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

mbeddr C User Guide

This document focuses on the C programmer who wants to exploit the benefits of the extensions to C provided by mbeddr out of the box. We assume that you have some knowledge of regular C (such asK&R C, ANSI C or C99). We also assume that you realize some of the shortfalls of C and are "open" to the improvements in mbeddr C. The main point of mbeddr C is the ability to extend C with domain-specific concepts such as state machines, components, or whatever you deem useful in your domain. We have also removed some of the "dangerous" features of C that are often prohibited from use in real world projects.

This document does not discuss how to develop new languages or extend existing languages. We refer to the *Extension Guide* instead. It is available from **http://mbeddr.com**.

Contents

1 Tutorial

This tutorial showcases many of the features of mbeddr in an integrated example. The sources are available from the download page at **`mbeddr.com`**. Notice that the tutorial does not discuss every aspect of every mbeddr extension — please refer to the respective chapters in the user guide.

This tutorial assumes that you have successfully run through the Hello World tutorial in Section **??**. If you haven't, you can use one of the Wizards in the **`Code`** menu to create a **`Minimal Test Case`**. From this, the starting point for our tutorial looks as follows:

- We have a default type size configuration.
- We have a build configuration with one **`Executable`**.
- We have an **`ImplementationModule`** with a **`main`** function.

1.1 Function Pointers

1.1.1 The Basic Program

As the first example, we will add a configurable event handler using function pointers. We create a new module **`FunctionPointers`** using the context menu **`New -> c.m.core.module -> ImplementationModule`** on the current model.

Inside it, we will add a **`struct`** called **`DataItem`** that contains two members. You create the **`struct`** by just typing **`struct`** inside the module. You add the members by simply starting to type the **`int8`** types.

```
struct DataItem {
  int8 id;
};
```

We then create two functions that are able to process the **DataItem**s. Here is one function that does nothing (intentionally). You enter this function by starting out with the **DataItem** type, then typing the name and then using the **(** to actually create the function (the thing has been a global variable up to this point!):

```
DataItem process_doNothing(DataItem e) {
  return e;
}
```

Other functions with the same signature may process the data in some specific way; We can generalize those into a function type using a **typedef**. Note that entering the function type **()=>()** is in fact a little bit cumbersome. The alias for entering it is **funtype**:

```
typedef (DataItem)=>(DataItem) as DataProcessorType;
```

We can now create a global variable that holds an instance of this type and that acts as a global event dispatcher. We also create a new, empty **test case** that we will use for making sure the program actually works. In the test we assign a reference to **process_doNothing** to that event handler.

```
DataProcessorType processor;
exported test case testProcessing {
  processor = :process_doNothing;
}
```

We can now write the first simple test:

```
exported test case testProcessing {
  processor = :process_doNothing;
  DataItem i1;
  i1.id = 42;
  DataItem i2 = processor(i1);
  assert(0) i2.id == 42;
}
```

Let us complete this into a runnable system. In the **Main** module we change our **main** function to run our new test. Note how we import the **FunctionPointers** module; we call the test case, which is visible because it is **exported**:

```
module Main imports FunctionPointers {
  exported int32 main(int32 argc, string*[] argv) {
    return test testProcessing;
  }
}
```

Looking at the build configuration we see an error that complains that the binary is inconsistent, because the **FunctionPointers** module is not included. We can fix this with a quick fix. This results in the following binary:

```
executable MbeddrTutorial isTest: true {
  used external libraries
  used mbeddr libraries
  included modules
    Main (mbeddr.tutorial.main.m1)
    FunctionPointers (mbeddr.tutorial.main.m1)
}
```

1.1.2 Building and Running

We can now build the system using **Ctrl-F9** or by selecting **Rebuild** from the context menu of the solution in the logical view on the left. If you have installed **gcc** correctly the binary should actually be compiled automatically. Here is the info message you should get in the MPS messages view:

```
make finished successfully for mbeddr.tutorial.main/mbeddr.tutorial.main.m1
```

Let us run this program from the command line. To get to the respective location in the file system, select the solution in the logical view, open the properties and copy the **Solution File** location. In my case it is:

```
/Users/markusvoelter/Documents/mbeddr/mbeddr.core/code/applications/tutorial/solutions/
    mbeddr.tutorial.main/mbeddr.tutorial.main.msd
```

We can open a console and **cd** to this directory, removing the last segment **mbeddr.tutorial.main.msd** from the path. In that directory we can **cd** to **source_gen**, and from there we can navigate down to the directory for the model: **cd mbeddr/tutorial/m1**

Using **ls** there, we can see the following:

```
$ ls -ll
total 104
-rw-r--r--  1 markusvoelter  staff   1189 Oct 30 21:11 FunctionPointers.c
-rw-r--r--  1 markusvoelter  staff    159 Oct 30 21:11 FunctionPointers.h
-rwxr-xr-x  1 markusvoelter  staff   9028 Oct 30 21:11 Main
-rw-r--r--  1 markusvoelter  staff    338 Oct 30 21:11 Main.c
-rw-r--r--  1 markusvoelter  staff    162 Oct 30 21:11 Main.h
-rw-r--r--  1 markusvoelter  staff    433 Oct 30 21:11 Makefile
drwxr-xr-x  4 markusvoelter  staff    136 Oct 30 21:11 bin
-rw-r--r--  1 markusvoelter  staff    943 Oct 30 21:11 module_dependencies.gv
-rw-r--r--  1 markusvoelter  staff  14069 Oct 30 21:11 trace.info
```

Importantly, we see a **Makefile**, so we can call **make**. This will build the binary:

```
$ make
rm -rf ./bin
mkdir -p ./bin
```

```
gcc -std=c99   -c -o bin/Main.o Main.c
mkdir -p ./bin
gcc -std=c99   -c -o bin/FunctionPointers.o FunctionPointers.c
gcc -std=c99 -o Main bin/Main.o bin/FunctionPointers.o
```

This results in an executable **MbeddrTutorial**. We can run it by calling **./MbeddrTutorial** or by calling **make test**. The output should look as follows:

```
$ make test
./MbeddrTutorial
$$runningTest: running test () @FunctionPointers:test_testProcessing:0#767515563077315487
```

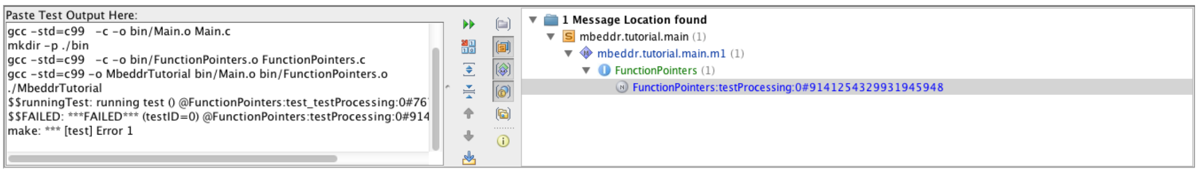
So the test succeeded, everything seems fine. Let us try to introduce an error by somehow breaking the assertion.

```
assert(0) i2.id == 0;
```

After regenerating the code (**Ctrl-F9** or **Rebuild**) we can call **make test** again and we get:

```
$ make test
mkdir -p ./bin
gcc -std=c99   -c -o bin/Main.o Main.c
mkdir -p ./bin
gcc -std=c99   -c -o bin/FunctionPointers.o FunctionPointers.c
gcc -std=c99 -o MbeddrTutorial bin/Main.o bin/FunctionPointers.o
./MbeddrTutorial
$$runningTest: running test () @FunctionPointers:test_testProcessing:0#767515563077315487
$$FAILED: ***FAILED*** (testID=0) @FunctionPointers:testProcessing:0#9141254329931945948
make: *** [test] Error 1
```

As you can see the test fails. It says that in the **FunctionPointers** module, **testProcessing** test case, assert number **0**. You can either navigate to the offending **assert** manually. Alternatively you can copy the console output into the clipboard and then paste it into the text box into the window that opens from the **Analyze -> mbeddr Analyze Error Output** menu:



Double clicking on the respective line opens the editor on the offending **assert** statement.

1.2 Physical Units

Let us go back to the definition of **DataItem** and make it more useful. Our application is supposed to work with tracking data (captured from a bike computer or a flight logger). We change the **struct** as follows (note that renaming the thing is trivial — you just change the name!):

```
struct Trackpoint {
  int8 id;        // sequence ID of the track point
  int8 timestamp; // timestamp as taken from GPS time
  int8 x;         // longtitude, simplified as a number
  int8 y;         // latitude, simplified as a number
  int8 alt;       // altitude as of the GPS
  int8 speed;     // current speed, if available
};
```

We can now enhance our test case the following way:

```
exported test case testProcessing {
  Trackpoint i1 = {
    id = 1
    x = 0
    y = 0
    alt = 100
  };
  processor = :process_doNothing;
  Trackpoint i2 = processor(i1);
  assert(0) i2.id == 1 && i2.alt == 100;
```

We can now use physical units to add more semantics to this data structure. We add the **com.mbeddr.physicalunits** devkit to the model properties. Then we can add units to the members. To add a unit, simply press **/** at the right side of one of the **int8** types:

```
struct Trackpoint {
  int8 id;
  int8/s/ timestamp;
  int8/m/ x;
  int8/m/ y;
  int8/m/ alt;
  int8 speed;
};
```

s and **m** are SI base units, so they are available by default. For the **speed** member we need to add **m/s**. Since this is not an SI base unit we first have to define it. To do so we create a new **UnitContainer** root in the current model (you can find the **UnitContainer** in the **c.m.ext.physicalunits** submenu of the **New** context menu on the current model). In it we can create a **derived unit**. Entering the **mps** and **speed** is trivial. Entering the meters per second is a bit cumbersome right now: press **m** for the meters, press **Enter**, press **s** for the seconds and then type **-1** directly on the right side of the **s** to enter the exponent:

```
Unit Configuration
                          -1
derived unit mps = m s      for speed
```

We can now go back to the **Trackpoint** and make the **speed** property use a unit: **int8/mps/ speed;**

Adding these units results in errors in the existing code because you cannot simply assign a plain number to a variable or member whose type includes a physical unit (**int8/m/ length = 3;** is illegal). Instead you have to add units to the literals as well. You can simply type the unit after the literal to get to the following:

```
Trackpoint i1 = {
  id = 1
  timestamp = 0 s
  x = 0 m
  y = 0 m
  alt = 100 m
};
...
assert(0) i2.id == 1 && i2.alt == 100 m;
...
assert(1) i3.id == 1 && i3.alt == 0 m;
```

If you try to rebuild now you will run into build errors:

```
no configuration item "physical units" found in this model. Please add a configuration item
     in your Build Configuration.
```

To fix this we have to go to the build configuration and add the respective configuration item:

```
Configuration Items
  reporting: printf (add labels false)
  physical units (config = Units Declarations (mbeddr.tutorial.main.m1))
```

If we rebuild now, everything should generate normally and we should be able to run the test again. Nothing should have changed so far. However, if we were to write the following code, we would get an error:

```
int8 someInt = i1.x + i1.speed; // error, adding apples and pears
```

The problem with this code is that you cannot add a length (**i1.x**) and a speed (**i1.speed**). And the result is certainly not a plain **int8**, so you cannot assign the result to **someInt**. Adding **i1.x** and **i1.y** will work, though. Also, you can calculate with units to write the following code. This gives you an additional level of type safety.

```
Trackpoint i4 = {
  id = 1
  timestamp = 10 s
  x = 100 m
  y = 0 m
  alt = 100 m
};
int8/mps/ speed = (i4.x - i2.x) / (i4.timestamp - i2.timestamp);
```

1.3 Components

Let us now introduce components to further structure the system. We start by factoring the **Trackpoint** data structure into a separate module and export it to make it accessible from importing modules.

```
module DataStructures imports nothing {
  exported struct Trackpoint {
    int8 id;
    int8/s/ timestamp;
    int8/m/ x;
    int8/m/ y;
    int8/m/ alt;
    int8/mps/ speed;
  };
}
```

1.3.1 An Interface with Contracts

We now define an interface that handles **Trackpoint**s. To be able to do that we have to add the **com.mbeddr.components** devkit to the current model. We can then enter a client-server interface in a new module **Components**. We use pointers for the trackpoints here to optimize performance. Note that you can just press * on the right side of **Trackpoint** to make it a **Trackpoint***:

```
module Components imports DataStructures {

  exported cs interface TrackpointProcessor {
    Trackpoint* process(Trackpoint* p);
  }

}
```

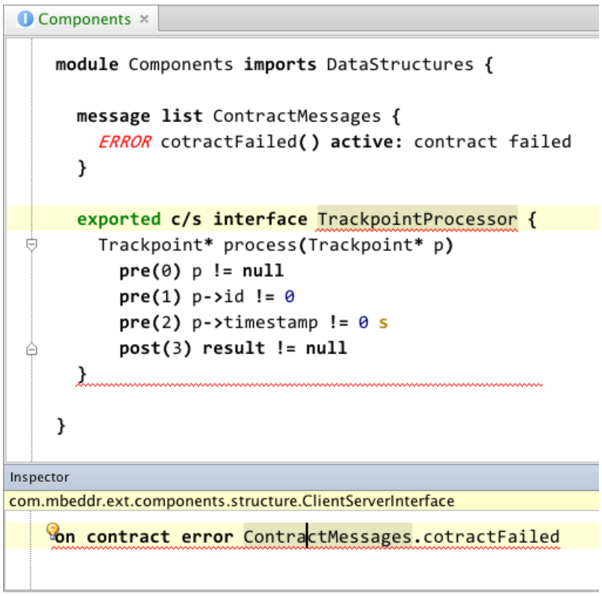
To enhance the semantic "richness" of the interface we can add preconditions. To do so, use an intention **Add Precondition** on the operation itself. Please add the following pre- and postconditions (note how you can of course use units in the precondition):

```
Trackpoint* process(Trackpoint* p)
  pre(0) p != null
  pre(1) p->id != 0
  pre(2) p->timestamp != 0 s
  post(3) result->id != 0
```

After you have added these contracts, you will get an error message on the interface. The problem is this: if a contract (pre- or postcondition) fails, the system will report a message (this message can be deactivated in case you don't want any reporting). However, for the program to work you have to specify a message on the interface. We create a new message list and a messge:

```
messagelist ContractMessages {
  ERROR contractFailed() active: contract failed
}
```

You can now open the inspector for the interface and reference this message from there:



There are still errors. The first one complains that the message list must be exported if the interface is exported. We fix it by exporting the message list (via an intention). The next error complains that the message needs to have to integer arguments to represent the operation and the pre- or postcondition. We change it thusly:

```
exported messagelist ContractMessages {
  ERROR contractFailed(int8 op, int8 pc) active: contract failed
}
```

1.3.2 A First Component

Let us create a new component by typing **component**. We call it **Nuller**. It has one provided port called **processor** that provides the **TrackpointProcessor** interface:

```
exported component Nuller extends nothing {
  provides TrackpointProcessor processor
}
```

We get an error that complains that the component needs to implement the operations defined by the **TrackpointProcessor** interface; we can get those automatically generated by using a quick fix on the provided port. This gets us the following:

```
exported component Nuller extends nothing {
  provides TrackpointProcessor processor
  Trackpoint* processor_process(Trackpoint* p) <- op processor.process {
    return null;
  }
}
```

The **processor_process** runnable is triggered by an incoming invocation of the **process** operation defined in the **TrackpointProcessor** interface. The **Nuller** simply sets the altitute to zero:

```
Trackpoint* processor_process(Trackpoint* p) <- op processor.process {
  p->alt = 0 m;
  return p;
}
```

Let us now write a simple test case to check this component. To do that, we first have to create an instance of **Nuller**. We create an instance configuration that has an instance of this component. Also, we add an adapter. An adapter makes a provided port of a component instance (**Nuller.processor**) available to a regular C program under the specified name **n**:

```
instances nullerInstances extends nothing {
  instance Nuller nuller
  adapt n -> nuller.processor
}
```

Now we can write a test case that accesses the **n** adapter — and through it, the **processor** port of the **Nuller** component instance **nuller**. We create a new **Trackpoint**, using 0 as the **id** — intended to trigger a contract violation (remember **pre(1) p->id != 0**). To enter the **&tp** just enter a **&**, followed by **tp**:

```
exported test case testNuller {
  Trackpoint tp = {
    id = 0
```

```
  };
  n.process(&tp);
}
```

Before we can run this, we have to make sure that the **instances** are initialized (cf. the warning you get on them). We do this right in the test case:

```
exported test case testNuller {
  initialize instances;
  Trackpoint tp = {
    id = 0
  };
  n.process(&tp);
}
```

To make the system work, you have to import the **Components** module into the **Main** module so you can call the **testNuller** test case from the **test** expression in **Main**. In the build configuration, you have to add the missing modules to the executable (using the quick fix). Finally, also in the build configuration, you have to add the **components** configuration item:

```
Configuration Items:
  reporting: printf (add labels false)
  physical units (config = Units Declarations (mbeddr.tutorial.main.m1))
  components: no middleware
              wire statically: false
```

You can now rebuild and run. As a result, you'll get contract failures:

```
./MbeddrTutorial
$$runningTest: running test () @FunctionPointers:test_testProcessing:0#767515563077315487
$$runningTest: running test () @Components:test_testNuller:0#767515563077315487
$$contractFailed: contract failed (op=0, pc=1) @Components:null:-1#1731059994647588232
$$contractFailed: contract failed (op=0, pc=2) @Components:null:-1#1731059994647588253
```

We can fix these problems by changing the test data to conform to the contract:

```
Trackpoint tp = {
  id = 10
  timestamp = 10 s
  alt = 100 m
};
n.process(&tp);
assert(0) tp.alt == 0 m;
```

Let us provoke another contract violation by returning from the implementation in the **Nuller** component a **Trackpoint** whose **id** is 0:

```
Trackpoint* processor_process(Trackpoint* p) <- op processor.process {
  p->alt = 0 m;
  p->id = 0;
  return p;
```

```
}
```

Running it again provokes another contract failure. Notice how the contract is specified on the *interface*, but they are checked for each *component* implementing the interface. There is no way how an implementation can violate the interface contract without the respective error being reported!

1.3.3 Collaborating and Stateful Components

Let us look at interactions between components. We create a new interface, the **`TrackpointStore`**. It can store and return trackpoints[1]. Here is the basic interface:

```
exported cs interface TrackpointStore1 {
  void store(Trackpoint* tp)
  Trackpoint* get()
  Trackpoint* take()
  boolean isEmpty()
}
```

Let us again think about the semantics: you shouldn't be able to get or take stuff from the store if it is empty, you should not put stuff into it when it is full, etc. These things can be expressed as pre- and postconditions. The following should be pretty self-explaining. The only new thing is the **`query`** operation. Queries can be used from inside pre- and postconditions, but cannot modify state[2]

```
exported cs interface TrackpointStore1 {
  void store(Trackpoint* tp)
    pre(0) isEmpty()
    pre(1) tp != null
    post(2) !isEmpty()
  Trackpoint* get()
    pre(0) !isEmpty()
  Trackpoint* take()
    pre(0) !isEmpty()
    post(1) result != null
    post(2) isEmpty()
  query boolean isEmpty()
}
```

These pre- and postconditions mostly express a valid sequence of the operation calls: you have to call **`store`** before you can call **`get`**, etc. This can be expressed directly with protocols:

[1]Sure, it is completely overdone to separate this out into a separate interface/component, but for the sake of the tutorial it makes sense.

[2]Currently this is not yet checked. But it will be.

```
exported cs interface TrackpointStore2 {

  // store goes from the initial state to a new state full
  void store(Trackpoint* tp)
    protocol init(0) -> new full(1)

  // get expects the state to be full, and remains there
  Trackpoint* get()
    protocol full -> full

  // take expects to be full and then becomes empty (i.e. init)
  Trackpoint* take()
    post(0) result != null
    protocol full -> init(0)

  // and isEmpty has no effect on the protocol state
  query boolean isEmpty()
}
```

The two interfaces are essentially equivalent, and both are checked at runtime and lead to errors if the contract is violated.

We can now implement a component that provides this interface. Most of the following code should be easy to understand based on what we have discussed so far. There are two new things. There is a field **`Trackpoint* storedTP;`** that represents component state. Second there is an **`on-init`** runnable: this is essentially a constructor that is executed as an instance is created.

```
exported component InMemoryStorage extends nothing {
  provides TrackpointStore1 store
  Trackpoint* storedTP;

  void init() <- on init {
    storedTP = null;
  }

  void trackpointStore_store(Trackpoint* tp) <- op store.store {
    storedTP = tp;
  }
  Trackpoint* trackpointStore_get() <- op store.get {
    return storedTP;
  }
  Trackpoint* trackpointStore_take() <- op store.take {
    Trackpoint* temp = storedTP;
    storedTP = null;
    return temp;
  }
  boolean trackpointStore_isEmpty() <- op store.isEmpty {
    return storedTP == null;
  }
}
```

To keep our implementation module **`Components`** well structured we can use sectios. A **`section`** is a named part of the implementation module that has no semantic effect beyond that. Sections can be collapsed.

```
module Components imports DataStructures {

  exported messagelist ContractMessages {...}

  section processor {...}

  section store {
    exported cs interface TrackpointStore1 {
      ...
    }
    exported cs interface TrackpointStore2 {
      ...
    }
    exported component InMemoryStorage extends nothing {
      ...
    }
  }

  instances nullerInstances {...}
  test case testNuller {...}
  instances interpolatorInstances {...}
  exported test case testInterpolator { ... }
}
```

We can now implement a second processor. For subsequent calls of **`process`**, it computes the average of the two last speeds of the passed trackpoints. Let us start with the test case. Note how **`p2`** has its speed changed to the average of the **`p1`** and **`p2`** originally.

```
exported test case testInterpolator {
  initialize interpolatorInstances;
  Trackpoint p1 = {
    id = 1
    timestamp = 1 s
    speed = 10 mps
  };
  Trackpoint p2 = {
    id = 1
    timestamp = 1 s
    speed = 20 mps
  };

  ip.process(&p1);
  assert(0) p1.speed == 10 mps;

  ip.process(&p2);
  assert(1) p2.speed == 15 mps;
}
```

Let us look at the implementation of the **`Interpolator`**. Here it is.

```
exported component Interpolator extends nothing {
  provides TrackpointProcessor processor
  requires TrackpointStore1 store
  init int8 divident;
  Trackpoint* processor_process(Trackpoint* p) <- op processor.process {
    if (store.isEmpty()) {
      store.store(p);
      return p;
```

```
    } else {
      Trackpoint* old = store.take();
      p->speed = (p->speed + old->speed) / divident;
      store.store(p);
      return p;
    }
  }
}
```

A few things are worth mentioning. First, the component **requires** another one. More specifically it only expresses a requirement towards an interface, **TrackpointStore1** in our case. Any component that implements this interface can be used to fulfil this requirement (we'll discuss how below). Second, we use an **init** field. This is a regular field from the perspective of the component (i.e. it can be accessed from within the implementation), but it is special in that a value for it has to be supplied when the component is instantiated. Third, this example shows how to call operations on required interfaces (**store.store(p);**).

The only remaining step before running the test is to define the instances. Here is the code:

```
instances interpolatorInstances extends nothing {
  instance InMemoryStorage store
  instance Interpolator ip(divident = 2)
  connect ip.store to store.store
  adapt ip -> ip.processor
}
```

Two interesting things. First, notice how we pass in a value for the init field **divident** as we define an instance of **Interpolator**. Second, we use **connect** to connect the required port **store** of the **ip** instance to the **store** provided port of the **store** instance. If you don't do this you will get an error on the **ip** instance since it *requires* this thing to be connected (there are also **optional** required ports which may remain unconnected).

You can run the test case now. On my machine here it works successfully :-)

1.3.4 Mocks

Todo

1.4 Decision Tables

Let us implement another interface, one that lets us judge flights (we do this in a new section in the **Components** module). The idea is that clients add trackpoints, and the **FlightJudger** computes some kind of score from it (consider some kind of biking/flying competition as a context):

```
exported cs interface FlightJudger {
  void reset()
  void addTrackpoint(Trackpoint* tp)
  int16 getResult()
}
```

Here is the basic implementation of a component that provides this interface.

```
exported component Judge extends nothing {
  provides FlightJudger judger
  int16 points = 0;
  void judger_reset() <- op judger.reset {
    points = 0;
  }
  void judger_addTrackpoint(Trackpoint* tp) <- op judger.addTrackpoint {
    points += 0; // to be changed
  }
  int16 judger_getResult() <- op judger.getResult {
    return points;
  }
}
```

Of course the implementation of **addTrackpoint** that just adds **0** to the **points** doesn't make much sense yet. The amount of points added should depend on how fast and how high the plane (or whatever) was going. The following screenshot shows an embedded decision table that computes points:

```
void judger_addTrackpoint(Trackpoint* tp) ← op judger.addTrackpoint {
  points += int16, 0
```

	tp->alt < 2000 m	tp->alt >= 2000 m
tp->speed < 150 mps	0	10
tp->speed >= 150 mps	5	20

```
;
} runnable judger_addTrackpoint
```

Notice h

Bibliography