mbeddr C Extensions User's Guide

Markus Voelter

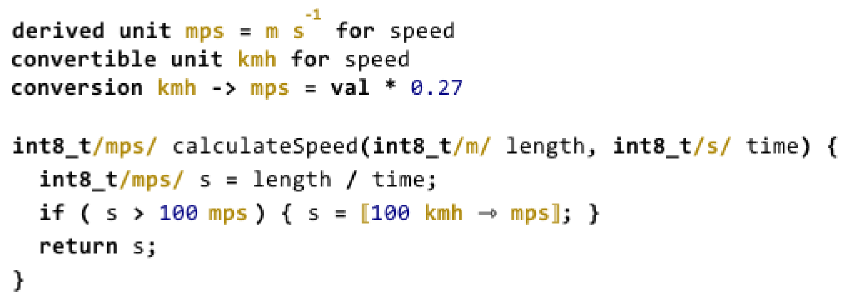
independent/itemis

Abstract. This document describes the *mbeddr.ext* package that currently contains physical units, exceptions, components and state machines.

Table of Contents

1 Physical Units

Physical Units are new types that, in addition to specifying their actual data type, also specify a physical unit (see Fig. 1). New literals are introduced to support specifying values for these types that include the physical unit. The typing rules for the existing operators (`+`, `*` or `>`) are overridden to perform the correct type checks for types with units. The type system also performs unit computations to, for example, handle an `speed = length/time` correctly.



```
                       -1
derived unit mps = m s    for speed
convertible unit kmh for speed
conversion kmh -> mps = val * 0.27

int8_t/mps/ calculateSpeed(int8_t/m/ length, int8_t/s/ time) {
  int8_t/mps/ s = length / time;
  if ( s > 100 mps ) { s = [100 kmh → mps]; }
  return s;
}
```

Fig. 1. The *units* extension comes with the sevel SI units predefined and lets users define arbitrary derived units (such as the `mps` in the example). It is also possible to defined convertible units that require a numeric conversion factor to get back to SI units. Type checks ensure that the values associated with unit literals use the correct unit and perform unit computations (as in speed equals length divided by time). Errors are recorded if incompatible are used together (e.g. if we were to add length and time). To support this feature, the typing rules for the existing operators (such as `+` or `/`) have to be overridden.

To use physical units use the `com.mbeddr.physicalunits` devkit in your model.

1.1 Basic SI Units in C programs

Once the devkit is included, types can be annotated with physical units. We have defined the seven SI units in mbeddr:

```
int8_t/m/ length;
int8_t/s/ time;
```

It is also possible to define composite units. To add additional components press enter after the first one (the `m` in the example below):

```
          -1
int8_t/m s   / speed;
```

To change the exponent, use intentions on the unit. Types with units can also be `typedef`'ed to make using them more convenient:

```
                  -1
typedef int8_t/m s  / as speed_t;
```

If you want to assign a value to the variable, you have to use literals with units, otherwise you will get compile errors:

```
int8_t/m/ length = 3 m;
```

Note that units are computed correctly:

```
speed_t aSpeed = 10 m / 5 s;
speed_t anotherSpeed = 10 m + 5 s; // error; adding apples and oranges
```

Of course, the type system is fully aware of the types and "pulls them through":

```
int8_t/m/ someLength;
int8_t/m/ result = 10 m + someLength;
```

To get values "into" and out of the units world, you can use the `stripunit` and `introduceunit` expressions:

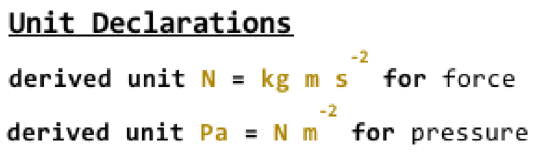
```
int8_t/m/ someLength = 10 m;
int8_t justSomeValue = stripunit[someLength];
int8_t/m/ someLength = introduceunit[justSomeValue -> m];
```

1.2 Derived Units

A derived unit is one that combines several SI units. For example, $N = kg\frac{m}{s^2}$ or $mps = \frac{m}{s}$. Such derived units can be defined with the units extension as well.

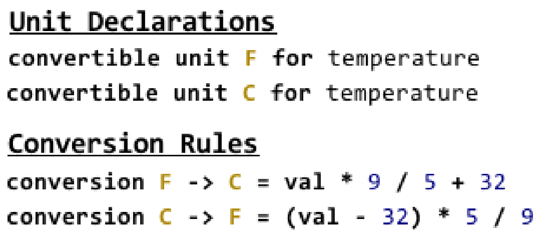
To define derived units, create a `UnitContainer` in your model. There you can define derived units (Fig. 2).

Unit computations in C programs work as expected; compatibility is checked by reducing both to-be-compared units to their SI base units.



```
Unit Declarations
derived unit N = kg m s^-2 for force
derived unit Pa = N m^-2 for pressure
```

Fig. 2. Derived units are combinations of other units.



```
Unit Declarations
convertible unit F for temperature
convertible unit C for temperature

Conversion Rules
conversion F -> C = val * 9 / 5 + 32
conversion C -> F = (val - 32) * 5 / 9
```

Fig. 3. Convertible units require the definition of conversion rules.

1.3 Convertible Units

The derived units discussed in the previous section are still within the SI system and require no value conversions. For convertible units, this is different. They can be declared in the unit container as well; but they need conversion rules to be usable (Fig. 3).
Typically, a convertible unit is a non-SI unit, and the conversion rules map it back to the SI world. Notice how in the conversion rule the `val` keyword represents the value in the original unit.

Conversions in the C programs do not happen automatically, since such a conversion produced runtime overhead. Instead, an explicit conversion has to be used which relies on the conversion rule defined in the units container. A conversion can be added with an intention.

```
int8_t/F/ tempInF = 10 F;
int8_t/C/ tempInC = [tempInF -> C];
```

1.4 Extension with new Units

The units extension can also be "misused" to work with other type annotations such as money, time or coordinate systems. To achieve this, you have to define new elementary unit declarations in a languge extension. Here is some example code with coordinate systems:

```
int8_t/#global/ aGlobalVariable = 10#global;
int8_t/K/ aTemp = aGlobalVariable; // error; #global != K
aGlobalVariable = 10K; // error; #global != K
aGlobalVariable = 230#local; // error; #global != #local
```

In the example above, `#global` and `#local` are two different coordinate systems; technically, both are subtypes of `ElementaryUnitDeclaration`:

```
concept GlobalCoords extends ElementaryUnitDeclaration
                     implements <none>
  concept properties:
    alias = #global
```

Even though these are not technically *convertible* units, we can still define conversion rules:

```
Conversion Rules
  conversion #global -> #local = val + 20
  conversion #local -> #global = val - 20
```

The C code can then use conversions as shown above:

```
int8_t/#local/ aLocalVariable = 20#local;
aGlobalVariable = [aLocalVariable -> #global];
```

The coordinate systems extension is especially useful if combined with a new type and literal, for example, for vectors:

```
intvec/#global/ globalVector = (10,20)#global;
intvec/#local/ localVector = (20,20)#global;
```

2 Components

Modularization supports the divide-and-conquer approach, where a big problem is broken down in to a set of smaller problems that are easier to understand and solve. To make modules reusable in different contexts, modules should define a contract that prescribes how it must be used by client modules. Separating the module contract from the implementation also supports different implementations of the same contract.

Object oriented programming, as well as component-based development exploit this notion. However, C does not support any form of modularization beyond separating sets of functions, `enum`s, `typedef`s etc. into different `.c` and `.h` files. mbeddr, in contrast supports a rich component model.

2.1 Basic Interfaces and Components

To use the components in your C programs, please include the `com.mbeddr.components` devkit.

Interfaces An interface is essentially a set of operation signatures, similar to function prototypes in C. `query` marks functions as not performing any state changes; they are assumed to be invokable any number of times without side effects (something we do not verify automatically at this time).

```
exported interface DriveTrain  {
  void driveForwardFor(uint8_t speed, uint32_t ms)
  void driveBackwardFor(uint8_t speed, uint32_t ms)
  void driveContinouslyForward(uint8_t speed)
  void driveContinouslyBackward(uint8_t speed)
  void stop()
  query uint8_t currentSpeed()
}
```

Components Components can provide and require interfaces via *ports*. A *provided* port means that the component implements the provided interface's operations, and clients can invoke them. These invocations happen via required ports. A *required* port expresses an expectation of a component to be able to call operations on the port's interface. The example below shows a component `RobotChassis` that provides the `DriveTrain` interface shown above, and requires two instances of `EcRobot_Motor`.

```
exported component RobotChassis {

    provides DriveTrain dt
    requires EcRobot_Motor motorLeft
    requires EcRobot_Motor motorRight

    void dt_driveForwardFor(uint8_t speed, uint32_t ms) <- op dt.driveForwardFor {
      motorLeft.set_speed(((int8_t) speed));
      motorRight.set_speed(((int8_t) speed));
      ...
    }
    ...
}
```

Note how the `dt_driveForwardFor` runnable implements the operation `driveForwardFor` from the `dt` provided port. The two signatures are automatically synchronized. Inside components, the operations on required ports can be invoked in the familiar dot notation.

Components can be instantiated. Each component instance generally must get all its required ports connected to provided ports provided by other instances. However, a required port may be marked as `optional` (this is toggled via an intention), in which case, for a given instance, the required port may *not* be connected. Invocations on this required port make no sense in this case, which is why code invoking operations on an optional port must be wrapped in a `with port (optionalReqPort) { .. }` statement. The body of the `with port` is not executed if the port is not connected. The IDE reports an error at editing time if an invocation on an optional port is *not* wrapped this way.

```
exported component RobotChassis {

    provides DriveTrain dt
    requires EcRobot_Motor motorLeft
    requires EcRobot_Motor motorRight
    requires optional ILogging log

    void dt_driveForwardFor(uint8_t speed, uint32_t ms) <- op dt.driveForwardFor {
      with port(log) {
        // other stuff
        log.error(...)
        // more other stuff
      }
    }
    ...
}
```

Components can have fields. These get values for each of the component instances created:

```
exported component OrienterImpl extends nothing {
  ports:
    ...
  contents:
    field int16\_t[5] headingBuffer
    field int8\_t headingIndex

    void orienter\_orientTowards(int16\_t heading, uint8\_t speed, DIRECTION dir) <- ... {
      headingIndex = heading;
    }
}
```

Fields can be marked as `init` fields (via an intention). In this case, when a component is instantiated, a value for the field has to be specified. We will show this below.

We have seen above how a component runnable is tied to the invocation of an operation on a provided port (using `<- op port.operation`). This triggering mechanism can also be used for other events, for example, to react to component instantiation. This effectively supports constructors:

```
exported component OrienterImpl extends nothing {
  contents:
    void init() <- on init {
      compass.initAbsolute();
      compass.heading();
    }
}
```

Instantiation A key difference of mbeddr components compared to C++ classes is that mbeddr component instances are assumed to be allocated and connected during program startup (embedded software typically allocates all memory at program startup to avoid failing during execution), not at arbitrary points in the execution of a program (as in C++ classes). The following piece of code shows an instance configuration:

```
exported instance configuration defaultInstances extends nothing {
  instance RobotChassis chassis
  instance EcRobot_Motor_Impl motorLeft(motorAddress = NXT_PORT_B)
  instance EcRobot_Motor_Impl motorRight(motorAddress = NXT_PORT_C)
  connect chassis.motorLeft to motorLeft.motor
  connect chassis.motorRight to motorRight.motor
}
```

It allocates two instances of the `EcRobot_Motor_Impl` component (each with a different value for its `motorAddress` init parameter) as well as a single instance of `RobotChassis`. The `chassis`' required ports are connected to the provided ports of the two motors. Note that an `instance configuration` just *defines* instances and port connections. The actual *allocation and initialization* of the underlying data structures happens separately in the startup code of the application, for example, in a `main` function:

```
int32_t main(int32_t argc, int8_t*[ ] argv) {
  initialize defaultInstances;
  ...
}
```

To be able to call "into" component instances from regular, non-component C code, adapters can be used. They can be defined inside of instance configurations as well as outside of them. Here is an example:

```
exported instance configuration instances extends nothing {
  instances:
    instance EcRobot_Display_Impl i_display
  connectors:
    ...
  adapter:
    adapt i_display.displayPort as disp
}

void main() {
  initialize instances;
  disp.show("some message");
}
```

Transformation Configuration Components must be able to work potentially with various off-the-shelf middleware solutions such as AUTOSAR. In this case, components will have to be translated differently. Consequently, the transformation for components has to be configured. This happens — like any other configuration — in the `BuildConfiguration`:

```
Configuration Items
  components: no middleware
              wire statically: false
```

The default configuration uses the `no middleware` generator, where components are transformed to plain C function. mbeddr components support polymorphic invocations in the sense that a required port only specifies an *interface*, not the implementing *component*. This way, different implementations can be connected to the same required port (we implement this via a function pointer in the generated C code). This is roughly similar to C++ classes. However, to optimize performance, the generators can also be configured to connect instances statically. In this case, an invocation on a required port is implemented as a direct function call, avoiding additional overhead. This optimization can be performed globally or specifically for a single port. Polymorphism is not supported in this case — users trade flexibility for performance. To do this, select `wire statically: true`. You then have to reference the instance configuration you intend to use:

```
components: no middleware
            wire statically: true instance config: instances
```

If you connect a specific component port to different provided ports in different instances of this component, an error will be reported.

You can also make the decision to wire statically (without polymorphism) on a port-by-port basis, directly in the component (via an intention):

```
requires EcRobot_Compass compass restricted to OrienterImpl.orienter
```

In this case you specify not just the interface but also the particular component and port. This allows the generator to directly refer to the implementing C function — without any overhead.

2.2 Contracts

An additional difference to C++ classes is that mbeddr interfaces support contracts. Operations can specify pre and post conditions, as well as sequencing constraints. Here is the interface from above, but with contract specifications:

```
exported interface DriveTrain {
  void driveForwardFor(uint8_t speed, uint32_t ms)
    pre(0) speed < 100
    post(1) currentSpeed() == 0
    protocol init -> init
  void driveContinouslyForward(uint8_t speed)
    pre(0) speed <= 100
    post(1) currentSpeed() == speed
    protocol init -> forward
  void accelerateBy(uint8_t speed)
    post(1) currentSpeed() == old(currentSpeed()) + speed
    protocol forward -> forward
  query uint8_t currentSpeed()
}
```

The first operation, `driveForwardFor`, requires the `speed` parameter to be below 100. After the operation finishes, `currentSpeed` is zero (notice how the `query` operation `currentSpeed()` is called as part of the post condition). The protocol specifies that, in order to call the operation, the protocol has to be in the `init` state. The post condition for `driveContinouslyForward` expresses that after executing this method the current speed will be the one passed into the operation — in other words, it keeps driving. This is also reflected by the protocol specification which expresses that the protocol will be in the `forward` state. The `accelerateBy` operation can only be called legally while the protocol is in the `forward` state, and it remains in this state. The post condition shows how the value of a `query` operation *before* the execution of the function can be accessed.

The contract is specified on the *interface*. However, the code that checks the contract is generated into the components (i.e. the implementations of the interface operations). The contracts are then checked at runtime.

To add a pre- or postcondition or a protocol, use an intention on the operation. In the inspector, you will have to provide a reference to a message definition that will be `report`ed in case the condition or the protocol fails.

2.3 Mocks

Mocks are used in tests to verify that a component sees a specific behavior at its ports. In mbeddr, a mock verifies the behavior of one specific interface only. Let us look at an example. Here is an interface and a `struct` used by that interface:

```
exported struct DataPacket {
  int8_t size;
  int8_t* data;
};

exported c/s interface PersistenceProvider {
  boolean isReady()
  void store(DataPacket* data)
  void flush()
}
```

This interfaces assumes that clients first call the `isReady` operation, call `store` several times and then call `flush`. We could specify this behavior via contracts shown above. However, we may also want to test that a specific *client* behaves correctly. Here is an example client:

```
exported c/s interface Driver {
  void run()
}

exported component Client extends nothing {
  ports:
    requires PersistenceProvider pers
    provides Driver d
  contents:
    void Driver_run() <- op d.run {
      DataPacket p;
      if ( pers.isReady() ) {
        pers.store(&p);
        pers.flush();
      }
    }
}
```

As we can see, this client does indeed behave correctly. However, if we wanted to write a test to see if it does, we could use a mock to verify it. Here is a mock implememtation for the `PersistenceProvider` interface that verifies this behavior:

```
mock component PersistenceMock {
  report messages: true
  ports:
    provides PersistenceProvider pp
  expectations:
    sequence {
      0: pp.isReady return false;
      1: pp.isReady return true;
      2: pp.store {
          0: parameter data: data != null
        }
      3: pp.flush
    }
    total no. of calls is 4
}
```

It specifies that it expects four invocations in total; the first one should be `isReady` and the mock returns `false`. We then expect `isReady` to be called again, then we expect `store` to be called with a `data` argument not being `null`, and then we expect `flush` to be called.

Here is a test case that uses an instance of the `Client` shown above. It also uses an instance of the `PersistenceMock`:

```
exported test case runTest {
  client.run();
  client.run();
  validate mock mock report messages.mockDidntValidate();
}
```

Notice the `validate mock` statement. If the `mock` instance of `PersistenceMock` has seen anything else but the expected behavior, the `validate mock` statement will fail — and hence the test case will fail.

3 State Machines

State machines can be used to represent state-based behavior in a structural way (they can also be analyzed via model checking, however, we do not discuss this in this section). They are module contents and can be added to arbitrary models. Use the devkit `com.mbeddr.statemachines`.

3.1 Hello State Machine

The following is the simplest possible state machine. It has one state and one in event. Whenever it receives the `reset` event, it goes back to its single state `start`. In other words, it does nothing.

```
statemachine WrappingCounter {
  in events
    reset()
  states ( initial = start )
    state start {
      on reset [ ] -> start {  }
    }
}
```

However, it is a valid state machine and can be used to illustrate some concepts. State machines have in events. These can be "injected" into the state machine from a C program. State machines have one or more states, and one of them must be the initial state. A state may have transitions; a transition reacts to an in event and then points to the new state.

Events can have arguments; they are declared along with the event. Notice how the `int` type requires the specification of bounds. This is to simplify model checking.

```
in events:
  reset()
  increment(int[0..10] delta)
```

A state machine can also have local variables. While these could in principle be handled via separate states, local variables are more scalable.

```
local variables
  int[0..100] current = 0
  int[0..100] LIMIT = 100
  int[0..100] steps = 0
```

We are now ready to write a somewhat more sensible `WrappingCounter` state machine. Whenever we enter the `start` state, we set the `current` variable and the `steps` variable to zero (entry and exit actions can be added to a state via an intention). We have two transitions: if the `increment` event arrives, we go to the `increasing` state, incrementing the `current` value by the `delta`, passed in via the event. Notice how in transition actions we can access the arguments of the event that triggered the respective transition.

```
state start {
  entry {
    current = 0;
    steps = 0;
  }
  on increment [ ] -> increasing { current = current + delta; }
  on reset [ ] -> start {  }
}
```

Let us look at the `increasing` state. There we also react to the `increment` event. But we use guard conditions to determine which transition fires. We can also access event arguments in the guard condition.

```
state increasing {
  entry { steps++; }
  on increment [current + delta <= LIMIT] -> increasing { current = current + delta; }
  on increment [current + delta > LIMIT] -> start { current = 0; }
  on reset [ ] -> start {  }
}
```

3.2 Integrating with C code

State machines can be instantiated. They act as a type, so they can be used in local variables, arguments or in global variables:

```
var WrappingCounter wc;
```

The `trigger` statement is used to "inject" events from regular C code. Notice how we pass in the argument to the `increment` event.

```
void someFunctionCalledByADriver(int8_t ticks) {
  trigger(wc, increment(ticks));
}
```

State machines can also have out events. These are a means to define abstractly some kind of interaction with the environment. Currently, they can be bound to a C function:

```
out events
  wrapped(int[0..100] steps) -> wrapped
```

These out events can then be fired from an action in the state machine, for example, in the exit action of the `increasing` state:

```
exit { send wrapped(steps); }
```

3.3 The complete WrappingCounter state machine

```
module WrappingCounterModule imports nothing {

  statemachine WrappingCounter {
    in events
      increment(int[0..10] delta)
      reset()
    out events
      wrapped(int[0..100] steps) -> wrapped
    local variables
      int[0..100] current = 0
      int[0..100] LIMIT = 100
      int[0..100] steps = 0
    states ( initial = start )
      state start {
        entry {
          current = 0;
          steps = 0;
        }
        on increment [ ] -> increasing { current = current + delta; }
        on reset [ ] -> start {  }
      }
      state increasing {
        entry { steps++; }
        on increment [current + delta <= LIMIT] -> increasing { current = current + delta; }
        on increment [current + delta > LIMIT] -> start { current = 0; }
        on reset [ ] -> start {  }
        exit { send wrapped(steps); }
      }
  }

  void wrapped(int8\_t steps) {
    // do something
  }

  var WrappingCounter wc;

  void someFunctionCalledByADriver(int8_t ticks) {
    trigger(wc, increment(ticks));
  }
}
```

3.4 Testing State Machines

In addition to model checking (discussed in the chapter on analyses) state machines can also be checked regularly. The following piece of test code checks if the state machine works correctly:

```
exported test case testTheWrapper {
  trigger(wc, reset);
  assert(0) isInState(wc, start);
  trigger(wc, increment(5));
  assert(1) isInState(wc, increasing);
  assert(2) wc.current == 5;
  assert(3) wc.steps == 1;
}
```

Notice that in order to be able to access the variables `current` and `steps` from outside the state machine, these variables have to be marked as `readable` (via an intention).

There is also a shorthand for checking if a state machine reacts correctly to events in terms of the new current state:

```
exported test case testTheWrapper {
  ...
  test statemachine wc {
    reset -> start
    increment(5) -> increasing
    increment(90) -> increasing
    increment(10) -> start
  }
}
```

4 Exceptions

To be documented later.