Towards Cyber-Physical Systems as Services: the ASIP Protocol

Mirco Bordoni*, Michele Bottone†, Bob Fields†, Nikos Gorogiannis†,
Michael Margolis†, Giuseppe Primiero†, Franco Raimondi†
*Ocado Group, Hatfield Business Park, Hatfield, UK
Email:mirco.bordoni@ocado.com
†School of Science and Technology, Middlesex University, London
Email: {m.bottone, b.fields, n.gkorogiannis, m.margolis, g.primiero, f.raimondi}@mdx.ac.uk

***Abstract*—The development of Cyber-Physical Systems needs to address the heterogeneity of several components that interact to build a single application. In this paper we present a model to enable easy integration and interaction of micro-controllers. Specifically, we describe the Arduino Service Interface Protocol (ASIP), we provide an implementation and client libraries for Java, Racket and Erlang, together with the description of a practical example.**

I. Introduction and related work

Cyber-Physical Systems (CPS) are typically built using multiple interacting components, including sensors for a range of parameters (temperature, acceleration, pressure, humidity, etc) and actuators (such as servo motors, relays, stepper motors, etc.). These components may be connected to micro-controllers that need to be programmed individually to take into account both low-level implementation details and the high-level requirements of the application of which the microcontrollers are part. Several solutions have been put forward in the past to address the issue of modeling CPS and abstracting low-level implementation details, see for instance [1] and references therein. In terms of concrete applications, the Robot Operating System (ROS) [2] is one example of an abstraction mechanisms for concrete hardware implementations. ROS can be seen as middleware that enables the integration of different platforms with support for a range of programming languages. However, ROS targets mainly robotic platforms and the hardware requirements for a minimal installation are beyond the capabilities of microcontrollers such as Arduino boards. Even a more powerful board such as the Raspberry Pi presents a number of performance issues with ROS. TinyOS [3] is another example of operating system developed with a specific focus on wireless sensor networks; a typical TinyOS application requires approximately 30 Kb of flash space, making it appropriate for a range of motes, but it may still be too large for some Arduino boards, such as the common Arduino Uno.

Instead of relying on a dedicated operating system, in this paper we propose that sensors and actuators are exposed by microcontrollers as *services*, so that more complex software applications can be built by composing them. Concretely, in this paper we show how this can be achieved on Arduino micro-controllers by means of the Arduino Service Interface Protocol (ASIP). The ASIP protocol is similar, in spirit, to the Firmata protocol [4] in that it enables a computer to discover, configure, read and write a microcontrollers general purpose IO pins. However, ASIP has a smaller footprint than Firmata (using around 20% less RAM). And uniquely, it supports high level abstractions that can be easily attached to hundreds of different services for accessing sensors or controlling actuators. These abstractions can decouple references to specific hardware, thus enabling different microcontrollers to be used without software modification. Although ASIP is currently implemented for Arduino boards, the protocol is hardware agnostic. We provide an overview of the ASIP service model in Section II and we describe the actual protocol with clients for Java, Racket and Erlang in Section III. We provide a concrete example application in Section IV.

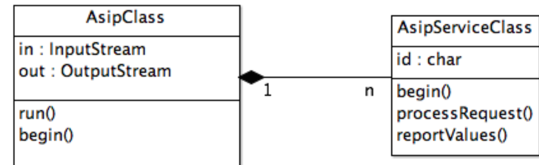

Fig. 1. The overall ASIP architecture

II. The ASIP service model

Figure 1 depicts the high-level architecture of ASIP. An `AsipClass` models the core functionality that should be implemented by a micro-controller. In particular, an implementation of an `AsipClass` should include a `run()` method (described below), input and output streams for data exchange (these could be a serial port, a network or Bluetooth connection, etc.), and an initialisation mechanism. More importantly, an `AsipClass` is composed of one or more *services*, modelled by means of an `AsipServiceClass`. Examples of services offered by a micro-controller are a set of motors connected to wheels and sensors for: distance, temperature, acceleration, heading etc.

A. The Main ASIP loop

After an initialisation step to set up the input and output streams for communication, the implementation of an

`AsipClass` executes the `run` method in which the following operations are performed:

- A listener listens to incoming messages on the input stream and dispatches the messages to the appropriate registered service.
- In case of system messages (such as a request for the list of active services, or for remapping of pin resources) the `run()` method invokes the appropriate methods.
- System messages, such as error messages, periodic status (if enabled), etc. are written to the output stream at the appropriate time intervals (which can be reconfigured).
- Services are queried for periodic messages to be written to the output stream, for instance to continuously report the value of analog pins or sensor services.

Essentially, the main ASIP loop acts as a dispatcher of messages to and from registered services and performs appropriate error checking.

B. Analysis of an ASIP Service

An ASIP service abstracts some functionality of a micro-controller. Examples of services are servo motors, distance sensors, pair of wheels with quadrature encoders for distance, etc. Each service is characterised by a unique ID and must implement a `processRequest()` method. This is the method that processes the messages dispatched by the `run()` method of the main `AsipClass`. A service can implement a `reportValues()` method if it needs to report values (either continuously or upon request). For instance, a distance sensor service could be configured to report distance every 100 ms, while a servo motor service may not need to report values at any time.

In our implementation (described below) we treat standard pin-level operations as a single service, the *Input-Output Service*. This service includes operation to set pin modes at run-time; currently, the following modes are supported: digital and analog output, digital and analog input, input with pull-up resistor and Pulse-Width Modulation (PWM) mode. The service also supports configuration of the reporting interval for values of analog pins and implements error checking for valid bounds in the arguments sent and received.

The access to pins by means of a service gives developers the option of choosing how to implement a specific application. For instance, consider an ultrasound distance sensor attached to the analog pin of a micro-controller: in this case, the developer has two options to measure distance: (1) read the value of the analog pin using the Input-Output Service, or (2) write a service that reserves a specific pin and calculates and reports the distance, either upon request or continuously. In the latter case, the computation of the distance as a function of the value of the analog pin can be delegated to the micro-controller. The choice of which implementation is more appropriate depends on several factors, including the need for accurate timing that can be achieved by the micro-controller, or the memory requirements of the overall application on a specific micro-controller. In some other cases the implementation of a service is influenced by the availability of dedicated libraries. For example, motors with quadrature encoders may be supported through a manufacturer-supplied library. In this case the main `AsipClass` delegates the interaction with the encoders to the appropriate service (a motor service, in this case). As described in the following section, the interaction with a main `AsipClass` and with its services happens by means of plain text messages exchanges through the input and output streams.

III. The ASIP protocol

ASIP messages are exchanged over the input and output streams as ASCII text strings. Individual messages are terminated by the new line character and fields are usually separated by means of commas. Messages *from* an `AsipClass` implementation begin with one of the following reserved characters: `@`, `~` and `!`, where `@` denotes the beginning of a standard event message, `~` is used to start an error message and `!` is used to start a debug or info message. The start of an event message is usually followed by a service ID and by the actual message from the service. For instance, the following message reports the values of the analog pins on an Arduino board with 6 analog inputs:

```
@I,a,6,{0:320,1:340,2:329,3:200,4:129,5:450}
```

This message from the implementation of the `AsipClass` notifies a client that the Input-Output service (that has ID I) is sending a message. In this specific case, this is a message about analog pins (hence the `a` character after the first comma), saying that there are 6 analog pins whose input values are listed in brackets in the form pin:value.

Messages *to* an `ArduinoClass` implementation start with the ID of the service to be invoked, followed by appropriate parameters, typically a first character encoding a command for the service, followed by optional values. For instance, this is a sequence of two messages to set pin 13 to output mode and then to write a value of 1 to pin 13, thus turning on an LED, if connected:

```
I,P,13,3
I,d,13,1
```

In the first message, command `P` for the Input-Output service sets pin 13 to mode 3 (OUTPUT) mode; in the second message, command `d` writes the value 1 to pin 13. Commands available for the Input-Output service include reading and writing digital and analog pins, setting pin modes, requesting pin modes, etc. The full list of commands, pin modes, and additional messages for the Input-Output service are available in the file `asipIO.h` available at https://github.com/michaelmargolis/asip.

Messages for other kinds of services have the same structure. For instance, the message `D,A,100` to an implementation of the `AsipClass` sets the auto-reporting interval for a service with ID `D` to 100 ms (this is normally a distance service). The `AsipClass` implementation could then reply every 100 ms with messages of the form: `@D,e,2,{42,118}` This message encodes the fact that the distance service `D` is

reporting a periodic event `e` with two values (which means that the board is equipped with 2 distance sensors): the first distance sensor measures 42 cm and the second measures 118 cm. Analogously, the message `M,m,0,190` sends the command `m` to service `M` with parameters 0 and 190. This message is currently used to set the speed of a motor service `M` (motor 0 in this case) to the value of 190, where the max value is normally 255.

The code available at https://github.com/michaelmargolis/asip contains an implementation of `AsipClass` for Arduino boards with serial connections, either through a USB cable or using the pins 0 and 1 for serial communication. The repository contains code for the Input-Output service, for ultrasound distance sensors, for IR sensors (used for line following), for HUB-ee wheels [5] (including their quadrature encoders), and for NeoPixels strips [6], in addition to a base class that can be easily extended to incorporate new services just by providing an implementation for their `processRequest()` and `reportValues()` methods. The code is an Arduino library that can be incorporated into an Arduino sketch and uploaded to an Arduino board. Several sketch examples are available in the repository. The library has been tested on a range of boards, including: Uno, Mega, Leonardo, Micro, Mini Pro 3.3 V 8 MHz. It has also been tested on a bespoke board based on the Atmel ATmega328 micro-controller and used for controlling a robot (see Section IV). After installing the ASIP library, an Arduino board keeps listening to incoming messages and dispatches requests to the appropriate service. For testing purposes, interaction with the board can happen from the command line; however, we have developed libraries to parse the text messages of the ASIP protocol for several common programming languages: Java, Racket (a dialect of LISP) and Erlang. A Python implementation is currently under development.

A. The ASIP Java Client

The Java library for the ASIP protocol is available at https://github.com/fraimondi/java-asip. It contains abstract classes for ASIP services and the implementation of concrete classes for the services described above (Input-Output, distance, motor with encoders, Infra-red sensors for line following, and NeoPixels). The implementation relies on the Java Simple Serial Connector library available at https://code.google.com/p/java-simple-serial-connector/, but serial communication can be easily replaced by any input/output stream, for instance over a TCP connection. The repository provides a simple client implementation for an Arduino board with just the Input-Output service, available in the `SimpleSerialBoard` class. Objects of this class can be created by passing the name of the USB port to which the Arduino board is attached, for instance:

```
SimpleSerialBoard myBoard1 = new
    SimpleSerialBoard("COM5");
SimpleSerialBoard myBoard2 = new
    SimpleSerialBoard("COM10");
if (myBoard1.digitalRead(5) == AsipClient.HIGH)
 myBoard2.digitalWrite(13,AsipClient.HIGH);
```

In the example above two objects are created, corresponding to two boards attached to the machine running the Java code. The code then checks whether pin 5 is reading a value HIGH (which is a static constant defined in the class `AsipClient`, and also assumes that this pin was set to INPUT mode using the appropriate method provided by `SimpleSerialBoard`). If this is the case, then the code writes the value HIGH to pin 13 of the *other* board.

B. The ASIP Racket Client

Racket [7] is a LISP dialect used for teaching purposes in a number of institutions worldwide. It is the only language taught at Middlesex University for the first year of the Computer Science degree and it has been employed in conjunction with Arduino and Raspberry Pi boards in Physical Computing sessions for the past two years [8]. The Racket ASIP client library is available at https://github.com/fraimondi/racket-asip together with an implementation for the services described above (Input-Output, distance, motor with encoders, Infra-red sensors for line following, and NeoPixels). The following is an example of Racket code to set pins 11, 12 and 13 to HIGH:

```
(map (lambda (x) (digital-write x HIGH))
     (list 11 12 13))
```

The code above makes use of the high-order function `map` that takes as its first argument a function (a lambda-function in this case, which given an integer x applies the ASIP library function `digital-write` to x) to the list of number 11, 12 and 13. As shown in this example, Racket provides an opportunity to teach functional programming languages in physical computing sessions.

C. The ASIP Erlang Client

Erlang [9] is a programming language originally developed by Ericsson and focussed on the development of distributed applications. Erlang is used in a second year course at Middlesex University to teach Networking and Distributed Systems. The ASIP Erlang client allows the development of distributed applications running, for instance, on a network of Raspberry Pi, each of which is connected to one or more micro-controllers running an implementation of `AsipClass`. The Erlang ASIP client is available at https://github.com/ngorogiannis/erlang-asip, together with some examples. The client supports the Input-Output service described above (the other services are currently under development). The following is an excerpt of Erlang code turning on a LED attached to pin 13 when a button is pressed on pin 2 (declared as input pull-up):

```
-define(LED1, 11).
-define(inputPin, 2).

CurInput = asip:digital_read(?inputPin),
case CurInput of
  0 ->
    %% it's a pull-up, so LOW means pressed
    asip:digital_write(?LED1, 1),
  _ ->
    asip:digital_write(?LED1, 0),
end
```

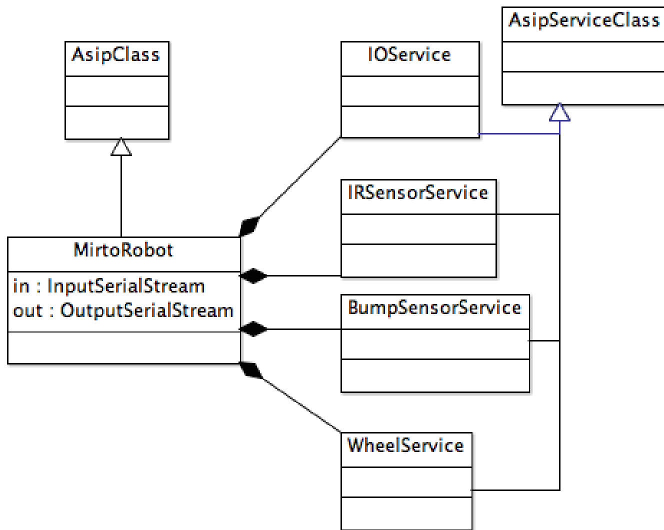

Fig. 2. The ASIP architecture for a Robot

IV. Application: controlling a Robot

In this section we describe how the ASIP protocol and its implementation can be used to control a robot. Middlesex University has developed the Middlesex Robotic plaTfOrm (MIRTO), described in more detail in [8]. Essentially, this is a robotic platform composed of an Arduino layer controlling two HUB-ee wheels, 3 infra-red sensors for line following, and two bump sensors. The Arduino layer is connected to a Raspberry Pi using the serial GPIO pins. Figure 2 shows the class diagram for the implementation of the `AsipClass` in a sub-class called `MirtoRobot`. Objects of this class comprise 4 services, each of which is a sub-class of the generic `AsipServiceClass`: the standard Input-Output Service, an IRSensorService to control the 3 IR sensors, a BumpSensorService for the two bumpers and a WheelService to control the two wheels (both the speed of the motors and the values of the quadrature encoders to measure the actual distance travelled). The implementation of the `MirtoRobot` class is available at https://github.com/fraimondi/java-asip in the file `JMirtoRobot.java`. This class exposes several methods, including `setMotors(int,int)` to set the two wheels at a certain speed, `getIR(int)` to get the value of an IR sensor, and the boolean function `isPressed(int)` to check whether a bump sensor is pressed. These methods allow students to develop a PID algorithm for line following focusing only on the logic of the application. The following is an excerpt of the full application available in the file `AsipMirtoPIDFollower.java` at the link reported above.

```
// This code runs on a Raspberry Pi
JMirtoRobot robot = new JMirtoRobot("/dev/ttyAMA0");

// The core PID loop
while (true) {
  int leftIR = robot.getIR(2);
  int middleIR = robot.getIR(1);
  int rightIR = robot.getIR(0);

  // the function computeError computes the error,
    i.e. the displacement from the line to follow
  int curError =
    computeError(leftIR,middleIR,rightIR,prevError);

  // the function computeCorrection implements the
    PID error correction
  correction =
    computeCorrection(curError,prevError);

  // This function computes the new speed
  int newSpeedLeft = computeLeftSpeed(correction);
  int newSpeedRight = computeRightSpeed(correction);

  // Finally, set the new motors speed
  robot.setMotors(newSpeedLeft,newSpeedRight);
} // end of core PID loop
```

A video of this example is available at http://youtu.be/KH_3766gNcM. This example shows that the overhead introduced by the ASIP protocol does not affect the line following capabilities of this robot (the error is computed every 20 ms).

V. Conclusion and future work

In this paper we have presented how sensors and actuators attached to micro-controllers can be modelled with the notion of high-level services, abstracting away many implementation details that make the integration of heterogeneous components a difficult task. We have described the implementation of the ASIP protocol and of three software libraries, describing an example application (a PID line follower on a robot). We are currently working at several extensions of this work: both the Arduino and the client code (Java, Racket, Erlang) could be generated automatically from a model like the one in Figure 2, by re-using the implementation of standard services, such as IR sensors, distance sensors, etc. Automatic code generation has the additional benefit of enabling model-based testing and, ultimately, it may ease the certification process when applications need to be employed in industrial settings.

References

[1] P. Derler, E. A. Lee, and A. S. Vincentelli, "Modeling Cyber–Physical Systems," *Proceedings of the IEEE*, vol. 100, no. 1, pp. 13–28, 2012.

[2] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "ROS: an open-source Robot Operating System," in *ICRA workshop on open source software*, vol. 3, no. 3.2, 2009, p. 5.

[3] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, and D. Culler, "TinyOS: An operating system for sensor networks," in *Ambient Intelligence*. Springer Berlin Heidelberg, 2005, pp. 115–148.

[4] H.-C. Steiner, "Firmata: Towards making microcontrollers act like extensions of the computer," in *New Interfaces for Musical Expression*, 2009, pp. 125–130.

[5] "HUB-ee Robot Wheels with Quadrature Encoders," http://www.creative-robotics.com/About-HUBee-Wheels, accessed: 27/01/2015.

[6] "NeoPixels light strips for Arduino boards," https://learn.adafruit.com/adafruit-neopixel-uberguide/arduino-library, accessed: 27/01/2015.

[7] "The Racket Programming Language," http://www.racket-lang.org/, accessed: 27/01/2015.

[8] K. Androutsopoulos, N. Gorogiannis, M. Loomes, M. Margolis, G. Primiero, F. Raimondi, P. Varsani, N. Weldin, and A. Zivanovic, "A Racket-Based Robot to Teach First-Year Computer Science," in *7th European Lisp Symposium*, 2014, p. 54.

[9] "The Erlang Programming Language," http://www.erlang.org/, accessed: 27/01/2015.