BAE SYSTEMS
INSPIRED WORK

EmbedWithECOA

Introduction

This document describes an ECOA® example of using the ECOA for an embedded program.

This document presents information about the principal user generated artefacts required to create an "*EmbedWithECOA*" program using the ECOA. It is assumed that the reader is *thoroughly* conversant with the ECOA Architecture Specification (ref.[1]) and the process of defining and declaring ECOA Assemblies, ASCs (components), Modules, and deployments in XML, and then using code generation to produce Module framework (stub) code units and ECOA Container and Platform code. If not, then let me suggest working through some of the other examples/samples provided, starting with "*Hello World*" and working your way up to "*Pub Sub*".

The ECOA has been predominantly designed with complex aerospace Mission System software in mind, executing on a modern operating system or runtime kernel, on high performance multi-processing hardware. But what about the other end of the computing spectrum? Can the ECOA be used, and bestow benefits, to small scale, embedded, applications – such as a washing machine controller.

This document will explore this realm by describing an example application that can be programmed into a microcontroller chip interfaced to "real world" push-buttons and lights.

Aims

This ECOA *EmbedWithECOA* example is intended to show how ECOA ASCs (Components) can be used for small-scale embedded programs whilst still achieving the benefits expounded for the ECOA (application code reusability, ease of integration, etc.).

The example is also intended to show the explicit invocation of ECOA Operations without breaking Inversion of Control (ref.[1]).

ECOA Features Exhibited

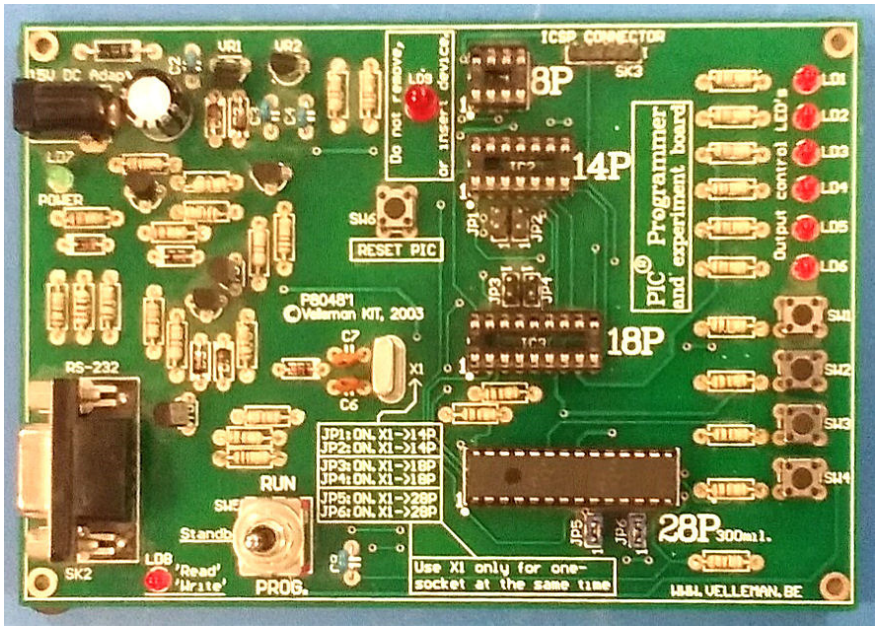
- The ECOA External Interface capability.
- Separation of functional behaviour implementation (in an ECOA ASC) from platform specific inter-process and interfacing code.
- Comparison of ECOA ASCs running in a microcontroller chip and in a general purpose computer (with an operating system).

Design and Definition

System Design

The widely available Velleman VM111 Programmer and Experiment Board (ref.[2]) as well as allowing a microcontroller to be programmed easily, provides four push-buttons and six LED lights that interface directly to a PIC microcontroller (e.g. ref.[3]). With a suitably programmed chip, it therefore provides an easy to use demonstration of software-hardware interfacing.

Figure 1 Vellemen VM111 Programmer & Experiment Board

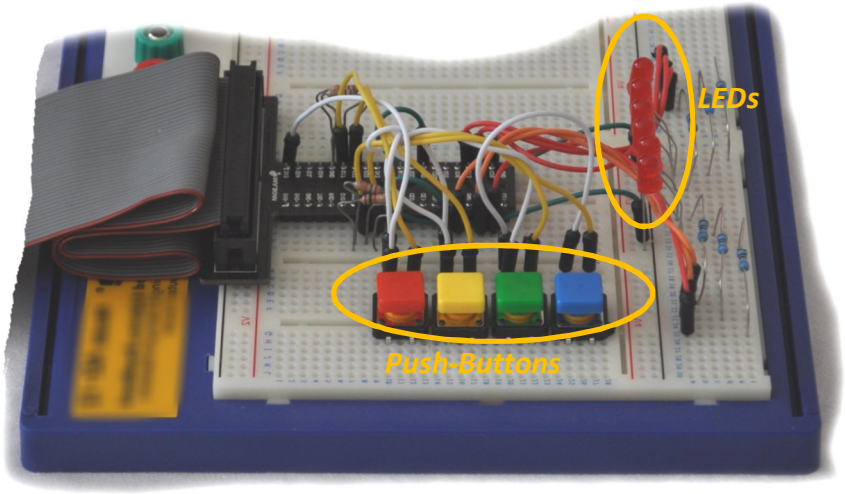


Each LED is driven from an output pin of the microcontroller (through a current limit resistor. When the pin is set to “on” (5v) the LED comes on; when the pin is set “off” (0v) the LED is extinguished.

Each push-button, when pressed, pulls an input pin of the microcontroller to 5v (“on”) (again through a current limit resistor). When the push-button is un-pressed, the input pin is pulled to 0v (“off”).

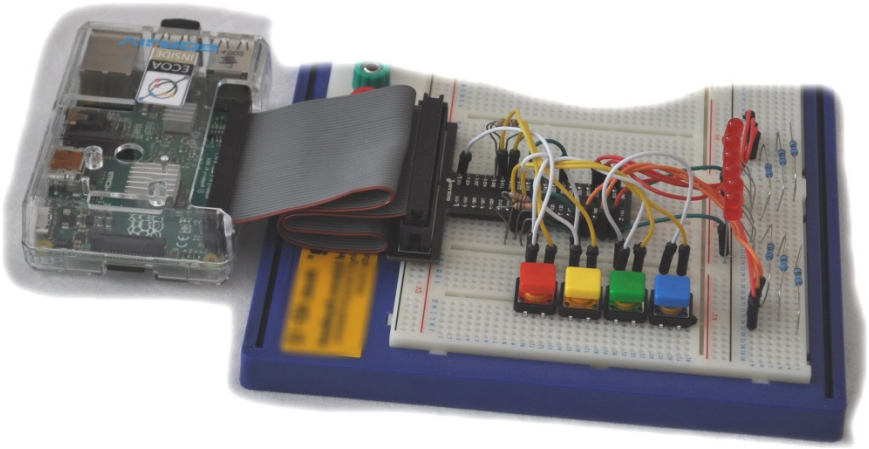
In order to demonstrate the software application behaviour in the microcontroller AND in a general purpose computer (with an operating system/runtime kernel), it is necessary to interface that computer to the same hardware configuration. This can be done replicating the push-buttons and lights using a prototyping breadboard, interfaced to the computer. In Figure 2 the four push-buttons are horizontal along the front of the board (with red, yellow, green, and blue caps) and the six red LEDs sit vertically to the right of the push-buttons.

Figure 2 Prototyping Breadboard Push-Buttons-and-LEDs



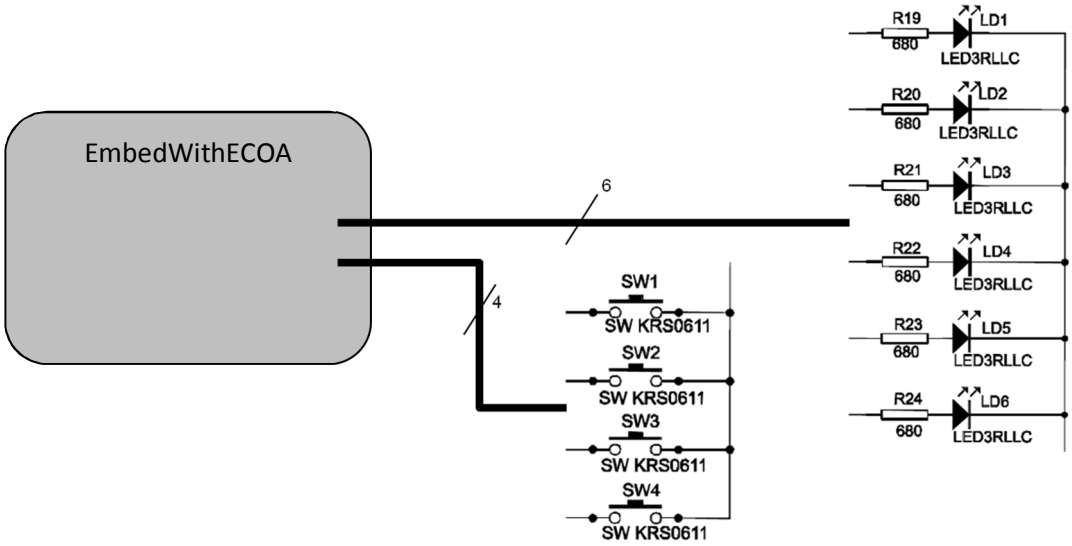
This example was developed using a Raspberry Pi (e.g. ref.[4]) as the "general purpose computer", interfaced to the breadboard using a T-Cobbler Breakout (e.g. ref.[5]).

Figure 3 Raspberry Pi and Breadboard



So we now need a software *EmbedWithECOA* application that we can sit in the microcontroller and the computer and have do something with the LEDs and push-buttons:

Figure 4 "*EmbedWithECOA*" Application



That application will simply sequence the LEDS in one of four patterns, a pattern selected by the four push-buttons:

Pattern A: Each LED will turn on and then off again in the sequence 0,1,2,3,4,5,0,1,2,3,4,5,0,1,2,3,4…

Pattern B: The LEDs will light in a binary count from 0 (no LEDs on) to 63 (all LEDs on), and then repeat from 0.

Pattern C: LEDs 2 and 3 will go on then off, then LEDs 1 and 4 will do the same, and then LEDs 0 and 5. The pattern then repeats. The light therefore starts in the middle of the array an moves out to the edges, and then repeats.

Pattern D: Each LED will turn on and then off again in sequence 0,1,2,3,4,5,4,3,2,1,0,1,2,3,4,5,4,3,2…

ECOA Assembly Design and Definition

In order to isolate the hardware (push-buttons and LEDs) interfacing from the application logic (the sequencer) the *EmbedWithECOA* application will comprise two ECOA ASCs, "*EwECOA*" and "*EwECOAIF*". The *EwECOA* ASC will be completely independent of the execution hardware – it will be a "pure" ECOA ASC. All the interfacing will be in the *EwECOAIF* ASC. A single ECOA Service is defined, "*IO*", provided by the interface ASC.

Figure 5 ECOA "*EmbedWithECOA*" Assembly Diagram

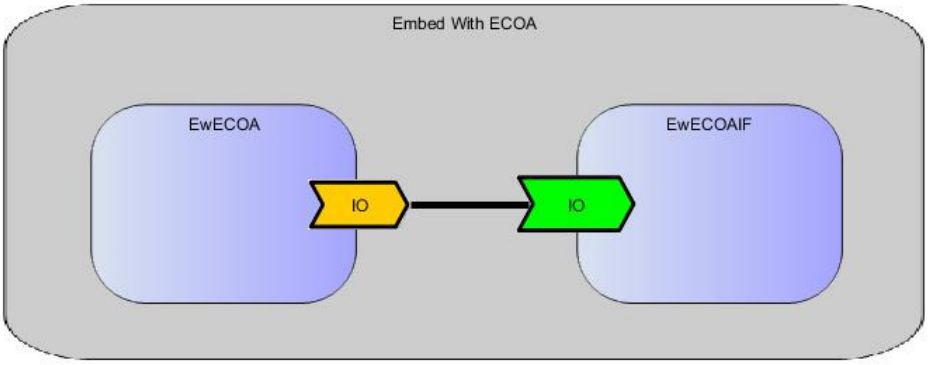
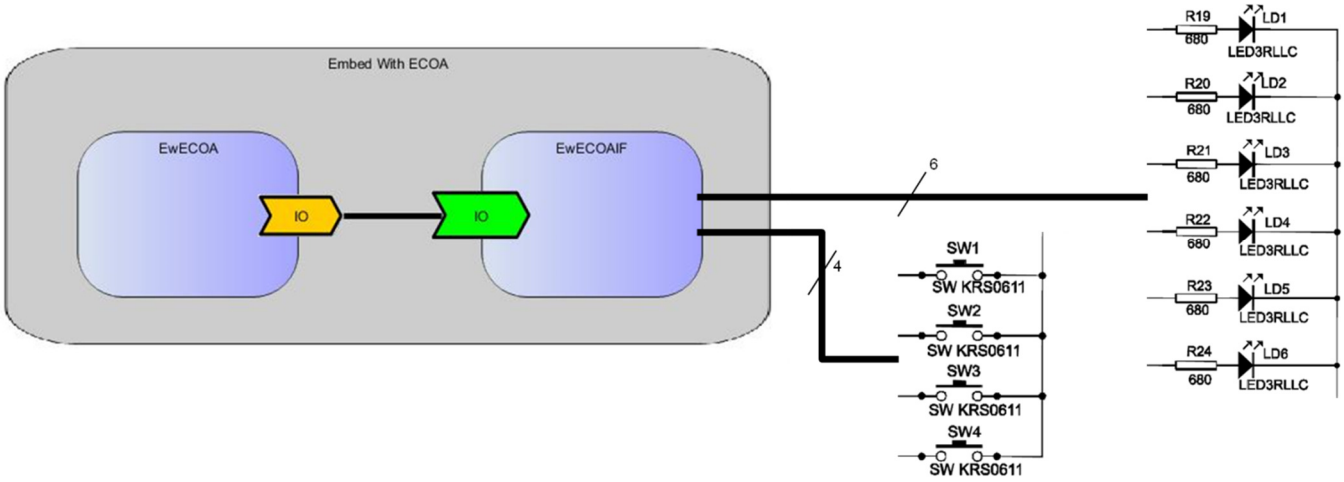


Figure 6 "*EmbedWithECOA*" Assembly in Context



The ECOA Assembly is defined in an Initial Assembly XML file, and declared in a Final Assembly (or Implementation) XML file (which is practically identical). The Final Assembly XML for the ECOA *EmbedWithECOA* Assembly is as follows (file *EwECOA.impl.composite*):

```
<csa:composite xmlns:csa="http://docs.oasis-open.org/ns/opencsa/sca/200912"
               xmlns:ecoa-sca="http://www.ecoa.technology/sca-extension-2.0"
               name="EwECOA" targetNamespace="http://www.ecoa.technology">
  <csa:component name="EwECOA">
    <ecoa-sca:instance componentType="EwECOA">
      <ecoa-sca:implementation name="EwECOA"/>
    </ecoa-sca:instance>
  </csa:component>
  <csa:component name="EwECOAIF">
    <ecoa-sca:instance componentType="EwECOAIF">
      <ecoa-sca:implementation name="EwECOAIF"/>
    </ecoa-sca:instance>
  </csa:component>
  <!-- System Wiring...  -->
  <csa:wire source="EwECOA/IO" target="EwECOAIF/IO"/>
</csa:composite>
```

I will spare much detail and repetition in this description of material presented with other examples, trusting you to have worked your way through other ECOA examples.

ECOA Service and Types Definition

The IO Service, which is provided by the *EwECOAIF* ASC and referenced by the *EwECOA* ASC, is (of course) defined in a XML file (*IO.interface.xml*):

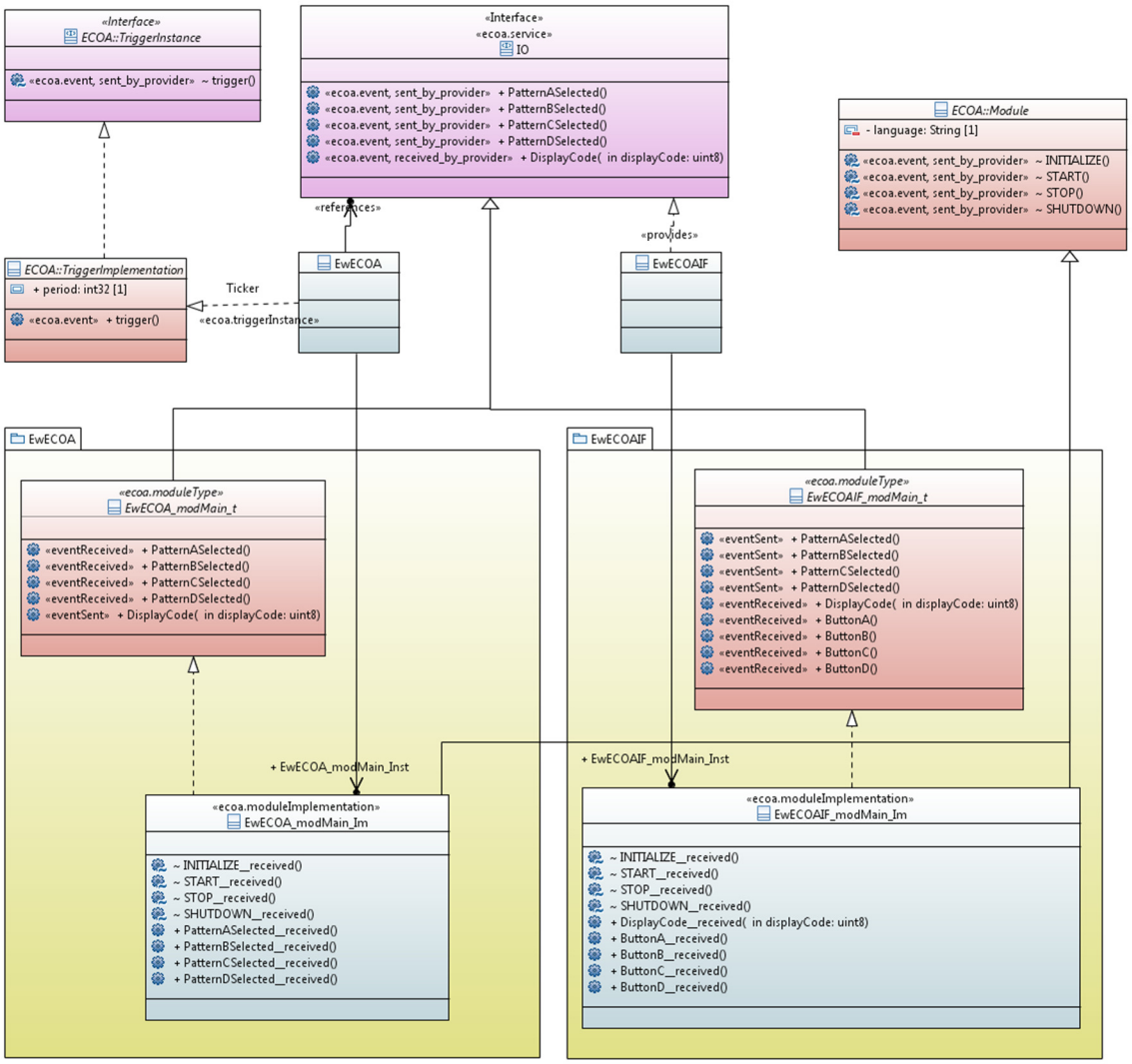
```
<serviceDefinition xmlns="http://www.ecoa.technology/interface-2.0">
  <operations>
    <event name="PatternASelected" direction="SENT_BY_PROVIDER"/>
    <event name="PatternBSelected" direction="SENT_BY_PROVIDER"/>
    <event name="PatternCSelected" direction="SENT_BY_PROVIDER"/>
    <event name="PatternDSelected" direction="SENT_BY_PROVIDER"/>

    <event name="DisplayCode" direction="RECEIVED_BY_PROVIDER">
       <input name="displayCode" type="uint8"/>
    </event>

  </operations>
</serviceDefinition>
```

It defines four ECOA Event "*SENT_BY_PROVIDER*" Operations which tell the sequence controller recipient (i.e. *EwECOA*) that a push-button has been pressed, selecting a new pattern. It also defines a "*RECEIVED_BY_PROVIDER*" Event Operation by which the sequence controller (*EwECOA*) requests a particular LED light *displayCode* pattern. The *displayCode* is given as an 8-bit unsigned integer (byte), of which bits 0 to 5 will be mapped onto the 6 LEDs.

ECOA Module Design and Definition

The *EwECOA* and *EwECOAIF* ASC (component) types are composed of a single ECOA Module each (Module Implementations *EwECOA_modMain_Im* and *EwECOAIF_modMain_Im* of Module Types *EwECOA_modMain_t* and *EwECOAIF_modMain_t* respectively) as illustrated in UML in Figure 5. Here is depicted in UML the *EwECOAIF* ASC (component) ***providing*** the *IO* ECOA Service, whilst the *EwECOA* ASC ***references*** the Service. As always in the ECOA, the Module Implementations implement the Module Lifecycle operations defined by the ECOA (as represented in UML by the abstract class *ECOA::Module*).

Figure 7 "*EmbedWithECOA*" Module Design (as UML Class Diagram)



The *EwECOA* ASC

The *EwECOA* ASC is defined as a normal ECOA Component Implementation, with Module Operations defined in a Module Type (*EwECOA_modMain_t*) declaration:

```
<moduleType name="EwECOA_modMain_t" hasUserContext="true"
            hasWarmStartContext="false">
    <operations>
        <eventReceived name="PatternASelected"/>
        <eventReceived name="PatternBSelected"/>
        <eventReceived name="PatternCSelected"/>
        <eventReceived name="PatternDSelected"/>
        <eventSent name="DisplayCode">
            <input name="displayCode" type="uint8"/>
        </eventSent>
        <eventReceived name="Tick"/>
    </operations>
</moduleType>
```

The Module Type is implemented by the concrete Module Implementation *EwECOA_modMain_Im* (depicted in the UML expanded in the form of the code class produced by the code generation process), which in turn is instantiated at runtime as the Module Instance *EwECOA_modMain_Inst*.

The *IO* Service Operations are linked to Module Operations with <*eventLink*> XML tags, e.g.:

```
<eventLink>
    <senders>
        <reference instanceName="IO" operationName="PatternASelected"/>
    </senders>
    <receivers>
        <moduleInstance instanceName="EwECOA_modMain_Inst"
                        operationName="PatternASelected"/>
    </receivers>
</eventLink>
```

The Module Type also declares a *Tick* operation which is linked to an ECOA Trigger Instance (*Ticker*), and is called periodically (every 0.1 seconds):

```
<eventLink>
    <senders>
        <trigger instanceName="Ticker" period="0.1"/>
    </senders>
    <receivers>
        <moduleInstance instanceName="EwECOA_modMain_Inst"
                        operationName="Tick"/>
    </receivers>
</eventLink>
```

The *EwECOAIF* ASC

The *EwECOAIF* ASC is also defined as an ECOA Component Implementation, with Module Operations defined in a Module Type (*EwECOAIF_modMain_t*) declaration:

```
<moduleType name="EwECOAIF_modMain_t" hasUserContext="false"
            hasWarmStartContext="false">
    <operations>
        <!-- Service Operations -->
        <eventSent name="PatternASelected"/>
        <eventSent name="PatternBSelected"/>
        <eventSent name="PatternCSelected"/>
        <eventSent name="PatternDSelected"/>
        <eventReceived name="DisplayCode">
            <input name="displayCode" type="uint8"/>
        </eventReceived>
        <!-- Interface operations -->
        <eventReceived name="ButtonA" />
        <eventReceived name="ButtonB" />
        <eventReceived name="ButtonC" />
        <eventReceived name="ButtonD" />
    </operations>
</moduleType>
```

The Module Type is implemented by a concrete Module Implementation *EwECOAIF_modMain_Im* (depicted in the UML expanded in the form of the code class produced by the code generation process), which in turn is instantiated at runtime as the Module Instance *EwECOAIF_modMain_Inst*.

The *IO* Service Operations are again linked to Module Operations with <*eventLink*> XML tags, e.g.:

```
<eventLink>
    <senders>
        <moduleInstance instanceName="EwECOAIF_modMain_Inst"
            operationName="PatternASelected"/>
    </senders>
    <receivers>
        <service instanceName="IO" operationName="PatternASelected"/>
    </receivers>
</eventLink>
```

The *EwECOAIF* ASC also has a set of External Interfaces declared, again using *<eventLink>* elements, linking to the four Interface Module Operations, *ButtonA* to *ButtonD*, e.g.:

```
<eventLink>
    <senders>
        <external operationName="ButtonA" language="C"/>
    </senders>
    <receivers>
        <moduleInstance instanceName="EwECOAIF_modMain_Inst"
                        operationName="ButtonA"/>
    </receivers>
</eventLink>
```

That is, in each case, an *external interface* function *ButtonA* is defined that, when called, invokes the Module Operation *ButtonA*.

The external interfaces become defined in a source code header file, in this case for C, as:

```
void EwECOAIF__ButtonA();
void EwECOAIF__ButtonB();
void EwECOAIF__ButtonC();
void EwECOAIF__ButtonD();
```

and allow these functions, implemented in the *EwECOAIF_modMain_Im* Module Implementation, to be called from software outside of the ECOA “Inversion Of Control” domain (see ref.[1]).

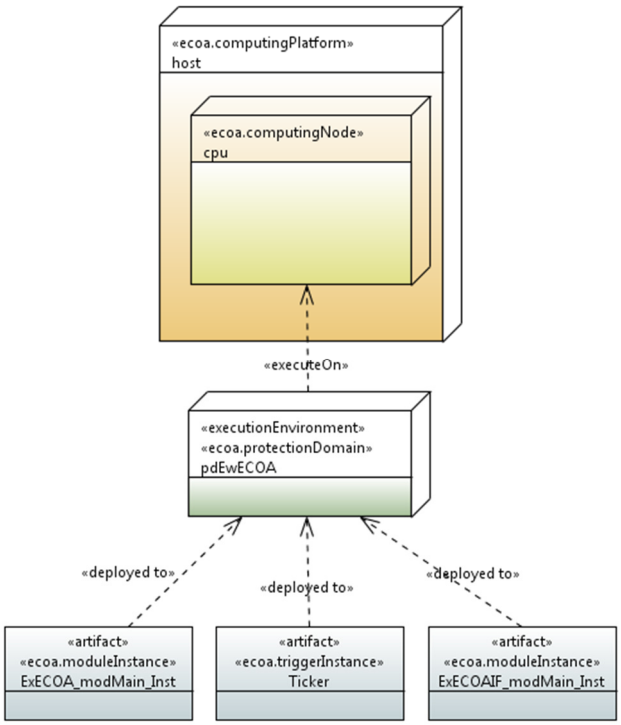
ECOA Deployment Definition

The ECOA *EmbedWithECOA* Assembly is deployed (that is, the declared Module and Trigger Instances are allocated to ECOA Protection Domains, which are themselves allocated to computing nodes) by the following XML (file *EwECOA.deployment.xml*):

```
<deployment xmlns="http://www.ecoa.technology/deployment-2.0" finalAssembly="EwECOA"
                              logicalSystem="hostbased">
  <protectionDomain name="pdEwECOA">
    <executeOn computingNode="cpu" computingPlatform="host"/>
    <deployedModuleInstance  componentName="EwECOA"
                              moduleInstanceName="EwECOA_modMain_Inst"
                              modulePriority="50"/>
    <deployedTriggerInstance componentName="EwECOA"
                              triggerInstanceName="Ticker"
                              triggerPriority="40"/>
    <deployedModuleInstance  componentName="EwECOAIF"
                              moduleInstanceName="EwECOAIF_modMain_Inst"
                              modulePriority="50"/>
   </protectionDomain>
   <platformConfiguration faultHandlerNotificationMaxNumber="8"
                              computingPlatform="host" />
</deployment>
```

That is, the two Module Instances (*EwECOA_modMain_Inst* and *EwECOAIF_modMain_Inst*) and the Trigger Instance (*Ticker*), are deployed into an ECOA Protection Domain, *pdEwECOA*, executing on the Computing Node *cpu*, which is part of the ECOA Computing Platform *host*.

Figure 8 "*EmbedWithECOA*" Deployment



Service Availability Considerations

Since the *EwECOAIF* ASC *provides* an ECOA Service (*IO*) it might be useful that the Service be declared (at runtime) as "available" or as (currently) "not available". Clients of the Service (i.e. *EwECOA*) could then check and take alternate action if the Service is not currently being provided. In the present simple example, availability of the *IO* Service has not been implemented – it is just assumed to be available once the Protection Domain (executable) has started.

Implementation

The EwEOA ASC

The function of the *EwECOA* ASC (and therefore the Module Implementation *EwECOA_modMain_Im*) is to sequence the LEDs. This is done by the *Tick* Module Operation; each time the *Ticker* Trigger Instance fires, the *Tick* Module Operation calls *DisplayCode IO* Service Operation with a new *displayCode*. The *displayCode* is set according to the next step in the currently selected sequence.

```
void EwECOA_modMain_Im__Tick__received(EwECOA_modMain_Im__context *context)
{
    switch( context->user.curPattern ){
        case A:
            if( context->user.displayCode == 0 ){
                context->user.displayCode = 0x01;
            }else{
                context->user.displayCode = context->user.displayCode << 1;
                if( context->user.displayCode > 0x20 )
                    context->user.displayCode = 0x01;
            }
            break;
        case B:
            context->user.displayCode += 1;
            if( context->user.displayCode > 63 )
                context->user.displayCode = 0;
            break;
        case C:
            switch( context->user.displayCode ){
                case 0x00:
                    context->user.displayCode = 0x0C;
                    break;
                case 0x0C:
                    context->user.displayCode = 0x12;
                    break;
                case 0x12:
                    context->user.displayCode = 0x21;
                    break;
                case 0x21:
                    context->user.displayCode = 0x00;
                    break;
            }
            break;
        case D:
            context->user.displayCode = codes[ context->user.seqNo ];
            context->user.seqNo = (++(context->user.seqNo)) % 12;
            break;
```

```
        }
        EwECOA_modMain_Im_container__DisplayCode__send( context,
                                                        context->user.displayCode );
    }
```

In order to maintain the current *displayCode* (and two other variables) across invocations of the *EwECOA_modMain_Im__Tick__received()* function, it is included in the "User Context" (see ref.[1]), and accessed as *context->user.displayCount*. The User Context is declared in the (C) code header file *EwECOA_modMain_Im_user_context.h*:

```
typedef struct {
    ECOA__uint8 curPattern;
    ECOA__uint8 displayCode;
    ECOA__uint8 seqNo;
} EwECOA_modMain_Im_user_context;
```

The EwECOAIF ASC

The function of the *EwECOAIF* ASC (and therefore the Module Implementation *EwECOAIF_modMain_Im*) is to associate the *IO* Service Operations with the physical platform specific hardware.

If you have been following closely, and have worked your way through (some of) the other ECOA samples, you may have noticed that up until now everything has been totally in the ECOA domain. Module Implementation *EwECOAIF_modMain_Im* marks the end of that road. Well, almost.

In order to be able to maximise the ability to host *EmbedWithECOA* on different hardware, even *EwECOAIF_modMain_Im* is abstracted onto another API layer (called "*EwECOAIF*"), a very simple one, comprising just three APIs:

```
void ioinit();
void pollButtons();
void putLights( unsigned char x );
```

- *ioint()* initialises (if/as necessary) the hardware to be used.
- *putlights()* turns on or off the LEDs according to the *displayCode* given.
- *pollButtons()* scans the four push-buttons. If a push-button is pressed, *pollButtons()* will call directly the appropriate ECOA External Interface function implemented by the Module Implementation (i.e. for C code, the functions *EwECOAIF__ButtonA()* &co.)
  Not all implementations will need the push-buttons to be explicitly polled – some hardware implementations may cause an interrupt when a push-button is pressed and the interrupt handler will call the ECOA External Interface function.

The Module Implementation *EwECOAIF_modMain_Im* itself is therefore the same for all hosting hardware, only the implementation of the low-level *EwECOAIF* API layer changes for each specific hardware. The implementation of the *DisplayCode* Service Operation, as an ECOA <*eventReceived*> Module Operation is therefore (in C):

```
void EwECOAIF_modMain_Im__DisplayCode__received(EwECOAIF_modMain_Im__context
*context,
```

```
        const ECOA__uint8 displayCode)
{
    /* Light the lights... */
    putLights( displayCode );
}
```

The implementation of the ECOA External Interface functions simply map to the corresponding *IO* Service Operation. So, for instance, when the *ButtonA()* External Interface function is called (implemented in C as *EwECOIF__ButtonA()*), the *ButtonA* Module Operation (declared in the *EwECOAIF_modMain_t* Module Type) is invoked, which in turn invokes the *PatternASelected* Service Operation. In C the implementation code for this Module Operation is:

```
void EwECOAIF_modMain_Im__ButtonA__received(EwECOAIF_modMain_Im__context *context)
{
    EwECOAIF_modMain_Im_container__PatternASelected__send( context );
}
```

EwECOAIF API Layer Implementation for Desktop Computer

A desktop computer tends not to have LEDs that can be arbitrarily turned on and off. They do normally have push-button switches that can be pressed though – called a keyboard. So we can devise an implementation of the low-level API layer that maps LEDs onto a screen display, and looks for specific key presses to use as push-buttons.

For the LEDs we will simply display a line of six dots ('.') and ohs ('o') – a '.' for an off LED, and an 'o' for an on LED:

..o.o.

putLights() therefore becomes:

```
void putLights( unsigned char x )
{
    unsigned char i;
    unsigned char xc = x;
    char res[9];
    //
    for( i = 0; i < 8; i++ ){
        if( xc & 0x80 )
            res[i] = 'o';
        else
            res[i] = '.';
        xc = xc << 1;
    }
    res[i] = '\000';
    //
    printf( "%s                    \r", res ); fflush( stdout );
}
```

The keyboard reader for a desktop computer needs to wait for key presses and then handle them. Normally a key press needs to be followed by the 'Enter' key, so we create a separate thread that can wait indefinitely for a key+Enter to be typed[1]:

```
void* keyReader( void* p )
{
    int key;
    // Wait for key presses...
    for(;;){
        key = getchar();
        switch( key ){
            case 'A': case 'a':
                EwECOAIF__ButtonA();
                break;
            case 'B': case 'b':
                EwECOAIF__ButtonB();
                break;
            case 'C': case 'c':
                EwECOAIF__ButtonC();
                break;
            case 'D': case 'd':
                EwECOAIF__ButtonD();
                break;
            default:
                break;
        }
        nanosleep( &Zzzz, NULL );
    }
    return NULL;
}
```

ioinit() therefore needs to spin off the keyboard reader thread:

```
void ioinit()
{
    static pthread_t keythrd;
    //
    // Spin the button (keyboard) reader
    pthread_create( &keythrd, NULL, keyReader, NULL );
}
```

EwECOAIF API Layer Implementation for the Raspberry Pi

The Raspberry Pi can be used as a (small scale) desktop computer so the previous implementation would work quite happily. However, we want the more interesting case of interfacing to the prototyping breadboard with "real" push-buttons and LEDs.

The breadboard is interfaced to the Raspberry Pi's GPIO pins, so we need a way to set and read the GPIO hardware. The following code uses the "Wiring Pi" library (ref.[7]).

ioint() is similar to the previous, but also initialises the WiringPi library:

[1] Most runtime systems will offer a means to read single key presses without the 'Enter' key being pressed, such as the *curses* (ref.[6]) library's *getch()* function. However for generality, the described implementation sticks with just the standard C library *getchar()* function which does require 'Enter'…

```
void ioinit()
{
    static pthread_t keythrd;
    unsigned char i;
    //
    // Setup the wiringPi library
    wiringPiSetup();
    //
    // Setup the GPIOs we want as inputs...
    pinMode( 0, INPUT );
    pinMode( 2, INPUT );
    pinMode( 4, INPUT );
    pinMode( 5, INPUT );
    //
    // Setup the GPIOs we want as outputs
    for( i = 23 ; i < 29 ; ++i )
        pinMode( i, OUTPUT );
    //
    // Spin the button poller
    pthread_create( &keythrd, NULL, buttonPoller, NULL );
}
```

The push-button state is read by polling the appropriate GPIO input pins:

```
void* buttonPoller( void* p )
{
    // Poll the buttons...
    for(;;){
        if( digitalRead(0) ){
            EwECOAIF__ButtonA();
        }
        else if( digitalRead(2) ){
            EwECOAIF__ButtonB();
        }
        else if( digitalRead(4) ){
            EwECOAIF__ButtonC();
        }
        else if( digitalRead(5) ){
            EwECOAIF__ButtonD();
        }
        nanosleep( &Zzzz, NULL );
    }
    return NULL;
}
```

In similar vein *putLights()* sets the appropriate GPIO output pins:

```
void putLights( unsigned char displayCode )
{
    digitalWrite( 28, ( displayCode & 0x01 ? 1 : 0 ));
    digitalWrite( 27, ( displayCode & 0x02 ? 1 : 0 ));
    digitalWrite( 26, ( displayCode & 0x04 ? 1 : 0 ));
    digitalWrite( 25, ( displayCode & 0x08 ? 1 : 0 ));
    digitalWrite( 24, ( displayCode & 0x10 ? 1 : 0 ));
    digitalWrite( 23, ( displayCode & 0x20 ? 1 : 0 ));
}
```

EwECOAIF API Layer Implementation for the PIC Microcontroller

In the case of the PIC microcontroller, we cannot spin separate threads, so the push-buttons are periodically polled by the low-level API layer *pollButtons()* API, and reads an input pin for each push-button. These input pins set bits in the *BTNPORT* register:

```
void pollButtons()
{
    switch( BTNPORT & 0x1F ){
        case 0x01:
            EwECOAIF__ButtonA();
            break;
        case 0x02:
            EwECOAIF__ButtonB();
            break;
        case 0x04:
            EwECOAIF__ButtonC();
            break;
#if   PIC==0x16f627
        case 0x08:
#elif PIC==0x16f873
        case 0x10:
#elif PIC==0x16f876
        case 0x10:
#elif PIC==0x16f876a
        case 0x10:
#endif
            EwECOAIF__ButtonD();
            break;
    }
}
```

putLights() for the PIC microcontroller is very simple – because of the way the *displayCode* parameter is defined:

```
void putLights( unsigned char displayCode )
{
    LEDPORT = displayCode;
}
```

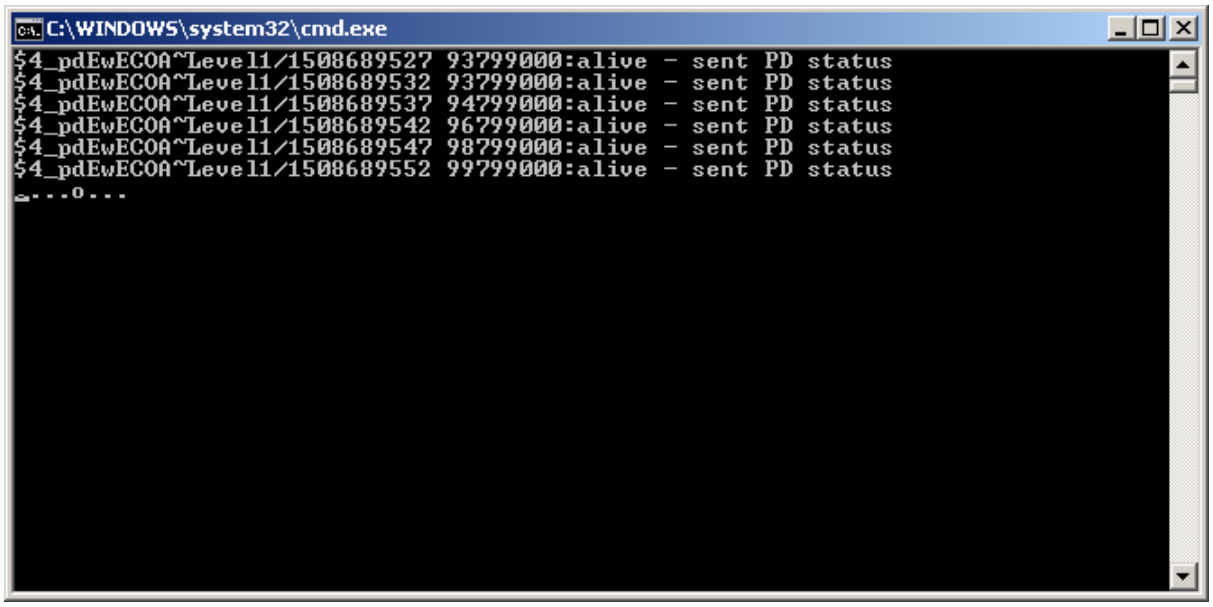
There is no initialisation to do as far as the low-level API layer is concerned:

```
void ioinit ()
{
}
```

Program Output

When the ECOA *EmbedWithECOA* Assembly is built for, and run on, a desktop computer, an output similar to Figure 9 should be achieved. The LED simulation is output to the system console, interleaved with any ECOA Platform logging messages (such as the 5 second periodic “alive” message in the example shown):

Figure 9 ECOA "*EmbedWithECOA*" in Execution (Desktop Computer Host)



It would of course be necessary to resort to video to show *EmbedWithECOA* in operation properly – particularly the Raspberry Pi and microcontroller builds…

References

1	European Component Oriented Architecture (ECOA®) Collaboration Programme: Architecture Specification (Parts 1 to 11) "ECOA" is a registered trade mark.
2	VM111[2]: PIC® Programmer & Experiment Board *Velleman Inc.*
3	PIC16F87X Data Sheet *Microchip Technology Inc.*
4	Raspberry Pi 2 Model B *Raspberry Pi Foundation* *https://www.raspberrypi.org/products/raspberry-pi-2-model-b/*
5	Pi T-Cobbler Plus – GPIO Breakout *Adafruit* *https://www.adafruit.com/product/1754*
6	curses (programming library) *https://en.wikipedia.org/wiki/Curses_%28programming_library%29* *- also -* "Screen Updating and Cursor Movement Optimization: A Library Package" University of California, Berkeley *Arnold, K. C. R. C. (1977).* *- also -* "Screen Updating and Cursor Movement Optimization: A Library Package" *Kenneth C. R. C. Arnold; Elan Amir (December 1992).*
7	Wiring Pi GPIO Interface library for the Raspberry Pi *http://wiringpi.com/*

[2] Also as "K8048" when bought in kit form.

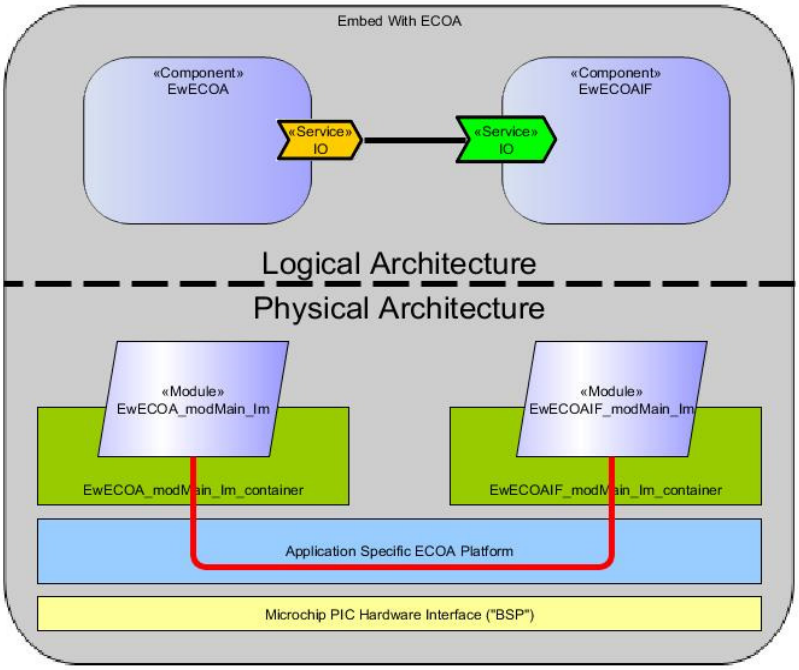
Appendix I

Microcontroller Container & ECOA Platform Implementation

ECOA applications are implemented as collections of ECOA ASCs that are built and run in ECOA Containers, integrated with an ECOA Platform infrastructure (see ref.[1]), which provides the communications links between ASCs, and the support functionality (logging, time, Trigger Instance implementations, etc.) The ASC application specific Container code and the host platform specific infrastructure code are normally generated by an ECOA Platform Code Generator[3].

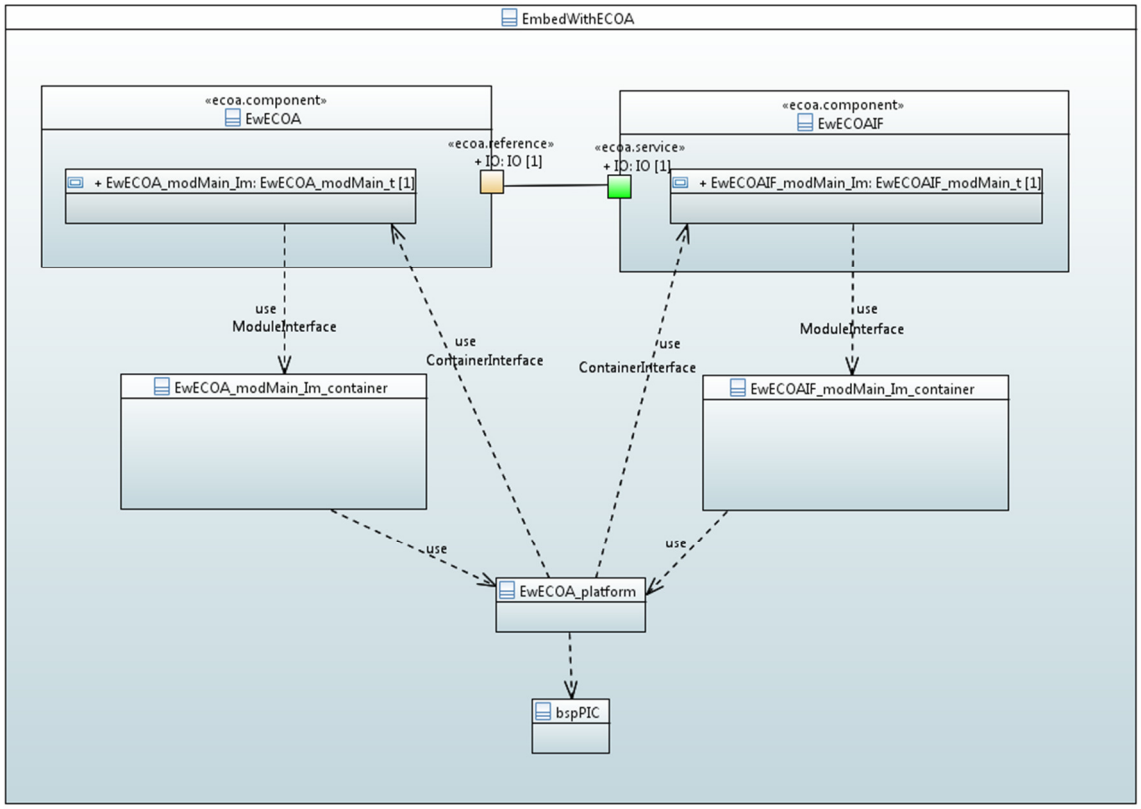
And this is the case for desktop computer and Raspberry Pi implementations of *EmbedWithECOA*, where an ECOA Platform Code Generator for POSIX can be used. For the microcontroller implementation (at least for the size of microcontroller that can be used with the Velleman board (ref.[2])) a POSIX runtime kernel is not practicable; so the Container and the ECOA Software Platform (infrastructure) code must be created explicitly "by hand". Figure 10 depicts the *EmbedWithECOA* Logical Architecture (comprising the two ASCs), and the Physical (implementation) Architecture (comprising the Module Implementations, the Containers, and the ECOA Platform), while Figure 11 interprets the same into UML.

Figure 10 "*EmbedWithECOA*" Logical and Physical Architectures



[3] This is usually separate from the ECOA "API" Code Generator that would be used to create the language specific data type header files and the framework (stub) code for the ASC Module Implementations.

Figure 11 "*EmbedWithECOA*" ECOA Software Platform Structure



ECOA Container Implementation

The two ECOA ASCs (*EwECOA* and *EwECOAIF*) are depicted at the top of Figure 11, and "comprise" (for the purposes of representation) their respective Module Implementations (*EwEOCA_modMain_Im* and *EwEOCAIF_modMain_Im*). As described in ref.[1], ECOA Module Implementation code invokes functions implemented in its Container through its *Module Interface*. The ECOA "API" Code Generator used for all these examples generates framework code for the ECOA Container for each Module Implementation (represented in the diagram as *EwECOA_modMain_Im_container* and *EwECOAIF_modMain_Im_container*), which can be populated with functional code to create the application specific Containers necessary[4].

Similarly, functionality implemented in the Module Implementation code is invoked (normally by the Container code) through its *Container Interface*. In the implementation described here, a short-cut is used. The Module Implementation code is invoked directly from the ECOA Platform implementation code.

In practise, the Containers implemented for this example are a very thin binding onto the ECOA Platform code, which in this case can be specific to the application – class *EwECOA_platform*. For

[4] Normally the Container code would be created by the ECOA Platform Code Generator.

instance in the *EwECOA_modMain_Im_container* the *get_relative_local_time()* ECOA API is implemented as:

```
void EwECOA_modMain_Im_container__get_relative_local_time
   (EwECOA_modMain_Im__context* context, ECOA__hr_time* relative_local_time)
{
    hrClock( relative_local_time );
}
```

where *hrClock()* is a function implemented in the *EwECOA_platform*.

Similarly, in the *EwECOAIF_modMain_Im_container* the *PatternCSelected* Service Operation invocation is implemented as:

```
void EwECOAIF_modMain_Im_container__PatternCSelected__send
   (EwECOAIF_modMain_Im__context* context)
{
    EwECOA_platform_PatternCSelected_send();
}
```

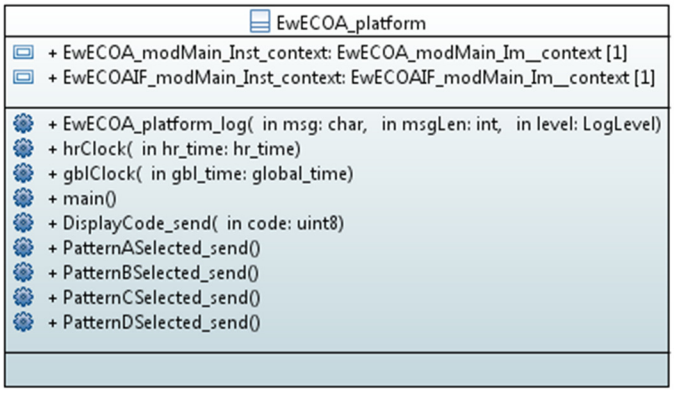
ECOA Platform Implementation

The ECOA Platform has four jobs to perform. One is to provide the communications path from one Module Implementation and Container to the other (as in Figure 11). The second is to provide the ECOA infrastructure functions such as time and logging. The third is to provide the "main" function, the program entry point.

Finally, the ECOA Platform declares and maintains the ECOA Module Context variables (as ever, see ref.[1]) for each Module Instance of the Assembly. In the present case there are only two Module Instances, one instance of each of the Module Implementations (*EwECOA_modMain_Im* and *EwECOAIF_modMain_Im*). The Module Instances are *EwECOA_modMain_Inst* and *EwECOAIF_modMain_Inst* shown way back in Figure 7.

The ECOA Platform for *EmbedWithECOA* on the microcontroller can be represented in UML as

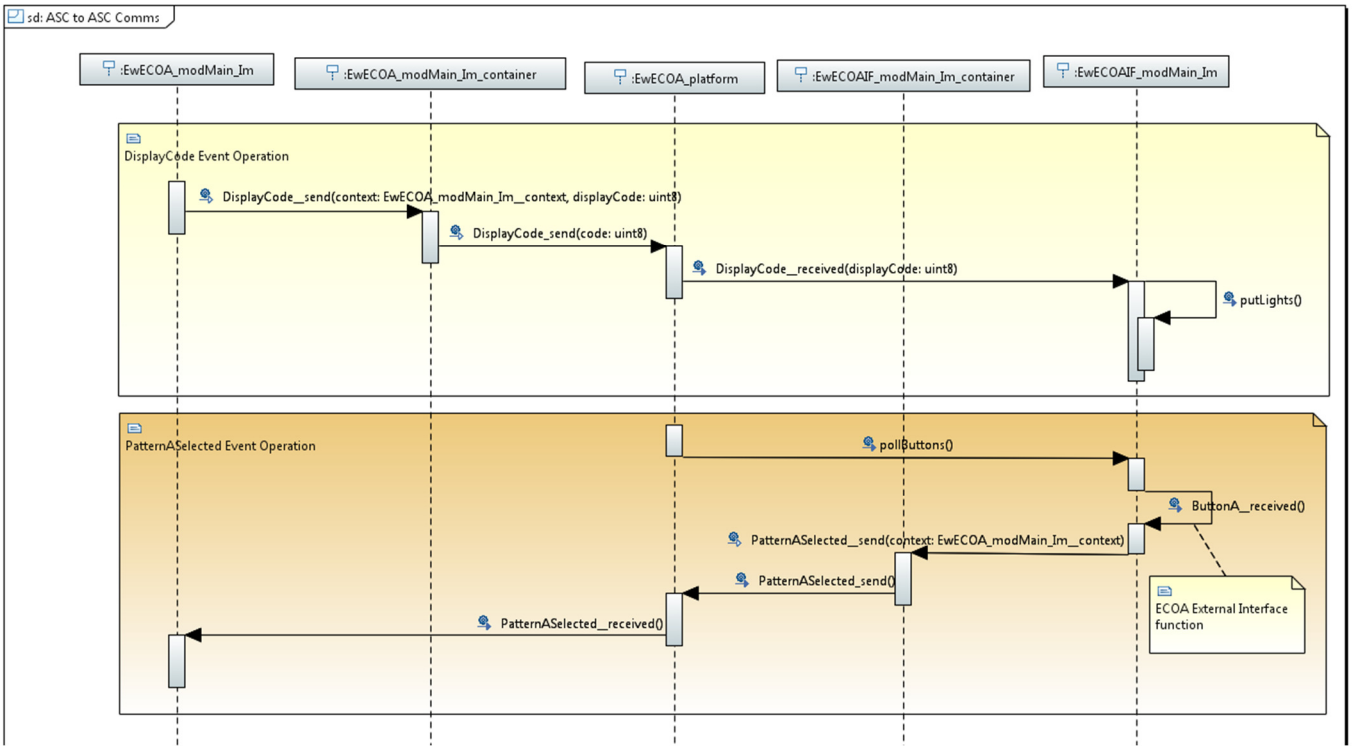
Figure 12 "*EmbedWithECOA*" Application Specific ECOA Platform



ASC-to-ASC Communications

Figure 13 illustrates, as a UML Sequence Diagram, the actions for two particular interactions. The first covers the interactions when the *EwECOA* ASC updates the LEDs by invoking the *DisplayCode* Service Operation – that is when the *EwECOA_modMain_Im* Module Implementation invokes its *DisplayCode_send()* Container function. That Container function in turn calls the ECOA Platform *DisplayCode_send()* function; which calls (directly) the *DisplayCode__received()* function of the *EwECOAIF_modMain_Im* Module Implementation, hence fulfilling the *DisplayCode* Service Operation. The *EwECOAIF_modMain_Im* Module Implementation completes the operation by calling the *putLights()* function of its low-level interface API layer.

Figure 13 "*EmbedWithECOA*" ECOA Platform ASC-to-ASC Comms.



The second interaction illustrated is the converse, where the ECOA Platform invokes the *pollButtons()* function of the *EwECOAIF* ASC low-level interface API layer; which invokes the ECOA External Interface function *ButtonA__received()* of the *EwECOAIF_modMain_Im* Module Implementation; which calls the *PatternASelected_send()* function of the *EwECOAIF_modMain_Im* Container (hence logically invoking the *PatternASelected* Service Operation); which calls the *PatternASelected__send()* function of the ECOA Platform; which calls the *PatternASelected__received()* operation of the *EwECOA_modMain_Im* Module Implementation and hence, finally, fulfilling the *PatternASelected* Service Operation.

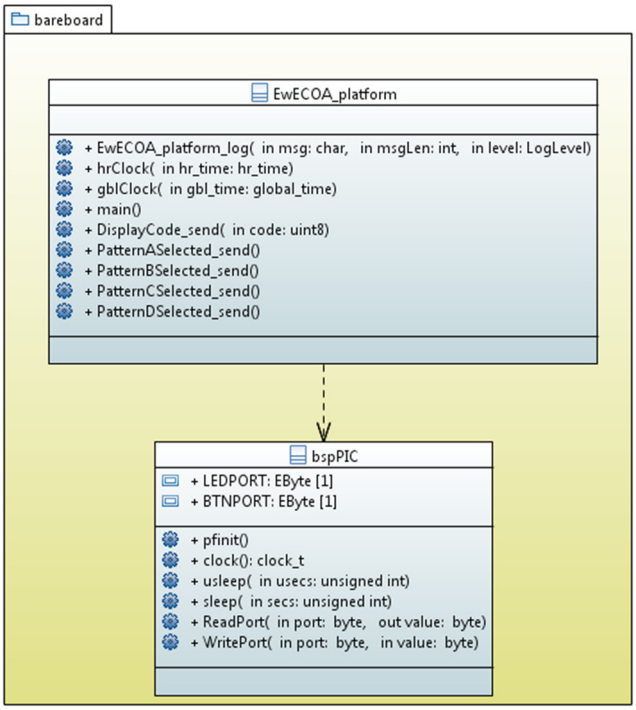
As you may tell from these sequences, the ASC-to-ASC communications functions of the ECOA Platform are again very thin bindings. For instance:

```
void EwECOA_platform_PatternASelected_send()
{
    EwECOA_modMain_Im__PatternASelected__received( &EwECOA_modMain_Inst_context );
}
```

ECOA Platform Services

For the purposes of providing the necessary ECOA infrastructure services (functions), the ECOA Platform for the microcontroller implementation is once more subdivided down, isolating out a number of very low level functions (Figure 14), functions that might in other circumstances be provided by a runtime kernel or operating system.

Figure 14 "*EmbedWithECOA*" ECOA Platform Implementation for Microcontroller



For instance, the ECOA Platform function *hrclock()* returns a time value in the ECOA *hr_time* data type (seconds and nanoseconds), and composes that value by getting a time value incremented by a hardware clock and accessed using the *clock()* function of the low-level "*bspPIC*" class. The ECOA Platform level *hrclock()* implementation is:

```
void hrClock( ECOA__hr_time *hr_time )
{
    clock_t now = clock();
    if( hr_time ){
        hr_time->seconds = now/CLOCKS_PER_SEC;
        hr_time->nanoseconds = now *
                               (1000000000L/CLOCKS_PER_SEC) % 1000000000L;
    }
}
```

That is, it gets a *clock()* reading (as some number of ticks) and converts it into seconds and nanoseconds, the number of ticks occurring in one second being given by the constant *CLOCKS_PER_SEC*.

Program Control

Every program needs a start point. When written in C this is normally a function called "*main()*". To maintain the software layering view, a functional *main()* is provided by the ECOA Platform (in C as *EwECOA_platform__main()*). Its job is to INITIALIZE and then START each Module and Trigger Instance, and to then to loop "forever" periodically "firing" any Trigger Instances – having this job to do because the microcontroller is single threaded:

```
int EwECOA_platform__main()
{
    // Initialize the Trigger & Module states
    EwECOA_modMain_Inst_pContext.tickerState  = ECOA_Platform__module_states_type_IDLE;
    EwECOA_modMain_Inst_pContext.moduleState  = ECOA_Platform__module_states_type_IDLE;
    EwECOAIF_modMain_Inst_pContext.moduleState  = ECOA_Platform__module_states_type_IDLE;
    //
    // INITIALIZE the Triggers & Modules themselves
    EwECOA_modMain_Inst_pContext.tickerState  = ECOA_Platform__module_states_type_READY;
    EwECOA_modMain_Im__INITIALIZE__received( &EwECOA_modMain_Inst_context );
    EwECOA_modMain_Inst_pContext.moduleState = ECOA_Platform__module_states_type_READY;
    EwECOAIF_modMain_Im__INITIALIZE__received( &EwECOAIF_modMain_Inst_context );
    EwECOAIF_modMain_Inst_pContext.moduleState = ECOA_Platform__module_states_type_READY;
    //
    // START the Triggers & Modules
    EwECOA_modMain_Inst_pContext.tickerState  = ECOA_Platform__module_states_type_RUNNING;
    EwECOA_modMain_Im__START__received( &EwECOA_modMain_Inst_context );
    EwECOA_modMain_Inst_pContext.moduleState = ECOA_Platform__module_states_type_RUNNING;
    EwECOAIF_modMain_Im__START__received( &EwECOAIF_modMain_Inst_context );
    EwECOAIF_modMain_Inst_pContext.moduleState = ECOA_Platform__module_states_type_RUNNING;
    //
    for(;;){
        if( EwECOA_modMain_Inst_pContext.tickerState ==
                                            ECOA_Platform__module_states_type_RUNNING ){
            EwECOA_modMain_Im__Tick__received( &EwECOA_modMain_Inst_context );
        }
        pollButtons();
        msleep( 100/*msecs*/ );   // This needs fixing...
    }
}
```

Note that in this implementation there are no explicit INITIALIZE or START functions for the Trigger Instance. The Trigger starts to "fire" as soon as this functional *main()* starts to loop, calling *EwECOA_modMain_Im__Tick__received()* on each cycle. Note also that the implementation as presented is flawed in that the loop period does not take account of the time taken by the *EwECOA_modMain_Im__Tick__received()* and *pollButtons()* functions.

This functional *main()* is called, in C, by the "real" *main()*:

```
int main()
{
    return EwECOA_platform__main();
}
```

Microcontroller "BSP"[5]

At the very lowest level of the software structure is the "*bspPIC*" layer, a small set of software functions providing commonly available routines that interface directly (or nearly so) to the system hardware. In the present case these routines are limited to clock and delay functions and IO port access (Figure 14).

For example, the "*bspPIC*" level *clock()* function counts occurrences of a hardware interrupt linked to a timer provided by the microcontroller chip itself – and therefore to the crystal controlled processor clock:

```
void Timer0_ISR() __interrupt
{
    if( T0IE && T0IF ){
        TMR0 = TMRCOUNT;
        T0IF = 0; // clear bit
        ++clockc; // Note: will wrap at some point
    }
}

clock_t clock()
{
    return clockc;
}
```

In the present implementation the interrupt is configured to occur every 100µsecs – hence the *CLOCKS_PER_SEC* constant referred to earlier has the value 10000.

In similar vein, the layer provides a *usleep()* function where processing stops for a given number of microseconds. The implementation uses a second hardware timer provided by the microcontroller to achieve an accurate delay. The *msleep()* function (stop for a given number of milliseconds) simply compounds calls of *usleep()*, whilst the *sleep()* function (given number of seconds) compounds calls of *msleep()*:

```
unsigned sleep( unsigned secs )
{
    while( secs > 0 ){
        msleep( 1000 );
        --secs;
    }
    return 0;
}
```

[5] "BSP" is an acronym for "*Board Support Package*", a layer of software providing the low-level hardware drivers and interface software – the very boundary between software and hardware. The acronym is normally applied to embedded system single board computers, but is a useful handle in the present context too.