ADOK: a Minimal Object Oriented Real-Time Operating System in C++

Salvatore Benedetto
Scuola Superiore Sant’Anna
salvatore.benedetto@gmail.com

Giuseppe Lipari
Scuola Superiore Sant’Anna and LSV - ENS Cachan
giuseppe.lipari@lsv.ens-cachan.fr

ABSTRACT
Most embedded software is currently developed using the C programming language, even though its low level of abstraction requires a lot of effort to the programmer. The C++ language is a better choice because: it raises the level of abstraction; it is strongly typed, so it prevents many common programming mistakes; it can be made as efficient as C through fine-grained customisation of memory mechanisms; it can be easily adapted to domain-specific needs. In addition, recent compilers have grown in maturity and performance, and the new standard considerably improves the language by introducing new concepts and an easier syntax.

In this paper we present ADOK, a minimal Real-Time Operating System entirely written in C++ with the exception of a few lines of assembler code. It directly offers a C++ interface to the developer, and it provides a flexible scheduling framework which allows the developer to customise the scheduling to its needs. In particular, we implement a two-level scheduler based on Earliest Deadline First, the Stack Resource Policy protocol for sharing resources and support for mode changes. We demonstrate through examples and a small case-study that ADOK can substantially improve productivity without sacrificing on performance.

Categories and Subject Descriptors
H.4 [Information Systems Applications]: Miscellaneous;
D.2.8 [Software Engineering]: Metrics—complexity measures, performance measures

General Terms
Systems

1. INTRODUCTION
Software for embedded systems has traditionally been developed in C. According to a recent study [15], in 2013 60% of embedded software was programmed in C, and only 20% in C++. There are two main reasons for this dominance: the first one is that the C language is very close to machine code, thus the programmer can easily access hardware registers. The second reason is efficiency: being so close to the machine, C compilers usually generated very fast an memory efficient code, and this is very important in resource constrained embedded systems.

However, C is a low-level language and it may not be always adequate for developing complex and large applications. In particular, C lacks many abstractions and features that make the life of the programmer easier. We would like to mention, among the others, the absence of namespaces; the weak type semantic; the absence of object oriented concepts; the lack of a strongly typed template mechanism; the abuse of pointers for dynamic programming techniques; etc.

These limitations are serious especially in safety-critical embedded systems, where it is necessary to verify and certify the correctness of the code. For this reason, many companies enforce restrictions on the use of the C language. For example, MISRA C [12] is a standard for developing software in C that limits the use of constructs that could lead to undefined behaviour, and it is widely adopted by the aerospace and the automotive industries.

To support object oriented programming, many years ago Bjarne Stroustrup proposed the C++ language which was initially thought as an extension of the C language. However, C++ actually brings in many more features than just objects. In facts, it is considered as a very flexible language in which many modern programming techniques can be easily expressed, from template meta-programming [7, 1], to functional programming [11]. The new ISO/IEC standard [8] extends the language along several direction, improving the usability and adding many more features.

Given the premises, one would expect C++ to be the obvious preferred choice of embedded systems developers. However, C++ has often been dismissed because of its complexity and steep learning curve. Another reason is a prejudice about its efficiency: many features of the language are regarded as sources of inefficiency, like the use of dynamic binding and polymorphism.

In our opinion, such prejudice has little practical basis. Modern compilers are now very efficient, and the introduction of the new C++11 standard has made programming in C++ simpler and more effective. Also, we believe that, by applying specific idioms and restricting the most unpro-
dictable features of C++, it is possible to generate very efficient and predictable code from C++ programs. It is now time to give C++ another chance.

Contributions of this paper. In this paper we present ADOK, a minimal real-time operating systems for embedded software development. ADOK is written almost entirely in C++ and directly offers a C++ interface to the developer. The RTOS has been designed for supporting minimal embedded programs, of the same class as the ones supported by the OSEK standard [13] for automotive systems. In particular, we will make the assumption that all tasks are statically created at boot time, and no task can be dynamically created or killed during the system lifetime. ADOK provides a two-level scheduler based on Earliest Deadline First [9] for real-time tasks and Round-Robin for non-real-time tasks; the Stack Resource Policy [3] protocol for sharing resources; support for mode changes [14].

Our goal is to demonstrate that it is possible to use a subset of the C++ language and obtain very efficient code (comparable to the one produced by C) in a safer and simpler way.

2. RELATED WORK

There are many RTOSs available on the market and as open-source academic projects, too many to be cited here. The great majority of these are kernels implemented in C and they provide a C-like API. Among the minimal RTOS for the automotive domain and wireless sensor networks, we would like to mention ERIKA Enterprise [6] and Contiki [4], as they are closer to the architecture and objectives of ADOK.

Not many C++-based operating systems have been proposed until now. The Embedded Parallel Operating System (EPOS) [5] is the one that most resembles our approach: all kernel structures are implemented as template classes that can be customised in a variety of ways, for example for selecting the scheduling algorithm or the task model. The main difference is that EPOS was conceived as a medium sized RTOS and supports more general features as dynamic creation of objects, whereas our ADOK kernel is expressly dedicated to small embedded systems with minimal memory requirements, and hence it does not support dynamic task creation.

3. C++ FOR EMBEDDED SYSTEMS

In this section we discuss the features of the C++ language that are relevant to embedded system development. Polymorphism. In embedded systems, polymorphism should be avoided because it is less predictable (difficult to analyse the worst-case response time of polymorphic code) and it generally requires additional memory. Also, the lack of polymorphism is not seen as a disadvantage, because in a typical safety-critical embedded systems all objects must be known at design time.

Consider polymorphism as implemented in many object oriented languages. In Java, for example, every class method is polymorphic, and reflection is enabled for all classes. As a consequence, the overhead of these features is always present even if no class in our program needs polymorphic behaviour. This simplifies the object model at the expenses of increased overhead.

In C++ polymorphism can be completely eliminated. In fact, a class method is polymorphic only if it is declared as virtual. If all methods of a class and of its base classes are non-virtual, the C++ compiler will not produce any virtual table and will not generate dynamic binding. The memory layout of a non-virtual class will resemble that of simple C structure, and Run-Time Type Identification (RTTI) will not be possible for that class.

Of course, even if our ADOK RTOS does not use polymorphism (see Section 4), the developer may decide to use polymorphism in its program. In such a case, she will indeed introduce some possible extra overhead in the program, but only limited to the polymorphic class hierarchy, without compromising the predictability of the rest of the system.

In some cases, it is possible to use the Curiously Recurring Template Pattern [2] to mimic polymorphism at the cost of one extra function call (that can be optimised by function in-lining).

Exceptions. Another feature that should be eliminated is the support for exceptions, again because it introduces unpredictability and overhead. In C++ it is possible to completely disable the exception mechanism.

Type Check. Due to the lack of inheritance and templates, generic libraries written in C often require function parameters to be pointers to void. Consider, as an example, function qsort() available in the stdlib to sort an array of elements. Since we do not know what type of elements are contained in the array, qsort() can only accept a pointer to void and a function for comparing elements that takes two pointers to void. This function is cumbersome and inefficient (because it requires many unnecessary indirections), and if not used properly can cause many strange run-time errors leading to memory corruptions.

In contrast, by using C++ strong typing together with templates, it is possible to perform many static checks at compile time. Static type checking may reveal common programming mistakes at compile time. For example, the std::sort() function provided by the C++ standard library is templatised with respect to the type contained in the array, so it is type safe. Also, if applied to an array of incomparable elements, the compiler will raise an error. This approach is also efficient since the template mechanism will generate only the code that is needed to implement the function. Indeed, idiomatic use of C++ restricts the use of pointers to a few cases.

Memory allocation. In C++, it is possible to overload the the new/delete operators for customising dynamic memory. This technique pays off in embedded systems where the standard memory allocation may be very inefficient; for example, it would be possible to write custom memory allocation strategy for small classes; or to implement memory allocation strategies expressly designed for embedded systems [10]. While similar strategies can also be used in C, in C++ the mechanisms of operator overloading permits to easily change the policy without changing the rest of the program. Policy-based design [2] has been advocated as one of the most effective programming techniques in C++.

In the following we show how we applied some of these techniques to the design of a C++ Real-Time Operating System.

4. ADOK

ADOK is the name of an innovative object-oriented RTOS we developed at Scuola Superiore Sant'Anna of Pisa as part
void sensorReader()
{
    int local_var = 0;
    sensor<BMA180>::read(samples);
    ...
}

TASK_PERIODMS(sensorReader, 20);

Figure 1: An example of task in ADOK.

of one of the authors’ Master Thesis. The name is not an
acronym, but its pronunciation is similar to the Latin words
ad hoc that, according to the Oxford Dictionary, can be
translated as “created or done for a particular purpose as
necessary”. In fact, one of our objectives is to let the
programmer customise the kernel to its own needs without in-
truding extra overhead, as it were created in an ad hoc
manner.

Other objectives of ADOK are:

- Automatise code generation and customisation; it must
  be possible, for the user and for the kernel developer,
to change the kernel policies without requiring any ma-
jor change neither in the user code, nor in the rest of
the kernel code;

- Minimise and simplify user interface; a simpler API
  has many advantages, among which less possibility for
programming errors, increased code readability, porta-
bility and maintainability.

- To introduce the embedded system programmer to the
  C++ language in a gentle way.

The system is entirely written in C++03 (with the excep-
tion of a few parts which use some limited features of the new
C++11 standard), and using template meta-programming
for customising code.

Figure 1 shows an example of all that is needed to create
a periodic task to be executed every 20 milliseconds. The
task uses a global array to store samples which are read
every instance using the sensor<T>::read() function (line
5). The task is created at system start-up using the macro
at line 8, which requires the body of the task and period
in milliseconds. The task has a run-to-completion semantic,
in the sense that at each instance the task body function is
called and every local variable is recreated on the stack.

4.1 Architecture

4.1.1 Tasks

ADOK has been designed from the scratch as a modular
RTOS which can easily support different types of tasks and
schedulers. Different schedulers need different task param-
eters, so it is not efficient to prepare a task structure that
will encompass all possible conceivable task parameters.

The solution adopted for ADOK consists in providing a
base structure, called TaskBase, which contains all data com-
mon to every type of task, and a separate structure that
contains specific data for each specific task type. This re-
sembles the inheritance mechanism provided by any object-
oriented programming language, where all the common data
and common functionality are grouped in a base class, while
the rest of the data is grouped in different classes, one for
each sub-type.

Despite of that, the inheritance mechanism provided by
C++ has not been used because the C++ standard does
not define how the data within an hierarchy of classes is
lay out. Therefore, since the scheduler relies on the fact
that the base data is followed by the specific task type data
for performance reasons, and in order not to rely on the
compiler implementation, we decided to use an alternative
mechanism based on template meta-programming.

The mechanism is shown in Figure 2. First TaskBase de-
defines the structure containing the common data to all tasks
regardless of the type. Then, the type-specific data is de-
defined. In the figure we show the content of the TaskPeriodic
structure which contains the task period and offset, and the
internal variables used by the scheduler. The TaskBuilder
class (shown in Figure 2) puts everything together. This al-

Figure 2: Template-based structure to be specialised
for each type of task.
and instead of blocking on a periodic timer, they block on a non periodic event.

In ADOK, the initialisation of task-specific variables is made in the Application::onInit() function, whereas the main body of the task is implemented using template programming. We generalised the wait function to work both for periodic and sporadic tasks. A generic template-based wait routine is provided in Figure 2 (see template function taskEndCycle): depending on the type of task, we perform a wait on the timer, or a wait on a specific event. The code that implements the task routine is in the TaskBuilder class. In this way, the static method run is the entry point of every task regardless of their type.

### 4.1.2 Scheduler

ADOK provides a two level scheduler: real-time tasks are scheduled using the Earliest Deadline First (EDF) scheduler; when no real-time task is active, non-real-time tasks are scheduled according to a Round-Robin policy. However, the scheduling architecture shown in Figure 3 has been made flexible, so that it is easy to customise the scheduler to the programmer needs.

First of all, we designed a Scheduler class which provides the common interface for the scheduling subsystem. The rest of the system will rely on this interface for scheduling tasks. The Scheduler class derives from its template parameter, which is the scheduler implementation class. This is an instance of the CRTP mentioned in Section 3: we simulate virtual function without actually relying on polymorphism.

Currently, we provide 3 different scheduler implementations: SchedImpEdf, SchedImpRR and SchedImpH2, which implement, respectively, an EDF scheduler, a Round Robin scheduler and a hierarchical composition of two schedulers. Finally, we initialise the system scheduler to be a hierarchical composition of the EDF scheduler and of the Round Robin scheduler. EDF tasks will have higher priority than Round Robin tasks: the hierarchical scheduler checks if there is something ready to be executed in the EDF scheduler, then it dispatches it; otherwise, it dispatches a RR task. In any case, the basic Scheduler class has an idle task that is run when no other task is ready.

The EDF scheduler is also a template class that can be customised with respect to the implementation of the ready queue. Currently, we implemented the ready queue as a simple linear list (ItemList), because when the number of tasks is low this implementation is the most efficient. However, linear list have complexity linear with the number of tasks: therefore, for systems with large number of tasks, the ItemList class can be substituted by a more efficient balanced binary tree which provides $O(\log n)$ complexity.

```cpp
template<typename S>
class Scheduler : public S { ... }

template<typename S1, typename S2>
class SchedImplH2 { S1 *high; S2 *low; ... }
typedef Scheduler<SchedImpH2<SchedImpEdf<ItemList>, SchedImpRR>> SysSched;
```
this technique reduces the amount of programming errors in taking locks, and it is now implemented in many C++ libraries. ADOK provide a MutexWithCeiling class that uses the Stack Resource Protocol [3]. The user has to specify the ceiling when the object is created at initialisation time, and then it uses it with a Lock object which implements the RAIi technique.

Other features of ADOK include a classical Semaphore class, the MessageQueue class for communicating between tasks, and functions for interrupt management and interaction with I/O devices.

4.1.4 Mode change

Most embedded systems are modelled by finite state machines, that is, the behaviour of the system in response to a certain event may be different based on the current state of the system. For example, in a control system it is possible to change the control algorithm based on the current operating mode. One possibility is to have different tasks implement the different control algorithms; and activate only the correct tasks when changing operating mode. However, changing mode is not trivial, as we have to ensure that all tasks will meet the deadlines under all conditions, even during the mode change. Many algorithms have been proposed in the literature for performing mode changes [14]. In ADOK we implemented the Idle Time Protocol.

To support mode changes, ADOK provides the possibility to easily define a finite state machine and thus allows the developer, through a very simple interface, to define the execution of a task only when the system is in a given set of states. The code is shown in Figure 4.

In order to define the application states, the developer needs to define the enumeration declared in the Application namespace called eState_t. Each bit can represent a different state, and each task can be run in one or more states. It can even run in every state if the value EXECUTE_ALWAYS is used, that is 0. This has been achieved with two more template-based functions in the TaskBuilder class. In the task body (method run of the TaskBuilder class), the task checks the operating mode by calling the checkState that are optimised by the compiler when the default behaviour of EXECUTE_ALWAYS is specified.

Another important function is the onChangeState method which is called by ADOK on every transition, with a parameter that defines the new state of the system. One possibility is to have different tasks implement the different control algorithms; and activate only the correct tasks when changing operating mode. However, changing mode is not trivial, as we have to ensure that all tasks will meet the deadlines under all conditions, even during the mode change. Many algorithms have been proposed in the literature for performing mode changes [14]. In ADOK we implemented the Idle Time Protocol.

5. EXAMPLE OF APPLICATION

In this section we will show a demo application composed of two tasks acquiring samples from a 3-axis gyroscope sensor. The first task acquires data during the Init state for calculating the zero-offset error, the second task acquires data during the operational state and will correct the samples with the zero-offset value calculated during the Init state. The code is shown in Figure 5.

We define the data we need within a namespace called data. The task gyroscopeInit() will be executed every 200 milliseconds only during the Init state. Task gyroscope is then executed during the Operative mode.

```cpp
namespace Application {
    enum class eState_t : uint8_t {
        Init = 1, Operative;
    }
    void onInit() {
        sensor<L3G4200D>::init();
        System::setState(Application::eState_t::Init);
    }
    void onChangeState(eState_t newState) {
        using namespace data;
        if (newState & eState_t::Operative) {
            for (uint8_t i = 0; i < 3; ++i)
                zeroOffset[i] = accumulator[i] / zeroSamplesAcquired;
        } else for (auto x : accumulator) x = 0;
    }
}
```

Figure 5: Demo: mode changes

As it is possible to see, the code is very readable and compact. The use of templates and macros makes it easy to change the structure of the application, for example by adding a new state, or by changing the tasks to be executed in each operating mode.

We measured the footprint of this application when compiled along with the kernel in number of bytes, and the results are shown in Table 1. Unfortunately, we cannot show here a direct comparison with a similar application implemented with another RTOS, however these numbers are in line with similar numbers obtained by OSEK compliant RTOS like ERIKA².

6. PERFORMANCE

We implemented ADOK for an embedded board with a STM32H103FB micro-controller from the STM32 family of ST-Microelectronics. It is a System-on-Chip with a 32bit ARM Cortex M3 core. It has 128KB of embedded non-volatile flash memory where the ADOK image can be stored. It also contains various controllers on board which facilitate the interaction with external hardware. The microcontroller also contains 20KB of SRAM memory for running

²Benchmarks for the ERIKA RTOS are available at http://erika.tuxfamily.org/wiki/.
the application, and a JTAG controller which made the debugging process during the development very easy.

For testing the overhead of the context switch, we developed an application with a variable number of tasks (3, 5, 7 and 9 tasks) with different periods and the same offset. The tasks in the application toggle the same GPIO pin state causing a train of impulses to be observed with an oscilloscope. The amount of time for switching from one task to the next has been measured to be always between 10 and 13µsec. As an indirect comparison, ERIKA on a Cortex M4 takes approximately 20µsec for a context switch with two tasks\(^3\) As it is possible to see, the overhead is low. Moreover, we believe there is some more space for optimising the code and further reduce the overhead. We plan to conduct more experiments for measuring the overhead of other kernel primitives in different operating conditions and compare with other RTOS running on the same processor.

### 7. CONCLUSIONS AND FUTURE WORKS

In this paper we presented ADOK, a minimal RTOS developed entirely in C++. The goal of this work is to demonstrate that it is indeed possible to use C++ as a language for embedded system programming. The resulting application code is compact and efficient; at the same time, C++ is a much safer language than C because it provides strong type checking; and much powerful because it provides many high-level features.

In the future we plan to expand ADOK by providing the implementation of other scheduling algorithms (like fixed priority and table-based scheduling). Similarly with what we did with the mode-change component of ADOK, we also plan to provide libraries for supporting fault-tolerance, energy-aware scheduling and adaptive scheduling.

### 8. REFERENCES


---


<table>
<thead>
<tr>
<th></th>
<th>text</th>
<th>data</th>
<th>bss</th>
</tr>
</thead>
<tbody>
<tr>
<td>standard</td>
<td>21044</td>
<td>592</td>
<td>7924</td>
</tr>
<tr>
<td>-ffunction-sections</td>
<td>15340</td>
<td>592</td>
<td>7920</td>
</tr>
<tr>
<td>-fdata-sections</td>
<td>8872</td>
<td>572</td>
<td>7940</td>
</tr>
</tbody>
</table>

Table 1: Footprint of the demo application