**A**rduino **S**ervice **I**nterface **P**rotocol

Michael Margolis

Document version 2c

26 September 2014

This document defines a simple protocol for controlling an Arduino from a computer. The protocol is intended to be a more extensible replacement for firmata. A prototype of this interface is currently implemented and feedback on this proposal is desired before finalising this interface.

**Summary:** The interface is intended to provide similar functionality to the current firmata mirtle implementation with the additional aim of being easier to extend to support higher level services (motor speed and direction, distance measurement etc). This new protocol abstracts services from the hardware which makes it easier to ensure that hardware attached to the Arduino board is correctly mapped to software running remotely on the client.

To achieve these goals, the interface exists at two levels. The lower level supports Arduino primitives such as:

pinMode

digitalRead

digitalWrite

analogRead

analogWrite

The higher level interface supports abstracted services that will include:

Set Motor Speed and direction

Get encoder counts and pulse widths

Get bump switch states

Get line sensor values.

Report distance read from a distance sensor

Set servo angle

Get acceleration/rotation/heading/altitude from IMU sensors

The protocol also provides messages that provide information about the asip server that is running on Arduino. Error messages are sent with a header tag ‘~” followed by a string describing the error. Debug messages (when enabled by a define in the asip.h header file) are sent with ‘!’ as the header tag followed by a the debug message string. Error and debug messages (like all asip messages) are terminated with a newline character. The system also has a message the provides the protocol version running on Arduino along with the microcontroller chip and the name of the sketch- here is an example of the version message, in this example, version 0.1 is running on an ATmega328 using a sketch named TestIO:

“@#,?,0,1,ATmega328P,TestIO\n”

The low level interface works at the pin level, similar to firmata. The higher level services interface uses enumerations so clients are isolated from specific pins and hardware implementation.

**Low Level Arduino Pin I/O Interface**

Digital pins (0-19 on the Uno) and Analog pins (0-6 on the Uno) will be supported . In the Mirtle implementation, most of these pins will be assigned to on-board robot hardware functions (motor control, encoder etc) and it is expected that the higher level functions described in the next section will be used to access these services. However, the low level functions could be on used if desired, particularly on a standard Arduino board to provide firmata-like remote access to Arduino pins. Low level functions can be used alongside the high level services on any pins that do not conflict with those used by the services.

Similar to firmata, analog pin values will be sent to the attached client at regular intervals that can be user configured. Unlike firmata, each service can have the interval set independently. Best efforts will be used to respect this interval and some sensors (such as the sonar distance sensor) with latency of 10 ms or more could result in the interval being greater than the value set. It is expected that intervals of at least 15 to 20 ms would be used in practice.

Digital pin state changes need to be reported as soon as quickly as possible so pins set for digital input are reported as soon as they are detected, although hardware functions that block (such as the sonar sensor) can delay this detection. In practice, the system should be as responsive as the current firmata implementation

**The pin level I/O protocol for the client to communicate with Arduino is as follows:**

The stream protocol consists of two single character tags separated by a comma followed by zero or more comma separated values. The first tag indicates the service class associated with the message, the second tag identifies the command or event within this service. See below for examples. Note that if reliable transport or connection support is required, these messages could be wrapped inside an appropriate transport protocol.

**Tags for the low level (pin I/O) service:**

IO\_SERVICE = 'I'; // tag indicating message is for I/O service

**Tags for service requests to Arduino:**

tag\_PIN\_MODE = 'P' // i/o request to Arduino to set pin mode

tag\_DIGITAL\_WRITE ='d' // i/o request to Arduino is digitalWrite

tag\_ANALOG\_WRITE ='a' // i/o request to Arduino is analogWrite)

**information requests to Arduino**

tag\_GET\_PORT\_TO\_PIN\_MAPPING = 'M'; // gets a list of pins associated with ports

tag\_GET\_ANALOG\_PIN\_MAPPING = 'm'; // gets a list of digital:analog pin associations

tag\_GET\_PIN\_MODES = 'p'; // gets a list of pin modes

tag\_GET\_PIN\_SERVICES\_LIST = 's'; // gets a list of pins indicating registered service

tag\_GET\_PIN\_CAPABILITIES = 'c'; // gets a bitfield array indicating pin capabilities

**IO events (messages from Arduino)**

tag\_PIN\_MODES = 'p'; // the event with a list of pin modes

tag\_PORT\_DATA = 'd'; // i/o event with data for a given digital port (tag changed from 'p' 24 June)

tag\_ANALOG\_VALUE = 'a'; // i/o event from Arduino is value of an analog pin

tag\_PIN\_CAPABILITIES = 'c'; // event providing a bitfield array indicating pin capabilities

tag\_PIN\_SERVICES\_LIST = 's'; // event providing a list of pins indicating registered service

**Header characters to indicate event message or error messages:**

SYSTEM\_MSG\_HEADER = '#'; // system requests are preceded with this tag

EVENT\_HEADER    = '@';  // event messages are preceded with this tag

ERROR\_MSG\_HEADER = '~'; // error messages begin with this tag

DEBUG\_MSG\_HEADER = '!'; // debug messages begin with this tag

**tags available to all services (Avoid using these for service specific functions)**

// request tags

tag\_AUTOEVENT\_REQUEST = 'A'; // this tag sets autoevent status

tag\_REMAP\_PIN\_REQUEST = 'M'; // for services that can change pin numbers (not

implemented in the initial release)

**Reply tags common to all services**

Tag\_SERVICE\_EVENT = 'e';

Querying pin capabilities and modes

The current mode of all pins can be queried by issuing the following request:

“I,p\n” (note lower case p)

A list of comma separated values is provided for each pin. These values are an enumeration defined in asip.h :

UNALLOCATED\_PIN\_MODE = 0,

INPUT\_MODE = 1, // digitial input

INPUT\_PULLUP\_MODE = 2, // digital input with pull-up resistors enabled

OUTPUT\_MODE = 3, // digital output

ANALOG\_MODE = 4, // analog input

PWM\_MODE = 5, //pwm output (analogWrite)

RESERVED\_MODE= 6, // pin is used by the server

OTHER\_SERVICE\_MODE = 7, // pin is used by a service

INVALID\_MODE = 8 // pin is not valid

For example, the following is the message where pins 0 & 1 are reserved (they are used for serial communication), and pins 4 & 5 are reserved by a service running on the server, the other pins are all available

@I,p,20,{6,6,0,0,7,7,0,0,0,0,0,0,0,0,0,0,0,0,0,0}

The capability of each pin can be queried by issuing the following request

“I,c\n”. An Uno board (or any board with a 328 chip) will reply with:

“@I,c,20,{1,1,1,5,1,5,5,1,1,5,5,5,1,1,3,3,3,3,3,3}\n”This indicates that there are 20 pins, all with digital IO capability, pins 3,5,6,9,10,11 are PWM capable, and pins 14 through 19 have analog input capability. The values are derived from a bitfield defined in asip.h:

DIGITAL\_IO = 1

ANALOG\_INPUT = 2

PWM\_OUTPUT = 4

The mapping of analog pin numbers to digital pin numbers can be queried by issuing the following request

“I,m\n”. An Uno board (or any board with a 328 chip) will reply with:

“@I,m,6,{14:0,15:1,16:2,17:3,18:4,19:5}\n”This indicates that there are 6 analog pins, digital pin 14 is mapped to analog pin 0, digital pin 15 to analog pin 1 etc.

pinMode()

Description: Set the mode for the given pin

API Syntax: PinMode(p,m);

Parameters:

p is the pin number (0-19 on Uno)

m is mode – the following modes can be requested:

INPUT\_MODE // digital input

INPUT\_PULLUP\_MODE // as above with pullups enabled

OUTPUT\_MODE // digital output

ANALOG\_MODE // analog input

PWM\_MODE // PWM output (as analogWrite)

The following modes are managed by the system

UNALLOCATED\_PIN\_MODE // this is the default mode for any pin not

// explicitly set to one of the other modes

RESERVED\_MODE // this pin is reserved for system use

OTHER\_SERVICE\_MODE // this pin is used by another service

INVALID\_MODE // the pin does not exist

Example API call (set pin 13 to OUTPUT): PinMode(13,OUTPUT);

stream protocol for this example: "I,P,13,3\n"

Pin requests are by their digital number, by default, requests will only succeed if the pin exists, supports the requested mode and is not reserved or allocated to a service.

digitalWrite()

Description: sets the given pin to the given state (if pin is set in OUTPUT mode)

API Syntax: DigitalWrite(p,s);

Parameters:

p is the pin number (0-19 on Uno)

s is the state (0-1)

Example API call (set pin 13 High) : DigitalWrite(13,1);

stream protocol for this example: "I,d,13,1\n"

analogWrite()

Description: sets the given PWM value on the given pin (if set to PWM mode)

API Syntax: AnalogWrite(p,s);

Parameters:

p is the pin number (must be a valid PWM pin set to PWM mode)

v is the value to write (0-255)

Example API call (set value on pin 9 to 512) : AnalogWrite(9,128);

stream protocol for this example: "I,a,9, 128\n"

Note there are no protocol messages to explicitly request sending of digital or analog values. Digital values for pins set as INPUT are sent to the client when a change is detected. A separate message is sent for each port that has changed data on a selected pin. Analog values for pins in ANALOG mode are sent at the interval determined by the ANALOG\_DATA\_REQUEST message, all selected analog channels are sent in a single message. Client requests are satisfied by reading locally cached values updated from these unsolicited messages. For example. To request events every 20 milliseconds, the stream protocol would be: “I,R,20\n"

A duration of 0 disables sending. The maximum interval is just over one minute (65535 milliseconds).

Example request selected analog pin data every 20 milliseconds

Stream protocol: “I,A,20\n”

Example turn off scheduled messages:

Stream protocol: “I,A,0\n”

**The pin level I/O protocol from the Arduino server to the client is as follows:**

sendDigitalPort()

Description: Internal method to send state of the given 8 bit port to the client

Syntax: SendPortData(p,v);

Parameters:

p is the digital port number (see firmata v2 docs for details)

v is a bitmap of the current state of all 8 pins on the given port

Example (send data on port 0 – pins 0-7) : SendDigitalPort(0, portData);

stream protocol for this example, (assume only pin 2 is high): “@I,p,0,4\n”

Note: ports are sent when one or more pins set to INPUT have changed state from the previous message for that port.

sendAnalog()

Description: Internal method to send the value of the given analog pin to the client

Syntax: SendAnalog(pin,value);

Parameters:

pin is the analog pin number (only pins set to ANALOG are sent)

value is a last analogRead value on the given pin

Example (send analog pin 3): SendAnalog(3,128)

Stream protocol for this example( “@Ia,3,128\n”

**High Level Services**

High level hardware devices are referenced by enumerations rather than actual pin numbers. The mapping between enumerations and pins is statically defined in a header file in the Arduino code. Services that provide data can be scheduled to send the current data at an interval that can be specified for each service.

Service event messages have the following comma separated format:

Event header character: ‘@’

Service Id that generated the event

The number of event fields

Opening curly brackets followed by the comma separated event data

Closing curly brackets followed by a newline character

For example, the IR service with 3 sensors sends the following (where the values are: 100,200, and 300):

Stream protocol for this example: “@R,e,3,{100,200,300}\n”

Services can have more than one sub-field for each instance, for example the encoder service has a pulse width and count for each wheel. These services separate two or more data sub-fields with a colon. For example, this is an encoder message where the counts are 110 & 120, pulses are 3000 & 3100):

“@E,e,2,{3000:110,3100:120}\n”

**Motor Service**

SetMotor()

Description: Control Motor speed and direction

API Syntax: setMotor(motor, speed);

Parameters:

motor is wheel id (0 or 1)

speed ( -100 to 100)

Example set motor 1 speed to 50): setMotor(1,50);

Stream protocol for this example: “M,m,1,50\n”

SetMotors()

Description: Controls both motor’s speeds and direction

API Syntax: setMotors(speed0, speed1);

Parameters:

speed0 ( -100 to 100)

speed1 ( -100 to 100)

Example set motor speeds to 45 and 50): setMotors(45,50);

Stream protocol for this example: “M,M,45,50\n”

StopMotor()

Description: stops the given motor

API Syntax: stopMotor(motor);

Parameters:

motor is Wheel id (0,1)

Example stop motor 0: stopMotor(0);

Stream protocol for this example: “M,s,0\n”

StopMotors()

Description: stops both motors

API Syntax: stopMotors();

Parameters: none

Example stop motors: stopMotors();

Stream protocol for this example: “M,S\n”

**Encoder Service**

RequestEncoders()

Description: Schedule the repeated reading and sending of pulse width and count for all encoders

API Syntax: RequestEncoders(interval)

Parameters: interval in milliseconds

Example request encoder data every 20 milliseconds

Stream protocol: “E,A,20\n”

Example turn off scheduled messages:

Stream protocol: “E,A,0\n”

encoderReply msgs

Description: informs count and duration for all encoders

Syntax: reply(,nbr,pulse0,count0, pulse1, count1)”

Parameters:

nbr is the number of encoders (currently 2)

pulse0 is the duration in microseconds for the first encoder

count0 is the count value for the first encoder

pulse1 is the duration in microseconds for the second encoder

count1 is the count value for the second encoder

Example event( encoder counts are 110 & 120, pulses are 3000 & 3100)

Stream protocol for this example: “@E.e,2,{3000:110,3100:120}\n”

Note that count in the current implementation is the ticks since the last reading. We may need to make the ticks accumulate if event messages could be skipped. Perhaps reset the count after a stop request is received?

**Bump sensor Service**

RequestBumpStates()

Description: Schedule the repeated reading and sending of digital switch states (bump switches)

API Syntax: RequestBumpStates(interval)

Parameters: interval in milliseconds

Example request data every 20 milliseconds

Stream protocol: “B,A,20\n”

Example turn off scheduled messages:

Stream protocol: “B,A,0\n”

bumpStateReply msgs

Description: internal method to send the states of all digital (bump) switches)

Internal function: SendBumps(nbr,state0,state1)

Parameters:

nbr is the number of bump switches (currently 2)

state0 is the state of the first bump switch

state1 is the state of the second bump switch

Example Bump0 is 0, bump1 is 1

Stream protocol for this example: “@B,e,2,{0,1}\n”

**IR Reflectance Sensor Service**

RequestIrSensors(interval)

Description: Schedule the repeated reading and sending of all IR sensor values

API Syntax: RequestIrSensors(interval)

Parameters: interval in milliseconds

Example request data every 20 milliseconds

Stream protocol: “R,A,20\n”

Example turn off scheduled messages:

Stream protocol: “R,A,0\n”

IrSensorReply msgs

Description: internal method to send the IR sensor values. IR emitters are automatically turned on before and off after readings)

Internal funcion: SendIrSensors(nbr, value0, value1,value2, value3)

Parameters:

nbr is the number of ir sensors (currently 3)

value0 is the value for the first ir sensor

value1 is the value for the second ir sensor

value2 is the value for the third ir sensor

Example sensors values: 100,200,300

Stream protocol for this example: “@R,e,3,{100,200,300}\n”

Other high level Arduino functions:

**Distance Sensor Service**

RequestDistance(interval)

Description: Schedule the repeated reading and sending of all distance sensor value

API Syntax: RequestDistance(interval)

Parameters: interval in milliseconds

Example request data every 20 milliseconds

Stream protocol: “D,A,20\n”

Example turn off scheduled messages:

Stream protocol: “D,A,0\n”

MeasureDistance()

Description: Takes a single reading for all distance and sends the value

API Syntax: Measure(interval)

Parameters: interval in milliseconds

Example

Stream protocol: “D,M\n”

distanceReply msgs

Description: internal method to send the distance sensor readings.

API Syntax: “sendDistance(nbr, value0);

Parameters:

nbr is the number of distance sensors (currently 0 or 1)

value0 is the value for the first sensor (if any)

Example distance is 25

Stream protocol for this example: “@D,e,1,{25}\n”

**Servo Service**

SetServo ()

Description: Set the position of the given servo

API Syntax: SetServo(id,angle)

Parameters:

id is the servo enumeration (0 is the first servo)

angle is the desired angle in degrees (0-180)

Example set servo 0 to 90 degrees

Stream protocol for this example: “S,W,0,90\n”

**Error messages**

Request messages that cannot be handled will result in a message as follows:

The ERROR\_MSG\_HEADER character

The command string that was invalid, terminated with a colon

An error number indicating the nature of the error,

A string that described the error, terminated with a newline character

Currently defined constants for errors include:

ERR\_INVALID\_SERVICE, // the service requested does not exist

ERR\_UNKNOWN\_REQUEST, // the service does not support this request

ERR\_INVALID\_PIN, // the given pin is not valid for the given request

ERR\_MODE\_UNAVAILABLE, // the pin is reserved or in use by another service

ERR\_INVALID\_MODE, // the mode does not exist

ERR\_WRONG\_MODE, // the current pin mode does not support this request

ERR\_INVALID\_DEVICE\_NUMBER, // the service does not have this many devices

ERR\_DEVICE\_NOT\_AVAILABLE //the service does not have a device available

Example Request AnalogWrite on pin 9 not set to PWM mode:

Request: “I,a,9,128\n”

Error message: ”~I,a:6(WRONG\_MODE)\n”

**System functions**

GetInfo – returns:

protocol version (major and minor)

Arduino chip ID (currently only 328 and 644/1284 are supported)

Board sketch name – currently: Mirtle

If needed , the following information may be provided:

Board sketch version (major and minor number)

Arduino board unique identifier number stored in Arduino EEPROM (each board programmed with a unique ID number)

Stream protocol for the request: “#,?\n”Example reply (version 0.1 running on Uno board): “@#,?,0,1,ATmega328p,Mirtle\n”

System clear – un-allocates all pins and turns off PWM and digital outputs. This function may not be implemented

**Notes on implementation**

All service classes are derived from an abstract base class that specifies the required methods and members that all services must provide. This includes:

* A constructor that is given the character constants that identify this service
* A begin method that is given an array of pin numbers used by this service. The pin modes and other service initialization is handled here. For I2C services that do not use dedicated pins, a begin method without pin arguments is supported.
* A method to report all events for this service
* A method to report a single event, called by the above for each service instance.
* A method to turn on auto reporting of events
* A method to process message requests for this service.

**Debugging support**

A printf function can be enabled to send debug information to the Serial port by defining the PRINTF\_DEBUG identifier. Debug strings must be 62 bytes or less in length.

**Common tasks:**

**Modify pin numbers (TODO)**

**Add a service to a sketch (TODO)**

**Create a new service class**

Define the class using asipServiceClass as a base – see OtherServices.h for examples

Implement the service specific methods for handling requests and generating event messages– see

OtherServices.cpp as an example. If the service does not have an event then override the reportValues() method (see the servo service as an example)

Define a character to identify this service. The character must be different from all other service identifiers and must be an upper case letter from ‘A’ through ‘Z’ inclusive.

Add the following code to the main sketch – see mirtle.ino for an example

Define an array of pins of type pinArray\_t that is used by this service

Create (instantiate) the service

Add the reference to the service in the services array

In setup(), call the service begin method – the arguments are:

Number of devices, total number of pins used by all devices, the array of pin numbers