OBJECT-ORIENTED EMBEDDED C

M. Neser* and G. van Schoor*

** School of Electrical, Electronic and Computer Engineering, North-West University, South Africa, Email: mneser@email.com / George.VanSchoor@nwu.ac.za*

Abstract: This paper presents an object-oriented implementation of ANSI-C for embedded systems. It offers practical guidelines for producing generic software libraries and portable applications. While various object-oriented implementations of C is available, the aim is to impress a culture of producing safe, robust embedded software which can easily be shared and integrated amongst developers and systems. Starting from coding standards, a design philosophy is proposed for creating reusable drivers, services, applications and finally, a complete real-time operating system.

Key words: modular design, coding standards, real-time, operating system

1. INTRODUCTION

Effective software engineering requires the ability to rapidly produce and integrate robust and secure code. The key to good software development lies in modular design, testing and code reuse. This is strongly supported by the Unified Modelling Language (UML) design philosophy [1]. By encapsulating functionality with clear interfaces, modules can be identified which can be re-implemented within single applications or across projects.

In the object-oriented environment, modules are typically defined as classes consisting of attributes and behaviours in the form of member variables and methods. In embedded systems, object-oriented design is frequently hampered by intensive platform dependent hardware interfacing and real-time performance requirements. In addition, C, which is still the most popular programming language for embedded applications, is not inherently object-oriented.

An object-oriented implementation of ANSI-C is proposed in this paper, which follows the UML approach for embedded real-time applications. It promotes code reusability through the development of platform independent modules for high-level functionality, but also allows for object-oriented hardware drivers and interrupt handling. In addition it allows for seamless integration with legacy code and supplied software libraries.

Another important aspect of code reusability is maintainability. While design documentation and code comments are of high importance, existing code is often discarded because it is poorly written and incomprehensible. Well-written code should be self-explanatory.

When code is shared among developers, code familiarity can be instilled through coding standards and design patterns ranging from commenting styles to driver interfacing protocols. This can drastically improve the understanding and reusability of code.

After a brief discussion of coding standards, the object-oriented implementation of C is presented. This is followed by design patterns for some common functionality in embedded application development. Note that a working knowledge of embedded software development in ANSI-C and object-oriented programming is required.

2. MODULAR DESIGN

In accordance with the UML philosophy, good software follows a hierarchical architectural design [2]. Through hardware abstraction and good interface design, application software will never need to access hardware directly. Furthermore, through the implementation of services and protocol managers, a further layer of functional abstraction can be created, allowing application software to be completely portable between different hardware systems.

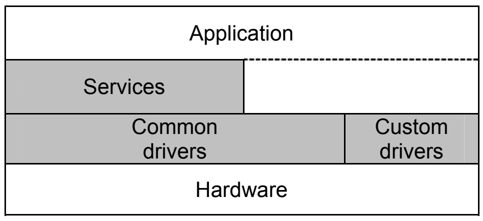

Figure 1: Layered architecture.

Drivers are task independent and can be shared among similar hardware with different applications. Common services and protocol managers can also be shared among different systems, independent of hardware and applications. This architecture is illustrated in Fig. 1. Together, the grey sections produce an operating system.
Common drivers comprise the software that communicates directly with generic hardware. It includes code for setting and reading specific address registers and handling interrupts. Devices on-board micro-controllers also require drivers. While software libraries for such devices are usually available, it rarely uses a common interface and does not implement any object-oriented approach. Fortunately, this code can easily be wrapped in

classes. The goal is to compile a library including all common drivers and services to be shared among systems. The aim is further to implement a common driver interface for various different devices. This allows for the swapping of hardware and drivers with minimal modification of application software, e.g. replacing EPROM with FLASH or an UART with LAN.

In most embedded systems, some hardware is application specific, e.g. discrete hardware and interrupts or custom PLD configurations. Custom drivers are implemented here. In such cases, the value of separate application software and hardware interfaces lies in the ability to upgrade the components independently.

The hardware independent, application specific code forms the application layer. Code in this layer should contain no direct interfacing with hardware. However, large parts of application specific software can often be reused. Typically, code used for communicating between processors may be duplicated on the two sides of the link, creating a communication protocol. This common code can be extracted from the application software and included in the library as a common service.

The required drivers and services from the library can be combined to form a hardware interface. The hardware will require custom configuration to create a hardware specific operating system. Through proper design, an application independent hardware interface can thus be constructed. By making use of common design patterns, this interface can be made generic. This allows the application software to be hardware independent. Depending on the purpose of the system, the applications will however have some minimum functional hardware requirements from the operating system.

Embedded systems are often event driven. Certain tasks are only executed if specific events occur. External interrupts generate the non-deterministic events used to trigger the associated tasks. Interrupts are asynchronous and may originate from an internal or external source, and from hardware or software. These are all handled by the interrupt manager. A scheduler can be used to dispatch deterministic repetitive tasks. With the use of a single timer, the scheduler can generate multiple events at different frequencies, which can be used to trigger synchronous tasks.

The notion of *foreground* vs. *background* processing is frequently used in embedded software design and distinguishes between the main execution loop and the various event triggered threads. Definition of these terminologies differs amongst users and therefore, this reference will not be used here.

While it is possible that all processing can be done within event-triggered threads, this is not good practice. To manage time-critical tasks and task prioritization in systems with asynchronously occurring events, interrupt triggered threads should be kept as short as possible. Only time-critical processing, such as loading values to and from hardware interfaces, should be done in these routines. The main thread should handle all other non-critical housekeeping. Fig. 2 illustrates how the main thread could execute various tasks and different event triggered threads. This ensures that when the processor gets temporarily heavily loaded with concurrent interrupts, the non-critical tasks will get delayed to when processing is less congested.

Furthermore, embedded systems usually have repetitive behaviour. After initialization, the main thread typically enters into an infinite loop. The loop produces the system behaviour, which may be required to change upon meeting certain conditions. These conditions could be detected through continuous polling or through interrupts. In either case, sections of code will need to be included or excluded from the loop. It is generally possible to identify different modes with distinct behaviour. Complex systems may require switching back and forth between modes based on internal or external conditions. In addition, mode changes often require special treatment like the starting and stopping of processes.

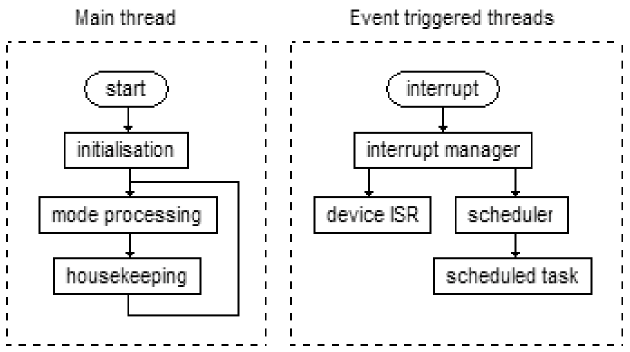

Figure 2: Main and event triggered threads.

The implementation of these requirements fits neatly into the modular design philosophy. Every mode can be defined by a class, which contains functions for normal behaviour as well as mode transitions. These functions can then be appropriately invoked as conditions change.

In an object-oriented language, it is advantageous to let all the modes inherit from the same generic mode interface. Mode changes can then be managed centrally by a mode controller. Generally, there is a great deal of code that has to be executed independent of the current mode. The mode controller can execute this common code by default.

Different instantiated modes can be registered at the mode controller. The main thread enters a loop, which executes mode behaviour and checks for conditions that may require mode changes. Upon detection, it calls the appropriate mode transition code and changes the current mode. Following this, it continues the loop, executing the new mode behaviour. Conditions, which are checked for mode changes, may also depend on the current mode and should be derived from a mode diagram in the system design.

In embedded applications, the main thread typically contains an unconditional loop. Any other loops should be deterministic. Tasks such as algorithm computation can utilise **`for`** loops. When processing serial input buffers where data can be processed faster than it can be received, completion is guaranteed and a while-loop may be used. Using open-ended loops are prohibited. When polling for non-deterministic conditions, the possibility exists that the processor can get trapped in the loop. One option is to implement a time-out counter, limiting the repetitions of the loop.

Often deterministic idle loops are implemented to pause between consecutive condition testing. This is strongly discouraged and for extended time periods it is prohibited, wasting useful processing time. As an alternative, the start of a waiting state can be time stamped and the conditions (including a possible timeout) can be polled once on every pass of the loop in the main thread. This allows for processing of other tasks while waiting.

3. CODING STANDARDS

An extremely effective way of improving code readability is by setting coding standards. This consists of cosmetic rules, token formats, common function prototypes and standard code structures [3].

Whereas code cosmetics give a familiar look-and-feel to the code, predefined types and variable name assignment formats can help readers anticipate compiler interpretations and processor actions. Finally, common function prototypes and interface rules like: "Size is always defined in bytes", makes their usage much more intuitive. Table 1 contains a list of potential C coding standards. These rules can be customized to preferences. It is only important that developer alliances will share the same rules.

Table 1: Coding standards.

Issue	Standard	Reason
File headers	predefined templates define header filenames	Ensure inclusion of author, use, date etc.
Marcos	highly discouraged	Complicates single step debugging
Tabs	4 bank spaces	Prevent misalignment when changing editors
Blocks	for/if/else/while always followed by { in next line; code indented	Prevent assumption of code execution. Readability
Case clause	code indented under case, end with break	Prevent unintended code execution.
Assign-ment	space before & after =, no nested assignments	Readability

The use of common definitions can drastically improve code readability and standardization. However, they should be kept simple and limited in number to ensure that their definitions are well known to their users. Definitions, which are supposed to be shared by all code, are declared in a common header file in Sample 1.This should be included in other files by default.

Sample 1: Generic header predef.h

```
/* Default predefined types and prototypes        */
/* Author: M. Neser           Version 1.0         */

#ifndef predef_h
#define predef_h

#define class typedef struct
#define vol volatile unsigned char
#define TASK void (*)(void*)
#define INTERFACE int (*)(void*, int, int, void*)

typedef enum boolean
{
      FALSE = 0,
      TRUE  = 1
} boolean;

#endif
```

This file demonstrates a simple file header template in its first two lines. This is followed by a conditional block that ends at the last line. This block is included only if the file name is undefined. In this block, the first token defines the file name. This scheme prevents the compiler from attempting to re-include the definition. All header files should implement this scheme. This is followed by a set of common definitions which will be clarified on use in the next section. Finally, a new type '`boolean`' is defined for general condition assignment. Using this type improves readability of arguments.

By following basic naming conventions, tokens can bear additional information. The most widely used standard is Hungarian notation in which variable names are prefixed with key characters signifying their type [4]. A basic list of type keys is given in Table 2. This list can be freely extended to more types.

Table 2: Hungarian notation.

Key	Interpretation	Description
c	char	8 bit integer
i	integer	16 bit integer
f	float	16 bit real value
d	double	32 bit real value
b	boolean (custom)	TRUE = 1, FALSE = 0
v	volatile (custom)	volatile unsigned char
u	unsigned	modifier (prefix)
p	pointer	modifier (prefix)
a	array	modifier (prefix)

This notion can be extended beyond variable names and other token types can be distinguished through similar conventions. Examples of typical C token types and possible naming conventions are listed in Table 3.

Table 3: Token identification.

Type	Token
Predef values	All caps, words separated by underscore
Predef types	Type appended with _t, _st etc.
Class names	Capital C followed by name, capitalized 1st letter for every word, no underscore
Variables	Hungarian notation followed by upper case 1st letter for every word, no underscore
Methods	Capitalized 1st letter for every word, no underscore

Appending predefined types with an underscore and a postfix is in accordance with standard predefined C types like **`size_t`,** which is used to create code with target dependent variable types.

4. IMPLEMENTING CLASSES

The importance of object-oriented programming has been stated repeatedly for advanced software design [5], as seen in the UML development paradigm. In embedded software design, object-oriented programming is especially useful for the instantiation of multiple drivers for duplicated devices and data channels [6].

Real-time embedded software, however, holds a unique set of challenges for hardware interfacing and interrupt handling. While standard C is the programming language of choice for embedded systems, it relies on global variables and lacks classes and inheritance [7], which hinders the object-oriented programming philosophy.

The remainder of the paper describes how a company named Kreon Technology has successfully implemented the object-oriented philosophy for embedded software development with ANSI-C. As far as possible, newly introduced conventions attempt to emulate the Java™ syntax and event model [8].

An object consists of data in a set of member variables as well as a set of methods, which is relevant to the data. Objects and classes are not a standard part of C, but they can be emulated with structures. Methods can be implemented with the use of function pointers.

C classes are therefore defined as structures with member variables as well as function pointers to all the methods. Normally, a constructor function is used to initialise member variables after instantiation of an object. In this case, the constructor also assigns all other method addresses to the object's function pointers. In effect, this results in dynamic method binding. The constructor function is named to emulate the Java™ **`new`** operator.

Neither C nor C++ allows common definition and declaration of classes in one file as is the case with Java™. As a result the :: operator for method declaration in C++ is emulated with an underscore [7].

Class definitions and declarations are done in separate respective header and C files. These files are named according to the class names. It is suggested that the file containing the main function be named 'main.c'. This file could be made quite generic, instantiating and constructing an application class and calling its execution method.

Since the methods of a class instance normally require access to its related member variables, a pointer to the class instance accompanies every function call. This pointer is referred to as the **`this`** pointer and can be accessed from within the method as such. By default every method call will pass the **`this`** pointer as its first parameter.

A device driver provides an interface to hardware through input or output functions. Generally, data can be read from or written to a device. By standardizing this interface, changing hardware e.g. from UART to USB, can be done with minimal code changes.

An illustrative device driver class that gives the driver interface definition is presented in Sample 2. The definition starts with the standard file header, default inclusions and header file definition check block. This is followed by the definition of two utility structures, the context and the device structure. Their functions are discussed below.

Following these structures is the actual class definition. For every method of the class, a compatible function pointer is added to the class definition. This class contains one function pointer to a sample method and one member variable namely the device structure pointer. In practice, the class will contain various function pointers and member variables.

The method shown in the sample follows a predefined prototype. **`Read`** and **`Write`** are reserved interfacing method names that follow a strict design pattern. It extends the conventional C **`read`** and **`write`** function prototypes with *buffer pointer*, *data size* and *data count* [7]. The parameter list is prefixed with the **`this`** pointer and appended with a context structure pointer. These two pointers' types are class dependent. The context structure is used to pass any additional information to the methods. If no additional information is required, this is replaced with a **`void`** pointer to be consistent with the pattern.

Following this design pattern, services can be made independent of the underlying drivers. Driver methods can be dynamically assigned to generic function pointers in the services. By passing the driver methods and context to the service from an external source, the services can interface with the drivers generically.

The driver uses the device structure pointer to access the device hardware registers. By initializing it with the devices offset in memory, the structure forms a memory map to all the devices' mapped registers. Device structure members are defined as volatile.

The class definition is followed by the constructor prototype. This is the only method of the class that is defined outside of the class definition and can be accessed directly. The device offset is passed to the driver as a parameter, making the driver independent of the system memory map.

Sample 3 gives an implementation template of the driver. The standard file header is followed by inclusion of the header file. The methods defined in the class definition are then implemented. In this case, the function names emulate the C++ syntax [8]. Finally, the class constructor implements all *once-off* initialization. Most importantly, the function pointers need to be initialized to the appropriate methods. Any further object initialization is also done here. A parameter, which holds the memory offset of the device, is typically passed. Anything that might be required to be reinitialized should rather be done in a separate setup method.

Sample 2: Header of device.h

```
/* Demo device driver interface definition    */
/* Author: M. Neser              Version 1.0     */

#include "predef.h"

#ifndef driver_h
#define driver_h

/* additional info for read/write function calls */
typedef struct
{
   unsigned int uiOffset;
} ctx_st;

/* map device with config byte and 8 data bytes */
typedef struct
{
   vol vConfig;
   vol avData[8];
} dev_st;

/* device driver class definition */
class CDriver
{
   /* function pointer prototypes */
   size_t (*Write)(struct CDriver *this, void *pBuffer,
                        size_t uiSize, size_t uiCount,
                        ctx_st *stContext);

   /* device hardware pointer */
   dev_st *pstDev;
} CDriver;

/* class constructor prototype */
boolean new_CDriver(CDriver *this, dev_st *pstDev);

#endif /* driver_h */
```

Both internal and external devices are commonly used to generate interrupts, which are used to invoke associated interrupt service routines.

In the object-oriented paradigm, using device driver classes, event handler methods are implemented to handle associated interrupts. These method calls require passing the `this` pointer. This is especially important to distinguish between multiple driver instances, e.g. for duplicate UARTs, where the same method is called for different devices.

To handle this requirement in a modular fashion, an interrupt manager is introduced, which forms part of the set of operating system services.

5. ADVANCED SERVICES

Four services are proposed in this section. The first is the interrupt manager mentioned in the previous paragraph. This service is processor specific and is of great importance for real-time systems. This is followed by two processor independent services, the scheduler and the mode controller, as mentioned in Section 2. Finally, a thread manager is proposed for creating a cooperative multi-threading operating system.

Sample 3: Implementation of driver.c

```
/* Demo device driver implementation         */
/* Author: M. Neser              Version 1.0     */

#include "driver.h"

/* Write method implementation */
size_t CDriver_Write( CDriver *this, void *pBuffer,
                              size_t uiSize, size_t uiCount,
                              ctx_st *pstContext)
{
   /* implementation... */
   return uiCount;
}

/* constructor implementation */
boolean new_CDriver(CDriver *this, dev_st *pstDev)
{
   /* method binding */
   this->Write = CDriver_write;

   /* member initialization */
   this->pstDev = pstDev;

   return TRUE;
}
```

The interrupt manager is responsible for calling the appropriate event handing methods on the occurrence of interrupts. The manager should contain interrupt driven service routines for all the interrupt vectors and must be able to uniquely distinguish between the sources of multiplexed interrupts.

Borrowing from the Java^{TM} event model, event listeners can be registered in association with different interrupt sources [8]. Event listeners consist of pointers to the event handling objects and methods. On identification of an interrupt source, the interrupt manager calls the corresponding registered method. This allows for multiple instances of the same driver to be declared and used independently for duplicate devices.

Services are built on the concept of registering event listeners and tasks. Objects and methods can be generally linked to services, creating an independent functional layer. As is the case with Java^{TM}, a registration method is used to pass the event listener parameters to the services. Event handling methods have to be cast to a predefined method pointer prototype, which can be used by the interrupt manager.

One important function of an operating system is that of task scheduling. Often repetitive tasks such as status message generation and sampling need to be scheduled at fixed intervals. A scheduler can be used to dispatch tasks at different frequencies using a single timer.

The scheduler is invoked at the maximum required frequency for scheduled events. Tasks are registered to the scheduler with their respective periods. A registration method is used to pass all the required parameters. Counters are assigned to the tasks and decremented every time the scheduler is invoked. When a counter reaches zero, the scheduler calls the associated task method.

A mode controller can be implemented as an independent service in a similar way. By implementing every mode's methods in a separate class, each mode can be registered at the mode controller in a common manner. The mode controller typically enters an unconditional loop where it checks for mode change requests and accordingly, calls the mode’s processing method or the requested mode’s entry method.

Incorporating pre-emptive multi-threading in an operating system is an ambitious task. Although it might simplify software design, it is not essential for producing efficient and well-written code. That said, it is possible to realise cooperative multi-threading in the object-oriented paradigm using thread objects.

Each thread object contains its own software stack and a method to instigate a context switch. This method suspends the calling thread and resumes another. Threads are registered at a kernel class during construction, which is responsible for context switching. This typically involves intricate processor specific machine code, which is outside the scope of this paper. Pre-emptive multi-threading can be made possible by invoking context switches from scheduled time-slicing events [9].

As depicted in Fig. 1, a final notion is to combine all the implemented services and drivers into a system class to create a complete operating system. The system class is responsible for the interconnected construction and initialization of all these components. Sample 4 demonstrates how a timer and different services are constructed and a task is set up.

Sample 4: Code from system.c

```
void new_CSystem(CSystem *this)
{
   new_CInterrupt(&this->interrupt);
   new_CScheduler(&this->scheduler);
   new_CTimer0(&this->aTimer[0], (timer_dev*),TIMER0);

   this->timer.Setup(&this->aTimer[0],1000);

   this->interrupt.RegTask(&this->interrupt,
                  &this->scheduler,
                  (TASK)(this->scheduler.Process),
                  TIMER0_INT)
```

6. CONCLUDING REMARKS

It is good object-oriented practice to restrict access to member variables from outside an object. This is especially true for changing values – only the object itself should be allowed to modify its variables. To update member variables of another object, new values should be passed via access methods of that object, allowing the object to handle any potential consequences of the change.

`Get`***`Variable`*** and `Set`***`Variable`*** are common design pattern names for such access methods. For passing larger structures, the **`Read`** and **`Write`** design patterns can be utilised [8].

Various other regulations may also be implemented to improve determinism in the execution of software. These may range from enforcing single function exit points for improving traceability, to the prohibiting of dynamic memory allocation; depending on the implied burden on the CPU and RAM (dynamic memory allocation is considered a memory leakage risk and is therefore, not recommended in application specific systems).

The paper shows that the object-oriented philosophy can be followed in ANSI-C for all levels of embedded software development. Furthermore, the overheads accompanying this approach involve limited machine instructions, one additional pointer per function call on the stack and one pointer in memory for every object method.

The cost of software development for embedded systems often overshadows all other costs [2]. Since object-oriented programming can greatly reduce development time, the cost savings from implementing this approach usually outweighs the cost on resources, even if it implies the utilisation of more expensive hardware.

7. ACKNOWLEDGEMENTS

The authors would like to thank Kreon Technology Ltd. for its support and in particular, Mr. L. van Tonder (leonvt@kreon.co.za) for his contribution to the standards proposed in this paper.

8. REFERENCES

[1] B.P. Douglass: "Real Time UML Workshop for Embedded Systems", *Newnes*, 2006

[2] J.G. Ganssel: "The Art of Designing Embedded Systems, 2nd edition", *Newnes*, 2008

[3] S.R. Schach: "Object-oriented and classical software engineering, 7th edition", *McGraw-Hill Companies, Inc.*, 2006

[4] F. Kuester and D. Wiley, "Software Development and Coding Standards", vis.eng.uci.edu/standards/pdf/codingstandards.pdf, 23 October 2006

[5] A.T. Schreiner: "Object-oriented Programming with ANSI-C", *Hollage*, 1993

[6] S. Bhakthavatsalam and S.H. Edwards, "Applying object-oriented techniques in embedded software design", *Proceedings of the CPES 2002 Power Electronics Seminar and NSF/Industry Annual Review*, 2002

[7] B. Bramer and S. Bramer: "C++ for Engineers", *Butterworth-Heinemann*, 2001

[8] H.M. Deitel and P.J. Deitel: "Java How to Program, 7th edition", *Prentice Hall PTR*, 2007

[9] J.J. Labrosse: "MicroC OS II: The Real-Time Kernel, 2nd edition", *Newnes*, 2002