National Centre for Radio Astrophysics

Internal Technical Report

GMRT/SERVO/PC104/001-May 2013

PC104 Servo Software Manual

THIYAGARAJAN BEEMAN

thiyagu@gmrt.ncra.tifr.res.in

Giant Metrewave Radio Telescope

Tata Institute of Fundamental Research

Khodad - 410504

Document Status History

S.No	Document Status	Description
1	Document Title	PC104 Based Servo Software Manual
2	Document Reference No.	GMRT/SERVO/PC104/001-May 2013
3	Revision	0.1
4	Date	29-July-2013
5	Prepared By	Thiyagarajan Beeman
6	Reviewed By	Servo Group Members
7	Approved By	SMEC Committee

Document Change Record

DCR No	First Issue	
Date	29-July-2013	
Document Title	PC104 Servo Software Manual	
Document Reference No	GMRT/SERVO/PC104/001-May2013	
Document Revision No	0.1	
Page No	**Paragraph**	**Reason For Change**

ACRONYM

ISA BUS	Industry Standard Architecture Bus
SS BUS	Station Servo Computer Bus
AIO	Analog Input and Output
TTL	Transistor Transistor Logic
RS232-C	Recommended Standard 232 – C
CPU	Central Processing Unit
GMRTACU	GMRT Antenna Control Unit
SSC	Station Servo Computer
GUI	Graphical User Interface
FIFO	First In First Out
RTAI	Real Time Application Interface
GPOS	General Purpose Operating System
AZ Axis	Azimuth Axis
EL Axis	Elevation Axis
HHT	Hand Held Terminal
CAN	Controller Area Network
SPI	Serial Peripheral Interface
PWM	Pulse Width Modulation
I2C	Inter Integrated Circuit
GPIO	General Purpose Input and Output
LAN	Local Area Network
PCI	Peripheral Component Interconnect
IPC	Inter Process Communication
PLC	Position Loop Compensator

ABSTRACT

The objective and scope of the GMRT servo system is to upgrade servo computer. The development currently in the completion phase and mass production has started. The new servo computer is based on PC technology with real-time characteristics that will replace an existing antiquated 8086 based servo computer as well as PASCAL based control software. Maintaining the existing 8086 servo computer and software is becoming increasingly difficult and expensive as well as not able to strive astronomer’s requirements. As a result, the GMRT observatory has decided to replace an existing system with more up-to-date and state-of-art technology that is PC104 based servo computer and Real Time Application Interface (RTAI) based servo control software. The updated servo computer and software will have all the functionalities of current system. It also employs Qt based Graphical User Interface (GUI) to users of the system. This document gives the introduction about GMRT servo system and extensively covers the internal modules of antenna control software.

CONTENTS

List of Figures

List of Tables

1 Introduction to GMRT Servo System

The Giant Metrewave Radio Telescope (GMRT) is set up as a national facility for frontline research in radio astronomy and astrophysics. It is located at Khodad, about 80km north of Pune. This facility is one of the largest radio telescopes in the world. It is an aperture array comprising of 30 fully steerable parabolic dishes of 45m diameter each. Each of the 30 antennas is steered using an electronic servo system. The servo system is required to meet following requirement,

- It should be able to point the antenna anywhere in the sky,
- It should be able to point to ±10% of 3dB radio bandwidth for a given wavelength.
- To counter the effect of earth's rotation, the antenna must be able to track the object under observation.

To conform to the first specification, two axis mount is used. One axis, the Azimuth axis is normal to the horizon while the other, the Elevation axis is parallel to the horizon. The elevation axis is mounted on the yoke which can rotate about the azimuth axis.

To meet the second requirement, very high accuracy position servo control is used. The GMRT servo control system is located at the antenna base of each antenna and facilitates the accurate tracking of targets by positioning the Azimuth and Elevation axes of the antenna as per commands received from the remote command center.

The present version of the GMRT antenna servo control system is controlled by 8086 based servo computer which act as a primary CPU and carries out overall supervision of all servo subsystem. It performs the,

- Closed loop position control,
- Handling control and monitor commands from remote host and local HHT,
- Generating demand trajectory every 100 msec based on data received from remote computer,
- Position measurement and display after offset-set correction,
- Scanning digital inputs and performing operational and safety interlocking logic and driving relay outputs,
- Limit releasing operation,
- Stow, stow release and parking operation,
- Time of Day, Power on self -test and diagnostics.

All these functions are performed by the primary CPU by interacting with the other system resources like, Encoder Card, Analog I/O Card, Digital Input and Output Cards. The physical medium which combines the other resources into primary CPU is the parallel bus called SSBUS. This bus mainly carries the control, data and address signals across system resources. This bus is capable of 8/16-bit I/O and memory data transfer. The following Figure 1-1 shows the SSBUS architecture.

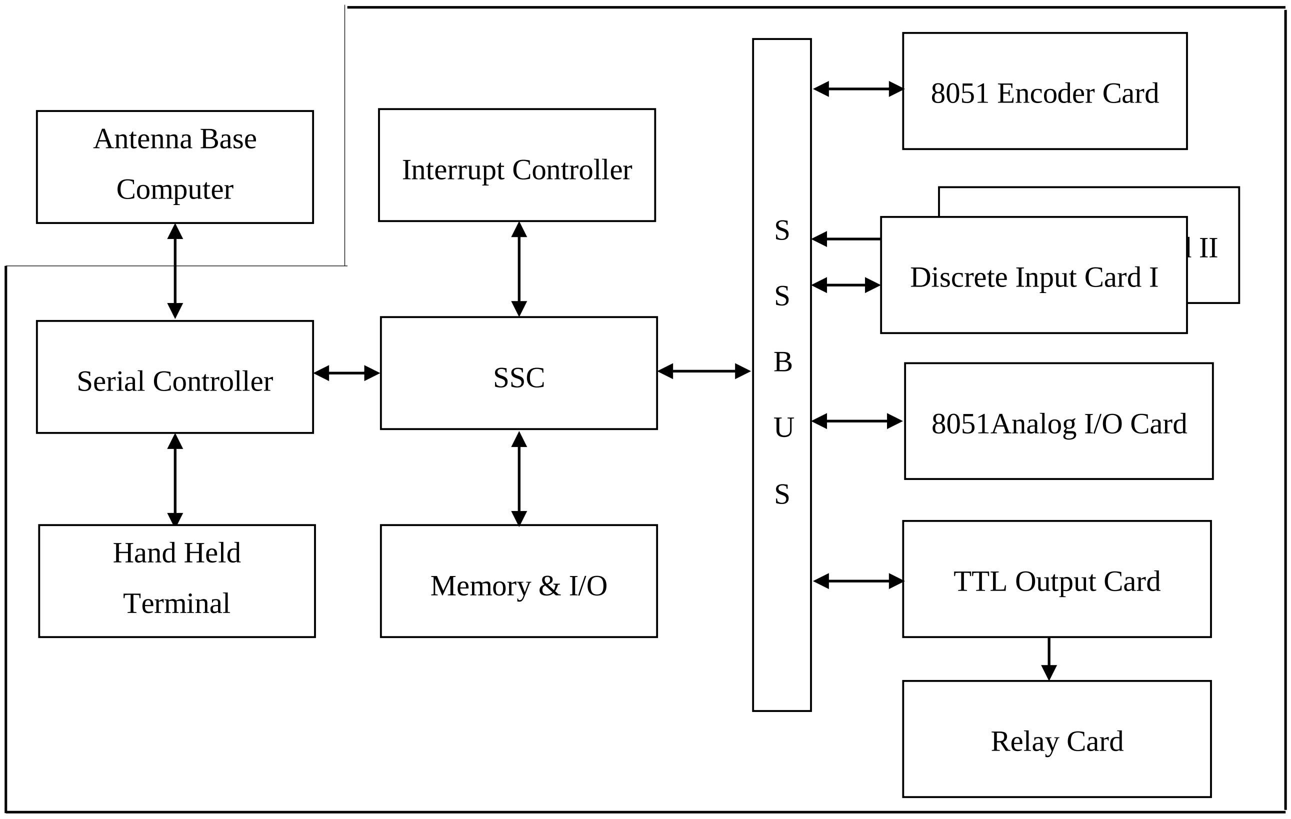

Figure 1-1 SSBUS Architecture

1.1 Limitations of Present System

- Processor speed and inter-processor speed can’t be increased beyond 8MHz. Because present hardware version of the station servo computer build with 8086 processor with limited system resources (like I/O, memory, interrupts etc). Some of the I/O peripherals in servo computer are outdated and readily not available.
- Low speed serial communication interface is available. There is no provision in present servo computer hardware to add additional communication I/O‟s (I2C, SPI etc) to enhance the data transfer.
- User (radio astronomer) requirement can't be easily met with present version of unsupported system hardware and unfamiliar software development tools (like compilers, assembles, debugging tools etc).
- Present version of servo control algorithm runs with frequency of 10Hz. For GMRT like high precision antenna position control systems, better sampling will improve the frequent interaction with filed resources and in case of failures corrective measures are taken soon without any damage to instrument as well as human beings.
- There is no frontend interface tool to analysis servo problems on its occurrence. Large amount of man power required for testing hardware and debugging the system, before it goes to antenna.

1.2 Plan – XI Development Plans

By considering the above limitations, it is decided to upgrade the GMRT antenna controller with state of art electronics with industry standards. The plan – XI development replaces the PASCAL based GMRT antenna control algorithm into highly structured and procedural C language. It eliminates the use of proprietary assemblers, compilers and operating systems (Turbo PASACL, Windows etc) used in GMRT earlier.

The present development fully depends on GNU Linux operating system with open source tools. As well as the real time feature is added to the software with the help of Real Time Application Interface (RTAI). Wide variety of Integrated Development Environments (IDE) and debugging tools are available for software development and testing this helps to reduce the product development cycle.

The plan – XI development not only talks about software development and replacement and also it replaces the old 16-bit 8086 processor based servo computer into 32-bit i386 compatible industry standard PC104 modules. This industry standard PC104 module has wide variety of in build I/O interface components like I2C, SPI, PWM, GPIOs, RS232, RS422, LAN etc, and also it supports on the shelf I/O expansion cards like CAN Controller, TTL based digital I/Os, PCI expansion cards etc. This eliminates the drawbacks of present servo computer limitations mentioned above and also gives the road map for future development. The following table summarizes the servo developmental plans,

S.No	**Development Plan**	**Present Servo System**	**Proposed Servo System**
1.	Plan XI	8086 based 16-bit Servo Computer	i386 compatible 32-bit Industry standard PC104 Module
		PASCAL based Servo Control Algorithm	RTAI based Servo Control Algorithm.
		DOS based Servo Simulator (NEWSMU)	Qt based GUI tool

Table 1-1 Development Plans

Next coming section explains the software design concepts and structure of control algorithm. While porting antenna control software from Pascal to C, all the good features of existing software was pertained as it is and some additional modules like data logging, enhanced user interface are added without de-grading its performance. Section-3 of this document explains individual modules greater in detail and finally presents the overall flow of the servo software through its different modules.

2 Software Design Philosophy

Overview

This section gives the overall introduction about the servo software and constraints which are considered while writing the software. The GMRT servo system software is a complex piece of code. The complexity arises from the fact that a large number of functions are handled by it. The servo software was designed with the following design concepts and constraints.

Compatibility

The GMRT servo software is able to operate with the different hardware platforms which compliance with the PC104 standards as well as it compatible with the higher versions of RTAI and GNU/GPL Linux kernels and their related packages (like GCC, Qt etc.).

Extensibility

It is a systematic measure of the ability to extend a system and the level of effort required to implement the extension. Extensions can be through the addition of new functionality or through modification of exciting functionality. The new capabilities such as data logging, user interface, TCP/IP network communications are added to the existing PASCAL based servo software, and enhancement of antenna controller algorithm to support for servo calibration tests with some minor modification in the present architecture.

Modularity

The servo software consist of several functional units called modules like compensator, communication, antenna controller, safety and interlock, data acquisition etc. Each module is responsible for certain kind of goals. These functional components or modules are implemented and tested in isolation before being integrated with the original software.

Re-usability

The re-usability built in the software by the use of modular design. The chunk of procedure or source code that can be used again to add new functionality with slight or no modification. Reusable modules reduce the implementation time and memory requirement.

Maintainability

Maintainability involves a system of continuous improvement – learning from the past in order to improve the ability of the system as well as to cope with the new technologies. Earlier GMRT servo software designed with the PASCAL language, this version of software has limited resources for software up-gradation and software troubleshooting as well as this software will run only with the 8086 GMRT CPU's. But the PC104 based servo software are designed with the high level C language it helps to upgrade the system without major hurdles at the same time it will run on any processors having x86 architecture base.

Robustness

The servo software is designed in such a manner to operate continuously in the event of the failure of one or more components as well as it can predict the invalid inputs.

Reliability

The reliability refers to the consistency of a measure, (i e ability of a system to perform its required functions under stated conditions for a specified period of time).

2.1 Structure of Servo Software

The following tree structured diagram is used to illustrate GMRT servo system software design hierarchy. This tree structured diagram is drawn with different functional units or modules of servo system software. The structured software follows the following rules,

- Modules or functional units are arranged hierarchically.
- There is only one top-level module or function (i.e. GMRTACU) and Execution begins with top-level function.
- Program control must enter at its entry point and leave at its exit point.
- Control returns to the calling function when the lower level module completes its execution.

The Figure 2-1 gives an overview of the servo software structure. It essentially breaking the complex system (i.e GMRTACU) into its compositional sub-systems like *antenna_controller, host_handler, gui_handler, timer_ handler* etc. The different operational units use various functions and data types, these functions and data types are declared as global variables in their respective sections. These global entities are only accessed by it. These entities are invisible outside the operational units and any attempt to use them must be a member function of that particular operational unit. This is similar to the concept of encapsulation in an object-oriented design.

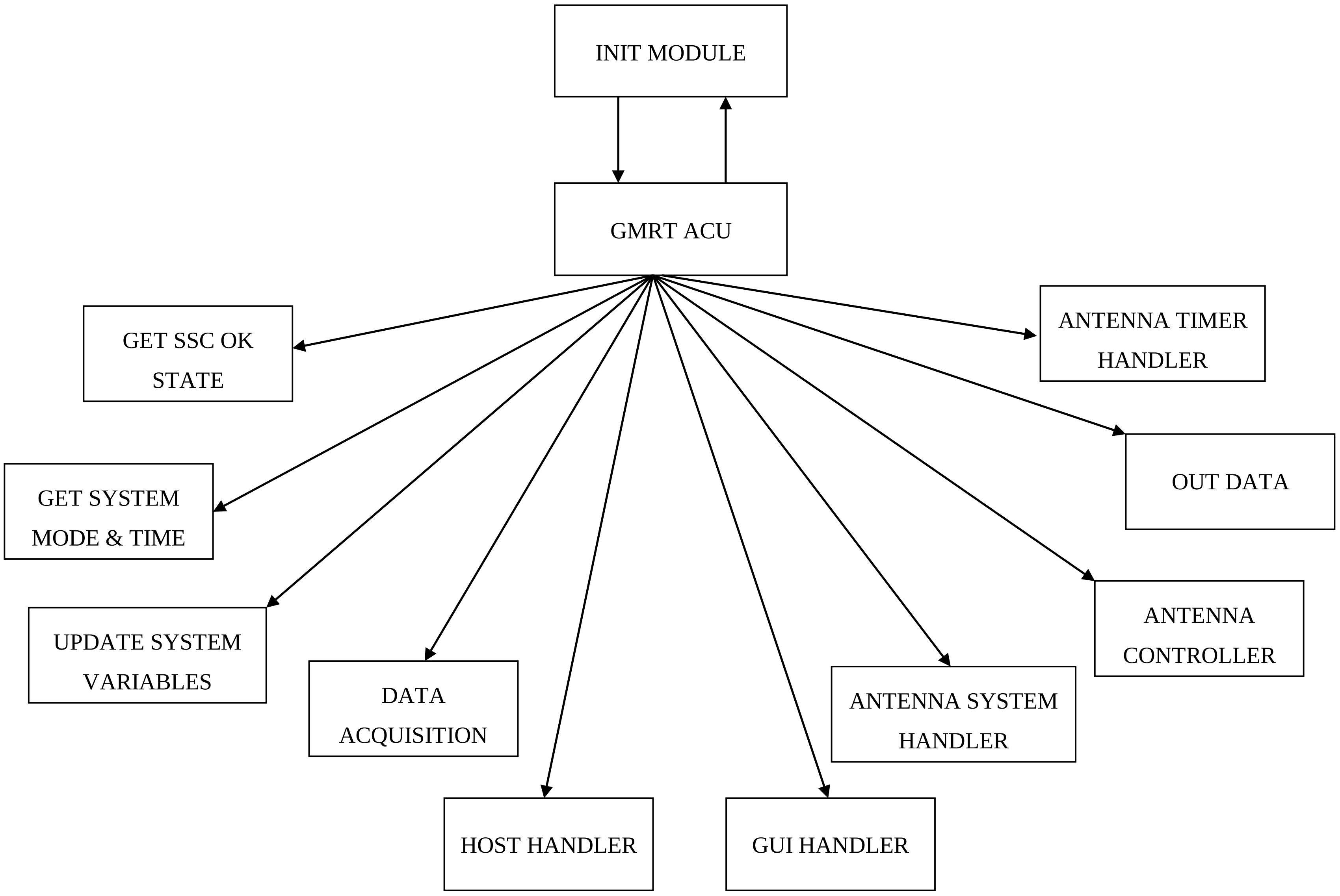

Figure 2-1 Structure of Servo Software

Another important feature of the software is its state machine based design. A state machine is a task which may, at any instant of time, be in one of the several states. A state is a condition of a system with regard to the completeness of a function. The actual need for the state machine concept can be appreciated by the realizing that software such as this has to take care of a number of functions in real time. So it cannot spend a large block of time in servicing one requirement to the exclusion of all others.

Another feature of the software is implicit multitasking and scheduling. A single processor can execute only one task at a time, meaning that the CPU is actively executing instructions for that task. So, an absolute simultaneity of operation cannot be achieved. However, if each job were to be given a very small chunk of processor time in which only the code of that job is processed, and at the expiry of that period the next job were to capture the processor for another small period and so on in a cycle, all jobs would be processed in a pseudo-simultaneous fashion. In this arrangement, each job would behave as if it had a processor solely to itself. Such systems, called time-sharing systems, therefore need a scheduling routine. This routine is responsible for allotment of time slices and swapping the program state variables at beginning and end of each jobs time slice. An implicit scheduler is built into the jobs and so they themselves have the responsibility of scheduling while the need for swapping is done away with.

In GMRT servo software, all major jobs are broken up into states thereby transforming jobs into state machines. Each state of state machine contains its related code in its definition. If any individual state code is long, it is further broken up into smaller states. An overall supervision routine called GMRTACU calls the state machine either directly or by calling some intervening routines. When a state machine called a conditional branching (if .... else) statement at the head of the machine transfers control to the appropriate state code. The decision of this branching is taken on the basis of the state valid at the time of exit from the machine in the previous iteration of the machine. This state stored in a global variable, and there is one state variable for each state machine. After branching to the applicable state, the state-code is executed and depending on whether the particular job is eligible for a state change the global state variable is updated and the machine is exited.

The servo software is made up of a collection of tasks. These tasks need to communicate with one another. This is enabled to an extent by the function implementing the task. The task located in various units need to pass message between them. However, because of the fact that a variable loses the value contained in it once it is out of its declared scope, there are chances that the message contained in a variable in one unit is lost when accessed from another unit. To overcome this, all data item which serve to carry message are declared global. Most of these global declarations are found in the GLBVARS.C. Some of the notable message passing variables is status of timers, command bytes, response to commands, system mode and the states of the various state machines.

3 Description of software modules

3.1 SSC and Host Communication

The antenna base computer and the servo computer are connected with RS232C communication link. The SSC_LINK layer module is entrusted with the responsibility for handling the hardware and link layer functions. The part of the software handling hardware is small. It consists of one function called *tx_rut* (transmit routine). This function reads from the serial communication port for reception and writes to the same for transmission. The availability of a byte for reading is indicated by a high B0 bit in the UART “Line Status Register (LSR)”. Transmission is done after ensuring that the transmit buffer is free that is indicated by the B1 bit of the LSR.

Apart from the hardware layer, this module incorporates the code for link layer. The job of link layer is to serve as a bridge between application and hardware layers so as to make the details of the hardware inconsequential to the application. To achieve this, the unit contains two state machines,

1. Transmit State Machine – tx_mc
2. Receiver State Machine – rx_mc

3.1.1 Transmitter State Machine

The function of transmitter state machine is to compile the data packet, transmit through physical channel and update the response received from host computer. The possible responses from the host are ACK and NAK,

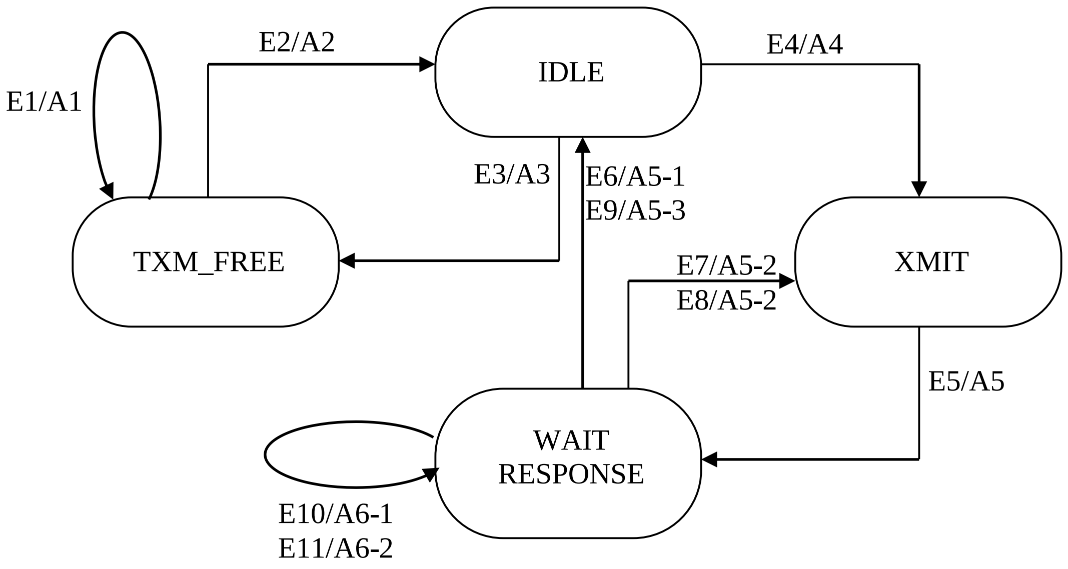

Figure 3-1Transmitter State Machine

For affirmative response (ACK), transmit state machine updates transmit buffer state into OK and put the transmitter buffer into conformed frame Q. For pessimistic response (NAK), re-transmits the same message packet, and waits for response and again NAK received, updates the transmit buffer state into ONLYNAK and sends the ENQ. Transmit machine waits for ENQ response, if NORESP is received, transmit machine reports link TIMEOUT.

The various actions performed by the transmitter state machine are listed as a event and action and Figure 3-1 shows the various states.

List of Events

E1 – req_frameq.hd is NULL

E2 – req_frameq is NOT NULL

E3 – req_frameq.hd is NULL

E4 – After Data Packet formed, update transmit machine state to "XMIT"

E5 – End of action A4 transmit machine state updated to "AWAIT_RESP"

E6 – Receiver response byte is ACK && tx_buff→stt is OK

E7 – Receiver response byte is NAK && ack_retry_count != 0

E8 – Receiver response byte is NAK && ack_retry_count == 0

E9 – Receiver response byte is NAK && ack_retry_count == 0 && tx_buff→stt is ONLY_NAK

E10 – tx_mc from gmrtacu with inv_code 3 && enq_retry_count != 0

E11 – tx_mc from gmrtacu with inv_code 3 && enq_retry_count == 0

List of Actions

A1 – No State Transition (Default State "TXM_FREE")

A2 – Transferred to IDLE state
 Copy the req_frameq.hd data into transmit buffer (tx_buff), and point to next pointer
 Assign acknowledge and retry count 2 for response and link status
 Compile the tx_buff data and form message packet with header, data and BCC.
 Update the transmit machine state to XMIT and enable COM_PORT Interrupt.

A3 – Change transmit machine state to default state "TXM_FREE".

A4 – Transmit data if transmit routine (txr_stt) state machine state is FREE.
 Initialize acknowledge timer (ack_timer) to 10
 Update txm_stt to AWAIT_RESP
 Update transmit data state (tx_data_stt) to PENDING.

A5 – If transmit machine is call from receiver machine with inv_code of 2,

1. Check the receiver response byte (rx_resp_byte) to ACK. If it is true,
    a. Increment link_tx_frames,
    b. Update transmit buffer state (tx_buff→stt) to OK,
    c. Put the transmit buffer (tx_buff) data into corresponding conformed frameq (conf_frameq),
    d. Update transmit machine state to TXM_FREE,
    e. Initialize ack_timer to zero.

2. Check receiver response byte (rx_resp_byte) to NAK,
    a. Initialize ack_timer to 0,
    b. Decrement ack_retry_counter,
    c. Increment link_retry_counter,
    d. Update txm_state to XMIT and transmit tx_buff again.

3. Check receiver response byte (rx_resp_byte) to NAK and ack_retry_count is zero,
    a. Initialize ack-timer to 0,
    b. Update tx_buff→stt to ONLYNAK,
    c. Put the tx_buff data into corresponding conformed frameq (conform_frameq),
    d. Update txm_stt to IDLE.

A6 – If tx_mc called from gmrtacu with inv_code 3,

1. If enquiry retry count (enq_retry_count != 0)
    a. Decrement enq_retry_counter,
    b. Increment link_enq_counter,
    c. Call transmit routine with ENQ byte,
    d. Update ack_timer to 10,
2. If enquiry retry count (enq_retry_count == 0)
    a. Initialize ack_timer to 0,
    b. Update tx_buff→stt to NORESP,
    c. Put the tx_buff data into corresponding conformed frameq (conform_frameq),
    d. Update txm_stt to IDLE.

3.1.2 Receiver State Machine

Receiver machine is responsible for decoding the information received from hardware routine (depends on status of LSR and interrupt generation *read_rx* reads serial data and put it into *rxd_char[256]*) after extracting data, receiver machine identifies the source and destination id's of received message and accordingly stores the data in the corresponding task queues. In order to decode the information *rx_mc* consist of seven states, Figure 3.2 shows the various states of this machine.

List of Events

E1 – Default state
E2 – rxd_char == DLE
E3 – rxd_char == ACK/NAK
E4 – rxd_char == ENQ
E5 – rxd_char == STX
E6 – rxd_char != DLE
E7 – rxd_char == DLE
E8 – rxd_char == ETX

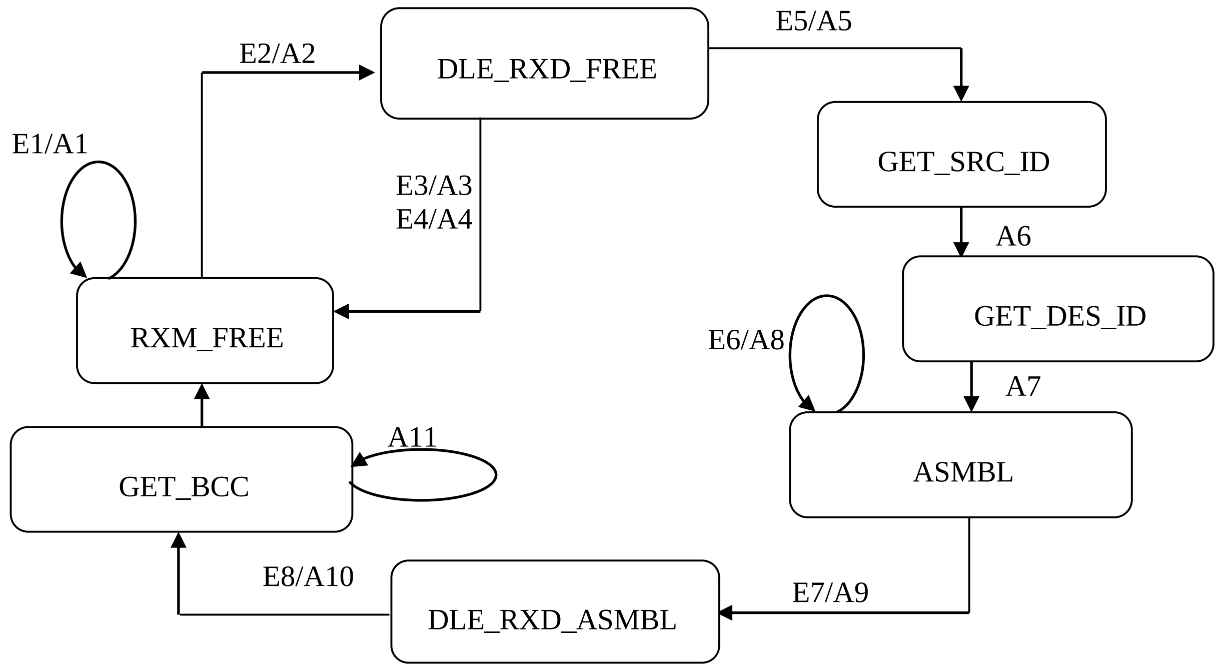

Figure 3-2 Receiver State Machine

List of Actions

A1 – rxm_stt = RXM_FREE

A2 – rxm_stt = DLE_RXD_FREE

A3 – rxm_stt = RXM_FREE
Update receiver response byte to rxd_char
Call transmit machine with inv_code 2, (tx_mc(2))

A4 – rxm_stt = RXM_FREE
Update res_tx_stt = PENDING
Call transmit routine (tx_rut()) to transmit DLE, last_response

A5 – Assign last response to NAK,
Assign binary check sum to zero
Assign data count to zero (rx_indx)
Update rxm_stt to GET_SRC_ID

A6 – Get source task id into rx_buff
Update rxm_stt to GET_DES_ID

A7 – Get destination task id into rx_buff
Update rxm_stt to ASMBL

A8 – Get the data into receiver buffer (rx_buff→data)
Update the binary Check Sum (rx_bcc) and data count (rx_indx)

A9 – Update rxm_stt to DLE_RXD_ASMBL

A10 – Update rxm_stt to GET_BCC

A11 – Update binary check sum

    no_buff != 1 && rx_bcc == 0 then update last response to ACK. Put the receiver buffer into corresponding task Q.
    no_buff == 1 && rx_bcc != 0 then update last response to NAK.
    Send last response.
    Update rxm_stt to RXM_FREE

3.2 Link Protocol

The protocol used for the communication between Station Servo Computer (SSC) and the Station Control Computer (SCC) is a RS-232-C full duplex link protocol with acknowledgment, error detection and retry mechanisms built in. It also provides task to task communication between the two stations.

This is a character oriented protocol using the following ASCII characters extended to 8 bits by adding a zero for bit 7.

3.2.1 Codes

The following protocol characters are used.

STX	02
ETX	03
ENQ	05
ACK	06
DLE	10
NAK	15

DLE - CONTROL/DATA delimiter.

1. Control Codes:

DLE STX Start of text

DLE ETX BCC End of text (followed by checksum)

DLE ACK Successfully received

DLE NAK Failed to receive

DLE ENQ Requests retransmission of last received

2. Data Codes:

DATA Characters 00-0F and 11-FF

DLE DLE to provide data transparency data 10h (DLE) is com-pended with DLE.

The messages exchanged between the two stations can be classified into message codes which are sent from the transmitter to the receiver, and response codes which are sent from the receiver to the transmitter DLE NAK and DLE ACK are the response codes whereas DLE STX, DLE ETX BCC, DLE ENQ and data codes are all message codes.

3.2.2 Link Layer Message Packets

A link layer message packet starts with a DLE STX, ends with a DLE ETX BCC, and includes all link layer data codes in between. Data codes can occur only inside a message packet. Response codes can also occur between a DLE STX and a DLE ETX BCC, but these response codes are not part of the message packet, they are called embedded responses.

DLE STX $\longleftarrow$ HEADER DATA (From Application Layer) $\longrightarrow$ DLE ETX BCC

The Header is a 3 bytes long arranged in the following order. Destination id, Source Id followed by the length of the data string.

Note: BCC is the two's complement of modulo-8 sum of all data bytes between DLE DLX BCC. It does not include any control code (or embedded response codes). For data 10 hex however only one of the DLE bytes is included in the BCC sum.

3.2.3 Message Flow

Both the stations A and B can initiate a message transmission simultaneously. The stations A and B have two machines each, a transmitter and a receiver. Transmitter A sends message frame on path1 to the receiver B which sends response codes on path 2 to the transmitter A. Similarly transmitter B sends message frame on path 3 to the receiver A which sends response codes to the transmitter B on path 4.

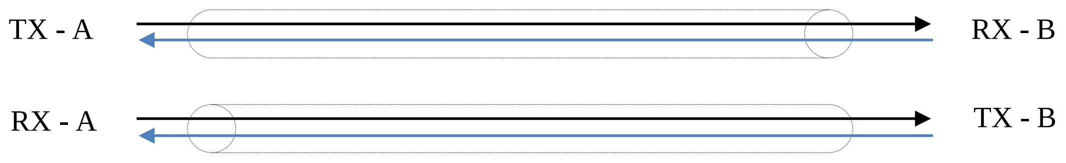

Figure 3-3 Message Flow Path

But physically only 2 paths (channels) are available for the communication between stations A and B.RECEIVE and TRANSMIT state machines at each station keep track of message / response codes flowing on single physical channel. Information and response codes originating at either of the stations does not required to be inter-leaved giving true full duplex performance.

Restrictions on messages

1. Minimum size of a valid user message is 1 byte and maximum size is 255 bytes.
2. The task number cannot be hex 10h.

3.2.4 Interface to the application software

A typical set of functions are required to provide the link layer interface to applications is given below,

1. com_buf_ptrget_free_buff (void)
2. **void** put_free_buff (com_buf_ptrptr)
3. **void** put_in_req (com_buf_ptrptr)
4. **com_buff_ptr**get_from_indq(BYTE task_id)
5. **com_buff_ptr**get_from_confq (BYTE task_id)

A. The function **"get_free_buff"** returns **TRUE** and a pointer to a free buffer if available otherwise returns FALSE.

B. The function **"put_free_buff"** puts a user specified buffer passed as a pointer to link‟s free pool.

C. The procedure **"put_in_req"** puts a user specified buffer passed as a pointer to link‟s request queue.

D. The function **"get_from_indq"** returns **TRUE** and a pointer to a buffer provided a frame is received for the indicated task_id otherwise returns false.

E. The function **"get_from_confg"** returns **TRUE** and a pointer to a buffer if link confirmation for the last transmission confirmation for the last transmission requests by the indicated task_id is /are ready (it returns the confirmation to the oldest frame that is not yet collected) otherwise returns **FALSE**.

3.2.5 Transmitter

1. The transmitter sends a message, starts a timeout and waits for response.
2. When DLE ACK is received it signals success.
3. When DLE NAK is received the message will be retransmitted. It restarts timeout and again waits. This will be repeated three times.
4. If timeout expires before a response is received, the transmitter sends a DLE ENQ to request the retransmission of last response. Its starts timeout and waits for a response. This also will be repeated 3 times.
5. Invalid responses are ignored.
6. The failures are indicated to the user, giving the reasons for failure, (i.e. **ONLY NAK** or **TIMEOUT**).

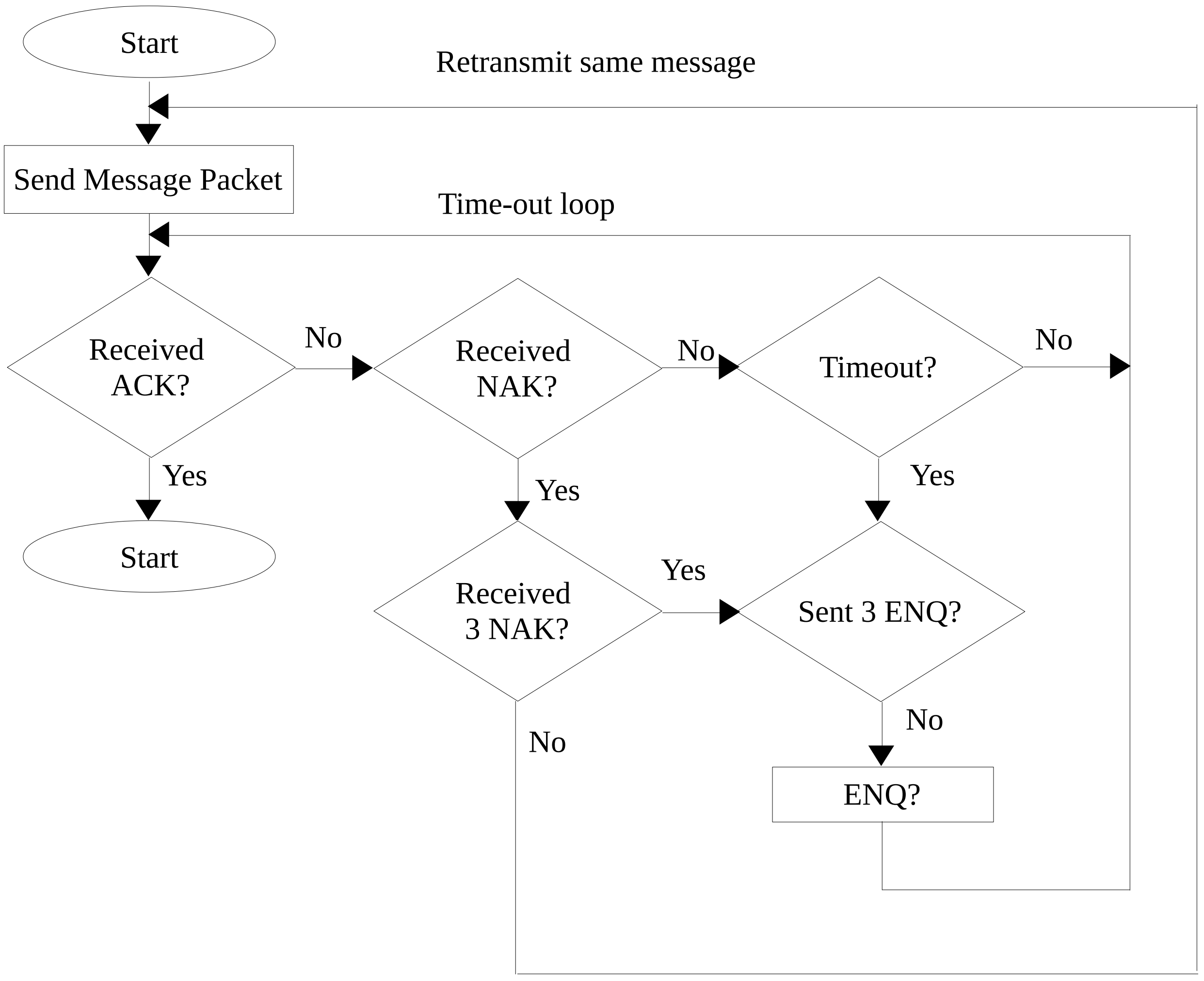

Figure 3-4 Transmit Message Protocol Diagram

3.2.6 Receiver

On the receiver side the possible error conditions are,

1. The message sink is full.
2. Parity Error.
3. BCC invalid.
4. DLE STX or DLE ETX BCC may be missing.
5. Too long or too short message.
6. Spurious control or data code outside the message.
7. DLE ACK is lost causing the transmitter to send a duplicate copy of message that has already passed to the message sink.

A last response sent is kept (ACK or NAK), which is initialized to NAK. The receiver sends this value for DLE ENQ. For a message with the same header byte ACK is sent, but the message is discarded. The last response is set to NAK if any code other than DLE STX or DLE ENQ is received before a message starts.

When building a message if the buffer overflows then the receiver continues summing BCC, but the data is discarded. If any other control codes other than ETX DLE BCC are received the message is aborted and DLE NAK is sent. If the message is OK, its header is compared with the last message. If it is the same then ACK is sent but the message is discarded. If the sink is full the response is recorded but not sent. It waits for DLE ENQ. If any other code is received then the response is changed from ACK to NAK and the receiver continues to wait for DLE ENQ. If DLE ENQ is received, it checks the sink status. If it is full, it continues waiting. If it is not full then the last response is sent and it accepts new message.

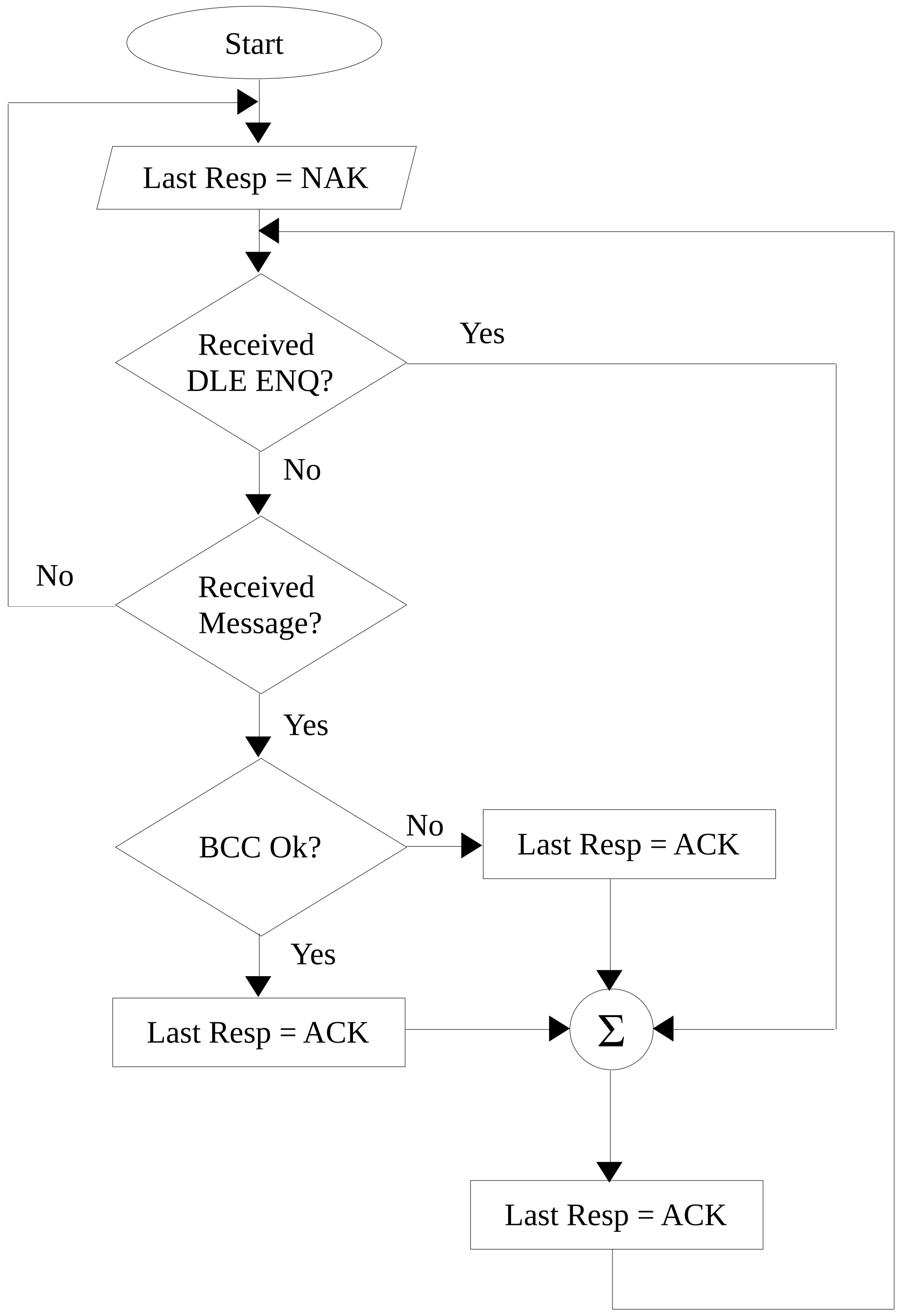

Figure 3-5 Receive Message Protocol Diagram

3.3 Host Interface

This module contains three major state machines namely,

- Event Machine
- Remote Machine
- Data Machine

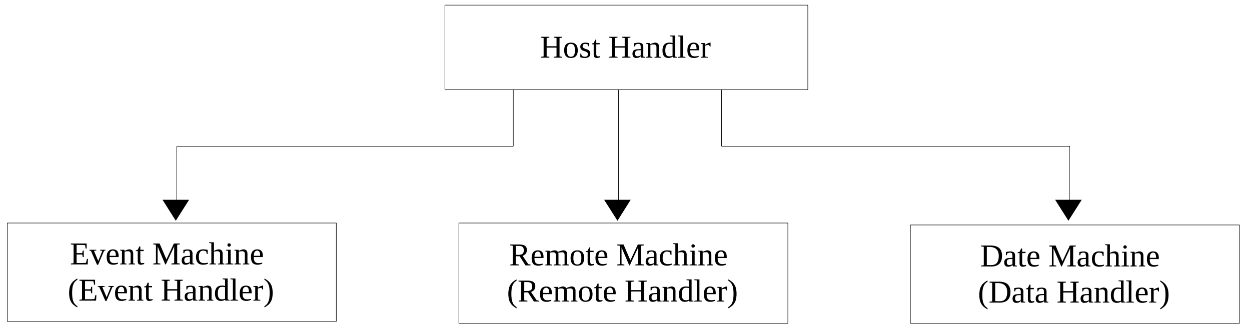
Host Handler
Event Machine (Event Handler)
Remote Machine (Remote Handler)
Date Machine (Data Handler)

Figure 3-6 Host Handler

The *host_handler* module is directly called by *gmrtacu* every 100 msec interval. The different task accomplished by this module is explained in following sections.

3.3.1 Event Machine

The event handler routine is responsible for updating the servo events to remote host computer as well as local HHT, irrespective of system mode. The event queue is of FIFO type, having character array of 256 variables. This queue was updated by an event code whenever,

1. There was a violation of system safety variables like motor currents, speeds, drive fault, wind speeds, link faults etc
2. Actuation of system safety limit switches in Azimuth, Elevations axis and Stow system.
3. Axis status like Axis ON, OFF and Interlocked.

Appendix.A lists the different event codes and messages for Stow system and AZ & EL axis.

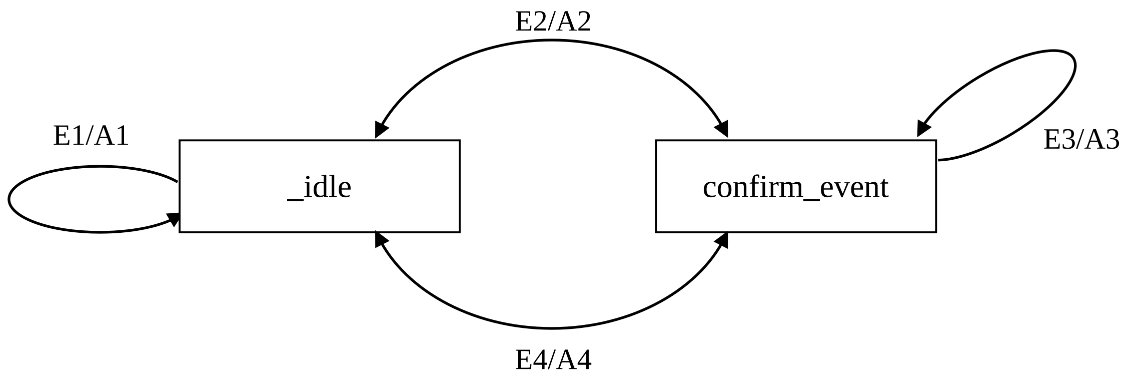

Figure 3-7 Event Handler State Machine

To transmit these high priority event messages to host computer and local GUI, event handler routine uses state machine approach having the two states namely*_idle* and *confirm_event*. Figure 3-7 shows the event state machine.

_idle: This is default state of this state machine. If there is no event in queue or there is no buffer pointer event machine retains in this state. On contrary event queue has event code and buffer pointer is available it compiles the event string with event id, response message code and event code and puts the buffer pointer into request queue to transmit. Update state to *confirm_event*.

_confirm_event: This state reads the response code from *get_from_confq(event_id)*, If response code is ACK it releases the buffer to *free_pool_buffer* and updates the state to _idle otherwise, it re-transmits the same event message to host and waits for response.

The following lists summaries the list of event and actions performed by this machine.

List of Events

E1 – (event_buff_hd == event_buff_tl) && (free_buff == 0)

E2 – (event_buff_hd != event_buff_tl) && (free_buff != 0)

E3 – get_from_confq(event_id) == NULL

E4 – get_from_confq(event_id) != NULL

List of Actions

A1 – retain in same state *(_idle)*

A2 – Compile the event string with event id, response message code and event code

Put the buffer pointer into put_in_req to transmit.

A3 – retain in *confirm_event* state.

A4 – release the buffer pointer and update the state to *_idle.*

3.3.2 Remote Machine

This module (Figure 3-8) is a bridge between the link layer and application layer (antenna system). It passes the received command message from host computer into host interface through *ind_frame_queue, confirm_frame_queue* and responses from the application layer into link layer through response string. Remote machine evaluates operational commands from host. It basically contains routines, which validates the messages received from host computer and accordingly updates *host_message_box*. The elements of *host_message_box* are listed in Table 3.1

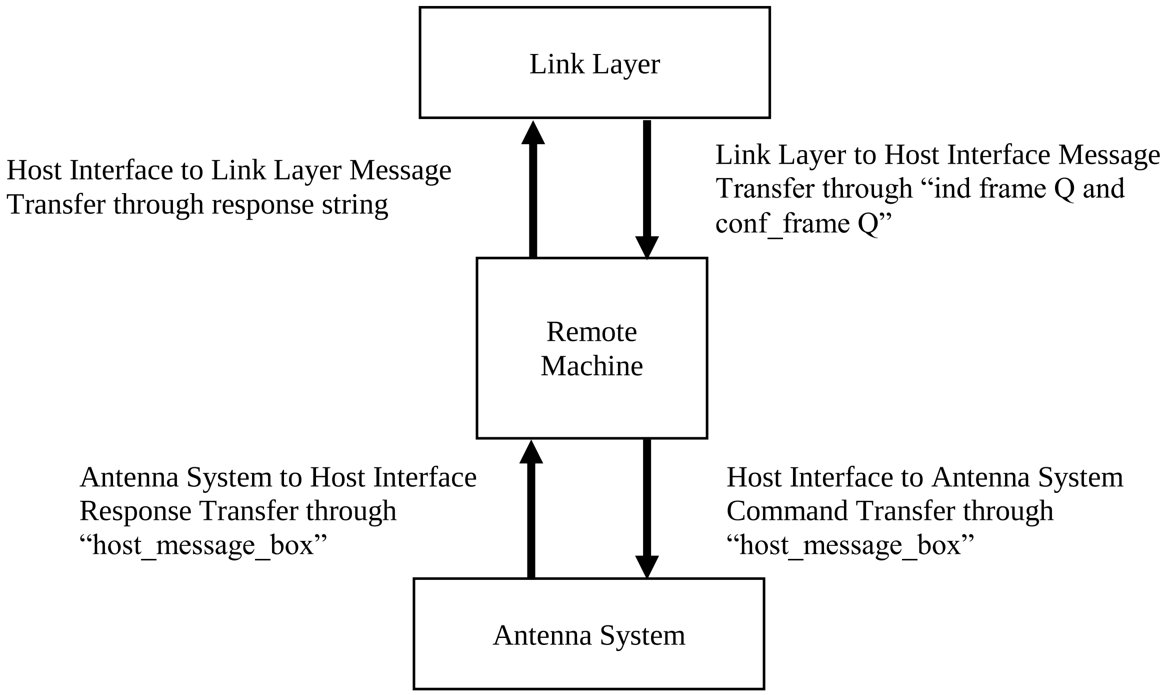
Link Layer
Host Interface to Link Layer Message Transfer through response string
Link Layer to Host Interface Message Transfer through "ind frame Q and conf_frame Q"
Remote Machine
Antenna System to Host Interface Response Transfer through "host_message_box"
Host Interface to Antenna System Command Transfer through "host_message_box"
Antenna System

Figure 3-8 Remote Interface State Machine

Elements of Host Message Box	**Control Flow**	**Description**
Command	H → A	Commands like Position, Track, Reset, Set time etc
Command String	H → A	Host command string with variables (P,B,-200:30:00,45:00:00)
Timer Count	H → A	Count Value for status reporting
Timer Status	H → A	Status of Timer either TMR_ON, TMR_OFF or TIME_OUT
Status of Command	H ↔ A	Either Pending or Accepted
Response	A → H	Response message from Antenna system (Accepted, Irreverent)

H → Host Interface (Link Layer),

A → Antenna System

Table 3-1 Elements of Host Message Box

The responses from antenna system are reported to host interface through host message box response element as listed in Table 3-1. The different tasks done by this machine are controlled by following three states,

- listen
- ring
- confirm_ack_nak

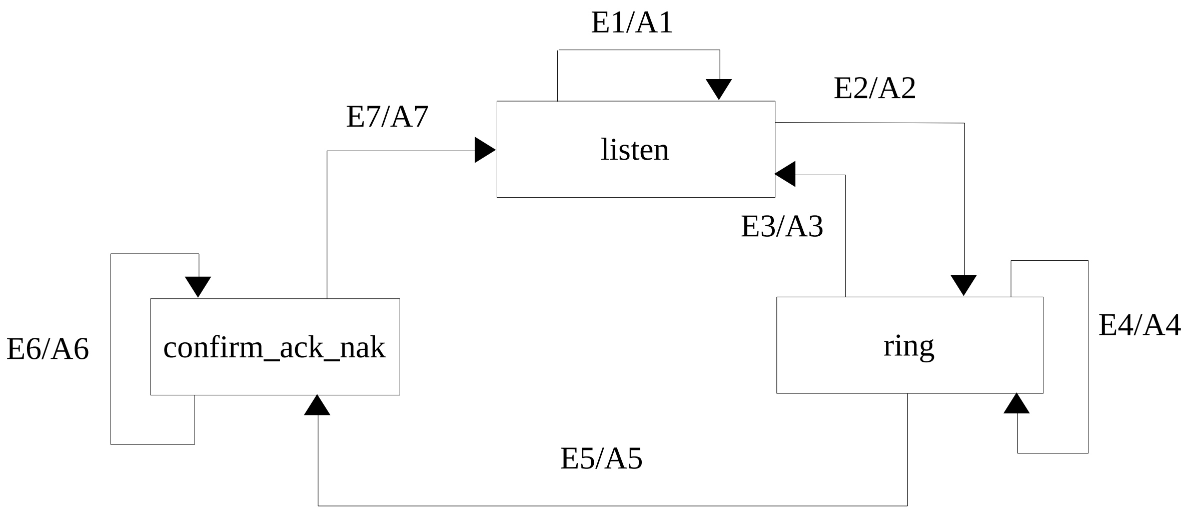

Figure 3-9 Host Handler State Machine

listen: In LISTEN state, remote machine is ready to read a message for command task from the received message queue and parse it. If a message is available on the queue, this will extract the first byte of the message which is identified as a command. Then this command byte is matched with the list of recognized command bytes. If command is not recognized, a failure message is sent to the host computer. If it is a recognized command, control is transferred to relevant functions which checks the syntax of the command and sends message to the antenna system about the arrival of a new command.

Commands accepted by remote machine in software version 2.6.5

- Abort
- Clod Start
- Close
- Hardware reset
- Hold
- Position
- Set Time of Day
- Stop
- Stow
- Stow Release
- Track

ring: This state basically used for inter-process communication between the other modules of software. In this state *rem_machine* is waiting for the response from *antenna_controller* for previously send command. Depends on the internal state of antenna controller it gives final response either ACCEPTED, IRREVALENT or TIMEOUT. Based on the response it forms response message packet and put it into *put_in_req*.

confirm_ack_nak: This state also used for the inter process communication with link layer. It checks the confirmation for sent final response to host and finally it updates *rem_machine* state to LISTEN.

Summary of Events and actions performed by the remote machine given below,

List of Events:

E1 – get_from_indq(cmd_id) == EMPTY
E2 – get_from_indq(cmd_id) != EMPTY
E3 – get_from_indq(cmd_id) != EMPTY
E4 – No response from antenna system
E5 – Response from antenna system
E6 – get_from_confq(cmd_id) == EMPTY
E7 – get_from_confq(cmd_id) != EMPTY

List of Actions:

A1 – Remains in Listen State
A2 – Get the first byte from message string
Command code matches with list of commands, and execute syntax check functions.
Update *host_message_box* and state to *ring*.
A3 – Get the first byte from message string
Command code not matches with list of commands, report failure, update state to *listen*
A4 – No response, remains in *ring* state
A5 – Update response string
Put response message to *put_in_req*
Update state to *confirm_ack_nak*
A6 – Remains in same state *(confirm_ack_nak)*
A7 – Update state to *listen*

3.3.3 Data Machine

This machine handles the monitoring commands requested by the host computer. The command identification and message passing protocol of this machine is similar to the previous section but only with two states, these states are namely *idle_* and *collect_confirm*. Commands accepted by this machine are (as per software version 2.6.5),

- analog variables
- digital variables
- angle variables
- set parameters
- antenna state
- software version
- error state

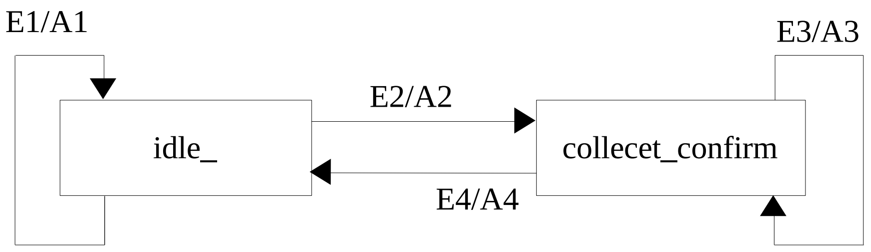

Figure 3-10 Data Handler State Machine

idle_: If data queue is not empty and command string is present, in this state *data_handler* reads the command string from *get_from_indq* with *data_id*. Identifies command by reading the first byte from command string, and accordingly calls the functions to send the requested monitoring parameter.

collect_confirm: This state is similar to *host_handler confirm_ack_nak* state. It waits for the response from *link_layer* before processing next command.

List of Events

E1 – (get_from_indq(data_id) == EMPTY) ||

((get_from_indq(data_id) != EMPTY) && Command code Not Exist)

E2 - ((get_from_indq(data_id) != EMPTY) && Command code Exist)

E3 – get_from_confq(data_id) == EMPTY

E4 – get_from_confq(data_id) != EMPTY

List of Actions

A1 - state = *idle_*

A2 – Call corresponding function to send monitoring data, update state to *collect_confirm*

A3 – Remain in *collect_confirm* state.

A4 - Update the state to *idle_* and release the previous command communication buffer.

3.4 GUI Interface

This module is exact duplicate of host interface. The only difference is that, whereas host interface module communicates with ABC, this unit does it with Graphical Servo Console that can optionally available at antenna base to control the antenna. The communication between Servo Console and this module are happening through Real Time Application Interface (RTAI) FIFOs.

RTAI FIFO is basically character based, asynchronous, one-way communication channel between RTAI task and GPOS task having a size limit given by user. The real-time task takes care of creation, destruction of FIFOs. RTAI modules uses *rtf_put()* and *rtf_get()* function calls to read and write into FIFOs. On the other hand, Linux user processes see RTAI-FIFOs as ordinary character device. It uses *open()* function call to access the RTAI FIFO in user process and *read()*, *write()* function calls for reading and writing on FIFOs. Figure 3-11 shows RTAI FIFOs in software version 2.6.5. The structure based data exchange mechanism implemented between RTAI task *(gmrtacu)* and GPOS task *(servoConsole)*.

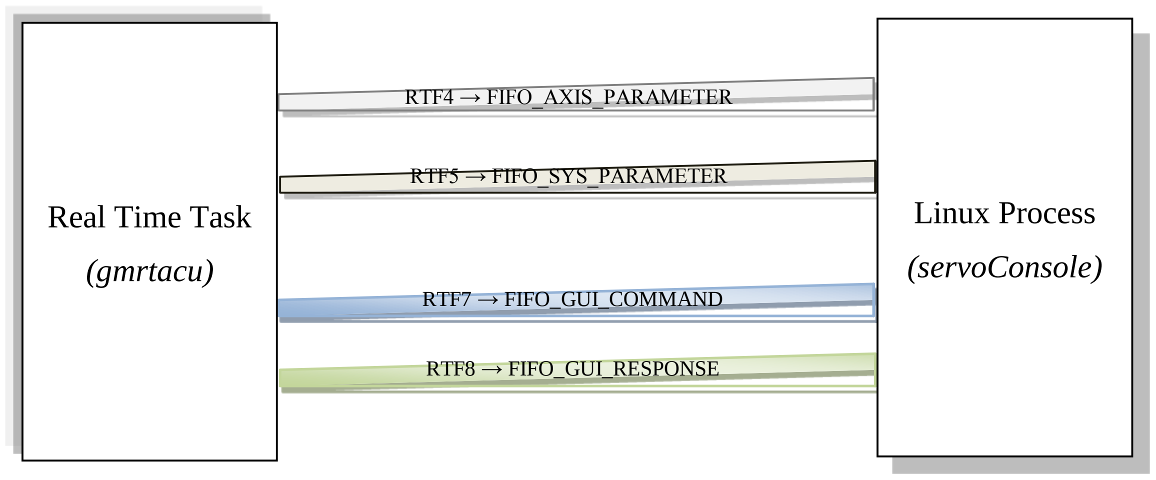

Figure 3-11FIFO Interface

3.4.1 GUI Data Machine

GUI data machine is responsible for updating system parameters and axis parameters to servoConsole every 100 msec interval with servo time. Table 3-2 & Table 3-3 list the elements of axis and system parameter communications structures.

S.No	Type	Parameter	Description
1.	float	current_pos	Encoder Position (deg)
2.	float	demand_target_pos	Target Position (deg)
3.	float	actual_target_pos	Soft start Position (deg)
4.	float	posn_err	Position Error in (deg)
5.	float	speed_dem	Speed Demand
6.	float	current_wind_vel	Wind Velocity (KMPH)
7.	float	motor1_current	Motor1 Current(amp)
8.	float	motor2_current	Motor2 Current(amp)
9.	float	motor1_speed	Motor 1 Speed (rpm)
10.	float	motor2_speed	Motor 2 Speed (rpm)
11.	int	state	Axis State (Positioning, Tracking, Limit Releasing etc)
12.	float	enc_offset	Encoder Offset (deg)
13.	float	axis_vel	Axis Velocity (deg/min)
14.	float	axis_acc	Axis Accleration (deg/min^2)

Table 3-2 GUI Interface Axis Parameter Structure Elements

S.No	Type	Parameter	Description
1.	char	station[20]	Station Number (C01,C02,etc)
2.	BYTE	event_code	Tag or identifier for servo events string
3.	BOOL	status[20]	Status Flags (dnpl, upl, ccw etc)
4.	rtc_timer	time	Servo Time (hrs:min:sec) 24Hrs Fromat
5.	date_time	date	System Date (dd-mm-yyyy)

Table 3-3 GUI Interface System Parameter Structure

3.4.2 GUI Command Machine

GUI command machine is similar to host interface remote machine, host interface remote machine serves the command request from host, where as GUI command machine serves the command request from local *servoConsole*. Similar to remote machine it receives the commands from *ind_frame_queue* after receiving it parses the command, and validates the syntax of command parameters. Finally it updates the *gui_message_box* with command code and parameters. Table 3.4 and Table 3-5 lists GUI command and response structure elements,

S.No	Type	Parameter	Description
1.	BYTE	command_code	Command identification number
2.	axis_type	axis	Command axis AZ, EL or BOTH
3.	flaot	Azangle	Commanded angle for Azimuth axis
4.	flaot	Ezangle	Commanded angle for Elevation axis
5.	rtc_timer	trk_time	Commanded track time

Table 3-4 GUI Commnad Structure

S.No	Type	Parameter	Description
1.	BYTE	cmd_type	Command identification number
2.	BYTE	stt	Response Accepted, Not Accepted

Table 3-5 GUI Command Response Structure

Commands accepted by servoConsole version 1.0 and servoSoftware version 2.6.5 are,

- Data Log Commands
  - ➢ Init_data_logger
  - ➢ Start Logger
  - ➢ Stop Logger

- Operational Commands
    - Hold
    - Position
    - Track
    - Stop
    - Close
    - Clod Start
    - Stow
    - Stow Release
    - Reset

3.5 Data Acquisition

This unit has the job of maintaining and regulating the flow of data to and from the various system variables. Further, such a data are often encoded in bit fields for hardware interaction. This unit carries functions which encode and decode data in the bit fields. The data it is concerned with are related to the auxiliary input – output card. This unit will play an important role in enabling the required modifications to the software that was undertaken.

The various I/O operations performed by this module are,

- sending configuration information (auxiliary display, display parameter, encoder offset etc) to slave cards,
- reading encoder angles for azimuth and elevation axis, reading wind readings,
- sending target angles, error angles,
- sending configuration information for ADC, DAC's maximum and minimum limits,
- reading the analog variables for motor speed, motor current, pot positions for both axis,
- reading digital inputs like mode input, CCW, CW, UP and DOWN limit switches, brake status, stow status, power supply status, etc.,
- Sending the AZ SIGNAL, EL SIGNAL, AZ_CAGE, EL_CAGE, AZ_POWER_ON and EL_POWER_ON signal to TTL cards.

Figure 3-12 shows the representation of slave I/O operations,

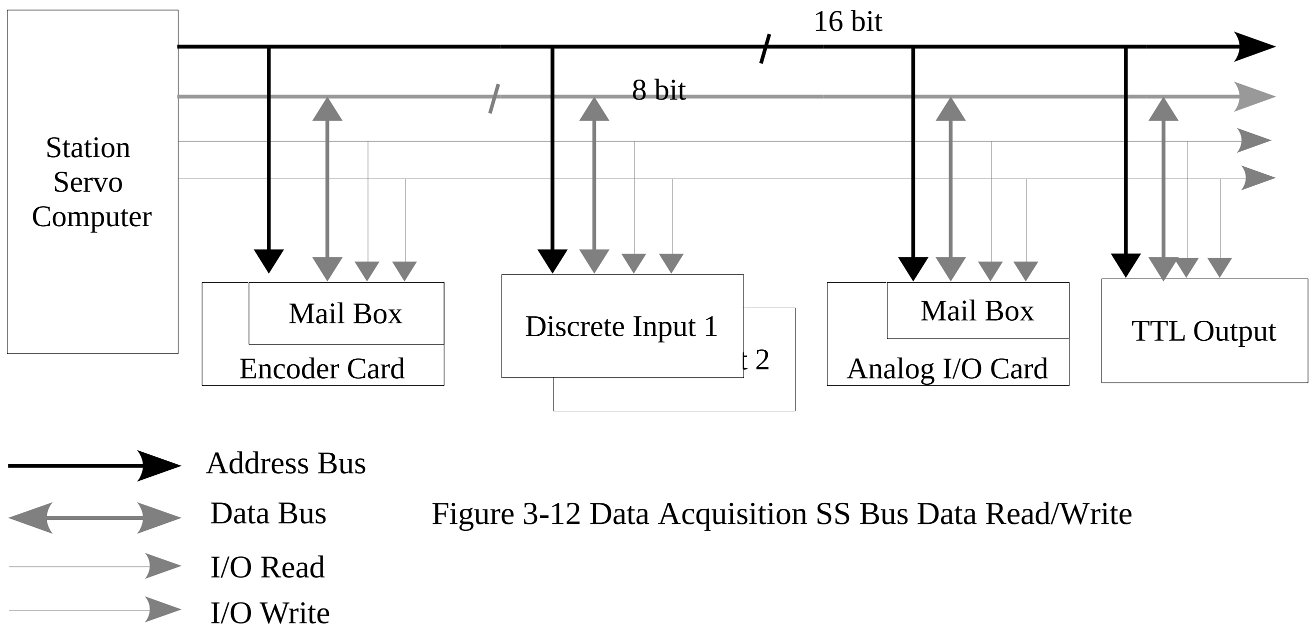

Figure 3-12 Data Acquisition SS Bus Data Read/Write

The I/O operation between host processor and slave boards are established through byte oriented serial mail box interface. During the I/O access, host process in busy wait mode (Poll mode), i.e polling the status of mail boxes for getting the field inputs like currents, speeds, encoder positions etc. However encoder and analog I/O slave processor reads the first byte through interrupt and polls for the subsequent bytes. Frame oriented command and response messages are exchanged through mail box. Figure 3-13 shows the command and response frame for slave board read and write.

FC	DATA

FC – Frame Control
DATA – 0 to 255 bytes of data depending on the command byte

Figure 3-13 Encoder and AIO command and response frame

The encoder and wind velocities are obtained in engineering units from the 8031 encoder board through mail-box interface. The host processor checks every time before obtaining the actual readings to see if the encoder board is reset recently. If so, it downloads a set of slave configuration information. These include the encoder offset (in binary 17 bits) and the auxiliary display mode (Wind/Target/Error/Pot positions/Display off) which are read from the *defaultsyscfg.h* file.

The analog inputs in engineering units (motor currents, speeds and pot positions and speed demand) are read and write from the analog I/O 8031 boards through mail box interface. The host processor downloads the channel configuration information to slave board at power on or slave board reset. This configuration information specifies span and zero for each channel which is required for binary to engineering conversion on the slave board. For further reference the detailed command and response list can be found in the GMRT design manual section 5.3 (Communication with Encoder and AIO Board).

3.6 Antenna System

This module contains one state machine *antenna_message_handler* whose main function is to see if a command message present in the message box. If a command is present, this machine identifier which axis the command is meant for (Azimuth, Elevation or Both) and accordingly updates the individual message box of the pertinent axis. Figure 3-14 shows the states of this machine.

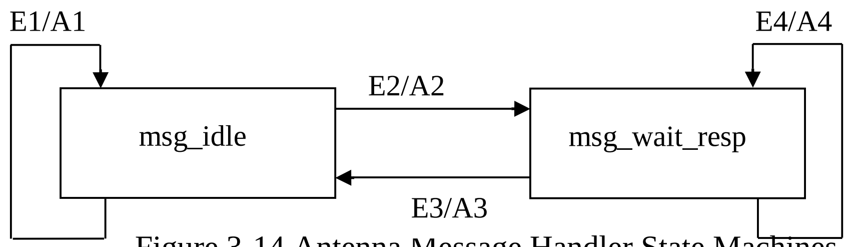

Figure 3-14 Antenna Message Handler State Machines

message_idle: During this state *antenna_message_handler* checks the *message_box* of host and GUI. If any command present in the box it accordingly calls *issue_antenna_command* function to pass the command parameters into *antenna_controller* and wait for response message from *antenna_controller*.

message_wait_response: In this state *antenna_message_handler* waits for response from antenn_controller. Depends on the received response in *antenns_sys_message_box* it updates host message box and restore its state to *message_idle.*

List of Events

E1 - (gui_msg_hndlr.stt != pending) || (host_msg_hndlr.stt != pending)

E2 - (gui_msg_hndlr.stt == pending) || (host_msg_hndlr.stt == pending)

E3 - antna_sys_msg.stt != pending

E4 - antna_sys_msg.stt == pending

List of Actions

A1 - Remain in msg_wait state

A2 - Call issue_antenna_command

A3 - Update state to msg_wait_resp

A4 - Update antna_sys_msg.resp to host_sys_msg.resp and update state to msg_idle.

A5 - Remain in msg_wait_resp state.

3.7 Antenna Controller

This modules contains broadly two state machines namely

- axis_controller_az
- axis_controller_el

Both the state machines are similar in structure and action. The AZ state machine is sub set of EL state machine because there is no stow logic in the AZ axis. Both of these state machines are highly modular containing a number of local functions themselves. The command in the message-box of each axis is used in the machine. The common states between the two machines are tabulated below,

State	AZ Axis	EL Axis
Stow Released Braked	_STRLSD_BRKD	STRLSD_BRKD
Position State	_POSITIONING	POSITIONING
Track State	_TRACKING	TRACKING
Manual State	_MANUAL_STT	MANUAL_STT
Limit Releasing	_LMT_RLSG	LMT_RLSG
Servo calibration	_SERVO_CAL	SERVO_CAL

Table 3-6 AZ and EL axis state table

In addition with above mentioned states, the following unique states are available for elevation axis to stow the antenna.

State	EL Axis
Stowed	STWD
Stowing	STWING
Stow Releasing	STRLSING
Stow Error	STWERR

Table 3-7 EL axis stow states

The further reference and detailed structure of the AZ and EL axis state machines are available in the GMRT design manual.

3.8 Command Pre-processor

The command pre-processor (CPP) presented in Figure 3-15 is used in the GMRT antennas to improve tracking and slewing motions. The CPP was designed to ensure smooth and stable antenna motion without limit cycling. However, these improvements do not prevent overshoots during position offsets, acquisition of targets or slewing. (An overshoot is an antenna position that exceeds the commanded position before approaching the commanded position).

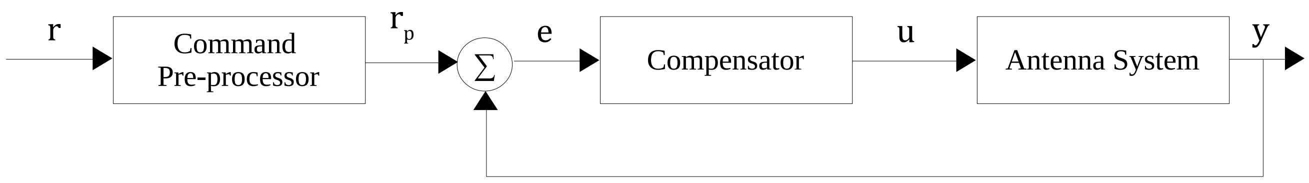

Figure 3-15 Location of Command Preprocessor

r – target position
r_p – actual target position
e – position error
u – speed demand
y – current antenna position

The CPP is an algorithm that generates a modified target command to the compensator. The modified target command (r_p) is identical to the target (r) one if the rate and accelerations of target command are within the limits. When the limits are met or violated, a command rate and acceleration are close to their maximal (or minimal) values. Figure 3.16 shows block diagram of command pre-processor,

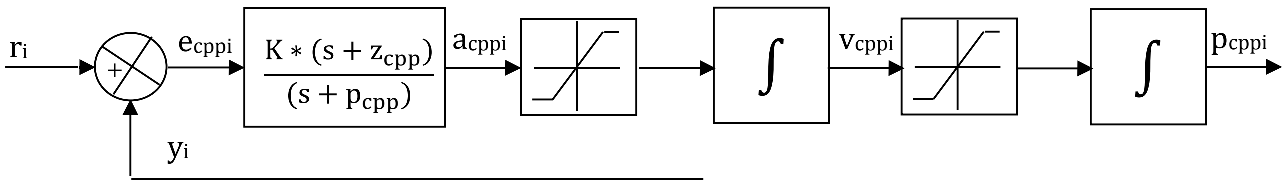

Figure 3-16 Command Pre-processor Block Diagram

r_i – current encoder position
e_{cppi} – command pre-processor error
a_{cppi} – command pre-processor acceleration
v_{cppi} – command pre-processor velocity
p_{cppi} – command pre-processor position
y_i – current antenna position

The CPP is a discrete system, with sampling time T_s = 0.1sec. The command at time instant T_i is denoted r_i, the preprocessed command output at same instant is denoted p_{cppi}. This p_{cppi} is actual input to the position loop compensator. The antenna rate limit and acceleration limit for AZ is 0.5deg/s, 0.1deg/s^2 and EL is 0.33deg/s, 0.06 deg/s^2 respectively. The typical values of pre-processor gain, pole and zero are K=9.8, z = 0.4082, p = 8.571.

$$G(s) = \frac{K * (s + z_{cpp})}{(s + p_{cpp})} \tag{3.1}$$

The digital implementation of equ (3.1) is given below,

$$a_{cppi}(T) = a * e(T) + b * e(T-1) + c * a_{cppi}(T-1) \tag{3.2}$$

$$v_{cppi}(T) = T_s * a_{cppi}(T-1) + v_{cppi}(T-1) \tag{3.3}$$

$$p_{cppi}(T) = T_s * v_{cppi}(T-1) + p_{cppi}(T-1) \tag{3.4}$$

The derivation of various constants a, b and c in equ (3.2) are listed in Appendix.B

$$a = \frac{K * (2 + T_s z_{cpp})}{(2 + T_s p_{cpp})}$$

$$b = \frac{K * (T_s z_{cpp} - 2)}{(2 + T_s p_{cpp})}$$

$$c = \frac{(2 - T_s p_{cpp})}{(2 + T_s p_{cpp})}$$

The various constants in the command pre-processor are computed real-time with the co-efficients in defaultsyscfg.h. These values are tunable at installation time through user interface.

3.9 Compensator

This module incorporate digital algorithm which serves as a compensator for digital position loop. The position loop is the outer loop of three nested loops that make up the GMRT servo system. Figure 3-17 shows block diagram of position loop model. It consists of PI controller, lead-lag compensator and speed demand limiter. The basic structure of position transfer function is common for both AZ and EL axis. The forward path of position loop compensator housed in PC104 servo computer which runs a compensation algorithm. The feedback loop has a 17 bit absolute position encoder. This encoder senses the instantaneous antenna position. It gives an angular resolution of 10 seconds of arc.

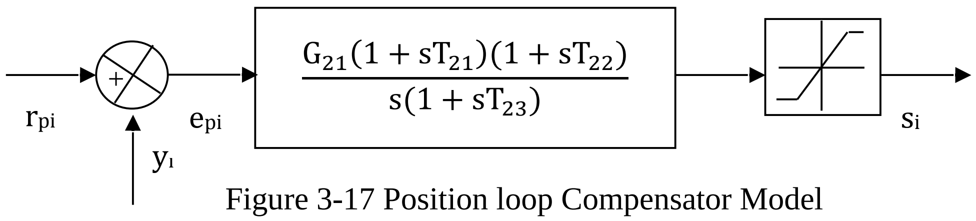

Figure 3-17 Position loop Compensator Model

r_{pi} – actual target position
e_{pi} – position error
s_i – speed demand
y_i – current antenna position
The position loop transfer function,

$$\mathrm{T.F} = \frac{G_{21}(1 + sT_{21})(1 + sT_{22})}{s(1 + sT_{23})} \tag{3.5}$$

The digital compensator uses Tustin Transformation to transform the above "s" domain transfer function to equivalent "z" domain. The resulting difference equation 3.6 is solved in real-time. The filter co-efficient stored in defaultsyscfg.h files. These co-efficient are evaluated while tuning the servo system, during installation.

$$Y_{z0} = a_1 * Y_{z1} + a_2 * Y_{z2} + b_0 * X_{z0} + b_1 * X_{z1} + b_2 * X_{z2} \tag{3.6}$$

Co-efficient's of equation 3.6 are,

$$a_1 = \frac{4T_{23}}{T_s + 2T_{23}}$$

$$a_2 = \frac{T_s - 2T_{23}}{T_s + 2T_{23}}$$

$$b_0 = \frac{G_{21} * [T_s^2 + 2T_s(T_{21} + T_{21}) + 4T_{21}T_{22}]}{2 * (T_s + 2T_{23})}$$

$$b_1 = \frac{G_{21} * [2T_s^2 - 8T_{21}T_{22}]}{2 * (T_s + 2T_{23})}$$

$$b_2 = \frac{G_{21} * [T_s^2 - 2T_s(T_{21} + T_{21}) + 4T_{21}T_{22}]}{2 * (T_s + 2T_{23})}$$

$G_{21} = \text{Integral gain}$

$T_{22} = \frac{K_p}{K_i}$

$T_s = \text{Sampling Time}$

$T_{21} = \text{Lead} - \text{Lag zero}$

$T_{23} = \text{Lead} - \text{Lag pole}$

$Y_{z0} = \text{Speed demand at } i^{th} \text{ instant}$

The derivation of equ 3.8 and filter constants are given Appendix.C. The various constants of filter are stored in defaultsyscfg.h during the initialization procedure all filter constants are read and co-efficients are computed.

3.10 GMRT Module

This module acts like master unit of software. It calls all other modules to perform different functions, because it is concerned with calling rather than executing code directly, some state machines are called directly by this unit like tx_mc, rx_mc, antenna_controller etc and other functions and modules are called in a logical sequence. Figure 3-18 shows the servo loop, it executes all the servo tasks at every 100 msec interval.

Summary of different functions and tasks done by main gmrtacu modules are listed below,

- The receiver machine *(rx_mc)* receives messages from host computer as well as local GUI through the interrupt, after extracting command string from message it stores command string into linked list queue with specific task id. (Eg: cmd_id, data_id and event_id).

- Communication handler reads linked list queue with specific task_id for serving request received from host computer like handling events, commands and monitoring datas, and accordingly updates *host_message_box*. Out of above mentioned three tasks events are handled with high priority, next operational commands and finally monitoring data.

- Antenna system handler check the availability of new command in the *host_messgae_box* as well as it identifies command source (either host or gui) and accordingly updates the *antenna_message_box* with relevant command.

- Data acquisition module executed to update filed inputs and system parameters, based on the acquired data all operational and safety limits are verified with pre-defined limits and gives the conformation to execute the antenna controller modules otherwise reports command failure to host.

- Next *antenna_controller* calls AZ and EL axis state machines in succession. Depends on the state of *antenna_message_box* individual axis state machines executes their command and accordingly updates *antenns_message_box* response.

- Next control reaches to data acquisition modules with *out_data* function to update speed demand, wind velocity, target position, antenna state to local display.

- At the end *antenna_timer_handler* called to update various internal software timers, these timers avoid control to be in specific state machine for longer time. If timer expires it generates the timeout signal and brings back control also it ensures the time synchronization between different tasks.

Appendix.D lists the various structures, unions, arrays, enumerated data types and circular buffers used in software.

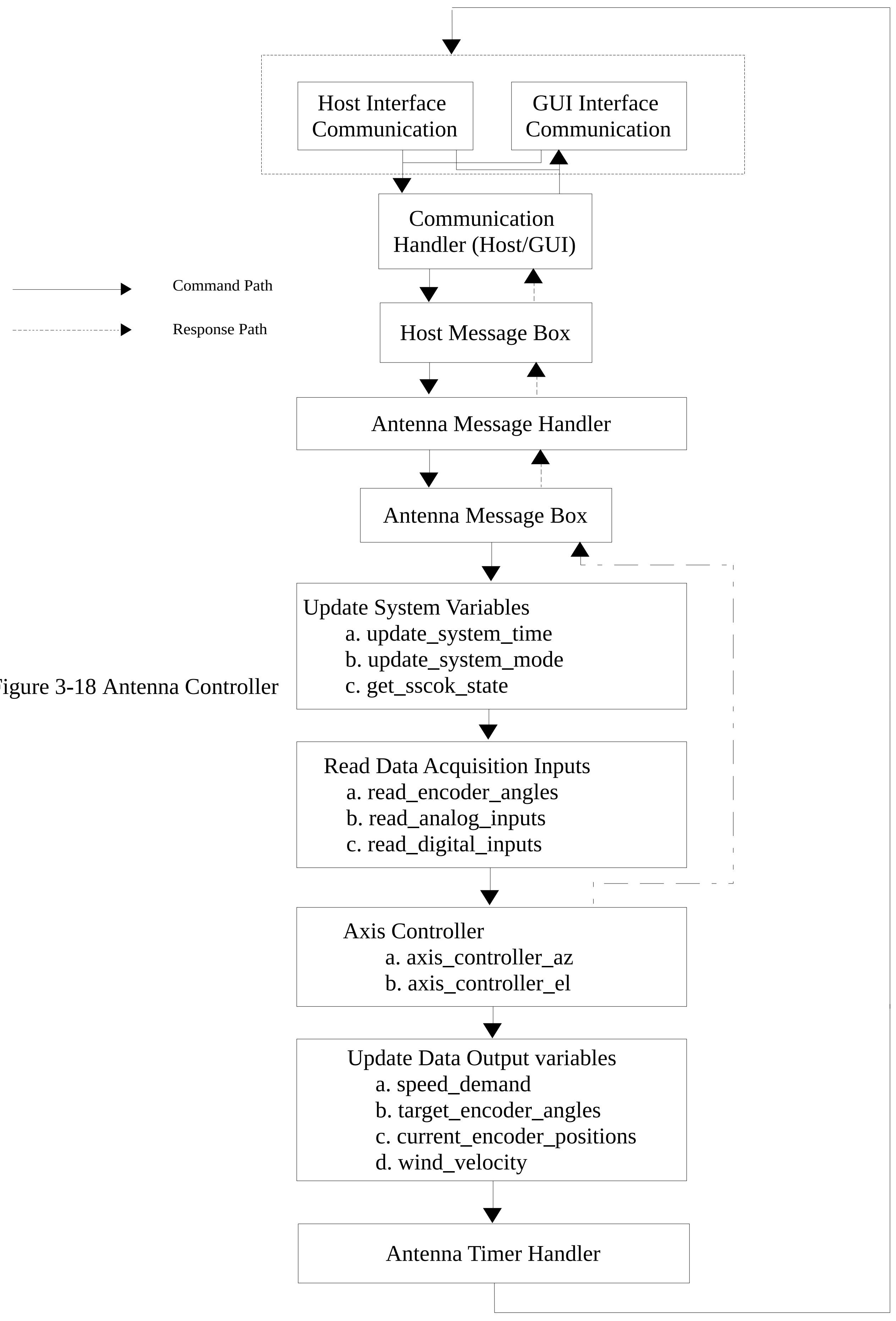

Figure 3-18 Antenna Controller

4 Verification and Validation

Various tests are done on software and hardware to verify its performance under different conditions in laboratory and antenna. The actual results of various tests are complying with the existing system standards and performance. The detailed report was prepared for different tests conducted, this report was circulated into different forums, after getting approval, phase – I development (replacement of 8086 based SSC into PC104 based SSC) moved from development stage to mass production and execution stage in GMRT antennas. Presently few of the central square (C01, C02, C03 and C05) antennas run with the upgraded servo computer.

Appendix.A Servo Events

List of servo Events as per software version 2.6.5

AZ CMD SUCCESSFUL

AZ CMD FAILED

AZ CMD ABORTED

AZ AXIS INTERLOCK

AZ AXIS ON

AZ AXIS OFF

AZ CW LIMIT

AZ CCW LIMIT

AZ LIMIT EXITED

AZ CURRENTS HI

AZ SPEED HI

AZ TRK Q DISCARDED

AZ AXIS/ENC FAULT

EL CMD SUCCESSFUL

EL CMD FAILED

EL CMD ABORTED

EL AXIS INTERLOCK

EL AXIS ON

EL AXIS OFF

EL UP LIMIT

EL DOWN LIMIT

EL LIMIT EXITED

EL CURRENTS HI

EL SPEED HI

EL TRK Q DISCARDED

EL AXIS/ENC FAULT

EL STOW ERR

EL STOW RLSG

EL STOW RLSD

EL STOWING

EL STOWD

EL STOW POSN

WIND HIGH

SYS MODE CHANGED

EMPARK STARTED

SSC RESET

SSC NOT OK

EVENT Unknown

Appendix.B Command Pre-processor Equations

Command Pre-processor Equation

Command pre-processor equation in section 3.8 as follows,

$$G(s) = \frac{K * (s + z_{cpp})}{(s + p_{cpp})} \qquad \text{B.1}$$

The above s domain transfer function (equ B.1) transferred into discrete z domain by sub-suiting s into,

$$s = \frac{2}{T}\left[\frac{z-1}{z+1}\right] \qquad \text{B.2}$$

By sub-suiting equ B.2 in equ B.1,

$$s + z_{cpp} = \frac{2}{T}\left[\frac{z-1}{z+1}\right] + z_{cpp}$$

$$= \frac{2(z-1) + T(z+1) * z_{cpp}}{T(z+1)}$$

$$s + z_{cpp} = \frac{z(2 + Tz_{cpp}) + (Tz_{cpp} - 2)}{T(z+1)} \qquad \text{B.3}$$

and

$$s + p_{cpp} = \frac{z(2 + Tp_{cpp}) + (Tp_{cpp} - 2)}{T(z+1)} \qquad \text{B.4}$$

Sub-suiting equ B.3 and B.4 in equation B.1

$$\frac{y(z)}{x(z)} = \frac{K * [\,\{z(2 + Tz_{cpp})\} + (Tz_{cpp} - 2)]}{\{z(2 + Tp_{cpp})\} + (Tp_{cpp} - 2)}$$

Re-arranging the above equation,

$$y(z)[\{z(2 + Tp_{cpp})\} + (Tp_{cpp} - 2)] = x(z)K * [\{z(2 + Tz_{cpp})\} + (Tz_{cpp} - 2)]$$

Re-writing above equation with coefficients of z,

$$z(2 + Tp_{cpp})y(z) + (Tp_{cpp} - 2) = zK(2 + Tz_{cpp})x(z) + K(Tz_{cpp} - 2)x(z)$$

Dividing above equation by $z(2 + Tp_{cpp})$,

$$y(z) + \frac{(Tp_{cpp} - 2)}{(2 + Tp_{cpp})} z^{-1} y(k) = \frac{K(2 + Tz_{cpp})}{(2 + Tp_{cpp})} x(z) + \frac{K(Tz_{cpp} - 2)}{(2 + Tp_{cpp})} z^{-1} x(z) \qquad \text{B.5}$$

Equation B.5 can be re-written as follows,

$$y(k) + \frac{(Tp_{cpp} - 2)}{(2 + Tp_{cpp})} y(k-1) = \frac{K(2 + Tz_{cpp})}{(2 + Tp_{cpp})} x(k) + \frac{K(Tz_{cpp} - 2)}{(2 + Tp_{cpp})} x(k-1)$$

$$y(k) = \frac{K(2 + Tz_{cpp})}{(2 + Tp_{cpp})} x(k) + \frac{K(Tz_{cpp} - 2)}{(2 + Tp_{cpp})} x(k-1) - \frac{(Tp_{cpp} - 2)}{(2 + Tp_{cpp})} y(k-1) \qquad \text{B.6}$$

Equation B.6 can be written in simplified form as follows,

$$y(k) = a * x(k) + b * x(k-1) - c * y(k-1) \qquad \text{B.6}$$

Where co-efficients a,b,c are

$$a = \frac{K(2 + Tz_{cpp})}{(2 + Tp_{cpp})}$$

$$b = \frac{K(Tz_{cpp} - 2)}{(2 + Tp_{cpp})}$$

$$c = \frac{(Tp_{cpp} - 2)}{(2 + Tp_{cpp})}$$

Appendix.C Compensator Equations

The position loop compensator implemented in software is proportional and integral controller with lead-lag filter.

Equation for PI compensator,

$$G_1(s) = K_p + \frac{K_i}{s} \tag{C.1}$$

$K_p = \text{Proportional Gain}$

$K_i = \text{Integral Gain}$

Equation for lead-lag filter,

$$G_2(s) = \frac{1 + sT_{21}}{1 + sT_{23}} \tag{C.2}$$

$T_{21} = \text{Lead} - \text{Lag Filter Zero}$

$T_{23} = \text{Lead} - \text{Lag Filter Pole}$

Complete transfer function of PLC,

$$G(s) = G_1(s) * G_2(s) \tag{C.3}$$

$$G(s) = \left[K_p + \frac{K_i}{s}\right]\left[\frac{1 + sT_{21}}{1 + sT_{23}}\right] \tag{C.4}$$

Equation C.4 can be re-written as follows,

$$G(s) = K_i\left[\frac{1 + s\left(\frac{K_p}{K_i}\right)}{s}\right]\left[\frac{1 + sT_{21}}{1 + sT_{23}}\right]$$

Replace $\frac{K_p}{K_i}$ with T22 and K_i with G_{21} in above equation,

$$G(s) = G_{21}\left[\frac{1 + sT_{22}}{s}\right]\left[\frac{1 + sT_{21}}{1 + sT_{23}}\right] \tag{C.5}$$

To convert the equation C.5 into discrete time domain, replace s with following function,

$$s = \frac{2}{T}\left[\frac{z - 1}{z + 1}\right] \tag{C.6}$$

Sub suiting equation C.6 in equation C.5,

$$1 + sT_{21} = 1 + \left[\frac{2(z-1)}{T(z+1)}\right] T_{21}$$

$$= \frac{\{T(z+1)\} + T_{21}\{2(z-1)\}}{T(z+1)}$$

$$1 + sT_{21} = \frac{(T + 2T_{21})z + (T - 2T_{21})}{T(z+1)} \tag{C.7}$$

Similarly,

$$1 + sT_{22} = \frac{(T + 2T_{22})z + (T - 2T_{22})}{T(z+1)} \tag{C.8}$$

$$1 + sT_{23} = \frac{(T + 2T_{23})z + (T - 2T_{23})}{T(z+1)} \tag{C.9}$$

Sub-suite C.6, C.7 and C.8 in equation C.5,

$$G(z) = \frac{G_{21}\left[\frac{(T + 2T_{22})z + (T - 2T_{22})}{T(z+1)}\right]\left[\frac{(T + 2T_{21})z + (T - 2T_{21})}{T(z+1)}\right]}{\left[\frac{2(z-1)}{T(z+1)}\right]\left[\frac{(T + 2T_{23})z + (T - 2T_{23})}{T(z+1)}\right]}$$

Re-arranging above equation in input, output form,

$$\frac{Y(z)}{X(z)} = \frac{G_{21}[(T + 2T_{22})z + (T - 2T_{22})][(T + 2T_{21})z + (T - 2T_{21})]}{2(z-1)[(T + 2T_{23})z + (T - 2T_{23})]}$$

After cross multiplication and writing co-efficient of z,

$$Y(z) = a_1 z^{-1} Y(z) + a_2 z^{-2} Y(z) + b_0 X(z) + b_1 z^{-1} X(z) + b_2 z^{-2} X(z)$$

The above equation can be written in discrete form as follows,

$$Y(k) = a_1 Y(k-1) + a_2 Y(k-2) + b_0 X(k) + b_1 X(k-1) + b_2 X(k-2) \tag{C.10}$$

The constants of equation C.10 are listed below,

$$a_1 = \frac{4T_{23}}{T + 2T_{23}}$$

$$a_2 = \frac{T - 2T_{23}}{T + 2T_{23}}$$

$$b_0 = \frac{G_{21}[T^2 + 2T(T_{21} + T_{22}) + 4 * T_{21}T_{22}]}{2 * (T + 2T_{23})}$$

$$b_1 = \frac{G_{21}[T^2 - 4T_{21}T_{22}]}{2 * (T + 2T_{23})}$$

$$b_2 = \frac{G_{21}[T^2 - 2T(T_{21} + T_{22}) + 4 * T_{21}T_{22}]}{2 * (T + 2T_{23})}$$

Appendix.D Data Structures

Owing to the complexity of the system, several data and associated structures are used in the software. There are some instances of simple data types like integer, double, character and boo-lean but majorities of data item are composite types containing a number of simple and other composite types. The description of the important data structures are given below.

Arrays

Arrays are collection of similar data items in a numbered sequence where any element can be accessed at any time. Even-though C language permits multidimensional array support, all explicitly declared arrays in the software are of single dimension. The two axis controller needs similar data items. These data items are, therefore made into a two element, single dimension array, which makes duplicating code for the two axes unnecessary.

Structures & Unions

A structure is a collection of variables under a single name. These variables can be of different types, and each has a name which is used to select it from the structure. A structure is a convenient way of grouping several pieces of related information together. Structures make up a large portion of the data types used in the software. These structures are sometimes used as sub-types in other structures. Unions and Structures are identical in all ways, except for one very important aspect. Only one element in the union may have a value set at any given time. Unions are mainly used to conserve memory. While each member within a structure is assigned its own unique storage area, the members that compose a union share the common storage area within the memory.

Circular Buffers

These are basically of array type but deserve a separate mention due to their important and extensive use. There is an RS 232 serial communication link for communication between remote computers and the antenna base computer. The buffers mentioned here are used for the communication over this channel. Each buffer can hold a message in an array, which is received one by one. For transmitting, the buffer holds the message compiled to add the start, stop, check-sum and header bytes.

Queues

These are data structures in which access is in First In First Out (FIFO) sequence. The data item that joins the queue earliest is put at the head if the queue. All extractions if data items are possible at the head only. Every member of the queue is an implementation of a linked list. If the queue is a linear queue, the last member or the tail points to a null location. In software, for every task there are three queues one for the send request, another for the conformation status of message sent, and a third for received messages. All three are queues of *com_buf_typ* structure.

Appendix.E Embedded System

Embedded System

This section gives the general introduction of Embedded Linux system, types of Embedded Systems and Real Time Linux.

An Embedded System is a combination of hardware and software, and other mechanical parts, designed to perform a specific function or dedicated task. As well as it is one of the component or sub-system of the larger system. The simplest embedded system contains the input and output capability as well as control logic which is stored in the system firmware.

Examples for such types of embedded systems are Mobile Phones, general purpose computes, and automatic braking systems in car. If embedded system is designed well the existence of processor, input and output circuitry and software could be completely unnoticed by a user of the device.

Real Time Systems

One sub class of the embedded system is Real Time System. A real time system is a computer system that has a timing constraint. In other words, real time systems are specified in terms of its ability to make certain calculations or decisions in a timely manner. These important calculations are said to have deadlines for completion. For all practical purpose a missed deadline is just as bad as a wrong answer. The more sever the consequence, the more likely it will be said that the dead line is “hard” and, thus a system is *hard-real time* system. The real time systems at the other end of this continuum are said to have “soft” deadlines such, systems are called *soft real time* systems.

Hard-Real Time Systems

A system is considered as a hard-real time if it can answer to an internal or external stimulus **“within a given maximum amount of time”**. The hard-real time systems are used wherever failing to react in time can cause a system failure or damage, or put its user in danger.

Typical examples,

- Flight Navigation Systems
- Antenna Control Systems
- Satellite Communications etc,

The following list of requirements defines a hard-real time system. The absence of predictability for any one of these items disqualifies a system from being a hard-real-time system. System time is a managed resource. Timing resources are managed with the highest possible level of precision.

- Guaranteed worst case scheduling jitter. If a task needs to be happening within a certain deviation, it is guaranteed to occur.

- Guaranteed maximum interrupt response time. As with scheduling latency, interrupt are guaranteed to be acknowledged and handled within a certain window.
- No real-time event is ever missed. Under no circumstances will a scheduled task not to be run on time, an interrupt be missed, or any other event the real-time code is interested n.
- System response is load independent. Execution of real-time tasks is guaranteed to fall within the worst case value range, regard less of system load factor.

A system that can fulfill these criteria is fully deterministic and considered to be "*hard real-time*".

Soft Real-Time Systems

In Some cases missing an event is not critical, such as in video applications where a missed frame or two is not fatal, a "*soft real-time* " system may do. Such a system is characterized by the following criteria,

- The system can guarantee a rough worst-case average jitter, but not an absolute worst case scenario.
- Events may still be missed occasionally. This is better than the non real-time system, as there is more control over response, but as the absolute worst case is unknown, events such as interrupts may still be lost.

Soft real-time systems are statistically predictable for the average case but a single event cannot be predicted reliably. *Soft real-time* systems are generally not suited for handling mission critical events.

Need of Real Time Operating Systems

An operating system is an environment that enables the user to execute their programs on the system hardware. Without an operating system users need to write applications at the machine level, which only the hardware can understand and execute. However, this extremely difficult, and besides, its not portable for different hardware sets-thus the need for an environment in which a person can use a high level languages to interact with the system. Besides interfacing between user and hardware, the operating system provides several other important and useful features, such as multitasking, in which many tasks/applications can run on a single system, with the operating system allocating available resources (CPU, main memory, input/output etc) to the applications, based on the priority and criticality of each. In addition, the operating system should provide the tools for application development – a software development kit, debugging, tracing and monitoring support. In short, the core objectives of the operating system are: user convenience, efficient resource allocation and support for development and testing of application. An operating system mainly consists of the kernel and utilities. The core of the operating system is kernel which runs directly on hardware. Its main job is to execute a given program on behalf of the user, and provide the result of execution back to the user in human readable format.

Monolithic Kernel & Micro Kernel

The main subsystems of a kernel are process management, file management, memory management and network management. The organization of these subsystem in the kernel leads to classification into monolithic and microkernel. A kernel that groups all these subsystems together in a single file is called monolithic kernel as shown in Figure E-1.

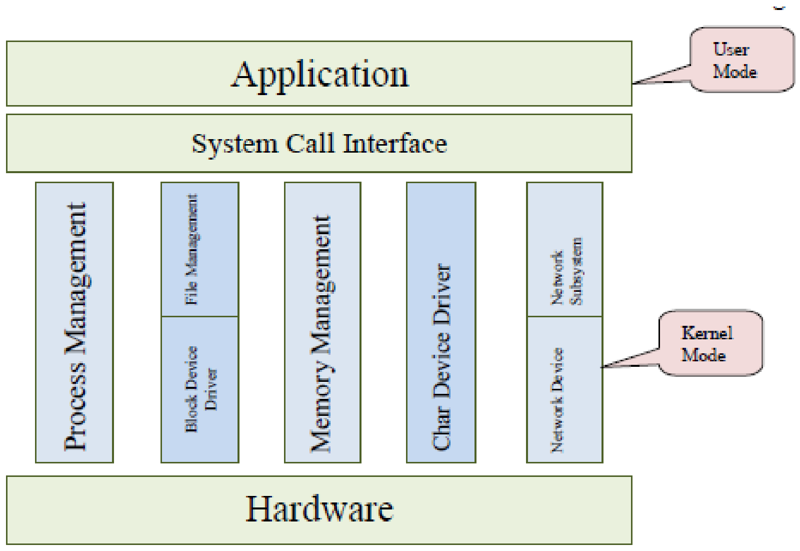

Figure E-1 Monolithic Kernel Architecture

Figure E-1 Monolithic Kernel Architecture Examples of such type of kernels are Linux, UNIX and Windows. GPOSs are time-sharing and designed to maximize the throughput and provide a fair system resource in a multi-user/multi-process environment. The monolithic GPOS design is not suitable for *hard real-time* systems, so we need a special purpose, differently designed, Real-Time Operating Systems (RTOS) mostly based on a micro kernel. In this arrangement the kernel houses the essential functions to access hardware like process scheduling, interrupt handling and intercrosses communications. The other kernel subsystems like memory management, file management and network management are dynamically loaded as and when required in user space. Figure A-2 shows the micro kernel architecture.

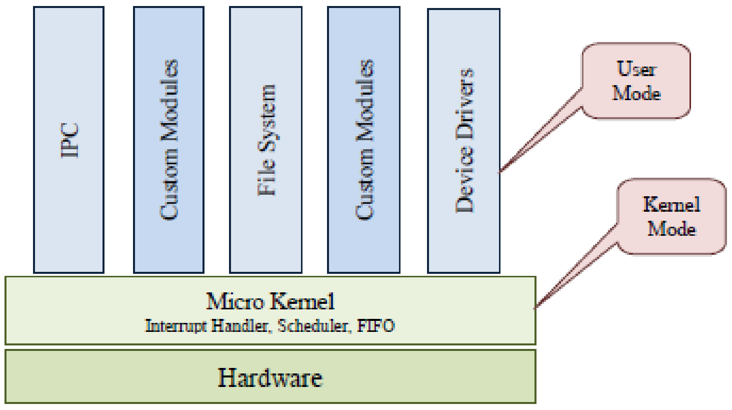

Figure E-2 Micro Kernel Architecture

Hybrid Kernel

Due to the increasing complexity of embedded devices, at times we need to execute both *hard real-time* applications and general-purpose applications on a single system.Figure A-3 Hybrid Kernel Architecture In such case, both GPOSs and RTOSs are unsuitable: with a GPOS, we cannot get predictable behavior for real-time applications, and with an RTOS, the general-purpose applications will considerably degrade system performance. To cater for such a contrasting requirement, we need a hybrid kernel shown in Figure A-3, which offers both real-time and general purpose features. Real-time applications execute directly in the real-time kernel, and when there are no real-time tasks running, then non real-time tasks are allowed to run. When a real-time task wants to access a certain feature that is not available in the micro-kernel, it exploits a feature from the monolithic kernel via a special communication mechanism using First in & First out (FIFO).

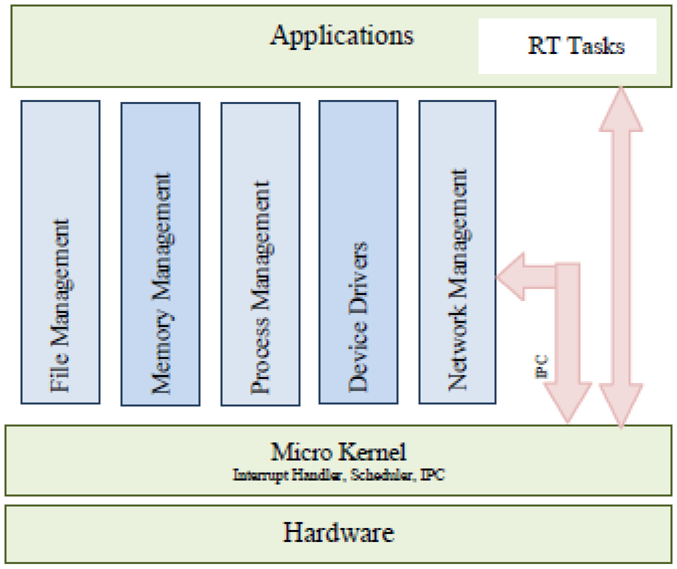

Figure E-3 Hybrid Kernel Architecture

Real-Time Application Interface (RTAI), Real-Time Linux (RT Linux) and Windows NT are examples of hybrid kernel-based operating system. Figure 4 depicts the structure of a hybrid kernel-based operating system.

References

[1] GMRT. Servo System. [Online]. Available: http://gmrt.ncra.tifr.res.in/gmrt_hpage/sub_system/servo/servo.html

[2] RTAI. Real Time Application Interface Documents [Online]. Available: https://www.rtai.org/

[3] GMRT Station Servo Computer Software Documentation [Online]. Available: http://tech1.gmrt.ncra.tifr.res.in/thiyagu/PC104/report/GMRTStationServoComputerSoftwareManual.pdf

[4] GMRT Servo System –Design Manual [Online]. Available: http://tech1.gmrt.ncra.tifr.res.in/thiyagu/SERVO/designmanual.pdf

[5] GMRT SSC to SCC Interface Details [Online]. Available: http://tech1.gmrt.ncra.tifr.res.in/thiyagu/SERVO/ssc2scc-interface.pdf

[6] GMRT Station Servo Computer – Micro Computer Boards Hardware Reference [Online]. Available:

[7] J. David Powell, Gene F.Franklin, Abbas Emami-Naeini. Feedback Control of Dynamic Systems, 6th Edition. Pearson Education (2006)