Radar Tracking System Development

by

Yue Hann Chin

Submitted to the Department of Electrical Engineering and Computer Science

in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

May 19, 2005

Author____
Department of Electrical Engineering and Computer Science
May 19, 2005

Certified by____________________________________
Eliahu H. Niewood
VI-A Company Thesis Supervisor

Certified by____________________________________
David H. Staelin
M.I.T. Thesis Supervisor

Accepted by____________________________________
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

Radar Tracking System Development
by
Yue Hann Chin

Submitted to the
Department of Electrical Engineering and Computer Science

May 19, 2005

In Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science

ABSTRACT

The Airborne Seeker Test Bed (ASTB) is an airborne sensor testing platform operated by the Tactical Defense Systems group at MIT Lincoln Laboratory. The Instrumentation Head (IH) is a primary sensor on the ASTB. It is a passive X-band radar receiver located on the nose of the plane. The IH serves as a truth sensor for other RF systems on the test bed and is controlled by an onboard tracking system, the Seeker Computer. The Seeker Computer processes IH data in real-time to track targets in Doppler, angle, and range. From these tracks it then produces angle-error feedback signals that command the IH gimbals, keeping targets centered along the antenna boresight.

Over three years, a new Seeker Computer was built to replace an old system constrained by obsolete hardware. The redevelopment project was a team effort and this thesis presents a systems-level analysis of the design process, the new Seeker Computer system, and the related team and individual contributions to software and digital signal processing research that took place during development.

VI-A Thesis Supervisor: Dr. Eliahu H. Niewood
Title: Group Leader, MIT Lincoln Laboratory

VI-A Thesis Supervisor: Steven C. Harón
Title: Associate Staff, MIT Lincoln Laboratory

M.I.T. Thesis Supervisor: Dr. David H. Staelin
Title: Professor of Electrical Engineering

Acknowledgements

Thanks to my mom, Meeho, my dad, William, and my sister, Yue Pui. My family always encourages me to do great things and gives me endless support in the process. I love them very much.

I thank my advisor at Lincoln Laboratory, Steve Harón, who has been an invaluable mentor to me over the last three years. His vision and tireless efforts made this thesis possible and I learned many things from him along the way.

I am grateful to my thesis supervisor at MIT, David Staelin, for working with me throughout the thesis process.

Finally, I thank everyone in the Tactical Defense Systems Group. Special thanks to my group leader, Eli Niewood, for his support and ideas. I also want to thank Ryan Parks and Jim Clarke, in particular, for their important contributions.

Contents

List of Figures

1 Introduction

The Airborne Seeker Test Bed (ASTB) is an airborne sensor testing platform operated by the Tactical Defense Systems group at MIT Lincoln Laboratory. Shown in-flight in Figure 1.1, the ASTB is a converted Gulfstream 2 which is retrofitted to carry electronics inside its cabin, and sensor systems in pods located on the wings of the aircraft.

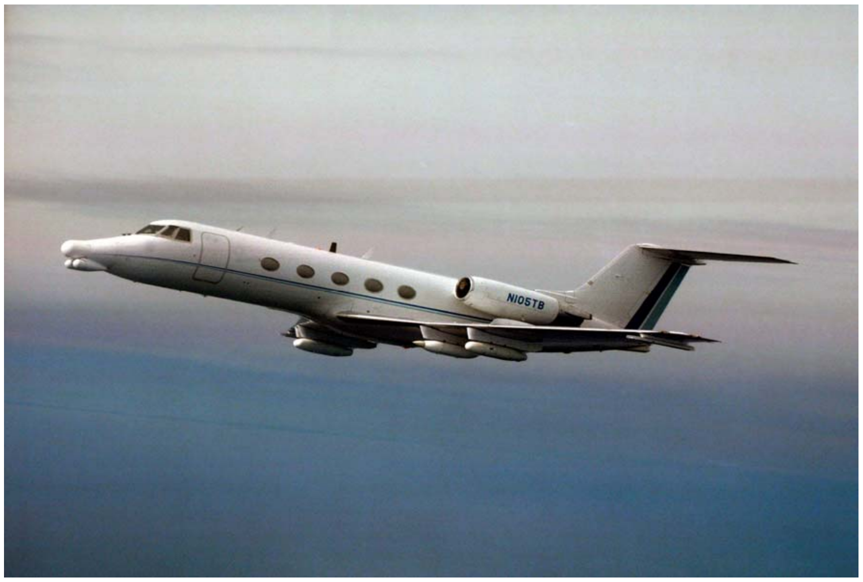


Figure 1.1 – Airborne Seeker Test Bed (ASTB)

1.1 Instrumentation Head

The Instrumentation Head (IH), located on the nose of the plane, is a primary sensor on the ASTB. The IH is a passive X-band radar receiver which consists of dual-polarized (vertically and horizontally) phased array antennas on a gimbaled mount followed by antenna electronics with receiver hardware inside the aircraft cabin. The IH is a monopulse radar system.

Monopulse describes a radar system which is capable of estimating the angle of a target from only a single measurement. By making angle measurements from only a single measurement, a monopulse radar is capable of producing more accurate estimates of angle compared to radars that estimate angle based on several measurements over time. This is because the monopulse radar is immune to variations in the signal over time that might affect angle calculations. Thus, monopulse is the preferred tracking technique when accurate angle measurements are required. [1]

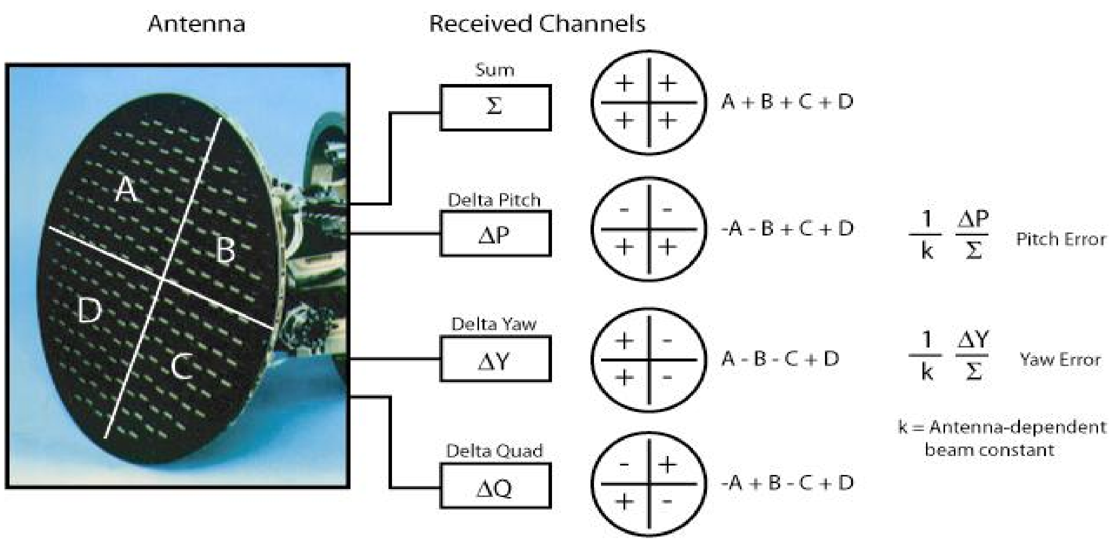


Figure 1.2 – IH Monopulse Antenna and Associated Radar Channels

Figure 1.2 shows one of the two phased array antennas from the IH (the second is mounted directly behind the first). The antenna slots on the head of the anntenna can be grouped into 4 quadrants: the top right, the top left, the bottom left, and the bottom right quadrants labeled A, B, C, and D, respectively, in Figure 1.2 and redrawn below in Figure 1.3.

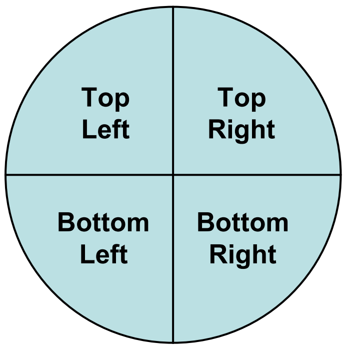


Figure 1.3 – Quadrants of the IH Antenna Head

The antenna electronics produce four radar channels: 1) the sum channel, 2) the delta pitch channel, 3) the delta yaw channel, and 4) the delta quad channel. Each of these radar channels is a linear combination of the energy received in the quadrants of the IH antenna head.

The sum channel is the sum of the energy from the array elements in all four quadrants of the antenna. The delta pitch channel is the difference in energy between the elements in the top and bottom halves of the antenna. The delta yaw channel is the difference in energy between elements in the left and right halves of the antenna. Finally, the delta quad channel is the difference in energy between the elements in the diagonal quadrants.

The linear relationship between the 4 radar channels and the 4 monopulse quadrants means that the radar channels encapsulate information about the quadrant energies. Given the 4 monopuse radar channels we can solve for the energy in each quadrant.

Figure 1.4 shows a drawing of a typical flight-test configuration with the ASTB closing in on a target of interest. A ground-based illuminator illuminates the target with an RF waveform, a portion of which reflects off of the target and is received by the IH for tracking.

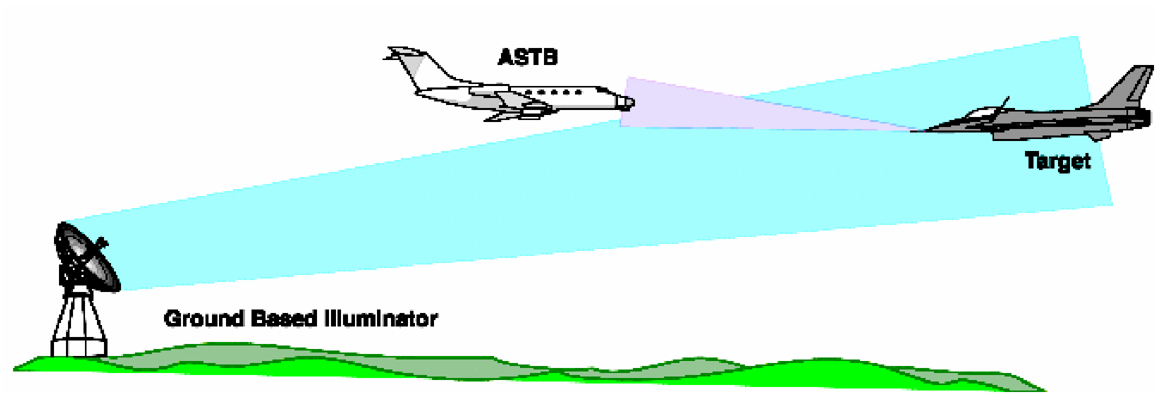


Figure 1.4 – Example of ASTB Flight-Test Configuration

The IH serves as a truth sensor for other RF systems on the test bed. As shown in Figure 1.5, signals from the IH antenna are fed to a receiver block with mixing, filtering, and amplification stages that produce radar data channels. The radar data channels are routed to a real-time processing system, a data recording system, as well as an onboard tracking system called the Seeker Computer. The Seeker Computer processes IH data in real-time to track targets in Doppler, angle, and range. From these tracks it then produces angle-feedback error signals that command the IH gimbals, keeping the IH antenna centered on the target.

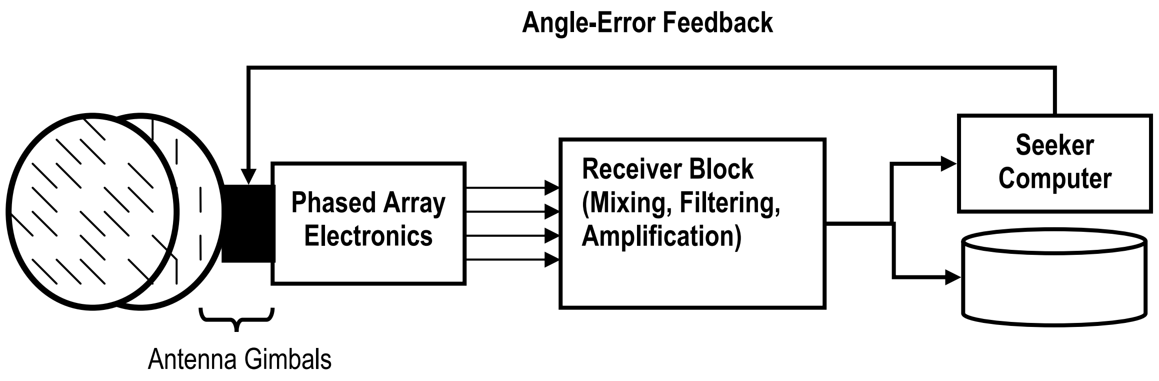


Figure 1.5 – IH System Diagram

1.2 Rebuilding the Seeker Computer

The Seeker Computer was completely rebuilt between 2002 and 2005. It replaced an old system based on obsolete hardware and software. This thesis presents a systems-level analysis of the development of the new Seeker Computer. The Seeker Computer upgrade was a team effort with substantial individual contributions made by the author. Research and development was performed in a variety of topics ranging from software design to digital signal processing (DSP). The three sections of this thesis each focus on one aspect of the upgrade process.

First, software middleware was created to support portable, object-oriented code development on the computer platform chosen to perform DSP in the Seeker Computer. A framework for creating middleware software for real-time and desktop computer systems, called TaskRunner, had previously been created at Lincoln Laboratory [2], and the implementation of this framework for the Seeker Computer was the first on a multiprocessor signal processing platform.

Second, new DSP software for the Seeker Computer was developed and tested using TaskRunner. The new software was tested against a set of existing specification requirements. The TaskRunner methodology was influential in guiding the development process.

Finally, advanced algorithms for the Seeker Computer DSP were developed to track an aircraft in angle in the presence of a towed decoy jammer. The upgrade to modern equipment produced a significant gain in processing power on the Seeker Computer. This gain in processing power opens up opportunities to extend the tracking capabilities of the Seeker Computer by using such advanced algorithms in real-time during test campaigns.

2 Mercury TaskRunner

It was decided that the software of the new Seeker Computer would be developed according to a software framework created at Lincoln Laboratory called TaskRunner. In order to use TaskRunner, a layer of middleware software, the TaskRunner Engine, must be developed for each hardware platform on which it is applied. However, a TaskRunner Engine did not yet exist for the Seeker Computer's target hardware platform, Mercury. Sections 2.1 and 2.2 describe TaskRunner, and Section 2.3 describes the principal individual contribution of the author which was the development of a TaskRunner Engine for Mercury.

2.1 TaskRunner Concept

TaskRunner is a general software framework for building portable, object-oriented software. TaskRunner software is independent of the operational, or target, hardware and its operating system. The TaskRunner framework was developed at Lincoln Laboratory by Louis Hebert in 2000 and has since been put to use on several software projects by the Tactical Defense Systems Group at Lincoln Laboratory.

The initial motivation for developing TaskRunner was to give real-time programmers the ability to develop more robust real-time applications in C. C is a low-level development language widely-used for building real-time applications because of its computational efficiency. For this reason, it is the one development language supported by nearly all real-time operating systems.

However, unlike modern programming languages such as C++ and Java, C was designed to be a procedural programming language, not an object-oriented programming language. Therefore, it does not support commonly-practiced computer science principles that derive from object-oriented programming, such as abstraction, encapsulation, and polymorphism. Furthermore, it also lacks portability when used to build real-time applications.

Real-time software developed in C is hardware-specific and is not portable because real-time hardware platforms run unique, custom-built operating systems. As a result, a substantial portion of the real-time software developed in C is specific to the target hardware. For example, various hardware systems may rely on different initialization procedures, they might have different rules for handling memory, and they may have different protocols for handling communication between processes.

Programs for real-time applications rely heavily on these hardware specific characteristics because they must meet strict timing requirements. Real-time programs typically use methods such as alarms, interrupts, and message passing for purposes of synchronization. However, these methods are not a standard part of the C language and vary in availability depending on the operating system being used. Furthermore, the function calls and processes needed to set these systems up vary widely. Therefore real-time C code developed for one operating system is not easily ported to run on other computer platforms.

Despite these limitations, C is still very popular for real-time applications. It is generally believed that the efficiency benefits of C outweigh its shortcomings. For this reason, most real-time hardware vendors and real-time software developers usually default to C and support for other languages in real-time hardware systems is limited.

The motivation for TaskRunner was the vision that basic object-oriented capabilities could be added to C while still preserving much of the efficiency required for real-time use. Secondly, TaskRunner would make real-time C code more portable. An application written for one real-time system would also work on other real-time systems. In fact the application could work on all systems, including standard desktop systems, without having to go through the time-consuming process of customizing the code to make it work. In Chapter 3 we will see how this proves especially useful in the testing phase of the design process.

In order to use TaskRunner on a given operating system, a custom-built TaskRunner Engine needs to exist for that operating system. The TaskRunner framework is based on the TaskRunner Application Programming Interface (API), a set of methods which are guaranteed to perform specific functions as set forth in the documentation. The TaskRunner Engine guarantees that the functionality of the TaskRunner API will be met for the operating system to which it was built. Similarly, each TaskRunner Task must satisfy the TaskRunner API so that it can interact with the TaskRunner Engine.

This concept is very similar to the concept behind Java, where the Java Virtual Machines on various operating systems enable Java programs to run. The Java Virtual Machines support an Application Programming Interface (API) which Java programs reference.

By 2002, the TaskRunner framework had been successfully implemented on a number of operating systems including Windows, UNIX, Linux, and pSOS. pSOS is a real-time operating system used on Motorola single board computers. Windows, UNIX, and Linux are operating systems for standard desktop systems which are not real-time.

In the typical real-time development environment, there are usually relatively few real-time systems compared to developers. There may be only one real-time system which is deployed in the field, one which is kept as backup, and one which is used for testing in the laboratory. However, standard desktop systems have become commodity systems which are abundant. Thus, the benefit to having TaskRunner implementations for non-real-time operating systems is that a larger number of programmers can have access to a development platform with which to develop real-time applications. Before TaskRunner, development of real-time applications was limited to programmers who had access to the real-time systems.

In 2002, a multiprocessor computer system built by Mercury Computer Systems was chosen as the hardware platform for the new Seeker Computer real-time radar system. The Mercury computer system ran a custom operating system called MCOS (Mercury Computers Operating System). The designers of the new real-time radar system also decided to use the TaskRunner framework since it had been successfully used for other real-time projects. However, no TaskRunner Engine existed for MCOS. Furthermore, MCOS would be the first multiprocessor real-time operating system which would support TaskRunner.

The following sections describe the development of TaskRunner for MCOS. First, we will take a look at the general TaskRunner architecture. We then see how TaskRunner was implemented for the Mercury Computers Operating System.

2.2 TaskRunner Architecture

TaskRunner has a two-tiered architecture. On the first-tier are TaskRunner Tasks. TaskRunner Tasks are analogous to classes in other object-oriented programming languages such as Java. Each TaskRunner Task is coded to support specific functionality set forth by the Task API. TaskRunner Tasks interact with other TaskRunner Tasks to achieve more complicated behavior. A TaskRunner application usually consists of a collection of interacting TaskRunner Tasks.

On the second-tier is the TaskRunner Engine. The TaskRunner Engine supports the TaskRunner API and manages the interactions between TaskRunner Tasks. The TaskRunner Engine is built once for a given operating system and the same implementation is used by all TaskRunner applications running on that operating system. Developers using TaskRunner then develop applications by constructing TaskRunner Tasks which call methods from the TaskRunner API.

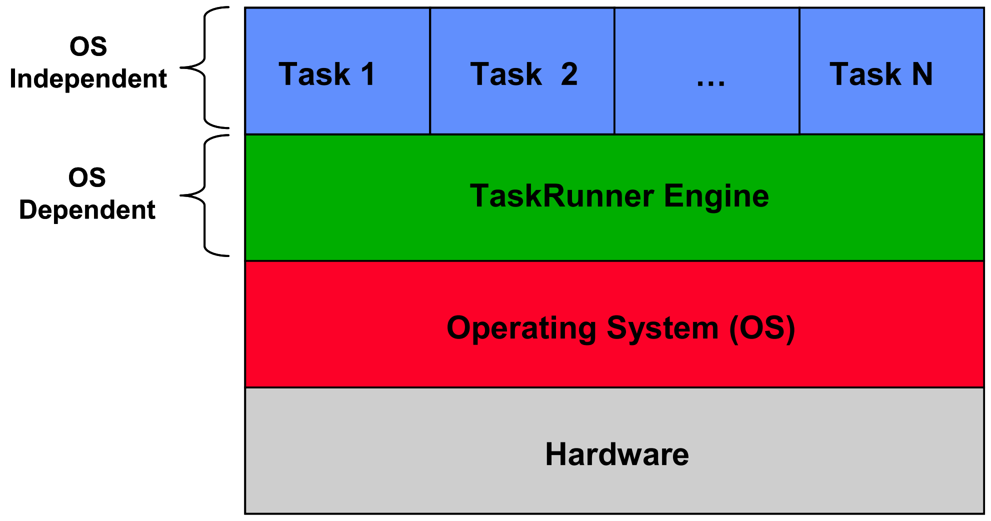


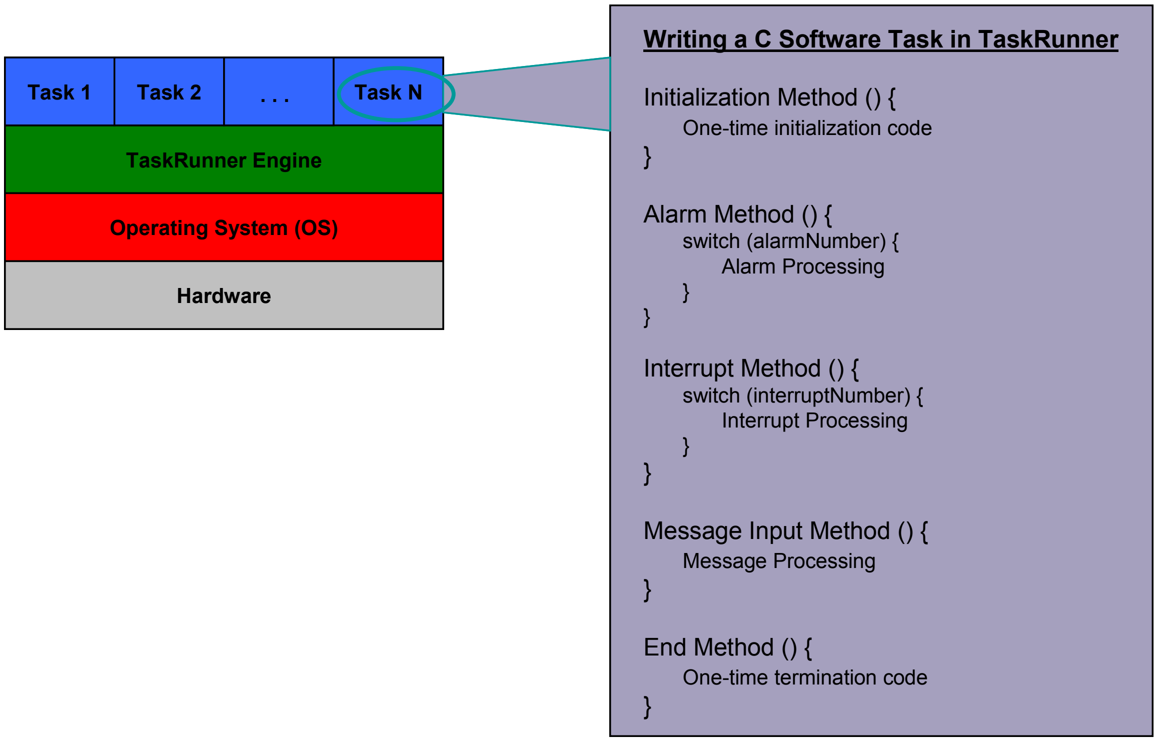
Figure 2.1 – TaskRunner Hierarchy

Figure 2.1 illustrates the TaskRunner Architecture. TaskRunner Tasks are operating system-independent. They can therefore run on any hardware/OS system for which a TaskRunner Engine exists. The TaskRunner Engine is operating system-dependent. It is built with a programming language supported by the local operating system. The operating system is usually developed by the hardware vendor specifically for use on the hardware they build. The TaskRunner Tasks and the TaskRunner Engine comprise the TaskRunner framework. The TaskRunner framework is applicable on any combination of operating system and hardware.

2.2.1 TaskRunner Tasks

Software is developed in TaskRunner as a collection of interlinked TaskRunner Tasks. A TaskRunner Task is an object-oriented software block whose behavior is fully defined in a set of internal methods predetermined by the TaskRunner API. TaskRunner Tasks are equivalent to software objects in the object-oriented programming model and are analogous to classes in C++ and in Java.

Each Task includes a Task Constructor and a set of 5 standard internal methods: 1) an initialization method, 2) an alarm method, 3) an interrupt method, 4) a message method, and 5) an end method. These method types were chosen due to their frequent use in real-time software.



Writing a C Software Task in TaskRunner

```
Initialization Method () {
      One-time initialization code
}

Alarm Method () {
      switch (alarmNumber) {
            Alarm Processing
      }
}

Interrupt Method () {
      switch (interruptNumber) {
            Interrupt Processing
      }
}

Message Input Method () {
      Message Processing
}

End Method () {
      One-time termination code
}
```

Figure 2.2 – Task Structure

The Task structure described above is illustrated in Figure 2.2. The internal methods define how Task objects react to input and produce output. The TaskRunner Engine facilitates communications between Tasks by calling a Task's associated internal method in the event of a message, alarm, or interrupt sent from another Task. It also calls a Task's methods when the Task is created or destroyed.

The Task Constructor is called by the TaskRunner Engine at the start of a TaskRunner application. It incorporates the Task into the TaskRunner environment.

The initialization method is called immediately after the Task's construction. In the initialization method, the Task typically initializes variables, allocates shared memory, initializes interrupts which it will be sending, registers for interrupts which it will receive, and sets alarms as it prepares to run.

The alarm method is called in the event that a Task's alarm goes off. Tasks have the ability to set alarms for themselves using the TaskRunner API. The alarm method includes a parameter for alarm number so that tasks can differentiate between different alarm types.

The interrupt method is called in the event that an interrupt the task has registered for is triggered. As is the case with the alarm method, the interrupt includes a parameter for interrupt number so that tasks can differentiate between different interrupt types.

The message method is called when another Task in the TaskRunner environment sends data to the Task via the TaskRunner API. The receiving task is passed a pointer to the block of data which was sent as well as a parameter specifying the size of that block of data.

Finally, the end function is called in the event that the task is explicitly closed, or when the TaskRunner simulation ends. The end function can include a variety of clean-up procedures as required for the given Task object.

By defining the Task Constructor and the 5 standard internal methods, a Task writer will have fully defined the behavior of a TaskRunner Task.

Conceptually, Tasks can be thought of as black boxes whose interactions with other Tasks are facilitated by the TaskRunner Engine. Tasks take input and produce output while the specifics of their internal methods are hidden from other objects in the TaskRunner environment. In addition, each Task has its own space in memory which is inaccessible by other Tasks. Thus, Tasks are able to keep track of their own state. Furthermore, Tasks can be instantiated several times in a TaskRunner application. Each instantiation of a Task runs independently from the other instantiations. Figure 2.3 illustrates a TaskRunner Application with two Tasks.

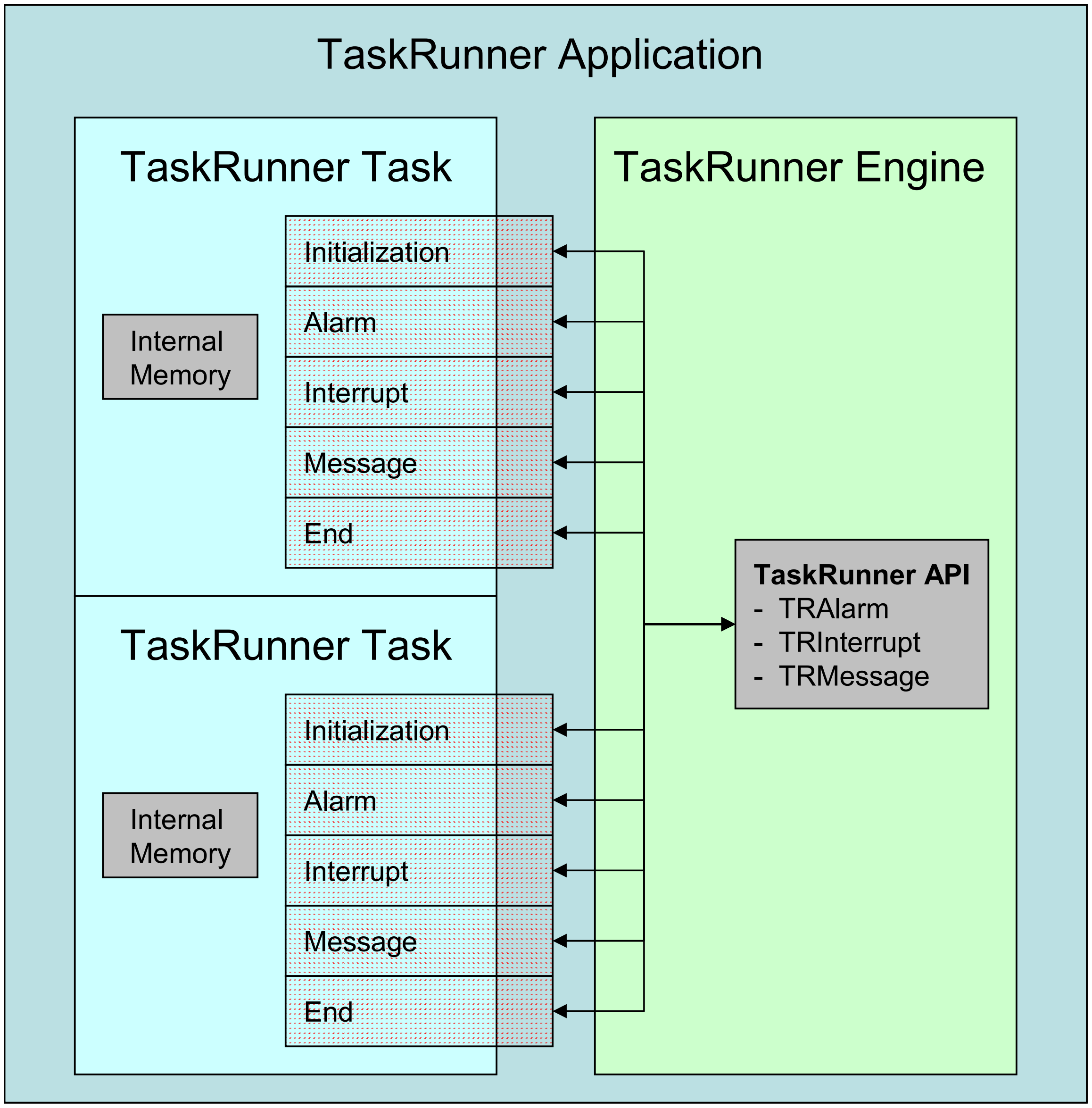


Figure 2.3 – TaskRunner Application

TaskRunner Tasks interact with each other by sending signals and messages which are received by other Tasks in the TaskRunner environment. In this way, a set of Tasks can be linked in a variety of ways to achieve a range of behavior. Once a Task is created, it can be run on any operating system which has a TaskRunner Engine. Thus, TaskRunner Tasks have the property of being portable across multiple platforms.

2.2.2 TaskRunner Engine

The TaskRunner Engine is the foundation of all TaskRunner applications. An operating system must have a TaskRunner Engine in order to support TaskRunner Tasks. During runtime the TaskRunner Engine manages the lifecycle of a TaskRunner application. The

TaskRunner Engine launches Tasks and runs their construction and initialization methods. It sets up communication pathways between Tasks, and, when necessary, the TaskRunner Engine activates the appropriate internal methods of each Task.

The TaskRunner API provides methods for Tasks to do a variety of things such as set alarms, register for interrupts, send interrupts, send messages, allocate shared memory, and access shared memory. The complete TaskRunner API is included in the appendix. Using the TaskRunner Engine API frees the Task developer to think more about software design rather than about how to handle event-driven action within a specific system.

By sitting between the target operating system and the Task layer, the TaskRunner Engine hides hardware and operating system-specific details such as memory management and signaling so that Tasks written to the TaskRunner API can be built and run for any computer system for which a TaskRunner Engine exists.

There are no restrictions on how a TaskRunner Engine should be implemented other than that it satisfies the functionality guaranteed by the TaskRunner Engine API. Thus, TaskRunner Engines can be designed to take advantage of any features in a given operating system or hardware platform.

2.3 Mercury TaskRunner Engine

One of the design specifications for the DSP software being developed for the Seeker Computer upgrade was that it be developed in accordance with the TaskRunner API. However, a TaskRunner implementation did not yet exist for the hardware and OS chosen to host the Seeker Computer DSP system, the Mercury Computer Systems multiprocessor digital signal processing platform and the associated MCOS operating system. As a result of this design choice, a new TaskRunner Engine needed to be created for the Mercury platform in order to support software written to the TaskRunner framework.

2.3.1 Mercury Computer Systems

Mercury Computer Systems is a vendor of parallel processing boards that host multiple CPU's connected via RACEway. RACEway is a high-speed data bus interface developed by Mercury and accepted as an IEEE standard. Data transfers over the RACEway on Mercury systems are specified to reach speeds of 160 MB/s. Figure 2.4 shows a Mercury motherboard hosting 4 PowerPC (PPC) processors, a typical configuration used for these systems. To increase processing capability, Mercury computing systems are easily scalable by adding motherboards that carry more CPU's. The Mercury platform was chosen to support signal processing in the Seeker Computer because of the need to process multiple channels of radar data in real time, which can be supported using the parallel processing capability that Mercury provides. Processing of channels can be split between the CPU's, and the scalability of a Mercury system enables processing of additional radar channels in the future as may be necessary.

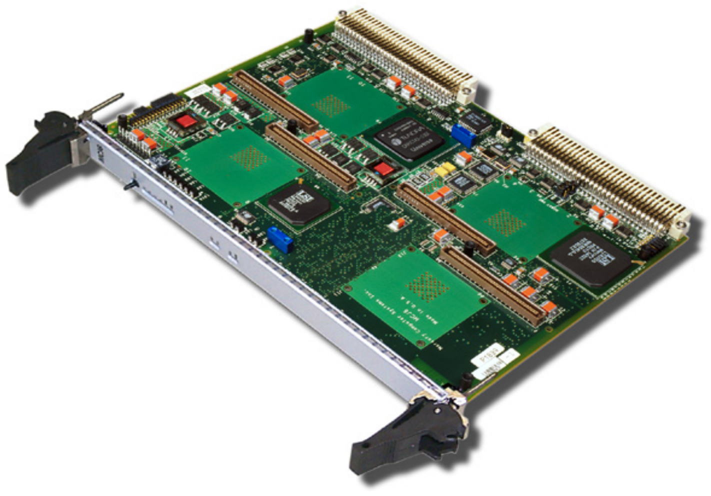


Figure 2.4 – Mercury Motherboard with 4 PowerPC Processors

2.3.2 Implementation Overview

The TaskRunner Engine created for Mercury was the first implementation of TaskRunner on a parallel computing system. TaskRunner Engines had previously been developed for Windows, Linux, UNIX, and pSOS, all operating systems designed for single-processor computing systems. The challenge in developing a TaskRunner Engine for Mercury was figuring out if and how TaskRunner could take advantage of multiple processors to improve performance while still supporting the complete TaskRunner API in real-time.

Parallel computing systems are useful when a program can be broken down into independent blocks, distributed over the multiple computers, and processed in parallel. When this is possible, a parallel computing system will provide reduced computation time over the analogous single-processor system.

TaskRunner Tasks are independent blocks which are naturally suitable for parallel computing. As illustrated in figure 2.5, the TaskRunner Engine on Mercury distributes TaskRunner Tasks across all available processors in an attempt to achieve efficient use of processing power. However, after distributing Tasks across multiple processors, Tasks on different processors no longer have a direct pathway via which to communicate. The challenge in creating the Mercury TaskRunner Engine was thus to use processing power efficiently while ensuring that Tasks would still be able to communicate via the TaskRunner API over multiple processors.

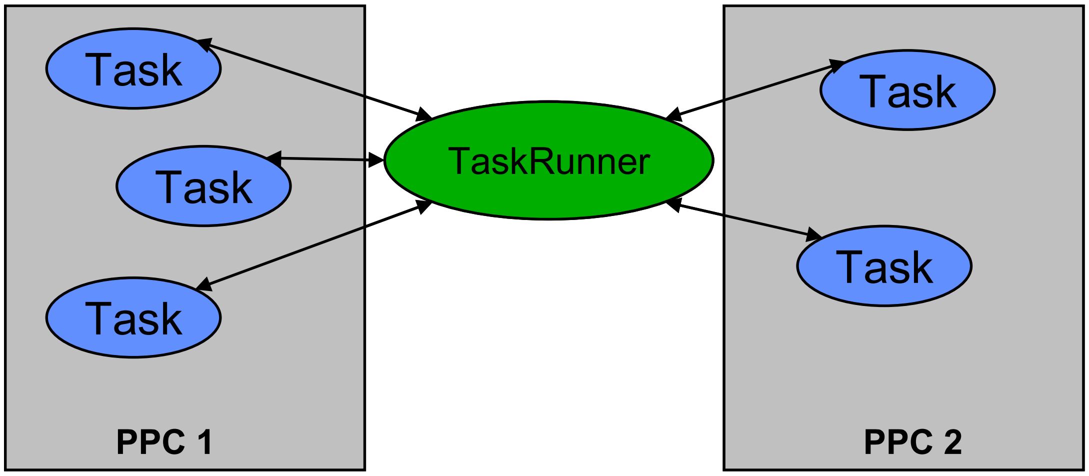


Figure 2.5 – TaskRunner Architecture on Mercury Platform

Fortunately, the Mercury operating system, MCOS, has built-in functions which mirror several of TaskRunner's communication functions across multiple processors, including alarm-handling, interrupt-handling, and signal-handling which is useful for basic synchronization. These built-in functions were used directly by the TaskRunner Engine to provide the same services for TaskRunner Tasks. However, MCOS does not have a built-in function for message handling. As a result, a message routing system needed to be designed and built for the TaskRunner Engine.

2.3.3 Message Router

The function of the message router is to take data from a Task running on one processor and route it to any other Task, whether that Task be running on the same processor or another processor. Furthermore, the message router needs to be efficient enough to meet real-time requirements.

There are two ways of transferring data on a Mercury computer system: 1) local transfers, and 2) RACEway transfers. Local transfers can be made between any two regions of memory. Those two regions of memory can lie on the same processor or on different processors. However, local transfers are slow when made between regions of memory that lie on different processors and will not support real-time performance. Local transfers between regions of memory on the same processor are fast enough to support real-time performance. The RACEway transfer is a special feature of the Mercury computer system. RACEway transfers move data at high speeds between any two processors over the RACEway bus. However, RACEway transfers can only access data which lies in special sections of shared memory on a processor, which must be pre-allocated. Furthermore, Mercury limits the number of special sections of shared-memory which can exist on each processor.

The implemented high-speed message router uses a combination of local transfers and RACEway transfers. The result is an efficient mechanism of passing data between Tasks that meets real-time requirements.

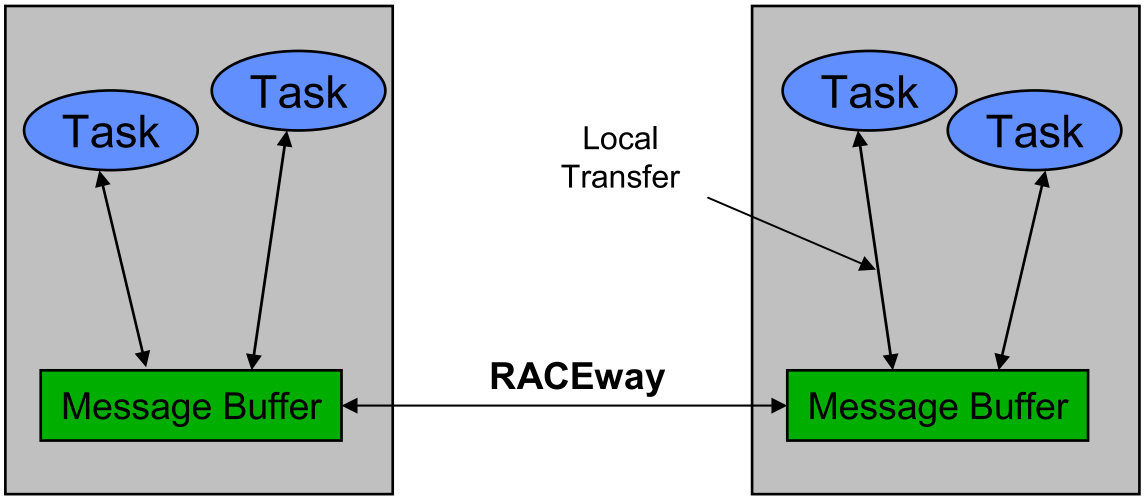


Figure 2.6 – Message Passing by Local and RACEway Transfers

Figure 2.6 illustrates the message passing procedure of the implemented Message Router. For each processor, one section of shared memory is pre-allocated and dedicated for Message Router use. The shared memory on each processor dedicated to the Message Router is called a Message Buffer because it is a temporary storage location for messages in transit.

The Message Router operates like a post office. The sending-Task alerts the Message Router that it would like to send a message to a receiving-Task. The sending-Task then hands the message to the Message Router. The Message Router uses a local transfer to move the message from the sending-Task's private memory to the local Message Buffer which lies on the same PPC as the sending-Task. If the receiving-Task lies on a PPC which is different from the PPC on which the sending-Task lies, the Message Router then uses a RACEway transfer to move the message from the local Message Buffer of the sending-Task to the local Message Buffer of the receiving-Task. The Message Router then notifies the receiving-Task that it has a message. When the receiving-Task is ready to receive its message, the Message Router hands the message off to the receiving-Task by using a local transfer to move the message from the Message Buffer to the Task's private memory.

This Message Router implementation successfully and efficiently facilitates message passing between any two Tasks in the system.

2.3.3.1 Message Router Performance

The implemented Message Router transfers data at speeds of up to 264 MB/s between Tasks on the same PPC, and at speeds of up to 125 MB/s between Tasks on different PPCs. Transfer rates vary as a function of the amount of data being transferred. The time of each transfer is dependent on a fixed setup cost to prepare the transfer route and variable transfer costs which depend on the transfer size.

Figure 2.7 plots the Message Router's intra-PPC transfer rate as a function of transfer size. As described in the previous section, intra-PPC message transfers are performed entirely via Mercury local transfers.

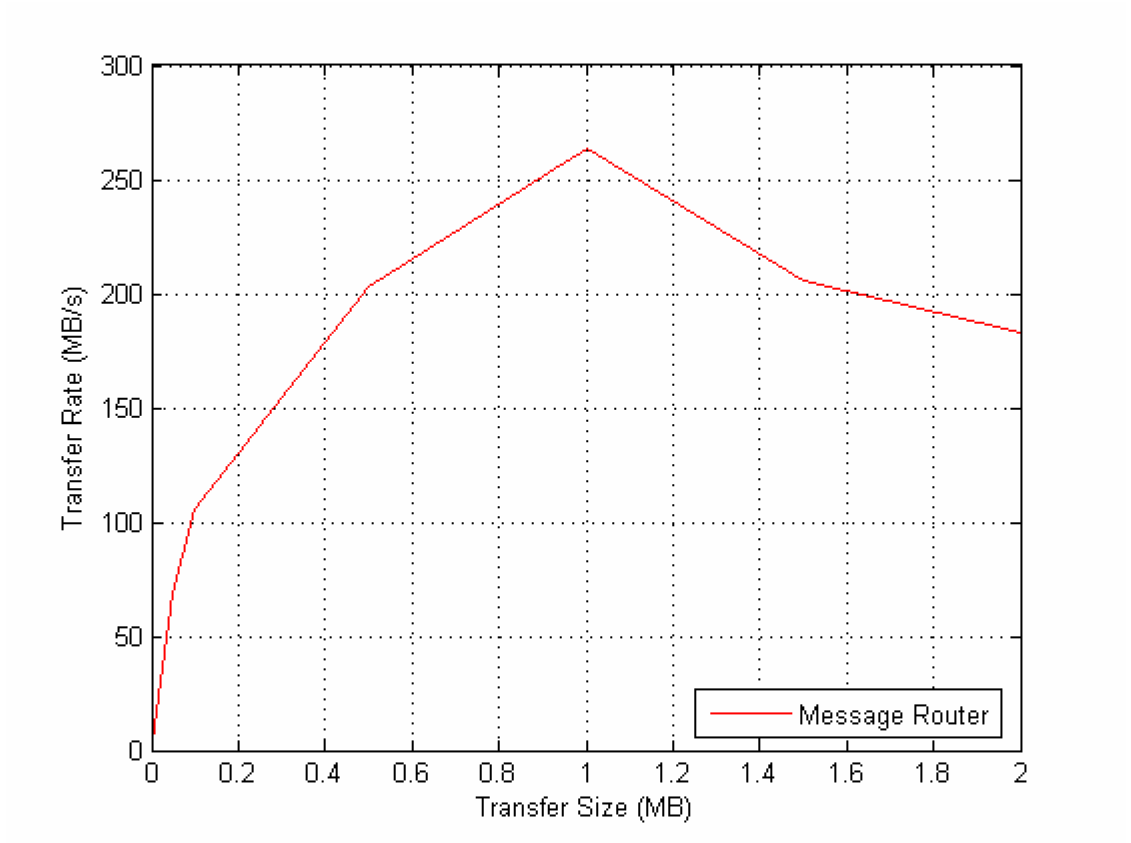


Figure 2.7 – Same-PPC Message Transfer Performance

Figure 2.8 plots the Message Router's cross-PPC transfer rate as a function of transfer size. To compare its performance, the transfer rate of a RACEway transfer as a function of transfer size is plotted in blue. While the Message Router's fixed overhead costs affects the transfer rate for small transfer sizes, these fixed costs are averaged out for large transfers and the performance of the Message Router approaches the performance of a pure RACEway transfer.

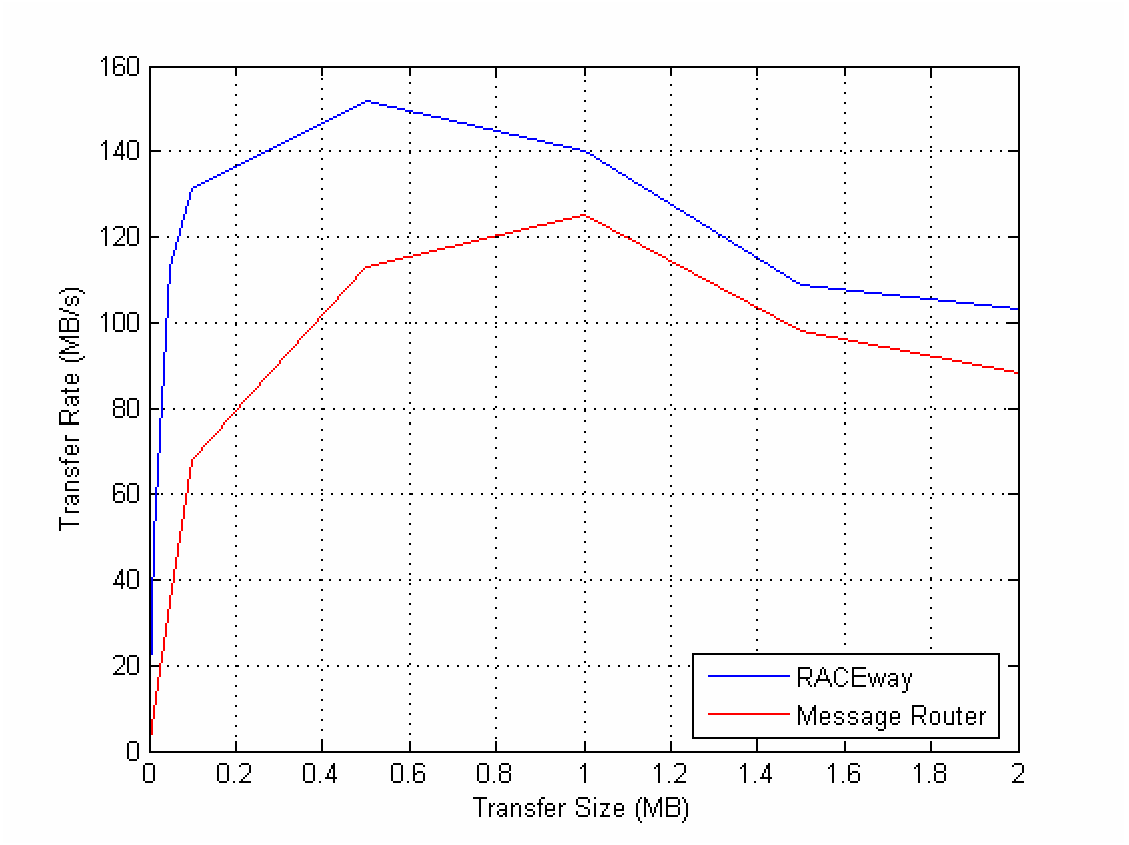


Figure 2.8 – Cross-PPC Message Transfer Performance

2.3.4 Design

Figure 2.9 is a block diagram showing the final design of the TaskRunner Engine on Mercury. The TaskRunner Engine supports the full TaskRunner Engine API including the methods TRAlarm, TRInterrupt, and TRMessage. These methods and others are built upon the extensive signaling capabilities of MCOS. In addition, the TaskRunner Engine manages a signal queue for each Task. A Task's signal queue stores pending signals which have been sent to that Task. Signals are serviced when the TaskRunner pops a signal from the queue and calls the Task's corresponding handling-method from the Task API.

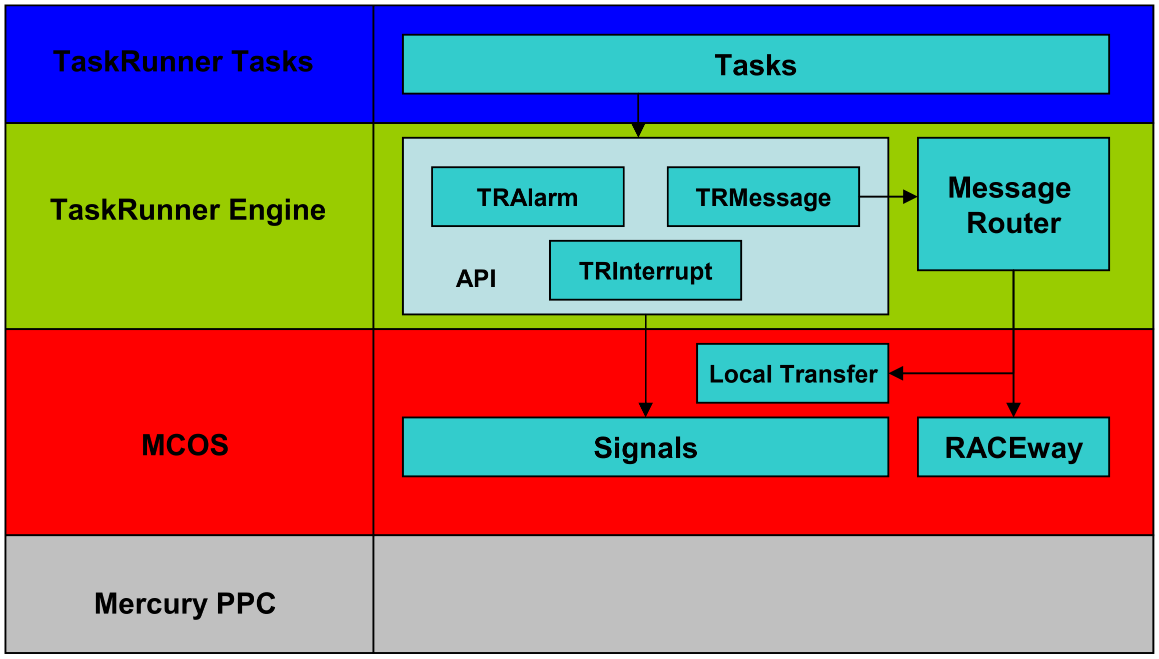


Figure 2.9 – Mercury TaskRunner Design

For example, when a Task interrupts another Task via the TRInterrupt command, the TaskEngine sends an interrupt signal to the receiving-Task on behalf of the sending-Task. The interrupt signal identifies the sending-Task and the time the interrupt was sent. It is placed in the receiving-Task's signal queue and waits to be serviced. When the interrupt signal reaches the top of the queue, the TaskRunner Engine pops the signal off of the queue and triggers the Interrupt Method in the receiving-Task's API.

Calls to the TRMessage command require more than signals to be exchanged. Blocks of data must be sent from Task to Task and, in addition to signals, the TaskRunner engine relies on the high-speed message router described in the previous section.

2.3.5 Mercury TaskRunner Engine Performance

The completed Mercury TaskRunner Engine implements the full TaskRunner API. Aside from the message transfer function whose execution time depends on the message size, the execution time for all other communication functions in the API is less than .01 ms. This is very little overhead and is more than suitable for real-time use.

There are two limitations to the Mercury TaskRunner Engine. These limitations are associated with the Mercury platform and fortunately they do not greatly affect the Mercury TaskRunner's usability. First, MCOS limits the frequency of alarms to 1000 Hz. Second, the Mercury TaskRunner Engine can support up to 13 Tasks per PPC due to constraints in processor memory. If additional Tasks are needed for an application, PPCs can be added to the system.

3 Development of the Seeker Computer DSP

The Seeker Computer Digital Signal Processor (SCDSP), the component of the Seeker Computer which processes IH data in real-time, was rebuilt completely with software written in TaskRunner. An existing specification created by the developers of the original SCDSP outlined the exact behavior of the system, and the new implementation was required to meet that specification. For purposes of validation, an independent simulation environment was developed to verify that the new SCDSP running on Mercury met the specifications.

Taking advantage of existing TaskRunner implementations and the new TaskRunner implementation for Mercury, development was carried out on Mercury as well as a combination of desktop systems. The resulting SCDSP was successfully tested within the simulation environment on Mercury.

Section 3.1 describes the overall development and testing methodology. Section 3.2 describes the SCDSP. Sections 3.3 and 3.4 describe the development of the simulation environment and the testing process which were the primary individual contributions by the author to the work in this chapter. Section 3.5 evaluates the new SCDSP which was built.

3.1 Development and Testing Methodology

Development of the simulation environment was completely separate from development of the SCDSP. No code was shared between the development team working on the simulation environment and the development team working on the SCDSP. This methodology allowed the simulation environment to give an independent evaluation of the SCDSP's performance. To be validated, the SCDSP had to run successfully within the simulation environment on the Mercury platform.

However, while separate teams were tasked to work on the simulation environment and the SCDSP, only one Mercury system was available for development. This posed a problem since both the SCDSP and the simulation environment had to eventually run on the Mercury platform.

Due to this limitation in hardware availability, TaskRunner was crucial in applying black-box testing. The team building the SCDSP used Mercury as their platform for development, and the team building the simulation environment used a combination of desktop systems as their platform for development. Since both teams developed their software according to the TaskRunner standard, the finished applications ran compatibly on the Mercury platform without problems.

3.2 Seeker Computer Digital Signal Processor

As shown in Figure 3.1, the IH is comprised of the IH Receiver and the Seeker Computer.

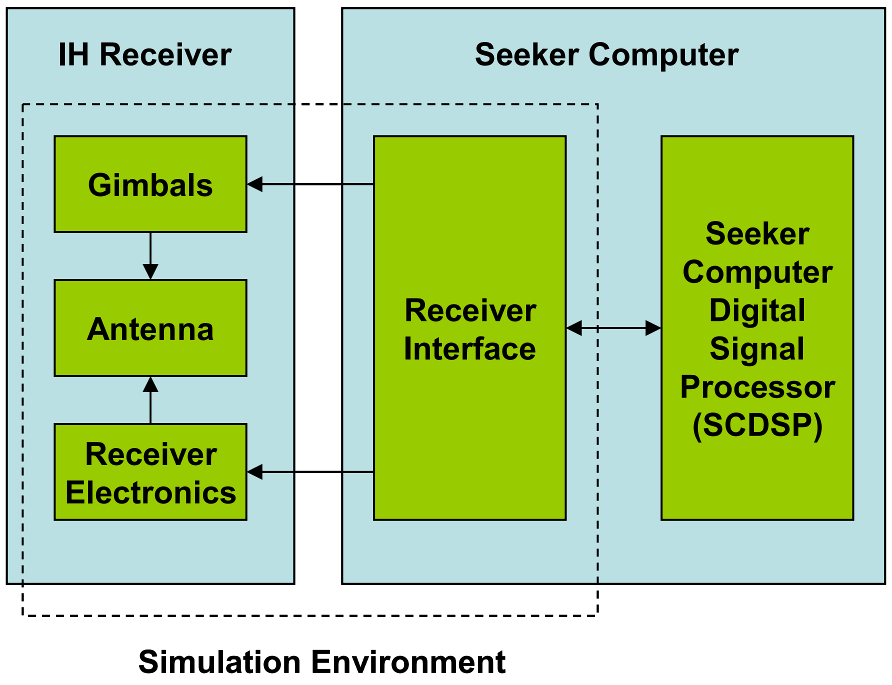


Figure 3.1 – Components of the Instrumentation Head

The IH Receiver converts the received analog radar signal into a digital signal. The components of the IH Receiver include the monopulse antenna, the mechanical gimbals, and the receiver hardware in the back-end. The monopulse antenna captures radar return and is mounted on the mechanical gimbals. The gimbals control the orientation of the antenna. The signal from the monopulse antenna is fed to the receiver hardware which filters, shifts, and samples the X-band signal. The IH Receiver takes a guidance signal from the Seeker Computer which directs the receiver gimbals.

The Seeker Computer processes the digital signal from the IH Receiver to extract targets and produce signal statistics. The Seeker Computer consists of the radar interface and the SCDSP. The digital signal from the IH Receiver is passed to the SCDSP via the radar interface. The SCDSP performs signal processing on the digital signal and calculates the angle error of the target. After processing, the SCDSP returns the calculated angle error as a guidance signal to the IH Receiver via the radar interface.

A block diagram of the SCDSP is shown in Figure 3.2. The SCDSP takes monopulse channels and a command message as input from the radar interface. Depending on the command message, the SCDSP operates in either calibration mode, acquisition mode, or track mode. It processes the monopulse channels in the time domain and in the frequency domain according to the settings in the command message. The SCDSP then outputs either a calibration report, an acquisition report, or a track report depending on which mode it was operating in. The track report contains time domain and frequency domain statistics that indicate the angle error, Doppler, and range of the target.

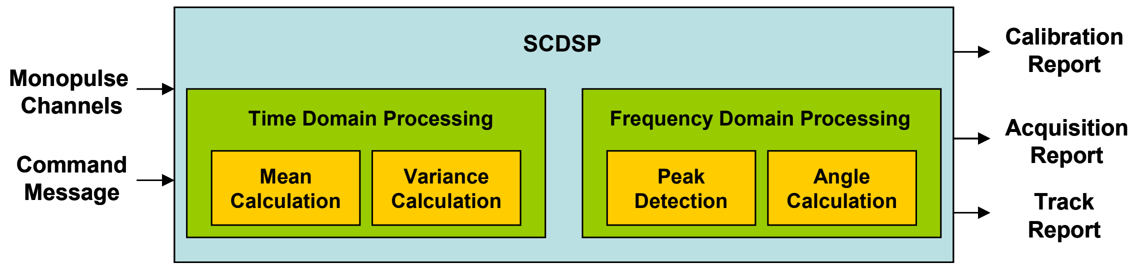


Figure 3.2 – SCDSP Block Diagram

3.2.1 Command Messages

The SCDSP receives two types of command messages: 1) the initialization message, and 2) the dwell message. Depending on the command message, the SCDSP adjusts its processing of the monopulse channels and produces different sets of output reports.

The SCDSP begins operation when it receives an initialization message. The SCDSP sets internal variables based on parameters in the initialization message. The initialization message defines parameters which describe the monopulse input channels and how the SCDSP should access them. The SCDSP uses the initialization message to determine which data channel corresponds to the sum channel, the delta pitch channel, the delta yaw channel and the delta quad channel. Other parameters include the number of sample points in each input channel, the length of FFT which should be taken, the block size to be used for processing, the underlying waveform of the illumination (either continuous or pulsed), and variables to be used for threshold detection.

Shortly after initialization, the SCDSP receives dwell messages at intervals of time called dwells. The dwell message instructs the SCDSP on how to process the monopulse input channels during each time interval. In each dwell, the SCDSP operates in one of three processing modes: 1) calibration, 2) acquisition, or 3) track.

The SCDSP is put into calibration mode to calculate weighting coefficients used to scale receiver data to account for imbalances between receiver channels. A controlled test signal is fed into the monopulse antenna head and the SCDSP prepares a Calibration Report with statistics on the test signal which are verified with the known values.

The SCDSP is put in acquisition mode when the IH attempts to acquire targets. The SCDSP is told to look for potential targets in the return and to prepare an Acquisition Report. In the Acquisition Report, the SCDSP identifies up to 7 targets in the return as determined by peak detection in the frequency domain. The Acquisition Report specifies the FFT bin location and magnitude of each target candidate as well as its pitch error and yaw error as calculated using monopulse amplitude comparison.

The SCDSP is put in track mode when the IH is in the process of tracking a target. Via the dwell message the SCDSP is passed the last known FFT bin location of the target being tracked and it is told to prepare a Track Report with updated tracking information. There are two components to the Track Report: 1) the Time Domain Track Report, and 2) the Frequency Domain Track Report.

The Time Domain Track Report contains the results of track-mode processing in the time domain. The report includes the mean and variance of peak samples in the time domain signal, the mean and variance of the pitch error, and the mean and variance of the yaw error. It also gives the mean magnitude, the mean pitch error, and the mean yaw error of sequential sub-blocks of the time domain signal whose size was specified in the initialization message.

The Frequency Domain Track Report contains the results of track-mode processing in the frequency domain. The report gives the magnitude, the pitch error, and the yaw error of the target FFT bin specified in the dwell message and the 3 FFT bins on each side of the target FFT bin. The report also gives the threshold of the noise floor.

In tracking mode, the radar interface takes the Frequency Domain Track Report and identifies the FFT bin with the largest magnitude as the location of the target. If the magnitude of the signal in that FFT bin location is greater than a threshold determined from constant false-alarm rate (CFAR) processing, the pitch error and the yaw error associated with that FFT bin is used as a guidance signal. [1] In the next dwell the dwell message is updated with the newly identified target FFT bin for the next tracking iteration. If the magnitude of the signal in the identified FFT bin is less than the CFAR threshold, track of the target is has been lost and the SCDSP is put back into acquisition mode to find another target in the next dwell.

3.2.2 Angle Error

The most important calculation which the SCDSP performs is of target angle error calculated from the monopulse data channels. The construction of the monopulse data channels was described in Chapter 1. The SCDSP determines target angle errors using monopulse amplitude comparison, comparing signal strengths at different locations of the antenna head to determine angle.

When a signal is emitted from above the boresight of the monopulse antenna, the signal received in the top half of the antenna will be stronger than the signal received in the bottom half of the antenna. The greater the angle between the source of the signal and the antenna boresight, the greater this signal difference will be. The reverse is true for signals which are emitted from below the boresight of the monopulse antenna. Thus, the delta pitch channel, which is the energy received by the top half of the antenna minus the energy received by the bottom half of the antenna, will be positive for targets which lie above the antenna boresight and negative for targets which lie below the antenna boresight.

Similarly, a signal emitted from the right of the antenna will cause the energy received by the right half of the antenna to be greater than the energy received by the left half of the antenna. The more a signal lies to the right of the antenna, the greater this difference in energy will be. The reverse can be said of a signal emitted from the left of the antenna. Thus, the delta yaw channel, which is the energy received by the right half of the antenna minus the energy received by the left half of the antenna, will be positive for targets which lie to the right of the antenna and negative for targets which lie to the left of the antenna.

By analyzing the monopulse difference channels, ΔP and ΔY, the SCDSP determines in which quadrant a target lies. Equation 3.1 is used to calculate pitch angle error, and Equation 3.2 is used to calculate yaw angle error.

$$\text{Pitch Error} = \frac{\Delta P}{\Sigma} \tag{3.1}$$

$$\text{Yaw Error} = \frac{\Delta Y}{\Sigma} \tag{3.2}$$

The difference channels are normalized by the sum channel, ∑, to get an angle error measurement which varies monotonically with the actual angle of the target.

One important point is that while there is a one-to-one mapping between the angle error and the actual angle of the target, this mapping is not necessarily linear. The actual angle of the target is the geometrical angle between the incoming signal and the antenna's boresight line. The angle error is a measurement that is used to guide the gimbal system in order to keep the antenna centered on the target. The only guarantee is that the angle error is expected to vary monotonically with the actual angle. A larger angle will result in a larger angle error and vice-versa. The true mapping depends on the specific monopulse antenna in use.

3.3 Simulation Environment for the SCDSP

The simulation environment tests the SCDSP as a black box, simulating monopulse inputs and command messages for input and verifying that the output reports are consistent with the injected inputs. The rules for exchanging inputs and outputs between the SCDSP and the radar interface are very specific on the real system and the simulation environment follows the same message structure, memory structure, and signaling protocol for this purpose.

3.3.1 Generating Inputs

The simulation environment generates sample data that resembles what the IH receiver might produce on a typical ASTB flight given a fixed geometry between the ground-based transmitter, the IH receiver, and any targets present. While the ASTB usually operates in situations where there is only target present, the simulation can produce multiple targets by calculating the return from individual targets and then summing to produce the total return. Figure 3.3 illustrates the signal generation process for one target.

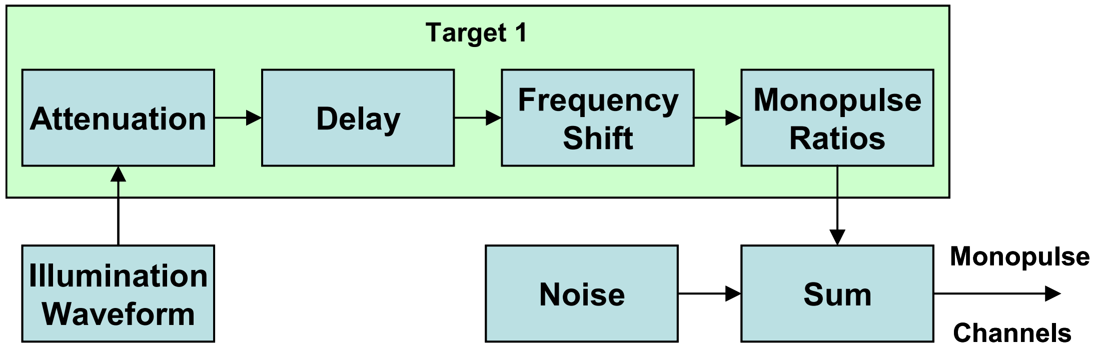


Figure 3.3 – Generating Simulated Data for One Target

Construction of simulated data begins with the illumination waveform. The ground-based transmitter provides either continuous or pulsed Doppler illumination. The continuous waveform is defined by its underlying carrier frequency. The pulsed Doppler waveform is defined by its underlying carrier frequency, its pulse repetition interval (PRI), and its duty cycle, which determines pulse width.

Attenuation between the transmitted signal and the received radar signal is calculated by the bi-static radar equation. This equation encapsulates characteristics of the transmitter, the target, and the receiver to estimate the power which is received as a ratio of the power which is transmitted. [3]

To incorporate the effects of range, the sum is taken of the distance between the transmitter and the target and the distance between the target and the receiver to get the total distance traveled by the illumination before it reaches the receiver. Since the illumination waveform travels at the speed of light, a delay is calculated directly from the distance traveled and injected into the signal.

Given that the velocity of the targets is fixed, a frequency shift is introduced into the received signal to simulate the Doppler Effect.

To capture the angle of a target, the simulation scales the received energy accordingly to produce sum and difference monopulse channels.

Finally, white noise is injected at the receiver to encapsulate any random effects.

3.4 Design and Integration Using TaskRunner

Development and testing of the SCDSP occurred in three phases. In Phase 1, the SCDSP and the Simulation Environment were developed concurrently using the TaskRunner framework. In Phase 2, the SCDSP was tested with the Simulation Environment on a desktop system. In Phase 3, the SCDSP was tested with the Simulation Environment on the Mercury system. Successful operation on the Mercury system completed the development and validation of the SCDSP.

Figure 3.4 depicts the initial development of the SCDSP and the Simulation Environment as object-oriented Task applications. The SCDSP was developed directly on the Mercury system while the Simulation Environment was developed on a desktop system.

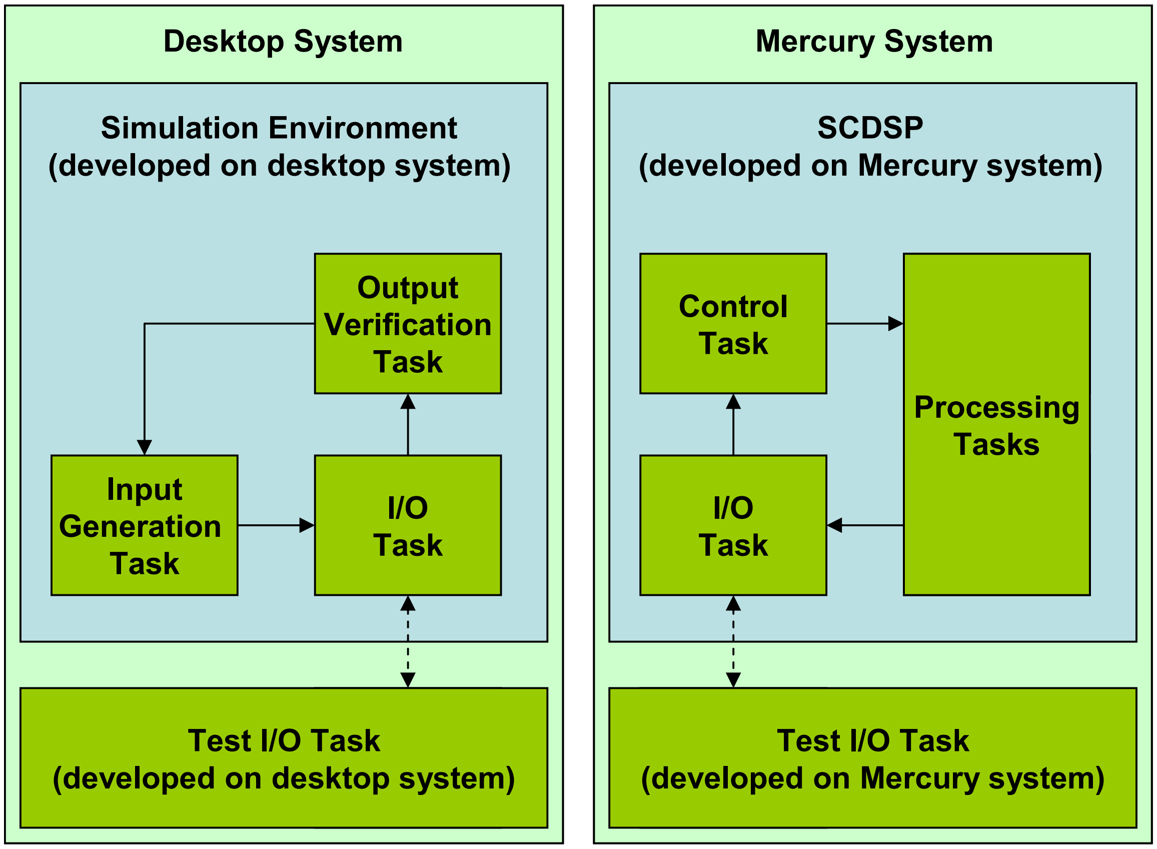

Desktop System
Simulation Environment (developed on desktop system)
Output Verification Task
Input Generation Task
I/O Task
Test I/O Task (developed on desktop system)
Mercury System
SCDSP (developed on Mercury system)
Control Task
Processing Tasks
I/O Task
Test I/O Task (developed on Mercury system)

Figure 3.4 – Concurrent Development of the SCDSP and Simulation

The SCDSP's design consists of an I/O Task, a Control Task, and a group of Processing Tasks. The I/O Task is responsible for the SCDSP's input and output with the external system, following the communication protocol described in the specification. When the I/O Task is triggered by new input, it passes the received monopulse channel data and associated command message to the Control Task. The Control Task interprets the command message and initializes processing by the Processing Tasks. The Processing Tasks produce a report which is sent to the I/O Task. The I/O Task then writes the report as output. In order to verify the I/O Task's correctness during development, a Test I/O Task was developed to mimic the external I/O. The SCDSP was developed and run exclusively on the Mercury system until it was completed.

The Simulation Environment's design consists of an I/O Task, an Input Generation Task, and an Output Verification Task. Mirroring the SCDSP's I/O Task, the Simulation Environment's I/O Task is responsible for the Simulation Environment's input and output with the SCDSP. Synchronized to a periodic alarm in order to simulate a dwell, the Input Generation Task generates radar data and an associated command message which it sends to the I/O Task. The I/O Task writes this the data and the command message as output. The I/O Task is triggered when it receives a report. It sends the report to the Output Verification Task which verifies that the results are consistent with the previous dwell of data which was generated. It then sends selected content from the report to the Input Generation Task so that it can generate the next dwell of data accordingly. Finally, a Test I/O Task was developed to mimic I/O from the SCDSP. The Simulation Environment was developed and run exclusively on desktop systems until it was completed.

In Phase 2 of development and testing, the completed SCDSP and Simulation Environment TaskRunner applications were tested together on a desktop system. As illustrated in Figure 3.5, the Test I/O Tasks were removed and the I/O Task of the Simulation Environment was linked with the I/O Task of the SCDSP on the desktop environment.

The purpose of this phase was to isolate and correct algorithmic bugs in the SCDSP. The desktop system was selected for this phase of testing because it was better understood than the new Mercury system which had not been used for such heavy applications before. Furthermore, the Mercury TaskRunner Engine was also relatively new while the desktop TaskRunner Engine had been fully debugged over 3 years of heavy usage.

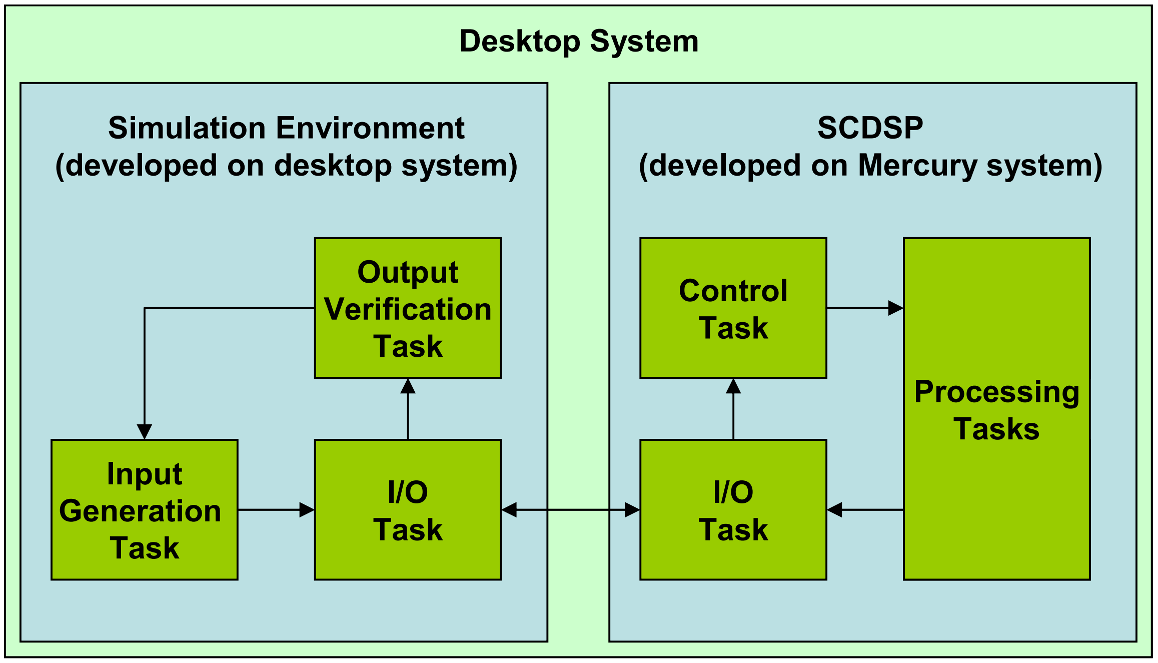


Figure 3.5 – SCDSP Testing on Desktop System

After the SCDSP was fully tested and verified on the desktop system, the SCDSP was tested with the Simulation Environment on Mercury, the target platform. Figure 3.6 illustrates the Simulation Environment running with the SCDSP on Mercury. As with testing on the desktop system, the Test I/O Tasks were removed and the I/O Task of the Simulation Environment was linked directly with the I/O Task of the SCDSP.

The purpose of Phase 3 was to verify that the SCDSP worked correctly on the target platform and to identify and correct any remaining software bugs which were particular to the Mercury system or the Mercury TaskRunner Engine.

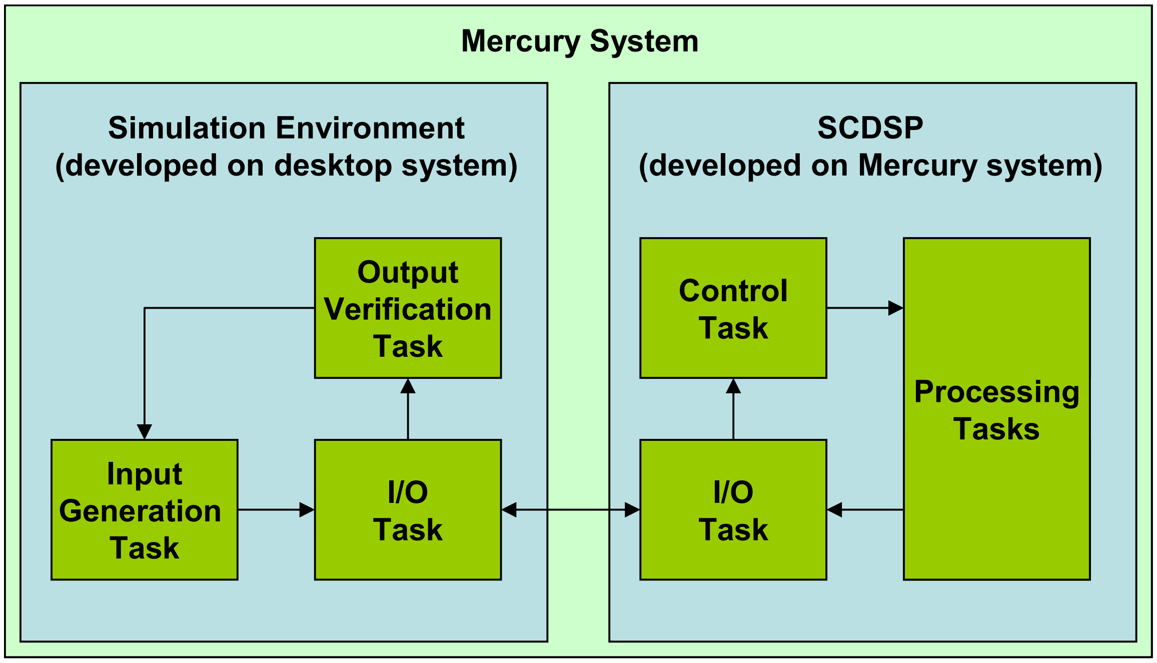


Figure 3.6 – SCDSP Testing on Mercury System

Once the SCDSP was verified while running with the Simulation Environment on the Mercury, the development and testing process was complete. The Mercury SCDSP was fully functional and was ready for full integration into the ASTB IH.

3.5 Results

Using TaskRunner to build the Mercury SCDSP resulted in significant savings in development time and higher code quality when compared to the development of the original SCDSP.

The Mercury SCDSP was successfully built and tested by 2 full-time developers in 3 months despite the fact that only one developer had access to a Mercury platform. As a metric for comparison, the original SCDSP took 4 full-time developers over 12 months to build and test. By enabling real-time code to be portable, TaskRunner allowed multiple developers to work together efficiently despite the severe bottleneck imposed by limited hardware availability.

By supporting object-oriented software, TaskRunner resulted in a Mercury SCDSP which had a more intuitive design, and also a better code structure than the original SCDSP. The benefit of higher code quality is a reduction in future maintenance costs.

The black box testing methodology proved successful when the team that later integrated the Mercury SCDSP into the ASTB IH encountered no software bugs.

4 Advanced Seeker Computer Tracking Algorithm

The performance of the new Seeker Computer Digital Signal Processor (SCDSP) significantly improves upon the performance of the old system. The SCDSP has a 20ms window within which to process each dwell of data and output its results before the next dwell of data arrives. While the old system took the entire 20ms window to complete its processing, the new system takes a little under 3ms to produce the same results. This improvement in performance presents a significant opportunity. We have over 17ms of idle time on the new system with which to do additional processing. We would like to take a look at ways we can use this processing time to extend the capabilities of the system.

The ASTB is often flown in engagement scenarios with aircraft that employ electronic countermeasures (ECM) to jam radar systems trying to obtain a track. As discussed in the introduction, the Seeker Computer is the system on the ASTB IH used to track aircraft. While aircraft are employing ECM to jam tracking, electronic counter-countermeasures (ECMM) may be employed by the tracking system to reduce the effectiveness of the ECM techniques.

With the new SCDSP as a platform for processing, the focus of this section is the development of one particular ECCM, leading-edge range track, for the ASTB. Leading-edge range track is an ECCM that is used against a common ECM device called the towed decoy.

The towed decoy is introduced in Section 4.1. The concepts of leading-edge track are described in Section 4.2. Section 4.3 describes the leading-edge track algorithm implemented by the author for the SCDSP.

4.1 Towed Decoy Problem

A towed decoy is a receiving and transmitting device that radiates electromagnetic energy specifically designed to jam enemy radar systems trying to track an aircraft. Towed decoys are deployed in-flight via a towline from the rear of the aircraft. Figure 4.1 shows the ASTB closing in on an aircraft carrying a towed decoy. The target is boxed in red and the decoy is boxed in green. For the remainder of this chapter, red is associated with the target and green is associated with the decoy.

An enemy radar system tracking an aircraft with a deployed towed decoy receives jamming energy from the decoy in addition to any signal energy it normally receives from the aircraft. The ratio of jamming energy to signal energy received by the radar system is known as the jamming-to-signal (J/S) ratio. When J/S is sufficiently high, the tracking system will track the decoy instead of the aircraft.

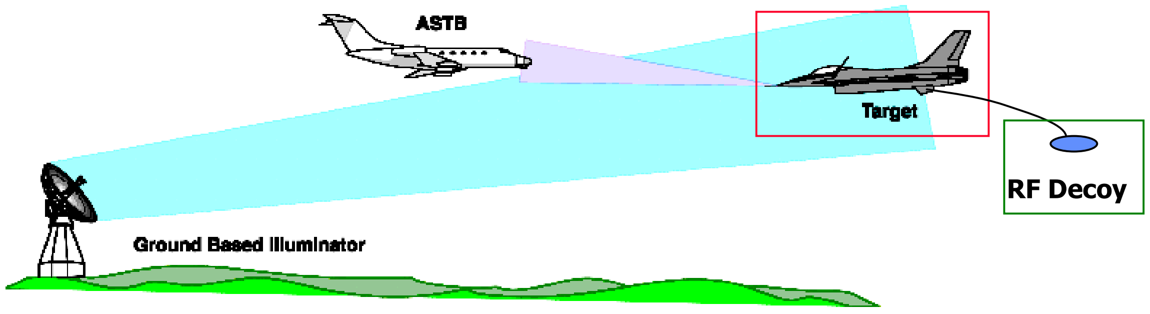


Figure 4.1 – ASTB Closing in on a Target with Decoy

Towed decoys come in a variety of configurations and can emit a number of jamming waveforms. We will focus our attention on one particular class of towed decoy, the repeater. A towed decoy operating as a repeater receives illumination from the tracking radar and retransmits this illumination with gain in an attempt to simulate return from a large aircraft.

Figure 4.2 shows the normal tracking procedure used on the ASTB. Time domain return is collected by the IH. An FFT of the return signal in the time domain gives a signal in the frequency domain. Peak detection in the frequency domain isolates the target's frequency bin and results in a track. Taking the monopulse channels at the track bin, the angle of the target is determined by Equations 3.1 and 3.2, which were introduced in Section 3.4.

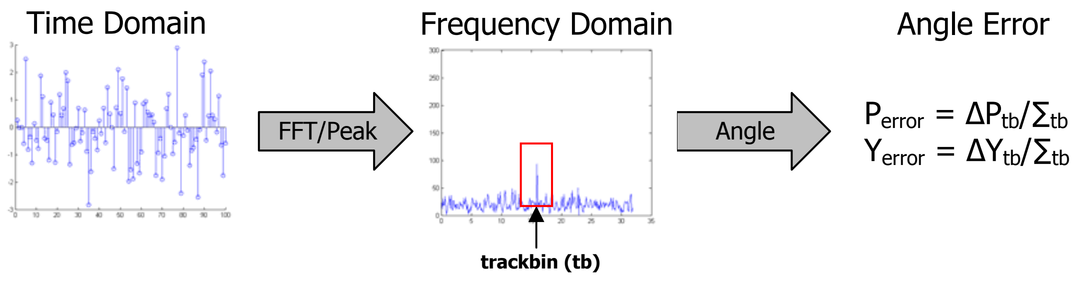

Time Domain
FFT/Peak
Frequency Domain
Angle
Angle Error
$P_{error} = \Delta P_{tb}/\Sigma_{tb}$
$Y_{error} = \Delta Y_{tb}/\Sigma_{tb}$
trackbin (tb)

Figure 4.2 – Angle Tracking Process in Frequency Domain

However, when a monopulse radar tries to track an aircraft carrying a decoy, peak detection in the frequency domain will find energy from both the aircraft and the decoy. Since the aircraft and the decoy travel at the same velocity, they produce identical Doppler shifts and their energies will lie in the same frequency bin.

This is illustrated in Figure 4.3. The decoy's contribution to the energy at the track bin is highlighted in green and the aircraft's contribution is highlighted in red. The decoy's contribution is much greater than the contribution from the aircraft.

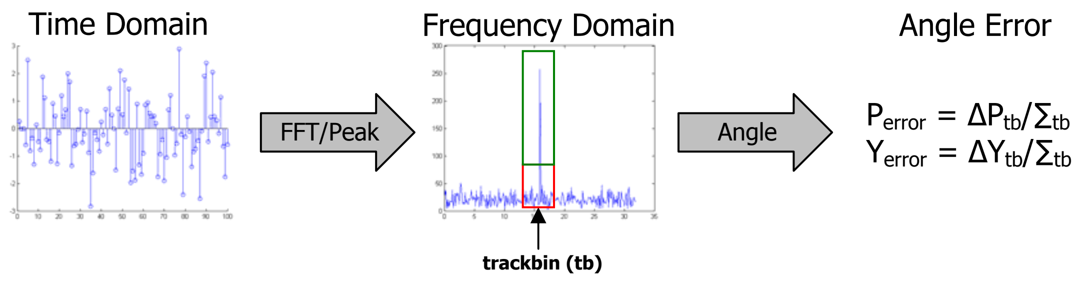

Time Domain
FFT/Peak
Frequency Domain
trackbin (tb)
Angle
Angle Error
$P_{error} = \Delta P_{tb}/\Sigma_{tb}$
$Y_{error} = \Delta Y_{tb}/\Sigma_{tb}$

Figure 4.3 – Angle Tracking Process in Frequency Domain with Decoy

The monopulse technique for determining angle assumes that all energy comes from a single source. When the energy being processed comes from multiple sources, the monopulse radar will bias its track towards the source with highest energy. This is why the J/S ratio is such a significant measure in determining the performance of electronic countermeasures. If the towed decoy's J/S is high enough, the tracking system will track the decoy instead of the aircraft.

4.2 Leading-Edge Range Track

Leading-edge range track is an electronic counter-countermeasure (ECCM) used to resolve the towed decoy problem. Leading-edge track takes advantage of the separation in range between an aircraft and a decoy in order to isolate aircraft energy for tracking.

The aircraft and the decoy are separated by a distance equal to the length of the towline. Assuming that the aircraft is flying towards the radar system, radar illumination will reach the aircraft before it reaches the decoy. As a result, a radar system tracking an aircraft carrying a decoy will receive reflected energy from the aircraft before it receives the corresponding reflection from the decoy. The difference in time-of-arrival between the two incoming signals allows leading-edge track to distinguish aircraft return from decoy return. Whereas the target and decoy were inseparable in the frequency domain, they can be separated in the time domain assuming the radar system employs the right illumination waveform.

One waveform which a radar system employing leading-edge track can use is a pulsed illumination waveform. Leading-edge track can then be applied independently to each pulse return.

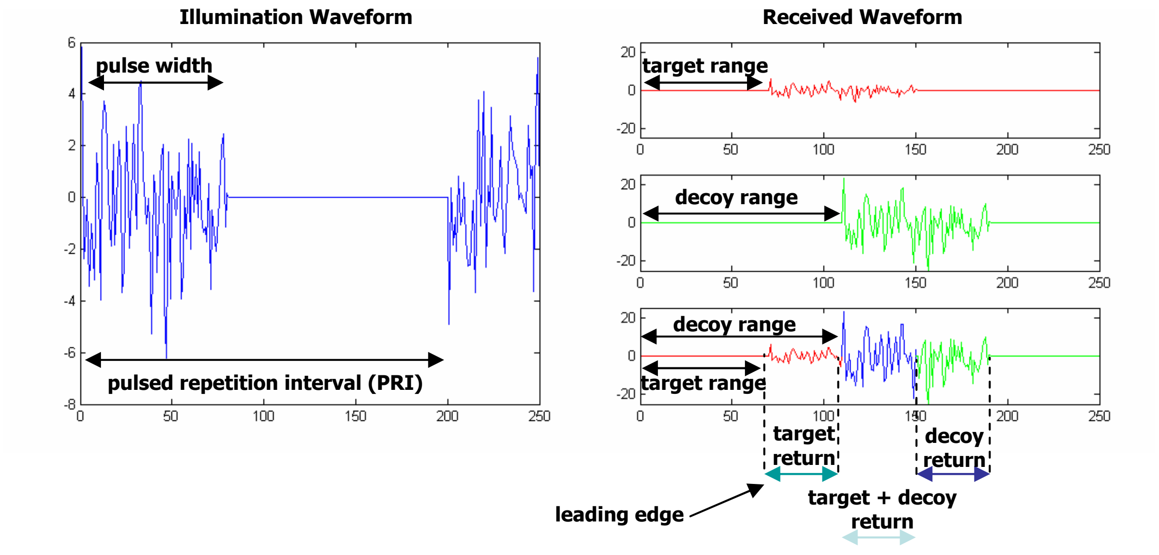


Figure 4.4 – Illumination and Received Waveforms

Figure 4.4 illustrates a pulsed illumination waveform and the associated return waveform which can be expected at the receiver. A transmitter transmits a pulsed waveform by sending pulses of energy followed by periods of silence. The duration of the pulse is known as the pulse width and the period between pulses is known as the pulse repetition interval (PRI).

As just described, the receiver receives return from the target-aircraft before it receives corresponding return from the decoy because of the difference in range between the target and decoy. If the decoy is close to the aircraft, for each pulse the received waveform will contain a section of target-only return, followed by a section where the target return overlaps with the decoy return, followed by a section of decoy-only return. In Figure 4.4 the target-only return is colored red, the overlap of target and decoy return is colored blue, and the decoy-only return is colored green. Finding the leading edge of the return allows us to isolate the target-only return.

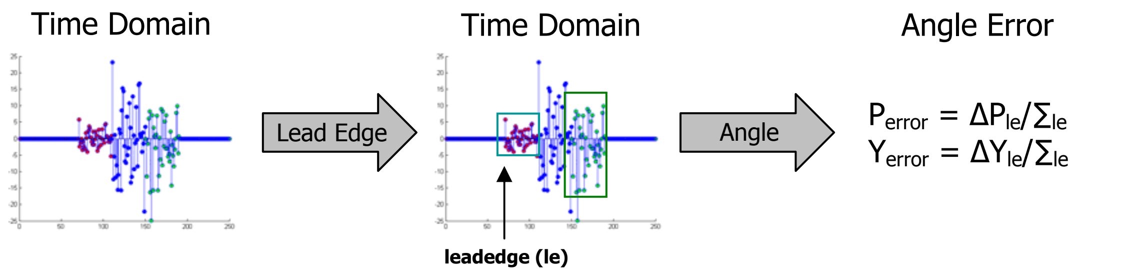

Time Domain
Lead Edge
Time Domain
leadedge (le)
Angle
Angle Error
$P_{error} = \Delta P_{le}/\Sigma_{le}$
$Y_{error} = \Delta Y_{le}/\Sigma_{le}$

Figure 4.5 – Angle Tracking Process Using Leading-Edge Track

Figure 4.5 shows an alternative angle tracking process using leading-edge track. As before, time domain return is collected. Next, the leading edge of the time domain return is identified. Finally, angle error is calculated by comparing the monopulse channels at the leading-edge. Since the energy at the leading edge comes entirely from the aircraft, this process will correctly determine the direction of the target-aircraft.

4.3 Leading-Edge Range Tracking for the ASTB

While directly implementing the leading-edge track algorithm described in Section 4.2 should be straightforward, 4 complications on the ASTB change the original problem. The leading-edge track algorithm described in Section 4.2 cannot be implemented directly on the ASTB because of 1) low signal-to-noise, 2) high jamming-to-signal, 3) under-sampling, and 4) a lack of time synchronization.

Figure 6 shows one pulse of simulated data representing the received waveform from the ASTB and illustrates each of these complications. Arrows indicate the different regions of return, which are color-coded as before.

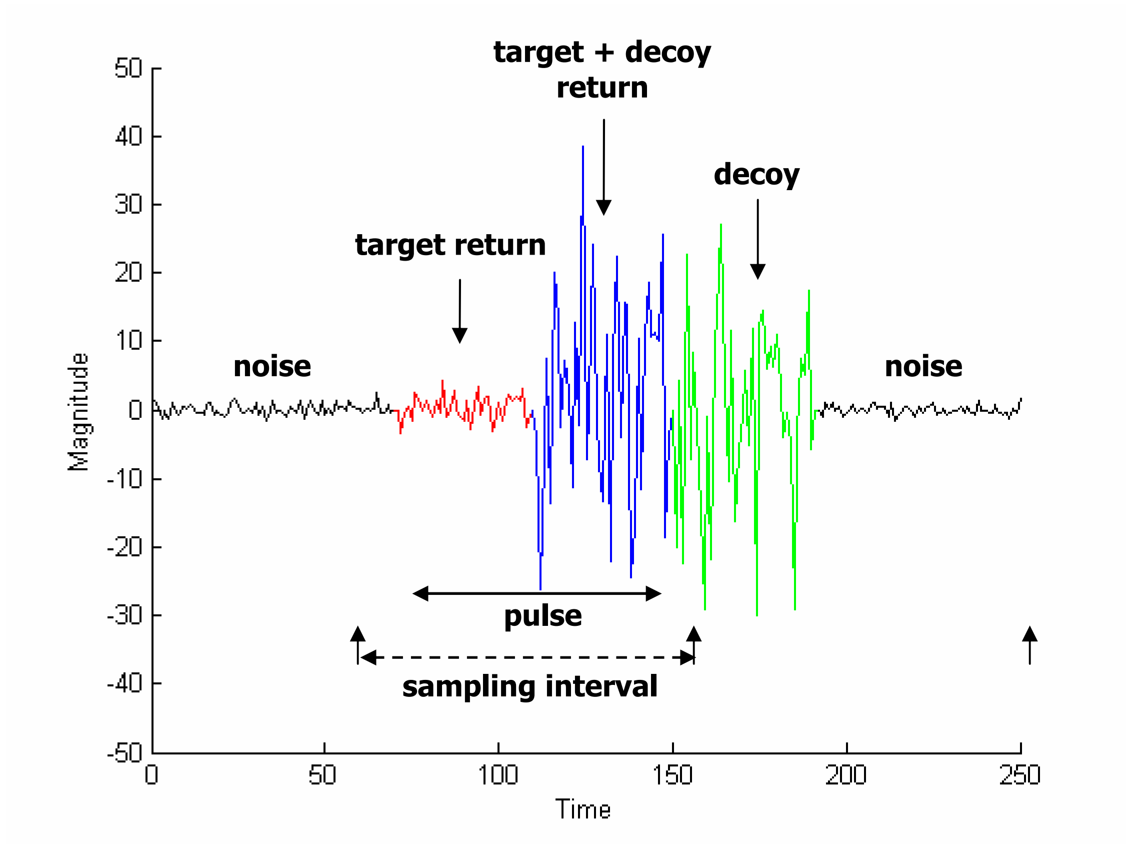


Figure 4.6 – Features of ASTB Data

First, low signal-to-noise (S/N) makes it difficult to find the leading-edge of the target return. The signal-to-noise ratio compares the power of target return with the power of the noise. In our previous description of leading-edge track we assumed we could find the leading-edge from the target return. However, in the presence of noise it can be difficult to identify target return from the noise. In Figure 4.6 the noise is colored in black and lies at the ends of the signal. The leading-edge of the target return is colored red and lies adjacent to the noise. In the Figure, the target return looks very similar to the noise. As the signal-to-noise ratio decreases, the magnitude of target return becomes comparable to that of the noise and it becomes difficult to differentiate the two.

A high jamming-to-signal (J/S) ratio in the return complicates matters even further. A good decoy may emit jamming energy which is an order of magnitude, or more, greater than the signal energy. In other words, the decoy return is likely to stand out much more from the noise floor than the target return. The Figure illustrates how the decoy return can dominate the target return. In trying to find the leading-edge it is easy overlook target return in favor of the larger decoy return.

A third problem is under-sampling by the ASTB receiver of the received waveform. The received waveform is a continuous signal. However, the SCDSP does not process this received waveform directly. The SCDSP processes samples of the received waveform taken by the A/D in the IH receiver. Unfortunately, ASTB's sampling rate is lower than the rate needed to reconstruct the signal. In fact, because the sampling interval is greater than the pulse width of the received waveform, we are not guaranteed to get any samples from the target at all. Figure 4.7 illustrates this problem.

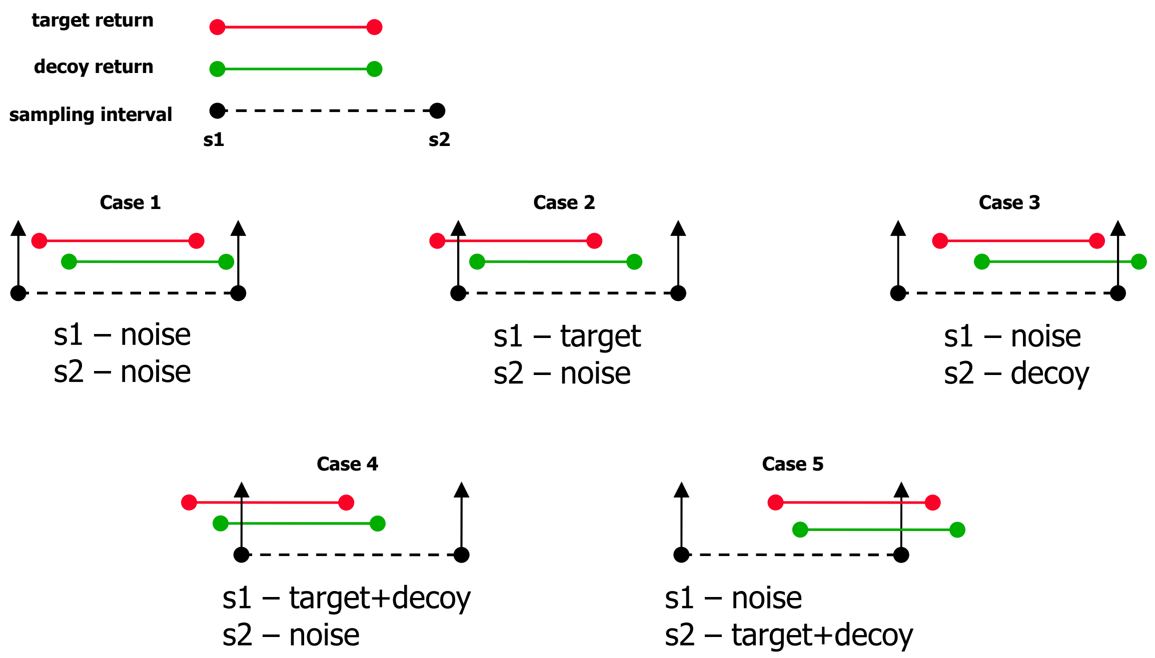


Figure 4.7 – Possible Sampling Cases

Figure 4.7 shows that due to under-sampling, we can get 5 different outputs from sampling the same pulse return depending on the alignment of the samples with the return. The target's pulse return is indicated by a red line segment. The decoy's pulse return is indicated by a green line segment. The sampling interval is greater than the pulse width and is indicated by a dotted black line segment. Samples are taken at the endpoints of the sampling interval. For each case, the relationship between the target return and the decoy return is fixed. The target return is received slightly before the decoy return. However, in each case the sampling interval is aligned differently to obtain a different output.

In Case 1 the target and decoy return lie within the endpoints of the sampling interval. No signal is sampled and both samples are noise. In Case 2 the first sample lies in the region of target-only return. The first sample is target return and the second sample is noise. In Case 3, the first sample lies well before the return and the second sample lies in the decoy-only region of return. The first sample is noise and the second sample is decoy return. In Case 4, the first sample lies in the region where the target and decoy return overlap. The first sample is target plus decoy return and the second sample is noise. In Case 5, the first sample lies well before the return and the second sample lies in the region where the target and decoy return overlap. The first sample is noise and the second sample is target plus decoy return.

Note that since the decoy return dominates the target return, sampling the region where the target and the decoy overlap is basically equivalent to taking a sample from the decoy itself. Only in Case 2 do we get useful information about the location of the target.

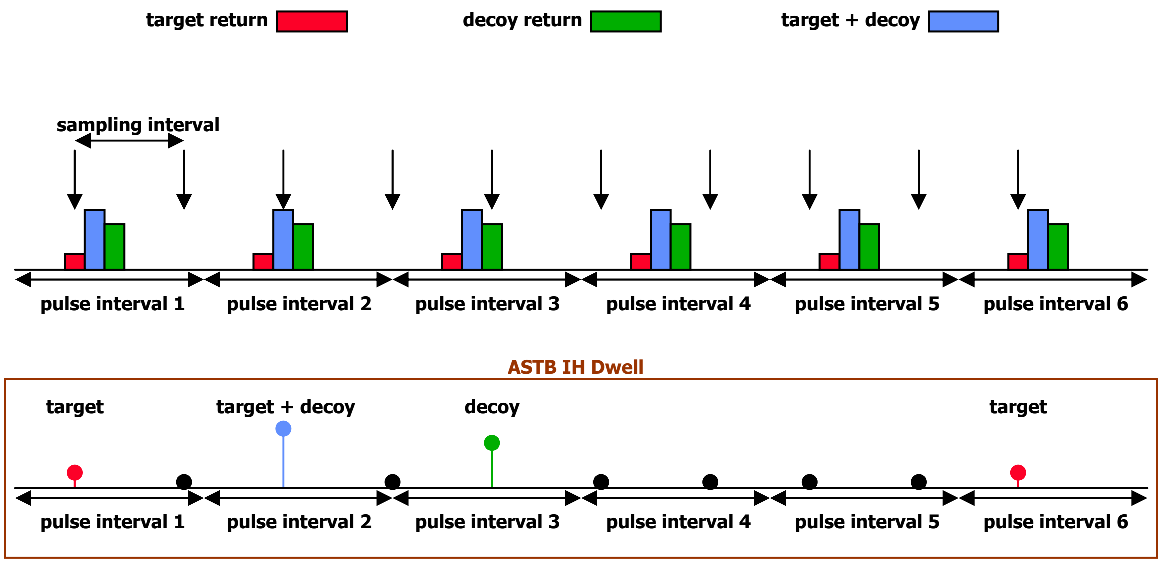


Figure 4.8 – ASTB Sampling of the Received Waveform

Figure 4.8 depicts 6 periods of the pulsed return waveform. Data is processed in the ASTB in blocks called dwells and each dwell contains samples from roughly 2000 pulse returns. The vertical arrows indicate A/D sample points within the IH receiver. Due to the relative size between the sampling interval and the pulse interval, the pulse return is under-sampled.

Looking at the samples over the whole dwell, we can see that in an ASTB IH dwell, there is no leading edge to track off of. The dwell covers many PRI's and the signal is a combination of noise samples, target return samples, target and decoy return samples, and decoy return samples in what appears to be an arbitrary order. Therefore, the sampled IH signal return for a given dwell is not well suited for the leading edge track algorithm.

Finally, a fourth problem is that the received samples are not synchronized with the transmitter emitting the illumination waveform. Without synchronization we cannot determine range because we have no reference point with which to measure against. This poses a problem because being able to discriminate target energy from decoy energy in range is the basis of leading-edge track.

4.3.1 Algorithm Outline

In order to determine the angle of the target we need some way of isolating the target-only samples from the return. The following section outlines an algorithm which accomplishes this and performs leading-edge track on the ASTB. This algorithm is illustrated in Figure 4.9. In the next section we will take a look at how the steps of this algorithm are implemented.

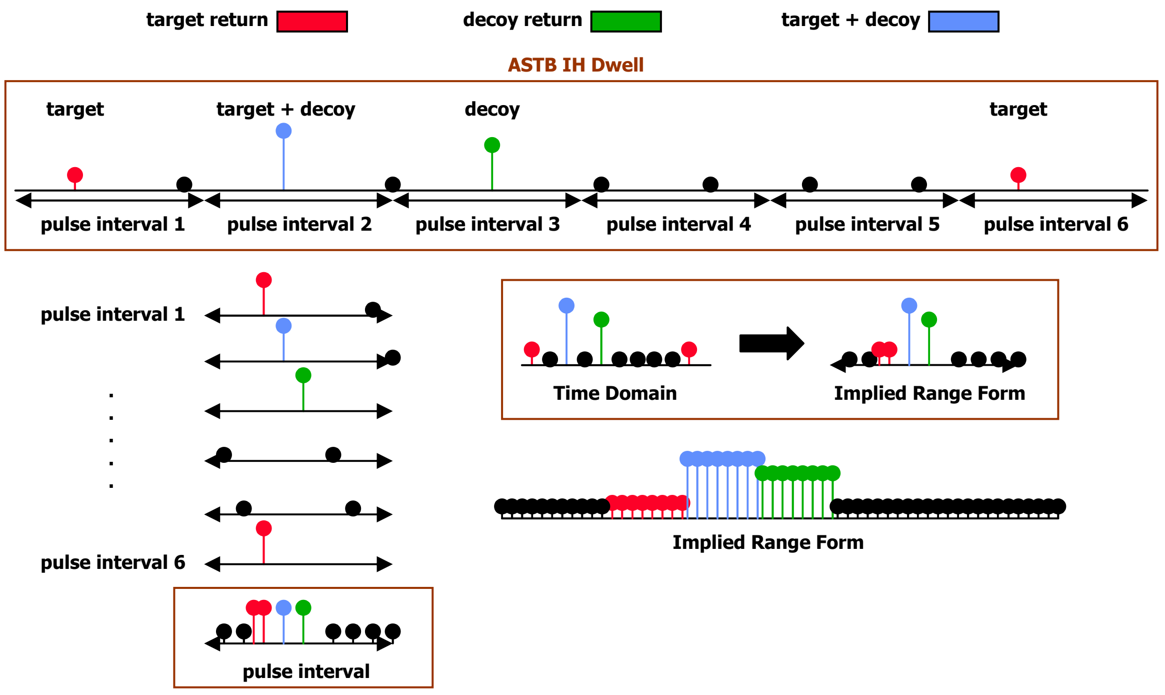


Figure 4.9 – Normalizing the ASTB IH Dwell

First, we normalize the data by the pulsed repetition interval (PRI) of the received waveform. The value of the PRI is either known in advance or can be calculated from the data. Starting from the first sample in the ASTB IH dwell we aligned the time domain samples against successive PRI intervals, subdividing regions of the ASTB dwell by the PRI. We then redistribute all of the samples by their positions relative to the PRI interval in which they lie. The samples from the original dwell are re-plotted in a normalized domain based on one PRI interval of the illuminator. We call this normalization the implied range form.

The implied range form is beneficial because it gives the range of all of the samples relative to the other samples in the dwell. While we cannot determine absolute range because the data is not synchronized with the illuminator, relative range is enough to perform leading edge track. The implied range form gives us the three distinct signal regions needed for leading edge track: the target-only signal, target plus decoy signal, and decoy-only signal. What the implied range form actually shows us is what we might expect if we finely sampled one period of the received waveform. From the implied range form we can identify a leading edge and correlate all samples with the target, the decoy, or noise.

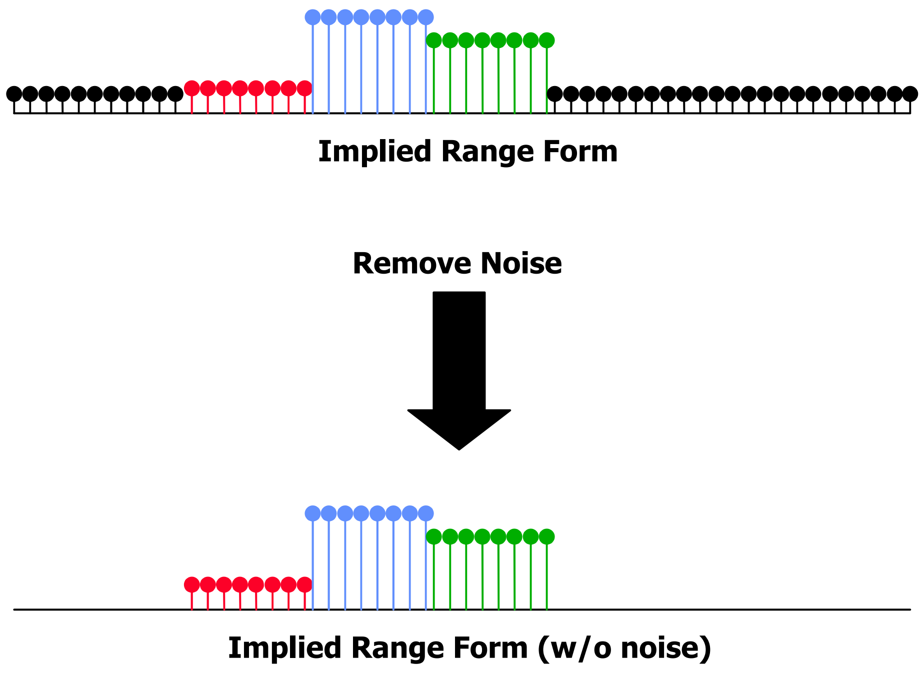


Figure 4.10 – Noise Removal

Given data in the implied range form, the next step is to remove noise so that we can identify the leading edge. Figure 4.10 demonstrates that it is easy to identify the leading edge once noise is removed.

Finally, we calculate the angle error of the target by comparing the energy in the monopulse channels at the leading-edge. Figure 4.11 summarizes the leading-edge angle tracking procedure modified for the ASTB.

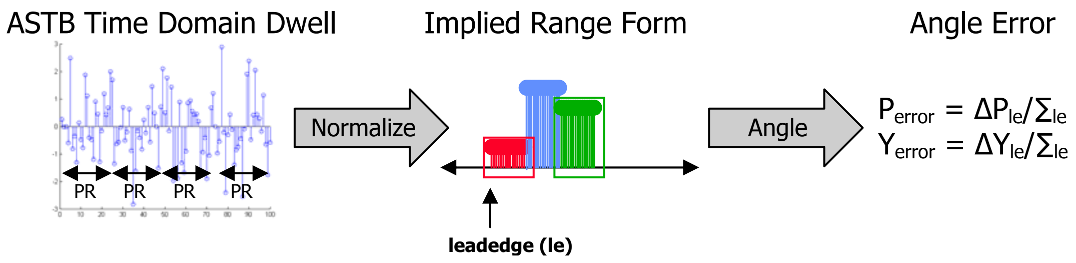


Figure 4.11 – Angle Tracking Process Using Leading-Edge Track on the ASTB

Time domain return is collected by the IH. The time domain data is normalized to obtain its implied range form. Next, the leading edge of the data in implied range form is identified. Finally, angle error is calculated by comparing the monopulse channels at the leading-edge of the implied range form. Since we have successfully isolated the target-only energy, this process will correctly determine the direction of the target-aircraft for the ASTB.

4.3.2 ASTB Implementation

The following algorithm performs leading-edge track on data received by the monopulse antenna on the ASTB. The algorithm processes 4 channels of input data: sum, delta pitch, delta yaw, and quad. Each channel contains one dwell of data, roughly 32,000 samples taken over 2000 pulses of the received waveform.

To help illustrate the algorithm, we will show intermediate outputs taken from running the algorithm on simulated data reflecting the geometry shown in Figure 4.12. A ground-based transmitter illuminates a target and a decoy with a pulsed Doppler waveform. As the ASTB closes in an the target and the decoy, the target lies 30° above the boresight of the monopulse antenna while the decoy lies 30° below. Therefore, the true angle error between the IH boresight and the target should be 30°. The sampling interval of the receiver is 1.1 times the pulse width of the received illumination. The pulse repetition interval is 9.14 times the sampling interval. The tow length is such that the delay between target return and decoy return is 0.5 times the pulse width. The decoy emits jamming energy which is 15 dB greater than the signal energy from the target. The signal energy from the target is 10 dB greater than the energy of the noise. The noise is white noise injected at the receiver.

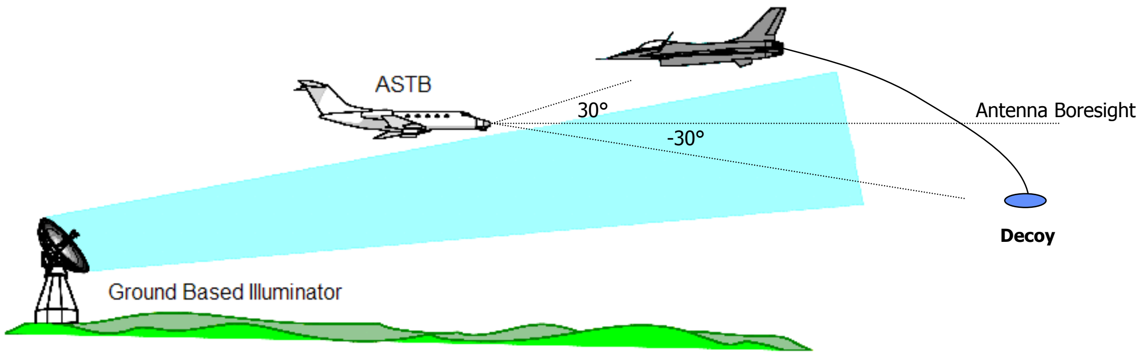


Figure 4.12 – Example Geometry for Simulated Data

Figure 4.13 displays the first 200 samples of simulated data from the sum channel. The delta pitch and delta yaw channels are scaled copies of the sum channel. The quad channel is not actually used in processing but may be useful for future improvements to the algorithm.

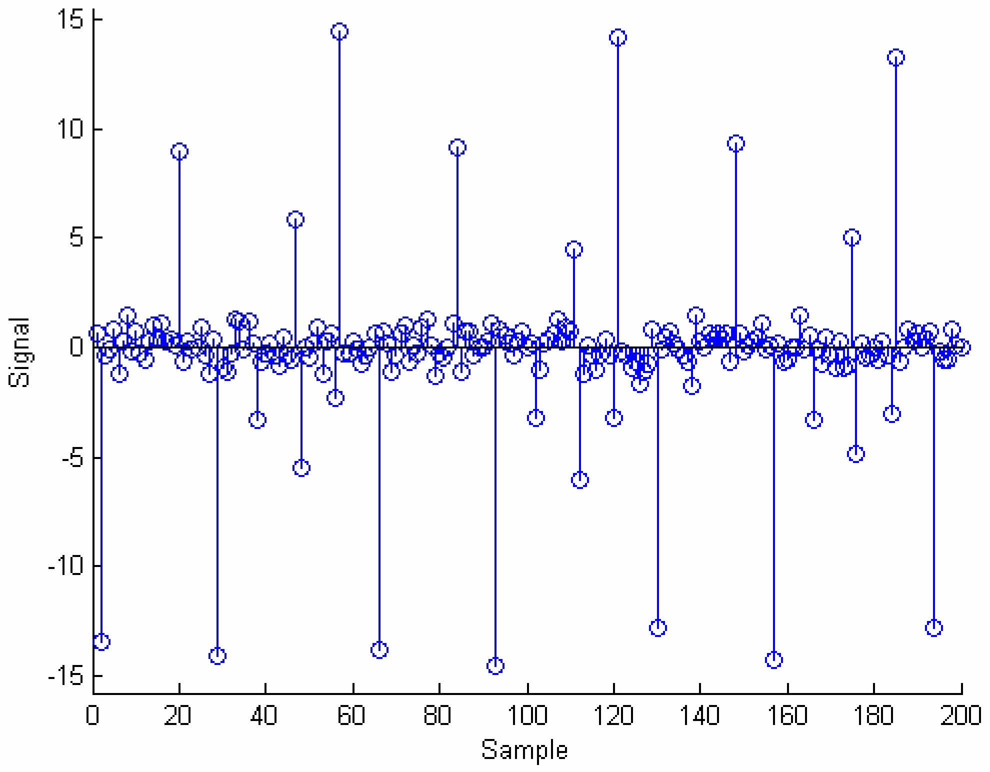


Figure 4.13 – Samples from Simulated Sum Channel

It is clear from Figure 4.13 that the raw dataset is not in the form in which we can apply leading edge track. The samples are somewhat random. There is a visible noise floor and there are also samples which lie clearly above the noise floor. However, it is not clear which samples belong to the target and which samples belong to the decoy.

The monopulse channels are dominated by energy from the decoy. Taking the ratio of the sum of the samples in the difference channel and the sum of the samples in the sum channel we get a monopulse angle error of -29.52°, which is approximately the location where we placed the decoy in our simulated data. Similarly, if we calculate an angle error by performing peak detection in the frequency domain we get an angle error of -29.27°. The jamming energy from the decoy obscures the true angle of the target at 30°.

4.3.2.1 Determining the PRI of the Pulsed Return

The first step in the algorithm is to normalize the data by the pulse repetition interval (PRI) of the illumination return. In the case of the ASTB, the pulse repetition interval is known. However in cases where it is unknown, the PRI can be accurately estimated directly from the data.

In the case of the ASTB the PRI of the received illumination can be calculated with information that is available or that is easily measured. The PRI of the received illumination is a function of the PRI of the transmitted illumination, the velocity of the target, and the speed of light. By the Doppler Effect, frequencies in the transmitted illumination are shifted in the received illumination according to Equation 4.1, where f_d is the Doppler shift, v is the velocity of the reflecting object, and c is the speed of light.

$$f_d = \frac{-2vf}{c} \tag{4.1}$$

In cases where the PRI of the illumination return is unknown, it can be estimated by determining the dominant frequency in the received data.

The assumption we make in using the received dataset to estimate PRI is that the return is a periodic signal and the return from a single pulse does not change significantly over the course of a dwell. This means that the PRI is nearly constant, and the return from each pulse is consistent with the return from all other pulses in the dwell. This assumption can be made on the ASTB because a dwell of data is collected over a very short span of time over which the target does not change position or velocity significantly. Furthermore, if we suspect that the characteristics of the return may have changed over the course of the dwell, we can reduce the size of the dataset to a portion of the signal where we can make this assumption.

To estimate the dominant frequency we take the autocorrelation of the sampled signal. Equation 4.2 defines the unbiased autocorrelation of a discrete signal, x[n] where N is the number of sample in x[n].

$$Cxx[t] = \sum_{n=0}^{N-t-1} \frac{x[n]x[n+t]}{N-|t|} \qquad (4.2)$$

$C_{xx}[t]$ is the correlation between samples from x[n] and a copy of x[n] shifted by t, x[n+t]. The received dataset is a periodic signal, thus we expect that the autocorrelation of the periodic dataset will exhibit peaks at samples equal to or multiples of the underlying period.

It is not necessarily true, nor should we expect, that the period of the return is equal to an integral number of samples. In the simulated dataset used to illustrate this algorithm, the PRI is equal to 9.14 times the sampling interval. This means that the period of the input data set is 9.14 samples and the underlying waveform repeats after 9.14 samples. Taking the autocorrelation of the sum channel of our simulated data we get the signal in the Figure 4.14.

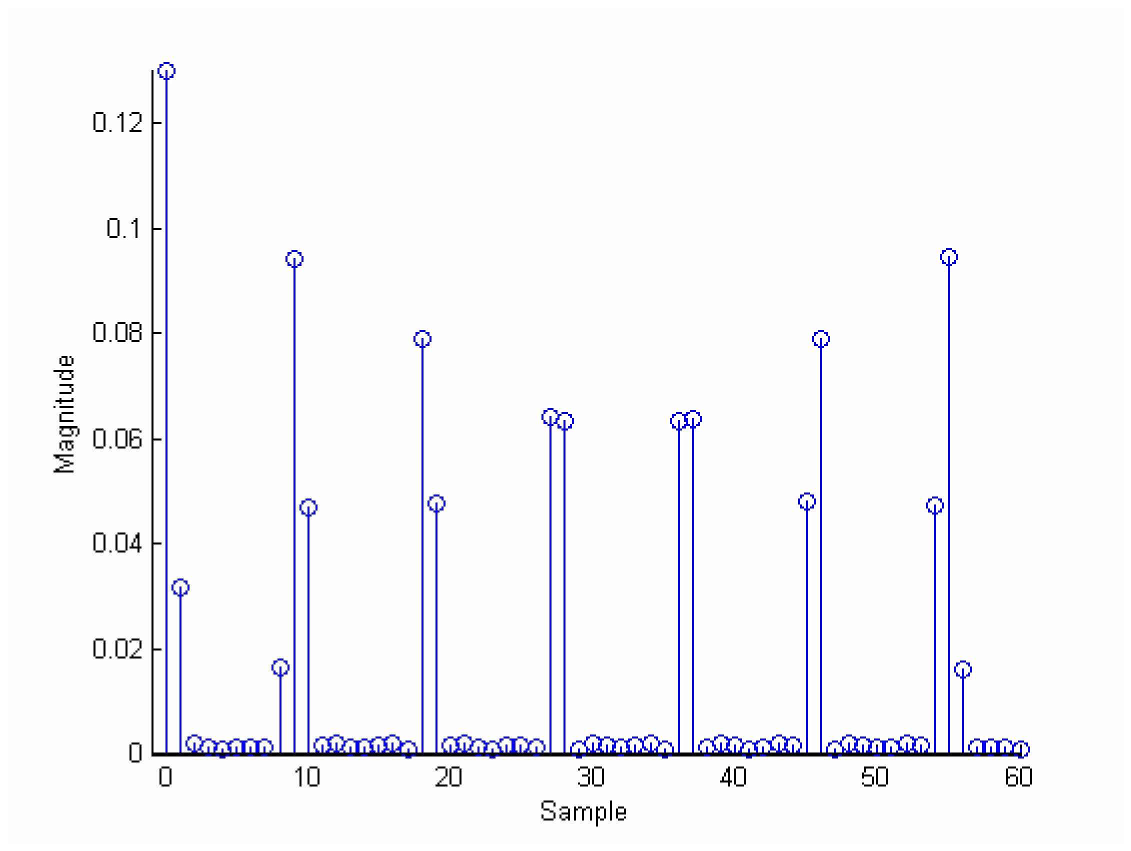


Figure 4.14 – Autocorrelation of Sum Channel

It is apparent from Figure 4.14 that the autocorrelation, $C_{xx}[m]$, contains peaks roughly every 9 samples. These peaks occur at m = 0, 9, 18, 27, 37, 46, 55. Each peak is separated by 9 or 10 samples. This is consistent with the PRI to sampling interval ratio of 9.14 which we set. Indeed, since 6 peaks occur over 55 samples the average number of samples between peaks is 55/6 = 9.16 which is fairly close to the true period of 9.14

samples which we have set. By using this process over the entire autocorrelation signal, we can estimate the period of the underlying pulsed return waveform. Calculation of the PRI is done by taking the autocorrelation of the sum channel, identifying the peaks, and calculating the average number of samples from peak to peak. The larger the number of samples which we have, the more accurate our estimation of the PRI will be. We can perform interpolation to increase the number of samples.

For this example, the algorithm calculates the value of 9.1428 for the PRI, which is correct to 5 significant digits.

4.3.2.2 Normalization to the Implied-Range Form

Once we have obtained the PRI, we then proceed to normalize the received dataset into the implied range domain.

Given that the 0th sample has an implied range of 0, implied ranges are assigned to the other samples based on the PRI. The implied range, η, of each sample is calculated by Equation 4.3, where n is the sample number.

$$\eta = n \bmod PRI \tag{4.3}$$

The intuition behind this transformation is that after every PRI samples we reach another period of the underlying pulsed waveform.

After calculating the implied range of all of the input samples, the points are sorted by their implied range and re-plotted to get the implied range form of the input data. The implied range values are normalized by the PRI so that the implied ranges have a minimum value of 0 and a maximum value of 1.

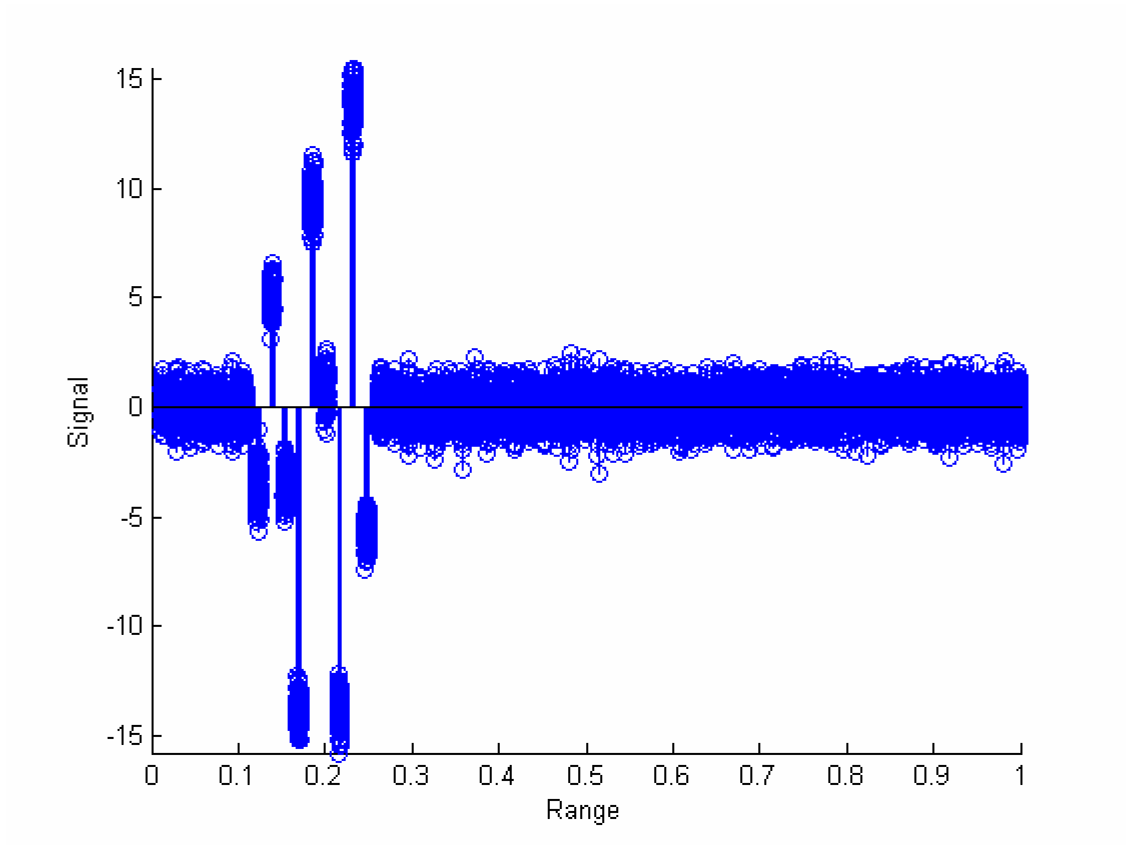


Figure 4.15 – Simulated IH Sum Channel in Implied Range Form

As we can see from Figure 4.15, the implied range graph gives a clear image of the pulse return. It is an estimate of what we might see if we sampled one pulse of the return at a rate much higher than the actual sampling rate.

In the implied range form, all of the significant points are grouped together. From the left side of the graph moving right, we see a group of points which are target return, a group of points which are target return plus the decoy return, and a group of points which are decoy return. There is a noise floor covering the entire signal. The implied-range form makes it very easy to identify samples from these distinct regions of the signal.

4.3.2.3 Finding the Leading Edge

With the data now ordered by range in the implied range form, the next step is to find the leading edge. The leading edge is the first non-noise signal returned in range. In order to find the first non-noise signal returned in range we need to be able to distinguish signal return from noise.

To do this, we make a few assumptions about the characteristics of the noise. We assume that the noise is Gaussian, which means it consists of independent random samples from a Gaussian distribution. Knowing the variance of the noise will give us enough information to distinguish signal from noise with some confidence. [1]

We remove noise by determining the characteristics of the noise present in the signal and then use likelihood testing to determine regions of return which are statistically significant. Threshold filtering is based on the assumption that the target return will be significantly higher than the noise floor. By calculating the statistics of the noise floor we can filter out all but the significant targets.

The major challenge is to determine the statistics of only the noise with a signal that contains both noise and signal samples. To calculate the variance of the noise we need to have noise samples with which to calculate the variance. Given that we do not know which samples are noise and which samples are signal in the first place, choosing noise samples with which to calculate variance is not a trivial task.

What we do know about the noise is that it lies in contiguous regions of the return signal. Figure 4.16 shows the implied range form with distinct returns from the target and decoy. Assuming that we only have two objects, the target and decoy, in our tracking space, the noise is split into at most two separate contiguous regions. If we break up the signal into blocks that are small enough, we are guaranteed that at least one of the blocks contains only noise samples.

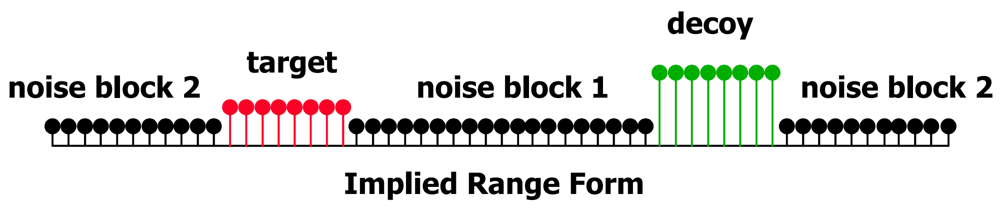


Figure 4.16 – Regions of Return in Implied Range Form

The proper size of these blocks in order to guarantee that at least one of the blocks is noise-only is half the width of the larger block size of noise. Since we do not know how the target and the decoy are positioned, we assume, for purposes of calculating the proper noise block size, the worst case scenario in which the noise is split into two blocks of equal size. The larger block of noise has a width that can be determined by Equation 4.4, and thus the proper block size, β, is determined by equation 4.5.

$$noisewidth = \frac{1}{2} - pulsewidth \tag{4.4}$$

$$\beta = \frac{1}{4} - \frac{1}{2} pulsewidth \tag{4.5}$$

If we assume that the pulse width during operation is no larger than .25 times the PRI we get a block size equal to 1/8 the PRI. The target and the decoy can occupy at most 3 blocks each, leaving 2 blocks which contain only noise samples.

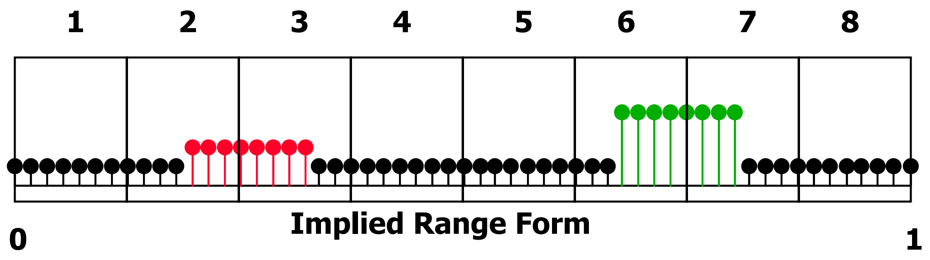


Figure 4.17 – Segmentation of Implied Range Form for Noise Calculation

The technique is shown in Figure 4.17, where blocks 1, 4, 5, and 8, contain only noise samples. Since we know that at least one of the blocks contains only noise samples, we can find the mean and variance of all of the blocks, and choose the one that looks most similar to noise.

Once we have the mean and variance of the noise, we filter out the noise by removing samples which are within 5 standard deviations of the mean noise. If the noise is truly Gaussian then the probability that noise samples exceed 5 standard deviations is extremely low. However, the threshold is set high to make the algorithm more robust in case the noise is not truly Gaussian. The data in implied range form without noise appears in Figure 4.18.

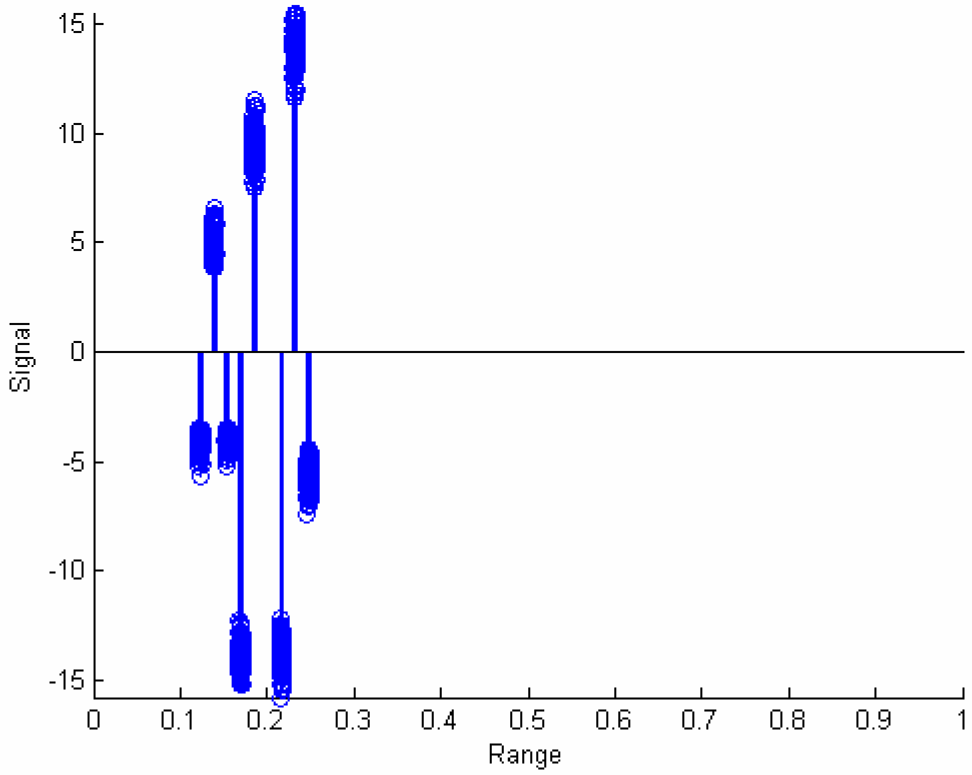


Figure 4.18 – Implied Range Form after Removal of Noise

We can now clearly identify the leading edge.

4.3.2.4 Angle-Range Form

We can take points on the leading edge to calculate angle error. The question, however, is what the best way is to do this is in order to get a robust estimate of angle error.

To answer this question we first calculate angle errors using Equations 3.1 and 3.2 as described in Section 3.4 for all of the remaining signal points and re-plot the data in a graph of angle error versus range. This is shown in Figure 4.19.

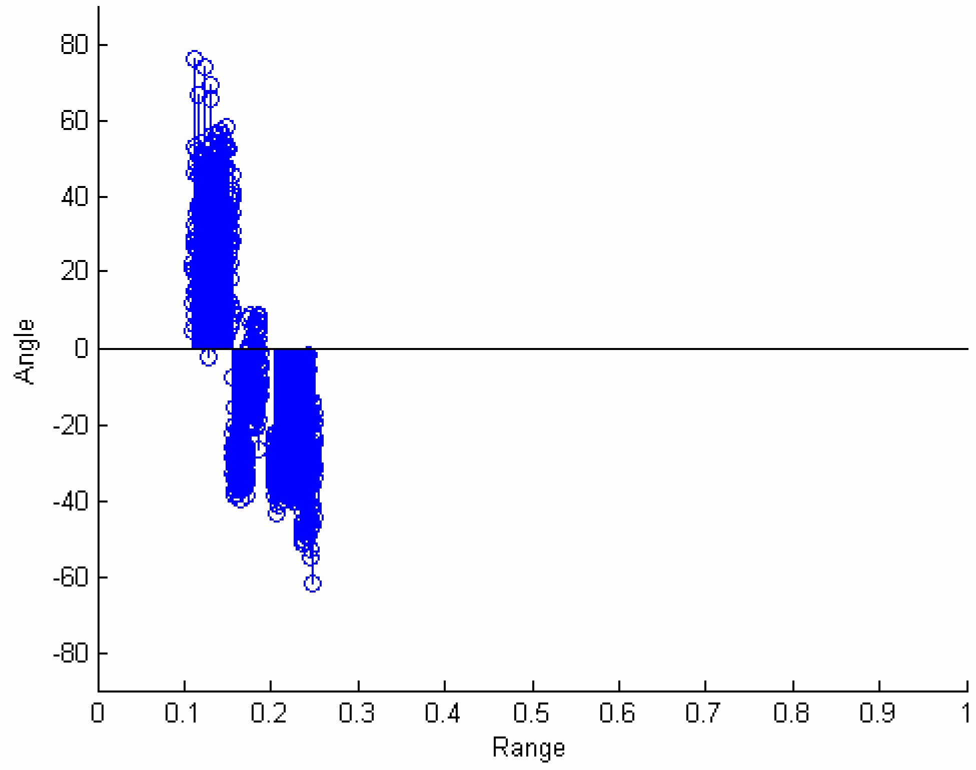


Figure 4.19 – Angle Error in Implied Range Form after Removal of Noise

The noise in the signal propagates into the angle error. Because the noise has zero mean, the calculated angle error varies around the true value of 30° but has high variance. If we zoom in closer at a portion of the leading edge as in Figure 4.20, we see that the angle errors are highly variable. For this simulated data set, the angle errors vary around 30° but range from 0° to 70°.

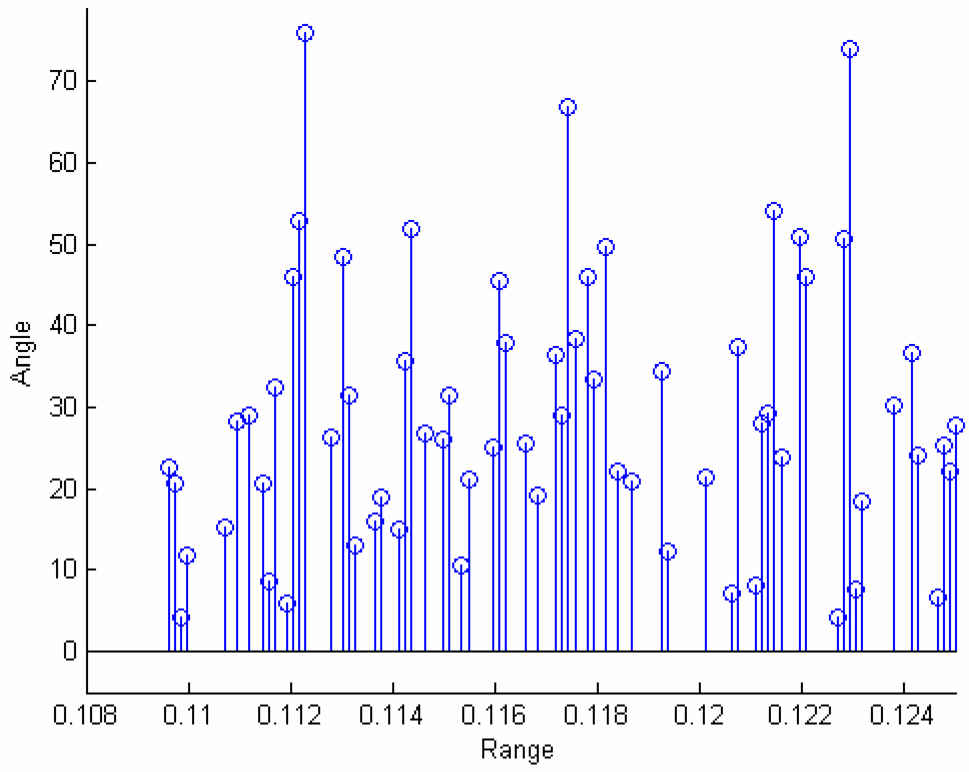


Figure 4.20 – Angle Errors at Leading-Edge

Clearly, calculating angle error from an arbitrary point on the leading edge will not give a robust estimate. While the angle from each point has an expected value which is equal to the true angle error, the variance is high because of the receiver noise.

4.3.2.5 Angle Smoothing and Correction

We use averaging to reduce the variance of our angle measurements and to obtain a more precise angle estimate. We smooth the raw angle estimates using a moving average. As we saw in the previous section, the expected value of the calculated angle with noise is equal to the true angle value. Taking a moving average will reduce variance while maintaining the same expected value.

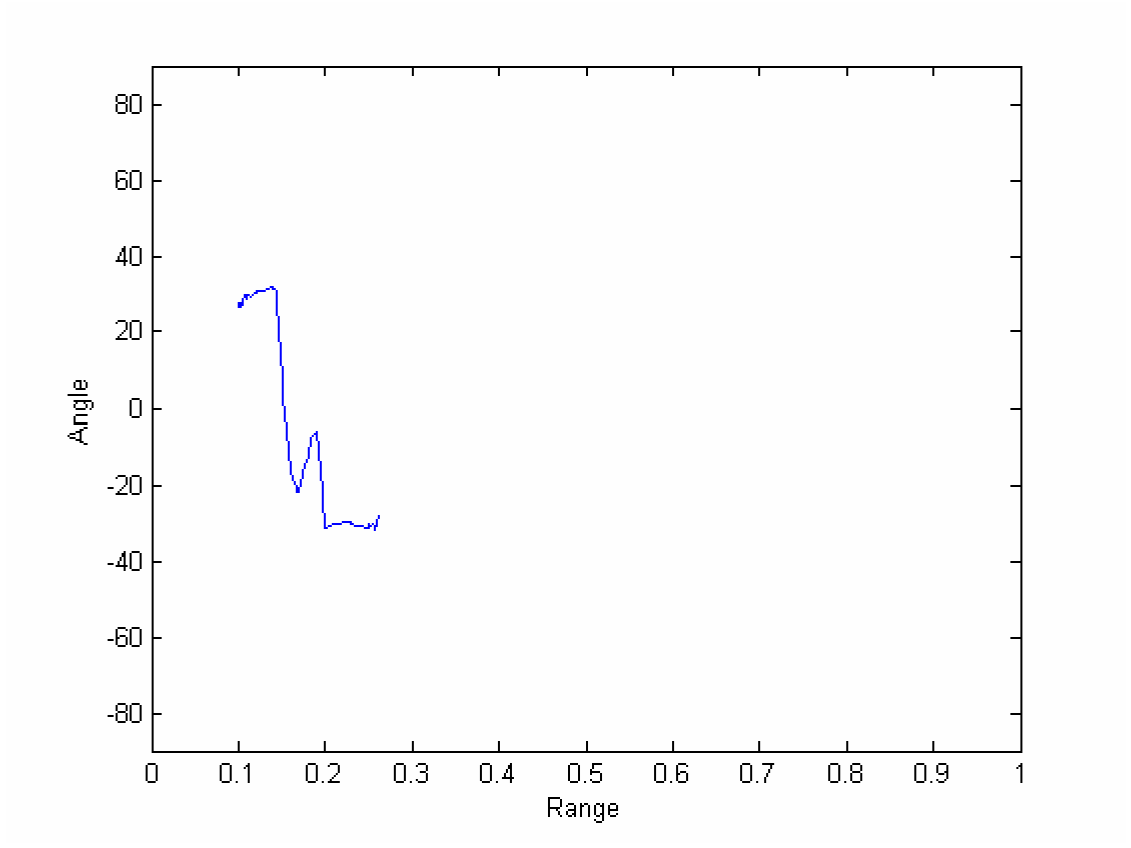


Figure 4.21 – Angle Errors after Averaging

The result of smoothing is a much clearer leading-edge range track of the angle of detected targets as shown in Figure 4.21.

Figure 4.22 shows the angle errors for the leading edge portion of the signal. The average angle error for this region is 29.44°, which is the final estimate of angle error from this algorithm. This estimate is less than 2.5% off the true error of 30°, demonstrating the effectiveness of this algorithm.

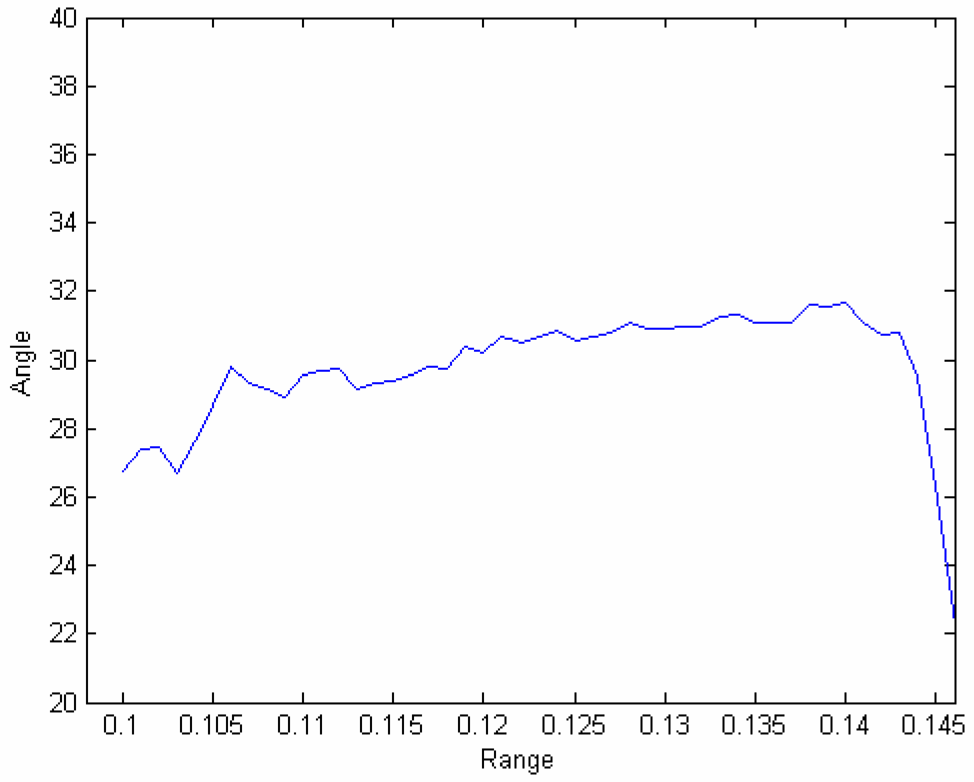


Figure 4.22 – Angel Errors at Leading-Edge after Averaging

Running this simulation 250 times we get an average angle estimate of 29.31° with a standard deviation of 2.22°.

4.3.2.6 Analysis of Performance and Limitations

The algorithm performs quite well on simulated data and gives an expected angle error estimate which is equal to the true angle error of the target. The interesting feature of this algorithm is that it does not require the evaluation of any Fourier transforms. Thus, it is a relatively lightweight algorithm for calculating the angle of a target. The most time intensive portion of this algorithm is taking the autocorrelation to find the PRI. The number of calculations needed to determine the autocorrelation grows with the number of input samples according to $O(n^2)$. The processing requirement of other steps grows linearly with the number of samples $O(n)$. With a few modifications, the autocorrelation can probably be replaced by another process for finding PRI which would probably reduce the running time.

Another interesting property of this algorithm is that is it is scalable to meet the desired input size. If processing time is a constraint, the algorithm will work with fewer samples. More samples simply allow the algorithm to fill in the implied range form with higher resolution. Even with samples from as few as 20 pulses, which is 1% of the data that is actually available in one dwell of the ASTB, the algorithm still delivers a fairly accurate estimate of the angle error, however with higher variance. If we run the simulation 1000 times with only 20 pulses we get an average angle estimate of 29.66° with a standard deviation of 8.73°.

5 Conclusion

As of May 2005, the Mercury Seeker Computer is in the process of being installed on the ASTB as the new tracking system for the IH. The Mercury Seeker Computer improves upon the software architecture, processing power, and algorithmic capability of the old system. It was developed over the course of 3 years.

In the first stage of development, a TaskRunner Engine was implemented for the Mercury Computer Operating System. The Mercury TaskRunner Engine allows TaskRunner software to be developed for the Mercury platform. TaskRunner software is object-oriented and portable across multiple platforms.

In the second stage of development, Seeker Computer Digital Signal Processing (SCDSP) software was developed for the Mercury Seeker Computer using TaskRunner. The new SCDSP software was thoroughly tested to meet all specifications. It behaves in exactly the same way as the original system; however it runs roughly 10 times faster. Furthermore, since the new SCDSP software was developed using TaskRunner, its code is object-oriented and can be reused in the future in case a new hardware platform is selected for a next-generation system.

Finally, the flexible software structure and the processing power of the Mercury Seeker Computer will allow it to support, in real-time, advanced tracking algorithms which were not possible in the old system. One such advanced tracking algorithm, leading-edge range track, was examined in detail and an algorithm was implemented to employ the technique using ASTB IH data. Against simulated data this algorithm proved to be very robust and efficient in tracking a single aircraft with a towed decoy.

References

[1] Skolnik, Merrill I. (2001), Introduction to Radar Systems, Third Edition. New York, NY: McGraw-Hill.

[2] Louis Hebert, HPEC September 24, 2002, Poster B.4 TaskRunner: A Method for Developing Real-Time System Software.

[3] Stimson, George W. (1998), Introduction to Airborne Radar, Second Edition. Mendham, NJ: Scitech Publishing.