Integra 85 Focusing Rotator from Gemini Telescope Design

Firmware coded by Tim Long of Tigra Astronomy

Design Philosophy

Within the limitations of the Arduino Uno (a resource-constrained embedded system), we have tried to apply the SOLID principles of object oriented design:

S - Single responsibility principle; each class should have only one responsibility O - Open/Closed principle; open for extension, closed for modification L - Liskov substitution principle; superclasses and subclasses classes should be interchangeable I - Interface segregation principle; no client should be forced to depend on methods it does not use D - Dependency inversion principle: depend on abstractions not details.

Well-factored object oriented code that adheres to the SOLID principles should be loosely coupled, highly cohesive, testable and have low viscosity for future maintenance.

Due to resource constraints imposed by the target hardware, we have had to refactor some of the code into a more procedural style in order to save memory.

Memory Management

Dynamic memory allocations have been agressively avoided. As a resource-constrained embedded system with just 2Kb of data memory, there is not much space available for a heap and we can't tolerate "Out Of Memory" errors at runtime. The system must be stable for days, months or even years at a time so the memory management strategy must be frugal, deterministic and stable.

Our solution to this is to statically pre-allocate as many objects as possible once, in global scope, then never delete them. The top level `.ino` file contains these allocations either as statically initialized global variables or in the setup() method and this essentially forms the Composition Root for the system.

Since we can assume that most objects are never freed, there is little to be gained from the use of smart pointers and we have chosen to avoid the overhead and use "raw" pointers where necessary. We do however make use of the `std::vector<T>` class to manage collections of pointers.

Motor Control

The two stepper motors have a Direction/Step/Enable interface and are driven by generating a square wave onto the Step pin. The process is logically devided into two parts:

Step Generator

A step generator (implements: `IStepGenerator`) is responsible for generating a pulse train where each rising edge causes the motor to make one step. The step generator is responsible for timing (step speed) and has no concept of position, direction or the type of steps (whole steps, microsteps, etc.)

We have provided a single implementation, `CounterTimer1StepGenerator`, which uses the Timer 1 block of the AVR processor to generate accurately timed pulses with 50% duty cycle. The timer is configured to generate interrupts using the `OCR1A` compare register. The timing source is the undivided system clock, which allows for a theoretical stepping bandwidth of about 244 steps/second up to 16,000,000 steps/second.

Step Sequencer

The step sequencer (implements: `IStepSequencer`) carries the responsibility of writing the correct hardware signals to the motor driver and keeping track of the step position.

Our `Motor` class provides the `IStepSequencer` implementation and allows for acceleration and deceleration. `Motor` also keeps track of the current step position and enforces limits of travel on the motors.

Derived from `Motor` is `BacklashCompensatingMotor` which performs backlash compensation by adding the backlash amount whenever the motor direction changes. This process is hidden from the client. Thus, if backlash is measured to be 100 steps, the last move was outwards and the client requests a move inwards of 1000 steps, then the reported position will decrease by 1000 but the motor will actually move 1,100 steps.

Acceleration

The `Motor` class implements acceleration and deceleration based on the equation of uniform acceleration, v = u + at. This reduces the risk of stalling, especially when moving heavy loads.

The `ComputeAcceleratedVelocity` method is called once per Arduino main loop to recompute the motor velocity and acceleration curves.

The "Ramp Time" (the time taken to accelerate from rest to maximum speed) is configurable by the user. Ramp time is specified in milliseconds. 250 to 500 milliseconds is usually sufficient for moderate loads but for more massive loads this can be increased.

Speed and Power Considerations

The Integra 85 uses a worm and worm wheel gearing arrangement to drive both the focuser and rotator mechanism. This arrangement provides a naturally stable system at rest which means that holding torque in the stepper motors is unnecessary. Therefore, the step drivers are disabled once motion has ceased. This reduces power consuption and keeps the motors and step drivers cool when not actively driving the mechanism.

While the firmware is capable of very high step rates, there is a trade-off between maximum stepping speed, available torque and power consumption. The firmware provides commands for adjusting maximum step rate (`VW`) and acceleration ramp time (`AW`) so that the end user can manage this speed/torque/power tradeoff.

In practice the motors used in the Integra 85 will perform well up to at least 16,000 microsteps, 1000 whole steps or about 5 revolutions per second when used with moderate loads.

Beyond that, the motors may be unable to provide sufficient torque and may stall. Therefore we do not advise increasing the maximum step speed much beyond the default 16,000 microsteps per second.

For larger loads with higher inertia, it may be desirable to lower the maximum step rate (`VW`) and increase the acceleration ramp time (`AW`).

Limitations

In this implementation, all motors share the same `IStepGenerator` because only one 16-bit counter/timer block is available on the Arduino's AVR processor core. Therefore only one motor can be in motion at any moment. No gaurds are in place to prevent misoperation. It is assumed that this will be controlled at a higher level (probably in the ASCOM driver).

Command Processor

Command processing is handled by the `CommandProcessor` class.

Originally, we had a nice object oriented design for the command processor but it used one class for each command and then sometimes multiple instances of each command processor for different devices. It all took up a bit too much memory for the Arduino Uno, which only has 2 Kb of data memory. The pattern is recorded in the version control repository and may be useful in future projects where resources are less constrained.

When a well formed command is received from the communications channel (serial or bluetooth) it is passed to `DispatchCommand()` which passes it on to `CommandProcessor::HandleCommand()`.

Each command verb has its own handler method, so for example, the `PR` (position read) command would be handled by `CommandProcessor::HandlePR()`. `CommandProcessor::HandleCommand()` decides which handler method to call based on the command verb.

All command handlers return a `Response` structure, which contains the text to be transmitted via the serial port or Bluetooth adapter to the client application.

Command Protocol

Command Grammar

Commands have the form: `@` `Verb` `Device` `,` `Parameter` `<CR><LF>`.

- `@` is a literal character that marks the start of a new command and clears the receive buffer. Use of the `@` initiator is optional, but recommended.
- `Verb` is the command verb, which normally consists of two characters. Single character verbs are also possible but in this case the entire command is a single character.
- `Device` is the target device for the command, generally a motor number `1` (focuser) or `2` (rotator). Where no device address is given, a default value of `0` is assumed.
- `,` is a literal character that separates the device ID from the parameter.
- `Parameter` is a positive integer. If omitted, zero is assumed.
- `RETURN` `LINE FEED` is the command terminator and submits the command to the dispatcher. Only one is required. If both are present then they can be in any order.

Example: `@MI1,1000`.

If the parameter field is not required for a command, then it can be omitted or, if specified, it will be ignored. For example, the following are all equivalent: `@PR1`, `@PR1,`, `@PR1,1000`

Errors

Any unrecognised or invalid command responds with the text `Err`.

Command Protocol Details

```
Command | Reply    | Min  | Max     | Default | Notes
========|=========|======|========|=========|==============================
=======================
Motor configuration: 1 whole step = 16 microsteps
--------|---------|------|--------|---------|------------------------------
-----------------------
AWm,n   | AW#     |    1 |  65535 |     500 | Set the acceleration ramp
time in milliseconds
RRm     | RRnnnn# |      |        |         | Reads the range of movement
in steps for motor m
RW1,n   | RW#     |    1 | 2^32-1 |  198000 | Sets the limit of travel for
the focuser in whole steps
RW2,n   | RW#     |    1 | 2^32-1 |   61802 | Sets the number of whole
steps per revolution for the rotator
PRm     | PRnnnn# |      |        |         | Read step position of motor m
in whole steps
PWm,n   | PW#     |    0 | RRm    |         | Sync current whole step
position. Max value is returned by RRm
VRm     | VRnnnn# |  16  |  65535 |    1000 | Read maximum motor speed in
steps/second
VWm,n   | VW#     |  250 |  65535 |    1000 | Write maximum motor speed in
steps/second
--------|---------|------|--------|---------|------------------------------
-----------------------
Motor movement
--------|---------|------|--------|---------|------------------------------
-----------------------
MIm,n   | MI#     |    0 | PRm    |         | Move in or anticlockwise n
whole steps
MOm,n   | MO#     |    0 | RRm-PRm|         | Move out or clockwise n whole
steps
SWm     | SW#     |      |        |         | Emergency stop (no
deceleration)
X       | Xn#     |      |        |         | 0=stopped; 1=focuser moving;
2=rotator moving
--------|---------|------|--------|---------|------------------------------
-----------------------
Backlash and calibration commands only valid for focuser (m=1)
--------|---------|------|--------|---------|------------------------------
-----------------------
BRm     | BRnnnn# |    0 |        | auto    | Read backlash amount, in
whole steps.
BWm,n   | BW#     |    0 | 0.5 RR | auto    | Write backlash amount. Max
value is half the limit of travel.
CSm     | CS#     |      |        |         | Start calibration. Measures
home position and backlash amount.
CEm     | CE#     |      |        |         | Stops calibration and sets
status to Cancelled
```

```
CRm     | CRn#    |      |        |         | Returns 0=Uncalibrated;
1=Calibrated; 2=In Progress; 3=Cancelled
CWm,n   | CW#     |    0 |      1 | auto    | Force the calibration state
to 0=Uncalibrated; 1=Calibrated
Clm,n   | Cl#     |    1 |   1023 |     300 | Set the touch sensor "first
contact" threshold
CLm,n   | CL#     |    1 |   1023 |     600 | Set the touch sensor "hard
stop" threshold
Cvm,n   | CV#     |  250 |  65535 |    2880 | Set calibration slow motion
motor speed
--------|---------|------|--------|---------|------------------------------
-----------------------
System-wide commands
--------|---------|------|--------|---------|------------------------------
-----------------------
ER      | ERnnnn# |    0 |   1023 |         | Reads the value of the touch
sensor
FR      | FRm.n#  |      |        |         | Reads the firmware version
major.minor
TR      | TRnn.n# |      |        |         | Reads the temperature in °C.
ZR      | ZR#     |      |        |         | Loads settings from
persistent storage.
ZW      | ZW#     |      |        |         | Writes settings to persistent
storage
ZD      | ZD#     |      |        |         | Reset to factory defaults.
Erases all saved data.
--------|---------|------|--------|---------|------------------------------
-----------------------
```

Note that omitting all of the optional parts of each command gives a more convenient syntax when entering commands manually in a terminal emulator. The table above gives the shortest possible form of each command. However, when commands are generated programmatically, it is recommended that the initial `@` character is always included.

Arduino Libraries Used

- ArduinoSTL - standard template library
- eeprom - for reading and writing the nonvolatile storage
- SoftwareSerial - used to access the Bluetooth module
- OneWire - used for low-level access to the temperature probe.
- DallasTemperature - intermediate level interfacing to the temperature probe

Revision Notes

Release 2.0

First version by Tigra Astronomy. Implements almost all of the commands from release 1.0 but with completely re-written firmware.

Release 2.1

Added the `ER` command (Read force-sensitive resistor value). This command was actually implemented in the previous version but not wired into the command processor, so there was no way to invoke it.