mbeddr C

An extensible version of the C programming language for Embedded Programming

C the Difference - C the Future
What if...

you could change languages like you can change programs?
A Test, written in essentially normal C

```c
module WriteATestCase from cdesignpaper.unittest imports nothing {

  var int8_t failedTests;

  int32_t main(int32_t argc, int8_t*[ ] argv) {
    testMultiply();
    return failedTests;
  } main (function)

  void testMultiply() {
    if ( times2(21) != 42 ) { failedTests++; } if
  } testMultiply (function)

  int8_t times2(int8_t a) {
    return 2 * a;
  } times2 (function)
}
```
The same test, but now using additional language concepts from the unit test extension

```plaintext
module UnitTestDemo from cdesignpaper.unittest imports nothing {

    int32_t main(int32_t argc, int8_t*[ ] argv) {
        return test testMultiply;
    } main (function)

    exported test case testMultiply {
        assert(0) times2(21) == 42;
    } testMultiply(test case)

    int8_t times2(int8_t a) {
        return 2 * a;
    } times2 (function)
}
```
The same test, but now using additional language concepts from the unit test extension.

Test Cases are a kind of void function, but with adapted syntax.
The same test, but now using additional language concepts from the unit test extension.

```plaintext
module UnitTestDemo from cdesignpaper.unittest imports nothing {

    int32_t main(int32_t argc, int8_t*[ ] argv) {
        return test testMultiply;
    } main (function)

    exported test case testMultiply {
        assert(0) times2(21) == 42;
    } testMultiply(test case)

    int8_t times2(int8_t a) {
        return 2 * a;
    } times2 (function)
}
```

Asset Statements check conditions; they are restricted to be used only in test cases.
The same test, but now using additional language concepts from the unit test extension

```c
module UnitTestDemo from cdesignpaper.unittest imports nothing {

  int32_t main(int32_t argc, int8_t*[ ] argv) {
      return test testMultiply;
  } main (function)

  exported test case testMultiply {
      assert(0) times2(21) == 42;
  } testMultiply(test case)

  int8_t times2(int8_t a) {
      return 2 * a;
  } times2 (function)
}
```

A special expression that executes tests, and evaluates to the number of failed tests (which is then returned to the OS here)
The unit testing extensions are implemented in separate language module.

The constructs become available to programmers only if they import the respective language module into their program.

This keeps the overall language clean --- a precondition for building extensions targeting different audiences.
mbeddr C Approach
An extensible C with support for formal methods, requirements and PLE.
IDE for Everything
A debugger for all of that

- The Debugger debugs the code on the level of the extensions!

- When defining new language concepts, language developers also specify how these concepts should be debugged.
SDK for building your own Language Extensions!

This SDK is essentially MPS 😊, plus some custom documentation.
IDE for Everything

JetBrains MPS
Open Source Language Workbench

- Apache 2.0
- Available at http://jetbrains.com/mps
Challenges in embedded software development
Abstraction without Runtime Cost

- Abstractions are important to write maintainable and analyzable software; however,
- Abstractions should not incur runtime overhead (or at least as little as possible)
C considered unsafe

- void pointers are evil
- standards like MISRA-C prohibit certain constructs from being used in many organizations
Program Annotations

- Things like physical units, value ranges, or access patterns to data structures are often defined outside the code program in some kind of XML.
- The C type checker doesn't know about them, a separate checker is used — cumbersome!
Static Checks and Verification

- Model Checking, SAT solving etc. are important to "proof" the correctness of programs, however,
- it is expensive to do on C code since C's abstractions are too low-level
Product Lines and Requirement Traces

- Trace links from code (or other implementation artifacts) back to requirements must be supported.
- Product Line Variability must be handled in a more maintainable way than #ifdefs.
Separate, hard to integrate Tools

- Modeling tools don’t integrate well with each other, or with manually written code
- Modeling tools aren’t really extensible, making them hard to adapt to specific domains
mbeddr C Solution Philosophy
Domains can be seen as specializations of others. Each may require specialized language support.

more specialized domains

more specialized languages
There is a general domain that encompasses all programs writable in C

more specialized domains
more specialized languages
Embedded software is a specialization of C --- requiring special language abstractions.

- more specialized domains
- more specialized languages

Extension

Automotive

Aerospace

Embedded S/W

Stock Trading

All programs writable in C
Automotive or Aerospace are subsequent specializations — ad infinitum, in principle.

- More specialized domains
- More specialized languages
Assume we have a module which contains a component which in turn contains a state machine. How is this compiled?
Assume we have a module which contains a component which in turn contains a state machine. How is this compiled?
In the first step, the state machine is reduced to a component operation that contains e.g. the usual switch/case way of implementing a SM.
In the next step, the component is reduced to a bunch of normal C methods; the contains switch/case statement just remains unchanged.
Finally, we generate text from the C program and feed it into a regular compiler, such as GCC. mbeddr uses incremental reduction!
Language Extension

- The core contains all of C plus a couple of utilities such as namespaces, closures, real boolean types and integration with make.
- A few changes have been made relative to standard C --- these are clearly explained in the docs.
- It is designed to be extensible by users, e.g. it is simple to provide an integration with a custom build infrastructure.

<table>
<thead>
<tr>
<th>Core</th>
</tr>
</thead>
<tbody>
<tr>
<td>C Programming Language</td>
</tr>
<tr>
<td>Expressions, Statements, Modules, Build/Make</td>
</tr>
</tbody>
</table>
These standard extensions are intended to be useful by many embedded software projects. Most of them will become Open Source during 2012.
The SDK lets users build their own language extensions in a modular way --- without changing the existing languages, and independent of other extensions.
Subset of Available Extensions
All of C (cleaned-up)

- no preprocessor (better replacements!), modules/namespaces, unit tests, C99 primitive types required, booleans, binary literals, function references, closures
module Calculator from cdesignpaper.helloWorld imports nothing {

    exported int8_t add(int8_t x, int8_t y) {
        return x + y;
    } add (function)

    exported int8_t multiply(int8_t x, int8_t y) {
        return x * y;
    } multiply (function)
}

module HelloWorld from cdesignpaper.helloWorld imports Calculator {

    int32_t main(int32_t argc, int8_t* [ ] argv) {
        return add(2, 2) + multiply(10, 2);
    } main (function)
}
module Calculator from cdesignpaper.helloWorld imports nothing {

    exported int8_t add(int8_t x, int8_t y) {
        return x + y;
    } add (function)

    exported int8_t multiply(int8_t x, int8_t y) {
        return x * y;
    } multiply (function)
}

module HelloWorld from cdesignpaper.helloWorld imports Calculator {

    int32_t main(int32_t argc, int8_t*[ ] argv) {
        return add(2, 2) + multiply(10, 2);
    } main (function)
}
Retargettable
Build
Integration
Build Configuration for model MultiBot_Test

Target Platform:
desktop
  compiler: gcc
  compiler options: -std=c99
  debug options: -g

Configuration Items
  reporting: printf
  components: no middleware

Binaries
  executable MultiBotTest isTest: true {
    used libraries
    << ... >>
    included modules
    Driver
    TestDriveTrain
    EcRobotAPI
    Messages
    TestOrienter
    DriveTrain
    Orienter
  }
Example: different target used for generating lego NXT Osek make files (special format)
Native Support for Unit Testing and Logging
module UnitTestDemo from cdesignpaper.unittest imports nothing {

    int32_t main(int32_t argc, int8_t* argv) {
        return test testMultiply;
    } main (function)

    exported test case testMultiply {
        assert(0) times2(21) == 42;
        if ( 1 > 2 ) {
            fail(1);
        } if
    } testMultiply(test case)

    int8_t times2(int8_t a) {
        return 2 * a;
    } times2 (function)
}
moduleUnitTestDemo from cdesignpaper.unittest import

int32_t main(int32_t argc, int8_t* argv) {
    return test testMultiply;
} main (function)

eexported test case testMultiply {
    assert(0) times2(21) == 42;
    if (1 > 2) {
        fail(1);
    } if
}
testMultiply(test case)

int8_t times2(int8_t a) {
    return 2 * a;
} times2 (function)
module ARealHelloWorld from cdesignpaper.helloWorld imports nothing {

message list HelloWorldMessages {

    INFO hello(string name) active: Hello World
    ERROR wrongNumberOfArguments(int8_t expected, int8_t actual) active: Wrong number of Arguments
}

int32_t main(int32_t argc, int8_t* [ ] argv) {
    report(0) HelloWorldMessages.wrongNumberOfArguments(1, argc) {
        if ( argc != 1 ) {
            report;
            return 1;
        } if
    };
    report(0) HelloWorldMessages.hello(argv[0]) on/if;
    return 0;
} main (function)
}

message list HelloWorldMessages {

    INFO hello(string name) active: Hello World
    ERROR wrongNumberOfArguments(int8_t expected, int8_t actual) inactive: Wrong number of Arguments
}
module ARealHelloWorld from cdesignpaper.helloWorld imports nothing { 

message list HelloWorldMessages { 
    INFO hello(string name) active: Hello World 
    ERROR wrongNumberOfArguments(int8_t expected, int8_t actual) active: 
} 

int32_t main(int32_t argc, int8_t*[ ] argv) { 
    report(0) HelloWorldMessages.wrongNumberOfArguments(1, argc) { 
        if ( argc != 1 ) { 
            report; 
            return 1; 
        } if 
    } 
    report(0) HelloWorldMessages.hello(argv[0]) on/if; 
    return 0; 
} main (function) 

message list HelloWorldMessages { 
    INFO hello(string name) active: Hello World 
    ERROR wrongNumberOfArguments(int8_t expected, int8_t actual) inactive: 
} 

Messages can be deactivated --- no reporting, zero overhead!
exported c/s interface Orienter on contract error MultibotMessages.prePostconditionFailed {
  int16_t heading()
  post(0) result >= 0 && result <= 359
void orientTowards(int16_t heading, uint8_t speed, DIRECTION dir)
  pre(0) heading >= 0 && heading <= 359
}

exported component OrienterImpl extends nothing {
  ports:
    provides Orienter oriener
    requires EcRobot_Compass compass
    requires EcRobot_Motor motorLeft
    requires EcRobot_Motor motorRight
  contents:
    field int16_t[5] headingBuffer

    void orienter_orientTowards(int16_t heading, uint8_t speed, DIRECTION dir) <- op oriener.orientTowards {
      int16_t currentDir = compass.heading();
      if (dir == COUNTERCLOCKWISE) {
        motorLeft.set_speed(-1 * ((int8_t) speed));
        motorRight.set_speed((int8_t) speed);
        while (currentDir != heading) { currentDir = compass.heading(); } while
      } else {
        motorLeft.set_speed((int8_t) speed);
        motorRight.set_speed(-1 * ((int8_t) speed));
        while (currentDir != heading) { currentDir = compass.heading(); } while
      }
      if
        motorLeft.stop();
        motorRight.stop();
    }

    int16_t orienitor_heading() <- op oriener.heading {
      return compass.heading();
    }
}
Interface with Operations

Optionally with pre- and post conditions --- automatically enforced in every implementing component

Instantiatable, stateful components that provide and require ports

Optional overhead-free translation to plain C - no polymorphism

```c
exported c/s interface Orienter on contract error MultibotMessages.prePostconditionFailed {
    int16_t heading()
    post(0) result >= 0 && result <= 359
    void orientTowards(int16_t heading, uint8_t speed, DIRECTION dir)
    pre(0) heading >= 0 && heading <= 359
}

exported component OrienterImpl extends nothing {
    ports:
        provides Orienter oriener
        requires EcRobot_Compass compass
        requires EcRobot_Motor motorLeft
        requires EcRobot_Motor motorRight
    contents:
        field int16_t[5] headingBuffer

        Orienter_orientTowards(int16_t heading, uint8_t speed, DIRECTION dir) {
            int currentDir = compass.heading();
            if (dir == COUNTERCLOCKWISE) {
                motorLeft.set_speed(-1 * ((int8_t) speed));
                motorRight.set_speed(((int8_t) speed));
                while (currentDir != heading) { currentDir = compass.heading(); }
            }
            motorLeft.set_speed(((int8_t) speed));
            motorRight.set_speed(-1 * ((int8_t) speed));
            while (currentDir != heading) { currentDir = compass.heading(); }
        }

        oriener_heading() <- op oriener.heading {
            compass.heading();
        }
```
exported test case testDriveTrain {
    initialize instances;
    assert(0) dt.currentSpeed() == 0;
    dt.driveContinuouslyForward(50);
    dt.stop();
    validate mock motorLeft;
    validate mock motorRight;
} testDriveTrain(test case)

instance configuration instances extends nothing {
    instances:
        instance MotorLeftMock motorLeft
        instance MotorRightMock motorRight
        instance DriveTrainImpl driveTrain
        instance EcUtil util
    connectors:
        connect driveTrain.motorLeft to motorLeft.motor
        connect driveTrain.motorRight to motorRight.motor
        connect driveTrain.util to util.util
    adapter:
    << ... >>
}

mock component MotorLeftMock {
    report messages: true
    ports:
        provides EcRobot_Motor motor
    expectations:
        total no. of calls is 2
        sequence {
            0: motor.set_speed {
                0: parameter speed: speed == 50
            }
            1: motor.stop
        }
}

mock component MotorRightMock {
    report messages: true
    ports:
        provides EcRobot_Motor motor
    expectations:
        total no. of calls is 2
        sequence {
            0: motor.set_speed {
                0: parameter speed: speed == 50
            }
            1: motor.stop
        }
}
Test case uses mocks; if behavior is different from specified expected behavior, the test fails.
State Machines
+
Model Checking
verifiable

statemachine Counter {
  in start() <no binding>
    step(int[0..10] size) <no binding>
  out someEvent(int[0..100] x, boolean b) => handle_someEvent
    resetted() => resetted
  vars int[0..100] currentVal = 0
    int[0..100] LIMIT = 10
  states (initial = initialState)
    state initialState {
      on start [ ] -> countState { send someEvent(100, true && false || true); }
    }
    state countState {
      on step [currentVal + size > LIMIT] -> initialState { send resetted(); }
      on step [currentVal + size <= LIMIT] -> countState { currentVal = currentVal + size; }
      on start [ ] -> initialState { }
    }
} end statemachine
```plaintext
verifiable
statemachine Counter {
  in start() <no binding>
    step(int[0..10].size) <no binding>
  out someEvent(int[0..100] x, boolean b) => handle_someEvent
    reseted() => reseted
  vars int[0..100] currentVal = 0
  int[0..100] LIMIT = 10
  states (initial = initialState)
  state initialState {
    on start [ ] -> countState { send someEvent(100, true && f }
  } state countState {
    on step [currentVal + size > LIMIT] -> initialState { send reseted(); }
    on step [currentVal + size <= LIMIT] -> countState { currentVal = currentVal + size; }
    on start [ ] -> initialState { }
  } end statemachine
}
```

- **States with entry and exit actions**
- **Statemachine-local variables**
- **Transition using C expressions as guards**
- **Transitions also have transition actions**
- **In events with arguments**
- **Bounded int types for better checking**
- **Out events with optional bindings to functions**
verbatim
statement Counter {
  in start() {no binding} step(int[0..10] size) {no binding} => handle_someEvent;
  out someEvent(int[0..100] x, boolean b) => handle_someEvent;
  reset() => reset;
}

vars int[0..100] currentVal = 0
int[0..100] LIMIT = 10
states (initial = initialState)
  state initialState {
    on start [ ] => countState { send someEvent(100, true) }
  }
  state countState {
    on step [currentVal + size > LIMIT] => initialState;
    on step [currentVal + size <= LIMIT] => countState;
    on start [ ] => initialState 
  }
} end statement
A number of default properties for reachability, nondeterminism, variable ranges. Additional properties can be described using an abstraction of LTL/CTL. Counter example if a property fails --- clicking on example highlights code in model.
Requirements

Tracability
requirements HighLevelRequirements
  show traces true

functional Main: Program has to run from the command line ...
  functional Arg2: Argument Count must be 2 ...
  functional FailOtherwise: Otherwise it should return -1 ...
functional Add: The program should return the sum of the two arguments ...
  functional AddFct: Adding should be a separate function for reuse ...

module ExampleCode from test.ts.requirements.code imports StrUtil {

  int8_t add(int8_t a, int8_t b) { return a + b; }

  int8_t main(string[ ] args, int8_t argc) {
    if ( argc == 2 ) {
      return add(str2int(args[0]), str2int(args[1]));
    } else {
      return -1;
    } if
  } main (function)
Simple way to specify
requirements (kind, ID, description)

Alternatively import them from external tool

Requirements traces can be attached to any program element expressed in any language --- no changes to host language necessary

Requirements kind and trace kind can be extended.
And code can also be edited without the traces, if developers prefer that.
Product Line Variability
feature model DeploymentConfiguration
root { 
  logging
  test
  valueTest [int8_t value]
}

configuration model Debug configures DeploymentConfiguration
root { 
  logging
  test
  valueTest [value = 42]
}

configuration model Production configures DeploymentConfiguration
root { 
  << ... >>
}
Code contains annotations with boolean expressions over the features in Feature Model.

Color depends on expression -- same expression, same color.

This page shows the product line mode --- all options in code.
Variability from FM: DeploymentConfiguration

```plaintext
module ApplicationModule from test.ex.cc.fm imports {

    message list messages {
        INFO beginningMain() active: entering main function
        INFO exitingMain() active: exitingMainFunction
    }

    exported test case testVar {
        report(0) messages.beginningMain() on/if;
        int8_t x = 42;
        report(1) messages.exitingMain() on/if;
        assert(2) x == 42;
        int8_t ww = value (variant Debug);
        assert(3) ww == 42;
        int8_t ww = 12 + value (variant Debug);
        assert(5) ww == 54;
    } testVar(test case)

    int32_t main(int32_t argc, string[ ] args) {
        return test testVar;
    } main (function)
}
```

```
feature model DeploymentConfiguration
root {  
    logging
    test
    valueTest [int8_t value] 
}
configuration model Debug configures DeploymentConfiguration
root {  
    logging
    test
    valueTest [value = 42] 
}
configuration model Production configures DeploymentConfiguration
root {  
    << ... >>
}
```

Code in the debug configuration --- „everything in“
Variability from FM: DeploymentConfiguration
Rendering Mode: variant rendering config: Production

module ApplicationModule from test.ex.cc.fm imports SensorModule {

exported test case testVar {
    int8_t x = getSensorValue(1);
    assert(2) x == 10;
    int8_t ww = 22;
    assert(4) ww == 22;
} testVar(test case)

int32_t main(int32_t argc, string[ ] args) {
    return test testVar;
} main (function)

Code in the production configuration --- „everything out”
Status and Availability
Developed in the

LWES
Language Workbenches
for Embedded Systems

- Project runs till June 2013
- itemis, fortiss, SICK, Lear

gefördert durch das BMBF
Förderkennzeichen 01S11014
Core is Open Source (EPL)

- Eclipse Public License
- Essentially no restrictions regarding commercial use
Some Extensions will be Open Sourced this year

- We have to finish/stabilize them before we make them available
- Statemachines & Components will certainly be part of the Open Source package
Custom Extensions and Professional Services by itemis

- Introducing the tool
- Language Definition and Extension
- We're looking for prototype customers!
support for graphical early 2013
- state machines and block diagrams
- integrated with text
Integration in early 2013

- Native integration with Eclipse UI
- EMF export already possible today
mbeddr C

An extensible version of the C programming language for Embedded Programming

http://mbeddr.com

C the Difference - C the Future