Design of the Antenna Control Software

G. R. Harp
May 2, 2002

This document gives an outline of the software architecture and design for the Ajile card at the ATA antenna. We begin by describing the antenna control system from the point of view of the application programmer. We describe the capabilities and constraints of the Ajile card (main controller) in the context of what the antenna must do. Next, we outline the antenna software, describing the main classes and how they interact with one another. Finally, a brief description of the host “AntennaServer” application is given, with special attention to how it interacts with the Antenna.

1. Introduction

As background, we review the hardware comprising the antenna control system. Each antenna has its own (powerful) processor, so that software commands to the antenna may occur at a relatively high level of abstraction. For example, with a single command, an antenna can be directed to follow a complicated pointing trajectory. All monitor/command communication between ATA control system and the antenna occurs through an Ethernet port running TCP/IP and multicast UDP/IP.

On the “inside” of the antenna, there are numerous systems that need control, including the LNA’s, refrigerator, tracking motors, etc. All these devices are controlled by the main processor module, which we call the “black box.” The black box exposes one interface to the outside world (Ethernet) and multiple interfaces toward the inside of the antenna (tracking motors, etc.). We developed the black box concept to make it a “replaceable” unit. The technology implementing the black box might change in the future (due to changes in availability and/or improvements in technology). The black box is therefore defined by its interfaces. Apart from power (TBD), the black box exposes six well-defined interfaces.

2. Description of the Black Box Interfaces

The black box exposes only one interface to the outside world (either ATA control system or field technician): a 10 base T Ethernet port, coming in on optical fiber. Command and monitor communication passes through this port via TCP/IP and multicast UDP/IP. The application level communication interface will be described later in this document.

On the “inside” of the antenna, the black box exposes five interfaces. Three interfaces are directed to the azimuth, elevation, and z-actuator stepper motors consisting of a number of parallel TTL signals. The other two are standard serial interfaces (initially RS232 for PTA installation, migrating to RS422 or other standard for ATA) that are directed to two 8-bit microprocessors (e.g. Microchips PIC).
For the azimuth and elevation drive interfaces, command consists of three balanced TTL lines (six wires grouped into three pairs per motor) identified as Direction, Step, and Enable. The motor moves in the forward or reverse directions depending on the value of the Direction line. Motion is achieved by pulsing the Step line at a rate from 0-10000 Hz (corresponding to ~ 0-1° per second). For each complete pulse delivered to the Step line, the stepper motor moves one step. Finally, the Enable line turns the motor on and off. If Enable is low, then pulses sent to the Step line are ignored.

On the monitor side of the motors, both azimuth and elevation drives have an incremental encoder. The encoder requires +5V and Ground input, and provides three balanced TTL outputs (six wires per motor) designated A, B, and Z. The A line delivers a complete pulse cycle every time the encoder rotates through an angle of $\frac{360}{28800} = 0.0125^\circ$, with the rising and falling edges separated by half this angle. The B line delivers pulses in the same way, except that the edges of the B pulses are displaced from the A pulses by approximately $0.0125 / 4 = 0.0031^\circ$. The Z line delivers a pulse (width ~ 0.0063°) only once per revolution of the encoder. By design, the Z line pulse will occur only once over the full range of motion of the axis, and can be used as a reliable fiducial. For more information on the encoders, see “Some notes on Az/El encoders,” a memo found on the ATA software website.

Each of the azimuth and elevation monitor interfaces has two more balanced TTL lines (four wires per motor), both of which are nominally in one state (TBD). These lines change state only when the antenna has reached the end of its range of motion in either the forward or reverse direction.

The monitor / command interface to the z-actuator is very similar to that for the azimuth and elevation drives except that it has no encoder. It has three balanced command lines (six wires) going to the stepper motor driver. In this case, each z-actuator step corresponds to motion of the feed either toward or away from the secondary surface (by an amount TBD). It also has two balanced monitor lines (four wires), which change state when the z-actuator reaches the end of its travel in either the forward or reverse directions.

All the rest of the “inward facing” monitor / command communication occurs through two standard serial interfaces. For the PTA installation, an RS232 interface is used, and the low-level protocol is compliant with that for a standard PC COM port (9600 baud, 8 bits, 1 stop bit, no parity). The hardware interface and low-level protocol may change for the ATA installation.

Each of the two serial ports is connected to an 8-bit microprocessor. For now we assume these microprocessors are Microchip PIC chips though this may change. Each PIC controls numerous devices on the antenna, including digital I/O monitor and command lines, analog to digital converters (e.g. sensors), and digital to analog converters (e.g. actuators), all of which comprise command and control of the feed electronics,
refrigerator, air circulation, temperature control, etc. See Appendix 1 for a description of the high-level interface to the PIC chips.

3. Design of the Black Box

The current design of the black box employs a single-board computer to perform its processing requirements. For the PTA installation, we have chosen the EVB100M manufactured by Ajile systems, Inc. (www.ajile.com). The main processor on this controller is an Ajile Systems aJ-100, a 100 MHz native Java processor. This board is equipped with 16 MB static RAM, 4 MB programmable flash memory, and provides 2 PC-compatible RS232 serial ports. It also exposes many 3.3 V (CMOS level) digital I/O lines, sufficient to implement all the stepper motor interfaces. However, the voltage levels must be converted to TTL standard (0 / +5V) and a RFI mitigating buffer is desirable. For this purpose we add a small card containing 21 opto-isolator buffer chips (one for each balanced signal line).

The ATA control system is written mainly in Java, and it is desirable that the black box is programmed in Java as well. Various options were considered for the single board computer and one based on the aJ-100 chip appears best suited for our application. The EVB100M is rather expensive and has many unnecessary facilities (e.g. LCD and touch screen controllers) for the ATA antenna. For this reason, a stripped-down version of this card is currently being prototyped and will probably be used in ATA deployment.

We also note that if, for any reason, the Ajile chip might become undesirable or unavailable then another java-enabled micro controller can be chosen as the heart of black box. In this case, no change in hardware or software is required either up or downstream of the black box. Because of the inherent portability of Java, the change to a different controller card is expected to be less difficult than might otherwise be the case.

4. “Operating System” and Programming

The aJ-100 employs no operating system to speak of (low-level process scheduling exists, but is transparent to the user). Instead, the user runs either one or two Java virtual machines (JVM’s) concurrently on the chip, and all applications run under these. The PTA antenna software will all run on a single JVM. (Distributing the antenna software across two JVM’s might hold advantages and may be pursued for ATA implementation.) This software is stored in flash memory, and automatically starts up when power is applied to the black box.

During development and initial installation, the Ajile card is programmed via a JTAG port. Subsequent program re-installation can, in principle, be performed via Ethernet using Ajile’s proprietary API (no standard Java API exists). I have experimented with this API and it is mostly functional. Applications can be downloaded into RAM from

---

1 See “Single Board Computer for Antenna Controller,” a memo on the ATA software website, for justification of this choice.
2 At present, our backup system consists of a Pentium 133 embedded controller with 64 MB of RAM running embedded Linux.
Ethernet for on the fly updates. Downloading to flash is possible but more complicated. Since we desire the latter, we choose to postpone implementation of this feature until after the PTA software requirements have been met. Since this is not a requirement for the PTA, it makes sense to wait as Ajile and third-party developers may improve the dynamic downloading API by the time we need it.

5. Java Support

For an embedded controller like the EVB100M, the level of Java support can vary greatly, and Sun has developed various “versions” of Java appropriate for devices with differing capabilities. For example, many cell phones are programmed via the Java CLCD API (Connected Limited Device Configuration). More recently Sun developed the CDC API (Connected Device Configuration), which is aimed at more capable embedded devices. The next step up is J2SE (Java 2 Platform, Standard Edition), which is aimed at PC-level systems. Because the primary ATA control system is written with J2SE, it is desirable to have the same support in the antenna controller.

On an orthogonal development path, Sun has recently released the Real Time Java (RTJ) API, providing extensions not available in the releases mentioned above. These extensions address limitations of “standard” Java for real time work. For example, automatic garbage collection takes place as a preemptive task in standard implementations. This means that occasionally, your application halts for several milliseconds (or more) while the garbage collector is run. In hard real time applications, such behavior is unacceptable, and RTJ give the programmer greater control over garbage collection.

The Ajile aJ-100 processor was designed as a highly efficient real time Java processor.\(^3\) It is ideally suited as a platform for Real Time Java and CDC, and was developed in parallel (actually in advance) of these. Because these API’s are new, Ajile has not completed the software implementation of either. Currently the Ajile API is fully compliant with CLDC and implements a growing fraction of the CDC and RTJ.

Specifically, Ajile uses exactly the same serial port API as J2SE, but uses a different (albeit closely related) socket API for the Ethernet. This is only slightly inconvenient. When CDC support is complete, the Ethernet API will be identical to J2SE. There exists no standard API for digital I/O, so by necessity this is not portable. Ajile provides a proprietary API for I/O control. Among other things, it provides three frequency-programmable timers that are ideal for controlling stepper motors.

Sun’s CLDC does not support built-in Object Serialization, Reflection, or Discovery, nor does Ajile at this time. This is a more serious inconvenience, since these elements are critical to the implementation of JSDA (Java Simple Distributed System, the distributed processing “glue” we have adopted for the main ATA control software). When CDC

---

\(^3\) Indeed, the Ajile processor is unique in its support for real-time embedded Java. On the other hand, recent announcements by ARM and others suggest that the field of true real time Java embedded systems may soon expand from one to several. This is good news for us, as we may soon have more options for Ajile replacements, and competition will spur Ajile to improve.
support is complete these elements will exist, but until then we make up for the lack in application software.

If one is careful to make all classes available to the application at compile time, it is possible to eliminate the need for Discovery. Serialization is realized by a straightforward (though tedious) implementation in every class that we transmit to or from the antenna. When CDC support is complete, switching over to Sun’s Serialization and Discovery API’s should be straightforward. Simulation of Reflection is probably unfeasible – if it isn’t built in then you choose an approach that doesn’t require it.

Considering the present Ajile API, we choose not to port JSDA over to the aJ-100. Instead, we port only a stripped-down application-specific fraction of the JSDA functionality. This isn’t as bad as it sounds, since the majority of antenna communications are multicast UDP-based, whereas JSDA is currently implemented with point-to-point TCP (this may change for the ATA). Hence, the antenna communication system is developed in parallel with JSDA, and by the time of ATA implementation these developments should converge.

6. Vision

An important driver of the software design is the vision of “how it all should work” held by the ATA software team. From the standpoint of the ATA control system, an antenna is just another device on the network, like a local oscillator. The software has no sense of the distance between the control room and the antenna.

In the control room, the highest level of communications travels over the JSDA network. JSDA servers and clients attach to (and detach from) this network over time. Each JSDA server has a unique name and may be addressed by any other server or client (although they may not get access depending on their security clearance). Ultimately each of all the 350 antennas will probably correspond to one or more JSDA servers. But whether or not this is the case, Antennas are expected to fail occasionally and eventually recover. When this happens the ATA control system should “discover” that an antenna has disappeared or reappeared, and celestial observations should not be interrupted by the failure or recovery of a single antenna.

The antennas are usually addressed in groups. Therefore it is convenient to place another layer of abstraction between the greater network and the individual antennas. We call this the AntennaServer, which is a JSDA server that manages antenna resources and resides (at least conceptually) in the control room. For example, with just a few commands to the AntennaServer, you can create an AntennaGroup consisting of all 350 antennas and then command this group to track Cassiopeia A.

Each antenna has unique pointing offsets, unique optimal LNA bias settings, and other parameters. However, our vision has every antenna behaving as if they were identical, so that a single command can control them all. For this reason, each antenna needs a capable processor of its own and local nonvolatile storage. All the unique parameters can be transmitted to and stored at each antenna. Thus when an antenna receives a command to
track Cas A, it can deduce the unique motor positions required to point in the right direction.

Because of the non-real time nature of J2SE and TCP/IP (the platform for the ATA control system), communication with the Antenna must be “lazy.” Lazy means that all communication loops to/from the antenna have a repetition rate of less than 10 Hz (usually, much less). It also means that required data (like a tracking trajectory) can always be issued in advance if desired (at least 1 second before it is needed). This way, we can guarantee that the data will always arrive before it is needed at least 99.99…% of the time. Furthermore, if data arrives late, then it is desirable for the antenna to carry on in a fashion that is the best guess of what would be the right thing to do.

To this end, we notice that almost everything the Antenna does can be modeled as a (piecewise) continuous function of a single variable, time. For example, to track Cas A, the azimuth and elevation settings can be calculated hours or even years in advance and expressed as smoothly varying functions of time. We abstract the notion of such a function as a Trajectory. Typically, trajectories correspond to pointing tracks, but they could also correspond to a voltage setting or anything else. In our vision, we use Trajectories to express all control variables that fit nicely into this paradigm, meaning that many control commands can be issued only once every few minutes.

If the Antenna is commanded with Trajectories, it must have an accurate notion of time. The most time-critical job at the Antenna is tracking, and a simple analysis shows that time accuracy must be of order 1 ms to meet our pointing requirements (see “PTA/ATA Requirements Working Document” on the ATA software website). Since our vision makes the Antenna completely autonomous apart from its Ethernet connection, time synchronization must be accomplished via the network. Thus, some kind of Network Time Protocol (NTP) must be established out to the Antenna.

We say that Trajectories are piecewise continuous because sometimes there are abrupt transitions separating continuous function segments. For example, suppose you are performing a mosaic of observations centered on Cas A. This means that you track a position which is a fixed distance from Cas A for a short period, and then abruptly change to track another position which has a different fixed distance from Cas A. Typically, the set of all fixed positions forms a rectangular grid centered on Cas A. Our software system encodes this mosaic Trajectory as the sum of two fundamental Trajectories: the Trajectory of Cas A, and the Trajectory of a rectangular grid. By specifying these two, and the times when the grid Trajectory should “increment,” one completely specifies the mosaic Trajectory.

Since each Antenna moves at a different rate, not all Antennas will arrive at the next grid point at the same time. The time when each Antenna arrives must be noted and stored by the ATA control system. Thus, each Antenna must return messages to the ATA control system at unpredictable times. Furthermore, when something goes wrong at the Antenna (by definition unpredictable) the Antenna must send a message to the control system. In
addition to this unpredictable communication, the ATA control system will sometimes query the Antenna for information about its state or one of its components.

The Antenna must also perform some self-monitoring. To the extent possible, it watches its own software and informs the ATA control system if something goes wrong (e.g. communications error, fatal software error, hardware unresponsive, etc.). It also performs corrective action (e.g. resend message, reboot itself) when this is feasible.

7. Program Design – Host Side

By now we have outlined most of the constraints of the PTA antenna application. We have a single application running in a single JVM. Raw socket connections provide communication with the main control system and serial ports fan out to the PICs. Twenty-one digital I/O lines command and monitor the stepper motors. Now we need to glue it all together and attach it to the ATA control system.

It is perhaps easiest to start from the top, in the ATA control system (host). The host runs an application (AntennaServer) that keeps track of all the Antenna’s that are on line. If any other client in the ATA control system wishes to communicate with the Antennas, it must go through AntennaServer. Similarly, the AntennaServer can communicate with each Antenna mainly through one entity (called Antenna). The only exception to this rule is NTP, and AntennaServer performs specialized timekeeping communication by direction communication with the NTP client (we defer discussion of NTP until later in this document).

7.1. Initialization

When it starts up, the AntennaServer first goes to the ATA control file store and downloads a list of all the antennas. This list associates a unique name to each antenna, based on its IP address (the name is just an integer number). The list also contains the (x, y, z) position of the antenna, pointing corrections, and other antenna-specific data.

Then AntennaServer starts up a server socket on a well-known port and a multicast socket on a well-known multicast IP address. Most of the communication between AntennaServer and the Antenna’s occurs over this multicast socket. At regular intervals, AntennaServer broadcasts an mUDP packet containing its own IP address. When an Antenna first wakes up, it opens a socket on the well-known multicast IP address and listens for such packets. When it discovers AntennaServer’s address, it initiates a TCP socket connection to the AntennaServer.

As each Antenna calls in, the AntennaServer examines its IP address and performs some initial negotiation – giving the Antenna its name, (x, y, z) position, pointing corrections, etc. It then registers the Antenna and makes it available to the rest of the system. As far as the AntennaServer is concerned, the TCP socket IS the Antenna. If this socket connection breaks, then the Antenna is assumed dead (until it calls back). If the Antenna becomes unresponsive over this socket, then it is recorded as being in “zombie” state and corrective action is taken.
7.1.1. Status
Because TCP communication is reliable, all critical communication between AntennaServer and Antenna passes over this socket. Such communication is always initiated by AntennaServer (the Antenna never “pushes” data over this socket). Each Antenna generates many status messages per second that are destined for AntennaServer. If each one were sent as a separate packet, then thousands of packets would arrive at the host each second, saturating the network. To prevent this situation, each Antenna buffers messages until the host asks for them, and then returns them all at once (probably once every few minutes).

At that time, Antenna dumps all of its status data to AntennaServer. Note that every status message has a time-tag with at least millisecond accuracy, so that later an accurate sequence of events (from all Antenna’s) can be reconstructed. When status arrives at AntennaServer, it examines each message and decides how to archive it (e.g. send to science database, send to local file, send to client, discard, etc.).

If each Antenna is polled serially, then the aggregate data rate from all Antenna’s is limited to the 10 Mbit/s rate of the Ajile Ethernet port. For 1000 Antenna’s, this limits the maximal amount of status data that can be transmitted to < 1 kB per Antenna. We overcome this limitation by leveraging buffers in the packet switching hardware. The AntennaServer has a fast (1 Gbit/s) Ethernet port, and polls many Antenna’s at a time in parallel threads. In each thread, the status packets dribble in slowly at the 10 Mbit/s rate, but in principle 100 such threads could run simultaneously without interference. Thus we anticipate that status data of at least 10 kB/s can be generated at each antenna without overloading the TCP network.

NOTES:
1) If necessary, the broadcast messaging could arrive at the AntennaServer through a separate Ethernet adapter. For that matter, the AntennaServer could run multiple Ethernet adapters just for the TCP network.
2) The concept of using hardware buffering in the packet switches is in common use, however, a network switch should be purchased and this idea tested to prove our architecture.

7.1.2. Logical Groups
The Antenna provides an interface to allow AntennaServer to assign it to one or more logical groups. Each Antenna already belongs to two groups: a group composed only of itself, and the group of all active Antenna’s. In addition, groups can be created by AntennaServer and given a name. To specify a group, the AntennaServer broadcasts a message specifying the group name and the names of the Antenna’s in the group. Upon receipt of this message, each Antenna in the group responds with a (high-priority, see below) message acknowledging creation of the group. Groups do not remain in existence indefinitely. If a group is not addressed for a long period of time (TBD, probably days) then it will automatically expire.
The Antenna keeps track of its own group memberships and performs message filtering on all broadcast messages by ignoring those sent to groups to which it does not belong. Of course, no filtering is necessary for messages arriving via TCP, since they are addressed specifically to that Antenna.

7.1.3. Broadcast Control

Most command messages are issued over the multicast socket via multicast UDP (mUDP). This way, a single message can be addressed to a large group of Antenna’s. Each mUDP message contains the name of the group to which it is addressed. Here we make a distinction between mUDP packets and mUDP messages. Just as for status, it is desirable to bundle many outgoing messages into a single packet before sending it. When the Antenna receives a command packet, it unpacks it and examines the destination group for each message in the packet, discarding or acting upon each message as required.

As mentioned above, status information returns from the Antenna when AntennaServer calls for it. However, there are occasional cases where it is desirable to have the Antenna alert AntennaServer to a time-critical or unusual condition. For such cases, we have the concept of a “high-priority” status message. Like ordinary status, a high-priority message is queued for TCP transmission. However, duplicate versions of high-priority messages are sent via ordinary UDP back to the AntennaServer. For example, an Antenna might use the high-priority channel to acknowledge the establishment of a new group, or to signal serious errors.

High-priority messaging gives AntennaServer the opportunity to react immediately to a problem. However, in case of massive failure the network (or AntennaServer) may become overloaded, causing UDP messages to be lost (or ignored). This is a positive feature, since the Antenna will not be delayed from taking its own action by redundant messaging. In any event, high-priority messages are never lost, since a duplicate copy is sent via TCP.

8. Program Design – Antenna Side

In the last section, we discussed Antenna initialization and much of its interaction with the ATA control system. Now we focus on communication between Antenna and the subsystems it controls.

8.1.1. Communications

To summarize from section 7, a single program runs on each antenna, whose main class is Antenna. Except for NTP, all communication to the outside world passes through Antenna.Antenna plays the role of JSDA server representing a single antenna, but in fact has a greater role than this. Much of the communication between Antenna and

---

4 Recall that with ordinary UDP, the packet is directed to a single endpoint, and is not broadcast widely. This prevents packets that are destined only for the AntennaServer from being redundantly dispersed to all the other Antenna’s.
AntennaServer takes place over broadcast channels (mUDP), which are outside the JSDA paradigm.

When Antenna starts up, it establishes a TCP socket connection with the AntennaServer, and bi-directional multicast UDP (mUDP) socket connection(s) on a standard IP address(es), to be used for broadcast communication with AntennaServer. If either of these sockets breaks, it is automatically restored.

During initialization, pointing corrections and other antenna-specific information is downloaded via TCP from AntennaServer. The TCP socket is also used for the return of status information, which is sent back to AntennaServer whenever it asks for it. AntennaServer broadcasts most of its commands via mUDP. Antenna initiates broadcast communication only occasionally, when a “high-priority” status message is generated. Otherwise all return communication from Antenna to AntennaServer takes place via TCP (except for NTP, see below).

8.1.2. Composite Classes

The Antenna creates instances of several classes that control the antenna at a lower level. There is the ATATime utility that provides an accurate local timekeeper at the antenna. The WatchDog utility monitors the health of all software threads held by Antenna and takes corrective action if one of these threads should hang. The Tracker class is the pointing abstraction at the antenna. The Antenna feeds Trajectory’s to Tracker, and it is the Tracker’s job to make sure pointing stays within tolerance. The MonitorCommand class abstracts the two PIC chips that control the blower, refrigerator, LNA’s, etc. Commands to control these devices are passed to MonitorCommand, whose role it is to control the PICs.

Sometimes, a class that is below Antenna in the hierarchy needs to communicate with the host. For example, the WatchDog utility needs to tell the host that the system is about to reboot. The Antenna exposes some public methods to allow classes to send status messages to AntennaServer (by either low or high-priority channels). Antenna also exposes various antenna-specific information to its composite classes. For example, ATATime can ask the Antenna for its given name. Tracker can ask Antenna for its pointing corrections, etc.

8.1.3. ATATime

The ATA standard time representation is a single signed long value indicating TAI nanoseconds since Jan. 1, 1970. This implies an upper limit of ~250 years from the present day before we have a “year 2000” crisis. Comparatively, in the J2SE standard the minimum time increment (for currentTimeMillis(), sleep(), wait(), etc.) is 1 ms. RTJ has extensions supporting some functionality to the nanosecond level, but this is not yet supported by Ajile. Ajile currently supports µs resolution with a propriety rawJEM.getTime() function (presumably, they will switch this function over to RTJ eventually).
We wish to use the maximal time resolution available (microsecond) yet not embed a non-standard time call throughout our code. For this reason, we create an abstraction layer in the form of a “singleton” class, called ATATime. ATATime has a public static function called ATATime.currentTimeNanos() that returns our best guess for the current time in the ATA standard. This also allows us (later) to replace proprietary time calls from the Ajile API with standard RTJ calls when these are supported.

Another job of the ATATime class is to compensate for free-wheeling of the on board system clock. The Ajile board will be exposed to large temperature variations in the field, and we expect that the vibration frequency of the on-board clock crystal will vary, especially from day to night. To cope with this, the ATATime class maintains a model of the clock behavior. (Initially, we use a two-parameter Kalman filter whose degrees of freedom are the crystal frequency and time offset from zero.\(^5\)) To train this model, ATATime collects NTP packets (sent via mUDP) from the network on the standard NTP port (these might be standard NTP packets, but they need not be). Such packets are issued by the AntennaServer, but they do not pass through Antenna to reach ATATime. ATATime extracts network time from the packet and compares it with local time. The difference between these two is fed back into the model.

ATATime supports one other action. Occasionally, the AntennaServer will send a special packet to the NTP socket that asks a single antenna to respond. As soon as the specified antenna receives this packet, it responds with a mUDP packet representing local time. All other antennas ignore the request. The AntennaServer uses the observed latency and the Antenna’s time to keep statistics regarding network delay. Subsequently, AntennaServer sends another mUDP packet back to this same antenna, telling it the best estimate of the network delay.

8.1.4. **WatchDog**

The WatchDog utility represents the “dead man’s switch” at the antenna. If some critical part of the software stops running or responding, then the WatchDog attempts to take corrective action, up to and including rebooting the antenna’s single board computer. There is a real, hardware watchdog timer attached to one of the PIC chips, and it is the WatchDog’s duty to send messages to this timer at sufficient frequency to prevent reboot (unless something goes wrong).

The WatchDog is a singleton and it provides static methods available to all classes. One of these methods allows classes to register with the WatchDog and specify a check-in period. After registration, the registering class agrees to check-in with the WatchDog at least once per check-in period forever (or until it unregisters). If the registering class fails to check-in, the WatchDog assumes a failure and takes corrective action.

Corrective action consists of the following. The WatchDog sends a high-priority message to AntennaServer to announce the problem. One action that AntennaServer may take is to flush the status queue on Antenna. The WatchDog

---

\(^5\) For more information about the Kalman filter, see “A Two Parameter Kalman Filter Implementation;” a memo on the ATA software website.
then attempts to send a shutdown message to all registered classes and gives them time to
clean up. Finally, the WatchDog allows the hardware watchdog timer to expire, which
initiates a reboot of the system.

To enable WatchDog to call “shutdown” on registered classes, these classes must
implement a standard interface called Watchable. This interface defines two methods,
shutdown() and getCheckInPeriod().

8.1.5. Tracker
Trajectory’s are broadcasted (mUDP) from the AntennaServer to all
Antenna’s, and are directed to a specific group name. Antenna’s in that group
respond to the Trajectory but others ignore it. The Antenna passes the
Trajectory’s on to the Tracker. There are several subtleties associated with this
process that we discuss here.

The complete pointing Trajectory may be composed of multiple sub-
Trajectory’s. Trajectory’s come in two flavors: continuous and incremental. A
continuous Trajectory varies constantly and continuously with time, and the antenna
follows the path of this Trajectory as closely as possible (usual case). An incremental
Trajectory is constant for some time period and then it abruptly changes to another
constant position. Applications of incremental Trajectory’s are, for example, A)
switching back and forth between “on-source” and “calibrator” in an observation, and B)
cycling through a rectangular grid one point at a time. In case B), this grid is centered on
a potentially moving target, as in mosaic observations, by “summing” the incremental
Trajectory with another continuous one.

One subtlety is to get all the antennas to move synchronously to a new trajectory, since
antennas may take varying amounts of time to get “on-source.” Rather than try to predict
in advance how long it will take to get on-source, we instead specify the time when the
antennas should begin moving toward the new Trajectory. In this scenario, the host
begins taking data immediately. When each antenna arrives on-source, it records the time
and sends it’s status back to Antenna for lazy transmission to the host.

A very similar mechanism is used to update incremental Trajectory’s. When it is
time to switch to the next increment, the host broadcasts an increment message. As soon
as it receives this message, the Antenna passes this message to the Tracker, which
starts to move to the next increment position. When the target position is achieved, the
Tracker records the time and sends a message back to the Antenna for eventual
archiving at the host.

Another subtlety arises when the Tracker loses track, as when a Trajectory passes
too close to the antenna keyhole or if a wind gust blows the antenna off source. In this
case, the Tracker sends a high-priority message back to the Antenna that it has lost
track. When tracking is once again established, the Tracker sends another message
back to Antenna indicating this state.
The Tracker keeps statistics of the pointing accuracy over the time periods for which it is on source. This information is sent back to the Antenna at regular intervals. The Antenna joins this information with other status messages and lazily sends it back to AntennaServer for archiving.

Another point is that Trajectory’s can be updated after they are installed. Each Trajectory has a unique name, and if the AntennaServer sends a Trajectory whose name is the same as an existing one, then Antenna merges the new Trajectory with the old one without stopping tracking. At merge time, the Trajectory will discard parts of itself relating to the distant past. This way, Trajectory’s cannot grow without limit.

8.1.6. Tracker Implementation

The Tracker holds two instances of the StepperMotor class, which are abstractions of the azimuth and elevation motors. Each stepper motor is controlled by setting its angular velocity. Tracker also holds two Encoder class instances, which are abstractions of the incremental encoders. The actual physical encoders generate an interrupt every time they increment. The interrupt handler notes the time of the encoder tick and places a message in a FIFO stack. The Encoder class processes these messages, in order, and uses them to train a model of the antenna motion. (Once again, we employ a two parameter Kalman filter as a model for each antenna axis. The parameters in this case are the axis position and velocity.)

At regular intervals, the Tracker examines the antenna position and compares it to the Trajectory. If the position goes out of tolerance with that specified by the Trajectory, then a high priority message is sent to AntennaServer.

8.1.7. MonitorCommand

The description of this class is still sketchy, but it should firm up as the devices that are to be controlled begin to appear.

There are dozens of devices that require remote control by the two PIC chips. Fundamentally, however, there are only three types of control: ADC, DAC, and digital I/O. At a higher level, however, a particular I/O pin might correspond to the blower, while another group of I/O pins might control a digital actuator. One ADC corresponds to temperature while another corresponds to refrigerator driving current.

To keep the programming of the PIC chips as simple as possible, the PIC chips know nothing about the devices they control. PIC’s are commanded by directly manipulating ADC’s, DAC’s, and I/O (see Appendix 1 for PIC interface). The MonitorCommand class, however, does know which I/O pin corresponds to the blower and exposes a human-readable interface to the Antenna class (i.e. a setBlower() method). The MonitorCommand class can be configured to return the values and settings of any and all the PIC devices at any frequency (up to a max of 10 Hz or so, TBD). These status messages are sent back to the Antenna for archiving in the usual way.
At present, there does not appear to be a need for anything more than immediate addressing of PIC devices. For example, there is no need to program a “ramp” on some device to be executed over a particular time interval. Such controls can be added later, on an ad-hoc basis. MonitorCommand has access to ATATime, so such programming is straightforward.

Similar to Tracker, MonitorCommand can be the recipient of broadcast messages from the host. This way, groups of antennas can be addressed with only a single command. Capture of the mUDP message really happens at the Antenna level, so MonitorCommand really doesn’t know the difference.

9. Conclusion

The above represents a first cut of the Antenna control software as envisioned by the author. Please note the date under the byline above to draw conclusions about whether it is up to date. As we move from the PTA into the ATA development phase, some of the architecture may change. For example, when Ajile provides full CDC support, some of the communications between AntennaServer and Antenna may occur over the JSDA channel. Also, JSDA may one day support multicast communication. If / when this happens, AntennaServer to Antenna multicast communication will switch over to JSDA. Indeed the largest driving force to provide JSDA multicast support is exactly for control of the Antenna network (and the IF Processor network which probably will also use Ajile computers.)

10. Appendix 1

In this section we summarize PIC API. It is very simple by design. Everything that passes through the serial port is ASCII text (plus CRC bytes). To command a PIC, you construct a string like

“SET BLOWER = 1\r”

and then feed it into a special class that generates a two byte CRC and appends it to the string. Then you send this string to the serial port for the associated PIC. The PIC receives the message, checks the CRC, acts upon the message and returns a string of its own, in this case

“BLOWER = 1”

is returned if the operation was successful (and after stripping CRC at end).

Here is a complete list of the callable methods and their successful return strings:

<table>
<thead>
<tr>
<th>Function Prototype</th>
<th>Return String (on success)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SET NAME = value</td>
<td>NAME = value</td>
</tr>
<tr>
<td>GET NAME</td>
<td>NAME = value</td>
</tr>
</tbody>
</table>
In this list, all strings are upper case. The placeholder “NAME” can be replaced with any ASCII string that contains no white space. “TYPE” may have only three values: IO, AD, or DA. The expression “(parms)” is a string of comma-delimited values that specify the entity to be named and its setup parameters.

The "LABEL" command works like a constructor. You can't address any device on the PIC until you have given it a unique name. At naming time, you can supply whatever input parameters are needed. For IO, this includes which pins are part of the "port" you are establishing. In the case of ADC, the key input parameter is the frequency at which the ADC shall sample its sensor. DAC’s are also given names, but they take no input parameters.

For example, to address digital I/O pins 4-8 as a 4-bit parallel port, one uses the command:

“SET IO(4-8) = Attenuator\n”

which generates the return string (after stripping CRC)

“IO(4-8) = Attenuator”

The PIC chip keeps a running tally of max, min, mean, and standard deviation for each ADC. Whenever "getAD" is called, statistical values are returned, all values are reset and the whole thing starts over.

The CLEAR command causes the PIC to completely purge its name list. All devices return to their uninitialized state, and the PIC behaves as if it were just turned on.

On the Ajile side, the CRC is handled by a special class called AjilePic8bitCRC. Before sending a string to the PIC, you must pass the string through this class to have it append two CRC bytes to its end. The PIC chip reads these bytes to verify the string. Similarly, on return messages the PIC appends a two-byte CRC. Before examining such messages, they should be passed through the AjilePic8bitCRC class for verification. AjilePic8bitCRC will strip the carriage return and CRC bytes from the end of the string for your convenience.

**10.1.1. Error Conditions**

The PIC might return one of several error messages, including “BADCRC”, “SYNTAX”, “DEVICE”, and “VALUE”.

“BADCRC” indicates that your last message was corrupted upon arrival. It should be sent once more.
“SYNTAX” indicates that your message doesn’t make sense. Because of the CRC validation, this probably indicates a programming error.

“DEVICE” indicates that the specified device does not exist (e.g. you are addressing an I/O line that is beyond the last one available.)

“VALUE” indicates that you have a valid device, but you have specified a value that is out of range. For example, if you sent the value 65535 to a 4-bit I/O port, this would generate a “VALUE” error.

Finally, two other errors might occur. Firstly, the returned message from the PIC might fail the CRC check (it is corrupted). In this case, you’ll have to decide for yourself what to do. Secondly, the serial port might fail (hardware failure). Depending on what kind of exception is generated by the serial driver, appropriate corrective action can be taken.