Bridging Remote Sensor Upload and Bluetooth Client Domains

Part of the Data Aggregator Component for Farfalle

Legal Notices and Information

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME
0.1	April 5, 2021	Start of document.	GAM
1.0	May 8, 2021	First release.	GAM
1.0.1	May 9, 2021	Clean of type declarations and other minor copy edits.	GAM

Contents

List of Figures

Introduction

The domains are part of the Data Aggregator component for the Farfalle project. The domains are subject-matter decomposed and, as part of the translation process, the system must be recomposed using a bridge. A bridge is a mapping of the semantics of one domain onto another and provides the means by which requirements attributed to the domains are met by their interactions with other domains.

This document is also a literate program document. As such it includes a complete description of both the design and implementation of the bridge operations between the two domains. Further information on the particular literal programming syntax used here is given in Appendix A.

Bridge Mappings

In this section, we present the logical mappings of the external between the Remote Sensor Upload and the Bluetooth Client domain.

The Remote Sensor Upload defines two external entities, `RADIO` and `CONN` which are the way the domain interacts with the Bluetooth Client domain.

The Bluetooth Client domain defines two external entities, `RADIO` and `CONN`.

To create a bridge requires specifying the association between egress operations and ingress operations between the external entities of two domain.

The following diagram gives a graphical representation of the bridge.

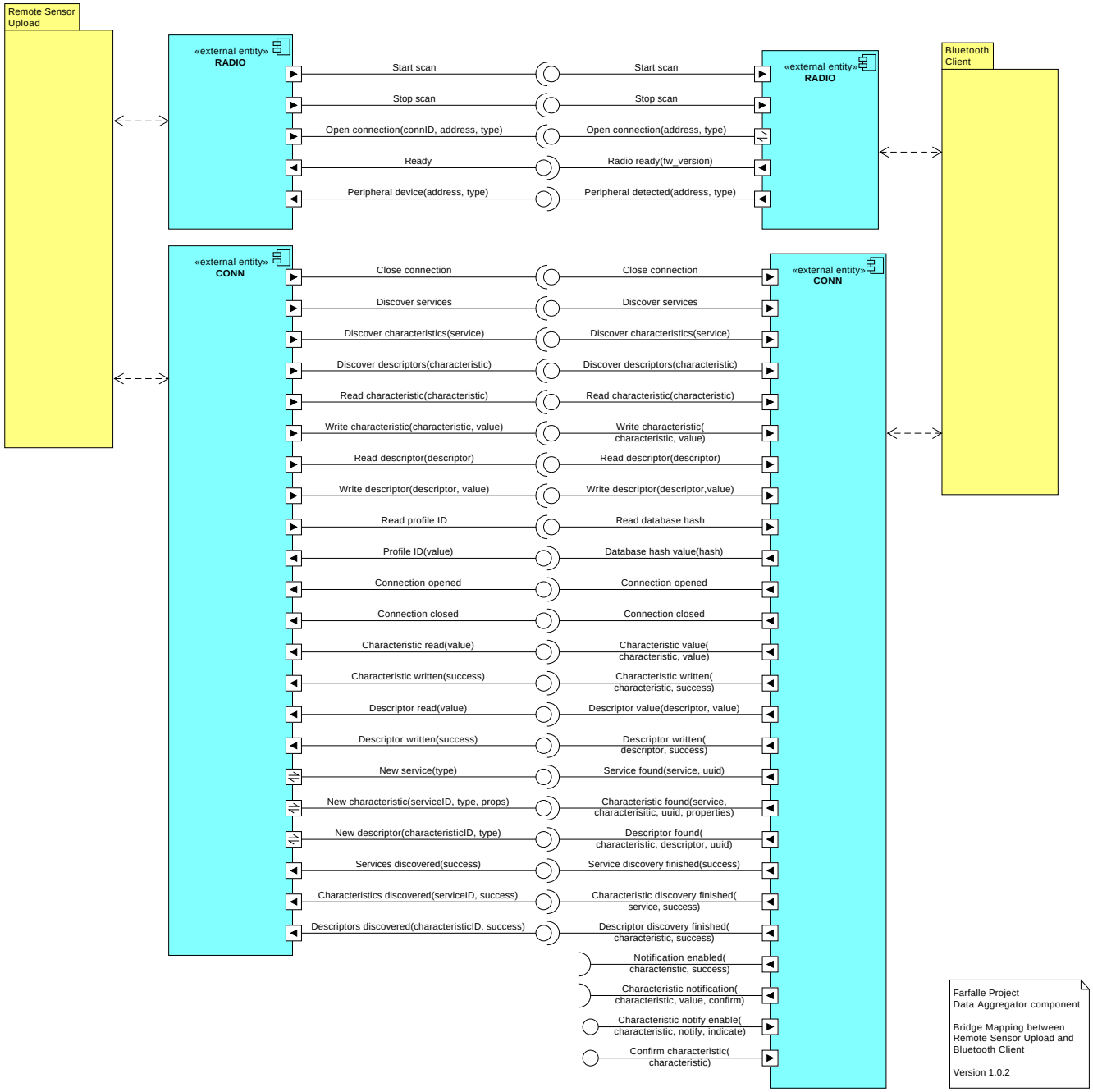

Figure 1: Remote Sensor Upload / Bluetooth Client Bridge

The notation uses UML graphical symbols which are defined as:

- Yellow package symbols are domains.
- Cyan component symbols are external entities.
- Component connectors show the mapping between external entities. Arrows indicate the flow of control. Terminators with two arrow heads are synchronous operations which return values.
- Operations using a circle terminator are ingress operations. Operations using a semi-circle terminator are egress operations.

RADIO Bridge

The following figure shows schematically the semantic association between the Remote Sensor Upload `RADIO` external entity and the Bluetooth Client `RADIO` external entity.

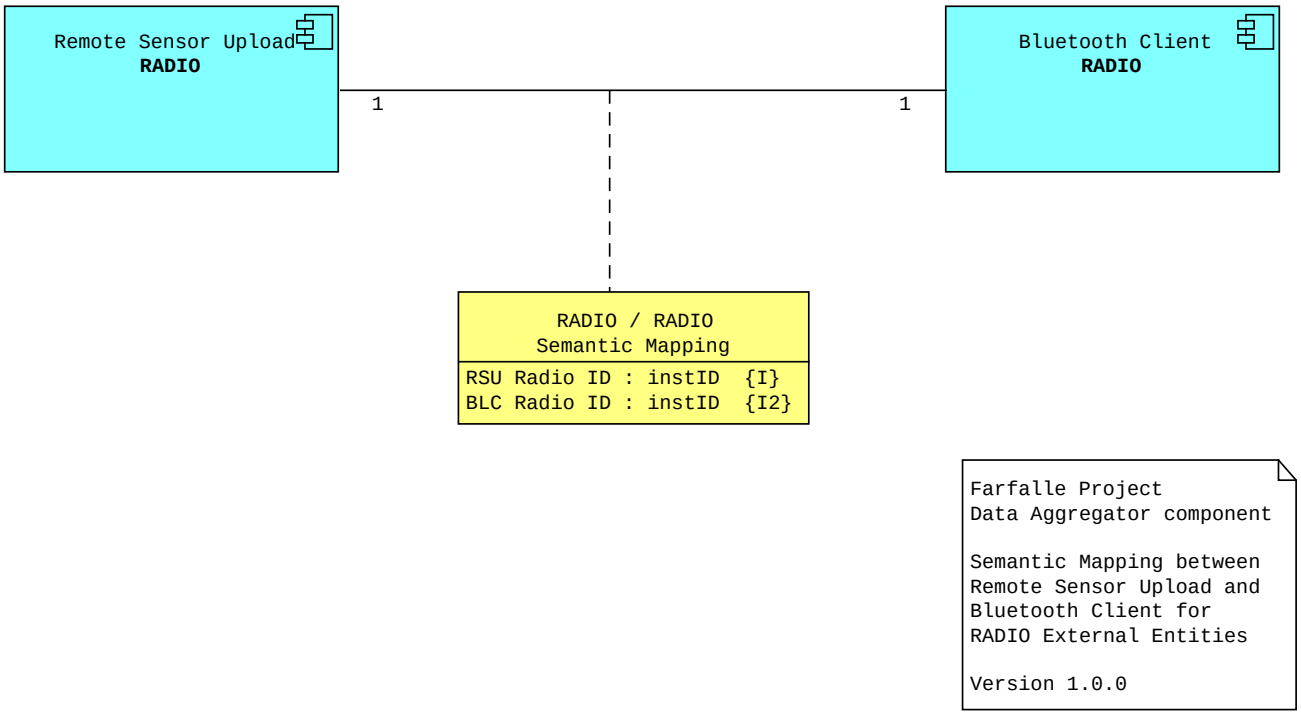

Figure 2: RADIO to RADIO Semantic Mapping

For this case, the semantic mapping is trivial and is the mapping between identifiers of the entities is the identity mapping. In practice, there is only one physical radio in the target system. The implication is that all operations directed at the Bluetooth `RADIO` entity resolve to operations on the singleton physical radio in the target system.

CONN Bridge

The following figure shows schematically the semantic association between the Remote Sensor Upload `CONN` external entity and the Bluetooth Client `CONN` external entity.

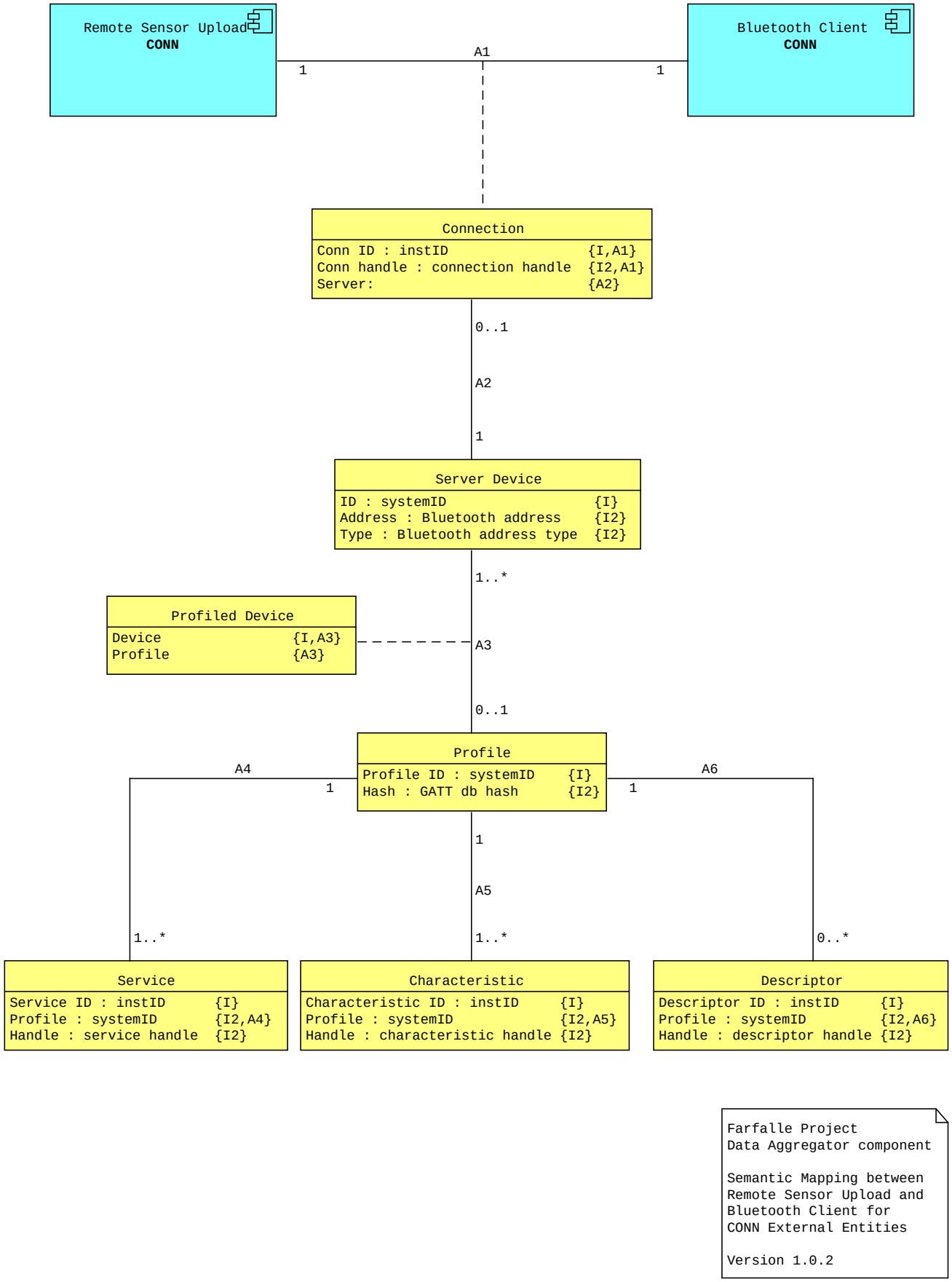

Figure 3: CONN to CONN Semantic Mapping

The mapping of semantics in this case is more extensive. An instance of the Remote Sensor Upload `CONN` entity is identified by a `micca` generated instance ID. These values are known to start at zero and be sequential to the maximum number of defined instances. The Bluetooth Client `CONN` entity instances are identified by a connection handle generated by vendor supplied Bluetooth protocol software. The range of values for connection handles is not well specified, but they appear to be small sequential numbers ranging from 1 to the maximum number of simultaneous connections configured for the Bluetooth protocol software. It appears that connection handle zero is not used and this would be analogous to GATT handle value of zero indicating the lack of a handle. In any case, we make no assumptions about the values beyond the range of the defined data type for the connection handle.

The semantic mapping of connections is the most important mapping in this bridge since no interaction with a server device is possible without a connection. This mapping is also dynamic, as connections are made only when needed for communications in order to lower the power consumption of the target system. The connection strategy of the Remote Sensor Upload domain is to form a connection and transfer data as a bundle before breaking the connection.

The other semantic mapping in the bridge is between the Remote Sensor Upload domain concepts of Service, Characteristic, and Descriptor and the analogous concepts in the Bluetooth Client domain. Since the Remote Sensor Upload domain is translated by `micca`, Service, Characteristic, and Descriptor instances are identified by `micca` generated instance identifiers which are unique for each class. The Bluetooth Client domain uses the concept of GATT handles. A GATT handle is unique for the service device from which it was *discovered*. Further, for our use case, we anticipate that the GATT handles will be the same value for most of the Server Devices since they will be running the same version of software. The GATT database hash serves as an identifier for the profile presented by a server device so the combination of GATT database hash value and GATT handle is universally unique. The bridge implements the semantic mapping between the two identification schemes. This mapping is used for operations which obtain Characteristic or Descriptor values to a connected Server Device.

With that background, we can describe the semantic associations in the previous figure.

A1
: Connections between the Remote Sensor Upload and the Bluetooth client domains are always singular and unconditional. Connections have stateful behavior inside the domains, but in the bridge they are bijectively mapped.

A2
: A connection is always to a specific server device which has a given Bluetooth address and type. Bluetooth addressing must be qualified by an address type to be unique. Server devices may exist without a connection. Server devices are found dynamically through a Bluetooth scanning process (as handled by the `RADIO` entities) and the Remote Sensor Upload domain maintains a set of server devices to which it communicates.

A3
: As part of the Bluetooth discovery process, the data profile for a server device is accumulated. There is a period of time when a Server Device is known, but its profile is not yet discovered. Hence, the conditionality of the association on the Profile side. The conditionality of **A3** on the Profile side and of **A2** on the CONN/CONN mapping side implies that instances of Server Device which are not connected and which also have no Profile may be deleted. This implication matches the action of the Remote Sensor Upload domain in the case where a server device is detected by scanning, but the profile discovery was terminated before completion.

A4, A5, A6
: The final three associations represent the mappings between the identification schemes of the two domains for the Service, Characteristic, and Descriptor concepts. It may seem unusual that these mappings do not have their own external entities, but, as previously described, all operations between devices must occur through a connection and so the mappings are not independent of that connection as would required by separate external entities.

Unused ATTR operations

The following Bluetooth Client operations are not used for bridging to the Remote Sensor Upload domain.

- Notification enabled
- Characteristic notification
- Characteristic notify enable
- Confirm characteristic

Bridge Implementation

In this section, the implementation of the bridge is described. We consider each external entity in separate section. Recall that bridging code only implements the external entity egress operations. All external entity ingress operations execute within a domain. The model level interface and semantics for the external entity egress operations of domain are part of the domain workbook for the domain. Having the workbook readily available for reference helps in understanding the implementation of the operation.

Since we are bridging two domains, we need the interface definitions for them.

```
<<include files>>=
#include "upld.h"
#include "bleclient.h"
```

Remote Sensor Upload `RADIO` $\Leftarrow \Rightarrow$ Bluetooth Client `RADIO`

In practice, there is only a single physical Bluetooth radio in the system. The identification schemes uses by the external entities are the same. Therefore the semantic mapping across the bridge is the identity mapping and no mapping data structures are needed.

Remote Sensor Upload `RADIO` Egress Operations

The correspondence between the Remote Sensor Upload `RADIO` egress operations and the Bluetooth Client `RADIO` ingress operations has been designed to be direct. Since the Bluetooth Client domain is a realized domain which encapsulates the vendor supplied Bluetooth the external entity operations have been designed to be convenient for the Remote Sensor Upload. Since the semantic mapping of radio identifiers is the identity mapping, in many cases the egress operation is a type of *trampoline* function which simply routes control from the egress operation to the ingress operation.

```
<<upld RADIO operations>>=
void
upld_RADIO_Start_scan__EOP(
    MRT_InstId radioID)
{
    MRT_INSTRUMENT_ENTRY

    bleclient_RADIO_Start_scan(radioID) ;
}
```

```
<<upld RADIO operations>>=
void
upld_RADIO_Stop_scan__EOP(
    MRT_InstId radioID)
{
    MRT_INSTRUMENT_ENTRY

    bleclient_RADIO_Stop_scan(radioID) ;
}
```

The request to open a connection is directed at the `RADIO` entity. This operation must handle the dynamic aspects of connection mapping and server devices. Those mapping functions are discussed below.

To support the connection semantic mappings, attempts to open a new connection are required to:

- Create a new instance of the Server Device identified by the `address` and `type` arguments if such a device has not already been used. The second and subsequent times a connection to a server device is opened, an instance will already exist.

- Request the connection attempt from the Bluetooth Client `RADIO` entity. A successful connection request returns the connection handle which the Bluetooth Client domain uses to reference the connection in the future. That value along with the `connID` argument makes up the CONN to CONN semantic mapping.
- Upon failure, we must inform the Remote Sensor Upload that the connection is closed. The Remote Sensor Upload acts upon this situation as a failure.

```
<<upld RADIO operations>>=
void
upld_RADIO_Open_connection__EOP(
    MRT_InstId radioID,
    MRT_InstId connID,
    BluetoothAddress_t address,
    BluetoothAddrType_t type)
{
    MRT_INSTRUMENT_ENTRY

    struct Server_Device *server_device = new_Server_Device(address, type) ;

    BLE_connHandle connHandle ;
    bool status = bleclient_RADIO_Open_connection(radioID, address,
            type, &connHandle) ;
    if (status) {
        new_Connection(connID, connHandle, server_device) ;
    } else {
        if (server_device->profile == NULL) {                    // ❶
            delete_Server_Device(server_device) ;
        }
        upld_CONN_Connection_closed(connID) ;
    }
}
```

❶ If we can't form a connection to a server device and that device has no associated profile, *i.e.* we have not connected to the device and discovered it profile previously, then it is deleted. This matches what will happen in the Remote Sensor Upload.

Bluetooth Client `RADIO` Egress Operations

The egress operations for the Bluetooth Client `RADIO` entity are simple forwarding functions.

```
<<bleclient RADIO operations>>=
void
bleclient_RADIO_Radio_ready(
    MRT_InstId radioID,
    char const *fw_version)
{
    MRT_INSTRUMENT_ENTRY

    upld_RADIO_Ready(radioID) ;
}
```

```
<<bleclient RADIO operations>>=
void
bleclient_RADIO_Peripheral_detected(
    MRT_InstId radioID,
    BLE_btAddress address,
    BLE_btAddrType type)
{
    MRT_INSTRUMENT_ENTRY
```

```
    upld_RADIO_Peripheral_device(radioID, address, type) ;
}
```

Half-table Infrastructure

To implement the semantic mappings between the CONN entities requires the bridge to maintain appropriate implementation data structures which correspond to the mappings described previously. The required mapping data are, as expected, sets of *half-tables*, *i.e.* class-like data which have two identifiers, one identifier coming from each external entity. Most of the operations required for the mapping can be considered "blind selects", *i.e.* searching for matches based solely on the values of identifying attributes, *cf.* to selection based on relationship navigation.

This design uses "C" structures for the half-tables. Each half-table is defined as an array of structures and uses a hash-based index to search for a match. The number of instances of Service, Characteristic, and Descriptor warrant a more capable search method. The number of instances for Connections, Server Devices and Profiles are small and the use of indices is probably detrimental to performance. However, we choose to use a consistent access means for all the mappings to simplify the design.

The hash index uses a Robin Hood hashing scheme.

We must now embark on a rather lengthy bit of code which is required to provide the data management and search capability for the half-tables. Arguably, this infrastructure could be separated out into its own code module to enable reuse. That probably should be done if another bridge with this type of dynamic activity occurs. Until that time, we include it here for easier reference.

```
<<include files>>=
#include "rh_hash.h"
```

The half-table infrastructure is composed of three parts:

1. Support for storage and allocation of half-table instances.
2. Defining the necessary hash tables which are used as indices into the instance storage.
3. Providing type-specific search operators tailored to the semantics of each half-table.

Half-table Storage and Allocation

Following our usual conventions, the number of entries in a given half-table is fixed at compile time. Since the half-tables are related to counterpart classes in the bridged domains, those counterpart class instance counts output from micca can be used to match the storage requirements of the half-tables with the storage allocated to the counterpart classes in the Remote Sensor Upload.

For dynamic allocation of half-table instances, we use a block-style allocator. The allocation code is made generic using the following data structure. As is usual for generic code in "C", we lose type information during the generic processing and must regain it in the type specific operations of each half-table. In short, be prepared for void pointers and pointer arithmetic scaling.

InstAllocBlock

The `InstAllocBlock` type defines the data structure used to manage the block allocation associated with the storage for a half-table.

```
<<data type declarations>>=
typedef struct {
    void *pool_start ;
    void *pool_end ;
    void *last_allocated ;
    uint32_t *alloc_status ;
    size_t instance_size ;
} InstAllocBlock ;
```

pool_start
: A pointer to the start of the array used to hold the half-table.

pool_end
: A pointer to one past the end of the half-table array.

last_allocated
: A pointer to the most recently allocated half-table array element.

alloc_status
: A pointer to an array of unsigned 32-bit integers which is used as a bit vector to track the allocation status of each element of the half-table array. A bit is set if the corresponding half-table element is currently allocated and in use, otherwise the bit is clear.

A storage pool requires three variables:

1. An array where instances are stored.
2. A bit vector array to record the allocation.
3. An initialized variable of type, `InstAllocBlock`.

To save some typing, we use a function-style pre-processor macro to define the needed variables.

DEFINE_STORAGE_POOL

The `DEFINE_STORAGE_POOL` macro defines the necessary variables and initializes them to create a block allocation memory pool. *N.B.* that the macro arguments are expanded multiple times. The arguments should not contain expressions with side effects.

```
<<macros>>=
#define DEFINE_STORAGE_POOL(pname, type, count)                                    \
    static_assert(count != 0, "instance pool counts must be non-zero") ;           \
    static type pname ## __pool[count] ;                                 /* ❶ */   \
    static uint32_t pname ## __status[(count + 31) / 32] ;               /* ❷ */   \
    static InstAllocBlock pname = {                                                \
        .pool_start = pname ## __pool,                                             \
        .pool_end = pname ## __pool + count,                                       \
        .last_allocated = pname ## __pool + count - 1,                   /* ❸ */   \
        .alloc_status = pname ## __status,                                         \
        .instance_size = sizeof(type),                                             \
    }
```

pname
A "C" identifier giving the name of the storage pool.

type
The "C" type name for the elements of the storage pool.

count
The number of elements in the storage pool.

❶ The `DEFINE_STORAGE_POOL` macro uses naming conventions to link together the storage and allocation bit-vector components to the initialized allocation variable.

❷ The calculation of the size of the allocation status bit vector computes the ceiling of the number of 32-bit unsigned integers. The number of bits in a `uint32_t` is hard coded because we just know how many bits there are and adding a level of symbolic naming doesn't help clarify the calculation.

❸ The `last_allocated` field is initialized to point to the last element of the storage pool. This causes the first block allocated to be the first element of the storage pool array. It makes debugging easier if the allocation order matches the storage pool element order, at least for the first time through the pool.

There are only two operations defined on an `InstAllocBlock`, `inst_alloc()` and `inst_free()`.

inst_alloc

The `inst_alloc` function allocates a block from the pool associated with the given Instance Allocation Block.

```
<<forward references>>=
static void *inst_alloc(
    InstAllocBlock *iab) ;
```

iab
A pointer to an Instance Allocation Block describing the storage pool from which the instance is allocated.

Returns a pointer the newly allocated instance. Note that the type of the returned pointer must be cast to be the same type as the elements of the underlying storage array. Attempts to allocate more instances than the capacity of the underlying storage array results in a panic condition. This is in keeping with instance allocation policies in the `micca` run time.

The allocation algorithm starts with the slot which is next after the slot last allocated. Our expected usage is that we will allocate and either not free or free in the same order as allocated. This usage implies that slot after the last one allocated has a high probability of being unused. If the assumption holds, then the search for an empty slot is shortened to examining a single slot.

In the search, the “next” slot is computed and examined first. Searching continues at the next sequential slot (modulo the number of elements in the storage array) until an empty slot is found. Overflow is detected if the search wraps around to the last allocated slot and that slot remains in use. Overflow results in a panic condition.

Implementation

```
<<static functions>>=
static void *
inst_alloc(
    InstAllocBlock *iab)
{
    assert(iab != NULL) ;
    assert(iab->last_allocated < iab->pool_end) ;

    void *instance = iab->last_allocated ;
    do {
        instance = inst_next_slot(iab, instance) ;
        if (!is_slot_allocated(iab, instance)) {
            set_slot_allocated(iab, instance) ;
            iab->last_allocated = instance ;                    // ❶
            return memset(instance, 0, iab->instance_size) ;    // ❷
        }
    } while (instance != iab->last_allocated) ;                 // ❸

    mrt_Panic("%s: unable to allocate instance\n", __func__) ;
}
```

❶ Record a successful allocation for the next time we allocate from this pool.

❷ Note that instances are zero’ed before they are returned. This is not strictly necessary, but is helpful in the logic associated with the instances and during debugging.

❸ The `do/while` control semantics insure that the case of an instance pool size of 1, works properly. When the pool contains only a single element, the “next” and “last” slots are always the same. It is necessary to compute “next” and test its allocation status before testing for terminating the loop by comparing to “last”.

The `inst_free` function returns a previously allocated instance to its storage pool. The instance must be freed to the same pool from which it was allocated, *i.e.* the value of `iab` must be the same as when `instance` was allocated by `inst_alloc`.

```
<<forward references>>=
static void inst_free(
    InstAllocBlock *iab,
    void *instance) ;
```

iab
A pointer to an Instance Allocation Block describing the storage pool from which the instance is allocated.

instance
A pointer to the instance to be freed. Instance must be a pointer to an instance allocated from the same storage pool as referenced by `iab`. Any attempts to free a `NULL` valued instance, an unallocated instance, or an instance outside of the storage pool results in a panic condition.

The implementation is uncomplicated. After some sanity checks to prevent corrupting the allocation pool, it is only necessary to clear the allocation indication for the instance.

Implementation

```
<<static functions>>=
static void
inst_free(
    InstAllocBlock *iab,
    void *instance)
{
    assert(iab != NULL) ;
    assert(instance != NULL) ;
    assert(instance >= iab->pool_start && instance < iab->pool_end) ;
    assert(is_slot_allocated(iab, instance)) ;

    if (instance == NULL ||
            instance < iab->pool_start ||
            instance >= iab->pool_end ||
            !is_slot_allocated(iab, instance)) {
        mrt_Panic("%s: illegal attempt to free instance\n", __func__) ;
    }

    clear_slot_allocated(iab, instance) ;
}
```

Computation of the "next" instance in the pool must account for the modulus arithmetic required to wrap back around in the storage array. Additionally, since we are dealing with `void` pointers, we need some casting to accomplish the low level address calculations.

```
<<static inline functions>>=
static inline void *
inst_next_slot(
    InstAllocBlock *iab,
    void *slot)
{
    void *next_slot = (void *)((uintptr_t)slot + iab->instance_size) ;  // ❶
    if (next_slot >= iab->pool_end) {                                   // ❷
        next_slot = iab->pool_start ;
    }
    return next_slot ;
}
```

❶ Note the run-time scaling of the pointer arithmetic.

❷ Wrap around is detected when we run off the end of the storage array. Note that calculating the pointer value, `next_slot`, is well defined — dereferencing the value if it is beyond the end of the storage array is undefined behavior.

A bit vector is used to keep track of which instances are allocated. The bits are kept in 32-bit objects. We use the index of the instance in its storage pool array as a bit index into the bit vector. This requires us to translate between pointers to instances and the array index of the instance.

Again, we are required to undertake the scaled pointer arithmetic at run time and there is some pointer casting to perform unsigned arithmetic. We isolate the incantations required for the compiler into a function.

```
<<static inline functions>>=
static inline size_t
instance_to_index(
    InstAllocBlock *iab,
    void *instance)
{
    assert(instance >= iab->pool_start && instance < iab->pool_end) ;

    return ((uintptr_t)instance - (uintptr_t)iab->pool_start) / iab->instance_size ;
}
```

Since we are using a bit vector to record allocations, we will need some bit operations.

```
<<include files>>=
#include "bit_twiddle.h"
```

We provide three operations on the bit vector: test, set, and clear. Since the bit vector may be composed of multiple 32-bit objects, we use division to determine which object holds the proper bit and the modulus operation to find the offset to the proper bit. A bit mask used with “C” bitwise operators determines whether the bit value, which encodes the status of the slot, is either allocated or free.

```
<<static inline functions>>=
static inline bool
is_slot_allocated(
    InstAllocBlock *iab,
    void *instance)
{
    size_t instance_index = instance_to_index(iab, instance) ;
    uint32_t status_bit = instance_index % 32 ;
    uint32_t status_mask = BITMASK(status_bit) ;
    uint32_t status_block = iab->alloc_status[instance_index / 32] ;

    return (status_block & status_mask) != 0 ;
}
```

```
<<static inline functions>>=
static inline void
set_slot_allocated(
    InstAllocBlock *iab,
    void *instance)
{
    size_t instance_index = instance_to_index(iab, instance) ;
    uint32_t status_bit = instance_index % 32 ;
    uint32_t status_mask = BITMASK(status_bit) ;
    uint32_t *status_block_ref = &iab->alloc_status[instance_index / 32] ;  // ❶
    *status_block_ref |= status_mask ;
}
```

❶ Note the use of a reference to the allocation status word. We want to modify the value in place and have no use for allocation word value itself.

```
<<static inline functions>>=
static inline void
clear_slot_allocated(
    InstAllocBlock *iab,
    void *instance)
{
    size_t instance_index = instance_to_index(iab, instance) ;
    uint32_t status_bit = instance_index % 32 ;
    uint32_t status_mask = BITMASK(status_bit) ;
    uint32_t *status_block_ref = &iab->alloc_status[instance_index / 32] ;
    *status_block_ref &= ~status_mask ;
}
```

Half Table Storage Definitions

For the Connection, Server_Device, and Profile mappings, we have counts of instances available from `micca` for sizing the number of instances the half-table mappings need.

```
<<static data>>=
DEFINE_STORAGE_POOL(conn_insts, struct Connection, UPLD_CONNECTION_INSTCOUNT) ;
```

```
<<static data>>=
DEFINE_STORAGE_POOL(sdev_insts, struct Server_Device, UPLD_SERVER_DEVICE_INSTCOUNT) ;
```

```
<<static data>>=
DEFINE_STORAGE_POOL(prof_insts, struct Profile, UPLD_PROFILE_INSTCOUNT) ;
```

For the Service, Characteristic and Descriptor mappings, we take a different tactic. The information for these classes, which is used for mapping instance ID's from the Remote Sensor Upload to GATT handles for the Bluetooth Client domain, can all use the same data structure. So, we create a single storage pool which is the sum of all the counterpart class instances in the Remote Sensor Upload. Then, the necessary hash indices are constructed for each different mapping type. The indices all point into the same storage pool, even though the indices separate the different mapping types.

```
<<static data>>=
DEFINE_STORAGE_POOL(attr_insts, struct Attribute,
        UPLD_SERVICE_INSTCOUNT + UPLD_CHARACTERISTIC_INSTCOUNT +
        UPLD_DESCRIPTOR_INSTCOUNT) ;
```

Connection Mapping

The implementation structure for the Connection mapping derives directly from the `CONN` semantic mapping diagram.

```
<<structure forward references>>=
struct Connection ;

<<data type declarations>>=
struct Connection {
    MRT_InstId connID ;
    BLE_connHandle connHandle ;
    struct Server_Device *server_device ;          // ❶
} ;
```

❶ The **A2** association is implemented using a pointer as a *systemID*.

We define two hash indices for the Connection mapping. The index based on a `connID` is used in Remote Sensor Upload egress operations.

```
<<static data>>=
DEFINE_IDENTIFIER(conn_id_index) = {
    ID_MEMBER(struct Connection, connID)
} ;
DEFINE_HASH_INDEX(conn_id_index, UPLD_CONNECTION_INSTCOUNT) ;
```

The index based on a `connHandle` is used in Bluetooth Client egress operations.

```
<<static data>>=
DEFINE_IDENTIFIER(conn_handle_index) = {
    ID_MEMBER(struct Connection, connHandle)
} ;
DEFINE_HASH_INDEX(conn_handle_index, UPLD_CONNECTION_INSTCOUNT) ;
```

Connection Mapping Operations

```
<<forward references>>=
static struct Connection *
new_Connection(
    MRT_InstId connID,
    BLE_connHandle connHandle,
    struct Server_Device *server_device) ;
```

connID
A connection ID representing a Remote Sensor Upload connection.

connHandle
A connection handle representing a Bluetooth Client domain connection.

server_device
A pointer to a Server Device mapping instance.

Returns a pointer to a newly created Connection mapping instance. Any failure to allocate storage for the mapping instance or add an entry to the hash indices for the Connection mapping results in a panic condition.

Implementation

```
<<static functions>>=
static struct Connection *
new_Connection(
    MRT_InstId connID,
    BLE_connHandle connHandle,
    struct Server_Device *server_device)
{
    struct Connection *connection = inst_alloc(&conn_insts) ;
    connection->connID = connID ;
    connection->connHandle = connHandle ;
    connection->server_device = server_device ;

    bool inserted = rh_insert(connection, HASH_INDEX(conn_id_index)) ;
    assert(inserted) ;
    if (!inserted) {
        inst_free(&conn_insts, connection) ;
        mrt_Panic("%s: failed to insert into connection ID index\n", __func__) ;
    }

    inserted = rh_insert(connection, HASH_INDEX(conn_handle_index)) ;
    assert(inserted) ;
    if (!inserted) {
        (void)rh_remove(connection, HASH_INDEX(conn_id_index), NULL) ;
        inst_free(&conn_insts, connection) ;
        mrt_Panic("%s: failed to insert into connection handle index\n", __func__) ;
    }

    return connection ;
}
```

The final disposition of connections arises from the Bluetooth Client domain, so a connection deletion is done with the connection handle as the identifier.

```
<<forward references>>=
static void
delete_Connection(
    BLE_connHandle connHandle) ;
```

connHandle
A connection handle representing a Bluetooth Client domain connection.

Deletes the Connection mapping instance identified by `connHandle`.

Implementation

```
<<static functions>>=
static void
delete_Connection(
    BLE_connHandle connHandle)
{
    struct Connection key = {
        .connHandle = connHandle
    } ;
    struct Connection *connection ;

    bool removed = rh_remove(&key, HASH_INDEX(conn_handle_index), (void **)&connection) ;
    assert(removed) ;
    if (!removed) {
        mrt_Panic("%s: failed to remove connection with handle = %u\n",
                __func__, connHandle) ;
    }

    removed = rh_remove(connection, HASH_INDEX(conn_id_index), NULL) ;
    assert(removed) ;
    if (!removed) {
        mrt_Panic("%s: failed to remove connection with ID = %u\n",
                __func__, connection->connID) ;
    }

    struct Server_Device *server_device = connection->server_device ;
    assert(server_device != NULL) ;

    if (server_device->profile == NULL) {                    // ❶
        delete_Server_Device(server_device) ;
    }

    inst_free(&conn_insts, connection) ;
}
```

❶ Here we enforce the notion that Server Devices for which no profile was established are deleted when the connection to them is closed. This mirrors a similar notion in the Remote Sensor Upload. It is possible to establish a connection to a Server Device and then fail to obtain any profile information. Here we clean up that possibility. The Remote Sensor Upload will scan an attempt connections in the future and those actions can be used re-establish the Server Device. Otherwise, Server Device mapping instances are never deleted.

We provide functions to search for connection instances based on either identifier.

```
<<static functions>>=
static struct Connection *
conn_find_by_id(
    MRT_InstId connID)
{
```

```
    struct Connection key = {
        .connID = connID
    } ;
    struct Connection *connection ;

    bool found = rh_lookup(&key, HASH_INDEX(conn_id_index), (void **)&connection) ;
    assert(found) ;
    if (!found) {
        mrt_Panic("%s: failed to find connection with connID = %u\n",
                __func__, connID) ;
    }

    return connection ;
}
```

```
<<static functions>>=
static struct Connection *
conn_find_by_handle(
    BLE_connHandle connHandle)
{
    struct Connection key = {
        .connHandle = connHandle
    } ;
    struct Connection *connection ;

    bool found = rh_lookup(&key, HASH_INDEX(conn_handle_index), (void **)&connection) ;
    assert(found) ;
    if (!found) {
        mrt_Panic("%s: failed to find connection with handle = %u\n",
                __func__, connHandle) ;
    }

    return connection ;
}
```

Most frequently, the egress operations need perform the semantic mapping of identifiers from one side to those of the other. The relationship is bijective and we provide a function for each direction.

```
<<static functions>>=
static inline BLE_connHandle
conn_id_to_handle(
    MRT_InstId connID)
{
    struct Connection *const connection = conn_find_by_id(connID) ;
    return connection->connHandle ;
}
```

```
<<static functions>>=
static inline MRT_InstId
conn_handle_to_id(
    BLE_connHandle connHandle)
{
    struct Connection *const connection = conn_find_by_handle(connHandle) ;
    return connection->connID ;
}
```

Tracking Server Devices

The bridge becomes aware of Server Devices when the Remote Sensor Upload attempt to connect to one. Once the Remote Sensor Upload has discovered a Server Device, it keeps track of it indefinitely and connects repeatedly.

```
<<structure forward references>>=
struct Server_Device ;

<<data type declarations>>=
struct Server_Device {
    BLE_btAddress address ;
    BLE_btAddrType type ;
    struct Profile *profile ;       // ❶
} ;
```

❶ The **A3** association is implemented via pointer. We use a value of `NULL` to represent the conditionality of **A3** and dispense with the need for an associative class.

The Remote Sensor Upload egress operations identify Server Devices using the Bluetooth address. The Bluetooth Client domain egress operations only work in terms of connection handles. So, we only a single index for the Server Device instances.

```
<<static data>>=
DEFINE_IDENTIFIER(sdev_addr_index) = {
    ID_MEMBER(struct Server_Device, address),
    ID_MEMBER(struct Server_Device, type)
} ;
DEFINE_HASH_INDEX(sdev_addr_index, UPLD_SERVER_DEVICE_INSTCOUNT) ;
```

Because the presence of a Server Device is known to the bridge only indirectly through connection attempts and because there are repeated connection attempts to the same Server Device, adding a new Server Device has union semantics, *i.e.* we only insert a new one if we fail to find a match.

```
<<static functions>>=
static struct Server_Device *
new_Server_Device(
    const BLE_btAddress address,
    BLE_btAddrType type)
{
    struct Server_Device key = {
        .type = type,
    } ;
    memcpy(key.address, address, sizeof(key.address)) ;

    struct Server_Device *sdev ;
    bool found = rh_lookup(&key, HASH_INDEX(sdev_addr_index), (void **)&sdev) ;

    if (!found) {
        sdev = inst_alloc(&sdev_insts) ;
        memcpy(sdev->address, address, sizeof(sdev->address)) ;
        sdev->type = type ;

        bool inserted = rh_insert(sdev, HASH_INDEX(sdev_addr_index)) ;
        assert(inserted) ;
        if (!inserted) {
            mrt_Panic("%s: failed to find or create server device\n", __func__) ;
        }
    }

    return sdev ;
}
```

Deleting Server Devices happens in the bridge only under the circumstance of failing to discover a Profile for the device.

```
<<forward references>>=
```

```
static void delete_Server_Device(struct Server_Device *server_device) ;

<<static functions>>=
static void
delete_Server_Device(
    struct Server_Device *server_device)
{
    assert(server_device != NULL) ;

    bool removed = rh_remove(server_device, HASH_INDEX(sdev_addr_index), NULL) ;
    assert(removed) ;
    if (!removed) {
        mrt_Panic("%s: failed to remove server device\n", __func__) ;
    }
    inst_free(&sdev_insts, server_device) ;

}
```

Tracking GATT Profile Hashes

A Profile provides part of the identification scheme for mapping the GATT handles of Services, Characteristics, and Descriptors. Similar to Server Devices, we need only a single identifier, namely the GATT database hash value itself.

```
<<structure forward references>>=
struct Profile ;

<<data type declarations>>=
struct Profile {
    Hash_t hash ;
} ;
```

```
<<static data>>=
DEFINE_IDENTIFIER(prof_hash_index) = {
    ID_MEMBER(struct Profile, hash)
} ;
DEFINE_HASH_INDEX(prof_hash_index, UPLD_SERVER_PROFILE_INSTCOUNT) ;
```

Profiles have union semantics like Server Devices.

```
<<static functions>>=
static struct Profile *
new_Profile(
    const Hash_t hash)
{
    struct Profile key  ;
    memcpy(key.hash, hash, sizeof(key.hash)) ;

    struct Profile *profile ;
    bool found = rh_lookup(&key, HASH_INDEX(prof_hash_index), (void **)&profile) ;

    if (!found) {
        profile = inst_alloc(&prof_insts) ;
        memcpy(profile->hash, hash, sizeof(profile->hash)) ;

        bool inserted = rh_insert(profile, HASH_INDEX(prof_hash_index)) ;
        assert(inserted) ;
        if (!inserted) {
            mrt_Panic("%s: failed to find or create profile\n", __func__) ;
        }
    }
```

```
    return profile ;
}
```

Mapping Instance ID's and GATT Handles

To use the same data structure for all the GATT handle objects, we define a structure that holds a Bluetooth *attribute* irrespective of type.

```
<<structure forward references>>=
struct Attribute ;

<<data type declarations>>=
typedef uint32_t AttributeHandle ;          // ❶

struct Attribute {
    MRT_InstId attrID ;
    struct Profile *profile ;
    AttributeHandle attrHandle ;
} ;
```

❶ Service handles are 32-bit quantities. Characteristic and Descriptor handles are 16-bit quantities. We choose something large enough to hold both.

Since there are three different attribute types and each have two identifiers, we need to define six hash indices to cover all the attribute types.

First for Services.

```
<<static data>>=
DEFINE_IDENTIFIER(svc_id_index) = {
    ID_MEMBER(struct Attribute, attrID),
} ;
DEFINE_HASH_INDEX(svc_id_index, UPLD_SERVICE_INSTCOUNT) ;

DEFINE_IDENTIFIER(svc_handle_index) = {
    ID_MEMBER(struct Attribute, profile),
    ID_MEMBER(struct Attribute, attrHandle),
} ;
DEFINE_HASH_INDEX(svc_handle_index, UPLD_SERVICE_INSTCOUNT) ;
```

Next for Characteristics.

```
<<static data>>=
DEFINE_IDENTIFIER(charact_id_index) = {
    ID_MEMBER(struct Attribute, attrID),
} ;
DEFINE_HASH_INDEX(charact_id_index, UPLD_CHARACTERISTIC_INSTCOUNT) ;

DEFINE_IDENTIFIER(charact_handle_index) = {
    ID_MEMBER(struct Attribute, profile),
    ID_MEMBER(struct Attribute, attrHandle),
} ;
DEFINE_HASH_INDEX(charact_handle_index, UPLD_CHARACTERISTIC_INSTCOUNT) ;
```

Finally for Descriptors.

```
<<static data>>=
DEFINE_IDENTIFIER(desc_id_index) = {
    ID_MEMBER(struct Attribute, attrID),
} ;
DEFINE_HASH_INDEX(desc_id_index, UPLD_DESCRIPTOR_INSTCOUNT) ;

DEFINE_IDENTIFIER(desc_handle_index) = {
    ID_MEMBER(struct Attribute, profile),
    ID_MEMBER(struct Attribute, attrHandle),
} ;
DEFINE_HASH_INDEX(desc_handle_index, UPLD_DESCRIPTOR_INSTCOUNT) ;
```

Since we are using multiple indices into the Attributes, creating a new one requires specifying which pair of indices to use. The indices define the Attribute type. Needless to say, the two indices should be for the same Attribute.

```
<<static functions>>=
static struct Attribute *
new_Attribute(
    MRT_InstId attrID,
    struct Profile *profile,
    AttributeHandle attrHandle,
    RH_HashTable const *id_index,
    RH_HashTable const *handle_index)
{
    struct Attribute *attribute = inst_alloc(&attr_insts) ;

    attribute->attrID = attrID ;
    attribute->profile = profile ;
    attribute->attrHandle = attrHandle ;

    bool inserted = rh_insert(attribute, id_index) ;
    assert(inserted) ;
    if (!inserted) {
        inst_free(&attr_insts, attribute) ;
        mrt_Panic("%s: failed to index attribute by ID, %u\n",
                __func__, attrID) ;
    }

    inserted = rh_insert(attribute, handle_index) ;
    assert(inserted) ;
    if (!inserted) {
        (void)rh_remove(attribute, id_index, NULL) ;
        inst_free(&attr_insts, attribute) ;
        mrt_Panic("%s: failed to index attribute by handle, %u\n",
                __func__, attrHandle) ;
    }

    return attribute ;
}
```

With generic Attributes, we need a pair of generic functions to map GATT handles to instance ID's and *vice versa*.

```
<<static functions>>=
static inline MRT_InstId
attribute_handle_to_id(
    struct Connection *connection,
    AttributeHandle attrHandle,
    RH_HashTable const *handle_index)
{
    struct Profile *profile = connection->server_device->profile ;
    assert(profile != NULL) ;
```

```
    struct Attribute key = {
        .profile = profile,
        .attrHandle = attrHandle,
    } ;
    struct Attribute *attribute ;

    bool found = rh_lookup(&key, handle_index, (void **)&attribute) ;
    assert(found) ;
    if (!found) {
        mrt_Panic("%s: failed to find attribute with handle = %u\n",
                __func__, attrHandle) ;
    }

    return attribute->attrID ;
}
```

```
<<static functions>>=
static inline AttributeHandle
attribute_id_to_handle(
    MRT_InstId attrID,
    RH_HashTable const *id_index)
{
    struct Attribute key = {
        .attrID = attrID,
    } ;
    struct Attribute *attribute ;

    bool found = rh_lookup(&key, id_index, (void **)&attribute) ;
    assert(found) ;
    if (!found) {
        mrt_Panic("%s: failed to find attribute with ID = %u\n",
                __func__, attrID) ;
    }

    return attribute->attrHandle ;
}
```

To complete the operations on Attributes, we provide a “curried” [1] version of each of the above three functions to specialize them for a specific type of attribute.

Note that each of the `new_XXX()` functions takes a Connection instance as the first argument. Since a profile instance reference is a key to any Attribute, the pointer is the manner in which the profile instance is found, *i.e.* it us used to navigate **A2** and **A3**.

The same reasoning applies to the `XXX_handle_to_id()` functions and each takes a connection instance reference as a first argument. Note the inverse function, `XXX_id_to_handle()` only requires the instance ID value since that value alone is an identifier.

First for Services.

```
<<static functions>>=
static inline struct Attribute *
new_Service(
    struct Connection *connection,
    MRT_InstId serviceID,
    AttributeHandle serviceHandle)
{
    struct Profile *profile = connection->server_device->profile ;
    assert(profile != NULL) ;
    return new_Attribute(serviceID, profile, serviceHandle, HASH_INDEX(svc_id_index),
            HASH_INDEX(svc_handle_index)) ;
```

[1] As much as currying is a concept in “C”

```
}
```

```
<<static functions>>=
static inline MRT_InstId
service_handle_to_id(
    struct Connection *connection,
    AttributeHandle serviceHandle)
{
    return attribute_handle_to_id(connection, serviceHandle, HASH_INDEX(svc_handle_index)) ↩
        ;
}
```

```
<<static functions>>=
static inline AttributeHandle
service_id_to_handle(
    MRT_InstId serviceID)
{
    return attribute_id_to_handle(serviceID, HASH_INDEX(svc_id_index)) ;
}
```

Next for Characteristics.

```
<<static functions>>=
static inline struct Attribute *
new_Characteristic(
    struct Connection *connection,
    MRT_InstId charactID,
    AttributeHandle charactHandle)
{
    struct Profile *profile = connection->server_device->profile ;
    assert(profile != NULL) ;
    return new_Attribute(charactID, profile, charactHandle, HASH_INDEX(charact_id_index),
            HASH_INDEX(charact_handle_index)) ;
}
```

```
<<static functions>>=
static inline MRT_InstId
charact_handle_to_id(
    struct Connection *connection,
    AttributeHandle charactHandle)
{
    return attribute_handle_to_id(connection, charactHandle, HASH_INDEX( ↩
        charact_handle_index)) ;
}
```

```
<<static functions>>=
static inline AttributeHandle
charact_id_to_handle(
    MRT_InstId charactID)
{
    return attribute_id_to_handle(charactID, HASH_INDEX(charact_id_index)) ;
}
```

Finally for Descriptors.

```
<<static functions>>=
static inline struct Attribute *
new_Descriptor(
    struct Connection *connection,
    MRT_InstId descID,
```

```
    AttributeHandle descHandle)
{
    struct Profile *profile = connection->server_device->profile ;
    assert(profile != NULL) ;
    return new_Attribute(descID, profile, descHandle, HASH_INDEX(desc_id_index),
            HASH_INDEX(desc_handle_index)) ;
}
```

```
<<static functions>>=
static inline MRT_InstId
desc_handle_to_id(
    struct Connection *connection,
    AttributeHandle descHandle)
{
    return attribute_handle_to_id(connection, descHandle, HASH_INDEX(desc_handle_index)) ;
}
```

```
<<static functions>>=
static inline AttributeHandle
desc_id_to_handle(
    MRT_InstId descID)
{
    return attribute_id_to_handle(descID, HASH_INDEX(desc_id_index)) ;
}
```

In summary, the previous functions provide the required operations on the half-table instances in the CONN entities semantic mapping. Most of the half-tables have three functions:

1. Create an instance.
2. Map an instance ID to a Bluetooth handle.
3. Map a Bluetooth handle to an instance ID.

Server Devices and Profiles have few functions to match their usage in the semantic mapping.

Remote Sensor Upload CONN ⇐ ⇒ Bluetooth Client CONN

With the half-table infrastructure in place, the egress operations for the Remote Sensor Upload and the Bluetooth Client domain can be defined.

Bluetooth Client CONN Engress Operations

The semantic mapping for the Bluetooth Client domain egress operations take GATT handles from the Bluetooth Client domain onto instance ID's from the Remote Sensor Upload.

```
<<bleclient CONN operations>>=
void
bleclient_CONN_Connection_opened(
    BLE_connHandle connHandle)
{
    MRT_INSTRUMENT_ENTRY

    upld_CONN_Connection_opened(conn_handle_to_id(connHandle)) ;
}
```

```
<<bleclient CONN operations>>=
void
bleclient_CONN_Connection_closed(
    BLE_connHandle connHandle)
{
    MRT_INSTRUMENT_ENTRY

    upld_CONN_Connection_closed(conn_handle_to_id(connHandle)) ;

    delete_Connection(connHandle) ;
}
```

The Bluetooth Client domain finds new Services, Characteristics, and Descriptors. Each of these egress operations follows the same pattern.

- The connection handle determines the connection ID required by Remote Sensor Upload.
- The Remote Sensor Upload ingress operations return an instance ID of the new GATT attribute.
- The instance ID is combined with the attribute handle to create an Attribute mapping.

```
<<bleclient CONN operations>>=
void
bleclient_CONN_Service_found(
    BLE_connHandle connHandle,
    BLE_serviceHandle serviceHandle,
    BLE_uuid uuid)
{
    MRT_INSTRUMENT_ENTRY

    struct Connection *connection = conn_find_by_handle(connHandle) ;

    MRT_InstId connID = connection->connID ;
    MRT_InstId serviceID = upld_CONN_New_service(connID, uuid) ;

    // add new service mapping
    (void)new_Service(connection, serviceID, serviceHandle) ;
}
```

```
<<bleclient CONN operations>>=
void
bleclient_CONN_Service_discovery_finished(
    BLE_connHandle connHandle,
    bool success)
{
    MRT_INSTRUMENT_ENTRY

    upld_CONN_Services_discovered(conn_handle_to_id(connHandle), success) ;
}
```

```
<<bleclient CONN operations>>=
void
bleclient_CONN_Characteristic_found(
    BLE_connHandle connHandle,
    BLE_serviceHandle serviceHandle,
    BLE_characteristicHandle charactHandle,
    BLE_uuid uuid,
    BLE_characteristicProperty properties)
{
    MRT_INSTRUMENT_ENTRY
```

```
    struct Connection *connection = conn_find_by_handle(connHandle) ;
    MRT_InstId connID = connection->connID ;
    MRT_InstId serviceID = service_handle_to_id(connection, serviceHandle) ;

    MRT_InstId charactID = upld_CONN_New_characteristic(connID, serviceID,
            uuid, properties) ;

    (void)new_Characteristic(connection, charactID, charactHandle) ;
}
```

```
<<bleclient CONN operations>>=
void
bleclient_CONN_Characteristic_discovery_finished(
    BLE_connHandle connHandle,
    BLE_serviceHandle serviceHandle,
    bool success)
{
    MRT_INSTRUMENT_ENTRY

    struct Connection *connection = conn_find_by_handle(connHandle) ;
    MRT_InstId connID = connection->connID ;
    MRT_InstId serviceID = service_handle_to_id(connection, serviceHandle) ;

    upld_CONN_Characteristics_discovered(connID, serviceID, success) ;
}
```

```
<<bleclient CONN operations>>=
void
bleclient_CONN_Descriptor_found(
    BLE_connHandle connHandle,
    BLE_characteristicHandle charactHandle,
    BLE_descriptorHandle descHandle,
    BLE_uuid uuid)
{
    MRT_INSTRUMENT_ENTRY

    struct Connection *connection = conn_find_by_handle(connHandle) ;
    MRT_InstId connID = connection->connID ;
    MRT_InstId charactID = charact_handle_to_id(connection, charactHandle) ;

    MRT_InstId descID = upld_CONN_New_descriptor(connID, charactID, uuid) ;

    (void)new_Descriptor(connection, descID, descHandle) ;
}
```

```
<<bleclient CONN operations>>=
void
bleclient_CONN_Descriptor_discovery_finished(
    BLE_connHandle connHandle,
    BLE_characteristicHandle charactHandle,
    bool success)
{
    MRT_INSTRUMENT_ENTRY

    struct Connection *connection = conn_find_by_handle(connHandle) ;
    MRT_InstId connID = connection->connID ;
    MRT_InstId charactID = charact_handle_to_id(connection, charactHandle) ;

    upld_CONN_Descriptors_discovered(connID, charactID, success) ;
}
```

```
<<bleclient CONN operations>>=
void
bleclient_CONN_Characteristic_value(
    BLE_connHandle connHandle,
    BLE_characteristicHandle charactHandle,
    sba_elem_type *value)
{
    MRT_INSTRUMENT_ENTRY

    struct Connection *connection = conn_find_by_handle(connHandle) ;
    MRT_InstId connID = connection->connID ;
    MRT_InstId charactID = charact_handle_to_id(connection, charactHandle) ;

    upld_CONN_Characteristic_read(connID, charactID, value) ;
}
```

```
<<bleclient CONN operations>>=
void
bleclient_CONN_Characteristic_written(
    BLE_connHandle connHandle,
    BLE_characteristicHandle charactHandle,
    bool success)
{
    MRT_INSTRUMENT_ENTRY

    struct Connection *connection = conn_find_by_handle(connHandle) ;
    MRT_InstId connID = connection->connID ;
    MRT_InstId charactID = charact_handle_to_id(connection, charactHandle) ;

    upld_CONN_Characteristic_written(connID, charactID, success) ;
}
```

```
<<bleclient CONN operations>>=
void
bleclient_CONN_Notification_enabled(
    BLE_connHandle connection,
    BLE_characteristicHandle characteristic,
    bool success)
{
    MRT_INSTRUMENT_ENTRY

    // no ingress mapping
}
```

```
<<bleclient CONN operations>>=
void
bleclient_CONN_Characteristic_notification(
    BLE_connHandle connection,
    BLE_characteristicHandle characteristic,
    sba_elem_type *value,
    bool confirm)
{
    MRT_INSTRUMENT_ENTRY

    // no ingress mapping
}
```

```
<<bleclient CONN operations>>=
void
bleclient_CONN_Descriptor_value(
```

```
    BLE_connHandle connHandle,
    BLE_descriptorHandle descHandle,
    sba_elem_type *value)
{
    MRT_INSTRUMENT_ENTRY

    struct Connection *connection = conn_find_by_handle(connHandle) ;
    MRT_InstId connID = connection->connID ;
    MRT_InstId descID = desc_handle_to_id(connection, descHandle) ;

    upld_CONN_Descriptor_read(connID, descID, value) ;
}
```

```
<<bleclient CONN operations>>=
void
bleclient_CONN_Descriptor_written(
    BLE_connHandle connHandle,
    BLE_descriptorHandle descHandle,
    bool success)
{
    MRT_INSTRUMENT_ENTRY

    struct Connection *connection = conn_find_by_handle(connHandle) ;
    MRT_InstId connID = connection->connID ;
    MRT_InstId descID = desc_handle_to_id(connection, descHandle) ;

    upld_CONN_Descriptor_written(connID, descID, success) ;
}
```

The bridge uses the `bleclient_CONN_Database_hash_value()` egress operation to obtain values for a Profile and to associate that Profile to a particular Server Device.

```
<<bleclient CONN operations>>=
void
bleclient_CONN_Database_hash_value(
    BLE_connHandle connHandle,
    sba_elem_type *hash)
{
    MRT_INSTRUMENT_ENTRY

    assert(sba_length(hash) == sizeof(Hash_t)) ;

    struct Connection *connection = conn_find_by_handle(connHandle) ;
    struct Server_Device *server_device = connection->server_device ;
    assert(server_device->profile == NULL) ;
    server_device->profile = new_Profile(sba_data(hash)) ;

    upld_CONN_Profile_ID(connection->connID, sba_data(hash)) ;
}
```

Remote Sensor Upload `CONN` Egress Operations

The semantic mapping for the Remote Sensor Upload egress operations take instance ID's from the Remote Sensor Upload onto GATT handles from the Remote Sensor Upload.

```
<<upld CONN operations>>=
void
upld_CONN_Close_connection__EOP(
    MRT_InstId connID)
{
```

```
    MRT_INSTRUMENT_ENTRY

    bleclient_CONN_Close_connection(conn_id_to_handle(connID)) ;
}
```

```
<<upld CONN operations>>=
void
upld_CONN_Discover_services__EOP(
    MRT_InstId connID)
{
    MRT_INSTRUMENT_ENTRY

    bleclient_CONN_Discover_services(conn_id_to_handle(connID)) ;
}
```

```
<<upld CONN operations>>=
void
upld_CONN_Discover_characteristics__EOP(
    MRT_InstId connID,
    MRT_InstId serviceID)
{
    MRT_INSTRUMENT_ENTRY

    BLE_connHandle connHandle = conn_id_to_handle(connID) ;
    BLE_serviceHandle serviceHandle = service_id_to_handle(serviceID) ;
    bleclient_CONN_Discover_characteristics(connHandle, serviceHandle) ;
}
```

```
<<upld CONN operations>>=
void
upld_CONN_Discover_descriptors__EOP(
    MRT_InstId connID,
    MRT_InstId charactID)
{
    MRT_INSTRUMENT_ENTRY

    BLE_connHandle connHandle = conn_id_to_handle(connID) ;
    BLE_characteristicHandle charactHandle = charact_id_to_handle(charactID) ;

    bleclient_CONN_Discover_descriptors(connHandle, charactHandle) ;
}
```

```
<<upld CONN operations>>=
void
upld_CONN_Read_characteristic__EOP(
    MRT_InstId connID,
    MRT_InstId charactID)
{
    MRT_INSTRUMENT_ENTRY

    BLE_connHandle connHandle = conn_id_to_handle(connID) ;
    BLE_characteristicHandle charactHandle = charact_id_to_handle(charactID) ;

    bleclient_CONN_Read_characteristic(connHandle, charactHandle) ;
}
```

```
<<upld CONN operations>>=
void
upld_CONN_Write_characteristic__EOP(
    MRT_InstId connID,
```

```
    MRT_InstId charactID,
    AttributeValue_t value)
{
    MRT_INSTRUMENT_ENTRY

    BLE_connHandle connHandle = conn_id_to_handle(connID) ;
    BLE_characteristicHandle charactHandle = charact_id_to_handle(charactID) ;

    bleclient_CONN_Write_characteristic (connHandle, charactHandle, value) ;
}
```

```
<<upld CONN operations>>=
void
upld_CONN_Read_descriptor__EOP(
    MRT_InstId connID,
    MRT_InstId descID)
{
    MRT_INSTRUMENT_ENTRY

    BLE_connHandle connHandle = conn_id_to_handle(connID) ;
    BLE_descriptorHandle descHandle = desc_id_to_handle(descID) ;

    bleclient_CONN_Read_descriptor(connHandle, descHandle) ;
}
```

```
<<upld CONN operations>>=
void
upld_CONN_Write_descriptor__EOP(
    MRT_InstId connID,
    MRT_InstId descID,
    AttributeValue_t value)
{
    MRT_INSTRUMENT_ENTRY

    BLE_connHandle connHandle = conn_id_to_handle(connID) ;
    BLE_descriptorHandle descHandle = desc_id_to_handle(descID) ;

    bleclient_CONN_Write_descriptor (connHandle, descHandle, value) ;
}
```

```
<<upld CONN operations>>=
void
upld_CONN_Read_profile_ID__EOP(
    MRT_InstId connID)
{
    MRT_INSTRUMENT_ENTRY

    bleclient_CONN_Read_database_hash(conn_id_to_handle(connID)) ;
}
```

Code Layout

```
<<upld_bleclient.c>>=
<<edit warning>>
<<copyright info>>
/*
 *++
 * Project:
```

```
 *   Farfalle
 *
 * Module:
 *   Remote Sensor Upload / Bluetooth Client Bridge
 *--
 */

#define MRT_DOMAIN_NAME "upld-bleclient"

/*
 * Include files
 */
#include "micca_rt.h"
#include "micca_debug.h"
#include <stdlib.h>
#include <stdarg.h>
#include <assert.h>
<<include files>>
/*
 * Constants
 */
<<constants>>
/*
 * Macros
 */
<<macros>>
/*
 * Structure Forward References
 */
<<structure forward references>>
/*
 * Data Type Declarations
 */
<<data type declarations>>
/*
 * Forward References
 */
<<forward references>>
/*
 * Static Inline Functions
 */
<<static inline functions>>
/*
 * Static Data
 */
<<static data>>
/*
 * Static Functions
 */
<<static functions>>
/*
 * RADIO External Entity Operations
 */
<<upld RADIO operations>>
<<bleclient RADIO operations>>
/*
 * CONN External Entity Operations
 */
<<upld CONN operations>>
<<bleclient CONN operations>>
```

Edit Warning

We want to make sure to warn readers that the source code is generated and not manually written.

```
<<edit warning>>=
/*
 * DO NOT EDIT THIS FILE!
 * THIS FILE IS AUTOMATICALLY GENERATED FROM A LITERATE PROGRAM SOURCE FILE.
 */
```

Copyright Information

The following is copyright and licensing information.

```
<<copyright info>>=
/*
 * This software is copyrighted 2021 the Farfalle Project.
 * The following terms apply to all files associated with the software unless
 * explicitly disclaimed in individual files.
 *
 * The authors hereby grant permission to use, copy, modify, distribute,
 * and license this software and its documentation for any purpose, provided
 * that existing copyright notices are retained in all copies and that this
 * notice is included verbatim in any distributions. No written agreement,
 * license, or royalty fee is required for any of the authorized uses.
 * Modifications to this software may be copyrighted by their authors and
 * need not follow the licensing terms described here, provided that the
 * new terms are clearly indicated on the first page of each file where
 * they apply.
 *
 * IN NO EVENT SHALL THE AUTHORS OR DISTRIBUTORS BE LIABLE TO ANY PARTY FOR
 * DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES ARISING
 * OUT OF THE USE OF THIS SOFTWARE, ITS DOCUMENTATION, OR ANY DERIVATIVES
 * THEREOF, EVEN IF THE AUTHORS HAVE BEEN ADVISED OF THE POSSIBILITY OF
 * SUCH DAMAGE.
 *
 * THE AUTHORS AND DISTRIBUTORS SPECIFICALLY DISCLAIM ANY WARRANTIES,
 * INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY,
 * FITNESS FOR A PARTICULAR PURPOSE, AND NON-INFRINGEMENT.  THIS SOFTWARE
 * IS PROVIDED ON AN "AS IS" BASIS, AND THE AUTHORS AND DISTRIBUTORS HAVE
 * NO OBLIGATION TO PROVIDE MAINTENANCE, SUPPORT, UPDATES, ENHANCEMENTS,
 * OR MODIFICATIONS.
 *
 * GOVERNMENT USE: If you are acquiring this software on behalf of the
 * U.S. government, the Government shall have only "Restricted Rights"
 * in the software and related documentation as defined in the Federal
 * Acquisition Regulations (FARs) in Clause 52.227.19 (c) (2).  If you
 * are acquiring the software on behalf of the Department of Defense,
 * the software shall be classified as "Commercial Computer Software"
 * and the Government shall have only "Restricted Rights" as defined in
 * Clause 252.227-7013 (c) (1) of DFARs.  Notwithstanding the foregoing,
 * the authors grant the U.S. Government and others acting in its behalf
 * permission to use and distribute the software in accordance with the
 * terms specified in this license.
 */
```

Bibliography

Books

[1] [mb-xuml] Stephen J. Mellor and Marc J. Balcer, Executable UML: a foundation for model-driven architecture, Addison-Wesley (2002), ISBN 0-201-74804-5.

[2] [rs-xuml] Chris Raistrick, Paul Francis, John Wright, Colin Carter and Ian Wilkie, Model Driven Architecture with Executable UML, Cambridge University Press (2004), ISBN 0-521-53771-1.

[3] [mtoc] Leon Starr, Andrew Mangogna and Stephen Mellor, Models to Code: With No Mysterious Gaps, Apress (2017), ISBN 978-1-4842-2216-4

[4] [ls-build], Leon Starr, How to Build Shlaer-Mellor Object Models, Yourdon Press (1996), ISBN 0-13-207663-2.

[5] [sm-data] Sally Shlaer and Stephen J. Mellor, Object Oriented Systems Analysis: Modeling the World in Data, Prentice-Hall (1988), ISBN 0-13-629023-X.

[6] [sm-states] Sally Shlaer and Stephen J. Mellor, Object Oriented Systems Analysis: Modeling the World in States, Prentice-Hall (1992), ISBN 0-13-629940-7.

Articles

[7] [ls-articulate] Leon Starr, How to Build Articulate UML Class Models, 2008, http://www.modelint.com/how-to-build-articulate-uml-class-models/

[8] [ls-time] Leon Starr, Time and Synchronization in Executable UML, 2008, http://www.modelint.com/time-and-synchronization-in-executable-uml/

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 code which is then given to the `micca` program. 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 code in an order suitable for a language processor. 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 chunk 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 in an acceptable order.