NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

AN/SLQ-32 EW SYSTEM MODEL: AN EXPANDABLE, OBJECT-ORIENTED, PROCESS-BASED SIMULATION

by

Chen-Kuo Li

September 1992

Thesis Advisor: Michael P. Bailey

REPORT DOCUMENTATION PAGE

Form Approved
OMB No 0704-0188

1a REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b RESTRICTIVE MARKINGS
2a SECURITY CLASSIFICATION AUTHORITY	3 DISTRIBUTION / AVAILABILITY OF REPORT Approved for public release; distribution is unlimited
2b DECLASSIFICATION / DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S)	5 MONITORING ORGANIZATION REPORT NUMBER(S)
6a NAME OF PERFORMING ORGANIZATION Naval Postgraduate School 6b OFFICE SYMBOL (If applicable) EW	7a NAME OF MONITORING ORGANIZATION
6c. ADDRESS (City, State, and ZIP Code) Monterey, CA 93943-5000	7b ADDRESS (City, State, and ZIP Code)
8a. NAME OF FUNDING / SPONSORING ORGANIZATION 8b OFFICE SYMBOL (If applicable)	9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State, and ZIP Code)	10 SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO / PROJECT NO / TASK NO / WORK UNIT ACCESSION NO

11. TITLE (Include Security Classification)

AN/SLQ-32 EW SYSTEM MODEL: AN EXPANDABLE, OBJECT-ORIENTED, PROCESS-BASED SIMULATION

12. PERSONAL AUTHOR(S)

Li, Chen-Kuo

13a TYPE OF REPORT	13b TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15 PAGE COUNT
Master's thesis	FROM _____ TO _____	1992, September	96

16 SUPPLEMENTARY NOTATION
The views expressed in this thesis are those of the author and do not reflect the official or position of the Department Of Defense or the U.S.Government.

17 COSATI CODES			18 SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	Object-Orientation, MODSIM, AN/SLQ-32 EW SYSTEM model, Programming, Process-based, Simulation.

19 ABSTRACT (Continue on reverse if necessary and identify by block number)

This thesis documents the design and implementation of a simulation of AN/SLQ-32 Fleet Defense EW System in a modern, object-oriented, process-based simulation language called MODSIM II by CACI Corporation of La jolla, CA. The main intent of the simulation is to build a model that simulates an AN/SLQ-32 EW system's capability in an environment having an arbitrary number of different emitters. The trials presented in this work use 15 distinct emitters. This simulation model is designed to provide a foundation that not only can be used to study AN/SLQ-32 EW system reliability, but also can be built upon as a part of a wargame or modified to study varied topics such as training effectiveness of naval EW system operators.

20 DISTRIBUTION / AVAILABILITY OF ABSTRACT	21 ABSTRACT SECURITY CLASSIFICATION
☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS	Unclassified
22a NAME OF RESPONSIBLE INDIVIDUAL Michael P. Bailey	22b TELEPHONE (Include Area Code) (408) 646-2085 22c OFFICE SYMBOL OR/BA

Approved for public release; distribution is unlimited.

AN/SLQ-32 EW SYSTEM MODEL: AN EXPANDABLE, OBJECT-ORIENTED, PROCESS-BASED SIMULATION

by

Chen-Kuo Li
Lieutenant Commander, Taiwan Navy
B.S., Chinese Naval Academy, 1982

Submitted in partial fulfillment
of the requirements for the degree of

MASTER OF SCIENCE IN SYSTEM ENGINEERING
(ELECTRONIC WARFARE)

from the

NAVAL POSTGRADUATE SCHOOL
September 1992

Jeffrey B. Knorr, Chairman
Electronic Warfare Academic Group

ABSTRACT

This thesis documents the design and implementation of a simulation of AN/SLQ-32 Fleet Defense EW System in a modern, object-oriented, process-based simulation language called MODSIM II by CACI Corporation of La Jolla, CA. The main intent of the simulation is to build a model that simulates an AN/SLQ-32 EW system's capability in an environment having an arbitrary number of different emitters. The trials presented in this work use 15 distinct emitters. This simulation model is designed to provide a foundation that not only can be used to study AN/SLQ-32 EW system reliability, but also can be built upon as a part of a wargame or modified to study varied topics such as training effectiveness of naval EW system operators.

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGEMENT

I would like to thank my adviser, Professor Michael P. Bailey, American friends whom I met in course OA 4333, Advanced Simulation, and Lawrence Altshuler, the **Raytheon** company's technical adviser, for their support and recommendations that helped me to complete this thesis.

The reason I chose to build an Electronic Warfare (EW) system simulation model for my thesis is that The Republic Of China (R.O.C.) Navy just bought several sets of AN/SLQ-32 Fleet Defense EW Systems from the U.S. Navy. The EW system's capability is always an unknown factor for ship defense capability. I believe that doing research for my thesis is a very good opportunity to study this EW system's performance. Through this simulation model the commanding officers of the R.O.C. Navy will be able to easily understand the capability of this system. The model will be used as a reference when the system sea trials commence.

I. INTRODUCTION

The hostile environment at sea today presents a formidable problem for the defense of a capital ship. Shipboard hostile radars are capable of detecting and targeting a ship from beyond the horizon, and those on airborne surveillance and attack aircraft operate from well beyond 100 nautical miles. Anti-ship missiles can be launched from ships, submarines, and aircraft. They can be given midcourse guidance corrections by another platform, and then home in on their targets using active radar seekers.

Today's high value ships are the first targets for the unseen enemy that seeks to strike a crippling blow to the defenses of any nation. To prevent sneak attacks, the electronic defense system must be able to detect the presence of an enemy, and to identify the platforms and weapons he is using, to prevent him from targeting capital ships at long range, and to deceive the missiles he launches into missing their intended targets. It must perform these tasks in the midst of a virtual storm of various electromagnetic signals.

How important Electronic Warfare will be in the war today has been completely understood. For self-protection, the Republic Of China (R.O.C.), has purchased several sets of AN/SLQ-32 Fleet Defense EW System from the US Navy. The

buyer, the Republic Of China Navy, was told the AN/SLQ-32 utilizes the most advanced technology to provide the system with outstanding operational availability and an unsurpassed capability to accomplish its mission in battle. But how reliable the system's performance is, is always a question.

System simulation was chosen to carry out research into this question in this thesis. Simulation involves the use of computers to imitate the operations of the actual system. Simulation can be used in evaluation of Electronic Warfare systems after they are built. This type of simulation often provides a more cost effective method of determining their optimum utilization and their effectiveness than actual tests.

This thesis documents the construction of a simulation of AN/SLQ-32 Fleet Defense EW System in MODSIM II. MODSIM II is a general-purpose, modular, block-structured high-level programming language which provides direct support for object-oriented programming and discrete event simulation. It can be used to build large process-based simulation models through modular and object-oriented development techniques. This simulation model, called Test, seeks to simulate an AN/SLQ-32 EW system's capability in the environment having an arbitrary number of different emitters. The trials presented in this work use 15 distinct emitters.

The simulated AN/SLQ-32 system is required to intercept the electromagnetic radiation and to locate and identify

sources. If the target has been recognized as a threat, the proper countermeasure will be initiated.

This simulation is designed to provide a foundation that not only can be used to study system reliability, but also can be built upon as a part of a wargame or modified to study varied topics such as training effectiveness of naval EW system operators.

II. AN/SLQ-32 EW SYSTEM DESCRIPTION

The AN/SLQ-32 system has two major subsystems, **ESM** and **ECM**, performing the signal interception, emitter identification and threat countermeasures.

In order to understand the simulation model, understanding the architecture of EW system is required. Because information concerning the **ECM** subsystem is classified, this thesis only stresses the operational behaviors of the **ESM** subsystem.

The **ESM** subsystem consists of the RECEIVER-ANTENNA GROUP and DISPLAY-PROCESSOR GROUP. A detailed description of the operational elements of both groups will be provided later in this chapter.

The general function of the **ESM** subsystem is provided in the following few paragraphs. This gives the reader a general picture of the electronic warfare system.

The direction finding antenna in the RECEIVER-ANTENNA GROUP consists of an array of elements fed through coaxial cables by a multiple beam parallel-plate lens constructed in stripline form using printed-circuit techniques. This lens-fed array provides a set of individual contiguous high-gain beams, all existing simultaneously, with each beam possessing the full gain of the array aperture. [Ref.1:p.8] The

operation of direction finding in the receive mode is illustrated in Figure 1.

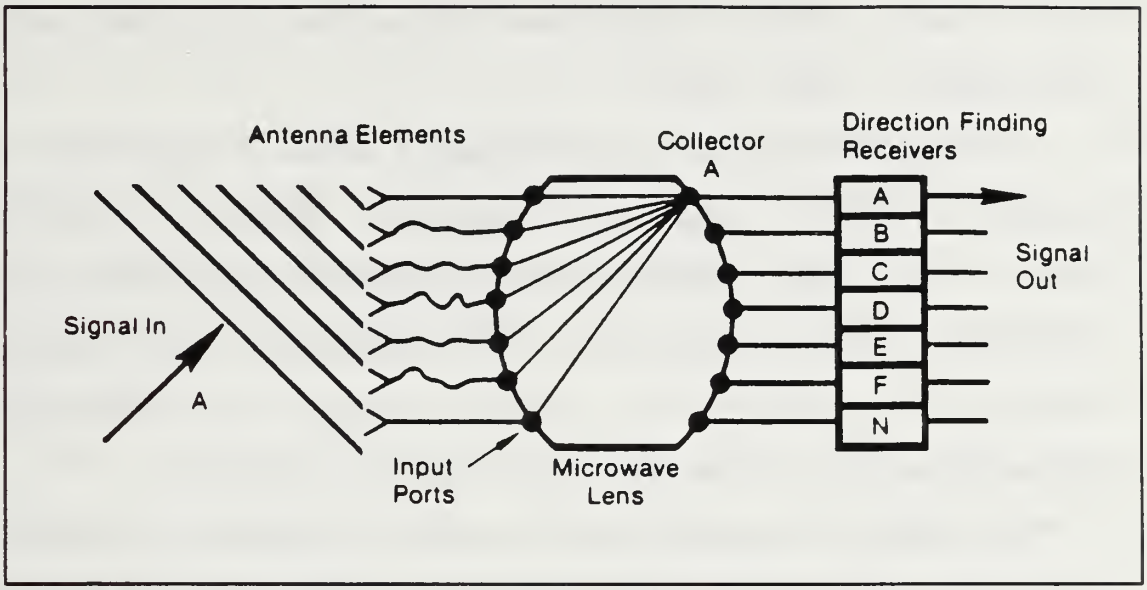

Figure 1 Multibeam Receive Antenna

An incoming signal wavefront from direction A is received by the array and produces a set of signals at the outputs of the array elements. These signals enter the multibeam lens through cables and are brought to a focus at point A, where receiver A is connected. The focusing property of the lens is independent of frequency. This means that the beam-pointing directions do not vary with frequency, thus accurate direction finding can be achieved over wide frequency bands and wide angular sectors simply by comparing the signal amplitudes received at adjacent beamports. [Ref.1:p.8]

As shown in Figure 1, a pulse arriving from a given angle is focused by the multibeam antenna array and sensed by one or more of the beamports in the direction finding receiver

(DFR). The output of the beamport is compared with that of all other beamports so that, if more than one signal is present, the dominant (highest amplitude) signal can be identified for processing. [Ref.1:p.9]

In parallel with the multibeam antenna and DFRs, the system utilizes a semi-omni antenna to sense the pulse and feed it to the instantaneous frequency measuring (IFM) receiver, which determines the frequency of the received energy by comparing the phase outputs of a series of frequency-sensitive delay lines. [Ref.1:p.9]

The Receiver System Block Diagram is shown in Figure 2.

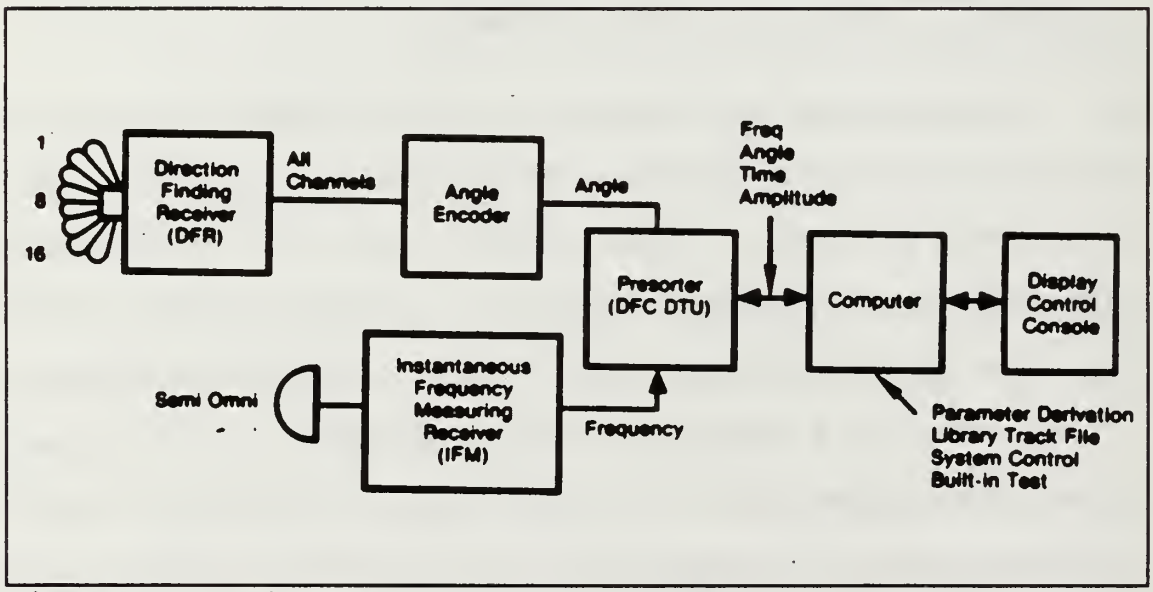

Figure 2 Receiver System Block Diagram

The presorter (DISPLAY-PROCESSOR GROUP) is a special-purpose digital processor composed of a direction/frequency correlator (DFC) and a digital tracking unit (DTU). Angle and amplitude data from the DFR and coarse frequency data from the IFM are correlated in the DFC. Time-of-arrival data is added

to form a pulse descriptor word (PDW), which is then stored by frequency and angle cell in the emitter file memory of the DTU. If three or more pulses of this frequency and from this angle are received within a programmable time interval up to 32 milliseconds, the DTU notifies the computer that a new emitter is present. The computer then requests the DTU to store subsequent pulses of that emitter over a time period sufficient to provide enough pulses for further analysis. [Ref.1:p.9]

The computer calculates from such data the pulse repetition interval (PRI), scan period, and type of scan. These parameters, along with frequency, are usually sufficient to characterize an emitter. [Ref.1:p.9]

Identification is completed by comparing the observed signal characteristics with the parameters of possible emitters in a library stored in the computer memory. When identification is complete, the computer sends the emitter information to the display to alert the operator and recommends appropriate action. [Ref.1:p.9]

A. RECEIVER-ANTENNA GROUP

Due to the frequency coverage of the AN/SLQ-32 EW system being classified, the frequency coverage in this simulation model is assumed to be in the 2 to 18 GHz range. There is no frequency band division. In the following paragraph, the original nomenclature is still used for convenience.

The receiver-antenna group, may be conveniently subdivided into three major functions: the band 2/3 angle and amplitude processing system, the band 2/3 frequency processing system, and the band 1 receiver. [Ref.2:p.3]

The band 2/3 angle and amplitude processing system comprises of band 2/3 direction finding receivers (DFR), and a port and starboard angle encoder. [Ref.2:p.3]

The band 2/3 frequency processing system outboard (port and starboard) components comprise of semi-omni antennas for both bands 2 and 3, band 3 switchable preamplifiers (SWPAs), and semi-omni preamplifiers (SOPAs). The inboard equipment includes an instantaneous frequency measurement/multiplexer (IFM/MUX) and an IFM/coarse frequency receiver(IFM/CFR). [Ref.2:p.3]

1. Band 2/3 Angle and Amplitude Processing

a. Direction Finding Receiver (DFR)

Four DFRs, two for band 3 and two for band 2, are mounted in each antenna array and boresighted 45 degrees and 135 degrees from the ship's heading, respectively. Since each DFR provides approximately 90 degrees of azimuth coverage, a total of eight DFRs (four port and four starboard) are required for 360-degree coverage in both bands. [Ref.2:p.3]

In the program of this thesis only one DFR is created covering 360 degrees instead of creating four DFRs to simplify the programming work.

Emitter signals are received by a multi-beam antenna that employs multiple separate beams to monitor a 90-degree quadrant. When an emitter is received, the differing amounts of RF energy in each beam provide amplitude information used by the angle encoder to determine emitter angle of arrival (AOA). To facilitate this process, the energy from the antenna array elements are focused by beam forming lens into separate beamports. [Ref.2:p.3]

b. Angle Encoder

One angle encoder mounted in each outboard ESM enclosure is used to process the outputs of the four DFRs (two band 3 and two band 2) on that side of the ship. [Ref.2:p.6]

The amplitude of each of the beamport signals from each band 3 DFR is measured and compared with the other beamports to determine which of the associated RF beamports produced the maximum signal in that quadrant.

In normal-mode processing, the greater of the two quadrant maximum signals is then selected for angle and amplitude encoding [Ref.2:p.6] (as shown in Figure 3).

When maximum video signals arrive that are below the receiver threshold level, no angle processing occurs. [Ref.2:p.6]

The band 3 sensitivity control function in the angle encoder uses a sensitivity select signal from the presorter to set the angle encoder input threshold level to

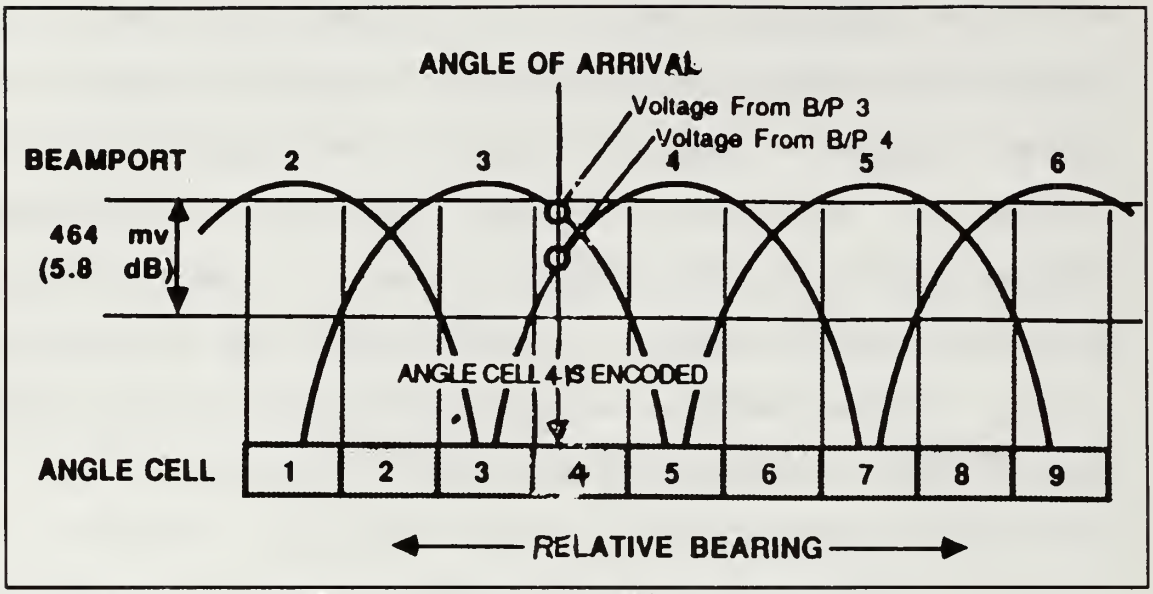

Figure 3 Band 2/3 Angle Encoding

either normal or high sensitivity. [Ref.2:p.7]

In this program only one Angle Encoder is created to process signals that are received by The Direction Finding Antenna with the sensitivity always set at the normal level.

2. Band 2/3 Frequency Processing

a. Semi-Omni Antenna

A separate semi-omni antenna for band 2 and band 3 is mounted in each outboard ESM enclosure. These antennas, each providing 180-degree azimuth coverage, are used for detection of signals to be processed in the frequency channel. [Ref.2:p.7]

b. Instantaneous Frequency Measurement/Multiplexer (IFM/MUX)

The inboard-mounted IFM/MUX receives band 2/3 RF energy from both port and starboard outboard enclosures.

Local oscillators and mixers are then used to divide the band 2/3 frequency spectrum into seven intermediate frequency subbands, each with a nominal 4 Ghz center frequency and a 2 Ghz bandwidth. Any signal within one of the seven subbands is forwarded to the IFM/coarse frequency receiver (IFM/CFR). [Ref.2:p.9]

In this thesis, the frequency spectrum is divided into eight intermediate frequency subbands, each with a nominal 5 GHz center frequency and a 2 GHz bandwidth.

c. Instantaneous Frequency Measurement/Coarse Frequency Receiver (IFM/CFR)

The IFM/CFR, which is also mounted inboard, receives a signal from the IFM/MUX on one of the eight intermediate frequency subbands. The received signal is amplified and then converted into a digital frequency word. The frequency word, along with its subband identification strobe, is sent to the display-processor group for correlation with the angle and amplitude data for the same emitter. Odd and even numbered subband signals are processed in separate paths. Each path has a separate instantaneous frequency measurement discriminator and encoder. [Ref.2:p.9]

In this simulation program frequency channels were not divided into even or odd numbered channel paths. Only one processing path was created to simplify the programming work.

B. DISPLAY-PROCESSOR GROUP

The inboard-mounted display-processor group interfaces with and provides control signals to both the receiver-antenna group and the ECM group. All countermeasures set control and display functions are performed within the display-processor group. The display-processor group receives emitter frequency, amplitude, AOA, and CW-emitter designation data from the receiver-antenna group. After time of arrival(TOA) data is appended, the received emitter data is formatted into pulse descriptor words(PDWs). These PDWs are used for real-time emitter activity sensing and AOA tracking in the Digital Tracking Unit(DTU). PDWs are input to and processed by the operational software, which analyzes the data and identifies possible sources of the emission based on emitter descriptions stored in system libraries. If the emitter is identified as hostile and operating in the band 3 frequency range, ECM may be initiated either via operator action or automatically by the operational software. The mode is operator selectable. The operational program selects an appropriate ECM group to establish an engagement. Engagement data is visually presented to the operator on the display and control console(DCC). [Ref.2:p.13]

1. Digital Presorter

The digital presorter receives and preprocesses the digitally encoded data from the receiver-antenna group. The

presorter is actually two separate functional entities housed in a single ship replaceable unit(SRU). The two functions are the direction frequency correlator and the digital tracking unit. [Ref.2:p.15]

The direction frequency correlator (DFC) receives digitally encoded band 2/3 emitter AOA, amplitude, and frequency data from the receiver-antenna group. Emitter AOA and amplitude data are encoded in the angle encoders independently of frequency data prior to arriving at the DFC. DFC timing assures that AOA and frequency data from any single emitter arrive simultaneously at the time coincidence correlator. The DFC correlates the emitter data, adds a TOA tag, formats a PDW identifying the emitter, and sends it to the digital tracking unit(DTU). [Ref.2:p.15]

The purpose of the DTU is to increase system emitter processing capacity by sorting and verifying PDWs at speeds much faster than could be accomplished by the digital processing unit(DPU). Emitter PDWs from the DFC are sorted by AOA and frequency and then stored in an emitter file memory (EFM). A new emitter is verified (considered firm) when three pulses have been identified in the same AOA frequency cell. New firm emitters are reported to the DPU, and they are fully reported only once. Subsequent reports contain AOA or emitter activity changes. The DPU uses the TOA data for pulse repetition interval (PRI) and scan calculations. The

amplitude data is used for scan analysis, PRI deinterleaving and data quality. [Ref.2:p.15]

2. Digital Processor Unit

The DPU is an upgraded general purpose computer. The computer includes an embedded disk used for high speed program loading, threat library storage and event recording. The memory stores the operations program, constants, and other data required for operation of the countermeasures set.

III. MODSIM II THE MODULAR, OBJECT-ORIENTED LANGUAGE

MODSIM II is the language in which model **Test** is written. The basic concepts will be covered in this chapter. This description touches on the several ways in which MODSIM II differs from traditional simulation languages.

A. MODULAR

MODSIM II programs may be divided into "modules". Each module is stored in a separate file. The advantages of this approach are that these modules may be compiled separately, saving time when only one of them is edited, and that a single module may serve multiple programs. This is because modules can import constructs and definitions from each other. The modular concept formalizes the notion of libraries of reusable code. [Ref.3:p.1]

B. OBJECT-ORIENTED

An "object" is an encapsulation of a data record which describes the state of the object and procedures called methods which describe its behaviors. Objects are more concrete than most programming constructs. They interact through a clearly defined protocol and the fields of an object instance are private. A new object type can inherit the attributes of an existing object type and elaborate on the

fields and methods of its ancestor type. Finally, objects are capable of polymorphism. A group of objects which share common ancestry can also share a method, yet each implements it differently. [Ref.3:p.1]

C. STRONGLY TYPED

Every expression, assignment statement and parameter is type checked at compile time for consistency. This eliminates errors which can go undiscovered until runtime in untyped languages. The concept of types also allows users to define their own types then declare variables of those types. [Ref.3:p.2]

D. BLOCK-STRUCTURED

A block is made up of declarations and executable statements. It may contain smaller blocks. The important feature of block-structured languages is that the scope or visibility of variables is restricted to the block in which they are declared and any subsidiary blocks. This control of scope of variables is fundamental to contemporary software engineering practices. [Ref.3:p.2]

E. SIMULATION

Simulation capabilities are provided in library modules. These modules provide direct support for all capabilities needed to program discrete-event simulation models. All MODSIM II objects have the capability of using Process

methods. A "Process" method is a method which can elapse simulation time. [Ref.3:p.2] A process might WAIT in simulation time and interact at specific simulation times with other processes. Also we can say that a WAIT statement elapses simulation time in a TELL method. The wait can be for a certain length of time (WAIT DURATION) or for an object to complete an action (WAIT FOR object TO {TELL method}). The key point is that only TELL methods can contain WAIT statements. In addition each WAIT construct has an optional ON INTERRUPT clause which is executed when the particular process of an object instance is interrupted. This allows a process that is dormant in a wait condition to be awakened before the completion of the WAIT condition.

IV. THE TEST SIMULATION MODEL

The Test model is a basic simulation of AN/SLQ-32 EW system made to answer specific questions about system performance, as measured by the ESM's probability of intercept in the midst of various electromagnetic signals, emitter identification capability and accurate ECM set-on. It is written in MODSIM II and designed to provide a foundation which can be used as a reference for system performance checks in sea trials and as an instructor's training tool for training R.O.C. Navy key personnel. This model will be upgraded in the near future to make it part of the R.O.C. Navy's wargame program.

A. OVERVIEW

The simulation program consists of 41 MODSIM II modules consisting of one main module and 20 paired definition and implementation modules. The main module is the name of the executable file and is called Test. The file naming convention used by MODSIM is that all MODSIM files will end in .MOD and be prefixed by a "M" if it is the main module, a "D" if it is a definition module, and an "I" if it is an implementation module. A main module contains the main program and is the name of the executable file created. A

definition module contains type and variable definitions that can be exported to other modules. An implementation module contains the actual code to implement the definition module. Each module is described in detail later on in this chapter.

B. SIMULATION EXECUTION

Execution of the simulation is straightforward. The user types "Test" to set up the simulation experiment, then the user will be asked " HOW MANY PULSETRAINS ?". In the present example the program was made by 15 emitters, so the user must input a number less than or equal to 15. The user will be asked again to input the PRI of each emitter. After this, the program will start to run. The results are presented in a file called DEBUG.OUT.

C. SIMULATION DESIGN

The design of the simulation is based on objects that correspond to their real world counterparts. There is a **pulsegenerator** which consists of an array of **PulseTrain** objects that correspond to unidentified signal emitters at sea. There are three objects in the ESM subsystem. The first one is called DFR which corresponds to the direction finding assembly of the ESM subsystem, the second one is called IFMMUX which corresponds to the ESM's frequency measurement assembly and the last one is called Presorter which corresponds to the signal processor assembly of the ESM subsystem. Each of these

objects is implemented as a separate module, consisting of a definition module and implementation module pair. These modules will be described in the following paragraphs.

1. Modules

a. Test

The Test main program module is contained in the file **MTest.mod**. It is a very simple main program. In this module a new **DFR** object, a new **IFMMUX** object, a new **DFC** object and a whole new array of **PulseTrain** objects are created. The basic flow of the simulation is that the user is asked to input both the number of **PulseTrain** operated in **pulsegenerator** and the **PRI** for each **PulseTrain**. Then enter a loop consisting of telling each **PulseTrain** object to **Radiate**, followed by **StartSimulation**. The simulation will execute until the simulation time reaches to **StopTime**.

b. Pulsegenerator

The **pulsegenerator** modules consists of the files **Dpulsegenerator.mod** and **Ipulsegenerator.mod**. There are two objects, **PulseObj** and **PulseTrainObj**, which are defined and implemented in these modules. The **PulseObj** is created to represent the single signal pulse. It has its own fields, which are angle, frequency, power and pulsewidth. Since the value of the fields of an object are modified only by its own methods, the **ASK** method **GetFiled** is created to perform this action.

The **PulseTrainObj** is set up for playing the part of a train of pulses that comes from the same emitter. The **PulseTrain** object first sets up the simulation when it is told to **Radiate** from the MAIN module. It does this by entering a loop in **TELL METHOD Radiate.** The **pulseTrain** generates a train of pulses after it is told to radiate. The pulses are produced one by one with the fixed time interval. Just like in the real world, the pulse generated by an emitter is separated by a certain amount of time between this pulse and the next one. This time interval is called pulse repetition interval (PRI). Once the pulse is produced, it is asked to go back to **PulseObj** to **GetFiled**, then the **pulseTrain** will **SendPulse.**

The way to represent that the pulse has been sent by the **pulsegenerator** is to ask **DFRObj** and **IFMMUXObj**, the direction finding subsystem and frequency measurement subsystem in AN/SLQ-32 system, to **ReceivePulse.**

There are an array of **PulseTrains** which have been created in main module, each one does the same job except having different parameters.

DFRinit and **IFMMUXinit** are two **ASK** methods, employed to be the bridges between the **pulsegenerator** and **DFRObj** and between **pulsegenerator** and **IFMMUXObj.** By these two bridges the messages from **pulsegenerator** can be received by **DFRObj** and **IFMMUXObj.**

c. DFR

The direction finding receiver(DFR) is one of the major subsystems of the AN/SLQ-32 system used to find the target direction. The **DFR** modules consist of the files **DDFR.MOD** and **IDFR.MOD**. There are five objects and related types defined and implemented in these modules. The five objects are BufferRecList, BufferBeamObj, BeamPortObj, BeamPortQueueObj and DFRObj, where only two of them own the method or methods to implement the actions. Their functions will be discussed in the following paragraphs.

The **BeamPortObj** is created to play the part of one beamport in the Multibeam Receiver Antenna. Each beamport has its own fields, orientation and gain, employed to define its unique characteristics and an **ASK** method **SetField** used to modify its fields.

The **DFRObj** is the heart of this subsystem and is used to find out the angle of signal received by the antenna. The azimuth coverage of the direction finding antenna is 360 degrees around the capital ship. The amount of energy received by the specific cells on the antenna is focused by a beam forming lens into separate beamports.

Every beamport provides 5.6 degrees of azimuth coverage with a different orientation. The orientation of the beamport is the center point of this beamport, for example the first beamport covers from 0 to 5.6 degrees and its orientation is at the 2.8 degree point.

The **BeamPortQueue** created in **ASK** method **ObjInit** is a collection of a series of **BeamPort**. Once a new **BeamPort** is created it is asked to go back to **BeamPortObj** to **SetField**. Then the well defined **BeamPort** is put in the **BeamPortQueue**.

The **DFRObj** is immediately asked to **ReceivePulse** after the **PulseTrain** object sends the pulse. When the pulse is received, the differing amounts of RF energy in each beamport are recorded in **BeamRec** by procedure **BuildBeamPortRec** contained in **BuildBeamPortRec** module.

The **BuildBeamPortRec** module is contained in the files **DBuildBeamPortRec.MOD** and **IBuildBeamPortRec.MOD**. It consists of only one procedure, called **BuildBeamPortRec**. This procedure is designed to create a **BeamRec** for each beamport with the signal amplitude received at this beamport. The orientation and amplitude are two elements of the beamrec's field. The orientation is the same as the orientation of **BeamPort** and the amplitude is derived by procedure **BeamPortGain**.

The amplitude measured in each beamport ($P_{BeamRec}$) is derived by Equations 4.1, 4.2 and 4.3.

$$\theta = |ANG\text{-}ORIENTATION| \quad (4.1)$$

$$G(\theta) = (1-\theta/5.6) \quad if\ \theta < 5.6$$
$$G(\theta) = 0 \quad otherwise \quad (4.2)$$

$$P_{BeamRec} = P_{Pulse} * G(\theta) \quad (4.3)$$

where

θ is the angle between the direction of arrival of the received pulse and the orientation of the beam port.

G(θ) is the gain of the beam port for the angle θ, the gain pattern can be easily replaced by altering **BeamPortGain**. If the θ is less than 5.6 degrees , the gain pattern is equal to $G(\theta)=(1-\theta/5.6)$, otherwise the gain of the beam port is zero.

$\mathbf{P_{pulse}}$ is the amplitude of the received pulses.

The **BeamRecs** provide amplitude information used by the **Angle Encoder** to determine emitter angle of arrival (AOA).

If the amplitude of **BeamRec** is higher than the threshold, this **BeamRec** is sent to the **BufferBeam** of **Angle Encoder**. The angle determination is done by **TELL** method **DetermineAOA**.

The first step of angle determination is finding pulse groups, this is done by procedure **FindPulseGroup** contained in **FindPulseGroup** module. In this procedure those **BeamRecs** having very close orientation and magnitudes higher than the threshold are picked from **BufferBeam** and put in **PulseGroup.**

In the **Angle Encoder** the emitter's angle of arrival provides two different voltages from two adjacent beamports. The angle encoder first selects the largest signal (V_{max}) then an offset voltage is generated (V_{max}-464 mv) which is compared to all beamport outputs. If the voltage difference between adjacent beamports is less than or equal to 464 mv (5.8dB), then a "cross-over" cell will be encoded. If the difference is greater than 464 mv, a "main-beam" cell will be encoded. The above procedure is done by the procedure **ProcessPulseGroup** contained in **ProcessPulseGroup** module.

The output of procedure **ProcessPulseGroup** is employed to create the **AngleRec** by the procedure **BuildAngleRec** which is contained in **BuildAngleRec** module. This **AngleRec** consists of the received pulse's AOA, amplitude, and time, which is sent to display-processor group (**PresorterObj**) by **ASK** method **DeliverRecord** for correlation with received pulse frequency data.

d. IFMMUX

The IFM receiver is the simplest, most mature technique for obtaining pulse-by-pulse frequency information over a broad frequency band. The **IFMMUX** module consists of the files **DIFMMUX.MOD** and **IIFMMUX.MOD**. There are three objects and related types defined and implemented in these modules. The three objects are **ChannelObj**, **ChannelQueueObj** and **IFMMUXObj**.

The IFM/MUX assembly reduces the whole signal frequency spectrum into eight discrete 2 GHz subbands (**Channels**). The received signal in each subband (**Channel**) is translated into the 4 to 6 Ghz Intermediate Frequency range by a mixer and a local oscillator. Since the subband (**Channel**) inputs are in the 4 to 6 Ghz range, the encoding frequency range is virtually identical within each subband. Thus, for actual frequency identification, each output frequency word is accompanied by a subband identification strobe.

The **ChannelObj** is created to represent the eight subbands in the frequency spectrum. The FreqStart and LOFreq are the two elements of its fields, and are used for subband identification and the frequency generated by local oscillator in this subband. The **ASK** method **SetField** is used to modify the object's fields, the LOFreq is derived by the procedure **DerivedLOFreq**, which is contained in **DerivedLOFreq** module.

The **ChannelQueueObj** is a collection of a series of **ChannelObj**, and it represents the whole frequency spectrum.

The new **Channel** is created in the **ASK** method **ObjInit** of the **IFMMUXObj**, and asked to go back to **ChannelObj** to **SetField**. The well defined **Channel** is put in **ChannelQueue**.

The **IFMMUX** is asked to **ReceivePulse** right after the **PulseTrainObj** has sent the pulse. The first step is to determine the channel with the strongest received signal then down-converter the Radio Frequency to 4 to 6 Ghz Intermediate Frequency by the procedure **RFDownConverted**, which is contained in **RFDownConverted** module. The output of this procedure is **ChannelRec**, which is used by the **TELL** method **MeasureFreq** to determine the frequency of the received pulse.

The digital instantaneous frequency measuring receiver uses a bank of frequency discriminators with the longest delay corresponding to the desired frequency resolution, while the shortest delay is determined by the highest frequency to be measured. [Ref.4:p.66]

The IFM COARSE FREQUENCY RECEIVER (IFM CFR) divides the instantaneous frequency signal received from the IFM/MUX into two paths, one having a delay line of known length and the other having zero delay. The signal passing down the delay line will experience a phase shift, with respect to the undelayed signal, which is a function of the input frequency. The two signals are applied to a phase correlator and envelope detector which form two video signals; one (V_c) proportional to the cosine of the phase shift and the other (V_s)

proportional to the sine of the phase shift. The phase angle for a monochromatic signal is given by $\theta = \omega_s \tau_d$, where ω_s is the signal's radian frequency and τ_d is the fixed delay time. The phase angle can be found by taking the $\tan^{-1}(V_s/V_c)$ of the ratio of the two quadrature video voltages. The intercepted signal's frequency can be found from the phase angle and the known delay as $f_s = \theta/2*\pi*\tau_d$. [Ref.4:p.65]

The discriminator of this simulated system is operated with a delay time (τ), 0.000045 μs, which was chosen to ensure the quadrature video voltages are unambiguous. The unit of frequency used in this model is MHz.

Equations 4.4, 4.5 and 4.6 are used to determine the quadrature voltages, **E** and **F**, of the IFM coarse frequency receiver.

$$\theta = 2*\pi*IF*\tau \qquad (4.4)$$

$$E = A*\sin(\theta) \qquad (4.5)$$

$$F = A*\cos(\theta) \qquad (4.6)$$

where

θ is the phase angle for the signal.

τ is the time delay of frequency discriminator.

IF is the instantaneous frequency.

A is the amplitude of the received pulse.

The phase angle can be found by measuring these two voltages, E and F, and taking the $\tan^{-1}(E/F)$. The frequency of the IF signal can be derived from the phase angle and the known delay as $IF = \theta/2*\pi*\tau$. Finally the RF can be encoded by adding the derived IF and frequency produced by the local oscillator. All of this care is taken in producing IF measurements which could contain random voltage measurement errors. This measurement error introduction has not yet been implemented in **Test** model.

The radio frequency and time of pulse received are used to create **FreqRec** by the procedure **BuildFreqRec**, which is contained in **BuildFreqRec** module. The **FreqRec** is sent to the display-processor group where they are correlated with related AOA and amplitude data.

e. Presorter

The Digital Presorter interfaces with and provides control signals to both the receiver-antenna group (**DFRObj** and **IFMMUXObj**) and the ECM group. The **Presorter** module consists of the files **DPresorter.MOD** and **IPresorter.MOD**. There are five objects and related types defined and implemented in these modules. The five objects are **BufferRecList**, **PulseRecBufferObj**, **DFCObj**, **DTUObj**, and **DPUObj**.

The **DFCObj** (Direction Frequency Correlator) receives emitter frequency, amplitude and AOA data from the

receiver-antenna group by the **ASK** methods **ReceiveARecord** and **ReceiveFRecord.**

The received frequency data (**FreqRec**), amplitude and AOA data (**AngleRec**) are formatted into **PulseRecs** by the procedures **BuildPulseFreqRec** and **BuildPulseAngRec.** These **PulseRecs** are put into **PulseRecBuffer** and sent to the correlator.

DFC timing assures that AOA and frequency data from any single emitter arrive simultaneously at the time coincidence correlator.

The correlated emitter data are formatted into pulse descriptor data words (PDWs), put into **PDWBuffer** and sent to Digital Tracking Unit (**DTUObj**).

The **DTU** sorts PDWs from the **DFC** into the Emitter File Memory(**EFM**) based on relative AOA and frequency, and is performed by the **ASK** method **BufferSort.**

The emitter is not reported to the Digital Processing Unit (**DPUObj**) until three pulses have been identified in the same **EFM** cell within a time interval. The **ASK** method **DeclareContact** is used to notify **DPU** that a new emitter is present.

Once a new emitter is reported, then the **DPU** requests the **DTU** to store subsequent pulses of that emitter over a time period sufficient to provide enough pulses for further analysis.

After a time interval (**ProcessInterval**), the records of stored pulses are sent to the computer by the **TELL** method **Process** to calculate the emitter's parameters.

The parameter calculation is performed by the **ASK** method **FindParameter**. Before the execution of the calculation, the match of stored pulse records with the record of new emitter must be done. Then the pulse repetition interval (PRI) and type of scan are calculated. These parameters, along with frequency, are usually sufficient to characterize an emitter.

The emitter identification is completed by comparing the observed signal characteristics with the parameters in a library stored in the computer memory (**EmitterLib.dat**).

The procedure **FindSHRec** contained in the **FindSH** module is used to search through the library (**EmitterLib.dat**) for the emitter having the same characteristics with the observed signal.

The process of the formation of **EmitterLib.dat** is done by the procedures **ReadEmAll**, **ReadEmitterLib** (contained in **Input** module), **ReadLst** (contained in **ReadLst** module) and the procedure **ReadSH** (contained in **ReadSH** module).

The procedures **ReadEmAll** and **ReadEmitterLib** are employed to locate the **EmitterLib.dat** in computer memory.

The procedure **ReadLst** is used to determine emitters' data are stored in this file. Reading each emitter's data is performed by procedure **ReadSH.**

The module **RGlobals** is contained in the file **DRGlobals.MOD.** This module defines the global types and the variable used in the formation of **EmitterLib.dat** and emitter identification.

The emitter data file, **EmitterLib.dat**, is a kind of **ARRAY** data type. The first line of the file is an integer showing the number of the emitter's data stored, and the following lines show each emitter's name, parameters, nationality and the techniques which will be used as a countermeasure.

If the observed signal is identified as "HOSTILE" and its frequency is in 8 GHz to 12 GHz, The **DPU** will initiate the electronic countermeasure by the **ASK** method **InitiateECM.**

f. ECM

The information of ECM subsystem of AN/SLQ-32 system is classified, so this module will not be built. Instead, a dummy object **ECMObj** was created to receive the command from **DPUObj.**

2. Model Output

The output of the simulation is put in the **debug.out** file not only to show the results, but also display the process of the simulation. If the user has a problem

understanding the program, reading through this file will be helpful.

V. THE ALGORITHMS OF TEST SIMULATION MODEL

The algorithms of this simulation program are divided into two major parts: the algorithms of EMITTERS and the algorithms of AN/SLQ-32 EW system.

A. THE ALGORITHMS OF EMITTERS

In order to let the AN/SLQ-32 EW system receive signals, creation of several friendly or hostile EMITTERS is necessary. In this program an array of PulseTrains are produced to represent EMITTERS on the sea or in the air.

Each EMITTER has its own parameters showing its distinctive characteristics. Once the EMITTER is asked to radiate, the AN/SLQ-32 EW system will receive the pulses, then the following procedures can be executed.

B. THE ALGORITHMS OF EW SYSTEM

The AN/SLQ-32 EW system has two kinds of antennas to sense the pulses: Direction Finding antenna and Semi-Omni antenna.

The information of arriving angle and amplitude of pulse are sent to the Direction Finding Receiver(DFR) which is connected to the DF antenna.

The pulse frequency information is sent to the Instantaneous Frequency Measuring Receiver(IFM) which is joined to the semi-omni antenna.

The angle and amplitude data from the DFR and the coarse frequency data from the IFM are sent to the Presorter for correlation.

The algorithm of each assembly will be described in the following paragraphs.

1. DFR

The amplitude and angle data from the multibeam Direction Finding antenna are sensed by the Direction Finding Receiver (DFR). Only data from the two beamports which have the highest and the second highest amplitude are formatted into beam records (BeamRec) and collected by bufferbeam.

If there are more than one pulse from different emitters which arrive at the same time, the DFR only receives one of them which appears first in simulation and blocks the rest.

The records in the bufferbeam are sent to the angle encoder to determine the pulse angle of arrival.

In the angle encoder the records in the bufferbeam are checked to ensure they arrive at the same time. After the time checking, the records are put into the pulsegroup and are ready to be processed.

The angle record is created after the pulsegroup is processed. The data in the angle record are the calculated pulse angle, amplitude and time tag.

2. IFM

The 2 to 18 GHz frequency band is divided into eight 2 GHz band width subbands(channels). Each channel is down converted to a 4 to 6 GHz Intermediate Frequency (IF) by a mixer and a local oscillator.

Only one of the eight channels can sense the signal received by the semi-omni antenna. The channel which has signals is processed and a channel record is created.

The data in a channel record are the channel strobe, which is used to identify the frequency range of this channel, and the IF, which is employed to measure the pulse frequency.

The delay time(τ) of the discriminator is 0.000045 μs, which was chosen to ensure the quadrature voltages are unambiguous. The unit of frequency in this program is MHz.

Equations 4.4, 4.5 and 4.6 in chapter 4 are used to figure the two measured quadrature video voltages of the IFM coarse frequency receiver. One voltage (E) is proportional to the sine of the phase shift ($\theta=2*\pi*\tau$) and the other (F) is proportional to the cosine of the phase shift.

The phase angle can be found by taking the $\tan^{-1}(E/F)$ of the ratio of the two quadrature video voltages. The frequency of the IF signal can be found from the phase angle and the known delay as IF = $\theta/2*\pi*\tau$.

The Radio Frequency (RF) can be encoded by adding the derived IF and frequency produced by the local oscillator. The frequency record is created after the RF is derived.

The data in a frequency record are the calculated pulse frequency and time tag.

3. Presorter

The frequency record from the IFM is correlated with the related angle record from the DFR in Direction Frequency Correlator (DFC). The matched records are formatted into a pulse descriptor data word(PDW). The PDWs are collected at a PDWbuffer which is sent to Digital Tracking Unit (DTU) for emitter sorting.

The DTU sorts the PDWbuffer at a preprogrammed time interval. The PDWs with the same frequency and angle data are stored in the emitter file memory (EFM).

The reporting of "a new emitter is present" to the Digital Processing Unit(DPU) is executed when 3 or more pulses from the same emitter are found in the same EFM.

The DPU asks the DTU to store the subsequent pulses of that emitter over a time period sufficient to provide enough pulses for parameter calculation. During this time period there may be more than one emitter reported. A reference pulse record is put into a reference pulse buffer whenever an emitter is reported.

Reference pulses are kept for each active emitter which the system has already observed. Matching the pulses received after the emitter is reported with the reference pulse records, which are stored in the reference pulse buffer, is done before starting parameter calculation.

The difference between time tags of pulse records from the same emitter is used to calculate the pulse repetition interval (PRI). The amplitude of the pulses are employed to find out the emitter's type of scan. In this program only one type of scan, fixed direction antenna, is created. These parameters, along with frequency, are usually sufficient to characterize an emitter.

Emitter identification is completed by comparing the observed signal characteristics with the parameters of possible emitters in the emitter data file (EmitterLis.dat).

The electronic countermeasures are initiated when the hostile emitter is identified and its frequency falls in the 8 to 12 GHz region.

VI. SIMULATION ANALYSIS

This simulation is a terminating simulation. The mission will end when the preprogrammed Stoptime is reached.

One of the major purposes for establishing this simulation model is to generate an AN/SLQ-32 EW system performance analysis program. The structure of this program is founded on the system's operational behavior and capability gathered from the system manufacturer, **Raytheon Company**.

In this simulation program, random error was not employed in a system's measurement or the probability of detection. The buyer was told the AN/SLQ-32 EW system has extremely high probability of detection, accurate emitter identification and accurate ECM set-on. So in this simulation model, the simulated system is performing with almost perfect results. The only performance trait that can be measured is congestion effect in various record buffers in the system.

A. MODEL VERIFICATION

The model has been verified to work correctly in the environment having up to 35 different emitters. The program runs correctly and the output has been examined and is believed to be correct. The model has not been validated as

this would require comparing its output with the results obtained from actual system sea trials.

In the future, when the results of the system sea trials are obtained, they will be compared with the output of this program which is produced in a simulated electromagnetic environment that is similar to the one in real sea trial.

Since it has not been validated, it has also not been calibrated and some results of the parameter calculation should not be considered as being absolutely precise. It is the intention to validate the model using data obtained from actual system sea trials in the future.

B. TERMINATION CONDITIONS

The simulation is terminated when the simulation time (SimTime) reaches or just passes over the stop time. The emitters transmit pulses by turns in different time intervals (PRIs). Each turn before the pulse is sent, the current simulation time is checked and compared with the preprogrammed stop time. If the current simulation time is less than the stop time, the pulse is sent immediately. Otherwise, this emitter is shut down.

The simulation termination is completed only when all emitters are shut down. Due to emitters having different pulse transmitting intervals, they are stopped at different times.

C. PROBLEMS ENCOUNTERED

The multiple concurrent processes that can occur in the simulation language MODSIM II are both a useful feature and a source of problems.

The processes are a useful feature in that they allow one object to be performing many different activities in the form of concurrent methods at the same time. This greatly simplifies the code that is needed in situations where this concurrence is applicable.

Problems arise when pulses collide, because the EW system is designed to process one pulse at a time. It is possible for pulses to collide in the real system, in which case the real system measures only the strongest pulse. However, MODSIM's timing routine cannot judge pulse strength to order the pulses' arrival by signal strength, since pulses are received and processed instantaneously. So the code could not be written to pass only the signal with the highest amplitude. The only way to keep the simulated system processing one pulse at a time is by receiving the first pulse appearing in simulation and blocking the remaining pulses. This is a small draw-back in this simulation model.

VII. CONCLUSIONS AND RECOMMENDATIONS

A. OBSERVATIONS

1. AN/SLQ-32 EW System

The performance of AN/SLQ-32 ESM subsystem can be analyzed by this simulation program. This is an advantage in operator training and system introduction. People can see what is going on in the system's signal processing and subsequent response on the program's output and without having to use the real system. That makes training more cost effective.

2. Object-Oriented, Process-Based Simulation

Object-oriented programming delivers on its promises in a simulation environment. The simulation code was easy to write, debug, maintain, and enhance in nearly all situations encountered. The process-based simulation concept simplified the design and maintenance of this large model.

B. SUGGESTIONS FOR FURTHER RESEARCH

1. The next logical step in the research of this model is to set up the simulated ECM subsystem, expand the emitters data file and modify the current program if it is necessary. That will make this simulation model more realistic.

2. It would be useful to validate the model against real world data by setting up a real test with actual AN/SLQ-32 EW

system and targets. The actual probabilities of correctly identifying signals can be measured and compared with the values determined in **Test** model. This would allow the model to be calibrated so that it would accurately simulate measurement of the target parameters.

3. The difference between sea trial results and the simulation output can be used as a reference for system performance checks. Since the simulation is operating in a perfect condition, this difference can also be employed to study the reasons for failures in intercepting signals and jamming hostile targets.

4. Wargaming and electronic warfare are two very new concepts in The Republic Of China Navy. It's hard to educate the fleet sailor who operates the AN/SLQ-32 EW SYSTEM with respect to EW when they can not see the effect of electronic warfare during an at-sea exercise. The techniques of electronic countermeasures are normally classified and the ECM system is rarely operated in an active mode during exercises for reasons of security. That makes the situation even worse. Currently there is no wargame model that can be used to show various engagement results focusing on the effects of electronic warfare techniques. This research is going to be extremely helpful in solving these problems. Building a wargame model which includes electronic warfare for the Republic Of China Navy is a goal to be reached in the future.

C. CONCLUSIONS

The AN/SLQ-32 ESM subsystem was successfully modeled by **Test** but there remains much work to be done to make the model more realistic and user friendly.

This simulation model as well as MODSIM II programming language will be introduced to the Republic Of China Navy in the near future.

APPENDIX Test SIMULATION PROGRAM

```
MAIN MODULE Test;
{-------------------------------------------------------------}

DESCRIPTION:
This is the main module of an AN/SLQ-32 EW SYSTEM simulation
model.

{-------------------------------------------------------------}
FROM pulsegenerator IMPORT EmitterArrayType;
FROM DFR IMPORT DFRObj;
FROM IFMMUX IMPORT IFMMUXObj;
FROM Presorter IMPORT DFCObj;
FROM DebugRun IMPORT SetUpD;
FROM SimMod IMPORT SimTime;
FROM SimMod IMPORT StartSimulation;

    VAR
        EmitterArray:EmitterArrayType;
        DFR          :DFRObj;
        IFMMUX       :IFMMUXObj;
        DFC          :DFCObj;
        N,i          :INTEGER;
        PRI          :REAL;

    BEGIN
        SetUpD(TRUE);
        OUTPUT("How many PulseTrains ?");
        INPUT(N);
        NEW(EmitterArray, 1..N);
        NEW(DFR);
        NEW(IFMMUX);
        NEW(DFC);
        FOR  i:=1 TO N
             OUTPUT("PRI FOR #",i);
             INPUT(PRI);
             NEW(EmitterArray[i]);
             ASK EmitterArray[i] DFRinit(DFR);
             ASK EmitterArray[i] IFMMUXinit(IFMMUX);
             ASK DFR DFCinit(DFC);
             ASK IFMMUX DFCinit(DFC);
             TELL EmitterArray[i] TO Radiate(i,PRI);
         END FOR;
         StartSimulation;
    END MODULE.
```

```
DEFINITION MODULE pulsegenerator;
{-------------------------------------------------------------}

DESCRIPTION:
This is a definition module of the Test simulation that
defines the EMITTER (pulsegenerator) Object.

{-------------------------------------------------------------}
FROM DFR IMPORT DFRObj;
FROM IFMMUX IMPORT IFMMUXObj;


EXPORTTYPE
        PulseObj = OBJECT; FORWARD;
        PulseTrainObj = OBJECT; FORWARD;
TYPE

        PulseObj = OBJECT;

                 Angle      :REAL;
                 Freq       :REAL;
                 Power      :REAL;
                 PulseWidth :REAL;
                 ASK METHOD GetField(IN i:INTEGER);
        END OBJECT; {Pulse}
        PulseTrainObj = OBJECT;
               pulse      :  PulseObj;
               StopTime   :  REAL;
               Angle      :  REAL;
               ScanRate   :  REAL;
               Power      :  REAL;
               DFR        :  DFRObj;
               IFMMUX     :  IFMMUXObj;
               ASK METHOD ObjInit;
               TELL METHOD Radiate(IN i:INTEGER;
                                   IN PRI : REAL);
               ASK METHOD DFRinit(IN DFRNEW:DFRObj);
               ASK METHOD IFMMUXinit(IN
                                     IFMMUXNEW:IFMMUXObj);
               ASK METHOD SendPulse(IN pulse : PulseObj);
         END OBJECT;  {PulseTrain}

        EmitterArrayType=ARRAY INTEGER OF PulseTrainObj;


END  MODULE.
```

```
IMPLEMENTATION MODULE pulsegenerator;
{-------------------------------------------------------------}

DECRIPTION:
This is an implementation module of the Test simulation that
implements the EMITTER (pulsegenerator) Object.

{-------------------------------------------------------------}
FROM SimMod IMPORT Interrupt;
FROM DFR IMPORT DFRObj;
FROM IFMMUX IMPORT IFMMUXObj;
FROM Debug IMPORT TraceStream;
FROM SimMod IMPORT SimTime;



OBJECT PulseObj;
{-------------------------------------------------------------}
ASK METHOD GetField(IN i:INTEGER);
{-------------------------------------------------------------}
BEGIN
       IF (i=1)
           Power:=20.0;
           Angle:=20.0;
           Freq :=9566.2;

       ELSIF (i=2)
           Power:=30.0;
           Angle:=135.0;
           Freq:=11000.0;

       ELSIF (i=3)
           Power:=25.0;
           Angle:=55.0;
           Freq:=8500.0;

       ELSIF (i=4)
           Power:=25.0;
           Angle:=5.0;
           Freq:=8900.0;
         ELSIF (i=5)
           Power:=25.0;
           Angle:=95.0;
           Freq:=15900.0;

       ELSIF (i=6)
           Power:=25.0;
           Angle:=82.0;
           Freq:=10900.0;

       ELSIF (i=7)
```

```
            Power:=25.0;
            Angle:=195.0;
            Freq:=7900.0;
          ELSIF (i=8)
            Power:=25.0;
            Angle:=215.0;
            Freq:=9900.0;

         ELSIF (i=9)
            Power:=25.0;
            Angle:=315.0;
            Freq:=9500.0;

         ELSIF (i=10)
            Power:=25.0;
            Angle:=355.0;
            Freq:=9700.0;
          ELSIF (i=11)
            Power:=25.0;
            Angle:=335.0;
            Freq:=9800.0;

         ELSIF (i=12)
            Power:=25.0;
            Angle:=235.0;
            Freq:=6900.0;

         ELSIF (i=13)
            Power:=25.0;
            Angle:=31.0;
            Freq:=9400.0;

         ELSIF (i=14)
            Power:=25.0;
            Angle:=144.0;
            Freq:=4900.0;
           ELSE
            Power:=25.0;
            Angle:=191.0;
            Freq:=4900.0;


         END IF;
END METHOD;

END OBJECT; {PulseObj}

OBJECT PulseTrainObj;
```

```
{-------------------------------------------------------}
ASK METHOD ObjInit;
{-------------------------------------------------------}
BEGIN
     StopTime:=105.0;
END METHOD;

{-------------------------------------------------------}
TELL METHOD Radiate(IN i:INTEGER;
                    IN PRI : REAL);
{-------------------------------------------------------}
    BEGIN
    LOOP
          WAIT DURATION  PRI
           ON INTERRUPT
              TERMINATE
          END WAIT;

          IF (SimTime()>StopTime)
               TERMINATE;
          END IF;
             ASK TraceStream TO WriteString ("New Pulse" +
                           "AT" +  REALTOSTR(SimTime()));
             ASK TraceStream TO WriteLn;
             NEW(pulse);
             ASK pulse GetField(i);
             ASK SELF SendPulse(pulse);
     END LOOP;
     END METHOD;

{-------------------------------------------------------}
ASK METHOD DFRinit(IN DFRNEW:DFRObj);
{-------------------------------------------------------}
   BEGIN
          DFR:=DFRNEW;
   END METHOD;
{-------------------------------------------------------}
ASK METHOD IFMMUXinit(IN IFMMUXNEW:IFMMUXObj);
{-------------------------------------------------------}
   BEGIN
          IFMMUX:=IFMMUXNEW;
   END METHOD;
{-------------------------------------------------------}
ASK METHOD SendPulse (IN pulse : PulseObj);
{-------------------------------------------------------}
    BEGIN
          ASK  DFR TO ReceivePulse(pulse);
          ASK  IFMMUX TO ReceivePulse(pulse);
    END METHOD;
    END OBJECT; {PulseTrainObj}
    END MODULE.
```

```
DEFINITION MODULE DFR;
{-------------------------------------------------------------}

DESCRIPTION:
This is a definition module of the Test simulation that
defines the Direction Finding Receiver (DFR) Object.

{-------------------------------------------------------------}
FROM GrpMod  IMPORT QueueObj;
FROM ListMod IMPORT QueueList;
FROM pulsegenerator IMPORT PulseTrainObj;
FROM pulsegenerator IMPORT PulseObj;
FROM Presorter IMPORT DFCObj;

EXPORTTYPE
        DFRObj = OBJECT; FORWARD;
        AngleRecType = RECORD; FORWARD;

TYPE
        BufferBeamRecType=RECORD
                orientation : REAL;
                power       : REAL;
                time        : REAL;
        END RECORD;

        BufferRecList=OBJECT(QueueList[ANYREC :
                    BufferBeamRecType]);
        END OBJECT;

        BufferBeamObj=OBJECT (QueueList [ANYREC :
                    BufferBeamRecType]);
        END OBJECT;

        BeamPortObj=OBJECT;
                 orientation : REAL;
                 gain        : REAL;
                 ASK METHOD SetField(IN
                           BeamPortOrientation:REAL);
        END OBJECT;

        BeamPortQueueObj=OBJECT(QueueObj
                              [ANYOBJ:BeamPortObj]);
        END OBJECT;

        AngleRecType=RECORD
                 ANGLE       : REAL;
                 Time        : REAL;
                 Power       : REAL;
                 Power1      : REAL;
        END RECORD;
```

```
        DFRObj=OBJECT;

                threshold     : REAL;
                AngleRec      : AngleRecType;
                DFC           : DFCObj;
                I             : INTEGER;
                BeamPort      : BeamPortObj;
                BeamPortOrientation:REAL;
                BufferBeam    : BufferBeamObj;
                BeamPortQueue :BeamPortQueueObj;

                TELL METHOD DetermineAOA(IN
                                BufferBeam:BufferBeamObj);
                ASK METHOD DFCinit(IN DFCNEW:DFCObj);
                ASK METHOD DeliverRecord(IN
                                AngleRec:AngleRecType);
                ASK METHOD  ObjInit;
                ASK METHOD  ReceivePulse(IN pulse :
                                PulseObj);
        END OBJECT;

END MODULE.
```

```
IMPLEMENTATION MODULE DFR;
{-------------------------------------------------------------}

DESCRIPTION:
This is a implementation module of the Test simulation that
implements the Direction Finding Receiver Object.

{-------------------------------------------------------------}
FROM SimMod IMPORT SimTime;
FROM pulsegenerator IMPORT PulseObj;
FROM Presorter IMPORT DFCObj;
FROM BuildBeamPortRec IMPORT BuildBeamPortRec;
FROM FindPulseGroup IMPORT FindPulseGroup;
FROM ProcessPulseGroup IMPORT ProcessPulseGroup;
FROM BuildAngleRec IMPORT BuildAngleRec;
FROM Debug IMPORT TraceStream;
FROM Globals IMPORT RECEIVEPULSE;

OBJECT BeamPortObj;
{-------------------------------------------------------------}
ASK METHOD SetField(IN BeamPortOrientation: REAL);
{-------------------------------------------------------------}
BEGIN
        orientation:=BeamPortOrientation;
END METHOD;
END OBJECT;{BeamPortObj}

OBJECT DFRObj;
{-------------------------------------------------------------}
ASK METHOD ObjInit;
{-------------------------------------------------------------}
BEGIN
NEW(BeamPortQueue);
FOR I := 0 TO 3600 BY 56
     NEW(BeamPort);
     BeamPortOrientation:=(FLOAT(I))/10.0 + 2.8;
     ASK BeamPort SetField(BeamPortOrientation);
     ASK BeamPortQueue TO Add(BeamPort);
END FOR;
NEW(BufferBeam);
END METHOD;

{-------------------------------------------------------------}
ASK METHOD ReceivePulse(IN pulse:PulseObj);
{-------------------------------------------------------------}
      VAR
            ANG    :  REAL;
            P      :  REAL;
            BeamRec : BufferBeamRecType;
```

```
      BEGIN
            RECEIVEPULSE:=FALSE;
            IF (BufferBeam.numberIn < 2)
              RECEIVEPULSE:=TRUE;
              ANG:=ASK pulse Angle;
              P:=ASK pulse Power;
              BeamPort:=ASK BeamPortQueue First();
              WHILE  NOT (BeamPort=NILOBJ)
                     BuildBeamPortRec(BeamPort,ANG,P,BeamRec);
                      BeamPort:=ASK BeamPortQueue
                                              Next(BeamPort);
                      IF (BeamRec.power > 0.1) AND
                                    (BufferBeam.numberIn < 2)
                         ASK BufferBeam TO Add(BeamRec);
                      END IF;
               END WHILE;
                 OUTPUT("I AM HERE");
                 TELL (SELF) TO DetermineAOA(BufferBeam);
             END IF;
       END METHOD;

{-------------------------------------------------------------}
TELL METHOD DetermineAOA(IN BufferBeam:BufferBeamObj);
{-------------------------------------------------------------}
       VAR
              PulseGroup                    : BufferRecList;
              ANG,Power,Power1,time          : REAL;
       BEGIN
               FindPulseGroup(BufferBeam,PulseGroup);
               ProcessPulseGroup
                  (PulseGroup,ANG,time,Power,Power1);
              BuildAngleRec(ANG,time,Power,Power1,AngleRec);
               IF AngleRec<> NILREC
                  ASK(SELF) DeliverRecord(AngleRec);
               END IF;
       END METHOD;

{-------------------------------------------------------------}
ASK METHOD DFCinit(IN DFCNEW : DFCObj);
{-------------------------------------------------------------}
       BEGIN
               DFC:=DFCNEW;
       END METHOD;
{-------------------------------------------------------------}
ASK METHOD DeliverRecord(IN AngleRec:AngleRecType);
{-------------------------------------------------------------}
        BEGIN
               ASK DFC TO ReceiveARecord(AngleRec);
        END METHOD;
        END OBJECT;
        END MODULE.
```

```
DEFINITION MODULE BuildBeamPortRec;

FROM DFR IMPORT BufferBeamRecType,BeamPortObj;


PROCEDURE BuildBeamPortRec(IN BeamPort:BeamPortObj;
                           IN ANG:REAL;
                           IN P  :REAL;
                           OUT BeamRec : BufferBeamRecType);

END MODULE.
{-------------------------------------------------------------}

IMPLEMENTATION MODULE BuildBeamPortRec;

FROM DFR IMPORT BufferBeamRecType,BeamPortObj;
FROM BeamPortGain IMPORT BeamPortGain;
FROM SimMod IMPORT SimTime;
FROM Debug IMPORT TraceStream;


PROCEDURE BuildBeamPortRec(IN BeamPort:BeamPortObj;
                           IN ANG:REAL;
                           IN P  :REAL;
                           OUT BeamRec : BufferBeamRecType);

      VAR
            theta     :    REAL;
        Orientation :    REAL;
        AntennaGain :    REAL;

      BEGIN
              Orientation:=ASK BeamPort orientation;
              ASK TraceStream TO WriteString
                          ("BeamPortorientation" + "is" +
                            REALTOSTR(Orientation));
              ASK TraceStream TO WriteLn;
              theta:=ABS(ANG-Orientation);
              ASK TraceStream TO WriteString ("theta" + "is"
                                        +REALTOSTR(theta));
              ASK TraceStream TO WriteLn;
              NEW(BeamRec);
              BeamRec.orientation:= Orientation;
              BeamRec.power:=P*(BeamPortGain(theta));
              ASK TraceStream TO WriteString ("BeamRecPower"
               + "is" + REALTOSTR(P*(BeamPortGain(theta))));
              ASK TraceStream TO WriteLn;

              BeamRec.time:=SimTime();
       END PROCEDURE;
       END MODULE.
```

```
DEFINITION MODULE BeamPortGain;

PROCEDURE BeamPortGain(IN theta:REAL):REAL;
END MODULE.

{-------------------------------------------------------------}
IMPLEMENTATION MODULE BeamPortGain;

PROCEDURE BeamPortGain(IN theta:REAL):REAL;


      BEGIN

           IF (theta>5.6)

              RETURN 0.0;
           ELSE

              RETURN(1.0-(theta/5.6));
           END IF;

      END PROCEDURE;
      END MODULE.
```

```
DEFINITION MODULE FindPulseGroup;

FROM DFR IMPORT BufferBeamObj ,BufferRecList;

PROCEDURE FindPulseGroup(IN BufferBeam:BufferBeamObj;
                         OUT PulseGroup:BufferRecList);
END MODULE.

IMPLEMENTATION MODULE FindPulseGroup;

FROM DFR IMPORT BufferBeamObj,BufferRecList,BufferBeamRecType;
FROM Debug IMPORT TraceStream;

PROCEDURE FindPulseGroup(IN BufferBeam:BufferBeamObj;
                         OUT PulseGroup:BufferRecList);
        CONST
               threshold     = 0.1;
               timetol       = 0.01;
               orientol      = 5.8;
        VAR
                rec,recLast   :  BufferBeamRecType;
                time,power                :  REAL;
                power1,orientation        :  REAL;
        BEGIN
                NEW(PulseGroup);
                rec:=ASK BufferBeam Remove();
                time:=rec.time;
                orientation:=rec.orientation;
                OUTPUT("ORIENREF",orientation);
                recLast:=ASK BufferBeam Last();
         WHILE(rec<>NILREC)
         IF((rec.power>threshold)AND((time-rec.time)<timetol)
              AND(ABS(orientation-rec.orientation)<orientol))
                ASK PulseGroup TO Add(rec);
                         IF (rec=recLast)
                             EXIT;
                         END IF;
                rec:=ASK BufferBeam Remove();

         ELSE
                ASK BufferBeam TO Add(rec);
                          IF (rec = recLast)
                              EXIT;
                          END IF;
                rec:=ASK BufferBeam Remove();
         END IF;
         END WHILE;

END PROCEDURE;
END MODULE.
```

```
DEFINITION MODULE ProcessPulseGroup;

FROM DFR IMPORT BufferRecList;

PROCEDURE ProcessPulseGroup(IN PulseGroup:BufferRecList;
                            OUT ANG : REAL;
                            OUT time : REAL;
                            OUT Power:REAL;
                            OUT Power1:REAL);
END MODULE.

{-------------------------------------------------------------}
IMPLEMENTATION MODULE ProcessPulseGroup;

FROM DFR IMPORT DFRObj, BufferRecList, BufferBeamRecType;
FROM FindPulseGroup IMPORT FindPulseGroup;
FROM Debug IMPORT TraceStream;
FROM SimMod IMPORT SimTime;

PROCEDURE ProcessPulseGroup(IN PulseGroup:BufferRecList;
                            OUT ANG : REAL;
                            OUT time : REAL;
                            OUT Power:REAL;
                            OUT Power1:REAL);
        CONST
                VoltageGap  =  5.8;

        VAR
                PulseRec  :  BufferBeamRecType;
                theta     :  REAL;
                Orientation : REAL;
                Orientation1  : REAL;
        BEGIN
                 PulseRec:=ASK PulseGroup First();
                 Power:=PulseRec.power;
                 Orientation:=PulseRec.orientation;
                 time:=PulseRec.time;
                 PulseRec:=ASK PulseGroup Next(PulseRec);
                 Power1:=PulseRec.power;
                 Orientation1:=PulseRec.orientation;
                 IF (ABS(Power-Power1)>VoltageGap)
                    IF (Power>Power1)
                       ANG:=Orientation;
                    ELSE
                       ANG:=Orientation1;
                    END IF;
                 ELSE
                    ANG:=(Orientation+Orientation1)/2.0;
                 END IF;
         END PROCEDURE;
         END MODULE.
```

```
DEFINITION MODULE BuildAngleRec;

FROM DFR IMPORT AngleRecType;

PROCEDURE BuildAngleRec(IN ANG:REAL;
                        IN time:REAL;
                        IN Power:REAL;
                        IN Power1:REAL;
                        OUT AngleRec:AngleRecType);
END MODULE.

{-------------------------------------------------------------}
IMPLEMENTATION MODULE BuildAngleRec;

FROM DFR IMPORT AngleRecType;

PROCEDURE BuildAngleRec(IN ANG:REAL;
                        IN time:REAL;
                        IN Power:REAL;
                        IN Power1:REAL;
                        OUT AngleRec:AngleRecType);

            BEGIN
                 NEW(AngleRec);
                    AngleRec.ANGLE:=ANG;
                    AngleRec.Time:=time;
                    AngleRec.Power:=Power;
                    AngleRec.Power1:=Power1;

            END PROCEDURE;
            END MODULE.
```

```
DEFINITION MODULE IFMMUX;
{-------------------------------------------------------------}

DESCRIPTION:
This is a definition module of Test simulation that defines
Frequency Measurement Receiver (IFMMUX) Object.

{-------------------------------------------------------------}
FROM GrpMod IMPORT QueueObj;
FROM pulsegenerator IMPORT PulseTrainObj,PulseObj;
FROM Presorter IMPORT DFCObj;

EXPORTTYPE
        IFMMUXObj = OBJECT; FORWARD;
        FreqRecType=RECORD; FORWARD;
TYPE
        ChannelRecType=RECORD
              Freq,LO,Time  :   REAL;
        END RECORD;

        ChannelObj=OBJECT;
              FreqStart :  REAL;
              LOFreq    :  REAL;
              ASK METHOD SetField(IN ChannelFreqStart:REAL);
        END OBJECT;

        ChannelQueueObj=OBJECT(QueueObj[ANYOBJ:ChannelObj]);
        END OBJECT;

        FreqRecType=RECORD
              Freq  :  REAL;
              Time  :  REAL;
        END RECORD;

        IFMMUXObj=OBJECT;
              FreqRec  :  FreqRecType;
              DFC         : DFCObj;
               I                  :  INTEGER;
              Channel             :  ChannelObj;
              ChannelFreqStart    :  REAL;
              ChannelQueue        :  ChannelQueueObj;
              ASK METHOD ObjInit;
              ASK METHOD ReceivePulse(IN pulse:PulseObj);

              TELL METHOD MeasureFreq(IN pulse : PulseObj;
                        IN ChannelRec : ChannelRecType);
              ASK METHOD DFCinit(IN DFCNEW:DFCObj);
              ASK METHOD
                        DeliverRecord(INFreqRec:FreqRecType);
       END OBJECT;
END MODULE.
```

```
IMPLEMENTATION MODULE IFMMUX;
{-------------------------------------------------------------}

DESCRIPTION:
This is an implementation module of the Test simulation that
implements the Frequency Measurment Receiver (IFMMUX) Object.

{-------------------------------------------------------------}
FROM MathMod IMPORT SIN,COS,ATAN,pi;
FROM pulsegenerator IMPORT PulseObj;
FROM Presorter IMPORT DFCObj;
FROM DerivedLOFreq IMPORT DerivedLOFreq;
FROM RFDownConverted IMPORT RFDownConverted;
FROM BuildFreqRec IMPORT BuildFreqRec;
FROM Debug IMPORT TraceStream;
FROM Globals IMPORT RECEIVEPULSE;


OBJECT ChannelObj;
{-------------------------------------------------------------}
ASK METHOD SetField(IN ChannelFreqStart:REAL);
{-------------------------------------------------------------}
BEGIN
       FreqStart:=ChannelFreqStart;
       LOFreq:=DerivedLOFreq(FreqStart);
END METHOD;
END OBJECT;{ChannelObj}



OBJECT IFMMUXObj;

{-------------------------------------------------------------}
ASK METHOD ObjInit;
{-------------------------------------------------------------}
BEGIN
NEW(ChannelQueue);
FOR I:=2000 TO 16000 BY 2000
     NEW(Channel);
     ChannelFreqStart:=FLOAT(I);
     ASK Channel SetField(ChannelFreqStart);
     ASK ChannelQueue TO Add(Channel);
END FOR;

END METHOD;
```

```
{-------------------------------------------------------------}
ASK METHOD ReceivePulse(IN pulse:PulseObj);
{-------------------------------------------------------------}
   VAR
        F             :    REAL;
        FStart        :    REAL;
        LOF           :    REAL;
        ChannelRec    :    ChannelRecType;

   BEGIN
       IF (RECEIVEPULSE)
         F:=ASK pulse Freq;
         Channel:=ASK ChannelQueue First();
         FStart:=ASK Channel FreqStart;
         LOF   :=ASK Channel LOFreq;
         WHILE NOT(Channel=NILOBJ)
              IF (F<(FStart+2000.0))
                  EXIT;
              END IF;
              Channel:=ASK ChannelQueue Next(Channel);
              FStart:=ASK Channel FreqStart;
              LOF :=ASK Channel LOFreq;
          END WHILE;
          RFDownConverted(F,LOF,ChannelRec);

          TELL (SELF) TO MeasureFreq(pulse,ChannelRec);
       END IF;
     END METHOD;

{-------------------------------------------------------------}
TELL METHOD MeasureFreq(IN pulse : PulseObj;
                        IN ChannelRec :ChannelRecType);
{-------------------------------------------------------------}
   CONST
              TOU = 0.000045;

    VAR
              IFreq,theta,A,E,F,f,LOFREQ  :    REAL;
              FREQ,TIME,timeinterval      :    REAL;

    BEGIN
              IFreq:=ChannelRec.Freq;
              theta:=2.0*pi*IFreq*TOU;
              A:=ASK pulse Power;
              E:=A*SIN(theta); {This is measured value in
                                IFMCFR assembly}
              F:=A*COS(theta); {This is measured value in
                                IFMCFR assembly}
              f:=ABS((ATAN(E/F))/(2.0*pi*TOU));
              LOFREQ:=ChannelRec.LO;
              FREQ:=f-LOFREQ;
```

```
                TIME:=ChannelRec.Time;
                BuildFreqRec(FREQ,TIME,FreqRec);
                IF FreqRec<> NILREC
                   ASK(SELF) DeliverRecord(FreqRec);
                END IF;

   END METHOD;

{-------------------------------------------------------------}
ASK METHOD DFCinit(IN DFCNEW : DFCObj);
{-------------------------------------------------------------}
   BEGIN
                DFC:=DFCNEW;
   END METHOD;
{-------------------------------------------------------------}
ASK METHOD DeliverRecord(IN FreqRec:FreqRecType);
{-------------------------------------------------------------}
   BEGIN
                ASK DFC TO ReceiveFRecord(FreqRec);
   END METHOD;

   END OBJECT;

   END MODULE.
```

```
DEFINITION MODULE DerivedLOFreq;

PROCEDURE DerivedLOFreq(IN FreqStart:REAL):REAL;
END MODULE.

{-------------------------------------------------------------}
IMPLEMENTATION MODULE DerivedLOFreq;

PROCEDURE DerivedLOFreq(IN FreqStart:REAL):REAL;

BEGIN
       IF (FreqStart<4000.0)
             RETURN (2000.0);

       ELSIF (FreqStart<6000.0)
             RETURN (0.0);

       ELSIF (FreqStart<8000.0)
             RETURN (-2000.0);

       ELSIF (FreqStart<10000.0)
             RETURN (-4000.0);

       ELSIF (FreqStart<12000.0)
             RETURN (-6000.0);

       ELSIF (FreqStart<14000.0)
             RETURN (-8000.0);

       ELSIF (FreqStart<16000.0)
             RETURN (-10000.0);

       ELSE
             RETURN (-12000.0);
       END IF;

END PROCEDURE;
END MODULE.
```

```
DEFINITION MODULE RFDownConverted;

FROM IFMMUX IMPORT ChannelRecType;


PROCEDURE RFDownConverted(IN F   :  REAL;
                          IN LOF : REAL;
                          OUT ChannelRec   : ChannelRecType);
END MODULE.

{-------------------------------------------------------------}
IMPLEMENTATION MODULE RFDownConverted;

FROM IFMMUX IMPORT ChannelRecType;
FROM SimMod IMPORT SimTime;
FROM Debug IMPORT TraceStream;

PROCEDURE RFDownConverted(IN F   :  REAL;
                          IN LOF : REAL;
                          OUT ChannelRec   : ChannelRecType);
VAR
        IFreq : REAL;

BEGIN
        IFreq:=F+LOF;
        NEW(ChannelRec);
        ChannelRec.Freq:=IFreq;
        ChannelRec.LO:=LOF;
        ChannelRec.Time:=SimTime();


END PROCEDURE;
END MODULE.
```

```
DEFINITION MODULE BuildFreqRec;

FROM IFMMUX IMPORT FreqRecType;

PROCEDURE BuildFreqRec(IN FREQ : REAL;
                       IN TIME : REAL;
                       OUT FreqRec : FreqRecType);
END MODULE.

{-------------------------------------------------------------}
IMPLEMENTATION MODULE BuildFreqRec;

FROM IFMMUX IMPORT FreqRecType;

PROCEDURE BuildFreqRec(IN FREQ : REAL;
                       IN TIME : REAL;
                       OUT FreqRec : FreqRecType);

BEGIN
       NEW(FreqRec);
           FreqRec.Freq:=FREQ;
           FreqRec.Time:=TIME;

END PROCEDURE;
END MODULE.
```

```
DEFINITION MODULE Presorter;
{-------------------------------------------------------------}

DESCRIPTION:
This is a definition module of the Test simulation that
defines Display-Processor Group (Presorter) Object.

{-------------------------------------------------------------}
FROM ListMod IMPORT QueueList;
FROM DFR IMPORT AngleRecType,DFRObj;
FROM IFMMUX IMPORT FreqRecType,IFMMUXObj;
FROM ECM IMPORT ECMObj;

EXPORTTYPE
      DFCObj = OBJECT; FORWARD;
TYPE

      PulseRecType=RECORD
              Time    :    REAL;
              Angle   :    REAL;
              Freq    :    REAL;
              Power   :    REAL;
              Power1  :    REAL;
      END RECORD;

      BufferRecList=OBJECT(QueueList[ANYREC : PulseRecType]);
      END OBJECT;

      PulseRecBufferObj=OBJECT(QueueList [ANYREC :
                                              PulseRecType]);
      END OBJECT;

      DPUObj=OBJECT;
             ECM           : ECMObj;
             DPUPDWBuffer  : BufferRecList;
             BufferNext    : BufferRecList;
             Processor     : BufferRecList;
             ProcessInterval:REAL;
             PROCESSED:BOOLEAN;
             ASK METHOD ObjInit;
             TELL METHOD Process;
             ASK METHOD GetBuffer(IN  Buffer:BufferRecList ;
                              IN DTUPDWBuffer:BufferRecList);
             ASK METHOD FindParameter;
             ASK METHOD ECMinit(IN ECMNEW: ECMObj);
             ASK METHOD InitiateECM(IN ANG:REAL;
                                    IN TECH:STRING);
      END OBJECT;
```

```
DTUObj=OBJECT;
        EFM  :BufferRecList;
        PollingInterval:REAL;
        DTUPDWBuffer : BufferRecList;
        Buffer       : BufferRecList;
        DPU:DPUObj;
        ASK METHOD ObjInit;
        TELL METHOD Poll;
        TELL METHOD GetBuffer(IN
                              PDWBuffer:BufferRecList);
        ASK  METHOD DeclareContact;
        ASK  METHOD BufferSort;
END OBJECT;

DFCObj=OBJECT;
        MATCH:BOOLEAN;
        PDWBuffer : BufferRecList;
        PulseRec : PulseRecType;
        PulseRecBuffer : PulseRecBufferObj;
        DTU:DTUObj;
        ASK METHOD ObjInit;
        ASK METHOD ReceiveFRecord(IN
                                  FreqRec:FreqRecType);
        ASK METHOD ReceiveARecord(IN AngleRec :
                                      AngleRecType);

        ASK METHOD Correlate;

 END OBJECT;



END MODULE.
```

```
IMPLEMENTATION MODULE Presorter;
{-------------------------------------------------------------}

DESCRIPTION:
This is an implementation module of the Test simulation that
implements the Display-Processor Group (Presorter) Object.

{-------------------------------------------------------------}
FROM pulsegenerator IMPORT PulseObj;
FROM DFR IMPORT AngleRecType,DFRObj;
FROM IFMMUX IMPORT FreqRecType,IFMMUXObj;
FROM ECM IMPORT ECMObj;
FROM BuildPulseFreqRec IMPORT BuildPulseFreqRec;
FROM BuildPulseAngRec IMPORT BuildPulseAngRec;
FROM BuildPDWRec IMPORT BuildPDWRec;
FROM MathMod IMPORT e;
FROM SimMod IMPORT SimTime;
FROM Debug IMPORT TraceStream;
FROM ObjMod IMPORT ObjectDump;
FROM RGlobals IMPORT SHierRecType,EmitterSHArray;
FROM Input IMPORT ReadEmAll;
FROM FindSH IMPORT FindSHRec;

OBJECT DFCObj;

{-------------------------------------------------------------}
ASK METHOD ObjInit;
{-------------------------------------------------------------}
    BEGIN
    NEW(DTU);
    NEW(PulseRecBuffer);
    NEW(PDWBuffer);
    END METHOD;

{-------------------------------------------------------------}
ASK METHOD ReceiveFRecord(IN FreqRec: FreqRecType);
{-------------------------------------------------------------}
    VAR
           TIME,ANG,FREQ:REAL;

    BEGIN
           BuildPulseFreqRec(FreqRec,PulseRec);
           TIME:=PulseRec.Time;
           ANG:=PulseRec.Angle;
           FREQ:=PulseRec.Freq;
           ASK PulseRecBuffer TO Add(PulseRec);
           ASK(SELF) TO Correlate;
    END METHOD;
```

```
{-------------------------------------------------------------}
ASK METHOD ReceiveARecord(IN AngleRec: AngleRecType);
{-------------------------------------------------------------}
    VAR
          TIME,ANG,FREQ:REAL;


    BEGIN
          BuildPulseAngRec(AngleRec,PulseRec);
          TIME:=PulseRec.Time;
          ANG:=PulseRec.Angle;
          FREQ:=PulseRec.Freq;
          ASK PulseRecBuffer TO Add(PulseRec);
    END METHOD;

{-------------------------------------------------------------}
ASK METHOD Correlate;
{-------------------------------------------------------------}
    CONST
           tolerance=0.05;
    VAR
           rec, rec2,Oldrec,Oldrec2,PDWRec  :  PulseRecType;
           Time,Angle,Freq,Freq1,Freq2,TIME :  REAL;
           ANG,FREQ,TIME1,ANG1,FREQ1        :  REAL;
           Power,Power1                     :  REAL;
           numberRec                        :  INTEGER;

    BEGIN
           MATCH:=FALSE;
           rec:=ASK PulseRecBuffer First();
           WHILE (rec <> NILREC)

             rec2:=rec;
             WHILE (NOT MATCH) AND (rec2 <> NILREC)

                rec2:=ASK PulseRecBuffer Next(rec2);
                 IF (rec2=NILREC)
                       EXIT;
                 END IF;
                 IF (ABS(rec.Time-rec2.Time)< tolerance)
                       Time:=rec.Time;
                       Angle:=rec.Angle+rec2.Angle;
                       Freq:=rec.Freq+rec2.Freq;
                       Power:=rec.Power+rec2.Power;
                       Power1:=rec.Power1+rec2.Power1;

           BuildPDWRec(Time,Angle,Freq,Power,Power1,PDWRec);
                        MATCH:=TRUE;
                   IF PDWRec<>NILREC
                       ASK PDWBuffer TO Add(PDWRec);
                   END IF;
```

```
                END IF;
             END WHILE;
             IF (NOT MATCH)
                rec:=ASK PulseRecBuffer Next(rec);

             ELSE
                Oldrec:=rec;
                Oldrec2:=rec2;
               Freq:=Oldrec.Freq;
               Freq2:=Oldrec2.Freq;
               rec:=ASK PulseRecBuffer Next(Oldrec);
               Freq1:=rec.Freq;
               IF (rec <> Oldrec2)
                   ASK PulseRecBuffer TO RemoveThis(rec);
                   ASK PulseRecBuffer TO RemoveThis(Oldrec);

                ELSE
                   rec:=ASK PulseRecBuffer Next(Oldrec2);
                   ASK PulseRecBuffer TO RemoveThis(Oldrec);
                   ASK PulseRecBuffer TO RemoveThis(rec2);

                END IF;

               END IF;
             MATCH:=FALSE;
          END WHILE;

          TELL DTU TO GetBuffer(PDWBuffer);
    END METHOD;
    END OBJECT;

OBJECT DTUObj;

{-------------------------------------------------------------}
ASK METHOD ObjInit;
{-------------------------------------------------------------}
   BEGIN
          NEW(DPU);
          NEW(DTUPDWBuffer);
          NEW(EFM);
          NEW(Buffer);
          PollingInterval:=50.0;
          TELL SELF TO Poll;

   END METHOD;
```

```
{-------------------------------------------------------------}
TELL METHOD Poll;
{-------------------------------------------------------------}
  BEGIN
        WAIT DURATION PollingInterval
        END WAIT;
        ASK SELF TO BufferSort;

  END METHOD;
{-------------------------------------------------------------}
TELL METHOD GetBuffer(IN PDWBuffer:BufferRecList);
{-------------------------------------------------------------}

  BEGIN
        DTUPDWBuffer:=PDWBuffer;

  END METHOD;

{-------------------------------------------------------------}
ASK METHOD DeclareContact;
{-------------------------------------------------------------}
   VAR
         rec:PulseRecType;

   BEGIN

              rec:=ASK EFM First();
              IF (rec<>NILREC)
                 ASK Buffer Add(rec);
                 ASK EFM RemoveThis(rec);
              END IF;
                rec:=ASK EFM First();
              WHILE( rec<>NILREC)
                   ASK EFM RemoveThis(rec);
                   rec:=ASK EFM First();
               END WHILE;

               ASK DPU TO GetBuffer(Buffer,DTUPDWBuffer);
   END METHOD;

{-------------------------------------------------------------}
ASK METHOD BufferSort;
{-------------------------------------------------------------}
    CONST
            AngleTol=0.5;
            FreqTol=0.5;
    VAR
           TIME,ANG,FREQ,Time,Angle,Freq       :     REAl;
           TimeFirst,TimeLast,TIMEIF,TIMEELSE  :     REAL;
           rec,EFMRec,DisRec                 : PulseRecType;
           RecFirst,RecLast,recLast          :  PulseRecType;
```

```
        numberRec   :INTEGER;

BEGIN
     rec:=ASK DTUPDWBuffer Remove();
       TIME:=rec.Time;
       ANG:=rec.Angle;
       FREQ:=rec.Freq;

        WHILE(rec<>NILREC)

        recLast:=ASK DTUPDWBuffer Last();

            WHILE(rec<>NILREC)

             IF (ABS(ANG-rec.Angle)<AngleTol) AND
                            (ABS(FREQ-rec.Freq)<FreqTol)

                    ASK EFM TO Add(rec);

                IF (rec = recLast)
                    EXIT;
                END IF;

                rec:=ASK DTUPDWBuffer Remove();
                TIMEIF:=rec.Time;

             ELSE
                  ASK DTUPDWBuffer TO Add(rec);
                  IF (rec = recLast)
                         EXIT;
                  END IF;
                  rec:=ASK DTUPDWBuffer Remove();
                  TIMEELSE:=rec.Time;

              END IF;

            END WHILE;
            numberRec:=ASK EFM numberIn;
            RecFirst:=ASK EFM First();
            TimeFirst:=RecFirst.Time;
            RecLast:=ASK EFM Last();
            TimeLast:=RecLast.Time;

            IF (numberRec<3)
               DisRec:=ASK EFM First();
               WHILE (DisRec<>NILREC)
                      ASK EFM RemoveThis(DisRec);
                      DISPOSE(DisRec);
                      DisRec:=ASK EFM First();
               END WHILE;
```

```
                ELSE
                   ASK SELF DeclareContact;
                END IF;

                rec:=ASK DTUPDWBuffer First();
                IF(rec<>NILREC)

                   TIME:=rec.Time;
                   ANG:=rec.Angle;
                   FREQ:=rec.Freq;
                   ASK DTUPDWBuffer RemoveThis(rec);
                END IF;
             END WHILE;

     END METHOD;
     END OBJECT;

OBJECT DPUObj;
{-------------------------------------------------------------}
ASK METHOD ObjInit;
{-------------------------------------------------------------}
   BEGIN
      NEW(DPUPDWBuffer);
      NEW(Processor);
      NEW(BufferNext);
      ProcessInterval:=50.0;
      PROCESSED:=FALSE;
   END METHOD;

{-------------------------------------------------------------}
ASK METHOD GetBuffer(IN  Buffer      :BufferRecList;
                     IN  DTUPDWBuffer:BufferRecList);
{-------------------------------------------------------------}
VAR
        rec1 : PulseRecType;
BEGIN
      DPUPDWBuffer:=DTUPDWBuffer;
      BufferNext:=Buffer;
      IF NOT(PROCESSED)
          TELL SELF TO Process;
      END IF;
END METHOD;

{-------------------------------------------------------------}
TELL METHOD Process;
{-------------------------------------------------------------}
  BEGIN
         WAIT DURATION ProcessInterval
         END WAIT;
         ASK SELF FindParameter;
  END METHOD;
```

```
{-------------------------------------------------------------}
ASK METHOD FindParameter;
{-------------------------------------------------------------}
 CONST
            AngleTol=0.5;
            FreqTol=0.5;

VAR
       PRI,TimeFirst,TimeLast,ANG,FREQ,TIME :       REAL;
       RecFirst,RecLast,rec,recLast    :     PulseRecType;
       RecProcessed,rec1               :     PulseRecType;
       N                               :          INTEGER;
       EmitterSHRec                    :     SHierRecType;
       Name,TECH,IFF,Hostile           :           STRING;

BEGIN

      NEW(ECM);
      rec1:=ASK BufferNext First();
      IF (rec1<>NILREC)
         rec:=ASK BufferNext Remove();

           TIME:=rec.Time;
           ANG:=rec.Angle;
           FREQ:=rec.Freq;

         WHILE(rec<>NILREC)
             rec:=ASK DPUPDWBuffer Remove();
             recLast:=ASK DPUPDWBuffer Last();

               WHILE(rec<>NILREC)
                  IF (ABS(ANG-rec.Angle)<AngleTol) AND
                                 (ABS(FREQ-rec.Freq)<FreqTol)
                      ASK Processor TO Add(rec);
                      IF (rec = recLast)
                             EXIT;
                      END IF;
                      rec:=ASK DPUPDWBuffer Remove();
                  ELSE
                      ASK DPUPDWBuffer TO Add(rec);
                      I^U^^(*^U*+* ecLast)
                              EXIT;
                      END IF;
                       rec:=ASK DPUPDWBuffer Remove();

                   END IF;

                END WHILE;

                N:=ASK Processor numberIn;
                RecFirst:=ASK Processor First();
```

```
                TimeFirst:=RecFirst.Time;
                RecLast:=ASK Processor Last();
                TimeLast:=RecLast.Time;
                PRI:=(TimeLast-TimeFirst) / FLOAT(N-1);

                RecProcessed:=ASK Processor First();
                WHILE (RecProcessed<> NILREC)
                    ASK Processor RemoveThis(RecProcessed);
                    RecProcessed:=ASK Processor First();
                END WHILE;
                ReadEmAll;
                FindSHRec(EmitterSHArray,PRI,FREQ,
                                              EmitterSHRec);
                IF(EmitterSHRec<>NILREC)
                  Name:=EmitterSHRec.TopString;
                  IFF:=EmitterSHRec.OwnedString[7];
                  TECH:=EmitterSHRec.OwnedString[8];

                  ASK SELF ECMinit(ECM);

                  IF ((IFF = "Hostile") AND (8000.0 <= FREQ)
                                  AND (FREQ <=12000.0))
                     ASK SELF TO InitiateECM(ANG,TECH);
                  END IF;
                 END IF;
                 rec:=ASK BufferNext First();
                 IF(rec<>NILREC)

                    TIME:=rec.Time;
                    ANG:=rec.Angle;
                    FREQ:=rec.Freq;
                    ASK BufferNext RemoveThis(rec);
                  END IF;

                  IF (rec = NILREC)
                      PROCESSED:=TRUE;
                  END IF;
          END WHILE;
       END IF;
END METHOD;

{-------------------------------------------------------------}
ASK METHOD ECMinit(IN ECMNEW:ECMObj);
{-------------------------------------------------------------}
 BEGIN
        ECM:=ECMNEW;
 END METHOD;
```

```
{-------------------------------------------------------------}
ASK METHOD InitiateECM(IN ANG:REAL;
                       IN TECH:STRING);
{-------------------------------------------------------------}
 BEGIN
       ASK ECM TO Jam(TECH,ANG);
 END METHOD;

END OBJECT;

END MODULE.
```

```
DEFINITION MODULE BuildPDWRec;

FROM Presorter IMPORT PulseRecType;

PROCEDURE BuildPDWRec(IN Time : REAL;
                      IN Angle: REAL;
                      IN Freq : REAL;
                      IN Power: REAL;
                      IN Power1 : REAL;
                      OUT PDWRec : PulseRecType);

END MODULE.

{-------------------------------------------------------------}
IMPLEMENTATION MODULE BuildPDWRec;

FROM Presorter IMPORT PulseRecType;

PROCEDURE BuildPDWRec(IN Time : REAL;
                      IN Angle: REAL;
                      IN Freq : REAL;
                      IN Power: REAL;
                      IN Power1 : REAL;
                      OUT PDWRec : PulseRecType);
BEGIN
    NEW(PDWRec);
        PDWRec.Time:=Time;
        PDWRec.Angle:=Angle;
        PDWRec.Freq:=Freq;
        PDWRec.Power:=Power;
        PDWRec.Power1:=Power1;
END PROCEDURE;
END MODULE.
```

```
DEFINITION MODULE FindSH;

FROM RGlobals IMPORT SHierRecType,SHArrayType;

PROCEDURE FindSHRec (IN SHArray : SHArrayType;
                     IN PRI : REAL;
                     IN FREQ: REAL;
                     OUT SHRec  : SHierRecType);

END MODULE.

{-------------------------------------------------------------}
IMPLEMENTATION MODULE FindSH;
FROM RGlobals IMPORT SHierRecType,SHArrayType;
FROM Debug IMPORT TraceStream;

PROCEDURE FindSHRec (IN SHArray : SHArrayType;
                     IN PRI :REAL;
                     IN FREQ :REAL;
                     OUT SHRec  : SHierRecType);

VAR
     ThisRec                        : SHierRecType;
     i                                  : INTEGER;
     PRILO,PRIHIGH,FREQLO,FREQHIGH     : STRING;
BEGIN

     NEW(SHRec);
     i :=0;
     REPEAT
           INC(i);
           ThisRec:=SHArray[i];
           FREQLO:=ThisRec.OwnedString[1];
           FREQHIGH:=ThisRec.OwnedString[2];
           PRILO:=ThisRec.OwnedString[3];
           PRIHIGH:=ThisRec.OwnedString[4];

     UNTIL ((i >= HIGH(SHArray)) OR ((STRTOREAL(PRILO)<=PRI)
AND (STRTOREAL(PRIHIGH)>=PRI)  AND (STRTOREAL(FREQLO)<=FREQ)
AND (STRTOREAL(FREQHIGH)>=FREQ)));

     IF ((STRTOREAL(PRILO)<=PRI) AND (STRTOREAL(PRIHIGH)>=PRI)
AND (STRTOREAL(FREQLO)<=FREQ) AND (STRTOREAL(FREQHIGH)>=FREQ))
           SHRec := ThisRec;
     ELSE
           OUTPUT("SHRec is a NILREC!");
           SHRec := NILREC;
     END IF;

END PROCEDURE;
END MODULE.
```

```
DEFINITION MODULE Input;

PROCEDURE ReadEmAll;
END MODULE.

{-------------------------------------------------------------}
IMPLEMENTATION MODULE Input;
FROM RGlobals IMPORT FileNameType,SHierRecType;
FROM IOMod IMPORT StreamObj, FileUseType(Input);
FROM RGlobals IMPORT MasterFileName,EmitterSHArray;
FROM ReadLst IMPORT ReadLst;
FROM Debug IMPORT TraceStream;

VAR
    EmitterFileName : FileNameType;

{-------------------------------------------------------------}
PROCEDURE ReadEmitterLib;
{-------------------------------------------------------------}
BEGIN

    ASK TraceStream TO WriteString("EmitterLib");
    ASK TraceStream TO WriteLn;
    ReadLst(EmitterSHArray, EmitterFileName);


END PROCEDURE;

{-------------------------------------------------------------}
PROCEDURE ReadEmAll;
{-------------------------------------------------------------}
VAR
    File : StreamObj;
    str  : STRING;
BEGIN
    NEW(File);
    ASK File TO Open(MasterFileName, Input);
    ASK File TO ReadString(EmitterFileName);
    ASK File TO ReadLine(str);
    ReadEmitterLib;

END PROCEDURE;
END MODULE.
```

```
DEFINITION MODULE ReadSH;

FROM RGlobals IMPORT SHierRecType;
FROM IOMod IMPORT StreamObj;

PROCEDURE ReadSH(IN File : StreamObj;
                 OUT SHierRec : SHierRecType;
                 OUT error  :  BOOLEAN);

END MODULE.

{-------------------------------------------------------------}
IMPLEMENTATION MODULE ReadSH;

FROM IOMod IMPORT StreamObj, FileUseType(Input);
FROM RGlobals IMPORT SHierRecType;
FROM Debug IMPORT TraceStream;
FROM IOMod IMPORT ReadKey;

PROCEDURE ReadSH(IN File : StreamObj;
                 OUT SHierRec : SHierRecType;
                 OUT error  :  BOOLEAN);

TYPE
    StringRecType = RECORD
       String : STRING;
       Next  : StringRecType;
    END RECORD;

VAR
    string  : STRING;
    numberOfStrings : INTEGER;
    StringRec, OldStringRec : StringRecType;
    first  :  StringRecType;
    arrow  :  STRING;
    stringRec : StringRecType;
    i : INTEGER;


BEGIN
    NEW(SHierRec);
    ASK File TO ReadString(SHierRec.TopString);
    ASK  TraceStream  TO  WriteString("Top  string  is"  +
SHierRec.TopString + " ");
    ASK TraceStream TO WriteLn;

    NEW(StringRec);
    numberOfStrings :=1;
    first := StringRec;

    ASK File TO ReadString(arrow);
```

```
    IF (arrow <> "->")
        OUTPUT("File not formatted correctly");
        error :=TRUE;
        RETURN;
    ELSE
        error :=FALSE;
    END IF;

    WHILE (string <> "\\")
         ASK File TO ReadString(string);
         IF (string = "..")
             ASK File TO ReadLine(string);
         ELSE
             OldStringRec := StringRec;
             StringRec.String := string;
             NEW(StringRec);
             OldStringRec.Next :=StringRec;
             numberOfStrings:=numberOfStrings + 1;
         END IF;
    END WHILE;

    ASK File TO ReadLine(string);
    IF ((numberOfStrings > 0) AND NOT error)
       NEW(SHierRec.OwnedString, 1..numberOfStrings -2);
       stringRec :=first;
       FOR i:=1 TO numberOfStrings -2
           SHierRec.OwnedString[i] :=stringRec.String;
           stringRec:= stringRec.Next;
       END FOR;
    END IF;
    ASK TraceStream TO WriteLn;

END PROCEDURE;
END MODULE.
```

```
DEFINITION MODULE ReadLst;

FROM RGlobals IMPORT SHArrayType,FileNameType;

PROCEDURE ReadLst( INOUT SHArray : SHArrayType;
                   IN  FileName  : FileNameType);

END MODULE.

{-------------------------------------------------------------}
IMPLEMENTATION MODULE ReadLst;

FROM IOMod IMPORT StreamObj, FileUseType(Input);
FROM RGlobals IMPORT SHArrayType,FileNameType;
FROM ReadSH IMPORT ReadSH;
FROM Debug IMPORT TraceStream;

PROCEDURE ReadLst( INOUT SHArray : SHArrayType;
                   IN  FileName  : FileNameType);

VAR
    File : StreamObj;
    numberOfSH : INTEGER;
    i : INTEGER;
    error : BOOLEAN;
    string : STRING;

BEGIN
    NEW(File);
    ASK File TO Open(FileName, Input);

    ASK File TO ReadInt(numberOfSH);
    ASK File TO ReadLine(string);
    ASK TraceStream TO WriteLn;
    ASK TraceStream TO WriteLn;

    NEW(SHArray, 1..numberOfSH);
    FOR i :=1 TO numberOfSH
        ASK TraceStream TO WriteString("rec" + INTTOSTR(i));

        ASK TraceStream TO WriteLn;
        ReadSH(File, SHArray[i], error);
        IF error
          OUTPUT("problem reading file", FileName,
                 "BAD FORMAT DETECTED.");
        END IF;
    END FOR;

END PROCEDURE;

END MODULE.
```

```
DEFINITION MODULE RGlobals;

CONST
      MasterFileName = "Master.dat";

TYPE
      FileNameType = STRING;
      SArrayType   = ARRAY INTEGER OF STRING;

      SHierRecType = RECORD
         TopString : STRING;
         OwnedString : SArrayType;
      END RECORD;

      SHArrayType = ARRAY INTEGER OF SHierRecType;

VAR
      EmitterSHArray : SHArrayType;

END MODULE.
```

EMITTER LIBRARY:

```
3
sps18 -> 9500.0 9550.0 9.0 11.0 USSR Surface Hostile BBN \\
sps58 -> 9450.0 9550.0 8.5  9.5 PROC Airbore Hostile SIN&BBN
\\
src11  ->  8850.0  8950.0  12.5  13.5  PROC  UnderWater  Hostile
TRI&BBN \\
```

```
DEFINITION MODULE ECM;


TYPE
     ECMObj=OBJECT;
           ASK METHOD Jam(IN TECH : STRING;
                          IN ANG  : REAL);

     END OBJECT;

END MODULE.

{-------------------------------------------------------------}
IMPLEMENTATION MODULE ECM;


OBJECT ECMObj;
{-------------------------------------------------------------}
ASK METHOD Jam(IN TECH : STRING;
               IN ANG  : REAL);
{-------------------------------------------------------------}
VAR
        Technique: STRING;
        Angle    : REAL;
BEGIN
        Technique:=TECH;
        OUTPUT("Technique IN ECM DURING JAMMING ",Technique);
        Angle    :=ANG;
        OUTPUT("Angle",Angle);
END METHOD;
END OBJECT;
END MODULE.
```

```
DEFINITION MODULE DebugRun;

PROCEDURE SetUpD(IN TraceOn : BOOLEAN);

END MODULE.

{-----------------------------------------------------------}
IMPLEMENTATION MODULE DebugRun;

FROM IOMod IMPORT FileUseType(Output);
FROM Debug IMPORT TraceStream;
FROM UtilMod IMPORT DateTime;
{-----------------------------------------------------------}
PROCEDURE SetUpD(IN TraceOn : BOOLEAN);
{-----------------------------------------------------------}
   VAR
       DT  :  STRING;

   BEGIN
       NEW(TraceStream);
       ASK TraceStream TO Open("debug.out", Output);

       DateTime(DT);
       ASK TraceStream TO WriteString(DT);
       ASK TraceStream TO WriteLn;
       ASK TraceStream TO WriteLn;
       ASK TraceStream TO WriteLn;

       IF (TraceOn)
               ASK TraceStream TO TraceOn;
               OUTPUT(" -----TRACE ON------------");
               ASK TraceStream TO WriteString("Initially,
                                        trace is on.");
               ASK TraceStream TO WriteLn;
       ELSE
               ASK TraceStream TO TraceOff;
               ASK TraceStream TO WriteString("Initially,
                                       trace is off.");
               ASK TraceStream TO WriteLn;
       END IF;
   END PROCEDURE;

   END MODULE.
```

LIST OF REFERENCES

1 **Raytheon Company**'s Document Of Introduction Of Electronic Defense System For THE REPUBLIC OF CHINA NAVY FFG Program, Goleta, CA 93117.

2. **Raytheon Company**'s Document of SLQ-32/SIDEKICK Operational Description, Goleta, CA 93117.

3. **CACI Products Company**, MODSIM II, The language for Object-Oriented Programming, Reference Manual, 1990 ed., 3344 North Torrey Pines Court, La Jolla, CA 92037.

4. Schleher,D.C., Introduction to Electronic Warfare, Artech House Inc, 1986.

INITIAL DISTRIBUTION LIST

		copies
a)	Defense Technical Information Center Cameron Station Alexandria, VA 22304-6145	2
b)	Naval Academy Library Kau-Hsiung, Tso-Ying P.O. Box 90175 Taiwan, R.O.C.	2
c)	Library, Code 52 Naval Postgraduate School Monterey, CA 93943-5002	2
d)	Chen-Kuo Li Nei-Hu District, Cheng-kung Rd., Section 4, Ave 61, Alley 19, #3, 4F. Taipei, Taiwan, R.O.C.	2
e)	Professor Michael P.Bailey, Code OR/BA Naval Postgraduate School Monterey, CA 93943-5000	2
f)	Professor Alan R. Washburn, Code OR/WS Naval Postgraduate School Monterey, CA 93943-5000	1
g)	Chairman, Code EW Electronic Warfare Academic Group Naval Postgraduate School Monterey, CA 93943	1
h)	Library Chung Cheng Institute Of Technology Tashi, Taoyuan County Taiwan, R.O.C.	1
i)	Wu, Tsung-Li 6-3 (4F) Lane 42, San-min Road, Section 1 Taoyuan City, Taiwan, R.O.C.	1
j)	LCol. Miguel A. Betancourt R. Calle Piar #37, Urb. Mario Briceno Iragorry Maracay, Edo. Aragua, Venezuela	1