Innovative Embedded Software Reuse: from Copy & Paste to Micro Software Components.

Thomas Klöker, Dirk Friebel

SoftComponents
industrial quality software

Geesmanns Kotten 24 – 45663 Recklinghausen - Germany

{tk,df}@softcomponents.de http://www.softcomponents.de

Abstract

This paper presents an innovative, extremely resource-friendly component technology for embedded software. This software component technology maximizes the possibilities to reuse software and minimizes the costs leading to a reduced development time and development costs and better software quality. It allows the reuse even of small software parts (so called micro components) without consuming additional memory or run time. It provides generic mechanisms for software component configuration (no more unreadable nested #if / #ifdef's) in order to fit a reusable software component to the needs of a certain application without resorting to error-prone manual source code modifications (no more copy & paste). It allows static or dynamic interconnections of components and interfaces either during build-time or during run-time with the ability to insert arbitrary glue code. It supports distributed systems through generic marshalling mechanisms. It is realized by a tool working as a pre-compiler for a C dialect, basically minimally extended by adding the two component-oriented keywords `component` *and* `interface`*. This tool produces ANSI-C code that can be compiled for any platform for which a conforming C compiler is available – no operating system or framework libraries are required. Due to a transparent naming convention, the C code generated by the tool can be mixed with any hand-coded or otherwise auto-generated C code. The tool also provides generic support for generating software component documentation and for automating software component tests. The approach solves typical problems encountered when trying to develop reusable software in C and is easy to understand for skilled C/C++ - programmers.*

1. Introduction

Component oriented software development is one of the most promising approaches to control the always-increasing complexity of computer programs. The use of reusable, tested and therefore mature software components can significantly accelerate the development of a complex computer program and at the same time, the quality of the final product normally increases considerably because of the assured quality of the individual components.

The use of existing component techniques out of the PC or server software area, like COM or Corba, is prohibited for embedded computer systems with very limited resources based on e.g. 8- or 16-bit microcontrollers due to the amount of additional computer resources like memory and run time these techniques require. Additionally required resources consequently require a more powerful microcontroller, which will be more expensive and whose price then increases the costs for each end device or decreases the earnings per end device respectively. Since these microcontrollers are often applied in large volume products where the cost pressure from the customers and competitors is very high, this normally isn't acceptable.

In addition, a reusable software component must cover the largest possible functionality for all imaginable applications in

order to be flexibly useable without changes. In case such software components are used with existing component techniques for a specific application, a major part of the functionality isn't used in this particular case, but then unnecessarily wastes resources like memory and run time, which is unacceptable on computer systems with limited resources because of the reasons mentioned before.

To be able to remove such unnecessary functionality out of a component afterwards, unlike in the PC or server software area the software components shouldn't be defined on a binary code level but on a source code level, since all optimisation possibilities are still available here. Software components on a binary code level would also limit the reusability a lot anyway because of the large amount of binary incompatible microcontroller platforms.

2. Problem definition

If one tries to develop a reusable C-module or a C++-class for an embedded software application, one often faces a conflict of interest between a possibly widespread and flexible and a resource optimised implementation during the design phase – in this case one has to decide between one of the solutions that normally limits the reusability again. The following problem classes can be distinguished:

1. Multiple use of a code module
2. Constant or variable parameters
3. Flexible linking of modules

Below, these problem classes are clarified using the example of the design of a software driver module for a serial port:

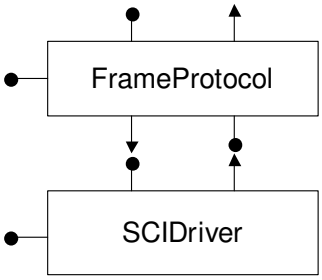
Depending on the amount of connection pins, individual members inside a microcontroller family are often configured with a different amount of peripheral devices. For instance a microcontroller of a specific family only has one serial port while another member is equipped with three, where these are typically identically accessed and only the registers and the interrupt vectors of the peripheral devices are located at different addresses respectively.

If a software driver module for the serial ports of this microcontroller family should be developed in a not specifically component oriented programming language like C or C++ now, the question arises if the driver functions should be provided with an index for the desired serial port or not. For microcontrollers with only one serial port, passing an index would be a waste of resources since no different ports can be accessed. Even for microcontrollers with more then one port, it should be considered if passing an index is useful or if it would be better if a copy of the driver module would be available for each serial port instead. The last option definitely makes sense when a program optimised for run time is required, sufficient memory is available and the meaning of the ports is fixed during build time and doesn't change during run time. Since the same program code is used more than once passing an additional index does save memory but will cost run time since on every access of the driver functions to the serial port registers, the corresponding indexing has to be resolved. Such different requirements can currently be met only by error-prone manual copying of source code (aka Copy & Paste) or by the help of rather intransparent and hardly maintainable C-preprocessor constructs using multiply included header files with renaming macros, which are also error-prone.

Other problems in a not particularly component oriented programming language like C or C++ may arise in the initialisation of the software driver for a serial port when transfer parameters like baud rate, number of data bits, number of stop bits, parity und/or the like are considered. In a lot of applications these parameters are defined during the build time, since the respective systems have fixed connections to other systems with fixed transfer parameters. Only on few systems, these parameters can change during run time. The transfer parameters can therefore be either variable parameters of the initialisation function of the driver module or can be constant parameters of the driver module defined by macros. For applications where the transfer parameters are fixed at build time, variable parameters would be a true waste of memory and run time, since the required calculations can already be carried out during compilation. Nevertheless, committing to fixed parameters considerably limits the flexibility and reusability of

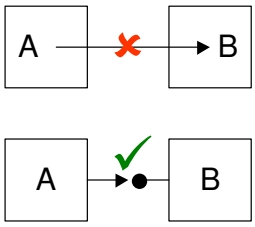
the driver module. Until now, such different requirements could only be realised by an error-prone manual modification of source code or by means of an also, because of its poor maintainability, error-prone conditional compilation using the C-preprocessor constructs `#if` and `#ifdef`.

Further problems in a not specifically component oriented programming language like C or C++ arise when linking such a software driver module for serial ports for instance with an overlying software module for the implementation of a communication protocol through given ports.



A software driver module for a serial port typically has, when realised via interrupts, exported functions that can be used by other modules and that can be called for actions like starting the transmission a character via the port and imported functions that are used by the software driver interface itself and that are called for events like indication the receipt of a character via the port. After an according linking, the communication protocol module, logically having a higher level then the software driver module, now calls the functions that are imported by this protocol and exported by the port driver and the port driver on its side would call the functions imported by this module, which are exported by the protocol module.

In case one wants to develop reusable software modules, the source code of these modules may not have direct references to functions from another module since in another application for instance a completly different communication protocol could be on top of the software driver or the communication protocol could use a completely different interface. Such a transparent component linking is very hard to achieve in C, especially in case of reciprocal references, since problems with circular included header files arise very quickly.



Although these problems can basically be solved by an extensive use of the C-preprocessor, one will end up with a hardly maintainable collection of deep nested code fragments that are conditionally compiled using the pre-processor constructs #if and #ifdef, which are furthermore littered with deep nested macro definitions using the pre-processor construct #define.

Since such a code becomes unmaintainable very quickly, one limits itself during the real design of C-modules or C++-classes to those special cases that appear to be important at that particular moment. As a consequence, this of course limits the possibility of reuse a lot (for an adequate example see [1]).

3. Solution proposal

We at SoftComponents have developed a solution for these problems based on our own, unsatisfying efforts spent in the past few years in using the standard C preprocessor in a clever way to program component oriented but nevertheless maintainable.

It consists of a dialect of the standard programming language C, which has been cautiously extended by a few component elements that are mapped to ANSI-C by a dedicated pre-compiler and which we call Component C therefore. Basically, the two keywords `component` and `interface` have been added and the syntax was extended in analogy to the other C language syntax elements. Since there are alternative language elements available now, an obligatory integration of the C-preprocessor can be abandoned and a clear language definition can be achieved.

An outstanding achievement of this solution is the **generic configurability** of every software component created with it without the need for the component developer to write any configuration dependend

code. Because of this, every component can be adapted to the respective application without additional efforts and the use of resources is minimised. Therefore, also the smallest software elements can be reused as so-called **MicroComponents** without any overhead.

From a syntactic point of view, a component definition in Component C equates a type declaration, from a semantic point of view it is more equivalent to a template in C++ (but significantly differs), since *no* binary code is created at this stage. However, a component configuration equates an initialised variable definition in C from a syntactic point of view, but from a semantic point of view it rather equals the use of a template in C++; since at this stage binary code will be generated.

Component definition

Components may contain types, constants, variables, functions, interfaces or subcomponents as exported or imported elements. Types, constants, variables and functions are defined as in C, optional attributes as similar to MIDL. Interfaces are logical groups of elements and can contain types, constants, variables, functions or sub-interfaces. Global variables and functions outside a component are, as in C, allowed.

All elements required by a component and provided by another component, i.e. which are imported, are labelled with the keyword `extern`. All elements that are only used inside the component and should not be imported or exported are labelled with the keyword `static`. All elements without such a label are exported. These definitions are equivalent to the usage of the keywords on module level in C. For a clean distinction of components, imported variables can only be written when they are additionally labelled with the keyword `volatile` - otherwise they will be read-only.

Below an example of a definition of two software components in Component C is shown:

```
// Defines Interface ia
interface ia
{
  int fa (int x);
  int fb (int z);
};
```

```
// Defines Component a
component a
{
  char v = 1;
  static int sv = 2;
  volatile float vv = 3;

  const long c = 42;
  static const short sc = 43;
  volatile const double vc = 44;

  int f (int x, int y)
  {
    return (x*y);
  }

  int fa (int x)
  {
    return (x*v/c);
  };
  int fb (int z)
  {
    return (z*c);
  }

  interface ia ia1;
};

// Defines Component b
component b
{
  extern int ev;
  extern const long ec;

  extern int ef (int x, int y);

  extern interface ia eia1;

  interface ib
  {
    int fc (int y);
  }
  ib1 =
  {
    .fc = {
      return (eia1.fa (y)+ev); }
  };
};
```

Component usage

An implemented component can be used once or multiple times in an application in the same, but also in a different configuration. If the component `a` defined above should be used once, this is possible in the following way:

```
// One component instance;
// Component functions do not
// contain a reference. A single
// variable with the name a_vk1
// is defined.
```

```
component a a_vk1;

void main (void)
{
    printf („%d“,a_vk1.ia1.
          fa (42));
}
```

If the above-mentioned component `a` should be used twice with an identical default configuration but, for instance, with different names, this is possible in the following way:

```
component a a_vk1, a_vk2;
```

or

```
component a a_vk1;
component a a_vk2;
```

This usage actually creates two copies of the source code and can be applied in the case that a component is used in different configurations or when sufficient memory is available and the access time has to be minimised by constant addresses.

If a component should be used N times with an identical default configuration, this is possible in the following way:

```
// Three component instances;
// Component functions contain
// an impicit index as a
// reference, a variable array
// with the name a a_vk3
// with three elements is
// defined.

component a a_vk3[3];

void main (void)
{
    printf („%d“,a_vk3[2].ia1.
          fa (42));
}
```

If a component should be used an arbitrary number of times possibly changing during runtime, this is possible in the following way.

```
// Arbitrary number of
// component instances;
// Component functions contain
// an explicit pointer as a
// reference, a type with
// the name a_vk4 is defined.

component a *a_vk4;

void main (void)
{
    a_vk4 a1;
    a_vk4 *pa2;

    pa2 = new a_vk4;

    printf („%d,%d“, a1.ia1.
          fa (42),pa2->ia1.
          fa (43));
}
```

We call a usage of a defined component in one of these ways a configuration of the component. The last two configurations create only one copy of the source code each, where in the first case a component index and in the second case a component pointer is passed implicitly to the component function. In the latter case, a component definition and configuration basically turns into a class definition from an object oriented programming point of view.

Configurating exported elements

In case the variable isn't required for the specific application, a variable exported by a component can be labelled unused by setting the respective variable to zero.

```
component a a_ev1 =
{
  .v = 0  // Exported
          // variable v unused
};
```

Because of this, the created source code can be optimised, especially when a variable in a component is only written, not read.

When an application always uses the same value for a function parameter, a function exported by a component can be configured by specifying a constant parameter instead of a variable one.

```
component a a_kf1 =
{
  .f = { .x = 42 } // Parameter
    // x of the function f fixed
};
```

By doing so, the created source code can be optimised by immediately using a constant instead of a variable when the function isn't called with other values inside the component definition itself.

Also, a function can be labelled unused by assigning a zero.

```
component a a_kf2 =
{
  .f = 0        // Function f
                // unused
```

```
};
```

By doing so, the source code can be dropped when the function isn't called inside the component definition itself.

An interface exported by a component can be configured appropriately for an application by labelling non-used functions and variables as well as constant function parameters.

```
component a a_ks1 =
{
  .ia = {
     .fa = { .x = 42 },
     .fb = 0 }
};
```

By doing so, the source code can be optimised.

Also, an interface can be labelled unused by assigning a zero.

```
component a a_ks2 =
{
  .ia = 0       // Interface
                // ia unused
};
```

By doing so, the source code and memory allocation can be omitted when the functions or variables respectively, aren't used in the component definition itself.

Linking imported elements

Imported elements can generally either be linked statically during build time, in order to minimise the resource footprint but can also be linked dynamically during runtime in order to maximise the flexibility.

In a component configuration, a constant imported by a component can be statically linked to a type compatible constant expression.

```
component b b_vk1 =
{
  .ec = 42
};
```

For a component used N times, two cases can be distinguished. In the first case a constant has the same value for all instances

```
component b b_vk2[3] =
{
  .ec = 42  // Same value for
            // all instances
};
```

In the second case it has different values for the different instances

```
const long c_vk[3] =
    { 1, 2, 3 };

component b b_vk3[3] =
{
  .ec = c_vk  // Different
         // values for different
         // instances
};

component b b_vk4[3] =
{
 .ec = {42, 43, 44} // Different
         // values for different
      // instances
};
```

Both an explicit as well as an implicit assignment of an array is possible here. In this way, components used N times can be parameterised equally but also unequally.

In a component configuration, a variable imported by a component can be statically linked to a type compatible variable or constant expression.

```
component b b_vv1 =
{
  .ev = 5
};

int v_vv1;

component b b_vv2 =
{
  .ev = v_vv1+3;
};

component a a_vv1;

component b b_vv3 =
{
  .ev = a_vv1.v;
};
```

For the first case attention has to be paid to the fact that the expression doesn't contain any unwanted side-effects since the time point on which the evaluation takes place depends on the component implementation and because of this, can't be predicted when required. In the second case, using a constant instead of a variable can optimise the created source code.

As for constants, for a component configuration used N times, two cases can be distinguished. In the first case, all in-

stances use the same expression where in the second case different expressions are used. By this, the pre-compiler tries to optimise the evaluation and, when possible, uses fields of values or pointers when only constant expressions or pointer expressions are used respectively.

If no static linking occurs in the component configuration, a function pointer instead of a variable is created, which has to be dynamically initialised by the application when the component is used.

```
int v_vv4 (void)
{
  return 42;
}

component b b_vv4;

void main (void)
{
  b_vv4.ev = v_vv4;
}
```

In this case, the variable type of the configured component becomes a pointer to a function that returns a value of the original type and in the implementation of the component a function call is implicitly inserted when the variables are referenced.

In a component configuration, a function imported by a component can be statically linked with a code block or with a type compatible function pointer expression.

```
component b b_vf1 =
{
  .ef = { return (x/2); }
};

int f_vf1 (int x, int y)
{
  return (x*y);
}

component b b_vf2 =
{
  .ef = f_vf1
};

component a a_vf1;

component b b_vf3 =
{
  .ef = a_vf1.f
};
```

When linking to a code block, the pre-compiler will always realise it as an inline function. It is always recommended to use an intermediary code block when individual arguments of a function are not evaluated, since the source code can then be optimised in such a way that only the required arguments have to be calculated.

For a component configuration used N times, two cases can be distinguished again. In the first case all instances call the same function, in the second case different functions are called.

In a component configuration, an interface imported by a component can be statically linked with a type compatible listing of individual elements or with a type compatible interface.

```
component b b_vs1 =
{
  .eia = { .fa={return (x/2);},
           .fb={return (0);} };
    // Interface connected
    // to Glue Logic
};

component a a_vs1;

component b b_vs2 =
{
  .eia = a_vs1.ia1;
    // Interfaces connected 1:1
};
```

For a component configuration used N times, two cases can be distinguished again. In the first case all instances use the same interface and a component index is passed through.

```
component a a_vs2[3];

component b b_vs3[3] =
{
  .eia = a_vs2.ia;
    // Index passes through 1:1
};
```

And in the second case different interfaces are used and no component index is passed through anymore.

```
component a a_vs3[3];

component b b_vs4[3] =
{
  .eia = { a_vs3[2].ia,
     a_vs3[1].ia, a_vs3[0].ia };
    // Explicit Index mapping
};
```

For a component used an arbitrary number of times, in the static component configuration only the first case can be used, which then looks as follows.

```
interface ia *ia_vs =
{
   .fa = { return (x+2); },
   .fb = { return (42); }
}

component b *b_vs5 =
{
  .eia = ia_vs
};
```

Interface inheritance

Component C supports both simple inheritance of interfaces, like in COM, as well as multiple inheritance, but not the inheritance of function code like in C++, since this disagrees with the basic thoughts behind components, which include a much stricter focus on the encapsulation of information.

To avoid the well known Fragile Base Class Problem, also in so-called object oriented programming one works more and more with abstract base classes – but abstract base classes are nothing different from interfaces!

External references

Like in Java, qualified names are used to refer to components and interfaces defined in separate files.

```
component driver.bus.can.toucan mycan;
//                                    ->
$(COMPROOT)\driver\bus\toucan.cc
```

Here integral parts of names are mapped on a directory structure until the name decomposition has reached file level. In the case that further integral name parts are available, it is then referred to an element inside the file. If the name refers to a complete file, the complete contents of the file will then be interpreted as a component or an interface respectively, without having the entire contents of the file to be parenthesed with the according keywords.

This concept leads to a very advantageous recursive structure; every summarised sentence of configured and linked components in the file can on his part be interpreted as a component again.

Because of this, a conventional C-module, in which no pre-processor statements like #include are present, can directly be interpreted and used as a component without any change.

Attribute and Marshalling

In the description of interface elements, additional attributes like in MIDL can be declared. Because of this, it becomes possible to realize a generic marshalling of function calls and by doing so, systems with different priority levels or distributed systems can be especially supported.

Further Features

The availability of a pre-compiler allows the implementation of some additional features that are very useful for the creation of Embedded Software components. For example arbitrary scaled fixed-point types are supported in the following advantageous form:

```
uint16 Vspd/10; //Vehicle speed
                //[3.1 km/h]
typedef sint16 TORQUE*0.05;
                //[3.2 Nm]
TORQUE T_trg; // Target torque
TORQUE T_act; // Actual torque
```

On top of this, the definition of individual bits, supported by some microcontrollers via bit addressed memory, is also allowed outside structures.

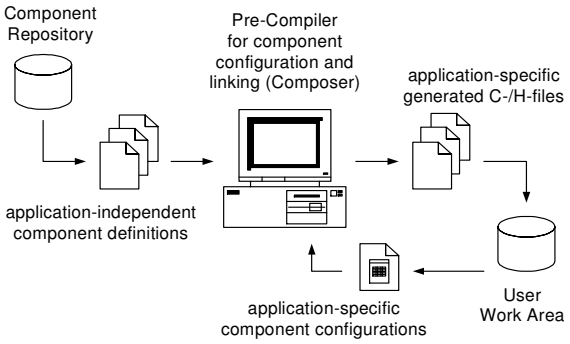
```
int B_ll : 1;      // idling
```

Finally, type templates can also be used inside the component definition, which then first have to be defined by a component configuration.

```
any a;         // Arbitrary type
any int i;     // An integer type
```

4. Pre-compiler operation method

The pre-compiler named Composer generates C code and header files from application-independent component definitions directed by application-dependent component configurations as shown below.



Here the configured number of instances, the component multiplicity, is mapped in such a way that a structure containing the variables of the component is defined and for components containing only one instance, a corresponding variable of this type is defined, to whose elements the functions of the component have direct access to. However, for components with a fixed number of N instances, a variable array of N structures is built and an additional first parameter of integer type is added to the component functions, through which an index between 0 and N-1 has to be passed for the respective component instance. For components with an arbitrary number of instances, a type instead of a variable is defined and the component functions are amended by an additional first parameter of pointer type, through which a pointer to a data structure of the defined type has to be passed, for which the application has to allocate memory either statically or dynamically.

When not used in the component itself, the pre-compiler removes exported elements of a component marked unused in the component configuration like variables, functions and interfaces when they aren't used inside the component itself. Directed by a command line switch the pre-compiler is also able to detect unused exported elements by itself, if the given component configuration is concluded in itself and no exported elements are used in another hand-coded or auto-generated C module.

For exported functions, having individual or all parameters replaced by constant values in the component configuration, the corresponding parameters are moved from the function header to the beginning of the function body before the function code and are initialised with the value defined in the component configuration.

The pre-compiler uses the value of imported constants for the optimisation of the source code when they are not marked with the keyword volatile as being adjustable during the runtime of the program by means of e.g. a configuration program.

When a Glue-logic expression is used in the source code, the pre-compiler will replace the access to imported variables by this expression, when a static link to another variable is used it is replaced by an access to this link and when a dynamic link is used, it is replaced by function linked by a pointer. If a Glue-Logic-expression has been specified, it will be used for the optimisation of the source code.

Basically, all expressions that are constant during build time are evaluated by the pre-compiler and are optimised accordingly by e.g. removing dead code that is not executed anymore due to for instance a specific constant configuration selection, so no warning messages will arise from the C compiler later on. The resulting C-source code is then optimised for both the memory allocation as well as the run time, so a completely manually carried out programming will not require remarkably less resources.

Similar to JavaDoc or DoxyGen, the pre-compiler provides support on the documentation of components on top of that. It is also capable of creating XML-descriptions of created software components. Furthermore, generic support for automatable component tests as well as debugging and profiling is available.

Via simple naming conventions, the component elements can also be accessed from standard C-code, so a combination with common development methods is possible without any problems.

5. Discussion

When compared to other methods, the approach introduced here shows some advantages for component oriented programming for embedded systems, but be-

cause it is a novelty, it can't be seen as a standard yet.

Our aim was to define a **minimal extension** of the **standard C** language in order to be able to support component oriented programming of embedded software at system level and therefore allow better reuse of embedded system software. The main unique advantage of our approach is the **generic configurability** of all software components defined with it and therefore the possibility to adjust every software component to the need of an application in order to **minimise** the **resource footprint** without error-prone manual modifications. These achievements make Component C an ideal programming language for low-level embedded software and embedded hardware with very limited resources.

Compared to the attempts to realise reusable, adjustable templates with C++, our approach offers the advantage that really self-containing components can be defined, which can be individually tested and checked, whereas when C++ templates are used, lots of checks can first be made by the compiler when the templates are instanced, i.e. when the user uses them and may result in error messages that refer to the code of the template, so no finalised components can be created when using C++ templates. Furthermore, C++ templates are often difficult to understand for average C/C++ programmers and notoriously hard to debug.

Already existing component models for Embedded Software (e.g. ECOS Component Model [2], Koala [3], Knit [4], TinyOs/nesC [5], Real-time Corba, Minimum Corba) are often specifically developed for specific projects, for specific industry sectors, for specific platforms or for specific compilers and the components created with these are not truly generally reusable again, for instance on 8 bit platforms that are very limited in resources.

Only the solution developed by TI, Real-Time Software Components (RTSC [6]) has a standard comparable to our solution in supporting as many platforms for which an ANSI-C-Compiler is available as possible. In contrast to the RTCS solution, in our approach not only the interface descriptions are evaluated by our tool but also the component implementations. Because of this, the Composer can optimise the source code for every component and every application without modifying the kernel of an already tested component, so the validation of the component remains. This, the XDCtools cannot offer. Our approach also seamlessly integrates in the programming language C, so experienced C-developers are quickly able to implement, configure and link components whereas for RTSC with XDCspec and XDCscript at the same time two new languages where defined.

6. Literature and Links

[1]Softwareentwicklung für eingebettete Systeme mit strukturierten Komponenten Teil 2:Komponentenorientierte Modellierung und Realisierung
Stephan Eberle und Peter Göhner, Universität Stuttgart, atp 45 (2003) Heft 2, p. 61ff, http://www.ias.unistuttgart.de/foschung/pub/atp_strukt_Komp_teil2_04_2004_eb_goe.pdf

[2] http://ecos.sourceware.org/

[3] The Koala Component Model for Consumer Electronics Software,
Rob van Ommering, Frank van der Linden, Jeff Kramer, Jeff Magee,
IEEE Computer, March 2000, p78-85

[4] Knit: Component Composition for System Software, Alastair Reid et al., Proceedings of the Fouth Symposium on Operating Systems Design and Implementation (OSDI 2000), p. 347-360

[5] http://www.tinyos.net/

[6] http://www.eclipse.org/dsdp/rtsc/