Module Interface Board - MIB

Framework Software

Version 1.1.0

Table of Contents

Revision History

Revision	***Date***	***Author(s)***	***Description of Changes***
1.0.0	November 11, 2003	Elwood C. Downey	Original version.
1.1.0	September 16, 2004	Pete Whiteis	Updated to reflect actual implementation of MIB Framework and change formatting. Implemented document revision number as document property "REVISIONLEVEL" which is accessed through the File->Property->Custom menu.

1 MIB Framework Software

1.1 Overview

All MIB software falls into three broad categories: Systems Software, the MIB Framework Software, and Module-Specific Software.

The Systems Software includes the Nucleus RTOS, the Nucleus NET network stack, and the Nucleus Shell.

Nucleus OS is a priority based, real-time, small memory OS marketed by Accelerated Technologies Inc. It is sold as a standalone OS with the option of adding on support packages for Network Protocols, File Systems, etc. It was selected in part, due to its small memory footprint, and because of ATI's experience with the TriCore architecture, of which the TC11IB is an example.

The MIB Framework Software was developed in order to provide a common Monitor and Control software platform that would facilitate rapid software development for a wide variety of hardware modules. Several requirements used in defining the Framework were:

1) SPI would be the primary means of communication with module electronics.
2) Ethernet would be the sole means of communication between the MIB and the outside world
3) MIB software would run entirely out of internal memory.
4) MIB software should support software upgrades over Ethernet.

The Framework software can be used standalone to provide M&C services for the simple hardware modules, and can be easily customized to support the needs of more complex modules.

1.2 Initialization

1.2.1 Bootstrap

When a MIB reset occurs, the MIB TC11IB processor will perform its first instruction fetch from address 0xA0000000, otherwise know as the 'reset vector'. This fetch will execute the boot loader located at this address. The boot loader will setup the critical TC11IB registers including those used for interrupt and trap handling, memory configuration, and the system timer interrupt. Next, the MIB image (containing both application and Nucleus OS) is copied from Flash to internal memory and executed, resulting in an initialization of the Nucleus OS.

1.2.2 Task Startup

After Nucleus initialization is complete, a user application startup routine named 'Application_Initialize' is called. The purpose of the 'Application_Initialize' routine is to define

the various Tasks that will be managed by the Nucleus OS. These tasks, their stack sizes, and priorities are defined in a table named 'Task_Startup_List'. Once 'Application_Initialize' completes, the Nucleus OS will begin scheduling these Tasks.

1.2.3 Peripheral Initialization

Two peripheral components critical to the operation of the MIB Framework software are the SPI controller, and Ethernet stack.

SPI initialization is implemented in a task named 'Ptsmon_Task'. When SPI initialization is successful, a software synchronization flag is released, which allows the 'netIF_Init_Task' to load the Ethernet driver, and initialize the TCP/IP stack.

When the Ethernet stack initializes it attempts to use an IP address obtained from a SPI EPROM device known as the slot-ID EPROM. The slot-ID EPROM is a memory chip that resides on the back of the antenna module racks, and serves as 1) a slot identifier for the rack, and 2) the source for the IP address of the modules in a rack. Network Gateway and DNS addresses are stored in the slot-ID EPROM as well. If the network startup routine detects a '0x0A' (first byte of our 10.xx.xx.xx network) in the first byte of the slot-ID EPROM, it assumes that the EPROM contains valid IP addresses and uses them to configure the TCP/IP stack with a static IP address. If the network startup routine does not detect a properly programmed slot-ID EPROM, it will use DHCP to obtain an IP address for the MIB. In either case, a Domain Name Server (DNS) will associate a name with the MIB's IP address. For MIB's that use DHCP, the name given will be in the form 'evla-mib-nn' where 'nn' is the last byte of the MIB's MAC address. MIB's using slot-ID base addressing (static), will have DNS entries that conform to those described in Wayne Koski's document titled 'EVLA Hardware Networking Specification', document number A23010N0003.

1.3 Memory Usage

The MIB software makes use of three different memory regions, each of which is distinct from the others, both physically and functionally.

EDRAM – Begins at address 0xC0000000 and contains 512Kbytes of storage. This region is internal to the TC11IB and is functionally dedicated to program code. The MIB program image executes entirely out of EDRAM in order to eliminate RFI activity normally associated with CPU fetches from external memory. The current MIB executable image size is approximately 225Kbytes, leaving 56% free space in EDRAM (size varies with module type).

ComDRAM – is located at address 0xBFE00000 and contains 1Mbyte memory region dedicated to data storage. Both static memory regions pre-allocated at link time and memory regions allocated dynamically during program execution reside in ComDRAM. The MIB application creates three different memory pools dedicated to dynamic memory needs. *NETMEM* is a 161Kbyte pool used exclusively for handling requests for Ethernet Transmit and Receive buffers. The *SYSMEM* pool dedicates 300Kbytes of dynamic memory for Nucleus and application requirements. *LOADMEM* is a 390Kbyte pool dedicated to loading new images and XML points configuration files. The LOADMEM pool size will support either a program image load of

390Kbytes, or an XML Points Configuration file load of up to 290 Logical Points.

Flash Memory - is located at 0xA0000000 and contains 8Mbytes of non-volatile memory. The MIB application uses a Table Of Contents scheme to assign flash memory sectors to different categories of code and data storage. Current flash sector assignments are defined in app_init.c and are as follows:

Flash Sector	Content
0-7	Boot Loader Image
8-15	Program Image
16-20	Spare sectors
21-23	XML Configuration file
24-31	Spare sectors
32	MAC address

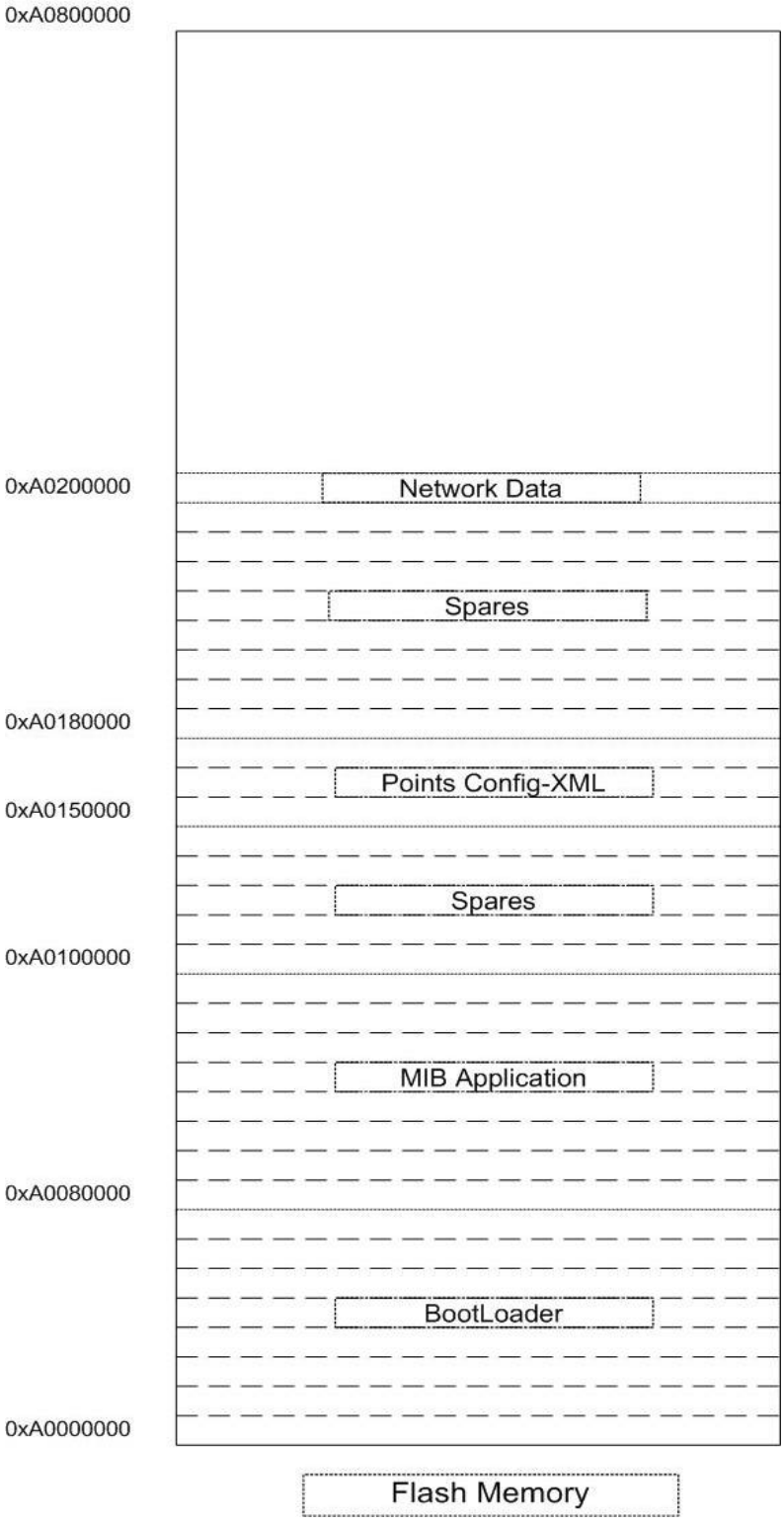

Figure 1 Map of flash memory usage

1.4 Data Structures (Logical Points)

From the point of view of a client connecting to the MIB, all functionality is cast in terms of Monitor and Control points, referred to collectively as Logical Points. The term Logical also refers to the fact that the specific module to which the MIB is connected also uses a notion of physical Monitor and Control points that are not necessarily mapped to those points visible outside the MIB.

The MIB always connects physically to only one module at a time. However, some modules can best be thought of as consisting of distinct functions. The MIB presents these functions as Devices and associates a set of Logical Points with exactly one Device. No Logical Point is ever associated with more than one Device within a module. The MIB also presents itself as a device, whose device name is "MIB".

Monitor Points are basically read-only. Their value reflects some detail of module state. Control Points are basically write-only. Setting a Control Point to a value causes something to happen.

Monitor and Control Points come in two flavors: Analog and Digital. Analog Monitor points have a floating-point value (double width) and include attributes that facilitate scaling raw data to engineering units, data logging intervals, and alert detection. Digital Monitor points have a unary value, and contain an attribute that specifies either high or low transition alerts.

All Monitor and Control Points are internally represented by a global 'C' language data structure named 'Logical_Points'. Any MIB application that wishes to access information in the Logical_Points database simply maps to it and reads from or writes to the attributes of interest.

Initialization of the 'Logical_Points' database occurs when the MIB application initializes and parses an XML-based points configuration file, usually known as logical_points.xml. This configuration file is stored in MIB flash memory and can be changed and reloaded during runtime. Rebooting the MIB will cause the points configuration change to take effect. Detailed information covering the format of the points configuration file is presented in the document titled 'XML_file_format_description.txt' (see Appendix 3).

1.5 Points Monitoring

The Monitor Points within the 'Logical_Points' database are periodically updated in order to provide monitor data and alerts in a timely manner. The source of a Monitor Point is usually (but not limited to) a hardware device that is polled for raw data, converted to engineering units, and stored in 'Logical_Points'.

The details about a monitored hardware device, its access mechanism (e.g. SPI, GPIO, etc), and monitor rate are contained within a data structure named 'Raw_Monitor_Points'. The 'Raw_Monitor_Points' database exists as a data entity separate from the 'Logical_Points' database in order to hide the quirks of the hardware interface from the remainder of the system. This database is initialized by a table of definitions within the ptsmon_usr_init.c file. Unlike data in 'Logical_Points', the data within Raw_Monitor_Points is fixed at compilation time.

The majority of the implementation for points monitoring is contained in ptsmon.c, within a task named 'Ptsmon_Task'. The 'Ptsmon_Task' is run off a 10Hz system timer and periodically

executes the following sequence:

- Find next point defined in 'Raw_Monitor_Points'. Read the corresponding hardware device
- Unpack raw data into one or more (up to 16) Logical Points.
- Convert to engineering units
- Flag any out-of-band data.
- Log data to archive

The 'Logical_Points' database contains the attributes used for the data conversion, alert detection, and data logging steps. These attributes are accessible through the service port, and may be modified at runtime.

1.6 Alert Handling

Alerts are defined as any monitored, out-of-range data that would be of interest to the user. The alert mechanism differs somewhat between Analog Monitor and Digital Monitors. Typically, an Analog Monitor point will generate an alert when its value goes above or below a limit for a specified period of time. Digital Monitor points will generate alerts when their value diverges from their normal value as given in one of their attributes. In either case, when an alert condition is present, or cleared, an asynchronous message is sent over the network indicating alert status of the point, along with its value. The message for in alert and alert cleared is generated exactly once.

Attributes within the 'Logical_Points' database can enable or disable alerts for a given monitor point, as well as establish the conditions under which an alert occurs. These attributes are initialized from the XML points configuration file and can be modified during runtime as well.

See the document "MIB Data Port ICD" for more information on alert detection and message generation.

1.7 Time

This section will discuss the use of time in the MIB from both the operating system perspective and the MIB Framework's perspective.

1.7.1 Nucleus System Timer

The Nucleus OS sets up the first timer (T0) on the first of the two General Purpose Timer Units (GPTUs) in the TC11IB processor for its OS timer. This timer is set up to generate an interrupt every 10ms (e.g., it runs at 100Hz). The interrupt is then processed by the OS' timer handling routines in order to perform all operating system functions that require time – this includes all of the Nucleus OS functions that have a timeout parameter as well as user-defined timers. This timer is purely an interval timer, and runs off the system's main oscillator at 48Mhz; as such it

will tend to drift.

1.7.2 Wall Clock Time

The Nucleus OS does not provide any wall clock time tracking. The OS clock is used purely as an interval timer and can drift.

The TC11IB provides another timer called the system timer which is a free running 56 bit counter that increments at the speed of the main oscillator (48Mhz). The clock can track time for up to 47 years between reboots of the TC11IB, but at every reboot of the TC11IB the system clock starts at 0 and increments up from there. (See the TC11IB documentation from Infineon for more information). This timer will not provide wall clock time by itself, but since it is not subject to interruption it does provide a good delta time from the time the MIB rebooted.

The MIB Framework implements the facility required to track wall clock time. The routines found in timeStamp.c provide the rest of the Framework code and the module-specific code the ability to get the current wall clock time in Modified Julian Date (MJD) format, as well as manage the wall clock time facility and get some useful information about it for the Service Port to display (see the MIB Device Points section of the MIB's Service Port ICD for more information on those points). The system timer is accessed via a routine in this same module so that it will be presented in units of days instead of clock ticks; this hides the hardware implementation of the system timer from the rest of the code, and allows the resulting value to be added or subtracted from the MJD values used in the rest of the code.

NOTE

Whenever the value of the system timer is referenced in this section, the actual value being used is the value in units of days returned by this function.

The MIB's wall clock time is a double float value representing the number of days since November 17, 1858 at 00:00:00.00 Greenwich Mean Time (GMT). It's called a Modified Julian Date because that time is precisely 2,400,000.5 days since the start of the Julian calendar at noon on January 1, 4713 BC. MJD is used primarily in astronomy.

The Julian calendar was proposed by J. J. Scaliger in 1583, so the name for this system derived from Julius Scaliger not Julius Caesar. Scaliger defined Day One as a day when three calendrical cycles converged. The first cycle was the 28-year period over which the Julian calendar repeats days of the week (the so-called solar number). After 28 years, all the dates fall on the same days of the week, so one need only buy 28 calendars. (Note that since the Gregorian calendar was adopted the calendar now takes 400 years to repeat.) The second was the 19 year golden number cycle over which phases of the moon almost land on the same dates of the year. The third cycle was the 15-year ancient Roman tax cycle of Emperor Constantine (the so-called indiction). Scaliger picked January 1, 4713 BC on the Julian calendar (of Julius Caesar) as Day One. (Definition of the Julian Date courtesy Eric W. Weisstein's astronomyWeb page at http://scienceworld.wolfram.com/astronomy/JulianDate.html).

The MIB determines what the actual wall clock time is by using the Network Time Protocol (NTP) to get a wall clock time value to provide a base time. It then can use the elapsed clock

ticks represented by the TC11IB's system timer as an offset from that time. The code module time_client.c contains the software and a task routine that performs the initial time request (discussed below) and the handling of NTP messages. That code will use routines in the timeStamp.c module to manage the base time for the time stamping function.

NOTE

Until the MIB receives the base wall clock time from a NTP message, it will always return a time value of 0.0 as the current time.

1.7.2.1 Network Time Protocol

This protocol, also called NTP for short, is used on the Internet to provide reliable wall clock time to systems that require it. The originator of the NTP messages is some server or other device that has a highly accurate clock, usually synchronized to some highly accurate time source such as the Global Positioning System (GPS) or a maser. This server runs the NTP server software that allows it to send NTP messages to devices connected to the network that request them, as well as periodically (approximately every 64 seconds) multicasting the current time over the NTP multicast group ntp.mcast.net.

1.7.2.2 MIB Procedure for Acquiring Initial Time

The MIB has no idea of the current wall clock time when it first boots. In order to obtain the current wall clock time, the Framework code will start a task called "time_client" which is in the time_client.c module. That task's first action will be to try to resolve the MIB's own TCP/IP name by using reverse name lookup and the MIB's assigned IP address. If this completes, it will use the network portion of the returned name, and replace the MIB's name with "timehost", which is the name of the time server in the NRAO networks. This procedure will handle a MIB at the AOC or at the VLA site. In any other location a MIB would have to supply a DNS entry in order to have its initial time request sent. Should the attempt at getting the MIB's TCP/IP name fail, the task will retry the operation in 5 seconds, and will continue to retry the operation until it completes.

Once the MIB has the node name constructed it will use it to obtain the TCP/IP address of the time server by using the DNS function NU_Get_Host_By_Name(). If this step does not complete, the task will again loop over the entire process after waiting 5 seconds.

If it does complete the task will set up its socket structures so it can receive UDP messages, and join the NTP multicast group so that it can receive NTP multicast messages. It will then send a NTP time request using a UDP socket to the "timehost" server whose name it resolved above.

At this point, the task will wait on the NTP incoming message socket and a timeout for 10 seconds. If no NTP message is received in 10 seconds, the task will once again send out the NTP time request to the "timehost" server. However, if the task should receive a NTP time message (either a multicast message or the response to its request), it will process that message and no longer wait on the socket plus a 10 second timeout. Instead it will just wait on the socket

for more NTP messages.

Once an NTP message is received, the task will take the NTP time in the message and convert it to a MJD time value. It will then call a routine in the timeStamp.c module to store the new time into an 4 position array of received times; that routine will scan the existing entries and this new one and determine which one more closely approximates the time that the MIB rebooted. It does this by taking the time values and subtracting the value of the MIB's system timer at the moment that time message was received; whichever time entry produces the smallest value is the one picked for the base time of the system, and the approximated MIB reboot time is stored as the base time for the system (see the discussion on the procedure for calculating time values).

The 4 positions in the array are overwritten, oldest entry first, once the array is full.

1.7.2.3 Heartbeat Timing Interrupt

The MIB hardware and Framework software for time supports an optional external interrupt which is used to provide time synchronization in the antenna array. Currently this is the 19.2Hz interrupt that the VLA uses; it can be different from that, as long as it is in units of 0.1 Hz. Some MIBs may get a different frequency pulse (such as a 1 PPS signal in the L353); others will not have any heartbeat pulse at all.

The MIB Framework always sets up the heartbeat timer ISR and the corresponding support structures to insure the heartbeat interrupt will function if it is present. This interrupt will also drive a Nucleus event flag so that any module-specific or Framework task that requires synchronization to the heartbeat can block on the event flag until the heartbeat occurs. This event flag is set and reset within the HISR routine that handles the heartbeat interrupt.

The Framework code will store the wall clock time of the heartbeat interrupt for use by the time stamping function. This wall clock time is an MJD value, initially 0.0, which the MIB will determine by taking the approximated reboot base time (see above) it is using and add the value of the system timer at the moment the interrupt occurred. It will then subtract off the days value to give the fraction of a day which should have elapsed, and calculate the integral number of interrupts that should have happened using the interrupt rate in order to get to that time, rounding the result up if required. Then, a new approximated reboot time is calculated by adding that number of interrupt times to the day value, and subtracting the value of the system timer at the moment the interrupt occurred.

Note that this can only be done once the MIB has determined the heartbeat interrupt rate; until then this calculation is skipped. The currently used heartbeat rate is shown in the heartRate monitor point of the MIB device itself.

Also, if the heartbeat rate calculation does not agree with the previously calculated value that is being used, an error counter is incremented; if that error count gets to 10, the current heartbeat rate is no longer used and the MIB will recalculate the rate. A count of such recalculations is

kept in the heartReset monitor point of the MIB device itself.

1.7.2.4 MIB Calculation of Wall Clock Time

As stated previously, until the MIB receives the wall clock time for its base time from an NTP message it will always report the time as being 0.0. Once that base time is received, the MIB will report the time by performing the following steps.

First, the code determines if there is a system heartbeat interrupt present. It does this by looking at the heartbeat interrupt interval; should that be 0 there is no heartbeat coming into the MIB or its rate is not known at present, and thus the approximate reboot time as calculated by the NTP time handling routine is used as the base time. If that rate is not equal to 0, then the approximate reboot time as calculated in the heartbeat interrupt handling (see above) is used as the base time.

Second, the current value of the system timer is captured, and added to the base time selected in the first step by converting it to a value in units of days. Since the system timer is incrementing at the rate of 48 million per second, this means it must be divided by 48 million and then multiplied by 86400 seconds/day.

Lastly, the resulting time is returned to the caller as a MJD value.

1.7.3 High Resolution Timer Support

The MIB Framework provides a high-resolution elapsed time facility, which can handle elapsed times of a very short duration (currently, 50us or more). This capability was implemented because the Nucleus OS timer granularity of 10ms is much too long for some applications.

This facility makes use of the rest of the timers contained in the TC11IB's GPTU 0 and GPTU 1 modules. If you recall, Nucleus itself is using the first timer (T0) in GPTU 0 for the OS timer. This leaves two timers (T1 and T2) in GPTU 0 and three (T0 to T2) in GPTU 1. All of these timers are used by the high-resolution timer facility; and GPTU 1 T0 is split into two timers of 16 bits each.

The facility's timeouts are expressed as unsigned 32 bit integer values representing the number of nanoseconds of delay desired. This means that the maximum delay is about 4.2 seconds. The value will be adjusted to the minimum high-resolution timer value of 50us if it is lower than that when the routine is called to get a timer. No error status will be returned in that case; the elapsed time value is simply changed. The elapsed time values are in turn converted into a number of clock ticks required to represent them, rounding normally. The clock tick is the system clock at 48Mhz; there is a hardware bug in the GPTU modules that prevents the use of any slower clock tick.

Two methods of elapsed timing are supported by this facility. The first is an implementation of a sleep routine which can be called like NU_Sleep() in order for a task to do sub-10ms delays. The function will find an available high-resolution timer, configure it with the given timeout,

start it, and then block the calling task on the event flag for that timer. Once the timer expires and the task again gets the CPU, the routine will stop the timer and release it back to the timer pool.

The other method of using the elapsed timers is to call the timer allocation routine to get a timer, then set that timer up and start it. Once started, any number of tasks can wait on that timer's event flag for the next timer expiration event by using the supplied high resolution timer event routine, and perform whatever actions are required by the application. The timer will continue to run and set/reset the event flags until it is stopped.

In addition to the event flag, the facility also supports a callback routine that can be assigned to a timer. This routine is called at the ISR level when the timer expires; as such great care must be taken by the application developer to avoid using Nucleus functions that do not work at ISR level (see the Nucleus documentation's Interrupts section for a discussion of that).

1.8 Software Upgrades

The remote location of MIBs has led to the need for a remote software upgrade capability in the MIB framework software. This capability allows one to use a simple workstation client to download both program code and point configuration files to a MIB from any workstation on the EVLA network. The MIB application that receives code/data files was written to ensure the amount of time spent erasing and reprogramming flash was minimal. When a code/data load is initiated, the entire download file is read into RAM over a TCP/IP socket and validated prior to erasing and reprogramming of Flash Memory sectors, thus reducing the risk of having a null-image MIB resulting from power or network dropouts.

The procedure for upgrading the MIB's point configuration file is as follows:

- The point configuration file must be in XML format and conform to the description given in the document 'XML_file_format_description.txt'.
- The XML configuration file load must be done on a Windows laptop from within the Cygwin environment.
- Invoke the XML configuration file loader client by the command: NRAO/EVLA/MIB/Tools/netloadflash/netloadflash [MIB IP or DNS name] filename
- The netloadflash client will Telnet into the MIB and issue the command 'set MIB.xmlLoader=1'. This releases the internal file upgrade lockout to allow the XML configuration file to be accepted by the XML loader server on the MIB.
- Progress can be monitored by the MIB command 'get MIB.xmlloader.msg'. Under normal, successful loading conditions, the MIB will reply back with "xmlloader-I-accepting socket", "xmlloader-I-reading socket for XML data", "xmlloader-I-done reading, shutdown server", "xmlloader-I-flash prep", "xmlloader-I-burn xml", "xmlloader-I-finished", and "xmlloader-I-waiting to start". Typically, the messages change so fast that only the "xmlloader-I-finished" message is seen; it will stay there for

20 seconds before the "xmlloader-I-waiting to start" message is displayed. Any other response will indicate a problem with the loading sequence.

The procedure for upgrading new code is as follows:

- For program loads, the file must be in Motorola S-Record file format, which is generated as a result of the software build.
- Invoke the loader client by the command: NRAO/EVLA/MIB/Tools/netloadflash/netloadflash [MIB IP or DNS name] filename [-b(for program code)]. When code loading, the client may appear to hang for about 20-30 seconds. This is normal due to the fact a 1MB+ file is being network loaded to the MIB.
- Progress can be monitored by the MIB command 'get MIB.codeloader.msg'. The MIB will reply back with 'SREC-I-Saving Code', 'SREC-I-Load Complete', or 'SREC-I-OK'. Any other response will indicate a problem with the loading sequence.

1.9 Commands

Commands can be sent to the MIB by way of Telnet or UDP messages to a logical I/O channel known as the 'service port'. The Telnet interface to the service port is intended for human interaction with the MIB, and therefore contains a simple command menu, and limited command recall capability. The number of simultaneous Telnet clients supported by a MIB is limited to two in order to prevent context block exhaustion. The UPD interface is intended for machine to MIB interaction, and can support up to 3 simultaneous clients. Any clients that are idle for a period of more than 1 hour are disconnected in order to prevent stacking of server tasks that would lead to resource exhaustion.

The two basic commands supported by the MIB are `Get` and `Set`. The `Get` command basically scans the `Logical_Points` array for entries that match a query pattern and forms a response of current values in an XML format. `Set` performs a similar scan but can set new values for logical points.

An exhaustive discussion of the service port, its use, and constraints can be found in the document MIB Service Port ICD.

1.10 Data Logging

The data port broadcasts UDP packets of MIB Monitor Points. These are broken into the three categories called Alert, Archive, and Screen. Packets are transmitted on an event basis for the Alerts and on a periodic basis for the others. The period can be set separately for each category and each MP.

The format of the Data port packets is XML. It is the same as the format of the responses from the Service Port.

See the MIB Data Port ICD document for more details on the Data Port and data logging.

1.11 Reliability

To ensure high availability, the MIB's must have a means of recovering from catastrophic software failure. These failures are defined here as trapping a software exception (invalid memory access, divide by zero, etc), or falling into a hard CPU loop which would prevent operation of the OS scheduler.

In order to recover from these failures, the MIB Framework makes use of the TC11IB's watchdog timer facility. The watchdog timer is clocked by the 48MHz system clock and will generate 1) a Non-Maskable Interrupt (NMI) and 2) a Watchdog Reset, if not serviced within a timeout period, defined as 5 times the Heartbeat interval (currently 1 second).

Servicing the watchdog timer is done within a low priority task known as the Heartbeat Task. The Heartbeat Task executes at a 1 Hz interval, blinking the MIB Test LED and servicing the watchdog timer. Any processing loops that prevent the Heartbeat Task from running for the duration of a watchdog timeout period will result in a MIB reboot.

In order to trace the root cause of a MIB reset, four counters have been implemented as Monitor Points within the MIB device. These counters keep track of the number of times a Watchdog, Hardware, Software, or Power-On reset has occurred. The counters are updated by reading the Reset Status register on application startup, incrementing the appropriate counter, and saving its value to Non-Volatile EPROM. The reset counters can be cleared at any time by setting the MIB control point 'ClrResetCnts' to 1.

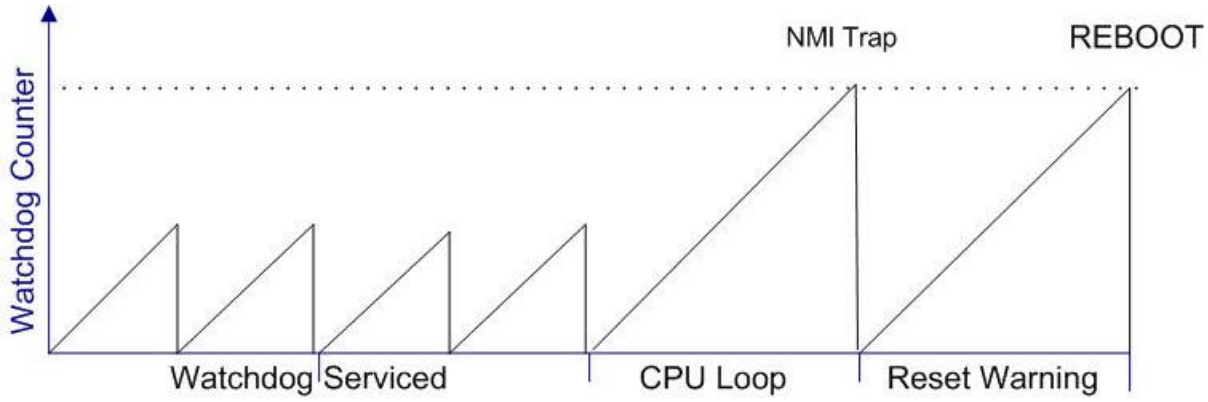

Figure 2

2 Module Specific Software

Software specific to a hardware module is segregated from the MIB Framework software by location in a directory named after the module (e.g. NRAO/EVLA/MIB/src/L304). At a minimum, there are three 'C' language files that implement module specific logic. Init_Devices.c, contains calls to initialize any hardware devices not used by the MIB framework software. Ptsmon_Usr_Init.c contains a table named Raw_Monitor_Points that defines the module specific hardware devices to be monitored during runtime. Module_Specific_Routines.c define application software 'plug-ins' that are invoked from the MIB Framework software as part of its execution loop. These 'plug-ins' include initialization routines, version identification,

debug routines, and IO routines, for a given module.

A build in a module directory will invoke the module specific Makefile, and subsequently, the MIB Framework Makefile. The binaries generated contain the name of their module source directory, and are copied to the network directory: /home/software/evla/MIB/bin, for public access.

Appendix 1 Development Environment

Appendix 1.1 General

The MIB development environment consists of a Windows 2000 laptop with the HiTOP embedded device debugger hardware and software system and the Tasking C/C++ cross compiler tool chain by Altium. We have elected to use the Cygwin UNIX-like environment for the editing and build process. The MIB processor is an Infineon TC11B. The real-time embedded operating system is Nucleus by Accelerated Technology.

Appendix 1.2 MIB Setup

Check the SW1 boot switches on the bottom of the MIB and make sure they are set as follows: 1-On 2-On 3-Off 4-Off. This ensures the MIB will attempt a power-on bootstrap from external RAM.

If using the auxiliary development baseboard, connect the MIB to the top of the board and connect an RS232 cable between the DB9 labeled ASC and COM1 on laptop. The communication parameters are 57600 baud 8-N-1. Window's HyperTerminal is known to work with this configuration.

Connect the fiber Ethernet to the MIB and connect power from a 5v supply. Once the MIB is running, it is accessible on the internal aoc.nrao.edu LAN with DNS name evla-mib-N where N is the last hex digit of the MIB MAC address (dropping any leading zeroes). The MIB MAC address is printed on the fiber connector. For example, a MIB with a MAC address of 00-0C-69-00-02 would have a DNS name of "evla-mib-2".

Connect the HiTEX emulator interface to the MIB using ribbon cable to J2 (match pins-1). Connect Ethernet and power.

Appendix 1.3 Windows Laptop Setup

Note: To perform the next steps, you will need to be a local Administrator for the laptop. Check with the Desktop group.

On the laptop, go to cygwin.com, download and run their setup file, connect to nasa.gov, download All-Install to d:\temp, run setup again to install to d:\cygwin.

Create directory `d:\cyghome`

Create file `d:\cyghome\cygwin.bat` containing the following :

```
if not exist z: net use z: \\filehost\evla
d:\cygwin\bin\bash --login -c 'HOME=/cygdrive/d/cyghome; cd;
exec csh -c "exec startx"'
```

Create file d:\cyghome\.xinitrc (note leading dot) containing:

```
xsetroot -solid '#225'
xterm -geometry +10+10 -e /bin/csh &
xterm -geometry +800+10 -e /bin/csh &
twm &
exec xclock -geometry 70x70-1+1
```

Note: this is set up using the 'C shell'. Replace '/bin/csh' with '/bin/bash' to use the Bash shell instead, and copy the `.bashrc` file from the "asg" share on "filehost", in folder `evla/Dev_Shell_scripts` to `d:\cyghome` to get command shortcuts' and environment variables' definitions.

If not copied above, create the file `d:\cyghome\.bashrc` (note leading dot) containing:

```
setenv CVSROOT /cygdrive/h/cvsroot
setenv XUSERFILESEARCHPATH ~/APPLRESDIR
```

Create directory `d:\cyghome\APPLRESDIR`

Create file `d:\cyghome\APPLRESDIR \XTerm` containing:

```
XTerm*background:  #ff9
XTerm*foreground:  black
XTerm*ScrollBar:   True
XTerm*SaveLines:   1000
XTerm*geometry:    100x70
XTerm*VT100*font:  7x13
```

On the Windows desktop, right-click the Cygwin icon, click Properties, set Target to d:\cyghome\cygwin.bat, set Run to minimized, click Ok. Now double-click the Cygwin icon to start cygwin. The X Window server should start full-screen with two xterm windows and a clock applications running.

Note: Sometimes Cygwin does not install cleanly. If you have problems with missing DLLs, check for them at: filehost://asg.evla.Distributions.Cygwin_missing_DLLs

Appendix 1.4 HiTOP Debugger Setup

Start HiTOP from its desktop icon. Very briefly, the HiTOP embedded tool can load an image to

the MIB, control execution and inspect memory. File->Load allows downloading an image. F9 will start execution. Shift-F9 will halt execution. F11 resets the MIB. See the Go menu for various methods to single-step execution using F6, F7 and F8. View->HLL displays the current stack trace. View->Memory allows displaying any arbitrary memory block. Local->Examine allows the viewing of a selected variable, but for local stack variables (including routine parameters) be aware that the compiler may have optimized the use of registers in such a way that the debugger will not always display the correct data for that type of variable.

To setup the TC11IB processor using HiTop, go to menu option Setup->Processor and choose the EBU Config 1 Pane. For EBUCON use 0x10068. Choose EBU Config 2 Pane. Enter the following parameters:

BUSCON0 0x420000 ADDSEL0 0xA0000041 BUSAP0 0xFFFFFFFF

BUSCON2 0x820000 ADDSEL2 0xA08000F1 BUSAP2 0xFFFFFFFF

Choose EBU Flash Programming Pane. Check 'Enable Flash Programming'. Enter 0xA0000000 for 'Flash Base Address'.

Appendix 1.5 Software organization

All MIB software falls into two broad categories: the Nucleus operating system and the NRAO application. Nucleus is in VENDOR/ATI/Nucleus_TC11IB. Source is included but need not be used in routine development because all Nucleus functions are already compiled into libraries.

The application code is further divided into a framework expected to be the same on all MIBs and one directory for the software pertaining to each module to which the MIB may be attached. The framework code is in `NRAO/EVLA/MIB/src/mib_app`. The module directories are in `NRAO/EVLA/MIB/src/<module-name>`. For example, the software specific to an L301/2 module is in `NRAO/EVLA/MIB/src/L30x`.

Appendix 1.6 Software development

Initially, a MIB software developer will need to create a directory structure and populate it with Nucleus and MIB source files.

Click in an xterm, create and cd into a directory named CVS_Projects, then check out the entire MIB software from the network repository as follows:

```
$ mkdir CVS_Projects
$ cd CVS_Projects
$ cvs co NRAO
```

Next, check out the VENDOR CVS Project as follows:

```
$ cvs co VENDOR
```

At this point, the directory structure and content has been established for MIB code development.

There is a `Makefile` in each module directory. Typing just `make` will reach over to other

directories and insure their libraries are up to date, compile the software for the particular module, link everything into a load image and create symbol tables suitable for downloading and executing by the HiTOP development tool. All modules put their load images and associated files in \\filehost\evla\software\MIB\bin with names beginning with the name of the module (e.g. L30x_edram.sre). The Makefiles expect the \\filehost\evla network directory to be mapped (on Windows) to the z: drive.

Appendix 2 Troubleshooting

Known Tasking Compiler Bugs

```
[PROB1]
      Title: Bug in test code generation
      Description:
      The compiler will not generate the proper code for a test of
the form:
            x = a?b:c

      where: 'a' is some constant expression, and 'b' and 'c' are
values.

      The compiler will generate code to extract bits from 'a'
instead of doing a simple test and branch, and will always pick the
true clause 'b' no matter the value of 'a'.


[PROB2]
      Title:      Bug in array address arithmetic
      Description:
      The compiler will not correctly calculate the number of array
elements between two address locations in the array under some
conditions which are not clearly understood.  For example:
             int array[10];
             int *p=&array[5];
             int b=p-array;

             'b' should be the number of elements in the array
between the address pointed to by 'p' and the array's beginning address.

       The compiler will sometimes generate code that will produce a result
of 0, apparently by dividing the difference in addresses by a huge
number.  The exact cause of this problem is not known since it
occurred in a complex block of code, and the problem disappeared once
the complexity was reduced.


[PROB3]
      Title:      Syntax error for initialization of arrays of structures
      Description:
```

```
      The compiler will issue a warning message on the syntax:

          typedef struct {
                int a,b,c
                } samplestruct;

          samplestruct x[]={ {1,2,3}, {1,2,3}, };

      The problem is the last comma "," before the closing curly
brace.  That syntax is allowed in ANSI C, but the compiler does not
want to see that comma there.


[PROB4]
      Title:      No vsnprintf() function implemented in ANSI C library
      Description:
      This function is not available.  vsprintf() is.
```

Appendix 3 XML File Format Description

Appendix 3.1 Introduction

```
---------------
This file describes the XML file format (usually named "logical_pts.xml")
used by the EVLA's MIB framework software to describe and set up the data
points used to monitor and control the EVLA module hardware.
```

Appendix 3.2 Basic File Structure

```
---------------
The XML file is a simplified hierarchical description of the major hardware
components of a module and the monitor and control points associated with
those components.

The file has an outer level description that identifies it as a MIB XML file.
This outer level has the form:

<Logical_Pts>
      .
      . (hardware component definitions listed here)
      .
</Logical_Pts>


The module will have 1 or more hardware components, each component referred
to as a "device".  There is no arbitrarily imposed upper limit on the number
of devices.  Typically, there are no more than 2 devices per module; each is
listed one after the other inside the outer level definition.
```

```
Device Entry Format
-------------------
A device entry has the form:

   <device name="devicename">
         .
         . (logical data point entries listed here)
         .
   </device>
```

The device name is an attribute of the device and is any alphanumeric string with the addition of the underscore ("_") character, that is no longer than 23 characters. The name may be upper or lower case, but typically searches on the device name are not case-sensitive. The entire name is enclosed in double quotes as shown in the example.

```
Logical Data Point Entry Format
-------------------------------
```

Inside each device entry are lists of logical point entries, which can be either "monitor" or "control" type entries. These entries are referred to as "properties".

A property entry will have one of the following forms:

```
         <monitor ... />
```

or

```
         <control ... />
```

where the ellipses are replaced by a list of fields, called "attributes", which describe the monitor or control data point entry more fully.

```
Attribute Format
----------------
```

Each attribute is composed of the name of the attribute followed by the value for that attribute. The format is:

```
        attributename="value"
```

where the "attributename" is the name of an attribute for that property, and the value is any valid string representing the value for that attribute. Attribute values are generally numbers, such as floating point or integer values, but can also be strings and enumerations, which are represented by their string equivalents. The name attribute associated with a device entry is an example of a string-valued attribute.

Each attribute is separated from the ones around it and from the property type by 1 or more space or tab characters.

Attribute Descriptions

This is a description of all of the currently available and defined attributes that are recognized by the MIB framework software. Some of the attributes are only used by module-specific software, and their values (but not value type) and use are defined by that software. Thus, different modules may use the same attributes for different things.

Each property type has its own set of attributes, some of which may be the same. In addition, each property is further qualified by a "type" attribute that will select the list of attributes available for that type. (See the Service Port ICD for additional information on attributes.)

Required Property Attributes

All properties must have the common attributes "name" and "type" (see below). All other attributes are optional.

Missing Optional Attributes

If an attribute is not present in an XML file, that attribute will default to a value of 0, which depending on the attribute's type can mean different things; enumeration strings generally will default to the first enumeration value present in the list, strings will be null, etc.

Common Property Attributes

Both "control" and "monitor" properties have the following attributes.

AttributeName	Type	Range	Description
name	string	1 to 23 char	Alphanumeric name, plus "_" char
type	enumstring	analog or digital	Type of property classification Analog includes any multibit quantity, digital is only a 0 or 1 value.
a_period	integer	0 to maxint	archive rate, in 1/10th second increments (600=1 minute)
s_period	integer	0 to maxint	screen update rate, in 1/10th second increments
o_period	integer	0 to maxint	observe layer rate, in 1/10th second increments
aa_period	integer	0 to maxint	alert archive rate, replaces archive rate if pt is in alert
msg	string	0 to 47 char	Any string value (double quotes can be in the string but delimiters must then be single quotes)

Monitor Property Attributes

Common Attributes

There are no common attributes specifically for the monitor property.

Analog Monitor Attributes

AttributeName	Type	Range	Description
value	double	any double	initial value of the point
target	float	varies	value desired (for control loops)
conv_type	enumstring	see desc	method to get from raw counts to engineering value (see below)
slope	float	any float	used in the LINEAR and SIGNED_LINEAR conversion formulae
intercept	float	any float	used in the LINEAR and SIGNED_LINEAR conversion formula as an offset, + or -.
engr_unit	enumstring	various	the "bitfield" type will produce a hexadecimal output for any double float attribute.
lo_alert_flg	boolean	0 or 1	1 enables alerts on low values
min	float	any float	minimum value below which the point will go into low alert
hi_alert_flg	boolean	0 or 1	1 enables alerts on high values
max	float	any float	maximum value above which the point will go into high alert
alert_count	integer	0 to maxint	consecutive cycles with value under the "min" or over the "max" before alert state entered; also consecutive cycles with value over or at the "min" or under or at the "max" before alert state exited;

The possible values for conv_type are:

SIGNED_LINEAR - packed 12-bit signed value linearly converted
POLYNOMIAL - module software defined polynomial function
LINEAR - rawcount*slope+intercept = engineering value
NO_CONVERT - rawcount is used directly.

Digital Monitor Attributes

AttributeName	Type	Range	Description
value	boolean	0 or 1	initial value of the point
alert_arm	boolean	0 or 1	1 enables ability of point to go into alert state.
alert_on1	boolean	0 or 1	normal value of the point, different values cause alert (note no counter is used)

Control Property Attributes

Common Attributes

AttributeName	Type	Range	Description
dev_type	enumstring	various	communications interface used to send the command to the hardware

Analog Control Attributes

AttributeName	Type	Range	Description
value	double	any double	initial value of the point
conv_type	enumstring	see descr	method to get from raw counts to engineering value (see below)
slope	float	any float	used in the LINEAR and SIGNED_LINEAR conversion formulae
intercept	float	any float	used in the LINEAR and SIGNED_LINEAR conversion formula as an offset, + or -.
min	float	any float	lower range limit of engineering value, lower values rejected.
max	float	any float	upper range limit of engineering value, higher values rejected.
step	float	any float	for controlled outputs, how much to increase/decrease value (implemented on a per module basis)
engr_unit	enumstring	various	as above for analog monitor points
p0	double	any double	term for polynomial
p1	double	any double	term for polynomial
p2	double	any double	term for polynomial
p3	double	any double	term for polynomial

AttributeName	Type	Range	Description
p4	double	any double	term for polynomial
p5	double	any double	term for polynomial
p6	double	any double	term for polynomial
p7	double	any double	term for polynomial

The possible values for conv_type are:
SIGNED_LINEAR - packed 12 bit signed value linearly converted
POLYNOMIAL - module software defined polynomial function
LINEAR - rawcount*slope+intercept = engineering value
NO_CONVERT - rawcount is used directly.

Digital Control Attributes

AttributeName	Type	Range	Description
value	boolean	0 or 1	initial value of the point