Bridging the Lubrication and Signal I/O Domains

Legal Notices and Information

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME
1.0	September 13, 2016	Initial draft.	GAM
1.1	October 10, 2016	Rework to adapt to naming convention and class name changes.	GAM
1.2	March 24, 2017	Clean up for release as supplementary material to Models to Code.	GAM

Contents

List of Tables

Introduction

This document contains the specification and implementation of the code that bridges the Lubrication domain and the SIO domain. The bridging code is part of the examples from the book *Models to Code*. To bridge the Lubrication and Signal I/O domains, we will construct short pieces of code that map external operation calls from one domain onto domain or portal operations in the other. It is helpful to distinguish the *bridge* as a mapping of semantics of one domain onto another from a *bridge operation* which are the individual pieces of code that compose the bridge.

This document is also a literate program which means that it contains both the descriptive material for the domain as well as the `pycca` source code that implements the domain. A document, in many different formats (*e.g.* PDF), can be generated from the source using `asciidoc`[1]. The source file is a valid `asciidoc` file. The `pycca` source code that implements the model can be extracted from the source using a literate programming tool named, `atangle`[2].

Important

This literate program document intermixes model elements and implementation elements. This makes the correspondence between the model and its translation clearer and simplifies propagating changes into the translation. However, we must be clear that producing a bridge is a two step process. The domain models **must** be completed fully and the initial instance population must be determined before the bridge code can be derived from it. A complete domain model includes the specification of the bridging half tables. This is explained in Chapter 8 of *Models to Code*. Do not let the appearance of the final document suggest that the process of obtaining it was one of incremental refinement or that constructing the bridge took place at the same time as modeling the domains. Just as one does not write a novel from beginning to end in a single pass, one does not construct a system of bridged domains in the same order as it is presented here.

Scope

This document only deals with those external operations that are used to bridge between the Lubrication and Signal I/O domains. Each domain defines other operations on other external entities to deal with functionality they have delegated. Those operations are not covered here and are not includes as part of the examples from the book.

Lubrication Domain External Operations

The Lubrication domain defines four external entity operations which must be satisfied by the services of the SIO domain.

- Inject
- Stop injecting
- Start monitoring
- Stop monitoring

The first two operations deal with injecting lubricant into the machinery. The last two operations involve monitoring injector lubricant pressure.

Injecting Lubricant

The Lubrication domain uses the **Inject** operation to request that an injection of lubricant be started and the **Stop injecting** operation to request lubricant injection to be stopped. The Signal I/O domain uses Discrete Points to actuate an injector to deliver lubricant. So the bridge here is between the external operations in Lubrication to I/O Points in SIO.

It is convenient to consider the bridge mappings for these two external operations together. The only difference between starting and stopping the lubricant injection is the value written to the I/O Point that controls the injector lubricant actuator.

[1] http://www.methods.co.nz/asciidoc/
[2] http://repos.modelrealization.com/cgi-bin/fossil/mrtools

Bridge Tables

The table below shows the mapping of the external operations from the Lubrication domain to the domain operation of the SIO domain.

Table 1: Injection semantics half tables

Client Domain	EE	Operation	Service Domain	Operation	Parameter	Value
Lubrication	SIO	Inject	SIO	Write point	Value	1
Lubrication	SIO	Stop Injecting	SIO	Write point	Value	0

The external operations for lubricant injection have an identifier parameter that determines which injector is to be affected by the operation. Similarly, the domain operation for SIO uses an identifier parameter to determine which I/O Point is affected. The half tables below show the mapping between the identifier parameters of the bridge to determine which class identifier is used.

Table 2: Injection ID parameter half tables

Client Domain	EE	Operation	ID Param	Class	Service Domain	Operation	ID Param	Class
Lubrication	SIO	Inject	Injector	Injector	SIO	Write point	I/O Point	I/O Point
Lubrication	SIO	Stop Injecting	Injector	Injector	SIO	Write point	I/O Point	I/O Point

For this bridge, the participating classes in both domains have a constant instance population that is the initial instance population. The table below shows the mapping of the Injector instances to the I/O Point instances used to control lubricant injection.

Table 3: Injection instance half tables

Client Domain	Class	ID Attr	ID Value	Service Domain	Class	ID Attr	ID Value
Lubrication	Injector	ID	IN1	SIO	I/O Point	ID	IOP4
Lubrication	Injector	ID	IN2	SIO	I/O Point	ID	IOP5
Lubrication	Injector	ID	IN3	SIO	I/O Point	ID	IOP6

Bridge Implementation

Examining Table 1 shows that the only part of the table attributes that vary are the value of the SIO Write Point domain operations. Further, Table 2 shows that the identifying parameter mapping is between the Lubrication Injector class and the SIO I/O Point class. These observations imply we can implement a bridge operation using a simple function that directly codes the portions of the half table mappings that are constant.

```
<<lube external operations>>=
static void
controlInjector(
    InstId_t injectorId,
```

```
    bool starting)
{
    assert(injectorId < LUBE_INJECTOR_INST_COUNT) ;

    sio_Write_point(injToPointMap[injectorId], starting ? 1 : 0) ; // ❶
}
```

❶ The bridge operation simply invokes the SIO domain operation with with the value that is passed in as a parameter. Notice the change of type for the `starting` argument from a boolean to an integer. The identifier parameter argument for the `Write point` operation is a simple mapping from the injector identifier value to the I/O Point identifier value obtained from a mapping array.

Examining Table 3 shows that the identifier mapping is as single attribute and always between the same two classes. Since all the Injector instances participate in the bridge mapping we can use the `pycca` generated instance identifiers for the Injector instances as an index into the mapping array. The instance half table may be encoded as follows.

```
<<bridge static data>>=
static sio_Point_ID const injToPointMap[LUBE_INJECTOR_INST_COUNT] = {
    [LUBE_INJECTOR_IN1_INST_ID] = SIO_IO_POINT_IOP1_INST_ID,
    [LUBE_INJECTOR_IN2_INST_ID] = SIO_IO_POINT_IOP2_INST_ID,
    [LUBE_INJECTOR_IN3_INST_ID] = SIO_IO_POINT_IOP3_INST_ID,
} ; // ❶
```

❶ All the constant definitions are pre-processor macros generated by `pycca`. Note the use of *designated initializers* to place the array element values at the proper index in the mapping array. This insures the array elements are properly ordered regardless of the values of the index constants.

Finally, the external operations themselves are simple functions that invoke the common code for controlling an injector.

```
<<lube external operations>>=
void
eop_lube_SIO_Inject(
    InstId_t injectorId)
{
#       ifdef TACK
    harness_stub_printf("stub", "domain lube eop SIO_Inject "
        "parameters {injectorId %" PRIu8 "}", injectorId) ;     // ❶
#       endif /* TACK */

    controlInjector(injectorId, true) ;
}
```

❶ This is instrumentation code that can be conditionally compiled in and useful for debugging.

```
<<lube external operations>>=
void
eop_lube_SIO_Stop_injecting(
    InstId_t injectorId)
{
#       ifdef TACK
    harness_stub_printf("stub", "domain lube eop SIO_Stop_injecting "
        "parameters {injectorId %" PRIu8 "}", injectorId) ;
#       endif /* TACK */

    controlInjector(injectorId, false) ;
}
```

Monitoring Pressure

The Lubrication domain invokes `Start monitoring` when it wishes to monitor the pressure on an injector and `Stop monitoring` when it is no longer interested in pressure values. This bridge is asynchronous in nature. The Lubrication domain is sending an event to start or stop an action and expects to receive events at some time in the future that give feedback on the condition of the injector pressure. That feedback is provided by the [sio-new-value], [sio-in-range] and [sio-out-range] bridge operations below.

From the Signal I/O perspective, each injector has an attached pressure transducer. Values from the pressure transducer are sampled and converted to give the injector pressure. The injector pressure is represented by a Continuous Input Point and those points are part of a Conversion Group. A Conversion Group in SIO has a state model that controls the point conversion process So, we must build a mapping between the external signal from the Lubrication domain to events to a Conversion Group.

Bridge Tables

The table below shows the mapping of the external signal from the Lubrication domain to the event specification in the SIO domain.

Table 4: Pressure monitoring operation half tables

Client Domain	EE	Signal	Service Domain	Class	Event
Lubrication	SIO	Start monitoring	SIO	Conversion Group	Sample
Lubrication	SIO	Stop monitoring	SIO	Conversion Group	Stop

Signals to external entities implicitly know the instance where the signal originates (just like signals to instances within a domain). So, we do not need a half tables for the instance parameter mapping.

Again, the population of Injectors in the Lubrication domain and Conversion Groups in the SIO domain are constant and are the initial instance population. The instance mapping between the domains is as follows.

Table 5: Pressure monitoring instance half table

Client Domain	Class	ID Attr	ID Value	Service Domain	Class	ID Attr	ID Value
Lubrication	Injector	ID	IN1	SIO	Conversion Group	ID	CG1
Lubrication	Injector	ID	IN2	SIO	Conversion Group	ID	CG2
Lubrication	Injector	ID	IN3	SIO	Conversion Group	ID	CG3

Bridge Implementation

Similar to the bridge for controlling injections, if we examine Table 4 we find that the only column values that varies is the event signaled to Conversion Group instances. This leads us to create a function to perform the signaling where the event that is signaled is a parameter.

```
<<lube external operations>>=
static void
```

```
signalConversionGroup(
    InstId_t injectorId,
    EventCode event)
{
    assert(injectorId < LUBE_INJECTOR_INST_COUNT) ;

    int pcode = pycca_generate_event(&sio_portal,
        SIO_CONVERSION_GROUP_CLASS_ID,
        presToConvGrpMap[injectorId],
        NormalEvent,
        event,
        NULL) ;

    assert(pcode == 0) ;
    (void)pcode ;                                   // ❶
}
```

❶ This prevents compiler warnings when the assertions are removed.

Table 5 can be encoded as an array since all Injector instances participate in the mapping.

```
<<bridge static data>>=
static sio_Point_ID const presToConvGrpMap[LUBE_INJECTOR_INST_COUNT] = {
    [LUBE_INJECTOR_IN1_INST_ID] = SIO_CONVERSION_GROUP_INJ1_CG_INST_ID,
    [LUBE_INJECTOR_IN2_INST_ID] = SIO_CONVERSION_GROUP_INJ2_CG_INST_ID,
    [LUBE_INJECTOR_IN3_INST_ID] = SIO_CONVERSION_GROUP_INJ3_CG_INST_ID,
} ;
```

The external operations are then simple functions that pass the appropriate event code to be signaled.

```
<<lube external operations>>=
void
eop_lube_SIO_Start_monitoring(
    InstId_t injectorId)
{
#       ifdef TACK
    harness_stub_printf("stub", "domain lube eop SIO_Start_monitoring "
        "parameters {injectorId %" PRIu8 "}", injectorId) ;
#       endif /* TACK */

    signalConversionGroup(injectorId, SIO_CONVERSION_GROUP_SAMPLE_EVENT_ID) ;
}
```

```
<<lube external operations>>=
void
eop_lube_SIO_Stop_monitoring(
    InstId_t injectorId)
{
#       ifdef TACK
    harness_stub_printf("stub", "domain lube eop SIO_Stop_monitoring "
        "parameters {injectorId %" PRIu8 "}", injectorId) ;
#       endif /* TACK */

    signalConversionGroup(injectorId, SIO_CONVERSION_GROUP_STOP_EVENT_ID) ;
}
```

Signal I/O Domain Notify Operations

The Signal I/O domain defines four external entity operations as part of the NOTIFY external entity which are used to give feedback to client domains about I/O Points.

- New point value
- In range
- Out of range
- Signal point

The first three external operations are used to realize the bridge between Lubrication and SIO dealing with monitoring pressure. Recall that this bridge was asynchronous in nature and these external operations provide the unsolicited feedback on lubricant pressure values. Lubrication delegates two feedback actions to the pressure monitoring bridge. First, it expects that reading the Pressure attribute of an Injector yields an update value. Second, it expects to receive alerts when the Pressure value meets certain conditions, such as above the maximum allowed pressure.

The last external operation in the above list is used to inform the Lubrication domain of changes in the Machinery and Reservoir conditions.

New Point Value Notification

The SIO domain invokes the `New point value` operation on the NOTIFY external entity each time a newly sampled value for a Continuous Input Point is obtained. To bridge to the Lubrication domain, we must map (and transport) the Continuous Input Point value to the Pressure attribute of an Injector.

Bridge Tables

The following half tables show the semantic mapping of the Value attribute of the Continuous Input Point in the SIO domain to the Pressure attribute of the Injector class in the Lubrication domain.

Table 6: Point Value semantics half tables

Receiving Domain	**Class**	**Attribute**	**Originating Domain**	**Class**	**Attribute**
Lubrication	Injector	Pressure	SIO	Continuous Input Point	Value

The instance populations of the bridged classes in both domains are static, so we can provide an instance mapping table.

Table 7: Point Value instance half tables

Receiving Domain	**Class**	**ID Attr**	**ID Value**	**Originating Domain**	**Class**	**ID Attr**	**ID Value**
Lubrication	Injector	ID	IN1	SIO	Continuous Input Point	IO Point	IOP1
Lubrication	Injector	ID	IN2	SIO	Continuous Input Point	IO Point	IOP2
Lubrication	Injector	ID	IN3	SIO	Continuous Input Point	IO Point	IOP3

Bridge Implementation

Considering Table 7, we realize that some I/O Point do *not* participate in the instance mapping, *i.e.* there are IO Point values for I/O Points that are *not* of the Continuous Input Point type. Consequently, our previous strategy of using an array as the data structure to hold the instance mapping will not work well. We could use the array if we were willing to size it to include all the IO Point values, but such an array would be sparely populated and so is wasteful of memory. It also would not scale well. So we will implement the mapping more directly by searching.

We hold the instance mapping in an array whose element are given as follows.

```
<<bridge data structures>>=
typedef struct {
    InstId_t fromInst ;
    InstId_t toInst ;
} BridgeIDMap ;
```

This structure holds both the instance ID for the Continuous Input Point (`fromInst`) and the instance ID for the Injector (`toInst`). Placing the instance map into an array of such structures and then searching for a particular `fromInst` provides the required mapping.

```
<<bridge static data>>=
static BridgeIDMap const presToInjMap[LUBE_INJECTOR_INST_COUNT] = {
    {
        .fromInst = SIO_IO_POINT_IOP1_INST_ID,
        .toInst = LUBE_INJECTOR_IN1_INST_ID,
    },
    {
        .fromInst = SIO_IO_POINT_IOP2_INST_ID,
        .toInst = LUBE_INJECTOR_IN2_INST_ID,
    },
    {
        .fromInst = SIO_IO_POINT_IOP3_INST_ID,
        .toInst = LUBE_INJECTOR_IN3_INST_ID,
    },
} ;
```

Because the number of elements in the instance mapping array is small, we implement the search as a linear iteration over the array. For a larger number of mapping elements, we would consider either a binary search or a hash table.

```
<<bridge static functions>>=
static BridgeIDMap const *
mapIOPoint(
    BridgeIDMap const *mapping,
    int numMappings,
    InstId_t from)
{
    for ( ; numMappings > 0 ; numMappings--, mapping++) {
        if (mapping->fromInst == from) {
            return mapping ;
        }
    }

    return NULL ;
}
```

The external operation maps the I/O Point ID to an Injector ID and then uses the `pycca` portal function, `pycca_update_attr` to transport the I/O Point value across the domain boundary to the Pressure attribute of an Injector.

```
<<sio external operations>>=
void
eop_sio_NOTIFY_New_point_value(
    sio_Point_ID point,
```

```
    sio_Point_Value value)
{
#       ifdef TACK
    harness_stub_printf("stub", "domain sio eop NOTIFY_New_point_value "
        "parameters {point %" PRIu8 " value %" PRIi32 "}", point, value) ;
#       endif /* TACK */

    assert(point < SIO_IO_POINT_INST_COUNT) ;

    BridgeIDMap const * pointMap = mapIOPoint(presToInjMap, COUNTOF(presToInjMap), point) ;

    assert(pointMap != NULL) ;
    if (pointMap == NULL) {
        return ;
    }

    int pcode = pycca_update_attr(&lube_portal,
        LUBE_INJECTOR_CLASS_ID,
        pointMap->toInst,
        LUBE_INJECTOR_PRESSURE_ATTR_ID,
        &value,
        sizeof(value)) ;

    assert(pcode > 0) ;
    (void)pcode ;
}
```

Pressure Alerts

The Lubrication domain assumes that it is alerted when the following conditions occur:

- The injector pressure is above the minimum required for injecting lubricant.
- The injector pressure is below the minimum required for injecting lubricant.
- The injector pressure is above the minimum allowed when dissipating.
- The injector pressure is above a maximum allowed pressure.

The invocation interfaces for the bridge operations in this case are more complicated. When the injector pressure is above the maximum allowed, the `Max system pressure` domain operation for Lubrication must be invoked. For the other three conditions, an event must be signaled to the appropriate instance of an Injector. This means that we must bridging tables for both circumstances and that the bridge operation will have to choose whether a domain operation is invoked or an event is signaled based on the details of the pressure alert condition that is detected.

Bridge Tables

The following table maps the model elements for the maximum injector pressure alert.

Table 8: Pressure Alert semantics half tables for max pressure

Originating Domain	EE	Operation	Parameter	Value	Receiving Domain	Operation
SIO	NOTIFY	Out of range	Threshold ID	ix77b_max_pres	Lubrication	Max system pressure
SIO	NOTIFY	Out of range	Threshold ID	ihn4_max_pres	Lubrication	Max system pressure

The following table maps the model elements for the alerts associated with injection and dissipation pressure.

Table 9: Pressure Alert semantics half tables for inject and dissipation pressure

Originating Domain	EE	Operation	Parameter	Value	Receiving Domain	Class	Event Spec
SIO	NOTIFY	Out of range	Threshold ID	ix77b_above_inj	Lubrication	Injector	Above inject pressure
SIO	NOTIFY	Out of range	Threshold ID	ihn4_above_inj	Lubrication	Injector	Above inject pressure
SIO	NOTIFY	In range	Threshold ID	ix77b_above_inj	Lubrication	Injector	Below inject pressure
SIO	NOTIFY	In range	Threshold ID	ihn4_above_inj	Lubrication	Injector	Below inject pressure
SIO	NOTIFY	Out of range	Threshold ID	ix77b_above_disp	Lubrication	Injector	Above dissipation pressure
SIO	NOTIFY	Out of range	Threshold ID	ihn4_above_disp	Lubrication	Injector	Above dissipation pressure

The instance mapping table is the same as given for the point value notifications.

Bridge Implementation

As we have done in other bridge operations, we can factor out a small piece of code that signals an event to an injector. Since the event number is a parameter to the function, is may be used in several places.

```
<<bridge static functions>>=
static void
signalInjector(
    InstId_t injectorId,
    EventCode event)
{
    assert(injectorId < LUBE_INJECTOR_INST_COUNT) ;

    int pcode = pycca_generate_event(&lube_portal,
        LUBE_INJECTOR_CLASS_ID,
        injectorId,
        NormalEvent,
        event,
        NULL) ;

    assert(pcode == 0) ;
    (void)pcode ;
}
```

It is important to realize that not all invocations of `In range` and `Out of range` result in bridge operations to the Lubrication domain. The Range Limitation concept in SIO notifies both when a point value is in range and when it is out of range. However, the Lubrication domain only cares about in range alerts for the injection pressure. Thus there will be times when the in range bridge operation is invoked and nothing happens.

```
<<sio external operations>>=
void
eop_sio_NOTIFY_In_range(
    sio_Point_ID point,
    sio_Threshold_ID threshold)
{
#       ifdef TACK
    harness_stub_printf("stub", "domain sio eop NOTIFY_In_range "
        "parameters {point %" PRIu8 " threshold %" PRIu8 "}", point, threshold) ;
#       endif /* TACK */

    assert(point < SIO_IO_POINT_INST_COUNT) ;
    assert(threshold < SIO_POINT_THRESHOLD_INST_COUNT) ;

    BridgeIDMap const *pointMap = mapIOPoint(presToInjMap, COUNTOF(presToInjMap), point) ;

    assert(pointMap != NULL) ;
    if (pointMap == NULL) {
        return ;
    }

    switch (threshold) {
    case SIO_POINT_THRESHOLD_IX77B_ABOVE_INJ_INST_ID:// fall through
    case SIO_POINT_THRESHOLD_IHN4_ABOVE_INJ_INST_ID:
        signalInjector(pointMap->toInst,
            LUBE_INJECTOR_BELOW_INJECT_PRESSURE_EVENT_ID) ;
        break ;

    /*
     * N.B. no default case.
     * Unexpected Range Limitation instances are silently ignored.
     */
    }
}
```

For out of range alerts, the distinction between signaling an event invoking a domain operation is made depending upon the Point Threshold that is alerted.

```
<<sio external operations>>=
void
eop_sio_NOTIFY_Out_of_range(
    sio_Point_ID point,
    sio_Threshold_ID threshold)
{
#       ifdef TACK
    harness_stub_printf("stub", "domain sio eop NOTIFY_Out_of_range "
        "parameters {point %" PRIu8 " threshold %" PRIu8 "}", point, threshold) ;
#       endif /* TACK */

    assert(point < SIO_IO_POINT_INST_COUNT) ;
    assert(threshold < SIO_POINT_THRESHOLD_INST_COUNT) ;

    BridgeIDMap const *pointMap = mapIOPoint(presToInjMap, COUNTOF(presToInjMap), point) ;

    assert(pointMap != NULL) ;
    if (pointMap == NULL) {
        return ;
```

```
    }

    switch (threshold) {
    case SIO_POINT_THRESHOLD_IX77B_ABOVE_INJ_INST_ID:// fall through
    case SIO_POINT_THRESHOLD_IHN4_ABOVE_INJ_INST_ID:
        signalInjector(pointMap->toInst,
                LUBE_INJECTOR_ABOVE_INJECT_PRESSURE_EVENT_ID) ;
        break ;

    case SIO_POINT_THRESHOLD_IX77B_ABOVE_DISP_INST_ID:// fall through
    case SIO_POINT_THRESHOLD_IHN4_ABOVE_DISP_INST_ID:
        signalInjector(pointMap->toInst,
                LUBE_INJECTOR_ABOVE_DISSIPATION_PRESSURE_EVENT_ID) ;
        break ;

    case SIO_POINT_THRESHOLD_IX77B_MAX_PRES_INST_ID:// fall through
    case SIO_POINT_THRESHOLD_IHN4_MAX_PRES_INST_ID:
        lube_Injector_max_pressure(pointMap->toInst) ;      // ❶
        break ;

    /*
     * N.B. no default case.
     * Unexpected Range Limitation instances are silently ignored.
     */
    }
}
```

Signal Point Notifications

The SIO domain represents a two state switch using a Signal Point. A Signal Point may issue a notification when it changes to a given state or on any state change. This facility is used to satisfy the Lubrication domain requirements to know when Machinery instances are locked out and when a Reservoir is low on lubricant.

For Reservoir fluid level, we signal events.

Bridge Tables

Signaling point notifications bridge to the Lubrication domain in two ways.

For Machinery lockout, there are domain operations in Lubrication that must be invoked. If the Signal Point is active, this means the Machinery is locked and, conversely, an inactive Signal Point value means the Machinery is unlocked. This gives us the following half table for the semantics of Signal Points when applied to Machinery.

Table 10: Signal Point semantics half tables for Machinery

Originating Domain	EE	Operation	Parameter	Value	Receiving Domain	Operation
SIO	NOTIFY	Signal Point	isActive	true	Lubrication	Lock machinery
SIO	NOTIFY	Signal Point	isActive	false	Lubrication	Unlock machinery

For the Reservoir, similar logic applies, except when the point is active the Lubrication domain expects to receive a **Normal lube level** event and when inactive a **Low lube level** event must be signaled. This is shown in the following half tables for the semantics of Signal Points when applied to Reservoirs.

Table 11: Signal Point semantics half tables for Reservoir

Originating Domain	EE	Operation	Parameter	Value	Receiving Domain	Class	Event Spec
SIO	NOTIFY	Signal Point	isActive	true	Lubrication	Reservoir	Normal lube level
SIO	NOTIFY	Signal Point	isActive	true	Lubrication	Reservoir	Low lube level

We must also construct half tables to determine how the ID parameters of the domain operations are obtained.

Table 12: Signal Point ID parameter mapping half tables for Machinery

Originating Domain	EE	Operation	ID Param	Class	Receiving Domain	Operation	ID Param	Class
SIO	NOTIFY	Signal Point	point	Signal Point	Lubrication	Lock machinery	machineID	Machinery
SIO	NOTIFY	Signal Point	point	Signal Point	Lubrication	Unlock machinery	machineID	Machinery

We must also state how the recipient of Reservoir signals is identified.

Table 13: Signal Point ID parameter mapping half tables for Reservoir

Originating Domain	EE	Operation	ID Param	Class	Receiving Domain	Event Spec	Receiver ID	Class
SIO	NOTIFY	Signal Point	point	Signal Point	Lubrication	Normal lube level	ID	Reservoir
SIO	NOTIFY	Signal Point	point	Signal Point	Lubrication	Low lube level	ID	Reservoir

The following table is the instance mapping from instances of Signal Point in the SIO domain to instances of Machinery or Reservoir in the Lubrication domain.

Table 14: Signal Point instance mapping half tables

Originating Domain	Class	ID Attr	ID Value	Receiving Domain	Class	ID Attr	ID Value
SIO	Signal Point	ID	IOP7	Lubrication	Machinery	ID	M1
SIO	Signal Point	ID	IOP8	Lubrication	Machinery	ID	M2
SIO	Signal Point	ID	IOP9	Lubrication	Machinery	ID	M3
SIO	Signal Point	ID	IOP10	Lubrication	Reservoir	ID	RES1
SIO	Signal Point	ID	IOP11	Lubrication	Reservoir	ID	RES2

Bridge Implementation

Just as with the bridge operations for new point values and pressure alerts, we will encode the instance mapping in an array that can be searched. We do this because not all I/O Points participate in the mapping, implying the mapping array would be sparsely populated. We can use the same searching function, `mapIOPoint`, to find the mapping entry.

```
<<bridge static data>>=
static BridgeIDMap const
    sigPtToLubeInstMap[LUBE_MACHINERY_INST_COUNT + LUBE_RESERVOIR_INST_COUNT] = {
    {
        .fromInst = SIO_IO_POINT_IOP7_INST_ID,
        .toInst = LUBE_MACHINERY_M1_INST_ID,
    },
    {
        .fromInst = SIO_IO_POINT_IOP8_INST_ID,
        .toInst = LUBE_MACHINERY_M2_INST_ID,
    },
    {
        .fromInst = SIO_IO_POINT_IOP9_INST_ID,
        .toInst = LUBE_MACHINERY_M3_INST_ID,
    },
    {
        .fromInst = SIO_IO_POINT_IOP10_INST_ID,
        .toInst = LUBE_RESERVOIR_RES1_INST_ID,
    },
    {
        .fromInst = SIO_IO_POINT_IOP11_INST_ID,
        .toInst = LUBE_RESERVOIR_RES2_INST_ID,
    },
} ;
```

The external operation operates by mapping the Signal Point ID to a Lubrication domain instance. The point ID also serves to determine whether a domain operation is invoked or whether and event is signaled. The values of the mapping half tables are realized in code in this case.

```
<<sio external operations>>=
void
eop_sio_NOTIFY_Signal_point(
    sio_Point_ID point,
    bool isActive)
{
#       ifdef TACK
    harness_stub_printf("stub", "domain sio eop NOTIFY_Signal_point "
        "parameters {point %" PRIu8 " isActive %" PRIu8 "}", point, isActive) ;
#       endif /* TACK */

    assert(point < SIO_IO_POINT_INST_COUNT) ;

    BridgeIDMap const * pointMap = mapIOPoint(sigPtToLubeInstMap,
            COUNTOF(sigPtToLubeInstMap), point) ;

    assert(pointMap != NULL) ;
    if (pointMap == NULL) {
        return ;
    }

    switch (point) {
    case SIO_IO_POINT_IOP7_INST_ID: // fall through
    case SIO_IO_POINT_IOP8_INST_ID: // fall through
    case SIO_IO_POINT_IOP9_INST_ID: {
        void (*lubeOp)(InstId_t) = isActive ?
                lube_Lock_Machinery : lube_Unlock_Machinery ;
```

```
        lubeOp(pointMap->toInst) ;
    }
        break ;

    case SIO_IO_POINT_IOP10_INST_ID: // fall through
    case SIO_IO_POINT_IOP11_INST_ID: {
        EventCode levelEvent = isActive ?
                LUBE_RESERVOIR_NORMAL_LUBE_LEVEL_EVENT_ID :
                LUBE_RESERVOIR_LOW_LUBE_LEVEL_EVENT_ID ;
        int pcode = pycca_generate_event(&lube_portal,
            LUBE_RESERVOIR_CLASS_ID,
            pointMap->toInst,
            NormalEvent,
            levelEvent,
            NULL) ;
        assert(pcode == 0) ;
        (void)pcode ;
    }
        break ;
    /*
     * N.B. no default case.
     */
    }
}
```

Code Layout

This section describes how the literate program chunks are organized to produce a “C” code file that is acceptable to a compiler.

```
<<lube_sio_bridge.c>>=
/*
 * DO NOT EDIT THIS FILE!
 * THIS FILE IS GENERATED FROM THE SOURCE OF A LITERATE PROGRAM.
 * YOU MUST EDIT THE ORIGINAL SOURCE TO MODIFY THIS FILE.
 **++
 * Copyright 2017 by Leon Starr, Andrew Mangogna and Stephen Mellor
 *
 * Permission is hereby granted, free of charge, to any person obtaining a copy
 * of this software and associated documentation files (the "Software"), to deal
 * in the Software without restriction, including without limitation the rights
 * to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
 * copies of the Software, and to permit persons to whom the Software is
 * furnished to do so, subject to the following conditions:
 *
 * The above copyright notice and this permission notice shall be included in
 * all copies or substantial portions of the Software.
 *
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
 * IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
 * FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
 * AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
 * LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
 * OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
 * SOFTWARE.
 *
 * Project:
 *   Models to Code Book
 *
 * Module:
```

```
 *   Lubrication / SIO domain bridge
 *
 **--
*/

#include <stdbool.h>
#include <inttypes.h>
#include <stddef.h>
#include <assert.h>
#include "pycca_portal.h"
#include "lube.h"
#include "sio.h"
#ifdef TACK
#   include "harness.h"
#endif /* TACK */

#ifndef COUNTOF
#define COUNTOF(a)   (sizeof(a) / sizeof(a[0]))
#endif /* COUNTOF */

<<bridge data structures>>
<<bridge static data>>
<<bridge static functions>>

<<lube external operations>>
<<sio external operations>>
```

Literate Programming

The source for this document conforms to asciidoc syntax. This document is also a literate program. The source code for the implementation is included directly in the document source and the build process extracts the source that is then given to the Tcl interpreter. This process is known as *tangle*ing. The program, `atangle`, is available to extract source code from the document source and the `asciidoc` tool chain can be used to produce a variety of different output formats, although PDF is the intended choice.

The goal of a literate program is to explain the logic of the program in an order and fashion that facilitates human understanding of the program and then *tangle* the document source to obtain the Tcl code in an order suitable for the Tcl interpreter. Briefly, code is extracted from the literate source by defining a series of *chunks* that contain the source. A chunk is *defined* by including its name as:

```
<<chunk name>>=
```

The trailing = sign denotes a definition. A chunk definition ends at the end of the source block or at the beginning of another chunk definition. A chunk may be *referenced* from within a chunk definition by using its name without the trailing = sign, as in:

```
<<chunk definition>>=
    <<chunk reference>>
```

Chunk names are arbitrary strings. Multiple definitions with the same name are simply concatenated in the order they are encountered. There are one or more *root chunks* which form the conceptual tree for the source files that are contained in the literate source. By convention, root chunks are named the same as the file name to which they will be tangled. Tangling is then the operation of starting at a root chunk and recursively substituting the definition for the chunk references that are encountered.

For readers that are not familiar with the literate style and who are adept at reading source code directly, the chunks definitions and reordering provided by the tangle operation can be a bit disconcerting at first. You can, of course, examine the tangled source output, but if you read the program as a document, you will have to trust that the author managed to arrange the chunk definitions and references in a manner so that the tangled output is acceptable to further processing.