mbeddr C Extensions User's Guide

Markus Voelter

independent/itemis

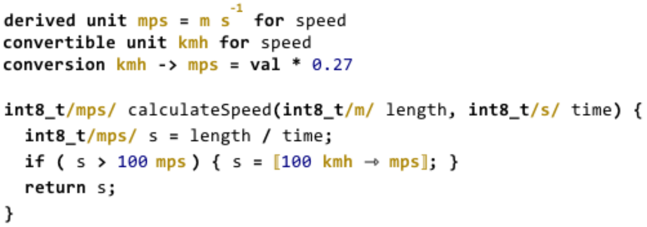
Abstract. This document describes the *mbeddr.ext* package that currently contains physical units, exceptions, components and state machines.

This document is part of the
mbeddr project at `http://mbeddr.com`.

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.

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.

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));
      ...
    }
    ...
}
```

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`, 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 IDE reports an error at editing time if an invocation on an optional port is *not* wrapped this way.

This is an example of how better language abstractions improve analyzability, a major goal of mbeddr. If a regular `if` statement were used to check whether the port is connected for an instance, then there would no easy way to check statically whether the code in the `if`'s body is really executed only if the port is connected. This is because the condition in an `if` statement may be arbitrarily complex, making static analysis hard. The `with port` statement does not allow arbitrary expressions in its "condition" — it only supports referencing a required port, making static analysis trivial. The `with port` statement is restricted to be used inside component operation implementations. In regular functions where it would make no sense, it is not available. This keeps the language clean and helps avoid confusing users with out-of-context keywords.

It is instructive to compare mbeddr components to C++ classes. 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.

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` configuration 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;
  ...
}
```

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 `driveContinously- Forward` 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. We are will investigate whether these specifications can be checked statically as part of our future work.

3 State Machines

4 Exceptions