC) has undertaken the development of a Smart Sensor as part of their Advanced Technology Lab efforts [56]. The KSC Smart Sensor includes both a general-purpose microcontroller as well as a DSP processor made by Texas Instruments.

3.1.3 Sliding Window Mean

A useful function in signal analysis is the arithmetic average shown in Eqn. 3.2. Computation of the mean requires N additions, where N is the number of values the average is to be computed over, followed by a single division. While not computationally complex, on the Rabbit processor it takes 283 ticks per addition and 796 ticks for a single division operation.

$$\bar{x} = \frac{1}{N}\sum_{1}^{N} x \tag{3.3}$$

If a particular algorithm required a mean computed over 256 data points, the total computation for Eqn. 3.3 equates to 2.0ms on the Intelligent Sensor Rabbit CPU running at 44MHz. Since the target maximum sampling interval is 4ms, computing the mean of a modest sliding window in this manner appears prohibitive.

The solution to this problem is to use an iterative computation of the mean. This approach eliminates the need to perform all calculations as each new value is presented, but instead incrementally adds the contribution of each new point and deducts the contribution of the oldest points from an already accumulated running sum. The mean is then computed by dividing the desired incremental sum by the size of the window at that time. Thus, the mean is broken into two parts: The computation of the sum, and the division of that sum by the size of the window. This is presented in Equation 3.4a, which computes the intermediate average for the new data point and adds that to the accumulated average where the number of data points

accumulated is less than the desired window size. The only complication to the iterative method is the need for a piecewise equation to handle the conditions of a partial window and a full window. Equation 3.4b addresses this issue, removing the contribution of the oldest data point to maintain a constant window size. Equation 3.4b should be used only once the window is full.

$$A_{n+1} = A_n + S_n \tag{3.4a}$$

$$A_{n+1} = A_n + S_n - S_{n-N} \tag{3.4b}$$

For each execution of Equation 3.4a, two additions are required, and three are required for Equation 3.4b. The total CPU usage required is 12.9μs and 19.3μs, respectively.

$$\bar{x}_n = \frac{A_n}{N} \tag{3.5}$$

The second part of the computation is the operation shown in Equation 3.5. This single operation takes 18.1μs. The total CPU usage to compute a mean using Equation 3.4 and 3.5 comes to 31.0μs for a partial window and 37.4μs for a full window. The most costly component is the division operation. If the window size is a power of two, the division instructions may be replaced with simple bit shifting, drastically reducing the CPU requirements to that for Equation 3.4. In the case that specific algorithms require a divisor that is not a power of 2 or if the window is not full, a multiplication by the (constant) reciprocal of the window size would be an acceptable shortcut. Since the multiplication instruction is 2.25 times faster than division, the iterative solution of Eqn. 3.5 optimized this way and factoring in the incremental processing of Eqn. 3.4a would take a total of 17.5μs for a partial window and 23.9μs for a full window. The computational complexity for the real-time mean presented here is O(n).

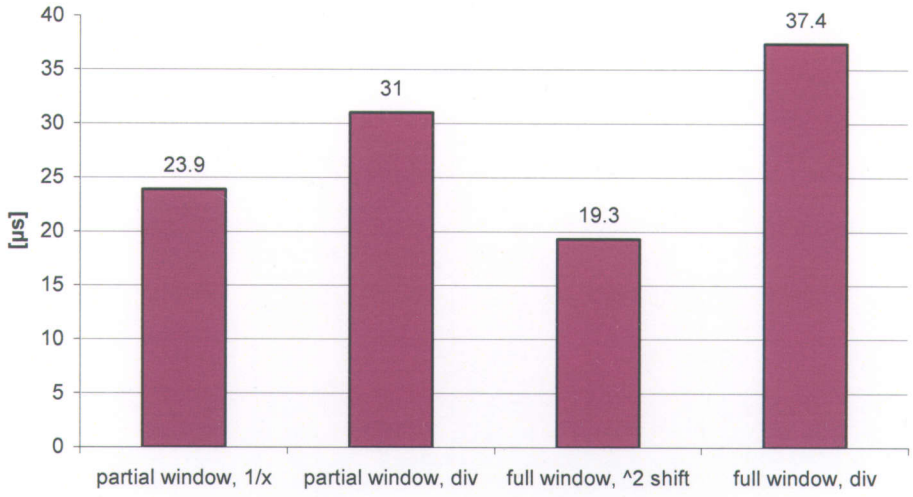


Figure 3.5: Real-time mean performance.

3.1.4 Sliding Window Standard Deviation

The standard deviation is a useful tool in signal processing for understanding the distribution of a dataset. Recall that the standard deviation is defined as in Eqn. 3.6.

$$\sigma = \sqrt{\frac{\sum_{0}^{N}(x_i - \bar{x})^2}{n-1}} \tag{3.6}$$

For the summation, each iteration requires a subtraction operation, squaring operation, and then summing to provide the accumulated sum. For a sample size of 256 points, 846,848 clock ticks are required to perform the total additions. The denominator is a constant expression, so the third operation is division by a constant, along with the final square root to obtain the result. The division and square root collectively require 1740 CPU ticks. The total computation time for a 256 point standard deviation is 848,588 ticks. That corresponds to 23.1ms with the CPU running at 44MHz.

An acceptable iterative approach to this problem is given in Eqn. 3.7 and 3.8. An incremental sum is computed for each sensor data point. The difference between Eqn. 3.6 and

Eqn. 3.7 deducts the oldest sample to maintain a constant size window when the window is full. The incremental sum in the partial window case requires two additions plus the square of the data point at 2,743 ticks for a total of 3309 ticks or 75.2μs. In the case of the full window, the number of ticks increase to 3,592, which yields a time of 81.6 μs.

$$b_{n+1} = b_n + S_n^2 \tag{3.7a}$$

$$\sigma_n = \sqrt{\frac{b_n}{N} - \bar{x}_n^2} \tag{3.7b}$$

$$b_{n+1} = b_n + S_n^2 - b_{n-N} \tag{3.8}$$

To minimize the effect of the squaring operation, a multiplication of the datapoint by itself would provide the same result without a loss of accuracy or precision. Doing this saves 54.3μs for both a partial window and a full window. The computational complexity for the incremental sum is O(n).

With the incremental sum computed, the standard deviation at the current data point is obtained using Eqn. 3.7b. Using the result of Eqn. 3.4 and 3.5, the mean is already computed, leaving window length as the only variable. Since this part contains a square root, division, and squaring operation, the computational costs are a bit higher. The division requires 796 ticks, the square requires 2,743 ticks, the subtraction requires a mere 283, and the square root requires 944 ticks. The total computation load for this part of the process comes to 4766 ticks or 108.3μs; however, several simple optimizations may be performed without sacrificing accuracy. The division is replaced by multiplication of the reciprocal (which is essentially a constant), and the squaring of the mean is replaced with a multiplication of the mean by itself. These optimizations yield a performance gain of 44.0μs. Thus, the total computational load for the standard deviation

is 3,138 ticks (71.3µs) for a full window and 2855 ticks (64.9µs) for a partial window. Refer to Fig. 3.6 for a graph summarizing the performance specifications.

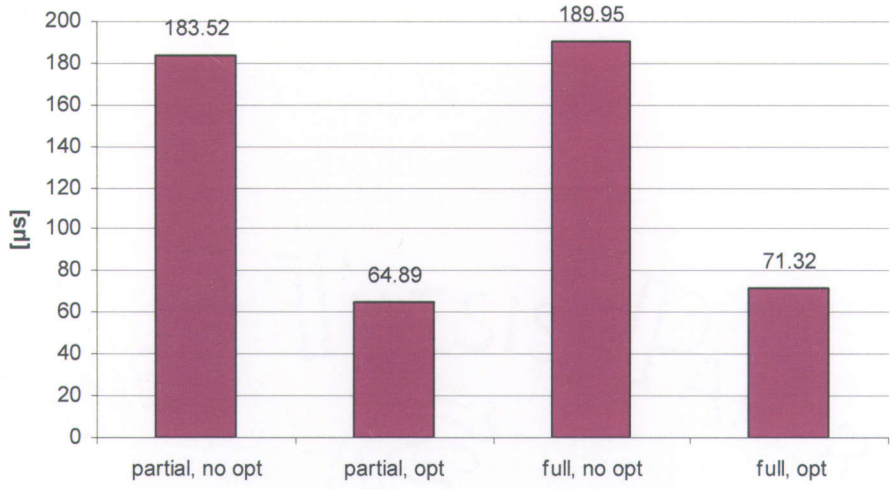


Figure 3.6: Real-time standard deviation performance.

3.1.5 First and Second Order Derivatives

The first derivative can provide valuable insight to the time dependent behavior of a data set. Rate of change is an important descriptor when compared to transducer time constants, expected measurand behavior, and prediction of signal trends. Numerical methods for performing differentiation in this section are based on the central difference formula given in Equation 3.8.

$$\frac{df_{n-1}}{dt} = \frac{1}{2h}[f_n - f_{n-2}] \tag{3.9}$$

In a real-time application the dataset is causal, so it is necessary to use a backward variation of the central difference. Eqn. 3.9 is linear and time invariant, and therefore can be rewritten this way, although now there is a delay introduced of two samples. The error of the central difference is proportional to the square of the period between measurements *(h)*, whereas the error for the forward and backward difference is proportional to *h*. The *h* is the interval between

the discrete values represented by *f*. In theory, the goal is to use an infinitely small interval to minimize approximation error. However, since we are working with a discrete data set, the interval is the sample rate of the sensor. Remember that in Section 2.2.1.2 the optimum step size (sampling period) may not necessarily be the smallest; the ideal sampling period depends on required derivative quality, the spectrum of interest, and source SNR. The ratio of source SNR to derivative SNR is shown in Fig. 3.7 for 10, 50, and 100Hz. The figure reinforces earlier arguments and shows that higher frequencies (i.e. larger sample period) signals require less input SNR at the lower ends of the spectrum, though level off at a lower maximum. In terms of loading, the central difference requires 14.5µs per computation, making its impact almost inconsequential to overall Intelligent Sensor operations.

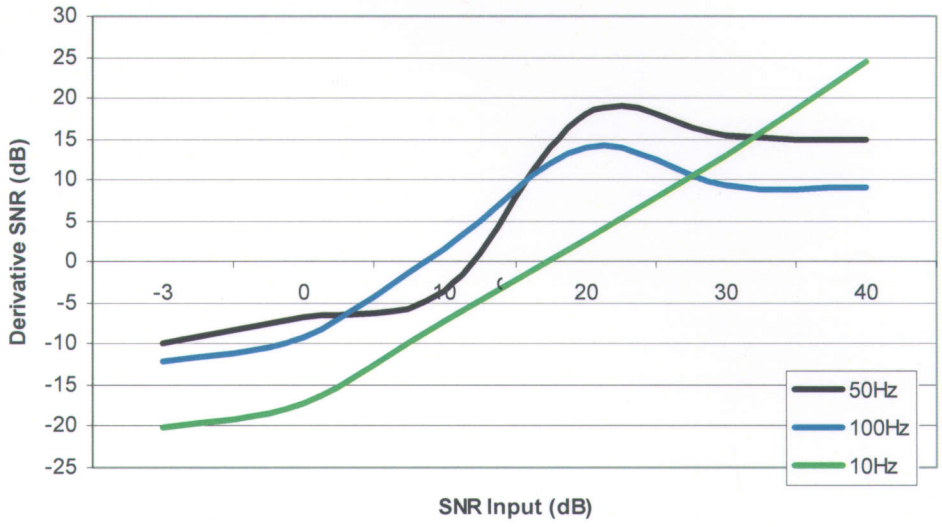


Figure 3.7: Source SNR vs 1st Derivative SNR for 10, 50 and 100Hz sinusoids.

The discussion thus far concerns only the first derivative, although the second (and higher-order) derivatives can also be important. For example, opening and closing characteristics associated with variable position valves may require that the 2nd derivative of valve position be determined.

Building on the foundation for numerically computing the first derivative, a second derivative may be computed according to Eqn. 3.10.

$$\frac{d^2 f_{n-1}}{dt^2} = \frac{1}{2h^2}[f_{n-2} - 2f_{n-1} + f_n] \tag{3.10}$$

Here there is dependency on the current and the previous two data values. The second derivative has a net performance of 37.0μs. Higher order derivatives follow similar form to those described here and may be obtained by using the central difference formula.

3.1.6 Discrete Fourier Transform

The Discrete Fourier Transform (DFT) is also a valuable health feature that is a key part of the health enabled Intelligent Sensor implementation. The DFT is extremely useful for signal processing, spectral analysis, and filtering applications, but it is also CPU intensive. The DFT requires significant computational capability compared to the other routines already described, on the order of $O(n^2)$. Using optimized FFT libraries that are available for the Rabbit, it is possible to perform a 256-point FFT in about 8.5ms, as the complexity reduces to $O(n\log[n])$. The only requirements are that the waveform to be analyzed contains an integer power of two elements, and the signal values are interpreted as 16-bit signed integers. This loss of precision, as shown in Chapter 4, has a minimal impact in most cases. While there are other routines [57, 58] that may be faster, there are also more adverse effects on precision. The natively supported windowing function is a Hanning window, although other windowing techniques may be implemented and utilized as needed. The number of descriptors in the FFT must be chosen based on the desired frequency resolution and signal bandwidth. Generally, the sampling rate divided by the desired frequency step in Hz will yield an acceptable number of FFT descriptors. The

catch here is the size of the frequency step depends on the nature of the features one wishes to detect. Since the default application utilizing thermocouples has a characteristically large time constant, frequency resolution on the order of several Hz is sufficient. For the worst case estimate using the established maximum sampling rate of 250Hz (and therefore maximum band limited signal frequency of 125Hz), 126 FFT descriptors yields a very high resolution of about 1 Hz per descriptor. Using 126 descriptors for the forward FFT operation, unless the spectrum is to be transformed back to the time domain (theoretically, there is no need to do this unless the spectrum is to be convolved with another function, such as in the context of an FIR filtering operation), HEDS can specify that only a representative 64 descriptors be returned to ISHM, conserving network bandwidth.

3.1.7 Sensor Noise Events

Noise events are triggered when the observed noise exceeds predefined thresholds. Example sources of noise include the transducer, analog instrumentation (wiring/amplifiers/filters), environmental disturbances, and conversion/code noise introduced by the ADC. Separating these contributing factors from deviations in the measurand is very difficult. One simple measure of noise content is to determine the amount of energy in non-signal bandwidths. That is, if the expected bandwidth of the signal is 0-5 Hz, then energy in higher frequency terms is considered noise. This is especially easy for low to moderate bandwidth signals, such as those typically associated with valves, thermocouples, and temperature transmitters. The chosen method to evaluate signal energy is based on the HPF developed in Section 3.1.6. This routine calculates the magnitude of the filter output vs the raw output in decibels (dB) and compares it to a settable threshold measured in decibels (dB). When the output of the HPF exceeds the

threshold, a noise event is signaled by the Intelligent Sensor. It is understood that the noise event continues up until which time the Intelligent Sensor provides notification that the event has ended. While up to now the discussion has centered on the *energy* of the filter output, energy is defined in Eqn. 3.11 as the area of the signal squared. The problem with this is that in the case a signal does not decay (i.e. a cosine), the integration across all time would suggest infinite energy.

$$E = \int_{-\infty}^{\infty} [f(x)]^2 \, dt \tag{3.11}$$

Since this is a discrete signal with finite time boundaries, an ideal solution is to find the signal power, which is the time average of energy as shown in Eqn. 3.12. This yields the mean-square value of the signal, which becomes the root mean square (RMS) by taking the square root of Eqn. 3.12.

$$P = \int_{T/2}^{T/2} [f(x)]^2 \, dt \tag{3.12}$$

On the Intelligent Sensor we estimate the mean square power by accumulating the squares of each discrete value for a window of 25 points. Dividing the accumulator by the window size (25) yields the mean square of the dataset. The ratio of filtered signal power over the raw signal power (measured in dB) is then compared to a HEDS-settable threshold, indicating the presence or absence of noise. It is important to realize that the size of the dataset of which the RMS is computed affects the algorithm's sensitivity, and is analogous to the Minimum Event Time (MET) and Minimum Quiet Time (MQT) parameters for the GRC NoisyPID algorithm. An algorithmic flowchart is provided for the noise detection algorithm in Appendix D.

3.1.8 Sensor Spike and Flatline Events

The spike detection routine described in Chapter 2 provides a brute-force method for detecting spikes. In that routine, a spike is identified for events lasting for less than a set cutoff time and that are greater than a preset magnitude. Shortcomings of this routine include its inability to make the positive identification until after the event ends and limited generalization of input parameters. Similarly, due to the impulse nature of a spike (i.e. infinitely large amplitude over a unit area), the spike doesn't contain enough power to show up as high frequency noise in a spectrum analysis or digital filter application. Therefore, for the Intelligent Sensor, a method that is immune to these limits is to evaluate the signal *crest factor*. Crest factor (CF) is defined in Eqn. 3.13, and is simply the peak signal value divided by the root mean square (RMS) signal value evaluated over a finite time interval.

$$CF = \frac{Peak}{RMS} \tag{3.13}$$

The significance of this measurement can be understood by considering the crest factor of some common signals. The relationship of the peak, RMS, and mean values are shown for one half period of a sine in Fig. 3.8.

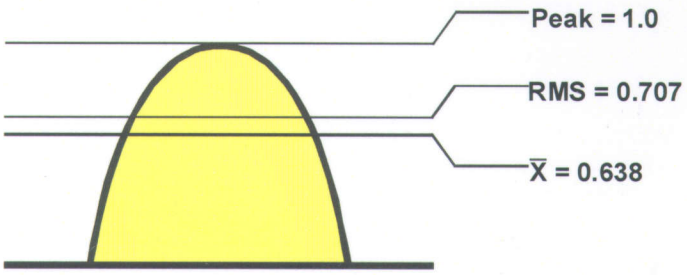


Figure 3.8: 1Vpp sinusoid example with crest factor of 1.414.

The crest factor for a DC signal is 1.0 as the peak and RMS are both the same. For a pure sinusoid, CF is 1.414, as shown in Fig. 3.8. On the other hand, the crest factor for an impulse with a peak of 1.414V as shown in Fig. 3.9 (with magnitude 41% higher than that of the sinusoid) is 6.42. While the short duration and limited energy of the impulse in Fig. 3.9 result in

low average and RMS readings, paired with the larger peak value (which is typically large for an impulse), the CF becomes considerably larger.

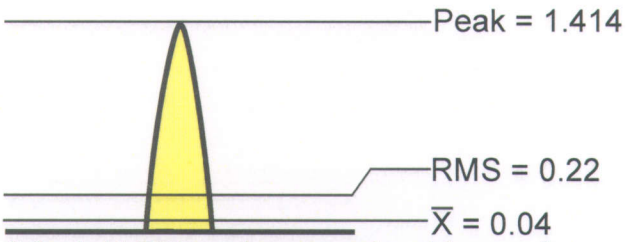


Figure 3.9: 1.414V impulse example with crest factor of 6.42.

Crest factor can be used to check for flatness, or the presence of a nearly-DC signal value. Due to the limited resolution of CF in low peak conditions (for example, a 1Vpp sine has a CF of 1.414, but a DC 1V signal has a CF of 1.000, only a delta of 0.414), the difference between peak and mean signal values are also compared to the flatness threshold to determine the existence of a flat condition. To this end, a crest factor entry for the health electronic data sheet (HEDS) provides a threshold for a maximum and minimal allowable limit to address the detection of spike behavior, and a percent difference of the peak and mean minimum threshold is also provided for flat signal behavior, as shown in the algorithmic flowchart in Appendix E.

3.2 Sensor Real-time Operating System and APIs

Health enabled Intelligent Sensors in ISHM require real-time processing capability to meet deadlines for measurement acquisition, measurement presentation, health processing, and health information presentation. The nature of the deadline qualifies it as *hard* or *soft* [59]. For a hard deadline, on-time completion of the associated activity results in the system receiving full benefit from that activity. On the other hand, completion after the deadline results in no benefit from that activity. In the context of Intelligent Sensors, measurement acquisition and presentation are hard deadlines, as without timely measurements the integrity of onboard health assessment routines and other ISHM processes depending on sensor data are jeopardized. The amount of risk introduced by missed deadlines varies; best case the missed deadline translates to a delay in processing while waiting for the late measurement, however if the activity is so late that it is skipped to avoid conflict with the next deadline, a missing feature has now been introduced.

Soft deadlines are those events that provide optimum benefit to the system when completed on-time, but may be completed marginally earlier or later at the cost of somewhat decreased benefit. Completion outside of an allowable margin results in zero benefit. The execution of health assessment algorithms and transmission of health information is considered a soft deadline because while still a critical part of Intelligent Sensor doctrine, the nature of health assessment (not to mention the need to buffer data, run intensive algorithms, etc) requires more scheduling flexibility.

Taking this and applying it to the Intelligent Sensor with a 50Hz sampling frequency, the hard real-time requirement for data acquisition and transmission is 20ms. This means that every 20ms a new measurement is sampled, converted, and transmitted to ISHM. While ISHM may have tolerance for missing measurements, those arriving too late have no value and in sufficient

quantity limit ISHM decision making effectiveness. However, the suite of health-enabled algorithms will all have some variation of real-time requirements that are between 20 and 1500ms to allow enough time for data buffering and processing, although health information evaluated for time *t* has limited utility when the delta from *t* to now becomes large. The upper real-time limit applies mostly to the DFT, as it requires 64 samples, or 1280ms worth of data to operate, plus a enough time to present the result.

Computer systems and embedded systems with real-time constraints commonly employ a real-time operating system (RTOS) to provide a structure for multitasking in time-critical applications [60]. The three major types of operating system architectures in order of decreasing complexity are preemptive multitasking, cooperative multitasking, and simple task scheduling. Note that a simple task scheduler is not necessarily real-time operating systems, as it typically does not contain other ancillary tools, such as interprocess messaging, semaphores, and task control flags. Nevertheless, all operating system architectures operate on the principle of periodically storing the state of the CPU on the system stack, loading the context of another CPU process, allowing the CPU to devote resources to the new process for a period of time (a time slice), then performing another context switch…etc. This periodic change of context is structured so as to best meet the application service requirements.

The performance of an RTOS is a function of the overhead associated with context switching, which is a strong determining factor for the maximum frequency of operating system context switching. However, in an embedded system there are additional factors that influence the overall performance. The number of CPU cycles it takes to perform a context switch depends on the size of the OS Kernel, the time it takes to access memory, and availability of specialized instructions. Additionally, embedded processors operate under limited power and

lower clock speeds. In real-time systems, critical service events are often triggered externally by interrupts, which result in additional unscheduled context switching [61]. Managing these scheduled and unscheduled events is the goal of the RTOS. The next paragraph briefly discusses the primary differences between the major architectures, along with analysis of suitability to the Intelligent Sensor.

While the bare bones task scheduler is not a fully featured RTOS, it is often used as a common approach to obtain inexpensive real-time performance where the real-time requirements are fairly forgiving. The task scheduler simply moves tasks between the CPU for active processing and back to the stack for storage. While it is possible to assign task priorities, the typical objective is to minimize processing such that it is guaranteed that each task will run and complete in a predictable amount of time. It is this determinism that is very difficult to gauge, and during a burst of processing, may result in the system becoming backlogged. A basic task scheduler is not sufficient for the Intelligent Sensor application, where many events are taking place continuously (messages are transmitting, measurements are taken, and health evaluations are processed, with very little room for overhead).

The cooperative multitasking environment is typical among general purpose computers, such as those running the Microsoft Windows, UNIX, or Linux operating systems. Starting with the Kernel, each task is popped from the system stack and allowed to process for a fixed period referred to as a time slice. After the time slice has elapsed, the Kernel allows the next task to run, and so on. While this type of operating system typically does not prioritize tasks (it is possible to change individual priorities, although this is not done on a normal basis), every task is guaranteed to run for a fixed duration of time during each iteration of the task list. However, the task is not guaranteed to have sufficient time to run to completion during a single time slice. The

typical OS system time slice is between 10ms and 15ms for personal computer and workstation platforms. Any peripheral interrupts on these systems are processed in the order received. This type of configuration is not suitable for a real-time application where interrupts are frequent and critical, or events must take place at a fixed time interval whether or not every task has been serviced. For these reasons such an operating system is not ideal for real-time applications. The Rabbit microcontroller supports cooperative multitasking through the use of *costates* and *cofunctions*. These keywords allow blocks of code or functions to be run in sequential order. Processing is switched from one *costate* or *cofunction* to another when the processor encounters nothing else to do in the current *costate* or *cofunction*, such as when a delay statement is used or the *costate* is forced to suspend.

The final choice, the preemptive multitasking kernel, is the best method for the Intelligent Sensor real-time application. Preemptive kernels (also referred to as real-time kernels) contain objects called tasks, which are responsible for executing code on the processor. Program control and flow is handled by the OS scheduler, which checks semaphores, flags, and time delay attributes before attempting to execute a task. Each task is assigned a priority, with the highest priority task assigned to interrupt handling. The event with the highest priority that is ready to execute will start execution until it is preempted by a higher priority task. When preempted, the current task context is stored until that task regains control of the CPU. The user may also cause a task to sleep after some number of executions through the task, thereby voluntarily permitting a lower priority task to run. Tasks may also relinquish control due to the state of a flag or semaphore, which may be set or cleared based on interrupts or other events. Since peripheral interrupts have the highest priority, they preempt all other tasks. The system time slice interrupt is adjustable up to the maximum frequency recommended by the hardware manufacturer, and

can achieve microsecond task switching speeds in many architectures. A graphical view of the execution process is shown in Fig. 3.10.

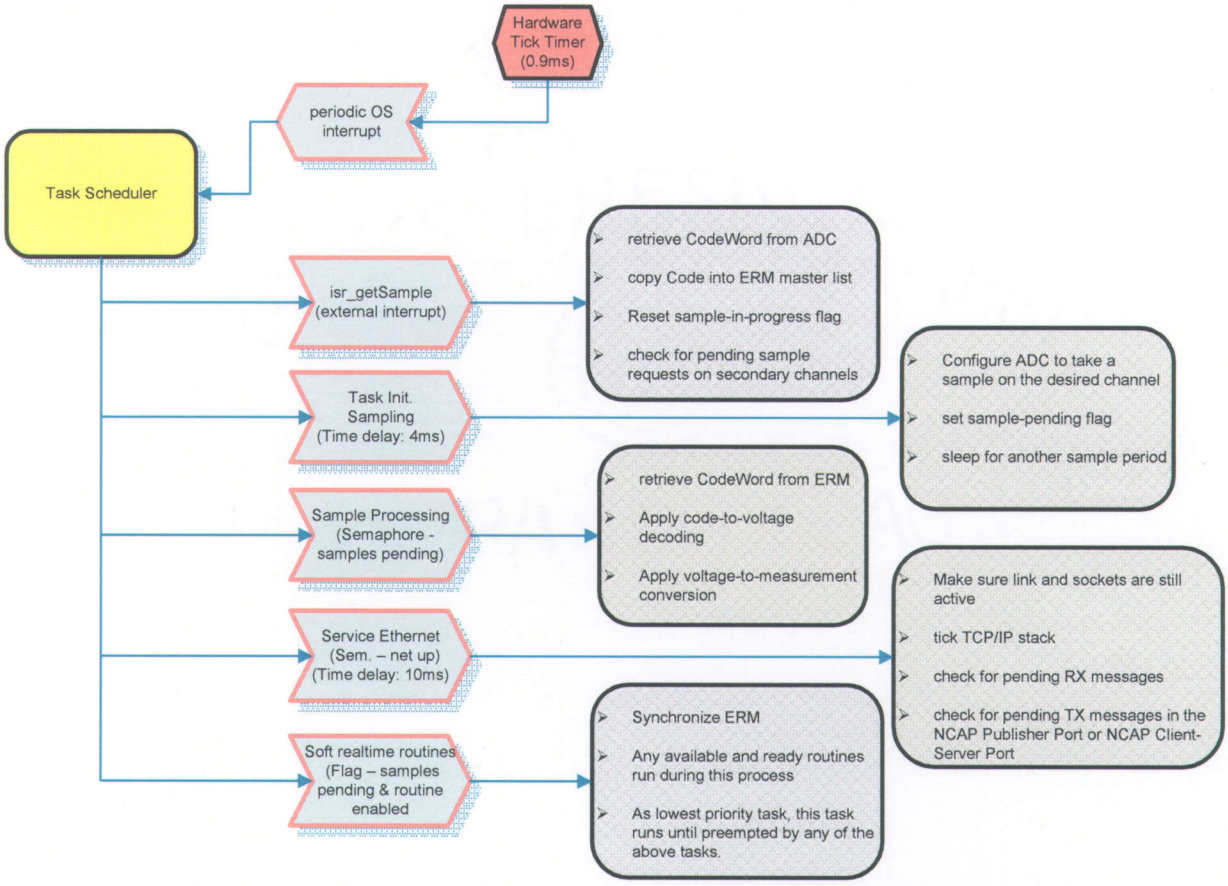


Figure 3.10: Intelligent Sensor operating system diagram.

The principal advantage of preemptive multitasking is that it gives the systems engineer the ability to ensure that critical tasks run on time. Generally, the highest priority tasks encompass actions that are part of the hard real-time requirements. In the Intelligent Sensor, the hard real-time operations include preparing the data converter to sample, processing an external peripheral interrupt when the converter has completed, and retrieving the data from the serial peripheral interface (SPI) bus. Handling of network messages (both inbound and outbound) is a high priority task. Advanced steps may also be taken to control task execution. Determining when a task is ready to run is based on whether the task is suspended (that is, set to delay for a time

interval, which must be greater than the OS tick), the status of semaphores, and the state of flags. There are two types of semaphores; those used for counting and those used to indicate a binary condition. An example of a counting semaphore is one that tallies the number of items in a queue as each new sample is retrieved from the data converter. Once the semaphore value is above zero, the conversion task is enabled, and samples are converted until there are none (the semaphore returns to zero) or the task is preempted by another higher priority task. The binary semaphore is used to indicate if a resource is available (bit is cleared) or if it is in use (bit is set). This type of task control prevents different tasks from corrupting the integrity of data due to simultaneous access. Similar in function is the flag group that can represent 16 flag conditions. A task may be set to run only if certain bit combinations in the flag group are set or cleared. Flags are used to keep track of the progress of health assessment routines. Using flags, semaphores, sleep statements, and prioritization, the real-time kernel is able to control program flow and optimally perform task switching. However, there are some very important pitfalls to avoid. One is to avoid subdividing tasks into such small component parts that there is a large amount of CPU processing overhead performing context switches versus application processing. Another is the problem of priority inversion, where a higher priority task ends up waiting for a lower priority task to release a resource. This means that the lower priority task is now actually of higher priority (or the converse: the higher priority task has been demoted) because of the dependency relationship. Finally, it is also possible to starve tasks, and keep the from running at all! All of these things must be kept in mind when developing a real-time application.

Now that the preemptive multitasking kernel is established as the RTOS of choice for the Intelligent Sensor, a specific vendor must be chosen from the many reliable real-time kernels available. One of the more popular commercial options is Wind River's VxWorks [62].

VxWorks is a full featured development suite. Z-World has developed the *Rabbit OS* that may be purchased as part of a Rabbit development kit [63]. SoftTools Inc., who develops third party compilers and integrated development environment (IDE) support for the Rabbit microcontroller family, offers *TurboTask* as a real-time operating system, which requires the use of the SoftTools integrated development environment (IDE) and compiler [64]. Since SoftTools products are not a normal distribution for Rabbit products, this option would result in additional costs, though not as costly as VxWorks. The last alternative is open source and freeware products, such as Micro-C/OS-2, created by Jean Labrosse, and maintained by Micrum. Micro-C/OS-2, also abbreviated MUCOS, is freely available [65] for download and has been ported to over 100 different platforms, including Intel x86, Rabbit 2000, and Rabbit 3000. Furthermore, Micro-C/OS-2 is approved by the Federal Aviation Administration (FAA) [66] for use in flight critical systems. While price is always an attractive benefit, the fact that C code written for the Rabbit 3000 port of uC/OS-2 can be cross compiled to a machine running an x86 port of uC/OS-2 is particularly attractive for developing virtual Intelligent Sensors and Intelligent Sensor testbeds. The most significant shortcoming of MUCOS compared to other more costly RTOS alternatives is the limitation of 64 individual tasks, with none sharing the same priority level, and no ability to control the time slicing of individual tasks. However, for many embedded applications these issues are workable.

With the Intelligent Sensor OS chosen, the remaining sections focus on two application programmer interfaces (API) developed to support health assessment in Intelligent Sensors. The first focuses on health electronic data sheets (HEDS), and the second concentrates on the embedded routine manager (ERM) for managing embedded health assessment routines.

3.2.1 Health Electronic Datasheets (HEDS)

Thresholds, coefficients, and other necessary parameters used by the suite of health evaluation algorithms and routines are stored and transmitted to the Intelligent Sensor using a health electronic data sheet (HEDS) structure, which is an extension of the IEEE 1451 TEDS. While the HEDS developed for this work are not exhaustive, it serves the purpose of establishing a baseline for future health enabled Intelligent Sensors. The HEDS are transmitted between host and Intelligent Sensor over a network such as Ethernet. HEDS are nonvolatile and are stored in the battery backed RAM on the Intelligent Sensor CPU motherboard. HEDS may also be copied from Intelligent Sensors for archival or cloning purposes. This allows the HEDS to be reloaded to new Intelligent Sensors as need dictates, analogous to the update and recall of TEDS. Section 3.2.3 lists each of the health routines and the respective HEDS structures and messages that were developed for this work.

A routine is identified by its health algorithm class (HAC) and Object ID. The HAC is a four byte unsigned integer that identifies the function of a routine that is consistent across a network of sensors – for example any instances of a statistics routine that computes mean, standard deviation, and correlation would be identified on the sensor network with the same HAC. Generally the HAC is used in the context of publish-subscribe communications where the specific underlying Intelligent Sensor is traceable using a publication topic. The Object ID is a nine byte unsigned integer used in the context of client-server communications to connect to a specific health assessment routine on a specific Intelligent Sensor. The individual routines running on the Intelligent Sensor are considered a special type of IEEE 1451 Function Block, and as such contain a unique Object ID assigned by the sensor in a fully dynamic system, or defined by the system architect at configuration time in a static system. In the static

configuration of the Intelligent Sensor, the lower 4 bytes of the HEDS Object ID encompass the HAC to promote uniformity and consistency.

System state is a system, subsystem, or process wide parameter that identifies modes of system operation where behavior is expected to change. A change of system state can either result in a routine pausing execution, restarting using new parameters, or continuation as though nothing has changed. One example is the engine exhaust thermocouple and run tank purge valve feedback on the MTTP. Assume that the MTTP can transition between the states shown in Fig. 3.11. Each of the abbreviations in the states of the state diagram corresponds to following states: **M**aintenance, **I**dle, **Pre**Test, **R**eady, **T**est, **Pos**tTest, and **A**bort.

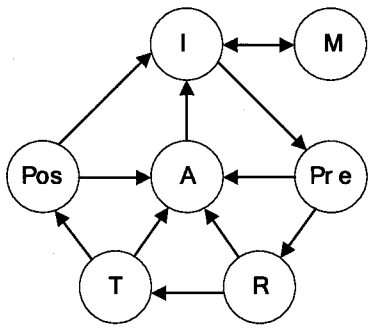


Figure 3.11: Possible state transitions for MTTP.

It is expected that the test article (TA) exhaust temperature will be at ambient during all states except *Test*, *PostTest,* and possibly during *Abort*. During *Test* the exhaust temperature will vary as the engine is throttled. During *Abort* and *PostTest*, a decaying temperature would be expected as the article is no longer energized. In the case of the Intelligent Sensor connected to the run tank purge valve, it may be aware of the various states, but may not change how it uses its health routines between states in the same way as the exhaust temperature Intelligent Sensor. Thus, HEDS allow the health routines to be state aware, establishing health management in the Intelligent Sensor through an optimal piecewise approach. This approach is optimum because it consists of expert system/process/subsystem knowledge, data analysis, feature extraction, and

event detection. At runtime all HEDS parameters that are state dependent are stored in the Intelligent Sensor; the central ISHM management node only needs to broadcast two arguments regarding state changes: That information is the state that the system, subsystem, or process is transitioning to and the UTC network synchronized time (as type *TimeRrepresentation*) at which the transition takes place. There is a section describing the application and functional description of HEDS and a section dedicated to the API for interfacing with HEDS that follow.

3.2.1.1 Application and Organization of HEDS

The payload of HEDS falls into two major categories:

- Basic HEDS
- Health routine configuration parameters

These categories establish the HEDS record for a given routine. Multiple records may exist for a given routine to account for any applicable state changes that the routine must be adapt to. The major categories are discussed in detail in the subsections to follow, followed by a section on the organization of HEDS in the Intelligent Sensor.

3.2.1.1.1 Basic HEDS

The Basic HEDS covers the essential information necessary to identify and configure the routine that is common to all routines. That information includes:

- Total number of supported system state changes (0-32767)
- Manufacturer ID (0-32767)
- Model (0-32767)
- Major Version Number (0-255)
- Minor Version Number (0-255)
- Reporting method (*On Event*, or *At Time*)

- Window Length (2-32767)
- Window Type (sliding window or block window)
- Default state (any valid State_ID)
- Health Algorithm Class (HAC) (1-32767)
- Return Message TypeCode (0-255)
- Return Message Argument Count (0-65536)

The manufacturer, model, and version information is not strictly controlled as is the case of IEEE 1451 TEDS. This information is required simply for version control and creating deployment packages. Reporting method specifies when the routine should report. Valid choices are *OnEvent*, that is, report when an event takes place, or report periodically according to the time epoch specified by *AtTime*. The specific event or time interval is coded into the routine and do not need to be passed as a parameter. The next field is the length of the data window. While the window has physical limits, it is also checked to ensure that there is sufficient memory available to satisfy the memory requirements for a given window size. Window type corresponds to either a sliding window or a block window for the purpose of supplying a list of data points to the health assessment routine, and is not to be confused with a signal processing window that truncates the input signal. A sliding window continues inserting new data points until it reaches a full condition. After reaching the full condition the oldest point is purged upon the insertion of the newest point. This provides continuous coverage of the dataset and eliminates discontinuities at fixed window boundaries. The block method is analogous to a fixed window, where the window grows until it reaches a full condition; the entire window is purged once processing on it has completed. Other window types, such as variable size windows, may be added to fulfill future needs in subsequent revisions. The next Basic HEDS parameter is the default system state

that the routine should assume on startup. Currently, the version of the HEDS engine on the Intelligent Sensor can accept up to 32 state identifiers. The next field is the health algorithm class (HAC), which identifies the classification of the routine on the Intelligent Sensor network. The final two messages describe the return network messages of the routine by TypeCode and argument count. Note that the argument count is only used if the corresponding TypeCode refers to an array type; a non array TypeCode is assumed to consist of only one argument.

It is important to insure that the window type and window length are defined and within bounds for the routine to run. It is also important to ensure the reporting method is properly defined to obtain any desired health messages from the routine.

3.2.1.1.2 HEDS Routine Configuration Parameters

The health routine parameters contain the information required by the routine to perform its analysis. This includes, but is not limited to, thresholds, filter weights, polynomial coefficients, multipliers, or synaptic weights. The number of parameters, their interpretation (integer, float, double precision, etc), and names are the part of HEDS that is solely defined by the user. As it is user defined, there must be a clear and precise control document for each routine that specifies the exact parameters so that applications interfacing with the Intelligent Sensor are aware of the correct formatting for this part of the HEDS. Note that the Intelligent Sensor handles the configuration parameters (in terms of network messaging and internal messaging) assuming the data is presented as an array of octets. As such, there is no need to perform network to host byte conversion, and no interpretation of individual HEDS configuration fields are made by the API. The meaning of the HEDS configuration parameters are only realized inside the actual routine. It is common to create the configuration HEDS as a c-style *struct* that identifies variable names to known data types. It is further recommended that these data types be synonymous with IEEE

1451 TypeCodes for cross-platform consistency. The complete HEDS record (Basic + Configuration) definitions for the routines implemented in Chapter 3.1.1 thru 3.1.8 are included and commented below.

Smoothing Filter

```
typedef struct
{
 uint8 AnalogChannel;          //channel to smooth
 float NumCoeff[9];            //numerator filter coefficients
 float DenCoeff;               //denominator filter coefficient is 1 for a moving avg filt
} x_SmoothingHEDS;
```

Butterworth HPF

```
typedef struct
{
 uint8 AnalogChannel;          //channel to filter
 float NumCoeff[5];            //numerator coefficients
 float DenCoeff[5];            //denominator coefficients
} x_HARHighPassButter4HEDS;
```

Statistics (mean, standard deviation, RMS)

```
typedef struct
{
 uint8 AnalogChannel; //channel to analyze
}x_HARStatsHEDS;
```

Derivatives (1st and 2nd)

```
typedef struct
{
 uint8 AnalogChannel;          //channel to compute derivatives on
  float MaxRoC;                //max limit on first derivative rate of change
```

```
 float MaxAccel;              //max limit on 2nd derivative acceleration
}x_HARCalculusHEDS;
```

Discrete Fourier Transform (FFT)

```
typedef struct
{
 uint8 AnalogChannel;         //channel to perform FFT on
}x_HARDft64HEDS;
```

Noise Events

```
typedef struct
{
 uint8 AnalogChannel;                  //channel to perform FFT on
 float EnergyLimitdB;                  //maximum noise energy from HPF
 uint8    NoiseMetaDataTC;             //typecode for noise metadata
 uint16   NoiseEvtMetaDataLen;         //length of noise metadata
 uint8    NoiseEvtMetaData[32];        //actual noise discovery metadata
 uint16   NoiseEndMetaDataLen;         //end of noise metadata length
 uint8    NoiseEndMetaData[32];        //actual end of noise metadata
}x_HARNoiseHEDS;
```

Spike/Flat Events

```
typedef struct
{
 uint8 AnalogChannel;                  //channel to perform FFT on
 float    MeanPeakDiffLimit;           //minimum mean less peak value in percent  required to declare a flatline
 float    CrestLimit;                  //maximum crest factor
 uint8    SpikeMetaDataTC;             //typecode for spike metadata
 uint16   SpikeEvtMetaDataLen;         //length of spike metadata
 uint8    SpikeEvtMetaData[32];        //actual spike discovery metadata
 uint16   SpikeEndMetaDataLen;         //end of spike metadata length
 uint8    SpikeEndMetaData[32];        //actual end of spike metadata
```

```
  uint8    FlatMetaDataTC;
  uint16   FlatEvtMetaDataLen;
  uint8    FlatEvtMetaData[32];
  uint16   FlatEndMetaDataLen;
  uint8    FlatEndMetaData[32];
}x_HARSpikeFlatHEDS;
```

3.2.1.1.3 HEDS Intelligent Sensor Organization

The HEDS are organized in the Smart Sensor into three major components: The HEDS_Block, HEDS_Record, and HEDS_Data. Refer to Fig. 3.12 for a complete graphical representation of the HEDS organization. The HEDS_Block is the master descriptor that links the API to the individual HEDS Records. Since it is the top level object, it also contains HEDS API specific features such as HEDS version, HEDS_Block identifier, and last HEDS update. In order to link to the HEDS_Records, the HEDS_Block also contains memory references to the individual HEDS_Records, and maintains a count of the total number of HEDS_Records.

The HEDS_Record stores routine specific, but state nonspecific parameters, such as the Basic HEDS. It is the HEDS_Record, along with each of the HEDS_Data corresponding to each system state that fully describes HEDS for a single routine. The HEDS_Record contains memory references to gain access to HEDS_Data based on system state. A NULL reference means that when the system transitions to the corresponding state, the HEDS are nonexistent and the routine does not run. Alternatively, the same HEDS data may be referenced for many different states. This methodology is chosen so that the only duplication in the HEDS API is the memory reference and system state pair (in this version less than a total of 10 bytes), opposed to duplicating the entire HEDS_Data, which is usually much larger in size.

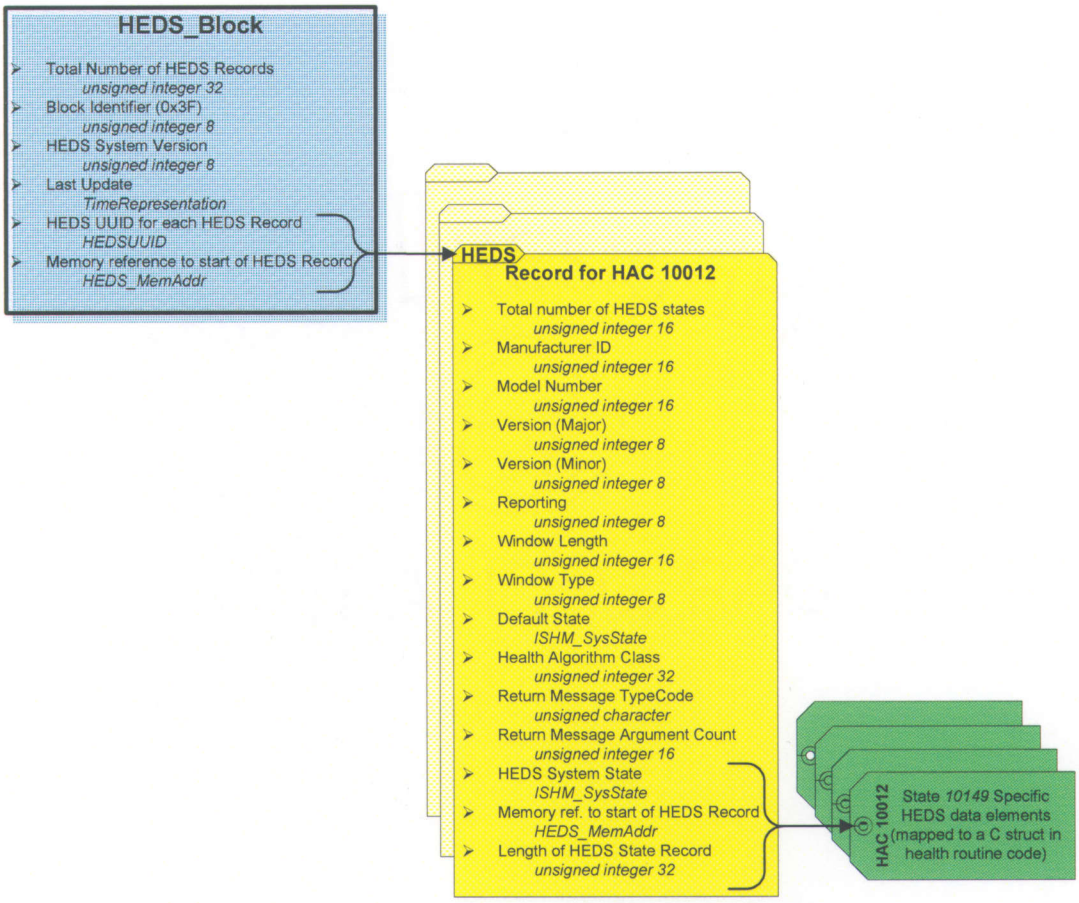


Figure 3.12: The HEDS object definition and hierarchy.

The HEDS_Data contains the actual health routine configuration parameters. This component is specific to each and every routine, and is implemented as mentioned previously as a C-style structure. The HEDS API makes no attempt to interpret the HEDS data; it simply passes it through memory as arrays of octets. As a result, each time a routine is ready to execute, the first line of the routine must map the HEDS to an appropriate local structure. The next section focuses on the HEDS API and network messages that allow other outside objects to interact with HEDS.

3.2.1.2 HEDS API and Network Messages

The HEDS API contains several interface functions for retrieving and decoding HEDS messages from the network, initializing and scanning the HEDS_Block and linked HEDS_Records, and associating HEDS_Data to the individual routines. The API includes the following user accessible functions:

HEDS_Format

HEDS_Init

HEDS_Associate

HEDS_MsgDecodeAndUpdate

HEDS_MsgEncodeAndSend

The *HEDS_Init* function must be run before any other HEDS functions. It checks the Intelligent Sensor for an existing HEDS_Block; if one does not exist it invokes the *HEDS_Format* function, which allocates a block of free memory and established a HEDS_Block. The HEDS memory allocation and record storage process is described in the next section.

HEDS_Associate is responsible for linking system state HEDS_Data to routines supervised by ERM. The function performs the described operation on each routine maintained by ERM in the routine configuration list (RCL). This function starts by retrieving the routine's HEDS_Record and resolves the pointer that corresponds to the anticipated system state change HEDS_Data. Next, the function accesses the RCL record for the same routine, comparing the pointer to the current system state in use to the pointer of the state that the Intelligent Sensor is in the process of switching to. If there is a match (match in this context means each bit in the pointers are equivalent), then no changes need to be made and the routine is left untouched. Comparison is performed on the pointers to system state dependent HEDS_Data rather than on the HEDS_Data itself because several states may utilize the same HEDS_Data, and checking for

equality between two 16-bit pointers is faster and less susceptible to ambiguity than looking for differences in arbitrary length HEDS_Data. If two adjacent states use the same HEDS_Data, it is wasteful to clear the routine, reload the same exact HEDS, and start over. In the event that the pointers do not match, the pointer in the RCL is updated to the location of the HEDS_Data corresponding to the new system state. Next, the routine is reset to purge any incomplete data set. Now, each time ERM synchronizes the routines, the proper HEDS will be loaded via the RCL for the routine. This function needs to be executed when the Intelligent Sensor comes online to load the default system state and each time a sensor-level state change event (SCE) takes place.

HEDS_MsgDecodeAndUpdate is responsible for taking a network message that contains new HEDS information and updating the HEDS_Block with the new information. The function takes a reference to a buffer and the size of that buffer as input arguments. The buffer reference is the location of the argument array from a SET_HEDS message placed on the network and subsequently received by the Intelligent Sensor. The objective of this function is to decode the argument array into machine-usable components comprising the HEDS_Record and HEDS_Data, and then subsequently storing it in local nonvolatile memory. During this process the HEDS Block is updated with the location and size of the new HEDS and the default system state is assigned to the associated routine in the RCL. The Object ID of the message, passed as a separate parameter, identifies the intended routine on a specific Intelligent Sensor that this HEDS is to be associated to. This function relies on *HEDS_Write* and *HEDS_Associate*.

HEDS_MsgEncodeAndSend provides the reverse functionality of *HEDS_MsgDecodeAndUpdate* to prepare the message arguments for response to a GET_HEDS network request. Once the HEDS for the given Object ID is marshaled into an argument array,

the message is passed to the client/server message port. This function utilizes HEDS_Read to retrieve a routine's HEDS Record and then a call to the NCAP Server Port Queue insertion function.

The remaining functions are used internally by the HEDS API, and are generally not intended to be called directly by the application. To better understand the inner workings of HEDS and in the interest of future expandability, the description for these functions follow:

HEDS_Write

HEDS_Read

HEDS_Retrieve

HEDS_Checksum

HEDS_Delete

HEDS_DeleteAll

HEDS_Write addresses the details of securing a block of free memory, clearing that memory, and copying decoded HEDS data into it. This function contains implementation and compiler specific functions for scanning memory, allocating memory, and writing to memory.

HEDS_Read addresses the details of retrieving a routine's HEDS Record. At this time, it is only used by HEDS_MsgEncode. It scans the HEDS_Block for the location of valid HEDS_Records and traverses the list until the desired record is found. It returns the HEDS_Record for the routine and each state specific HEDS_Data.

HEDS_Retrieve returns the reference to HEDS_Data stored for a given HEDS Object ID. This function is used by *HEDS_Associate* for retrieving a pointer to the HEDS data that corresponds to the desired system state.

HEDS_Checksum performs checksum calculation on a HEDS Record. This is currently a stub, and is recommended as future work to ensure HEDS integrity.

HEDS_Delete removes the HEDS_Record and all HEDS_Data that corresponds to a health routine Object ID and frees the memory. Because of memory allocation and freeing issues on the Rabbit microcontroller, this function is not used. In its place, *HEDS_DeleteAll* is used to delete all valid HEDS_Records and HEDS_Data on the Intelligent Sensor. Unfortunately, this means that updating a single HEDS requires removing all existing HEDS. To keep from having to reload all HEDS every time a single HEDS changes, this topic is identified as future work.

The HEDS as described are the first critical part of developing extensible and configurable health enabled Intelligent Sensors. The next piece of the puzzle is an embedded routine manager (ERM) to manage the instantiation, linking, and execution of health assessment routines.

3.2.2 Embedded Routine Manager (ERM)

Memory management is a very important feature of real-time embedded systems. Memory on an embedded system is usually on the order of kilobytes (kB), whereas computer systems are extensible to the order of gigabytes (GB). The failure of the ubiquitous c-style *malloc* memory routine can have dire consequences, and successful calls are indeterministic, making dynamism and real-time conflicting embedded goals. This is an issue of particular concern with the algorithms obtained from GRC in Chapter 2, where most of the algorithms dynamically allocate and free memory for local processing operations and exchange/traverse data vectors through memory pointers and indirect addressing techniques. As a result, routine timing is difficult to determine due to the extensive allocation/deallocation of memory resources and at any given

time there may be multiple copies of the same data in memory. Solving the inline allocation/deallocation problem is easy – allocate required memory at startup and make sure all memory size requirements are statically defined. The more difficult problems are eliminating duplication of the dataset, ensuring that results needed by multiple algorithms are available in the correct order of dependency without redundancy, and ensuring access to the dataset is standardized. All of these issues are addressed by the embedded routine manager (ERM). Before getting into the operation of the ERM, it is important to take a look at the Intelligent Sensor memory map and understand what limitations and advantages are present. The memory map, available from Rabbit Semiconductor for the Rabbit 3000 MCU [67], is shown in Fig. 3.13. The Rabbit 3000 contains a root segment, which contains executable code; data segment which contains C-style (root) variables; stack memory; and the XPC segment, which is extended code memory. The XPC segment is beyond the logical addressing capability of the processor and operates on the concept of an 8kB sliding page, where each individual page is directly addressable via logical addresses. This allows the processor to support up to 1MB of code space.

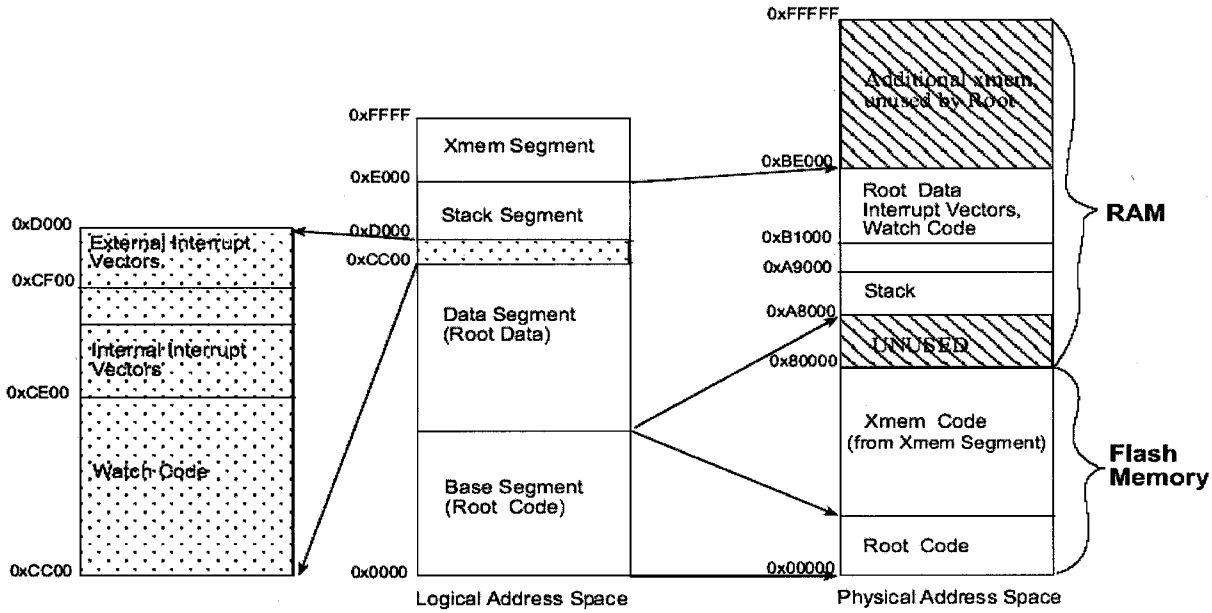


Figure 3.13: Rabbit memory mapping.

It is important to note that by changing the *SEGSIZE* variable, the sizes of both the stack and data segment are changeable. The data segment is capable of containing about 44kB of variables. Since our application uses TCP/IP there is additional memory used for socket buffers, so space in the data segment is valuable. In addition, the full 4kB of stack is needed for the RTOS tasks. While code and data may only be accessed in the regions shown in Fig. 3.13, additional memory (up to 6MB) may be added to the Rabbit memory management unit (MMU) that is not directly accessible. On our sensor there is an additional 256kB of RAM that is battery backed. This is where ERM keeps a large running data buffer and where nonvolatile HEDS are stored. The consequence of using this memory area is that the memory is not accessible by a C variable or a 16-bit pointer, but instead a 20-bit physical memory address. XMEM data is accessible by transferring bytes from a supplied XMEM start address to ROOT variables (such as an array or structure). Since it is not possible for ROOT to approach the capacity of XMEM, the copy operation must be limited to the smallest quantity of data that is needed at any given time, where this extra transfer (both to XMEM and from XMEM) takes an additional dozen clock cycles due to long pointer calls. In the Rabbit code libraries there are three important functions for accessing this XMEM region:

- _xalloc
- root2xmem
- xmem2root

The purpose of *_xalloc* is to update the locations and boundaries of allocated memory. It is abstracted into the *Mem_Alloc* ERM member function to assist with achieving platform independence. The second two functions are used to transfer the contents of a C variable (i.e. data in root memory) to XMEM and vise versa. The *root2xmem* and *xmem2root* functions have

been abstracted to *Mem_Set* and *Mem_Get*, respectively. Also note there are additional XMEM operations for copying between XMEM locations and functions optimized for copying specific datatypes that are not necessary for this implementation, but available for general use. Those functions are: *xmem2xmem, xgetfloat, xgetint, xgetlong, xmemchr, xmemcmp,* and *xmemset.*

The goal of ERM is to address the key issues of providing a dataset to an algorithm or routine for processing that may have dependencies on other routines or algorithms without duplicating the data set. The second premise is to provide an orderly and manageable method to control the execution of the algorithms or routines in an environment where the number of algorithms or routines is variable and varies between development mode and production mode. The next section develops the concept of the ERM and explains how it offers a solution to these issues. Lastly, a section is provided as an application brief for interacting and using the features of ERM through the ERM API.

3.2.2.1 ERM Theory of Operation

The ERM can be envisioned as a massive list of records structured as a doubly linked list, with a control module that supervises access to the list and manages the progression of data processing throughout the list. Each node in the list contains node-specific information and data/payload, as shown in for an example node in Fig. 3.14 used in this application.

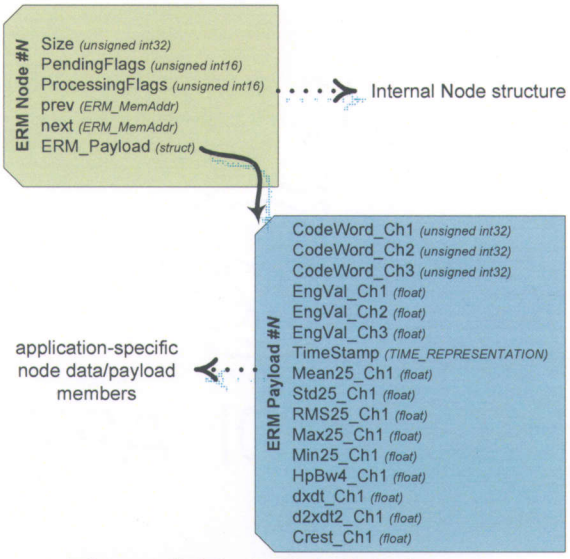


Figure 3.14: ERM Node layout.

The nodes consist of six fields, the first five of which contain node-specific data. The node-specific information includes total size of the node, two flag groups internal to ERM that track the status of each node (if it is processing or already processed) with respect to each available routine, a pointer to the previous node, and pointer to the next node. The final field is the data/payload portion that is used by routines as a data source for processing and a destination for results (routines do not have access to the actual node – only to the payload). The data/payload is extensible in that it may be expanded to allow the list to manage more than just ADC codewords or engineering values, but emerging routine output as well. The organization of the data/payload part is specific to the embedded application and the specific routines that are present. Several other methods for organizing ERM were evaluated. As a queue, it became prohibitive (in terms of CPU overhead, flags, and queue descriptors) to keep track of progression within multiple queues. Single linked lists (used as a FIFO buffer) made it difficult to resolve the relationship between adjacent lists without the convenient previous node address.

In order to control the activity of routines, ERM maintains a globally accessible routine descriptor for each instantiated routine. This descriptor records the occupancy of the window, location of the front (first node), rear (last node) list pointers, Internal ID for the list, window type, window size, the location of the current HEDS_Data, the HEDS_Data length, the Object ID of the routine (also HEDS UUID), pointer to function that is to be executed for full and partial window conditions, location of scratch memory, length of scratch memory, and the routine association. A routine descriptor equivalent to that used in the Intelligent Sensor with identification of where the individual data elements originate is shown in Fig. 3.15.

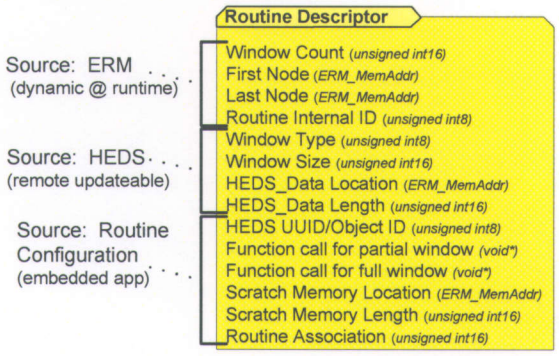


Figure 3.15: ERM Routine Descriptor that stores key routine information.

Some of the data in the routine descriptor is dynamic and updated during runtime (count, first/last node). Other data is obtained from the HEDS_Record (window type, window size) and contains links to the HEDS_Data corresponding to the current system state (running *HEDS_Associate(.)* due to a system state change alert message will result in this link updating, if necessary). The last part of the routine descriptor is proprietary to the specific embedded application including a void pointer that identifies the function to call for a full and partial window, the location and length of routine "scratch" memory that persists between executions, and the routine association parameter. The routine association is the way for conveying inter-routine dependencies, and is represented by a 16 bit value. The high byte and low byte

correspond to *indirect association* and *direct association.* Direct association refers to the dependence of routines that are sequentially registered (and therefore adjacent in Internal ID). The master list entity is automatically assigned highest priority (Level 0), as the master list is responsible for allowing new nodes to be inserted and expired nodes to be removed, causing the Internal ID to start enumerating at 1. The Internal ID is important as it indicates the order of execution (1 executes first, continuing sequentially through the routines). For example, an engineer may wish to run a smoothing routine on the data prior to applying an operation to the data set. To do this, the smoothing routine would be instantiated first, and then the routine for the other operation, with both having the same lower byte association. By doing this the transform will not have access to a set of data until the smoothing routine has first processed it. This chaining is valuable for breaking signal processing operations into small, manageable, and reusable components, although for each routine added to the chain there is a delay equivalent to the routine's window size introduced to each successive routine in the chain. Chaining is not required; routines may also be configured independently, with each operating on raw or some level of preprocessed data. The direct method works when there is one preprocessing routine that is used by one other routine. However, there are cases where a single preprocessing routine is used by multiple routines, which is not efficiently possible in a direct association. The indirect association is used to provide the same chaining capability for multiple routines that may not be adjacent in the priority list. Using the example from above, say there is also a fitting routine that needs the same smoothing. By setting the high association byte to the priority of the low byte of the smoothing routine, the fitting routine will now also depend on the smoothing routine. The association process is shown graphically in Fig. 3.16.

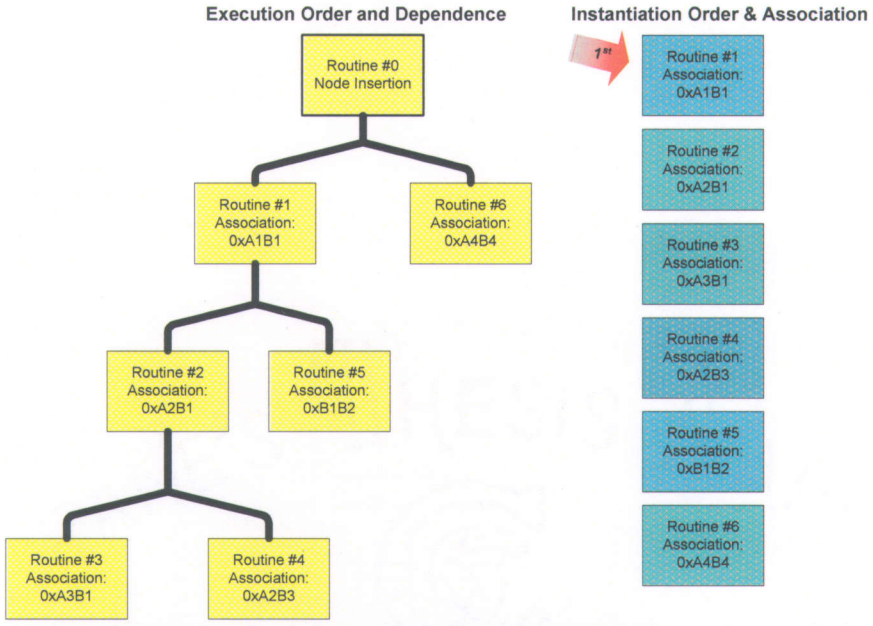


Figure 3.16: Routine dependency based on priority and association.

The final topic to discuss is the representation of data to the individual routines. Since the routine descriptor identifies the first and last node of the routine's dataset, nodes between are accessible by indirectly addressing an integer offset from the routine's first node for maximum speed. Each node is stored entirely in RAM, and the indirect addressing scheme works because the master list is statically allocated at run-time, assuring all nodes are adjacent. In addition to access of elements in a dataset, the user can specify the behavior of the dataset by setting the window type parameter to SLWIN or BLOCK. These mnemonics refer to the behavior of the window[3] and identify if it is a sliding window (SLWIN) or a block processing window (BLOCK). The sliding window operates according to the example in Fig. 3.17, where the occupancy in the window increases as each point becomes available to the routine. Once the window becomes full, it continues to stay full and purge the oldest data point as each new one arrives.

[3] Window in this context refers to the list of nodes that constitute a dataset, not to be confused with a window in the context of signal processing.

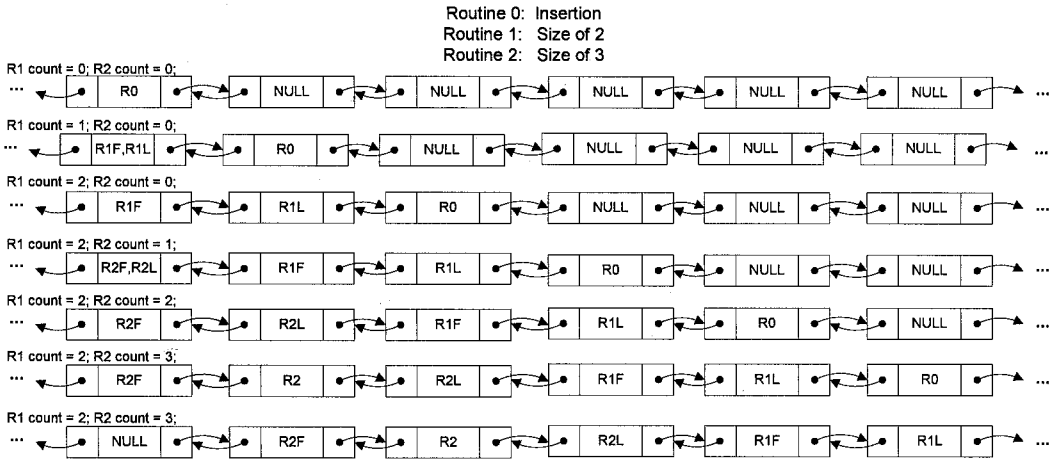


Figure 3.17: Example of a sliding window shared between two routines.

The block window operates differently in that data points are inserted into the list up until the window reaches its maximum limit. When the window is full the routine assigned to the full condition is executed, and when it finishes all nodes in the window are released. The next new node for the given routine is the node immediately after the last one that filled the previous window. Block windows are useful for routines where the algorithm requires a complete dataset before it can operate. Examples are some smoothing functions that utilize the contribution of initial and final data points in producing a weighted average and for highly optimized routines such as the Rabbit implementation of the FFT. There are times when a gap will form in the list, meaning there will be a node in between two adjacent routines that is available to be used by the latter routine, though it is not yet doing so. This happens when block and sliding windows are interspersed due to the behavior of block windows. This is not a problem for the Intelligent Sensor, as during the time the block window builds back up and the processing demands for that routine are minimal, the other routines will be able to catch up. The only caution here is to ensure the ERM master list contains enough nodes to account for any temporal delays in the list. The ERM API also includes I/O functions for externally indicating iteration progress through the

list of executing routines via toggling of an I/O pin and debugging printf statements that count the elapsed routine execution time in milliseconds and display ERM status messages. These features are used to help optimize the loading of ERM during development.

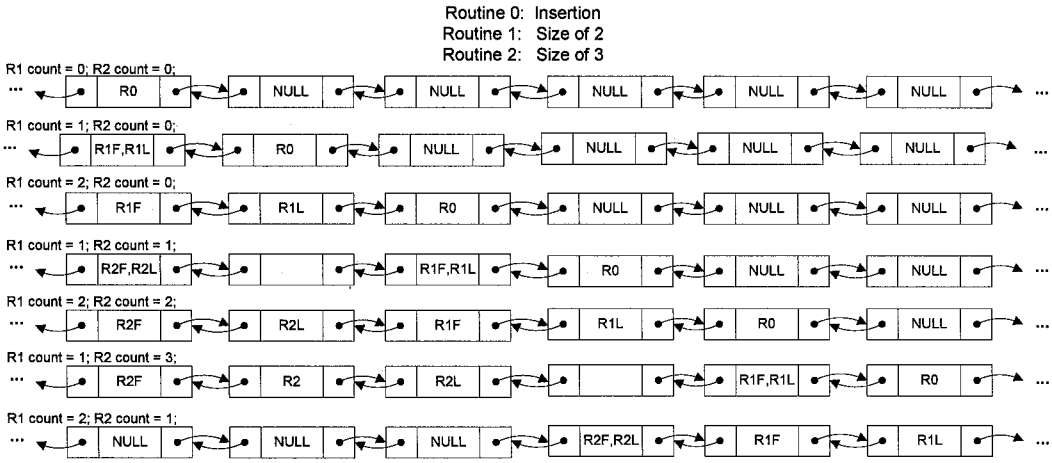


Figure 3.18: Example of a block window shared between two nodes.

This is important because the ERM places priority on timely execution and requires that each routine complete a single iteration of processing (in the case of a sliding window this is processing the newest node, in the case of a block window, this is performing the total computation when the window becomes full) before moving on to the next routine. Due to the embedded real-time constraints there is no ability for an individual routine to reserve a node or dataset for future analysis or defer/commandeer any CPU time. Thus, if ERM is continually overloaded, unprocessed nodes will queue up in the ERM until the list becomes full.

In conclusion, it has been shown in this section that the ERM establishes a solution for managing multiple datasets and routine execution in a flexible yet efficient way that improves the accessibility to Intelligent Sensor resources, touching the key topics of ERM nodes, data/payloads, routine descriptors, routine association, window types, and loading. Now that an

extensive background and explanation of the ERM is presented, the next section will discuss the developer's interaction with the ERM through the ERM API.

3.2.2.2 ERM API

The ERM API is the gateway to putting the power of ERM to work on the Intelligent Sensor. The API consists of a collection of functions that cover everything from instantiating routines to working with individual nodes of the ERM. This section is presented in a descriptive format, starting with the user accessible functions:

void ERM_Init(void)

*uint8 ERM_CreateSubList(uint16 uiAssoc, void (*HandleFull)(), void (*HandlePartial)(), uint32 ScratchSize, HEDSUUID HEDS_UUID)*

*uint8 ERM_Insert(ERM_Payload *ERMPayload)*

*uint8 ERM_Retrieve(uint16 *RoutineID, uint16 NodeOffset, ERM_Payload *ERMPayload)*

*uint8 ERM_Update(uint16 *RoutineID, uint16 NodeOffset, ERM_Payload *ERMPayload)*

void ERM_Purge(void)

void ERM_Sync(void ptr)*

*ERM_HealthEvent(uint16 *RoutineID, TIME_REPRESENTATION *EvtTimeStamp, void *HEDS_EvtData_Addr, void *HEDS_Meta_Addr, uint16 HEDSMetaLen, uint8 HEDSMetaTC);*

*void ERM_HealthPeriodic(TIME_REPRESENTATION *EventTime, void *ParameterPointer, uint16 *RoutineID);*

*uint16 ERM_HealthReport(HealthMsg *MsgArguments, uint8 ParameterPointer[], uint16 *RoutineID)*

The *ERM_Init* function is invoked as part of system startup. This function allocates the memory used by the nodes of the ERM list and initializes the routine descriptors. It also creates the first list, which is commonly referred to the *Master List* or *Node Insertion List*. The scope of the master list is global. The master list is the only list that may accept insertions and allow node deletions. No handle is needed to the master list, as it is automatically assigned an Internal ID of 0 and is internally accessible to the other API functions that must access it.

The *ERM_CreateSubList* function creates and links a new list to a routine. It may be called at any point in program execution after *ERM_Init*, however it is commonly suggested to make all calls together as part of the startup sequence to maximize runtime performance and prevent any possible "out of list" runtime error messages. The first parameter is the list association. The association defines groups of dependent routines. Associations are typically set as *#define* statements at the top of the user's code to any double byte value other than 0. The theory behind list associations was discussed in Section 3.2.2.1 and shown in Fig. 3.16. The next two parameters are the pointers to the code that implements the routine or algorithm that is to be used on full and partially full datasets, respectively. The same code may be used in both cases if suitable to the application, or a NULL may be utilized if there are no activities during one of the conditions. The function whose reference is provided must contain a declaration as shown below in Fig. 3.19. The final parameter required to create a new list is to include an Object ID. The Object ID is synonymous with the HEDS Object ID. The user must provide this as a manual input on this version of the Intelligent Sensor. It is required so that routines will be matched with respective HEDS, and so routines can be identified as 1451 objects. Other list parameters, such as size and window type are provided directly by HEDS.

The *ERM_Insert* function inserts the contents of a locally available ERM payload into the list position immediately succeeding the position of last node in the master list. This function requires very few CPU cycles to complete and may be called from an ISR or an OS task. The input parameter is simply a reference to an ERM payload object (see Fig 3.15).

ERM_Retrieve retrieves the node from the front of a list and provides a memory reference to the node. The user must identify the routine ID to access the appropriate list, as well as the desired node in that list provided as an offset from the start of that list. Valid offsets are 1

(the front node) thru the maximum size of the list (the last node). If the node is valid in the context of the supplied Routine ID, the results are retrieved to the memory space pointed to by the input parameter. This function DOES NOT remove the node from the list; it may be retrieved multiple times if necessary. This function uses indirect addressing to resolve the correct node.

ERM_Update is used to update an existing node in memory. The user must identify the routine ID of the desired list and the node to update identified as the offset from the start of the list. This function essentially replaces the existing node with the node supplied as an input parameter. If the node is valid in the context of the supplied routine, the results are copied from the memory pointed to by the payload pointer into the node. This function uses indirect addressing to resolve the correct node, and the offset range is the same as for *ERM_Retrieve*.

The use of *ERM_Purge* is to flush every list and reset all positions back to zero. This operation does not remove any existing routines from the ERM. The primary purpose is to clear all of the lists when the sensor transitions from a sensing state to an idle state. To ensure the contents of any partial lists are not incorrectly applied to a new measuring operation, *ERM_Purge* must be run in between measuring activities.

The most important function is *ERM_Sync*. This function is solely responsible for managing the lists within ERM as new nodes are inserted and old nodes become obsolete. In addition, this function is responsible for providing individual routines CPU time when ready to run. The determination of readiness to run is a function of routines that have nodes in their scope that require processing and the presence of valid HEDS for the routine. There is one input parameter, a void pointer as required by all operating system tasks. This function is implemented as an operating system task because it must be called on a periodic basis to ensure

the ERM does not get full and to ensure timely health processing takes place. The task is currently implemented as the lowest priority task, consuming all available CPU until a higher priority task becomes active. This ensures that sampling and messaging are higher priority and are not adversely affected (in terms of scheduling) by the health assessment process. This function relies on several smaller functions to perform the work of moving list pointers, executing individual routines, cleaning up each list, and removing nodes that have been expended. For the curious user, the explanation of these functions is left to the in code documentation, though for reference those individual functions are:

*void _ERMProcessRoutine(uint32 *AddrOfNodeToAdd, uint16 *RoutineID);*

*void _ERMInsertNodeIntoList(uint16 *RoutineID, uint32 *currNode);*

*void _ERMRoutineProcessor(uint16 *RoutineID)*

void _ERMStatIOUpdate(void);

*void _ERMRemoveNodeFromList(uint16 *RoutineID);*

*void _ERMRemoveSingleNode(uint16 *RoutineID);*

*void _ERMRemoveMultiNode(uint16 *RoutineID);*

void MasterNodeCleanUp();

uint16 uipow(uint16 uiBase, uint16 uiExponent);

uint16 GetArgLen(uint8 IEEE1451_TC)

The next function under examination is *ERM_HealthPeriodic*. It serves the purpose of providing a routine (namely feature extraction routines) capability to publish periodic results to the publisher port. As input it takes a reference to the timestamp of the event, a reference to the value(s) that are to be published, and the Routine ID to identify the requesting routine. This function then inserts the value(s) into a buffer maintained by ERM for health messages. Each routine has a number of message buffers controlled by the *#define ERM_MaxHealthMsgBuff* parameter for setting the maximum number of buffers, and *#define MaxHealthArgSize* for

specifying the maximum size of each buffer. *ERM_HealthReport* then checks these buffers when packing data + health messages.

The next function is *ERM_HealthReport* whose purpose is to provide the message processing side of routine message publication. This function checks the routine specified by a routine ID input argument for message buffers with any pending messages, retrieves the routine's oldest message contents to the location pointed to by the parameter pointer, and returns an array of health publication information consisting of TypeCode, length, and routine specific information to be used in constructing the IEEE 1451 on-the-wire message. This function needs to be called for each routine active in the Intelligent Sensor that publishes information through *ERM_HealthPeriodic*. The actual encoding of the message as an IEEE 1451 on-the-wire argument array occurs by the caller of *ERM_HealthReport*, which in the case of this implementation of the Intelligent Sensor is encapsulated in the *PubPort_EncodeData(.)* function.

The last function to be examined is *ERM_HealthEvent*, which provides event detection routines with the ability to generate a message in response to an event. The input parameters include the internal Routine ID (passed through ERM as a pointer), current effective timestamp (generally obtained from the current data point), a reference to a value or series of values used to determine the existence of the event if desired (the TypeCode of the value/array is determined by the message argument TypeCode in the HEDS), associated metadata (in this implementation its typically an array of characters, implicitly decoded as ASCII characters to verbosely describe the event, although it could be anything else, such as an integer coding scheme), the length of the metadata, and the TypeCode of the metadata. The last three parameters are obtained directly from the HEDS and need no manipulation by the code; the only user-definable parameters are the timestamp and data value(s). Recall that event detection HEDS contain metadata for both the

onset of an event and the absence of an event. The event is not continually reported to conserve network bandwidth.

Before leaving this section on ERM, the final topic that must be discussed for completeness is the formatting of the health routine code. All functions must contain the prototype as shown in Fig. 3.19, as the calls are indirect calls and there is no checking of the input arguments.

```
void HAR_RoutineName?(uint16 *RoutineID, uint32 HEDS_Record_Addr, uint16 HEDS_Length, uint32 pdata_addr, uint16 pDataSize)
```

Figure 3.19: Health analysis routine function declaration.

The return value is always void, and the name of the function begins with HAR as an abbreviation of health analysis routine, followed by a descriptive name of the routine, and suffixed with either F, P, or nothing to indicate use on full, partial, or all window conditions. The routine designer needs to supply the RoutineID when retrieving or updating nodes, so it is supplied for convenience as a pointer. RoutineID is simply the ERM's internal routine tracking identifier, and is passed through each function to ensure that all code is reentrant. The *HEDS_Record_Addr* is the memory address of the HEDS data for this routine, along with its size stored in *HEDS_Length.* The user must perform a memory copy using *MEM_GET* in order to map the HEDs to a local structure within the routine. It is at this step that the HEDS_Data obtains its meaning to the routine. The final parameter in the routine call is the address to and size of a memory segment that is reserved for the routine to store variables or parameters in between executions. The size of this memory is specified in the *ScratchSize* parameter when the list is created. This too must be mapped to a local structure. The basic template for a health analysis routine is given in Fig. 3.20.

```
void HAR_DFT64F(uint16 *RoutineID, uint32 HEDS_Record_Addr, uint16 HEDS_Length, uint32 pdata_addr, uint16 pDataSize)
{
  ERM_Payload EPL_curr;
  x_HARDft64HEDS HEDS;
  x_HARDft64Vars ScratchVars;
  HealthMsg HealthMsgArguments;

  Mem_Get(&HEDS_Record_Addr, &HEDS, HEDS_Length);     //intepret HEDS binary as a structure
  Mem_Get(&pdata_addr, &ScratchVars, pDataSize);  //retrieve routine variables

  memset(&EPL_curr, 0x0, PayloadSize );  //clear memory for current node
  ERM_Retrieve(RoutineID, ERM_RoutineList[*RoutineID].Count-1, &EPL_curr);      //copy newest node into local memory space


 fftreal(ScratchVars.Descriptors, 64, &ScratchVars.blockexp);

  //Update Messages
  ERM_Retrieve(RoutineID, 0, &EPL_curr);              //copy newest node into local memory space
  memcpy(&HealthMsgArguments.TimeStamp, &EPL_curr.TimeStamp, 8);
 ERM_HealthPeriodic(&HealthMsgArguments, ScratchVars.Descriptors, RoutineID);   //Generate health reporting message using HEDS
                                                                                 //descriptors and desired health reporting variable

  //reset everything for the next batch
 memset(ScratchVars.Descriptors, 0x0, 256 );
  ScratchVars.n = 0;
  ScratchVars.blockexp = 0;

   //Update Memory
  Mem_Set(&pdata_addr, &ScratchVars, pDataSize);  //update scratch memory
}
```

Figure 3.20: Example health analysis routine for 64 point DFT with a full window.

3.2.3 Network Firmware Update

Remote firmware updates are another primary operating requirement for the Intelligent Sensor and serve a variety of purposes. Firmware update is a mechanism for updating the Intelligent Sensor operating system, IEEE 1451 subsystems, health assessment routines (not to be confused with the routine parameters embodied in HEDS), and other proprietary application code. Intelligent Sensors running in a lab environment may be frequently flashed if the coding workstations are separated from the development and testing facilities[4]. The next section discusses methods of in situ firmware updates, followed by a section addressing the chosen Intelligent Sensor implementation.

3.2.3.1 Methods for Updating Firmware In Situ

There are several aspects of firmware updates. Processors that have dynamic code linking capability would permit the ability to selectively update portions of the firmware and relink it to the existing base operating firmware, much in the way a Windows PC can download new DLLs or executables. The new code, unless critical to the operating system, usually does not require the system to be restarted. For typical small scale embedded systems, this type of advanced capability is new and has not been adequately field demonstrated in mission critical systems. Typically, CoTS devices such as MP3 players, calculators, and other small embedded equipment statically update the complete operating firmware by downloading the new firmware over an interface network or bus to a separate memory, then restarting the microcontroller with a special instruction so it will first copy the new firmware into protected program memory, verify validity, then restart in normal operating mode. Rabbit processors support a similar type of remote firmware update on configurations that contain two FLASH memory chips by using the first one

[4] Though a very useful feature on the ground, mission safety protocol requires Intelligent Sensors certified for space flight that perform in mission critical roles to have any type of firmware update mechanism disabled in hardware once the checkout process is complete.

(0x0 – 0x40000) for storing the new program binary and the second (0x40002 – 0x80000) for running the code that handles the download process [68]. The code entry point is controlled by toggling the chip select pin on the respective FLASH memory chip. This method is potentially dangerous as the validity of the downloaded binary is not verified until after it is downloaded and the previous program is overwritten. There is also the risk of a power failure, which could cause the CPU to reboot into code that is not complete.

There is also a solution for Rabbit based devices that utilize a daughter board. The daughter board interfaces to the JTAG and several Rabbit I/O pins. The procedure starts with downloading the firmware image to the Rabbit CPU and storing it in any available buffer. Once the download is complete, it is transferred to the daughter board. The daughter board then places the Rabbit CPU into program mode and loads the new firmware via the JTAG interface. The daughter board is essentially emulating the connection to the development workstation and integrated development environment (IDE). This method surpasses the reliability of the online updating process described earlier, at the cost of additional hardware.

There is a more clever approach that leverages the advantages of hardware based updates without requiring a second FLASH chip and mitigates the risks of a bad firmware image. The Rabbit CPU gives us a special advantage because it can execute code from either FLASH or the root segment of RAM (refer to the Rabbit memory map in Section 3.2.2). As such, it is possible to compile a small application into FLASH that on startup checks some status flags (also stored in FLASH) to determine what firmware should be copied into RAM, where to copy it from, validate the CRC, etc. After copying the firmware, the execution point is changed to start execution in RAM. After a subsequent warm boot, the Rabbit processor is running the newly copied firmware. Sample libraries are provided by Rabbit Semiconductor for developing this

type of firmware download strategy [69]. The application in FLASH will persist between power cycles, and eliminates the need to attach to the JTAG port to reload firmware. A graphical representation of this process is shown in Fig. 3.21.

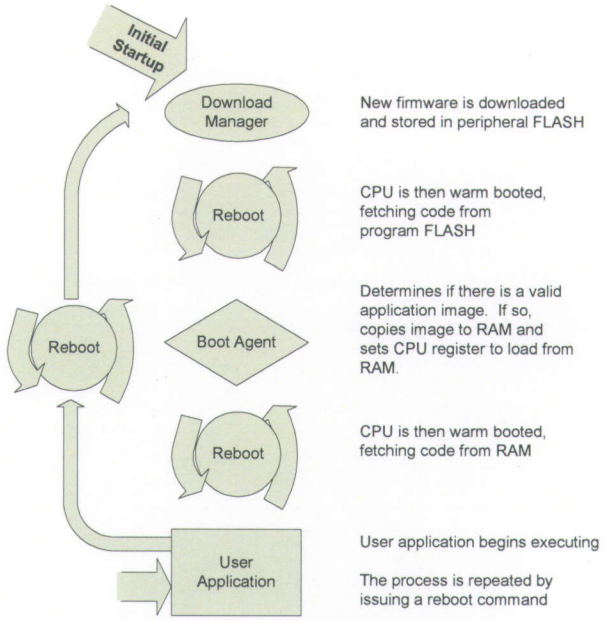


Figure 3.21: Smart Sensor firmware update process.

The downloaded programs may be stored in the onboard program FLASH or peripheral FLASH if it is available. This method supports multiple program binaries, up to the maximum amount of available memory. This is advantageous, because even if the program is downloaded but fails (for example, a bad pointer assignment in the code), a previously downloaded and verified program can be designated as the failsafe. There are also configuration options to avoid downloaded programs that exceed a user settable limit for watchdog timeouts and resets. This means there is no risk of affecting the reliability of the Intelligent Sensor due to bad firmware, whether the firmware was corrupted during the download/copying process or through an application fault. The Intelligent Sensor deviates from this implementation strategy by only storing images in peripheral FLASH to maximize the available program FLASH for other uses.

3.2.3.2 Implementation and Interface

The sensor contains a small boot agent that resides in FLASH that performs the function described in the prior section of determining the image to load, loading the image, and restarting the CPU. Since the starting address of FLASH is 0x0, the boot agent will be first to start in a default configuration that may exist after an unexpected power failure or software exception. Variables that correspond to behavior (where to load the image from, which specific image to load, etc) are stored in the user area of FLASH, and are protected between power cycles. In this implementation firmware images are stored only in the 8MB of serial FLASH peripheral memory. The reason for this is to provide more execution FLASH to maximize addressable memory space. The firmware update suite includes the boot agent discussed previously and a download manager application that resides in FLASH. The Intelligent Sensor is preconfigured out of the box to boot to the download manager, where application software may then be downloaded and executed over the Ethernet network. Should the download manager or boot agent become corrupted, they must be reloaded via the JTAG interface.

The only interface functions that the user needs to call are the download system initialization *(dlp_init)* and the reboot *(dlp_reboot)* command. All other activity is handled behind the scenes and without user intervention. The compiler must be set to *compile code to flash, run in RAM* for the process to work successfully. The options for transferring the firmware image from host to client is through an HTTP interface or a TCP socket stream. The TCP stream is useful for automated or machine-based reconfiguration scenarios, such as the proposed Health Assessment Database or Health Assessment Repository [70] where user interaction is minimal. The HTTP interface is ideal for development use or manually configured and maintained sensor networks, where firmware will be distributed to a small handful of sensors by an individual. Considering the current milestones of ISHM and the availability of HTTP-based code examples,

HTTP via web browser was the chosen interface. Images of the HTTP based download agent are shown in Fig. 3.22 thru 3.23.

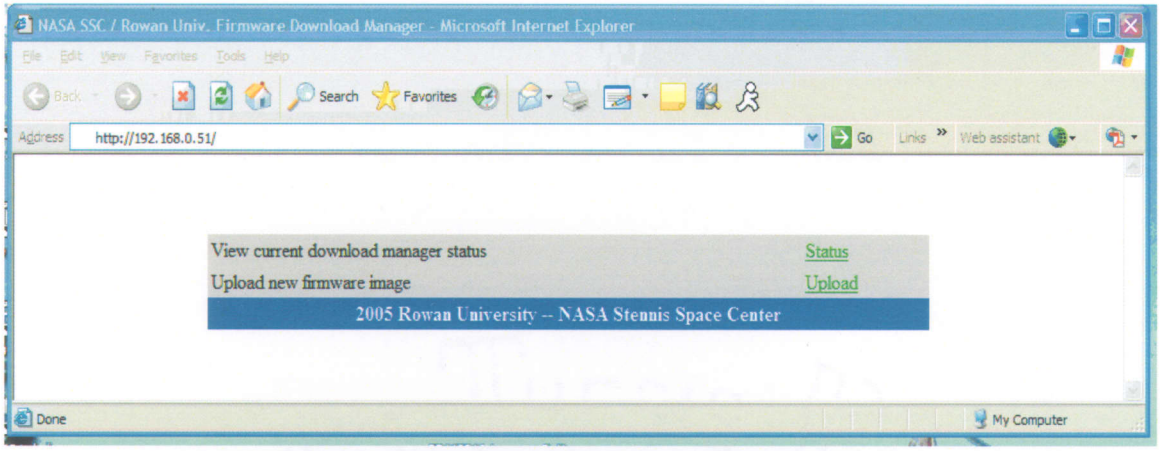


Figure 3.22: Download manager main page.

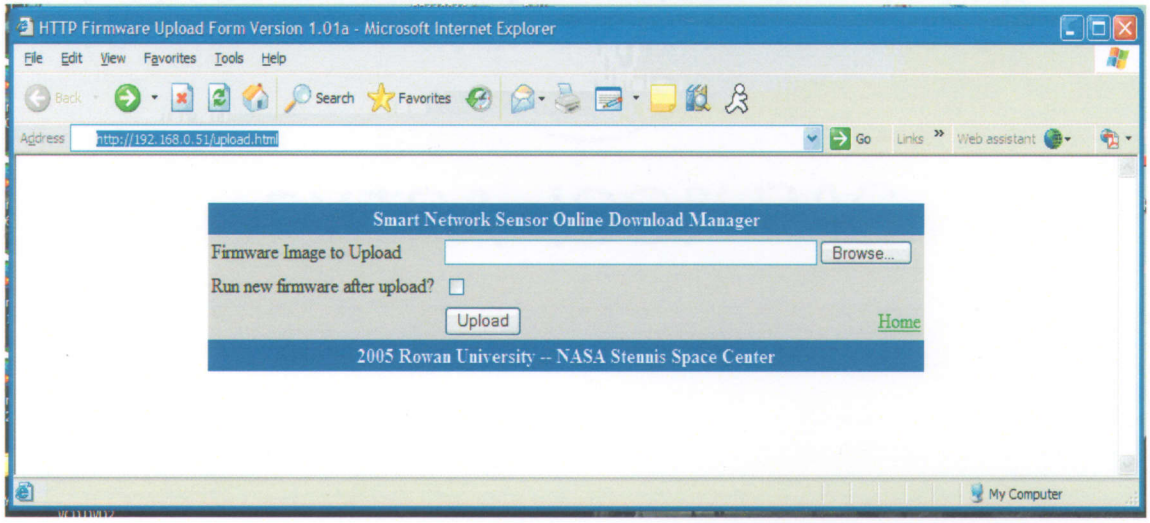


Figure 3.23: Download manager image upload screen.

With the success of this download demonstrator, it will be recommended as part of the future work to implement a TCP download interface to support automated update capability through ISHM.

3.3 Intelligent Sensor 1451 Interface

The final requirement of the Intelligent Sensor is a functional IEEE 1451 interface. It is important to realize that the Smart Sensor standard is very comprehensive and looks ahead at

sensors and actuators that are part of a fully dynamic, self identifying, self configuring network. All of that potential capability is not required for every application or to achieve compliance, nor is it reasonable to expect a fully mature product at the end of an early development cycle. Therefore the development objective is to reduce IEEE 1451 to the bare minimum set of objects and capabilities to allow the Intelligent Sensor to interface to an IEEE 1451 network. This reduction has been coined "1451 lite" for its minimalist approach. As the IEEE 1451 paradigm was discussed in depth in Chapter 2, the focus here is on the actual parts of 1451 that are implemented to provide that minimal interface, as well as the added capabilities in support of health assessment that are beyond the domain of the standard (however follow conventions and techniques conforming to the standard). It is expected that there will be revisions to the implementation as the Intelligent Sensor/ISHM mature and as a result of future revisions of the 1451 standard. In terms of an interface control document, the following sections are organized by the two major components of the NCAP interface: Publication/Subscription messaging and Client/Server communications. The following sections identify each message supported by the Intelligent Sensor, along with functional purpose, explanations of arguments, formatting, and any deviation from the standard. The messaging structure is loosely based on the Boeing/Agilent Open1451 interface control document [33].

3.3.1 1451.1 Publications

Publications are messages that do not have a specific end host, and are primarily used for discovery, configuration, batch operations, and communicating information from one source to multiple destinations with minimal network traffic. The messages are sent to every node on the network, and each individual node examines the message to determine if it is relevant. The messages minimally contain a message type identifier, message length, header length,

publication key that identifies the purpose of the message, and a publication domain that is used as a message domain filter. The IEEE 1451 publication port is synonymous with the broadcast or multicast address of an Ethernet network, though publications are still filtered by the Ethernet stack at the port level. The publication/subscription port reserved for commands is 11000, and port 12058 is reserved for measurement data and health information. This differentiation keeps the Intelligent Sensors from filtering and decoding one another's data packets, since the publication contents are not decoded until the application layer (refers to the ISO model in Fig. 3.24). This capability may be of use in future applications where Intelligent Sensors are aware of neighboring sensor activity, such as in the application of swarm intelligence [8] or multi-sensor data fusion [20, 71].

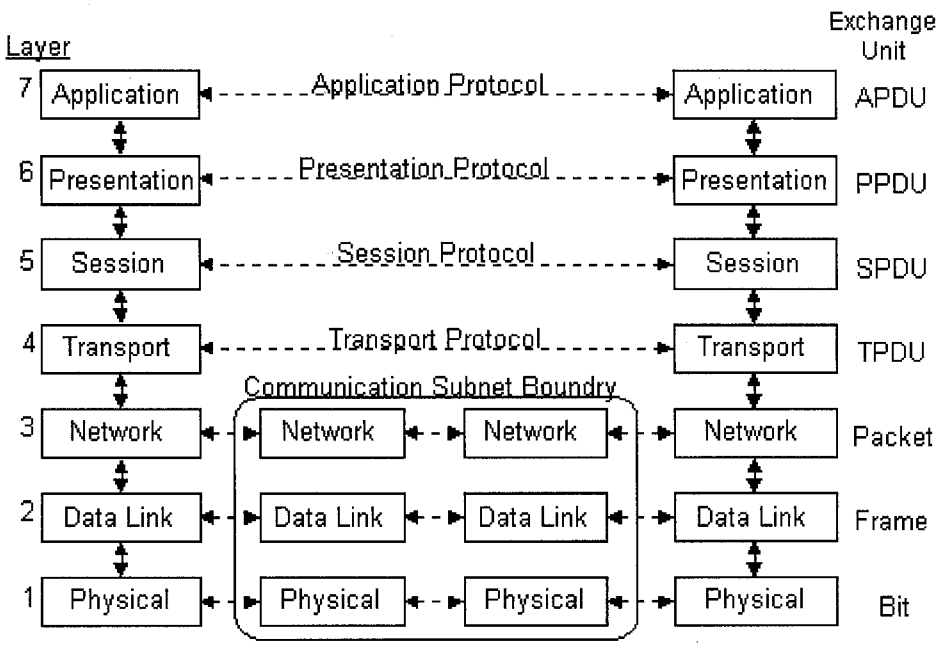


Figure 3.24: The ISO/OSI protocol model.

For a dynamic sensor network, the standard is designed so that each sensor may be discovered and then subsequently queried to determine its capability and functions. Batch operations that apply to a group of Smart Sensors may include tasks such as commence

measurement, halt measurement, reboot, or identify. Even on a statically configured network, some level of self identification is required, as is shown in the announcement messages to follow.

The first interface message that will be present on the network during sensor network commissioning is the *Request_NCAP_Block_Announcement*. This packet is transmitted by a sensor network configuration tool, and has the form of Fig. 3.25.

Publisher Port Message Format:

Type	Msg Len		Header Len		Pub Key	Publication Domain								Pub Topic
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
0xF7	0x00	0x0F	0x00	0x0F	0x04	O_0	O_1	O_2	O_3	O_4	O_5	O_6	O_7	0x00

Publisher Port Message Structure:

Name	Type	Details
Message Type	uint8	0xF7
Total Message Length	uint16	0x000F
Length of Header	uint16	0x000F
Publication Key	uint8	PSK_REQUEST_NCAP_BLOCK_ANNOUNCEMENT (0x04)
Publication Domain	uint8[8]	0xFFFFFFFFFFFFFFFF
Publication Topic	uint8	0x00

Figure 3.25: Interface for Request NCAP Block Announcement message.

The intention of this message is to solicit a response from each NCAP object with the pertinent information needed to establish further communication. As such, the message contains no topic or payload, only the header as described in the previous paragraph, with the Publication Key (0x04) identifying the purpose of the message. By default, an NCAP that initializes and boots successfully will be default to BL_INACTIVE state. In this state, the Smart Sensor is permitted to listen for and respond to these announcement messages. The Publication Domain is a technique for filtering messages between groups of sensors on the same network. By default, the Intelligent Sensors are configured to broadcast to and receive publications from the 0xFFFFFFFFFFFFFFFF domain, although this will later be modified to divide sensors into functional groups. Once the sensor receives the request to announce itself, it in turn responds

with an *NCAP_Block_Announcement* message. In an alternate configuration the Smart Sensor may be configured as part of its boot sequence (the startup configuration set) to publish this message automatically and periodically until instructed otherwise.

Publisher Port Message Format:

Type	Msg Len		Header Len		Pub Key	Publication Domain								Pub Topic	Number of Args in Pub. Contents		Arg. TC	Number of Bytes in Arg (Object Tag)		...
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	
0xF7	0x00	0x2F	0x00	0x0F	0x02	O_0	O_1	O_2	O_3	O_4	O_5	O_6	O_7	0x00	0x00	0x02	0x1A	0x00	0x0B	

Argument #1: Object Tag											Arg. TC	Argument #2: Object Dispatch Address														
												IP				Port		Object ID								
20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46
$ChID_{10}$	$ChID_9$	$ChID_8$	$ChID_7$	$ChID_6$	$ChID_5$	$ChID_4$	$ChID_3$	$ChID_2$	$ChID_1$	$ChID_0$	0X16	IP_3	IP_2	IP_1	IP_0	P_1	P_0	0x03	MAC_5	MAC_4	MAC_3	MAC_2	MAC_1	MAC_0	OID_1	OID_0

Figure 3.26: Interface response to an NCAP block announcement.

The return message published on the sensor network is shown in Fig. 3.26. This message contains two important pieces of information that identify the responding NCAP: the Object ID and the Object Tag. Both serve as an identifier of the NCAP node (or more generally, any object on the sensor), however the Object Tag is a logical identifier issued at configuration and the Object ID is algorithmically created for each Smart Sensor object. To keep Smart Sensor technology consistent with SSC deployment and naming conventions, the NCAP Object Tag is the technical identification of the sensor or actuator node. Examples of Object Tags are TT1185GM, TT1206GO for temperature transmitters; PE1134GO, PE1140GO for pressure sensors. This nomenclature will suffice until the Smart Sensors are configured to support multiple sensing/actuating devices per NCAP. At this time a neutral NCAP Object Tag will be chosen that is unrelated to the individual transducer nodes, but instead the function (i.e. B1 GOX Pressure) or location (i.e. B1 Subsystem A34). The Object ID, on the other hand, provides enough information to direct the message to the specific destination object on a specific sensor. The Object ID consists of a one byte identifier indicating how the ID is generated, the Sensor's MAC address, and two bytes that correspond to an UUID that is unique to each specific object

on the sensor. The two byte UUID need not be unique among all sensors on the network as the MAC fields eliminate inter-sensor ambiguity. As part of the announcement process, the reply message also contains an Object Dispatch Address (ODA). The ODA specifies how an object can locate and send a message to the announcing sensor's NCAP. The ODA is a concatenation of the Sensor's IP address, PORT and NCAP Object ID. The IP address/port combination is used to make a logical connection to the Smart Sensor, which in turn establishes a connection to the sensor through the network infrastructure. Once the individual sensor has been resolved, the Object ID is then available to address the NCAP. This completes the NCAP announcement process along with interpretation of the *NCAP_Block_Announcement* message is shown in Fig 3.27.

Publisher Port Message Structure:

Name	Type	Details
Message Type	uint8	0xF7
Total Message Length	uint16	0x002F
Length of Header	uint16	0x000F
Publication Key	unint8	PSK_NCAP_BLOCK_ANNOUNCEMENT (0x02)
Publication Domain	uint8[8]	0xFFFFFFFFFFFFFFFF
Publication Topic	uint8	0x00
# Args in Pub Contents	uint16	0x0002
1st Argument Type Code	uint8	OBJECT_TAG_TC (0x1A)
# Bytes in 1st Arg	uint16	0x000B
Object Tag	uint8[11]	Sensor Channel ID (ChID)
2nd Argument Type Code	uint8	OBJECT_DISPATCH_ADDRESS_TC (0x16)
ObjectDispatchAddress.IP	uint8[4]	Sensor Publisher IP Address
ObjectDispatchAddress.Port	uint8[2]	Sensor Publisher Port
ObjectDispatchAddress.OID	uint8[9]	NCAP Object ID

Figure 3.27: Interpretation of NCAP Block Announcement message.

There are two more optional operations associated with the announcement process. They are *Ignore_NCAP_Block_Announcement* and *Force_NCAP_Block_Announcement.* While they are optional, they are important for preventing a broadcast storm resulting from the announcement process, especially for large sensor networks. A possible implementation is to locally invoke the *Ignore_NCAP_Block_Announcement* after receiving and replying to a *Request_NCAP_Block_Announcement.* This sequence will prevent the sensor from responding

to any other announcements. The network configuration application may try several more *Request_NCAP_Block_Announcement* messages to ensure that all sensors had a chance to receive the message and generate a response. The configuration application can issue a publication with the *Force_NCAP_Block_Announcement* publication key to solicit an announcement from a sensor that is ignoring requests for announcements.

Now that the sensor is registered on the network and in the idle state, the next step to reaching full operational status is to determine what messages the sensors subscribe to. On a dynamically configured network, the configuration tool then queries each sensor for its supported operations. At this point, however the ISHM Smart Sensor network is static, so all sensor supported operations and messages are known prior to commissioning.

The final step to setting up the sensor network is to set the individual blocks on board each Intelligent Sensor into the BL_ACTIVE state. Recall from Chapter 2, there are three major blocks consisting of the NCAP Block, Transducer Block, and Function Block. Up to this point we have been discussing the NCAP Block, however to achieve an operational Smart Sensor we need to also set the Transducer Block active, and to engage in health assessment we need to start the respective Function Block(s). There are two methods to start the remaining blocks. One is to start each one individually, using Client/Server communications. This method is discussed in the section on client/server messaging. The other method is to issue a blanket publication to all sensors in a publication domain using the *NCAP_Block_GoActive* publication. The publication message format and message mapping is given in Fig. 3.28. A Publication Key of 0x07 is defined for this operation. This function causes the NCAP Block, as well as any owned block (transducer block and function blocks) to transition to the BL_ACTIVE state. This publication does not elicit a response message from the activated blocks.

Publisher Port Message Format:

Type	Msg Len		Header Len		Pub Key	Publication Domain								Pub Topic
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
0xF7	0x00	0x0F	0x00	0x0F	0x07	O_0	O_1	O_2	O_3	O_4	O_5	O_6	O_7	0x0

Publisher Port Message Structure:

Name	Type	Details
Message Type	uint8	0xF7
Total Message Length	uint16	0x000F
Length of Header	uint16	0x000F
Publication Key	unint8	PSK_NCAPBLOCK_GO_ACTIVE (0x07)
Publication Domain	uint8[8]	0xFFFFFFFFFFFFFFFF
Publication Topic	uint8	0x0

Figure 3.28: NCA_ Block_GoActive publication and message structure.

The status of the blocks may be ascertained by issuing the client/server communication *GetBlockMajorState* either before or after the transition is requested.

The next publication changes the transducer sampling frequency. While potentially useful, blanket setting of the sampling rate must be used with care (for example, it would be disastrous to have both accelerometers and thermocouples all sampling at 50Hz)! An effective strategy for this publication is to segment network domains by sensor type or function – thus causing the publication to be received by a select subset of the entire sensor population. On the SSC RETS, there are two levels of data acquisition: High speed DAS from 50 kHz to 200 kHz and Low speed DAS at 250 Hz. Sensors deployed as part of the existing RETS must fall into either of these two categories, and as such this publication is very useful (see Fig. 3.29 for the packet format and message structure). Be aware that migration to Smart Sensors supporting multiple analog channels, where individual channels may contain different types of transducers with different sampling requirements, will require the development and use of a client/server message to address individual Transducer Block elements.

Publisher Port Message Format:

Type	Msg Len		Header Len		Pub Key	Publication Domain								Pub. Topic	# Args. in Pub Contents		Arg. TC	Sample Rate (Fs) Selection			
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
0xF7	0x00	0x16	0x00	0x0F	0x81	O_0	O_1	O_2	O_3	O_4	O_5	O_6	O_7	0x0	0x00	0x01	0x0A	Fs_3	Fs_2	Fs_1	Fs_0

Publisher Port Message Structure:

Name	Type	Details
Message Type	uint8	0xF7
Total Message Length	uint16	0x0016
Length of Header	uint16	0x000F
Publication Key	unint8	PSK_USER_DEFINED_CHANGE_SAMPLE_RATE (0x81)
Publication Domain	uint8[8]	0xFFFFFFFFFFFFFFFF
Publication Topic	uint8[16]	NULL (0x0)
# Args in Pub Contents	uint16	0x0001
1st Argument Type Code	uint8	FLOAT32_TC (0x0A)
Sampling Rate Divider	float32	Desired sampling frequency in floating point decimal

Figure 3.29: Interface message and structure of sample frequency change publication.

In this message the Publication Key is set to 0x81, which is part of the user defined keys. Upon deciphering the Publication Key, the sensor then knows that the arguments to follow describe the sampling rate interpreted as an IEEE 754 single precision floating point number (float32 TypeCode). The Intelligent Sensor performs checking prior to setting a new sampling rate to ensure the value is within the allowable range of the sensor. If it is out of range, the sampling rate is not changed. The sampling rate is not retained between resets or power cycles, reverting to the default rate of 1 Hz. For this version of the Intelligent Sensor, the minimum is 1Hz and maximum is 50Hz, with a step size of one millisecond. Due to the minimum time resolution of milliseconds (as a result of the 1 ms periodic interrupt), the sampling rate argument is converted to sampling period and rounded to the nearest whole millisecond. For example, a sampling rate of 60Hz (16.667ms period) becomes 62.5Hz (16ms period), as the 670μs is beyond the resolution.

The next publication is the user-defined *NCAP_Block_GoInactive* publication. Curiously, while there is both a client/server and publication version of *GoActive*, the converse is not true. This could be for the fact that the most difficult and important part is getting everything online and running; for once all of the nodes are known transitioning to inactive as simple as

switching through each individual NCAP to shut it down. Regardless, in this implementation a publication version of *GoInactive* is useful for quickly and easily shutting down the network post test. The packet structure for this user defined publication is shown in Fig. 3.30.

Publisher Port Message Format:

Type	Msg Len		Header Len		Pub Key	Publication Domain								Pub. Topic
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
0xF7	0x00	0x18	0x00	0x0F	0x80	O_0	O_1	O_2	O_3	O_4	O_5	O_6	O_7	0x0

Publisher Port Message Structure:

Name	Type	Details
Message Type	uint8	0xF7
Total Message Length	uint16	0x0018
Length of Header	uint16	0x000F
Publication Key	unint8	PSK_USER_DEFINED_NCAPBLOCK_GO_INACTIVE (0x80)
Publication Domain	uint8[8]	0xFFFFFFFFFFFFFFFF
Publication Topic	uint8	NULL (0x0)

Figure 3.30: Interface message and interpretation for NCAP_Block_GoInactive publication.

In this case, the message consists of only a header, as the Publication Key provides the necessary indication of the requested action. The consequence of issuing this publication places the NCAP Block, any active Function Blocks, and any active Transducer Blocks into the BL_INACTIVE state. There is no reply message generated by any of the blocks that respond to this message.

In addition to commands, measurement data is also handled through the publisher port. Publication of measurements allows for multiple data-centric entities to collect measurements from a single message without additional network overhead. While 1451 supports multiple forms of measurement publication described in Table 3.1, we are focusing on representing only normal data from that list. As with all other publications reviewed thus far, the first 14 bytes is the same header format, with the Publication Key identifying the purpose of the message as PSK_PHYSICAL_PARAMETRIC_DATA (0x0B), an appropriate publication domain, and a publication topic that consists of the SSC sensor channel ID.

Table 3.1: IEEE 1451-1999 Publication Content Codes for parametric data publications.

Enumeration	Value	Meaning
PCC_NORMAL_DATA	0	The usual contents of the publication for the defined topic
PCC_METADATA	1	The information requested as a result of the invocation of the *PublishPublisherMetadata* operations.
PCC_BOTH_META_AND_NORMAL_DATA	2	The publication contains both the meta and normal data, defined however, for the application or publication.
PCC_GROUPED_NORMAL_DATA	3	Same as 0, but for a group of similarly formatted data
PCC_GROUPED_METADATA	4	Same as 1, but for a group of similarly formatted metadata.
PCC_GROUPED_BOTH_META_AND_NORMAL_DATA	5	Same as 2, but for a group of similarly formatted meta and normal
Reserved Values	6-127	
Open to Industry	128-255	

The publication topic contains 16 byte positions, right padded with zero if necessary. Deviating from the standard, the publication topic is to be decoded as ASCII characters, opposed to integer octets. The next field is the number of arguments expected in the publication contents, which is two for normal data publications. The first argument has a TypeCode of UINTEGER8_TC that corresponds to the publication content code (PCC). The Intelligent Sensor knows to expect a

Publisher Port Message Format:

Type	Msg Len		Header Len		Pub Key	Publication Domain								Publication Topic											
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	...
0xF7	0x00	0x33	0x00	0x1E	0x0B	O_0	O_1	O_2	O_3	O_4	O_5	O_6	O_7	$ChID_{10}$	$ChID_9$	$ChID_8$	$ChID_7$	$ChID_6$	$ChID_5$	$ChID_4$	$ChID_3$	$ChID_2$	$ChID_1$	$ChID_0$	

Publication Topic						Number of Args in Pub Contents	Arg.1 TC	Arg.1 PCC	Arg.2 TC	Arg.2 PhyParam	SingletonData.TimeStamp. Sec (UTC seconds)				SingletonData.TimeStamp. NanoSec (UTC nanoseconds)				Num Args in Arg. Array "Value"		Arg. TC	SingletonData.Value IEEE 754 32-bit floating point			
25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50
0x0	0x0	0x0	0x0	0x0	0x00	0x02	0x33	0x00	0x1F	0x00	Sec_3	Sec_2	Sec_1	Sec_0	$nSec_3$	$nSec_2$	$nSec_1$	$nSec_0$	0x00	0x01	0x0A	$fEng_3$	$fEng_2$	$fEng_1$	$fEng_0$

Figure 3.31: Publication for normal data.

PCC from the publication key decoded earlier in the message. Since this is normal data, the PCC is set to PCC_NORMAL_DATA (0x0). The next argument alerts the Intelligent Sensor as to what kind of data to expect. The physical parameter (TypeCode PHYSICAL_PARAMETER_DATA_TC) can be obtained from Table 3.2, although for this application PP_SCALAR_ANALOG (0x0) is used as the data represents analog measurements of scalar data that are not dimensioned, but contains units. The arguments for this physical parameter include a timestamp and a data value.

Table 3.2: Types of physical parametric data supported by IEEE 1451.1.

Enumeration	Value
PP_SCALAR_ANALOG	0
PP_SCALAR_DISCRETE	1
PP_SCALAR_DIGITAL	2
PP_SCALAR_ANALOG_SERIES	3
PP_SCALAR_DISCRETE_SERIES	4
PP_SCALAR_DIGITAL_SERIES	5
PP_VECTOR_ANALOG	6
PP_VECTOR_DISCRETE	7
PP_VECTOR_DIGITAL	8
PP_VECTOR_ANALOG_SERIES	9
PP_VECTOR_DISCRETE_SERIES	10
PP_VECTOR_DIGITAL_SERIES	11
Reserved Values	12-127
Open to Industry	128-255

For the temperature sensor, the units will be degrees Celsius, and the measurements are non-integer. On the other hand, SCALAR_DISCRETE is intended for physical quantities that are scalar, may or may not contain units, and is represented by an *n*-length integer. An example is a counter. Similar to the discrete case is SCALAR_DIGITAL, where the physical quantity contains no dimensions or units, and the values are not interpreted as integers or real numbers. Examples are binary sensors (proximity switch) or a coding (barcode). Each of these cases is again available with the suffix _SERIES, which essentially means that the data represented within consists of a series of values at that particular timestamp, that are computed at uniform intervals with respect to a physical quantity. Examples of series include frequency response, mass spectrum, and frequency spectrum. Each of the cases thus far presented are again repeated with the prefix PP_VECTOR_ in the place of PP_SCALAR_. Vector quantities include velocity, acceleration, electromagnetics, and are expected to be used as single dimensional vectors in the case of a single axis (for example, one axis accelerometer).

The next eight bytes to follow is the UTC timestamp, which consists of [s] and [ns] since the UTC epoch. Time is represented internally as a structure *TimeRepresentation* (refer to the

structural mapping of the message in Fig. 3.32) that contains both the second and nanosecond fields.

Publisher Port Message Structure:

Name	Type	Details
Message Type	uint8	0xF7
Total Message Length	uint16	0x0033
Length of Header	uint16	0x001E
Publication Key	unint8	PSK_PHYSICAL_PARAMETRIC_DATA (0x0B)
Publication Domain	uint8[8]	0xFFFFFFFFFFFFFFFF
Publication Topic	uint8[16]	Sensor Channel ID (SCID) - always 16 bytes, zero padded
# Args in Pub Contents	uint16	0x0002
1st Argument Type Code	uint8	UINTEGER8_TC (0x33)
1st Argument:	uint8	PCC_NORMAL_DATA (0x0)
2nd Argument Type Code	uint8	PHYSICAL_PARAMETER_DATA_TC (0x1F)
2nd Argument:	uint8	PP_SCALAR_ANALOG (0x0)
SingletonData.TimeStamp.Sec	uint32	Measurement UTC Timestamp [SI Seconds]
SingletonData.TimeStamp.nSec	uint32	Measurement UTC Timestamp [SI nanoseconds]
# Args in Argument Array	uint16	0x0001
Argument Type Code	uint8	0x0A (FLOAT32_TC)
SingletonData.Value	float32	Measured Temperature in Degrees Celsius

Figure 3.32: Message structure for normal data publication.

The remaining bytes in the message describe the argument array that contains the measurement values. Byte 44/45 of the message indicates the length of the argument array. Byte 46 indicates the appropriate way to interpret each element of the argument array. Together, these two pieces of information provide enough information to decode the elements of the argument array. In the case of normal data, there is only one actual argument encoded in the argument array, interpreted as FLOAT32_TC, though a series representation may be preferable to publish multiple measurements, achieving higher sampling rates with minimal increases in network bandwidth overhead.

Typical metadata includes measurement type, units, limits, uncertainty, etc. Metadata can be stand alone or grouped with normal data. Metadata publishing capability is not supported in this early version of the Intelligent Sensor in favor of incorporating the more useful benefits of health reporting. To minimize network traffic, there is no standalone health reporting option, but

instead health reporting is appended to normal data. Thus, the solution is a message designated PCC_NORMAL_DATA_AND_HEALTH, assigned user-defined PCC 0x80, and user-defined publication key PSK_USER_PHYSICAL_PARAMETRIC_DATA_AND_HEALTH assigned PSK 0x83. The general on-the-wire message format is shown in Fig. 3.33.

Publisher Port Message Format:

Type	Msg Len		Header Len		Pub Key	Publication Domain								Publication Topic										
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
0xF7	*L*		0x00	0x1E	0x83	O_8	O_1	O_2	O_3	O_4	O_5	O_6	O_7	$ChID_{10}$	$ChID_9$	$ChID_8$	$ChID_7$	$ChID_6$	$ChID_5$	$ChID_4$	$ChID_3$	$ChID_2$	$ChID_1$	$ChID_0$

Publication Topic					Number of Args in Pub Contents		Arg.1 TC	Arg.1 PCC	Arg.2 TC	Arg.2 PhyParam	SingletonData.TimeStamp. Sec (UTC seconds)				SingletonData.TimeStamp. NanoSec (UTC nanoseconds)				Num Args in Arg. Array "Value"		Arg.1 TC	SingletonData.Value IEEE 754 32-bit floating point			
25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50
0x0	0x0	0x0	0x0	0x0	0x00	0x02	0x33	0x80	0x1F	0x0	Sec_3	Sec_2	Sec_1	Sec_0	$nSec_3$	$nSec_2$	$nSec_1$	$nSec_0$	*m*		0x0A	$fEng_3$	$fEng_2$	$fEng_1$	$fEng_0$

...	Arg.2 TC	Number of Args in Arg Array	Arg.2.1 TC	Health Algorithm Classification (HAC)				Arg.2.2 TC	SingletonData.TimeStamp. Sec (UTC seconds)				SingletonData.TimeStamp. NanoSec (UTC nanoseconds)				Arg.2.3 TC	Number of Bytes in Arg. 2.3		Arg 2.3: Information forming "health info." May contain raw data values, health assessment features or computed results from routines.				
	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68	69	70	71	72	73	74	75
	0x01	0x00	0x03	0x2F	HAC_3	HAC_2	HAC_1	HAC_0	0x2B	Sec_3	Sec_2	Sec_1	Sec_0	$nSec_3$	$nSec_2$	$nSec_1$	$nSec_0$	*k*	*n*		Inf_n	Inf_{n-1}	Inf_{n-3}	...	Inf_0

Figure 3.33: Publication message format for normal data and health.

The message structure is designed to publish a health message for each routine running in ERM that utilizes the periodic health reporting feature via *ERM_HealthReport(.)*. As such, for every routine that publishes periodic health information, there are 18 bytes added to the total message size for identifying the routine with its HAC (bytes 55 thru 58) and providing the timestamp of the health information (bytes 60 thru 67). Then, the length of the health information depends on the associated TypeCode and the number of elements if the health information is an array format. Thus, the size of the message becomes variable, and is a function of the number of reporting routines, the reporting frequency of each routine, and the specific argument type of each routine.

The next publication message is for health routines that generate alerts or notifications whenever an event or condition is met. This type of message is typically associated with event detection routines, and is used to alert ISHM of a significant event that is taking place or has

ceased. Due to the asynchronous behavior and nature of this message, it is not grouped with health information and normal data, but is instead processed as a separate message. The message contains the standard publication header, with a user defined PSK of 0x82 for PSK_USER_HEALTH_EVENT, as shown in Fig.3.34. This message contains either three or four arguments in the publication contents (Fig 3.34 shows all four).

Publisher Port Message Format:

Type	Msg Len		Header Len		Pub Key	Publication Domain								Publication Topic										
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
0xF7	0x00	0x28	0x00	0x0F	0x82	O_0	O_1	O_2	O_3	O_4	O_5	O_6	O_7	$ChID_{10}$	$ChID_9$	$ChID_8$	$ChID_7$	$ChID_6$	$ChID_5$	$ChID_4$	$ChID_3$	$ChID_2$	$ChID_1$	$ChID_0$

... Publication Topic					# Args. in Pub Contents	Argument #1 TC	Argument #1: Health Algorithm Classification (unsigned 32 bit integer)				Argument #2 TC	ProfileEffectiveTime. Sec (UTC seconds)				ProfileEffectiveTime. NanoSec (UTC nanoseconds)				Argument #3 TC	Number of Bytes in Arg. #3 (metadata) ...	
25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48
0x0	0x0	0x0	0x0	0x0	0x00	0x04	0x2F	HAC_3	HAC_2	HAC_1	HAC_0	0x2B	Sec_3	Sec_2	Sec_1	Sec_0	$nSec_3$	$nSec_2$	$nSec_1$	$nSec_0$	0x1A	n	

... Argument #3: MetaData associated with health event. ASCII character array transmitted as octets. Describes health event and status.					Argument #4 TC	Number of Bytes in Arg. #4		Arg #4: Info. forming basis of the health event. Contains raw data values, health routine results, or other supporting values as deemed necessary.				
49	50	51	...	49+n	50+n	51+n	52+n	53+n	54+n	55+n	...	53+n+m
Ary_n	Ary_{n-1}	Ary_{n-3}	...	Ary_0	K	m		Inf_m	Inf_{m-1}	Inf_{m-3}	...	Inf_0

Figure 3.34: Health Alert publication for event routines.

The first three arguments are mandatory, and consist of the reporting algorithm's HAC, the timestamp for when the event begins, and the event metadata. The metadata is an array of octets, interpreted as ASCII that verbosely describes the event that is taking place. The metadata is provided to the Intelligent Sensor through HEDS, and event detection routines customarily contain a metadata record for when an event is first detected and when an event expires, while the routine is constantly checking for either of those cases to transpire. While this literal messaging technique is useful, by changing the TypeCode of the metadata and updating the HEDS it is possible to convert from a verbose alert to a coded alert. Coded alerts (i.e. Error 0x42) may be more practical in an autonomous system, while the operator console would provide the mapping of the coded alerts to a literal description (i.e. "the transducer on Sensor PE1140GO has failed"). The coding structure for this message follows in Fig. 3.35. The fourth

and optional field is for any supporting justification of the event. Depending on the specific application, it is favorable to broadcast the indicators that were used in determining the status change of the event. HEDS includes the TypeCode and number of arguments (if applicable) when this field is utilized; otherwise the field is suppressed in the message.

Publisher Port Message Structure/Payload:

Name	Type	Details
Message Type	uint8	0xF7
Total Message Length	uint16	0x0029+*n*+*m*
Length of Header	uint16	0x000F
Publication Key	unint8	PSK_USER_HEALTH_EVENT (0x82)
Publication Domain	uint8[8]	0xFFFFFFFFFFFFFFFF
Publication Topic	uint8[16]	Sensor Channel ID (SCID) - always 16 bytes, zero padded
# Args in Pub Contents	uint16	0x0004 or 0x0003 depending on inclusion of supporting evidence
1st Argument TypeCode	uint8	0x2F (unsigned int32)
1st Argument	uint32	Health Algorithm Class (HAC) to identify algorithm family
2nd Argument TypeCode	uint8	0x2B (time representation)
# Args in Argument #2	uint32	Timestamp UTC seconds corresponding to event start
2nd Argument	uint32	Timestamp UTC nanoseconds corresponding to event start
3rd Argument TypeCode	uint8	0x1A (octet array, intepreted as a character array)
# Args in Argument #3	uint16	n - based on size of metadata
3rd Argument	uint8[n]	Metadata provided by health routine HEDS for event alerts
4th Argument TypeCode	uint8	Varies based on health routine; float, octet, integer, etc.
# Args in Argument #4	uint16	m - based on size of data value/dataset/etc (1 thru 65k)
4th Argument	uint8[m]	Information or "evidence" providing the basis of the health alert.

Figure 3.35: Message structure for health alert messages.

The last and final publication is the system state change message, which is issued before a system, subsystem, or process state change is anticipated. Since the HEDS for all supported states are already loaded, this packet simply indicates the state profile that the Intelligent Sensor must switch to and the UTC time when the change becomes effective. This message contains the typical publication header with user-defined PSK 0x83 as PSK_USER_FUTURE_ STATE_TRANSITION_NOTIFICATION.

Publisher Port Message Format:

Type	Msg Len		Header Len		Pub Key	Publication Domain								Pub. Topic	# Args. in Pub Contents		Argument #1 TC	...
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	
0xF7	0x00	0x24	0x00	0x0F	0x84	O_0	O_1	O_2	O_3	O_4	O_5	O_6	O_7	0x0	0x00	0x03	0x2F	

...	Argument #1: Health Algorithm Classification (unsigned 32 bit integer)				Argument #2 TC	ProfileEffectiveTime. Sec (UTC seconds)				ProfileEffectiveTime. NanoSec (UTC nanoseconds)				Argument #3 TC	System State Profile Identifier			
	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35
	HAC_3	HAC_2	HAC_1	HAC_0	0x2B	Sec_3	Sec_2	Sec_1	Sec_0	$nSec_3$	$nSec_2$	$nSec_1$	$nSec_0$	0x1A	SS_3	SS_2	SS_1	SS_0

Figure 3.36: Future State Profile Transition Message

Publisher Port Message Structure/Payload:

Name	Type	Details
Message Type	uint8	0xF7
Total Message Length	uint16	0x0024
Length of Header	uint16	0x000F
Publication Key	unint8	PSK_USER_STATE_PROFILE_FUTURE_TRANSITION_NOTIFICATION (0x84)
Publication Domain	uint8[8]	0xFFFFFFFFFFFFFFFF
Publication Topic	uint8[16]	Sensor Channel ID (SCID) - always 16 bytes, zero padded
# Args in Pub Contents	uint16	0x0004 or 0x0003 depending on inclusion of supporting evidence
1st Argument TypeCode	uint8	0x2F (unsigned int32)
1st Argument	uint32	Health Algorithm Class (HAC) to identify algorithm
2nd Argument TypeCode	uint8	0x2B (time representation)
2nd Argument (s)	uint32	Timestamp UTC seconds corresponding to state profile effective time
2nd Argument (ns)	uint32	Timestamp UTC nanoseconds corresponding to state profile eff. time
3rd Argument TypeCode	uint8	0x2F (unsigned int32)
3rd Argument	uint32	State profile identifier that shall be set active

Figure 3.37: Structure of the future state change message.

This means that the state transition map may be downloaded in entirety before starting, although unexpected changes can cause it to need updates as system events occur. For example, going back to the previous example (Fig 3.11), the system expects to transition between Idle, Pretest, and Test. However, due to an ignition problem, the system unexpectedly transitions to abort. The decision to abort may be a result of exceeding the timeout for ignition or the confirmation of a critical fault condition reported by an Intelligent Sensor. Subsequently, purge valves will activate as part of the Abort procedure, and if the downloaded transition maps for the affected Intelligent Sensors are not updated, the sensors will report uncommanded activity on the purge valves. To address this issue, a message with publication key PSK_USER_IMMEDIATE_

STATE_TRANSITION_NOTIFICATION (PSK 0x84) is published, indicating that all stored transitions that are to occur after the provided timestamp are invalid and the transition to the

Publisher Port Message Format:

Type	Msg Len		Header Len		Pub Key	Publication Domain								Pub. Topic	# Args. in Pub Contents		Argument #1 TC	...
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	
0xF7	0x00	0x24	0x00	0x0F	0x85	O_0	O_1	O_2	O_3	O_4	O_5	O_6	O_7	0x0	0x00	0x03	0x2F	

...	Argument #1: Health Algorithm Classification (unsigned 32 bit integer)				Argument #2 TC	ProfileInvalidTime. Sec (UTC seconds)				ProfileInvalidTime. NanoSec (UTC nanoseconds)				Argument #3 TC	System State Profile Identifier			
	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35
	HAC_3	HAC_2	HAC_1	HAC_0	0x2B	Sec_3	Sec_2	Sec_1	Sec_0	$nSec_3$	$nSec_2$	$nSec_1$	$nSec_0$	0x1A	SS_3	SS_2	SS_1	SS_0

Figure 3.38: Message for commanding an immediate state change

new profile takes effect immediately. Keep in mind when reviewing this example that the behavior of the Intelligent Sensor depends on the level of capability afforded to the health algorithms; they may be designed only to verify that the signal is reasonable (i.e. less than some dB of noise) or to track specific activity (i.e. valve feedback signal should indicate 50% ±5% open at all times during this state, but the next state should show decreasing values at a rate of $7.5\frac{\%}{ms} \pm 1.0\frac{\%}{ms}$ until fully closed. The state transition message is shown in Fig. 3.38 with the mapping structure in Fig. 3.39.

Publisher Port Message Structure/Payload:

Name	Type	Details
Message Type	uint8	0xF7
Total Message Length	uint16	0x0024
Length of Header	uint16	0x000F
Publication Key	unint8	PSK_USER_STATE_PROFILE_IMMEDIATE_TRANSITION_NOTIFICATION (0x85)
Publication Domain	uint8[8]	0xFFFFFFFFFFFFFFFF
Publication Topic	uint8[16]	Sensor Channel ID (SCID) - always 16 bytes, zero padded
# Args in Pub Contents	uint16	0x0004 or 0x0003 depending on inclusion of supporting evidence
1st Argument TypeCode	uint8	0x2F (unsigned int32)
1st Argument	uint32	Health Algorithm Class (HAC) to identify algorithm
2nd Argument TypeCode	uint8	0x2B (time representation)
2nd Argument (s)	uint32	Timestamp UTC seconds corresponding to last valid state
2nd Argument (ns)	uint32	Timestamp UTC nanoseconds corresponding to last valid state
3rd Argument TypeCode	uint8	0x2F (unsigned int32)
3rd Argument	uint32	State profile identifier that shall be set active

Figure 3.39: Immediate state transition structure.

3.3.2 IEEE 1451.1 Client Server Communications

This section covers peer-to-peer messaging over the Smart Sensor network. The peer-to-peer communication is modeled as the ubiquitous client-server structure, where the definition of *client* is reduced to the individual object requesting data or service, and a *server* is the object that provides fulfillment of the request. This means that there are no statically defined client and server objects; the role depends on the originator of the conversation. The server is stateless and does not retain client information to participate in a conversation; each client to server message is handled individually. Client-server communications requires the client to be aware of how to send messages to the server's destination. For this to be possible, the Smart Sensor's NCAP must already be in the BL_ACTIVE state and registered on the network using the announcement publications examined in the previous section. The interface documentation for Open1451 specifies that each client-server interaction should occur over a *client port* that is unique [33]. While the intention of Open1451 is to standardize these ports with IEEE to promote homogeneity between implementations, at the time of publication no blocks of IP ports have been assigned or designated. As a result, in this version all client-server communications take place over port 11000, with a path for upgrading to message-specific port assignments flagged as a future goal. The list of supported client/server messages in the Intelligent Sensor are:

1. *GoActive*
2. *GoInactive*
3. *GetBlockMajorState*
4. *GetTEDS*
5. *SetTEDS*
6. *GetHEDS*
7. *SetHEDS*

Because of the point-to-point nature of these messages, the message structure is drastically different from the previously developed broadcast publications. Each client to server message contains a Message ID, Message Length, Client's Block Cookie, Server's Object ID, Operation ID, and an Execution Mode. The encoding of any applicable arguments at the end of this header segment is the same as for publications. Once the message has been deciphered and processed by the server, a reply is always generated and returned to the client. The server to client message elements are Message ID, Message Length, Return Code, Server's Block Cookie, Execution Mode, Server's Object ID, and Operation ID. Any return arguments are encoded and appended to the message header. The basic client to server and server to client message headers are shown in Fig. 3.40, followed by an explanation of the individual fields.

Client Port Message Format:

Type	Msg Len		Block Cookie (Client Port)		Object ID (Server)									Operation ID		Ex. Mode
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
0xED	0x00	0x11	0x00	0x00	0x03	MAC_5	MAC_4	MAC_3	MAC_2	MAC_1	MAC_0	OID_1	OID_0	$OPID_1$	$OPID_0$	EM

Server Port Message Format:

Type	Msg Len		Client/Server Return Code				Block Cookie (Server Port)		Ex. Mode	Object ID (Server)									Operation ID	
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
0xD5	0x00	0x15	$CSRC_3$	$CSRC_2$	$CSRC_1$	$CSRC_0$	BC_1	BC_0	EM	0x03	MAC_5	MAC_4	MAC_3	MAC_2	MAC_1	MAC_0	OID_1	OID_0	$OPID_1$	$OPID_0$

Figure 3.40: Example of client/server and server return messages.

The Block Cookie is a parameter maintained by the server object that will change should the context of that server object change. The server's Block Cookie is cached by the client and used by the server to verify that the client is aware of any (potentially critical) changes to the server object before processing the message. A failure to match the Block Cookie results in the abort of the client request and generates a reply message to the client with the appropriate Return Code. The Block Cookie defaults to zero, and increments for each and any change in the server's context.

The Object ID is used to identify the specific object that is the recipient of the message. The first byte identifies the method of Object ID generation, as it is a network and 1451 specific parameter. Since we are using Ethernet, we create a custom Object ID using Ethernet DCE (i.e. 0x03) consisting of the Smart Sensor's MAC address (uniquely identifying the sensor) and an individual object identifier (unique between every object on a given sensor) for each object, as discussed in the NCAP announcement reply message for the specific instance of the NCAP Object ID in Section 3.3.1.

The Operation ID specifies the operation that is intended to be carried out by the message. IEEE 1451.1 offers enumeration guidelines for assigning Operation IDs, and is based on the hierarchy of the specific operation with respect to the Root Abstract Class and whether the method is part of IEEE 1451-mandated functionality or part of industry/application specific functionality. Please see Table 3.3 for this table of guidelines, borrowed from IEEE 1451.1. This table ensures that all operations defined in 1451.1 have a unique Operation ID, and that

Table 3.3: Table of suggested IEEE 1451 Operation ID assignment.

Level	Values reserved for IEEE 1451 Standard	Values available to industry groups and application developers
0 - Root Class	0-2047	none
1	2048-3071	3072-4095
2	4096-5119	5120-6143
3	6144-7167	7168-8191
4	8192-9215	9216-10239
5	10240-11263	11264-12287
6	12288-13311	13312-14335
7	14336-15359	15360-16383
8	16384-17407	17408-18431
9	18432-19455	19456-20479
10	20480-21503	21504-22527
11	22528-23551	23552-24575
12	24576-25599	25600-26623
12 < N < 32	2048N through 2048N+1023	2048N+1024 through 2048N+2047

there is plenty of room to accommodate application/vendor specific functionality. The specific Operation ID for each client-server message will be introduced in the section that discusses the message.

The Client/Server Return Code is generated after the server has finished processing (or attempting to process) a particular client-server message. The Return Code is an unsigned 32 bit integer interpreted as four 1-byte fields. Those fields are listed in Table 3.4, and represent the return value for the Server Object, method Perform, and method Port, which are utilized in processing the client's command.

Table 3.4: Client/Server Return Code description.

Field Name	Bits in Client-Server Return Code	Information
portCode	High-order 8 bits intepreted as an UInteger8	The return code of the client-side Port Object *Execute* or *GetResult* operation. Values shall be selected from the MajorReturnCode enumeration.
performCode	Netxt 8 bits intepreted as an UInteger8	The return code of the server-side Object *Perform* operation. Values shall be selected from the MajorReturnCode enumeration.
operationMinorCode	Netxt 8 bits intepreted as an UInteger8	The Minor Field of the OpReturnCode of the operation invoked on the Server Object.
operationMajorCode	Lower-order 8 bits inter-preted as an UInteger8	The Major Field of the OpReturnCode of the operation invoked on the Server Object.

The Return Code enumeration specified in Table 3.5 indicates the acceptable values for all four fields of the client/server return code identified in Table 3.4. Note that while each field is constrained to the same enumeration, some individual enumerations are not applicable due to scope. For example, communications based return codes are not applicable to the execution of the operation (identified by operationMinorCode and operationMajorCode) and execution return codes may not be applicable to the communications subsystem as identified by portCode and performCode. Refer to IEEE 1451.1 (pp 58-62) for each specific case.

Table 3.5: Return Code enumerations for client-server return codes.

Client-Server Return Codes:		
Perform Code Name	***Perform Value***	***Perform Code Meaning***
MJ_COMPLETE	0x0	No errors detected during perform operation
MJ_NOP_OPERATION	0x01	Interface Only Implementation
MJ_NO_RETURN_CODE	0x02	No return code available
MJ_FAILED_NON_SPECIFIC	0X03	Nonspecific failure
MJ_COMMUNICATION_ERROR	0x04	No response from any obj at dest. ODA
MJ_BUSY	0x05	Server busy & does not support multiple requests
MJ_SERVICE_UNAVAILABLE	0x06	Perform not avail. due to server internal condition
MJ_ILLEGAL_OPERATION	0x07	Requested Op. ID is not valid on Server
MJ_FAILED_INPUT_ARGUMENT	0x08	One or more missing/invalid input arguments
MJ_FAILED_OUTPUT_ARGUMENT	0x09	One or more missing/invalid output arguments
MJ_FAILED_MARSHALING	0X0A	Unable to translate arguments to wire format
MJ_FAILED_DEMARSHALING	0x0B	Unable to translate arguments to parameter
MJ_ILLEGAL_TRANSACTION	0x0C	There is no such transaction as indicated in the call
MJ_OPERATION_INTERRUPTED	0x0D	Perform was interrupted and did not complete
MJ_OPERATION_TIMEOUT	0x0E	Op. did not complete before timeout on target object
MJ_INSUFFICIENT_RESOURCES	0X0F	Server object lacks sufficient resources
MJ_BLOCK_COOKIE_MISMATCH	0x10	Block Cooke mismatch detected
MJ_TRANSDUCER_ERROR	0x11	Transducer failed during operation
MJ_SUSPECT_INVOCATION_RESULT	0x12	Target value may be erronious (overload, overflow)
MJ_NOT_PROPERLY_CONFIGURED	0X13	Object detected some state not configured properly
Reserved	0x14 - 0x80	Reserved for IEEE 1451.1
User-Defined	0x81 - 0xFF	Industry/Application Specific

The execution mode parameter indicates the contents of the return message and behavior of the client in the context of client-server messaging. Valid execution modes are provided in Table 3.6. The presence of a return value (0x0) in the client port message indicates that the client will block waiting for a response from the server until a timeout occurs as the return message contains arguments required by the client. If the client times out before receiving a response, any received response is invalidated because the client block cookie is not updated, and the client can choose to attempt another request.

Table 3.6: Valid execution modes.

Enumeration	Value
EM_RETURN_VALUE	0
EM_NO_RETURN_VALUE	1
Reserved	2-255

If no return value is to be provided (0x01), the client continues processing immediately once the client port message has been sent. The server generates a response message consisting of only a header, as any arguments are unexpected and discarded. In summary, the Execution Mode

indicates the presence of arguments in the return message and whether the client waits for the response before continuing or sends the message and continues immediately. Now that the basic client/server messaging structure has been examined, we will use this base to construct the complete messages for peer-to-peer communications.

We begin by developing messages for the *GoActive* message. This is used to place individual blocks into the BL_ACTIVE state. Practical uses include starting/stopping individual health analysis routines, transducer blocks, or the NCAP block. The message structure, provided in Fig. 3.41, can be thought of as a function call in Fig. 3.42.

Client Port Message Format:

Type	Msg Len		Block Cookie (Client Port)		Object ID (Server)									Operation ID		Ex. Mode
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
0xED	0x00	0x11	BC_1	BC_0	0x03	MAC_5	MAC_4	MAC_3	MAC_2	MAC_1	MAC_0	OID_1	OID_0	0x10	0x06	0x01

Server Port Message Format:

Type	Msg Len		Client/Server Return Code				Block Cookie (Server Port)		Ex. Mode	Object ID (Server)									Operation ID	
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
0xD5	0x00	0x15	$CSRC_3$	$CSRC_2$	$CSRC_1$	$CSRC_0$	BC_1	BC_0	0X01	0x03	MAC_5	MAC_4	MAC_3	MAC_2	MAC_1	MAC_0	OID_1	OID_0	0x10	0x06

Figure 3.41: Interface for GoActive client-server communication.

```
[return_code, out_argument_array]=GoActive(exec.mode, ServerObjectID, in_argument_array);
```

Figure 3.42: Remote Procedure Call invoked by GoActive message and reply.

The client port message provides the Object ID of the server object to invoke an operation on, and the execution mode indicates the client behavior while execution occurs. The response from the server is the status or client-server return code (see Table 3.4) and any applicable arguments. For this example there are no arguments in either the client-server or server-client messages. The Operation ID of 0x1006 corresponds to the "handle" of the *GoActive* command.

The *GoInactive* command transitions the object referenced by the Object ID to the BL_INACTIVE state. It is identical to the *GoActive* message except for the Operation ID, which

in this case is 0x1007. Please see the corresponding client-server and return message in Fig. 3.43.

Client Port Message Format:

Type	Msg Len		Block Cookie (Client Port)		Object ID (Server)									Operation ID		Ex. Mode
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
0xED	0x00	0x11	BC_1	BC_0	0x03	MAC_5	MAC_4	MAC_3	MAC_2	MAC_1	MAC_0	OID_1	OID_0	0x10	0x07	0x01

Server Port Message Format:

Type	Msg Len		Client/Server Return Code				Block Cookie (Server Port)		Ex. Mode	Object ID (Server)									Operation ID	
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
0xD5	0x00	0x15	$CSRC_3$	$CSRC_2$	$CSRC_1$	$CSRC_0$	BC_1	BC_0	0X01	0x03	MAC_5	MAC_4	MAC_3	MAC_2	MAC_1	MAC_0	OID_1	OID_0	0x10	0x07

Figure 3.43: Interface message for GoInactive client-server communication.

The next client-server message of interest is *GetBlockMajorState*. This method allows the caller to retrieve the operational status of the object identified by the Object ID parameter. The valid selections for operating state are BL_ACTIVE, BL_INACTIVE, and BL_UNINITIALIZED. Since the client issuing this message expects return arguments, the execution mode is set to zero (see enumeration in Fig. 3.45) and the return message contains a single value argument interpreted as an unsigned 8-bit integer (0x33 TC). For completeness the message structure is shown in Fig. 3.44, with the mapping structure for the return message in Fig. 3.45.

Client Port Message Format:

Type	Msg Len		Block Cookie (Client Port)		Object ID (Server)									Operation ID		Ex. Mode
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
0xED	0x0	0x11	BC_1	BC_0	0x03	MAC_5	MAC_4	MAC_3	MAC_2	MAC_1	MAC_0	OID_1	OID_0	0x10	0x02	0x0

Server Port Message Format:

Type	Msg Len		Client/Server Return Code				Block Cookie (Server Port)		Ex. Mode	Object ID (Server)									Operation ID		Argument Count		Argument #1 TC	Blk. Major State
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
0xD5	0x00	0x18	$CSRC_3$	$CSRC_2$	$CSRC_1$	$CSRC_0$	BC_1	BC_0	0x0	0x03	MAC_5	MAC_4	MAC_3	MAC_2	MAC_1	MAC_0	OID_1	OID_0	0x10	0x02	0x0	0x01	0x33	BL

Figure 3.44: GetBlockMajorState message with return message and arguments.

Server Port Message Structure:

Name	Type	Details
Message Type	uint8	0xD5
Total Message Length	uint16	0x0018
Client/Server Return Code	uint32	See Return Code Table for enumerations
Block Cookie (Server Port)	uint16	BC (Set initially, changes when server context changes)
Execution Mode	uint8	EM_RETURN_VALUE (0x0)
ObjectID	uint8[8]	ObjectID referenced in original message (for verification on client)
Operation ID	uint16	OP_GET_BLOCK_MAJOR_STATE (0x1002)
Argument Count	uint16	0x0001 (Number of arguments in the contents of the message)
1st Argument TypeCode	uint8	UINTEGER8_TC (0x33)
1st Argument	uint8	BL {BL_UNINITIALIZED, BL_INACTIVE, BL_ACTIVE}

Figure 3.45: Message structure for GetBlockMajorState return message.

The next level of functionality that is required is the ability to read and write TEDS to the Intelligent Sensor. Typically TEDS are limited to the domain of the local NCAP and are paired with the transducers to provide self identification and configuration capability. Since the Intelligent Sensor is an integrated TIM/NCAP multi-purpose solution, there is no discrete TIM containing transducer specific TEDS. Therefore, when a transducer change is made, it is not possible to swap the actual TEDS records at the same time. The solution is to provide an interface to Intelligent Sensor TEDS memory through client/server messaging. Since TEDS are interpreted by an IEEE sanctioned template, the message contents consist of a field for the TEDS Template ID and an octet array field for the corresponding TEDS data, as shown in Fig. 3.46 / Fig. 3.47. Multiple sets of TEDS may be transmitted simultaneously by adding another TEDS Template ID/TEDS data pair of arguments at the end of the first and incrementing the message argument count by two. Basic TEDS may be loaded by indicating a NULL TEDS Template ID, as Basic TEDS are not assigned an ID. According to 1451.4, the Intelligent Sensor is Tier 1 Compliant, meaning that there is support for Basic TEDS, although other more advanced templates may be transmitted, though not actually used at this point.

Client Port Message Format:

Type	Msg Len		Block Cookie (Client Port)		Object ID (Server)									Operation ID	
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0xED	*L*		BC_1	BC_0	0x03	MAC_5	MAC_4	MAC_3	MAC_2	MAC_1	MAC_0	OID_1	OID_0	0x2C	0x01

...	Ex. Mode	# Arguments in Message Contents		Arg. TC 1,3,5...m-1	Argument #1,3,5...m-1: Template ID		Arg. TC 2,4,6...m	Argument #2,4,6...m count		Argument #2,4,6...m: TEDS Data				...
	16	17	18	19	20	21	22	23	24	25	26	...	24+*n*	
	0x01	2*m		0x2D	TDL_1	TDL_0	0x1D	*n*		TD_0	TD_1	...	TD_n	

Figure 3.46: Member function for uploading TEDS to a Smart Sensor.

Client Port Message Structure:

Name	Type	Details
Message Type	uint8	0xED
Total Message Length	uint16	$L = 0x19+\Sigma n_m+0x6*m$
Client Block Cookie	uint16	$\|BC_1\|BC_0\|$
Object ID	uint8[9]	Identifies object on server to receive this message
Operation ID	uint16	OP_SET_TEDS (0x2C01)
Execution Mode	uint8	EM_NO_RETURN_VALUE (0x01)
# Arguments in Msg Contents	uint16	2*m, where m is the number of TEDS templates to load
Argument #1 TypeCode	uint8	UINTEGER16_TC (0x2D)
Argument #1	uint16	TEDS Template ID
Argument #2 TypeCode	uint8	OCTET_ARRAY_TC (0x1D)
Argument #2 Count	uint16	n, where n is the number of octets in the TEDS data
Argument #2	uint8[n]	TEDS Data
...	...	...

Figure 3.47: Message Mapping for the SET_TEDS operation.

The Smart Sensor will prohibit assigning TEDS to any object (as indicated with the Object ID) that is not a member of the Transducer Block class. The *SET_TEDS* return message indicates if the upload was successful by utilizing the client-server return code in the return message structure shown in Fig. 3.48.

Server Port Message Format:

Type	Msg Len		Client/Server Return Code				Block Cookie (Server Port)		Ex. Mode	Object ID (Server)									Operation ID	
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
0xD5	0x00	0x15	$CSRC_3$	$CSRC_2$	$CSRC_1$	$CSRC_0$	BC_1	BC_0	0x01	0x03	MAC_5	MAC_4	MAC_3	MAC_2	MAC_1	MAC_0	OID_1	OID_0	0x2C	0x01

Figure 3.48: Reply message after TEDS have been uploaded.

The next message, *GET_TEDS*, performs the opposite operation of *SET_TEDS* and is used when an outside entity wishes to acquire the TEDS that are currently loaded in the Intelligent Sensor. The Object ID is used to indicate the Transducer Block from which to obtain the TEDS. Thus, the TEDS for each Transducer Block must be individually requested. Since the Intelligent Sensor is 1451.4 Tier 1 compliant, there is no need to handle the case of a multinode transducer, so it is guaranteed that each Transducer Block will only have a single transducer connected to it. The message structure for initiating the TEDS retrieval is shown in Fig. 3.49.

Client Port Message Format:

Type	Msg Len		Block Cookie (Client Port)		Object ID (Server)									Operation ID		Ex. Mode
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
0xED	0x0	0x11	BC_1	BC_0	0x03	MAC_5	MAC_4	MAC_3	MAC_2	MAC_1	MAC_0	OID_1	OID_0	0x2C	0x00	0x0

Figure 3.49: Message requesting TEDS from an Intelligent Sensor.

The return message provides the Intelligent Sensor's TEDS encoded in an argument array as shown in Fig. 3.50. There will be an element in the argument array for each TEDS Template ID/TEDS data pair. The TEDS Template is encoded as an unsigned integer, and the TEDS data is interpreted as an octet array. The execution mode in this pair of messages is set to EM_RETURN_VALUE, as the server is returning arguments.

Server Port Message Format:

Type	Msg Len		Client/Server Return Code				Block Cookie (Server Port)		Ex. Mode	Object ID (Server)								
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
0xD5	*L*		$CSRC_3$	$CSRC_2$	$CSRC_1$	$CSRC_0$	BC_1	BC_0	0x01	0x03	MAC_5	MAC_4	MAC_3	MAC_2	MAC_1	MAC_0	OID_1	OID_0

...	Operation ID		# Arguments in Message Contents		Arg. TC 1,3,5...m-1	Argument #1,3,5...m-1: Template ID		Arg. TC 2,4,6...m	Argument #2,4,6...m count		Argument #2,4,6...m: TEDS Data				...
	19	20	21	22	23	24	25	26	27	28	29	30	...	28+*n*	
	0x2C	0x00	2**m*		0x2D	TDL_1	TDL_0	0x1D	*n*		TD_0	TD_1	...	TD_n	

Figure 3.50: Reply to GET_TEDS request.

Server Port Message Structure:

Name	Type	Details
Message Type	uint8	0xD5
Total Message Length	uint16	$L = 0x1D+\Sigma n_m+0x6*m$
Client/Server Return Code	uint32	See Return Code Table for enumerations
Block Cookie (Server Port)	uint16	$\|BC_1\|BC_0\|$
Execution Mode	uint8	EM_RETURN_VALUE (0x0)
ObjectID	uint8[9]	ObjectID referenced in original message (for verification on client)
Operation ID	uint16	OP_GET_TEDS (0x2C00)
# Arguments in Msg Contents	uint16	2*m, where m is the number of TEDS templates to load
Argument #1 TypeCode	uint8	UINTEGER16_TC (0x2D)
Argument #1	uint16	TEDS Template ID
Argument #2 TypeCode	uint8	OCTET_ARRAY_TC (0x1D)
Argument #2 Count	uint16	n, where n is the number of octets in the TEDS data
Argument #2	uint8[n]	TEDS Data
...	...	...

Figure 3.51: Structure for GET_TEDS reply message.

The next client-server message is for the upload and download of HEDS. Recall from the previous section on HEDS that each routine contains a basic component and a routine-specific component. Also remember HEDS are linked to routines by matching the HEDS ID with the routine's Object ID to ensure there is never an accidental use of HEDS designed for another application/configuration, while the HAC is a functional descriptor. Thus, each routine obtains its HEDS through individual *SET_HEDS* messages targeted for the specific routine objects. The *SET_HEDS* message and structure is shown in Fig. 3.52 / Fig. 3.53.

Client Port Message Format:

Type	Msg Len		Block Cookie (Client Port)		Object ID (Server)									Operation ID		Ex. Mode	# Args in Msg Contents		Arg. #1 TC	Number of Elements in HEDS Chksm	Arg. #1: Checksums for the HEDS Data				
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
0xED	*L*		BC_1	BC_0	0x03	MAC_5	MAC_4	MAC_3	MAC_2	MAC_1	MAC_0	OID_1	OID_0	0x2C	0x03	0x01	0x00	0x03	0x30	*m*		CS_3	CS_2	CS_1	CS_0

Arg. #2 TC	Num Octets in BASIC HEDS		Basic HEDS				Arg. #3 TC	Num Args in Arg. Array "HEDS Data"		Arg. Array Arg.#2 TC	Num of elements in System State		System State #1				Arg. Array Arg.#3 TC	Number of octets in HEDS Record		HEDS Record octets for the ObjectID / System State Combination		
22+4m	23+4m	24+4m	25+4m	26+4m			35+4m	36+4m	37+4m	38+4m	39+4m	40+4m	41+4m	42+4m	43+4m	44+4m	45+4m	42+4m+4n	43+4m+4n	44+4m+4n	45+4m+4n		44+4m+4n+p
0x1D	0x0	0x0C	HB_0	HB_1			HB_{10}	0x01	0x0	0x02	0x30	*n*		SS_3	SS_2	SS_1	SS_0	0x1D	*p*		HR_0	HR_1	HR_p

Figure 3.52: Message for transmitting HEDS to a routine running in a Smart Sensor.

Client Port Message Structure:

Name	Type	Details
Message Type	uint8	0xED
Total Message Length	uint16	L = 0x2D+(0x04*m)+(0x04*n)+p
Client Block Cookie	uint16	$\|BC_1\|BC_0\|$
Object ID	uint8[9]	Identifies object on server to receive this message
Operation ID	uint16	OP_SET_HEDS (0x2C03)
Execution Mode	uint8	EM_NO_RETURN_VALUE (0x01)
# Arguments in Msg Contents	uint16	0x03: {checksums} {basic HEDS} {HEDS data/state identifiers}
Argument #1 TypeCode	uint8	UINTEGER32__ARRAY_TC (0x30)
Argument #1 Count	uint16	m: number of checksum values (0+)
Argument #1	uint32[m]	Checksums for each memory page
Argument #2 TypeCode	uint8	OCTET_ARRAY_TC (0x1D)
Argument #2 Count	uint16	Length of the basic HEDS (0x0B)
Argument #2	uint8[11]	Basic HEDS describing common features
Argument #3 TypeCode	uint8	ARGUMENT_ARRAY_TC (0x01)
Argument #3 Count	uint16	0x02: {state identifier(s)} {HEDS data}
Argument #3.1 TypeCode	uint8	UINTEGER32__ARRAY_TC (0x30)
Argument #3.1 Count	uint16	n: number of states associated with this HEDS Data (1+)
Argument #3.1	uint32[n]	each state identifier
Argument #3.2 TypeCode	uint8	OCTET_ARRAY_TC (0x1D)
Argument #3.2 Count	uint16	p: number of octets comprising the HEDS data (0+)
Argument #3.2	uint8[p]	individual HEDS data octets

Figure 3.53: Structure for decoding the arguments of the SET_HEDS message.

The message contains three arguments: an array of checksums, Basic HEDS, and HEDS data paired with corresponding system state identifiers. The checksums are in array form, each represented by an unsigned 32 bit integer, and evaluated the same as for the 1451.4 TEDS. The Basic HEDS are fixed length (11 bytes) and interpreted as an array of octets. The HEDS data are variable length and also interpreted as octets. Note that the pairing of system states and HEDS data as part of an array allows many states to be associated with the same HEDS data. While a client message containing HEDS data with no state relationship is an illegal mode, a state may have no HEDS data, indicating the routine is not to run during that state. HEDS may be appended at anytime once the sensor is in the BL_ACTIVE state, although for best performance it is advisable to not append new HEDS during live measurement. This message format reduces redundancy of the HEDS data while allowing flexibility. After parsing and processing a *SET_HEDS* request, the Intelligent Sensor will reply with the message shown in Fig. 3.54 indicating the status of the operation.

Server Port Message Format:

Type	Msg Len		Client/Server Return Code				Block Cookie (Server Port)		Ex. Mode	Object ID (Server)									Operation ID	
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
0xD5	0x00	0x15	$CSRC_3$	$CSRC_2$	$CSRC_1$	$CSRC_0$	BC_1	BC_0	0x01	0x03	MAC_5	MAC_4	MAC_3	MAC_2	MAC_1	MAC_0	OID_1	OID_0	0x2C	0x03

Figure 3.54: Reply message after HEDS are sent and parsed by the Smart Sensor.

The final client-server message is to invoke *GET_HEDS* to retrieve HEDS from an Intelligent Sensor. This is extremely useful for both development and deployment, where HEDS may be backed up for archival or captured for cloning to other sensors of the same configuration. The message transaction consists of the request shown in Fig. 3.55, followed by the message and structure of the reply in Fig. 3.56.

Client Port Message Format:

Type	Msg Len		Block Cookie (Client Port)		Object ID (Server)									Operation ID		Ex. Mode
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
0xED	0x0	0x11	BC_1	BC_0	0x03	MAC_5	MAC_4	MAC_3	MAC_2	MAC_1	MAC_0	OID_1	OID_0	0x2C	0x02	0x0

Figure 3.55: Message for requesting HEDS from an Intelligent Sensor.

Server Port Message Format:

Type	Msg Len		Client/Server Return Code				Block Cookie (Server Port)		Ex. Mode	Object ID (Server)									Operation ID		# Args in Msg Contents		Arg #1 TC	Number of Elements in HEDS Chksm		Arg. #1: Checksums for the HEDS Data			
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
0xD5	*L*		$CSRC_3$	$CSRC_2$	$CSRC_1$	$CSRC_0$	BC_1	BC_0	0x0	0x03	MAC_5	MAC_4	MAC_3	MAC_2	MAC_1	MAC_0	OID_1	OID_0	0x2C	0x02	0x00	0x03	0x30	*m*		CS_3	CS_2	CS_1	CS_0		

...	Arg. #2 TC	Num Octets in BASIC HEDS		Basic HEDS				Arg. #3 TC	Num Args in Arg. Array "HEDS Data"	Arg. Array Arg #2 TC	Num of elements in System State	System State #1				Arg. Array Arg #3 TC	Number of octets in HEDS Record		HEDS Record octets for the ObjectID / System State Combination			...
	26+4m	27+4m	28+4m	29+4m	30+4m	...	39+4m	40+4m	41+4m	42+4m	43+4m	44+4m	45+4m	46+4m	47+4m	48+4m	49+4m	46+4m+4n	47+4m+4n	48+4m+4n	49+4m+4n	...	48+4m+4n+p
	0x1D	0x0	0x0C	HB_0	HB_1	...	HB_{10}	0x01	0x0	0x02	0x30	*n*		SS_3	SS_2	SS_1	SS_0	0x1D	*p*		HR_0	HR_1	HR_p

Figure 3.56: Reply to request for HEDS Data.

The format and flow of the arguments in Fig. 3.55 and Fig 3.56 are the same as for the *SET_HEDS* message. Since that the number of HEDS/state pairs on any individual sensor is variable, the end of the message is marked with an ellipsis to indicate that the arguments may repeat for another HEDS/state pair.

In summary, this chapter presents the implementation of the key components that constitute the first generation of Intelligent (health-enabled, ISHM ready) Smart Sensors. Those core competencies include the development of the ERM, design of HEDS, development of an IEEE 1451.1 compatible interface, and deployment of exemplar event detection routines. This chapter serves as both a technical reference documenting the technology embedded into the Intelligent Sensor and an interface control document for specifying the interface to the Intelligent Sensor. Now in Chapter 4 the completed deliverable is presented and evaluated.

CHAPTER 4: RESULTS

This chapter takes the blueprint for the Intelligent Sensor developed in Chapter 3 and focuses on two key results: (1) the performance of the Intelligent Sensor in terms of command and control via the provided interfaces (namely network and coding) and (2) the performance of the Intelligent Sensor as a measurement instrument. There are two sections that follow, addressing each of these issues.

4.1 Interfacing with Network Messages and Software APIs

The Intelligent Sensor Baseline 3.20 firmware was used for all testing in this section, and is available for download [72]. Using this baseline, it is possible to implement the health algorithms described in Chapter 3 using the ERM and HEDS APIs. Once linked and compiled, I was then able to command the Intelligent Sensor to generate an NCAP Block Announcement, change the sampling period, load HEDS, commence measurement, and cease measurement. During the measurement process and without any HEDS loaded, only measurements were published. After loading HEDS, the measurements were appended with health parameters generated from the associated algorithms. Health alerts were also received when signal activities corresponding to noise, flat and spike events were observed. Testing was performed to ensure that ERM handles linked routines properly. Maximum sustainable sampling rate with all health algorithms active is 50Hz. Network messages of measurement data and health data are issued at the same rate, though jitter slightly due to the task switching of the OS. Network transactions for these events are available for design reference [72]. The Intelligent Sensor is also able to coexist with other Intelligent Sensors on the same network, and can operate in either a debugging mode

(connected to a Dynamic C IDE) or as a standalone unit. See Figure 4.1 for a view of the Dynamic C IDE running the 3.20 Baseline. In addition to IDE debugging (watches, stack trace, execution trace, and code stepping), there are debugging options in SNTS_Headers.lib for printing additional information to the standard IO (STDIO) window.

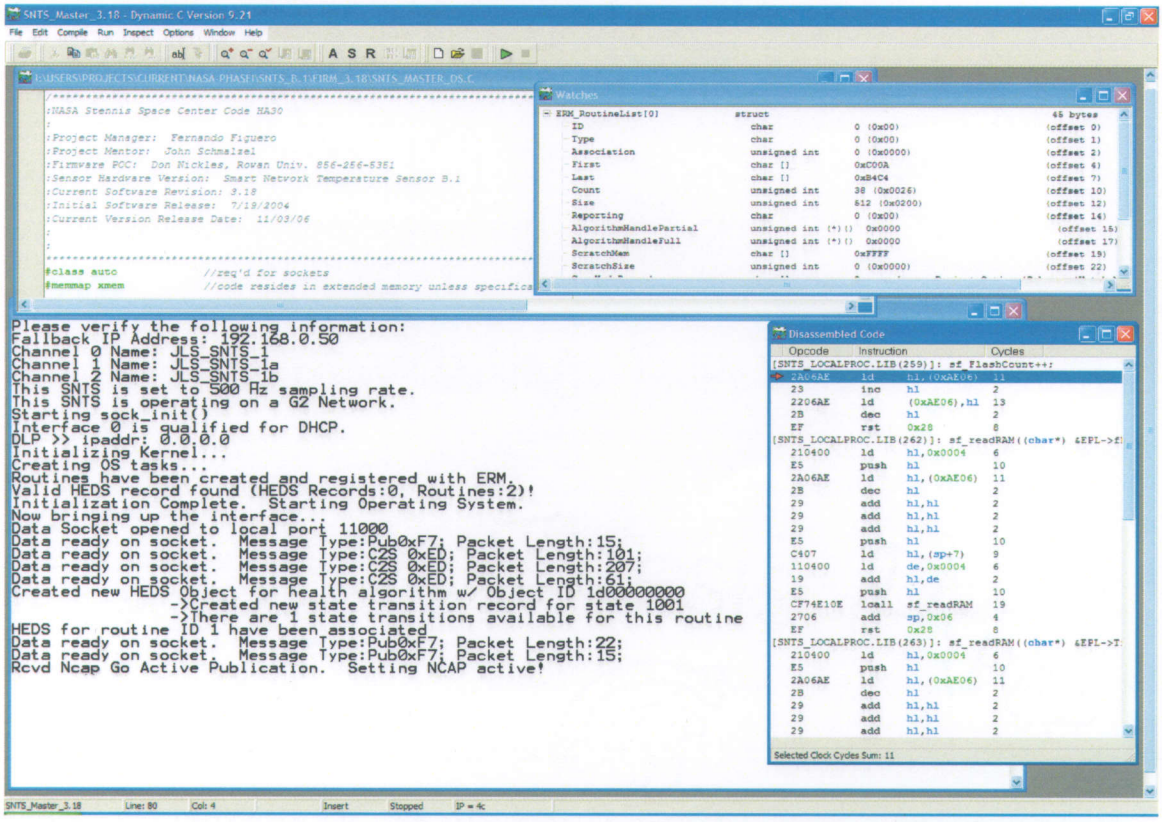


Figure 4.1: Dynamic C IDE with Intelligent Sensor in Debug Mode.

The next section discusses the details of the instrumentation and algorithms.

4.2 Evaluation of Form, Fit, and Function

Source data streams into the Intelligent Sensor for validation and verification include K-type thermocouple, sinusoid function generator, and previously recorded data from the MTTP Program. TC measurements are made using the Fluke Model 724 Temperature Calibrator as the data source, simulating a K-type thermocouple. The simulator's operational range is -200°C to

1370°C. The Intelligent Sensor is able to accurately measure the temperature within ±2°C over the extended/industrial temperature range (-40°C to 85°C) when the sensor is operating in an ambient temperature of 22°C with minimal air disturbance. A graph of the temperature sweep captured from the Intelligent Sensor is shown in Fig. 4.2. The sensor tends to deviate when the sweep is performed in an environment with fluctuating ambient temperature. This is due in part to the non-isothermal nature of the analog connecting block to the onboard temperature sensor and the location of the onboard temperature sensor near heat sources on the main board. The onboard ambient temperature sensor returns a voltage that is directly proportional to the temperature in °C. The coefficient is typically 0.81mV/ °C ±2°C [21], and the individual sensor has been calibrated using a single point calibration. Potential solutions are creating an isothermal environment around the Intelligent Sensor or attaching a separate temperature sensor directly to the connecting block and then shielding the block from nearby heat sources.

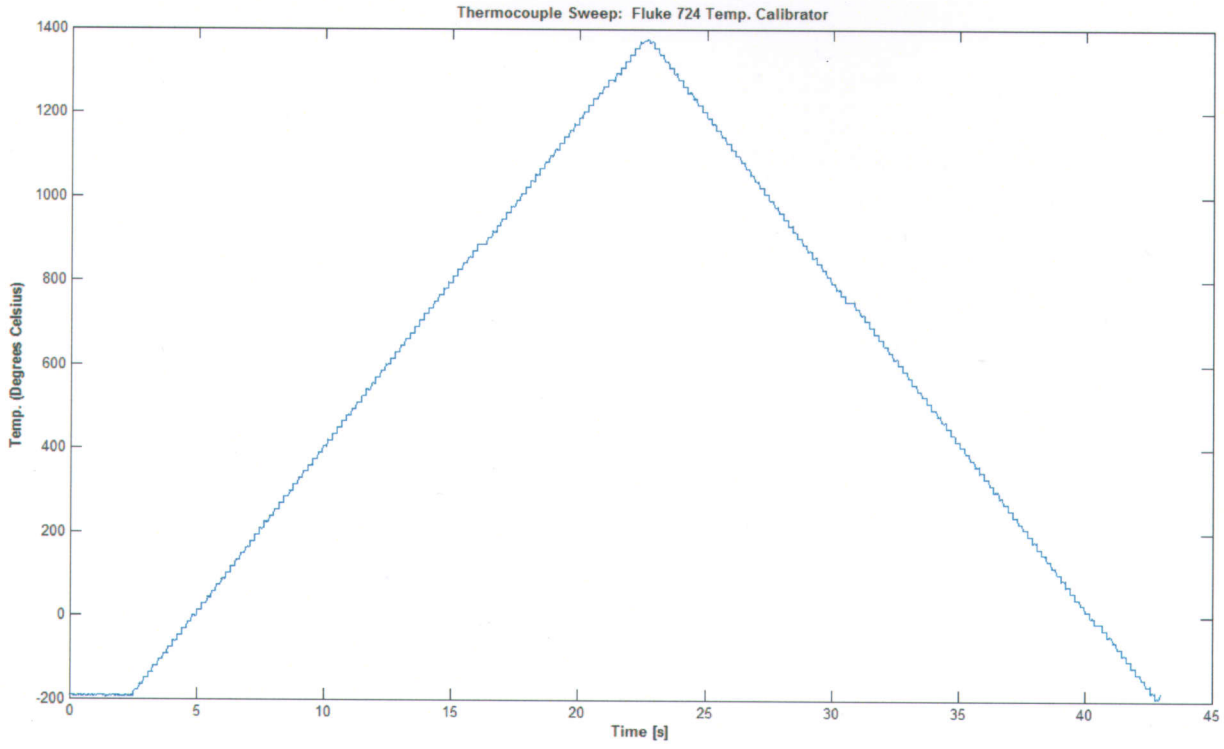


Figure 4.2: Thermocouple sweep using Fluke TC simulator from -200°C to 1370°C.

This test establishes the most fundamental function of the Intelligent Sensor – to accurately measure temperature. The next task is to evaluate the part of the Intelligent Sensor that makes it Intelligent – the quality of the ERM-based health routines. The evaluation is performed against a canonical sinusoid forcing function to establish credibility. Further evaluation is performed against recently obtained MTTP Program data to examine the overall capability against real data where there are events of interest (keep in mind the algorithms presented here are designed for general purpose use, as MTTP data was not available during the algorithm design phase).

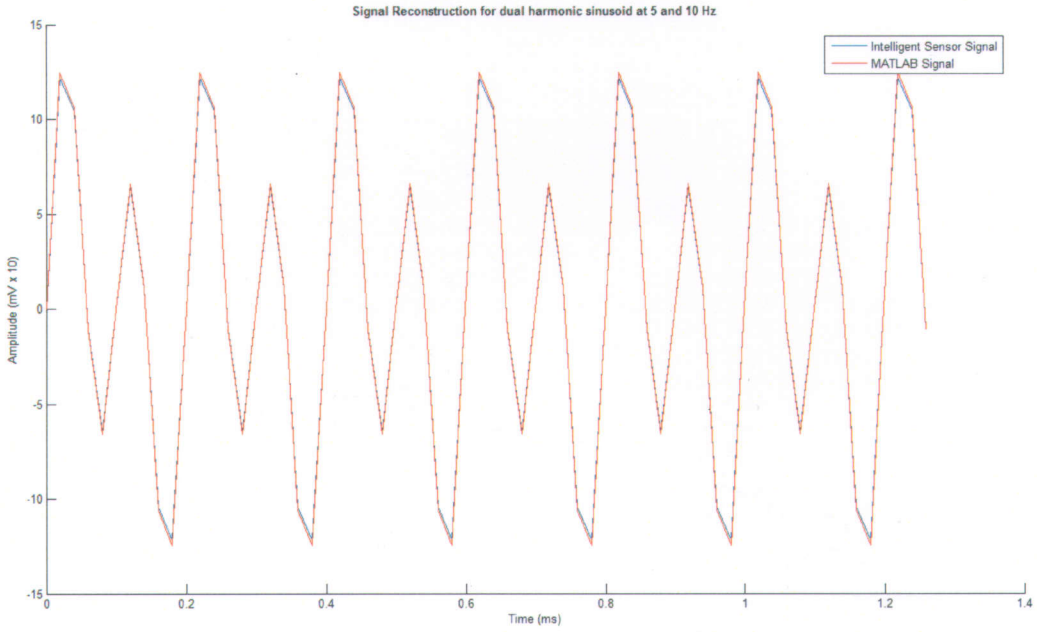


Figure 4.3: Dual tone sinusoid signal produced by Intelligent Sensor as compared to a MATLAB simulated equivalent with harmonics at 5Hz and 10Hz.

The basic sinusoid of Fig 4.3 is used to observe and verify the behavior of the DFT approximation as shown in Figure 4.4. The forcing function of Fig. 4.3 is applied to the input of the Intelligent Sensor and synthesized by MATLAB. The Intelligent Sensor reconstructed signal and DFT is compared to the MATLAB generated signal and associated DFT in Fig. 4.4. In both cases the Intelligent Sensor is indicated in blue and the MATLAB trace is in red. It is clearly

evident that the Intelligent Sensor performs well reconstructing the signal and approximating the DFT.

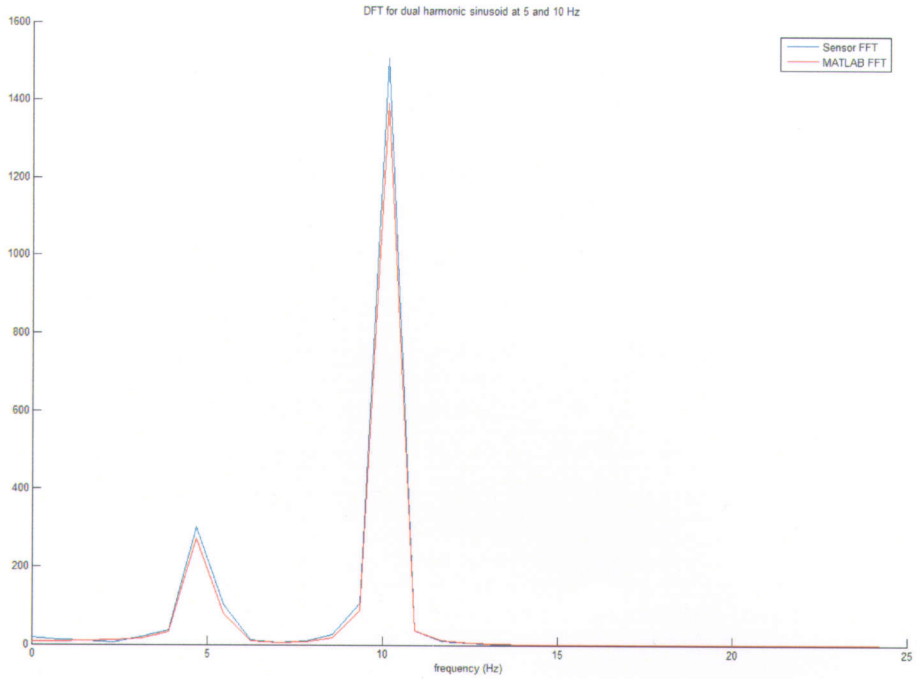


Figure 4.4: DFT spectrum of dual tone sinusoid evaluated by the Intelligent Sensor and verified with MATLAB.

The next evaluation is of the highpass filter and noise detection algorithm chain. For this analysis, a sinusoid signal was created on the HP function generator that steps from 1 to 25 Hz in 5Hz increments (see Fig. 4.5). Since noise is considered signal activity beyond 15Hz, the Intelligent Sensor will trigger events when the frequency exceeds that limit.

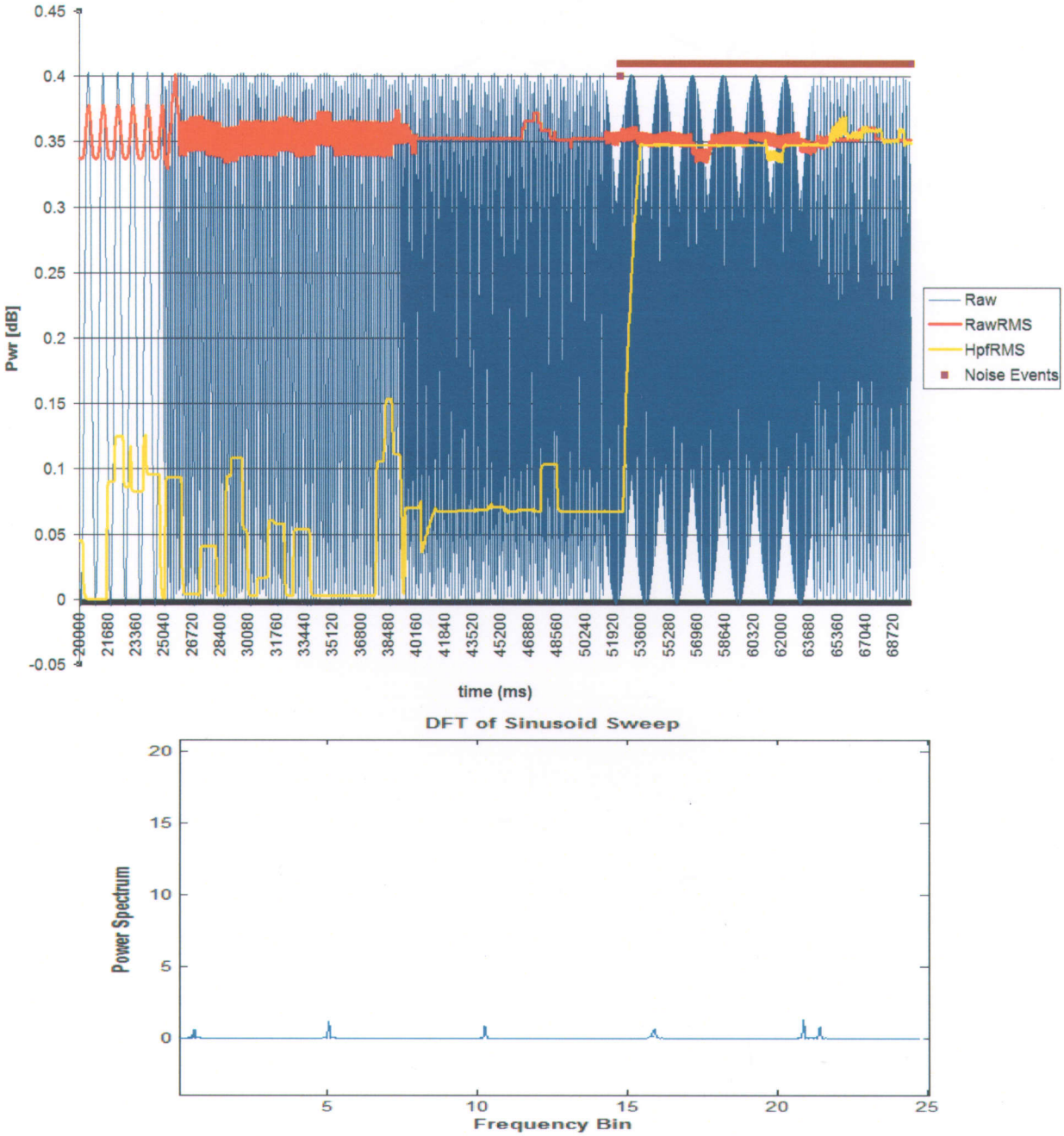


Figure 4.5: Sinusoid sweep to demonstrate noise event detection.

Once the algorithm chain enters the 15Hz region, a single event is started and then soon ended, due to the transition. Once completely in the 15Hz region, the HPF returns the same signal energy as the raw signal contains, resulting in a continuous noise event up until the end of the data.

The final evaluation is for the crest factor and spike event algorithm chain. Recently available MTTP data contains a good candidate for evaluation of this algorithm chain. The live test data is from tests 0092806-13-06-35, 0914-022D-6271, 0914-019C-6230, and 0914-018D-6223 utilizing the feedback signal of VPV1170. This signal was chosen for its characteristically spiky nature. Fig. 4.6 and Fig 4.7 show VPV1170FB for 0092806-13-06-35 and 0914-022D-6271. Overlaid onto the graphs are the crest factor and spike event status.

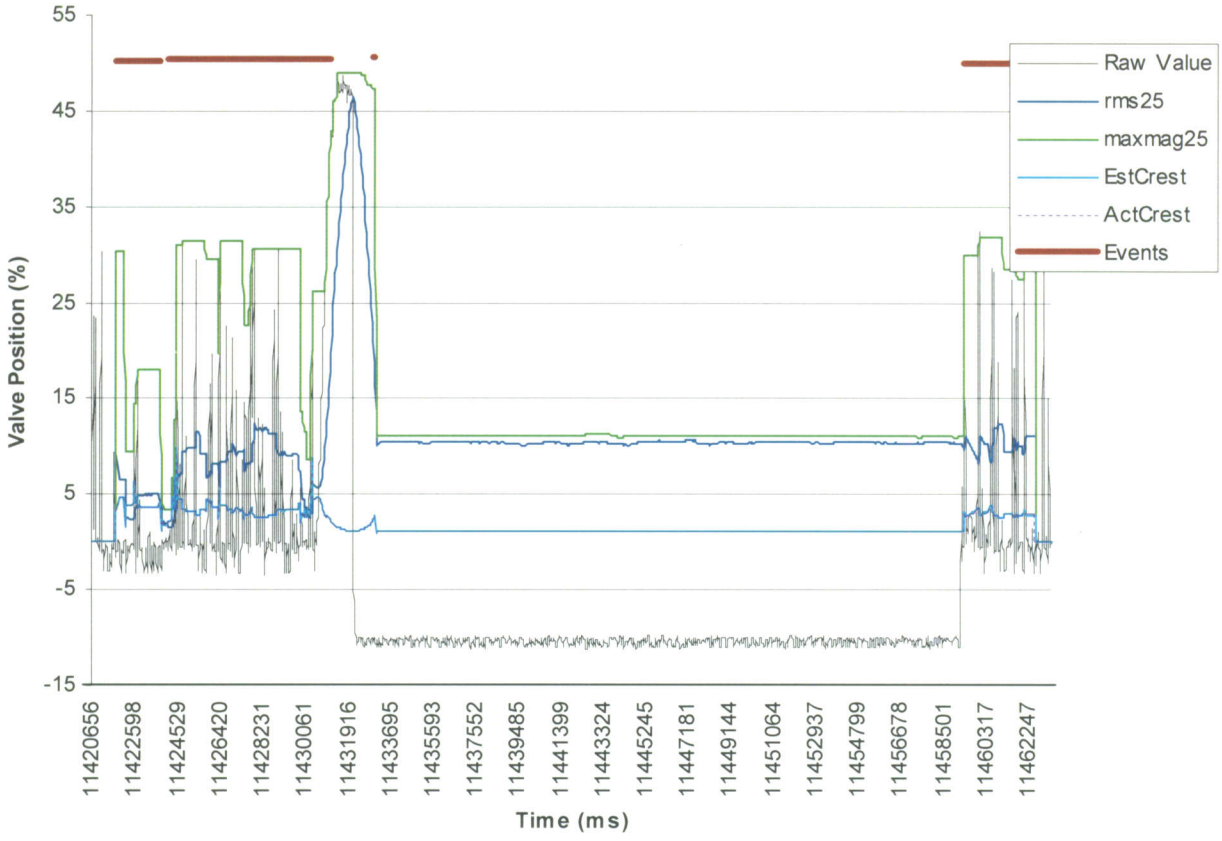


Figure 4.6: Crest Factor and Spike Event detection on 092806-13-06-35 VPV1170FB.

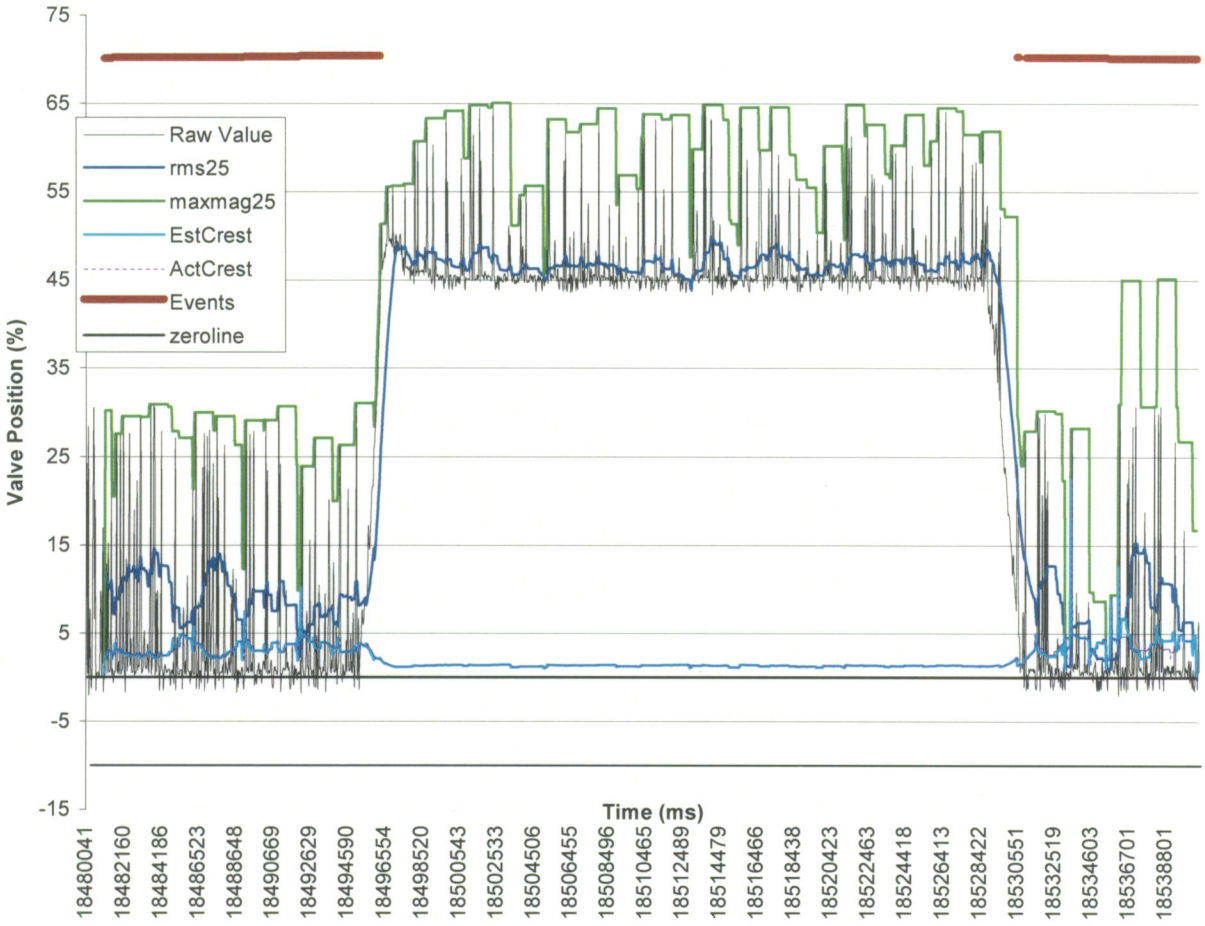


Figure 4.7: Crest Factor and Spike Event detection on 0914-022D-6271 VPV1170FB.

It is interesting to note the thorough detection of spike events in Fig 4.6, though in Fig 4.7 it seems to "miss" the spike events during the open phase. This is due in part to the use of 25-point window (providing lots of averaging), and the width of the spikes (raising the effective RMS) as can be seen in the graphs. Adjusting the size of the RMS and Max window would make the algorithm more sensitive to spikes of larger width. Also, if a known good signal typically has a very low crest factor, the crest factor threshold may also be lowered without incurring any false positives, while improving detection of less severe spikes.

CHAPTER 5: CONCLUSIONS

This research contributes to instrumentation technology through the realization of an IEEE 1451 compatible Smart Sensor that has the added capability to execute health assessment algorithms and signal processing routines while processing measurements in real-time. While successful, this work is by no means complete. There are several areas that are recommended for future work, some of which focus on improving the current baseline to resolve issues or limitations, and others that represent and evolutionary expansion to increase the contribution of the Intelligent Sensor to ISHM.

5.1 Future Work: Improving Capability

Several issues have been identified that should be addressed in the near term development spirals.

The first is resolving the ambient temperature issue. Suitability of the range of the onboard temperature sensor must be determined, and perhaps a more thorough calibration is required. Thermal synchronization between connecting block (an unavoidable TC junction) and the sensor is required, but may be achievable with shielding or external RTDs connected to the second or third analog input channel.

Another issue is the efficiency of the data acquisition subsystem. Currently the ADC operates in single sample mode to allow intermediate sampling of alternate channels (ambient, onboard voltage, etc). While the most aggressive solution is to use a more feature rich ADC (resulting in costly hardware development), other soft fixes may be achieved by using the continuous sampling mode for the main analog channel, but switching out to single sample mode

for the occasional alternate channel. The last option is to interface the ADC to the 8051 auxiliary CPU, freeing the Rabbit from controlling the data acquisition directly.

A full implementation of TEDS is required to support the truly dynamic nature of the Intelligent Sensor and IEEE 1451. IEEE 1451.1 provides psuedocode for parsing template description language (TDL) and can be the start for writing more advanced calibration TEDS.

5.2 Working Towards the Next Generation

There are many potential directions to aim future research efforts. Those areas consist of improving the IEEE 1451 interface, the addition of instrumentation-grade time synchronization, further development of HEDS, integrating the network firmware update to interface with a configuration/deployment tool, and integration of additional health assessment routines.

IEEE 1451 is a key component of ISHM. The 1451 capability in this baseline is a work in progress due to the need to work on other health-related capabilities at the same time. Furthermore, some 1451 standards are undergoing initial revision. It is important to fully develop the 1451 object model and interface specification, and to keep this up to date, as adoption of 1451 as an instrumentation framework by those in industry will inadvertently pave the way for ISHM proliferation, even if they are not ready or feel ISHM is worth the gamble at this point in time.

Time synchronization is necessary to ensure Smart Sensors remain time aligned while operating over extended periods of time. The only high accuracy instrument grade synchronization protocol for Ethernet networks is IEEE 1588, although lesser accuracy is obtainable by using NTP. Integration is a mix of both hardware and software goals, as there is the need for precision oscillators, hardware for packet timing, and the ability to adjust the timing hardware to compensate for drift.

The HEDS presented in this work offers the essential capability for configuration of health assessment routines. As ISHM development continues, there is an advantage to formally define the core components of HEDS and standardizing these across the board. A thorough development of HEDS would not be complete without integration into IEEE 1451, setting the standard for the future of health management. In conjunction, adding more capability to ERM will be necessary to support additional algorithm demands – such as saving list state for long term analysis, framework to support real-time fuzzy logic or neural network operations, a modularization to support different independent ERM subsystems on different channels, and more intelligence to automatically avoid serving a linked routine with nodes from the associated routine's list when it is in its delay phase and has not yet begun to produce output. Finally, more control of ERM's utilization of resources – essentially tighter integration as an operating system plug-in.

The firmware update service developed through this work demonstrates the powerful capability of remotely updating Smart Sensors anywhere in the world. The next level of achievement is to integrate the functionality into a configuration/deployment tool that can pull together application specific firmware that is stored in a VOB or database, compile it on the fly, and distribute it to a network of Intelligent Sensors for truly dynamic network configuration.

The last and final suggestion is integration of a complete set of health assessment or signal processing routines especially designed for specific Intelligent Sensors deployed on an ISHM-capable vehicle. The current routine suite demonstrates capability, and is not an integral part of a component or system model. This action is paramount if there is to be any benefit of the ISHM data, information, knowledge, and action paradigm.

REFERENCES

[1] Figueroa, F., Morris, J., Nickles, D., Schmalzel, J., Rauth, D., Mahajan, A., Utterbach, L., Oesch, C., "Intelligent Sensor and Components for On-Board ISHM," Proceedings of the 42nd AIAA/ASME/SAE/ASEE Joint Propulsion Conference & Exhibit, Sacramento, CA, 9-12 July 2006.

[2] Nickles D., Rauth D., Schmalzel J., "Enhanced Smart Sensor for Integrated System Health Management," *Sensors Applications Symposium 2006*, Houston, TX, 7-9 February 2006.

[3] Schmalzel J., Figueroa, F., Morris, J., Mandayam, S., Polikar, R., "An Architecture for Intelligent Systems Based on Smart Sensors." *Proc. IMTC 2004,* Como, Italy, 18-20 May 2004.

[4] Lewis S., Edwards T., "Smart Sensors and System Health Management Tools for Avionics and Mechanical Systems."

[5] Bos, A., "Model-based Health Tracking," S&T Corp. and Delft University of Technology, 2001.

[6] Jaw, L.C., Dong, W.N., Bloor, G., Daumann, A., "Anomaly Detection and Reasoning with Embedded Physical Model," 2002 IEEE.

[7] Figueroa F., Solano W., Thurman C., Schmalzel J., "A Future Vision of Data Acquisition: Distributed Sensing, Processing, and Health Monitoring." *Proc. IMTC 2001,* Budapest Hungary, 20-23 May 2001.

[8] Delsing, J., Lindgren, P., "Sensor Communication Technology Towards Ambient Intelligence," Institute of Physics Publishing, Measurement Science and Technology, Lulea, Sweden, 2005.

[9] "IEEE Standard for Smart Transducer Interface for Sensor and Actuators – Network Capable Application Processor Information Model," IEEE 1451.1, 2000.

[10] "Intelligent Ships: The future of the US Navy," Intelligent Ships Symposium, 2002.

[11] Fox, Jack J., "Impact of Integrated Health Management Technologies on Group Operations for Reusable Launch Vehicles and Spacecraft"

[12] Schwabacher, M., Samuels, J., Brownston, L., "The NASA Integrated Vehicle Health Management Technology Experiment for X-37" *Proc. SPIE AeroSense 2002.*

[13] Williams, B. C., Nayak, P. P., "A model-based Approach to Reactive Self-Configuring Systems," 13th National Conference on Artificial Intelligence, 04-08 Aug. 1996, Portland, OR., pp. 971-978.

[14] Open Systems Architecture for Condition Based Maintenance, http://osacbm.org, Applied Research Laboratories, Box 30, State College, PA.

[15] Crow E., Reichard K., Banks J., Weiss, L., "Integrated System health Management for Increased Autonomy, Reduced Operational Risk, Improved Capability," 8 Feb 2005, Penn State Applied Research Laboratory, State College, PA.

[16] Lebold, M., Thurston, M., "Open Standards for Condition-Based Maintenance and Prognostic Systems," MARCON 2001.

[17] Discenzo, F., Keller, K., Mitchell, C., Nickerson, W., "Open Systems Architecture Enables Health Management for Next Generation System Monitoring and Maintenance," White Paper, 10 April 2001.

[18] Discenzo, F., "OSA-CBM Description," OSA-CBM Milestone 4 Review, Virginia Beach, VA, 21 September 2000.

[19] Regan, Ronald, Executive Order 12641, May 20th, 1988.

[20] Hall, D., Llinas, J., "An Introduction to Multisensor Data Fusion," Proceedings of the IEEE, January 1997.

[21] Analog Devices Inc., "AD7794/5 6-Channel, Low Noise, Low Power, 24-/16-Bit Sigma-Delta ADC with On-Chip In-Amp and Reference," 6/2006.

[22] Epson Inc., "SG-3030JC 32.768kHz Crystal Oscillator with +5/-23ppm Tolerance."

[23] Analog Devices Inc., "ADR421: Ultraprecision, Low Noise, 2.500V, 3ppm/°C XFET Voltage reference," 6/2005.

[24] Institute of Electrical and Electronics Engineers, Inc., "IEEE 1588-2002 Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems," Technical Committee on Sensor Technology TC-9 of the IEEE Instrumentation and Measurement Society, http://grouper.ieee.org/groups/1588.

[25] Intel Corp., "Hardware-Assisted IEEE 1588 Implementation in the Intel IXP46x Product Line," White Paper, March 2005.

[26] National Instruments, Inc., "Sensor Calibration with TEDS Technology," http://zone.ni.com/devzone/cda/tut/p/id/4043, 2006.

[27] "IEEE 1451 Website," National Institute of Standards and Technology (NIST), Gaithersburg, Maryland.

[28] Institute of Electrical and Electronics Engineers, Inc., "IEEE 1451.2-1997 Standard for a Smart Transducer Interface for Sensors and Actuators – Transducer to Microprocessor

Communication Protocols and Transducer Electronic Data Sheet (TEDS) Formats," Technical Committee on Sensor Technology TC-9 of the IEEE Instrumentation and Measurement Society, 1997, http://grouper.ieee.org/groups/1451/2.

[29] Institute of Electrical and Electronics Engineers, Inc., "IEEE 1451.3-2003 Standard for a Smart Transducer Interface for Sensors and Actuators – Digital Communication and Transducer Electronic Data Sheet (TEDS) Formats for Distributed Multidrop Systems," Technical Committee on Sensor Technology TC-9 of the IEEE Instrumentation and Measurement Society, 2003, http://grouper.ieee.org/groups/1451/3.

[30] Institute of Electrical and Electronics Engineers, Inc., "IEEE 1451.4-2004 Standard for a Smart Transducer Interface for Sensors and Actuators – Mixed-Mode Communication Protocols and Transducer Electronic Data Sheet (TEDS) Formats," Technical Committee on Sensor Technology TC-9 of the IEEE Instrumentation and Measurement Society, 2004, http://grouper.ieee.org/groups/1451/4.

[31] Institute of Electrical and Electronics Engineers, Inc., "IEEE 1451.5 Standard for a Smart Transducer Interface for Sensors and Actuators – Wireless Communication Protocols and Transducer Electronic Data Sheets (TEDS) Formats," Wireless Sensor Working Group of the Technical Committee on Sensor Technology TC-9 of the IEEE Instrumentation and Measurement Society, http://grouper.ieee.org/groups/1451/5.

[32] Institute of Electrical and Electronics Engineers, Inc., "IEEE 1451.6 Standard for a Smart Transducer Interface for Sensors and Actuators – A high-speed CANopen based Transducer network Interface for Intrinsically Safe and non-Intrinsically Safe applications," CANopen Transducer Network Interface Working Group of the Technical Committee on Sensor Technology TC-9 of the IEEE Instrumentation and Measurement Society, http://grouper.ieee.org/groups/1451/6.

[33] Agilent Technologies, Inc., and Boeing Company, "1451.1 On-the-Wire Format for IP," Rev. 1, April 2002.

[34] Chronology of Apollo 11: http://history.nasa.gov/Timeline/apollo13chron.html.

[35] Glass, B.J., Erickson, W.K., Swanson, K.J., "TEXSYS: A Large Scale Demonstration of Model-Based Real-Time Control of a Space Station Subsystem," Proceedings of the Seventh IEEE Conference on Artificial Intelligence Applications, February 24th, 1991.

[36] NASA Kennedy Space Center, "Research and Technology 1996 Annual Report," Memorandum 112650, http://rtreport.ksc.nasa.gov/techreports/96report/96report.pdf, 1996.

[37] Rayman, M.D., Varghese, P., Lehman, D.H., Livesay, L.L., "Results from the Deep Space 1 Technology Validation Mission," 50th International Astronautical Congress, Amsterdam, Netherlands, 4-8Oct99.

[38] Clancy, D., Zakrajsek, J., Kruhm, D., "NASA IVHM Technology Experiment for X-vehicles (NITEX) Project Plan," May, 1999.

[39] Hedley, M., Johnson, M., Lewis, C., Carpenter, D., Lovatt, H., Price, D., "Smart Sensor Network for Space Vehicle Monitoring," International Signal Processing Conference 2003.

[40] Erickson, T., "Turbidity Sensing as a Building Block for Smart Appliances," IEEE Industry Applications Magazine, pp31-36, Vol. 3, Issue 3, May 1997.

[41] Okey, M.C., Ruane, P.M., "Advances in Appliance Control: The Breaking of a Paradigm," IEEE International Conference on Controls Applications, Dearborn, MI, 15Sept96.

[42] "Sensors plug & Play: Smart Sensor for Faster Setup and Development," National Instruments Webcast, http://sine.ni.com/apps/utf8/nievn.ni?action-display_offerings_by_event&event_id=14329&event_subtype=WEB_EVENT_DEMAND&site=NIC&l=US.

[43] "Two Approaches to Providing Smart Sensor capabilities: Incorporating Identification Capabilities per IEEE 1451.4 or Complete Data Acquisition and Networking Capabilities", Endevco Product Brochure, http://www.endevco.com/resources-brochures/networksensors.php.

[44] Honeywell Corp. Sensing and Control Division, "Equipment Health Monitoring Systems," Datasheet, December 2004, http://www.honeywell.com/sensing.

[45] Smart Sensor Systems Inc, 720 SW 14th St, Loveland CO 80537.

[46] ESensors Inc, 4240 Ridge Lea Rd, Amherst, NY 14226.

[47] Lee, K., "Synopsis of IEEE 1451: Empowering the Smart Sensor Revolution," Sensors Conference/Expo 2005, Chicago, IL, 07 June 2005

[48] Wall, R. W., Ekpruke, A., "Developing an IEEE 1451.2 Compliant Sensor for Real-Time Distributed Measurement and Control in an Autonomous Log Skidder" University of Idaho, *Proc.of the 29th IES.*

[49] The SensorNet Project, http://www.sensornet.gov/, Oak Ridge National Laboratory (ORNL), United States Department of Energy (US DoE), United States Department of Defense (US DoD), National Oceanic and Atmospheric Administration (NOAA), and United State Department of Homeland Security.

[50] "IEEE Standard for Information Technology 802.3af – Telecommunications and information exchange between systems – Local and Metropolitan Area Networks – Amendment: Data Terminal Equipment (DTE) Power via Media Dependent Interface (MDI)," 2003, IEEE Press.

[51] Nickles, D., Schmalzel, J., "Intelligent Validation of Intelligent Sensor for Integrated Systems Health Management," Sensors Applications Symposium 2007, San Diego, CA, 2007.

[52] "Event Detection Interface Control Document and User Manual for ISHM Testbed and Prototypes Project, Version 1.0," 1 April 2005, NASA Glenn Research Center, Cleveland, OH.

[53] Lee, K. B., Song, E. Y., "Object-Oriented Application Framework for IEEE 1451.1 Standard," IEEE Transactions on Instrumentation and Measurement, Vol. 54, No.4, August 2005.

[54] Dallas Semiconductor Corp., "1-Wire Master Device Configuration," Application Note 2965, http://www.maxim-ic.com/AN2965.

[55] Dallas Semiconductor Corp., "DS2430A: 256-Bit 1-Wire EEPROM with Unique 64-bit Factory-Lasered Registration Number," November 10th, 2005.

[56] Oostdyk, R., Mata, C. T., Perotti, J. M., Lucena, A. R., Mullenix, P., "A Kennedy Space Center implementation of IEEE 1451," Proceedings of SPIE Sensors For Propulsion Measurement Applications Conference, April 20-21st, 2006, Orlando, FL.

[57] James C. Schatzman, 1996, "Accuracy of the discrete Fourier transform and the fast Fourier transform," *SIAM J. Sci. Comput.* **17**: 1150–1166.

[58] M. Frigo and S. G. Johnson, 2005, "The Design and Implementation of FFTW3," *Proceedings of the IEEE* 93: 216–231.

[59] , "Rigorous Development of an Embedded Fault-Tolerant System Based on Coordinated Atomic Actions" IEEE Trans. On Computers, Vol. 51, No.2, February 2002.

[60] Petters, S., Farber, G., "Making Worst Case Execution Time Analysis for Hard Real-Time Tasks on State of the Art Processors Feasible." Institute for Real-Time Computer Systems, Technische Universitat, Munchen, Germany.

[61] Baynes, K., Collins, C., Fiterman, E., et al, "The Performance and Energy Consumption of Three Embedded Real-Time Operating Systems" Dept of Electrical and Computer Eng., University of Maryland at College Park, *Proc. CASES 2001,* Atlanta Georgia, 16-17 November 2001.

[62] Wind River, VxWorks Center, http://www.windriver.com/vxworks/index.html, Alameda, CA.

[63] Rabbit Semiconductor Inc., "RabbitSys User's Manual for Rabbit Semiconductor Microprocessors and Integrated C Development System," December 15th, 2005.

[64] SofTools Inc., "TurboTask User Manual," 2004.

[65] Labrosse, J. J., "MicroC/OS-II: The Real-Time Kernel 2nd Edition", CMP Books, Lawrence, KS, 2002, ISBN 1-57820-103-9.

[66] Radio Technical Commission for Aeronautics, Inc., "DO-178B: Software Considerations in Airborne Systems and Equipment Certification," December 1, 1992, Certified July, 2002.

[67] Rabbit Semiconductor Inc., "Rabbit 3000 Microprocessor User's Manual," Copyright 2002 - 2006.

[68] Rabbit Semiconductor, Inc., "Technical Note 224: Implementing a TCP-Based Download Manager," February 16th, 2001.

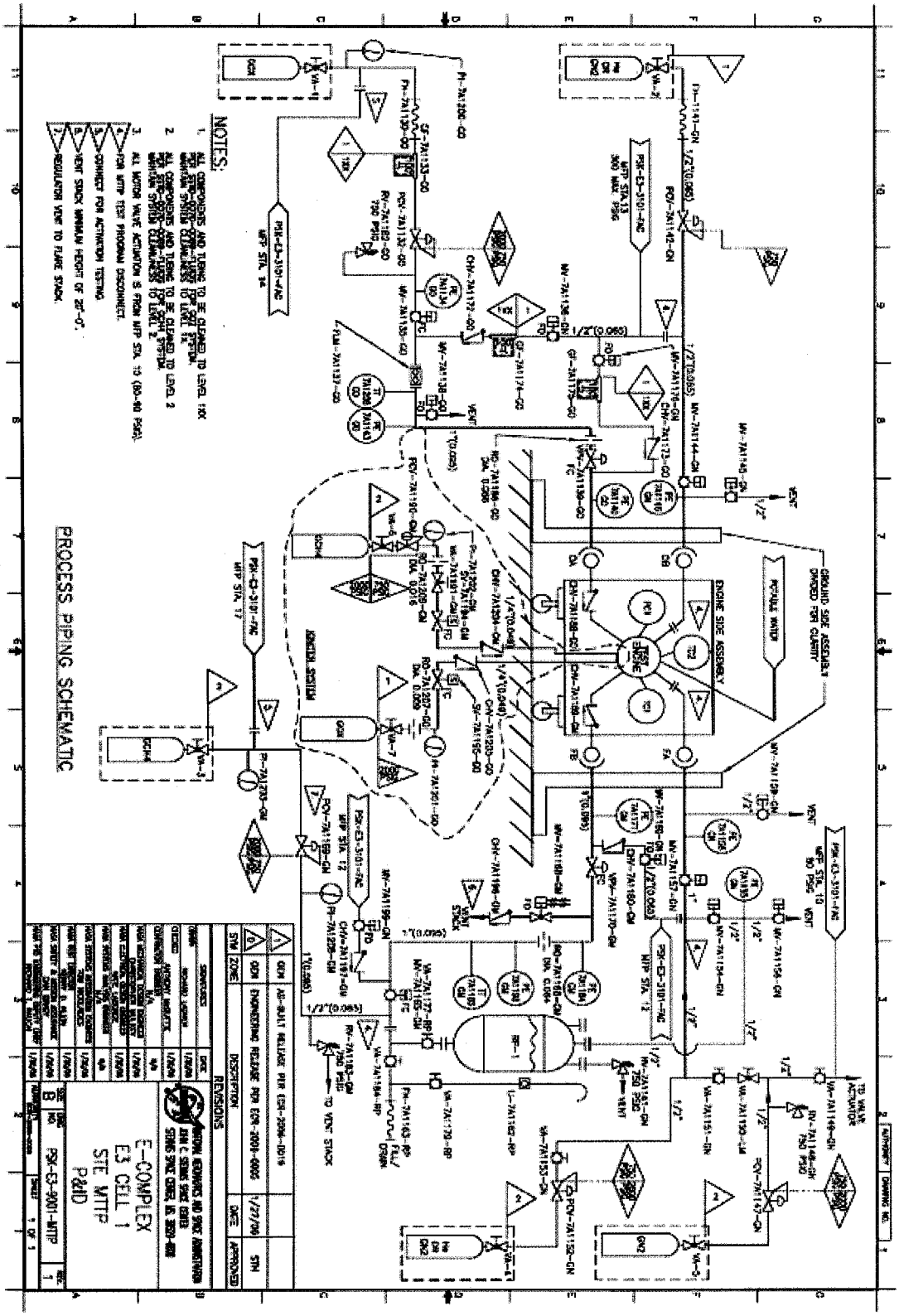
[69] "Remote Application Update," Application Note 022-0092 Rev A, Rabbit Semiconductor, CA.

[70] Turowski, M., "A Health Assessment Database System for Integrated Systems Health Management," Master's Thesis, in preparation.

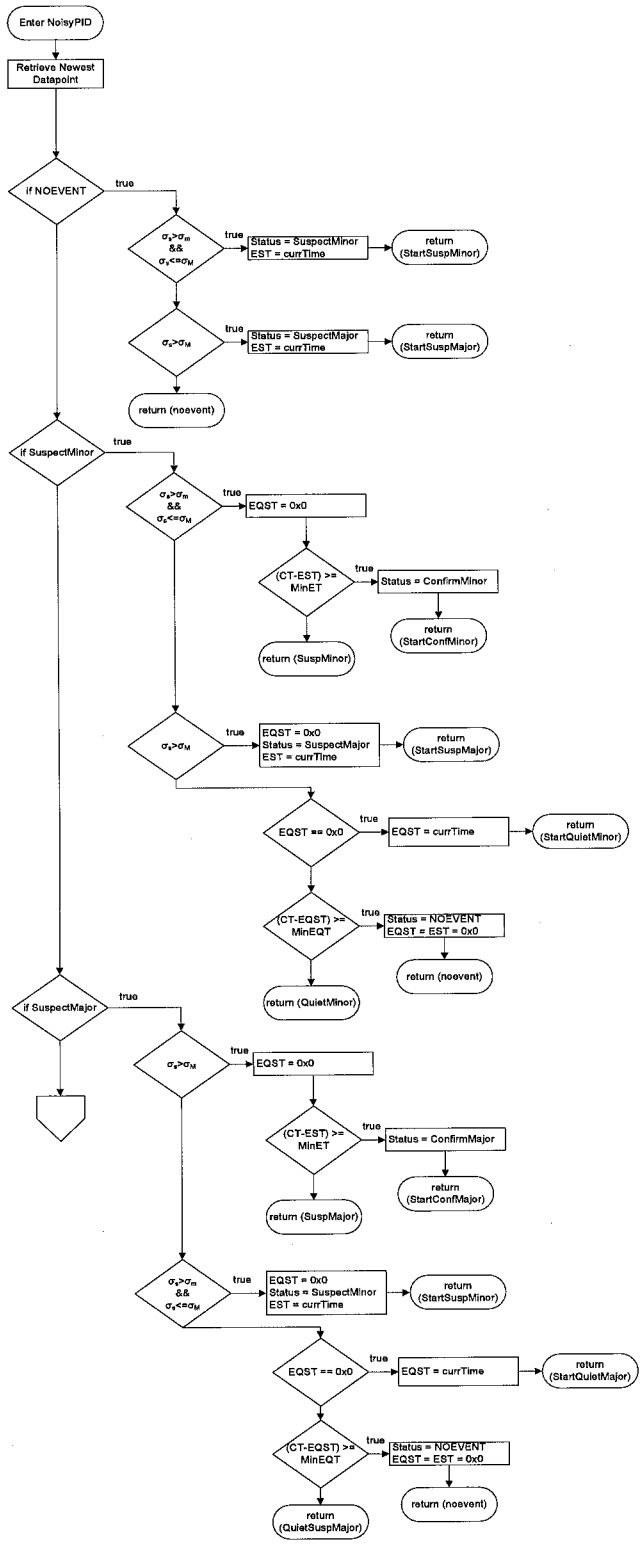
[71] Ogaro, J. A., "Heterogeneous multi-sensor data fusion using geometric transformations and Parzen Windows for the nondestructive evaluation of gas transmission pipelines," Master's Thesis, 2004.

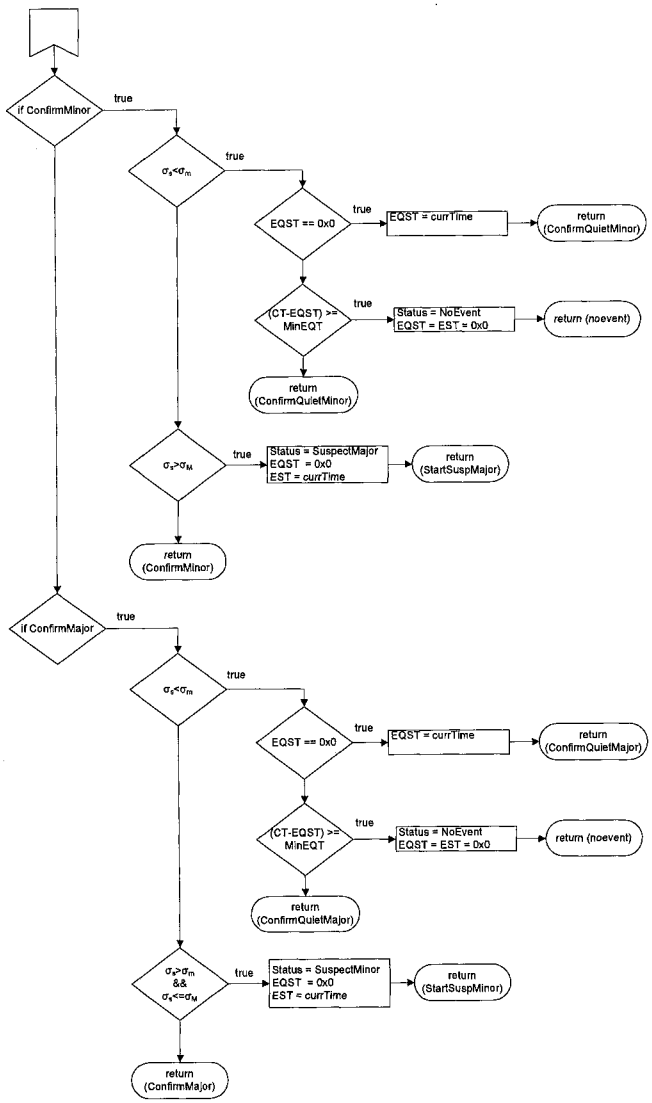
[72] Intelligent Sensor Online Resources, http://tcrc.cc/IntelligentSensor

Appendix A: PRETS MTTP Program PID

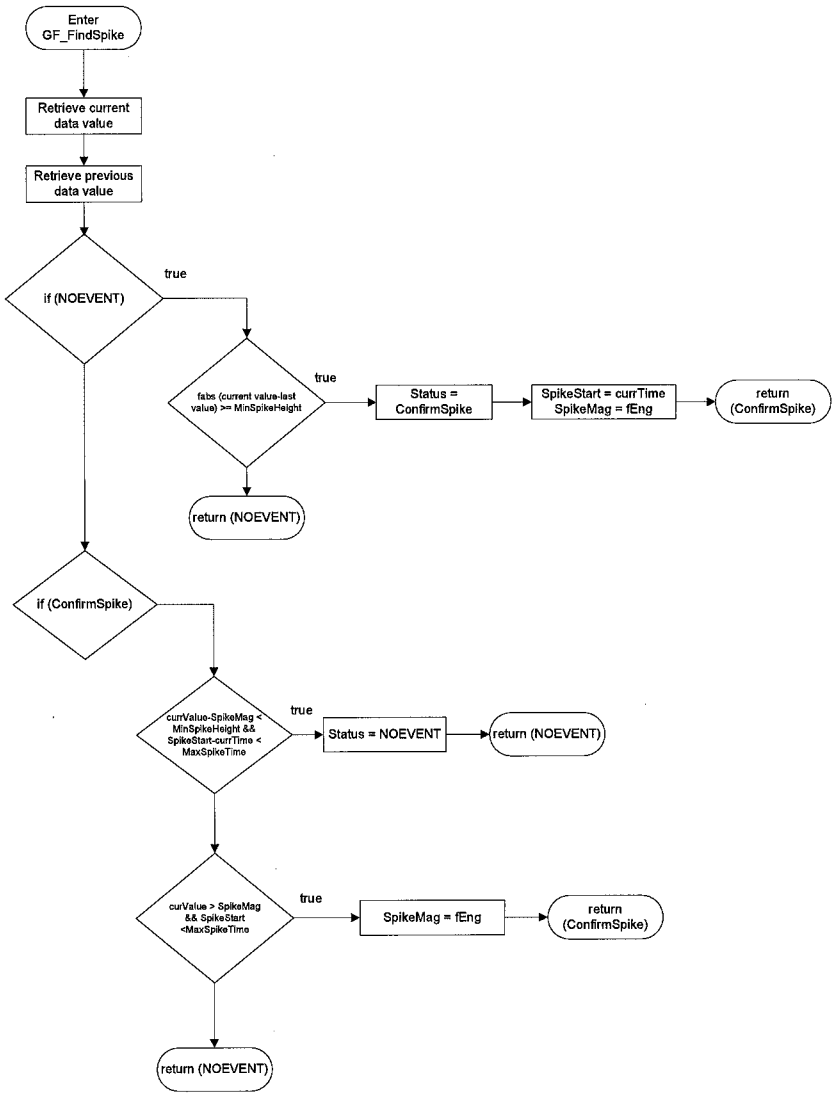


Appendix B: GRC Noise Detection Routine

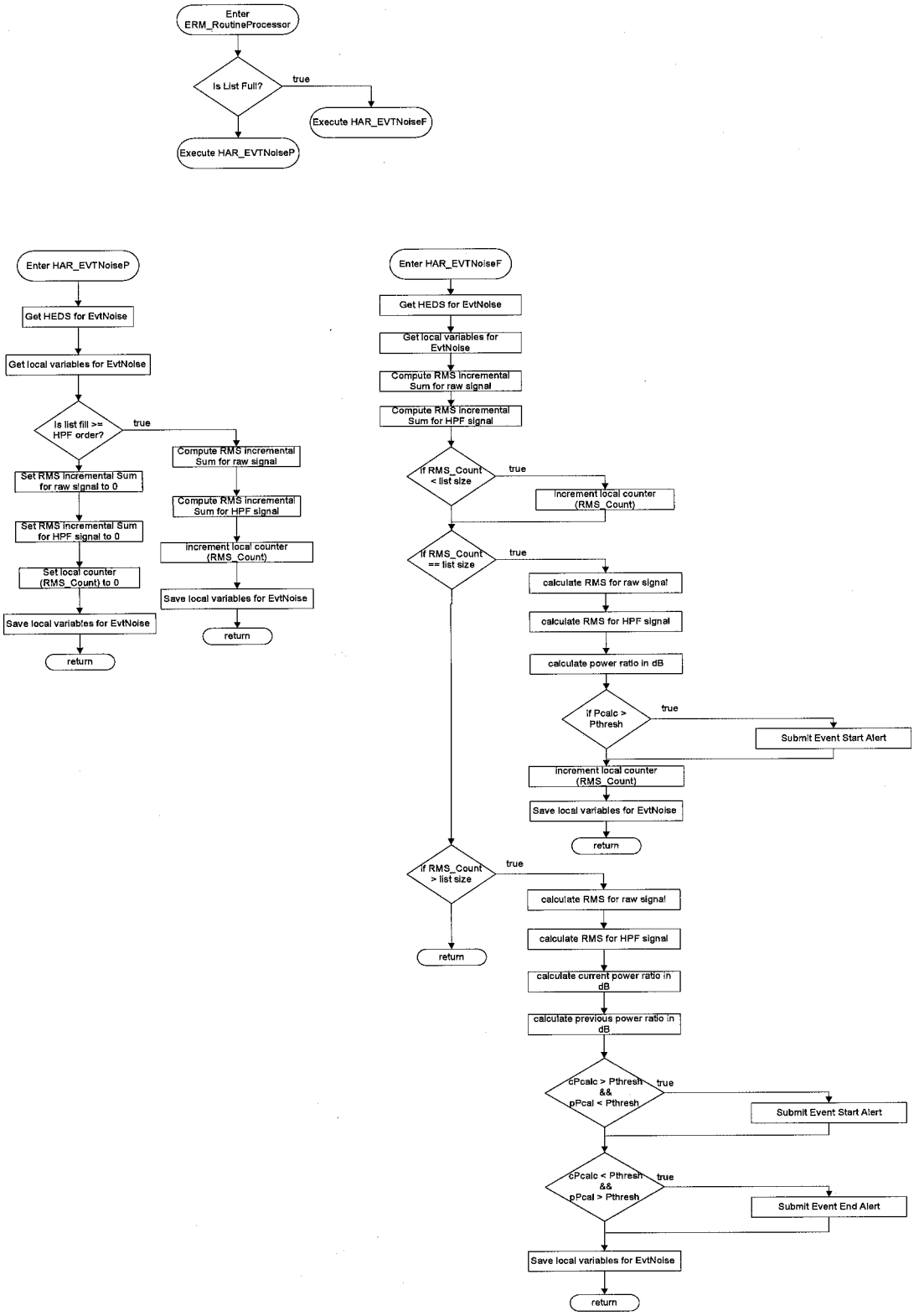



if ConfirmMinor
true
$\sigma_x<\sigma_m$
true
EQST == 0x0
true
EQST = currTime
return (ConfirmQuietMinor)
(CT-EQST) >= MinEQT
true
Status = NoEvent
EQST = EST = 0x0
return (noevent)
return (ConfirmQuietMinor)
$\sigma_x>\sigma_M$
true
Status = SuspectMajor
EQST = 0x0
EST = currTime
return (StartSuspMajor)
return (ConfirmMinor)
if ConfirmMajor
true
$\sigma_x<\sigma_m$
true
EQST == 0x0
true
EQST = currTime
return (ConfirmQuietMajor)
(CT-EQST) >= MinEQT
true
Status = NoEvent
EQST = EST = 0x0
return (noevent)
return (ConfirmQuietMajor)
$\sigma_x>\sigma_m$ && $\sigma_x<=\sigma_M$
true
Status = SuspectMinor
EQST = 0x0
EST = currTime
return (StartSuspMinor)
return (ConfirmMajor)

Appendix C: GRC IsSpike Routine



Appendix D: Noisy Signal Detection Real-time Algorithm



Appendix E: Spike/Flat-line Real-time Algorithm