Introduction

ButterflySDK is a set of software files written in C programming language to be used with ATMEL's Butterfly Evaluation Board. The Butterfly is popular among hobbyists and developers because of its unique features and its remarkably low price. Another aspect that adds to the appealing of the board is the free C compiler (AVR GCC) for the board's AVR microcontroller. ATMEL ships the Butterfly with a pre-loaded application and makes the sources, written for the commercial IAR AVR compiler, available from the Internet. The application has been quickly ported to the free GCC AVR compiler by vigorous members of the AVR community.

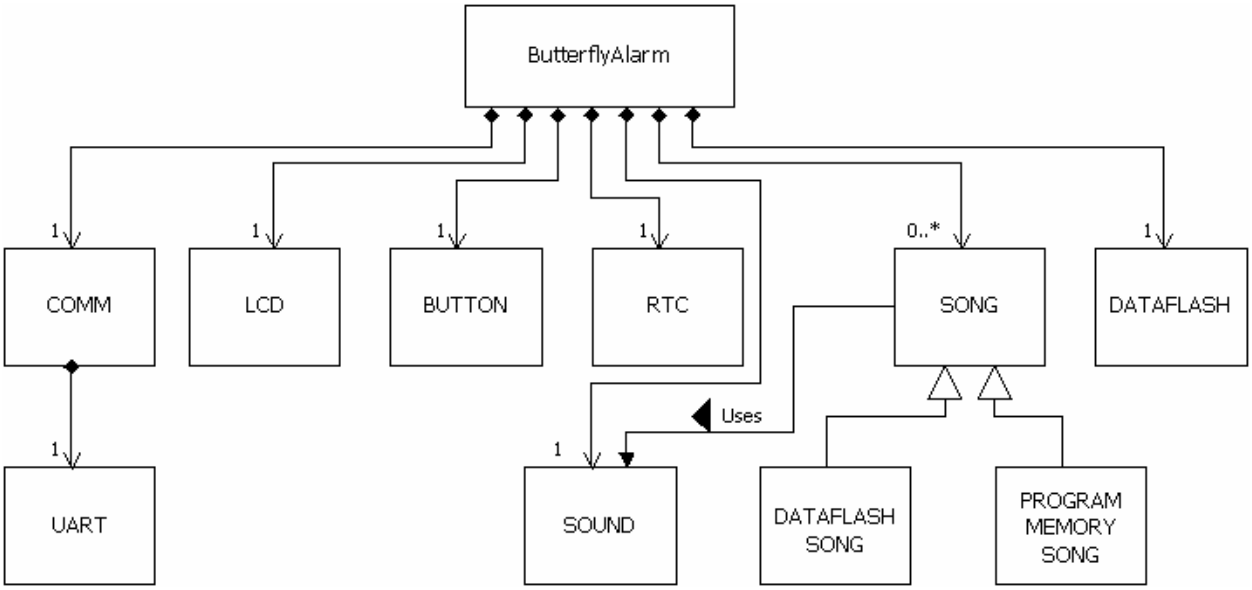
This code has introduced many developers to the C programming language for the AVR microcontrollers. That's why I consider very unfortunate the fact that the ATMEL code and the corresponding GCC port are of such a low quality. This code breaks many rules of the generally accepted good coding practices. I suppose this has happened, because the majority of the software developers for embedded 8-bit platforms are either new to programming in high level languages and therefore unaware of the modern high-quality code guidelines or they simply believe, that applying these rules to resource constrained platform is impossible. This last bit is far from being true however.

I developed ButterflySDK to try to correct this situation. I will try to demonstrate that following modern high-quality code guidelines is not only possible for 8-bit platforms but can also lead to faster, more compact and more robust applications. Furthermore the resulting code will be much more readable and maintainable and could be easily re-used in new Butterfly projects.

ButterflySDK will be presented as part of a real application, named ButterflyAlarm, which allows you to use the Butterfly board as a personal alarm clock with downloadable ring-tones! I hope this will make the process a little more interesting.

Software Architecture

The following figure depicts a UML class diagram of the ButterflySDK software. Embedded developers may be unfamiliar with UML diagrams, as UML is usually associated with object-oriented programming languages like C++, JAVA and C#. However UML can definitely be used to describe the design of an embedded system programmed in C or even assembly. In fact UML can be used to describe any system, not just software programs. A UML class diagram is beneficial for the structural design of an embedded system. It can help the designer to identify the system's components, the inner-workings of them and the required inter-connections.



Of course the implementation of the UML class diagram in C programming language is not so straight forward, as it would be if C++ or JAVA were to be used. However by applying certain techniques, which will be described in detail at the following section, it is possible to emulate many principles of an object-oriented language in C (abstract data types, information hiding and even some form of inheritance).

The UML class diagram is useful for the structural design of the system but does nothing for the behaviour modelling. The way that the modules interact with each other in run-time and their response to various events is of crucial significance in embedded software design. In fact, this requirement for the system to respond to events in well-defined limits differentiate embedded systems from other applications.

A number of software architectures, both with and without an RTOS, are widely known and used in embedded systems world. When no RTOS is used, the most common options will be 'Round-Robin' and 'Round-Robin with interrupts' architectures. Then there are co-operative and pre-emptive multi-tasking operation systems. From the above alternatives a pre-emptive

multi-tasking RTOS will be the optimum solution, as it offers the greatest flexibility and the smallest latencies. However in cases, where every single byte of program memory matters, its overhead in the program memory size will be a serious drawback. Anyway, there are many things that are important when choosing an architecture and these are best to be considered in a case-by-case basis.

For the ButterflySDK I chose a 'Round-Robin with interrupts' architecture. I can easier describe this architecture with a block of pseudo code:

```
while(1)
{
	BackGroundTask1

	BackGroundTask2

	if (interruptFlag1)
		ServeInterrupt1

	if (interruptFlag2)
		ServeInterrupt2
}
```

The above block shows how the endless loop in the main routine should look like. An endless loop in the main routine is very common in embedded platforms, because the main routine should never exit. All processing required runs from this loop. The only thing that runs outside of the loop are the Interrupt Service Routines (hence the name 'with interrupts'). These Interrupt Service Routines (ISR for short) run asynchronously from the main execution path, as they are triggered from external events. They have higher priority from the main routine, i.e. they stop the execution of the infinite loop to run. When the ISR exits, the execution of the main routine is resumed.

There are usually two kinds of tasks that run from the infinite loop. First there are the, what I call, background tasks. These are completely unrelated from the interrupts and run as often they can. For this reason they are sometimes referred as the 'idle' task. They should execute fast and perform no time-critical operations. The second type of tasks is the 'ServeInterrupt' tasks. A 'ServeInterrupt' task is co-related with a single interrupt source and shares some data with an ISR. When the ISR is run, it buffers some data and sets an interrupt flag. The main loop will check if this interrupt flag is set and when this happens, the corresponding 'ServeInterrupt' will be invoked. This task will process the data buffered from the ISR. Therefore the interrupts are actually served in two steps. First is the on-time processing from the ISR. The ISR performs only the absolutely necessary processing to guarantee that the latency from the actual event is as small as possible and that no data will be lost. Before it exits, the ISR sets the interrupt flag. Then at a later point, the data will be processed by the 'ServeInterrupt' task.

When using this architecture, it would be better to avoid nesting interrupts (allowing one ISR to run, while another one is still running). This would make hard to analyse the run-time behaviour of your code and can lead to unexpected results. Therefore, when an ISR is being executed, no other operation is allowed to run. For this reason the execution time of all Interrupt Service Routines should be as small as possible. All time consuming operations should be moved to the 'ServeInterrupt' tasks.

'Round-Robin with interrupts' is a simple and flexible architecture. It can help you produce a responsive system and also have a clean code. However it also have some serious drawbacks. First, it doesn't allow you to set priorities to the 'ServeInterrupt' tasks. (One not so perfect solution will be to make a task of higher priority appear more than once in the infinite loop). The most serious drawback however is, the silent assumption it makes, that the period of the interrupts and the duration of the tasks are such, so that a 'ServeInterrupt' task is allowed to run at least once between two successive interrupts. The worst part about this is that a period of an interrupt and the duration of a task affect all other tasks. You can't concentrate on a single ISR - 'ServeInterrupt' task pair at a time, but you need to consider the system as a whole. It is responsibility of the programmer to analyse all tasks, to calculate their duration in worst-case scenario in order to be able to determine, if the system will actually work. This is probably the most difficult part in embedded software without an RTOS design.

Run-time analysis of an embedded software system is inherently difficult. It's not good to make it even more difficult, by having a code-bloated main routine, like that in ATMEL's original Butterfly code. Instead we must have a simple and compact main routine, with distinguishable parts that we can easily analyse. This also facilitates the development, as it practices the 'golden rule' of software development, namely 'Divide and Conquer'. It will also help us, if we want to migrate to an RTOS design. We can see how is that by comparing the following code blocks.

Code with no RTOS	Code with RTOS

`int main(void)`
`{`
`  HWInit();`
 	
`  enable_interrupts();`
 	
`  while(1)`
`  {`
`  Task1();`
 	
`  Task2();`
`  }`
`}`
	`int main(void)`
	`{`
	`  HWInit();`
	`  osInit();`
	
	`  osStartTask(Task1);`
	`  osStartTask(Task2);`
	
	`  osStart();`
	`}`

One final note about the ButterflySDK architecture has to do with the power management mechanism. Since the board will be mainly operated by the coin-cell battery, we have to set the micro-controller to Power-Save mode as often as possible. To achieve this, the infinite loop in the main routine has to be modified as follows:

```
while(1)
{
      BackGroundTask1

      BackGroundTask2

      if (no interrupt pending)
            sleep

      // Wake from sleep here

      if (interruptFlag1)
            ServeInterrupt1

      if (interruptFlag2)
            ServeInterrupt2
}
```

Design Guidelines

Abstract Data Types

The UML class diagram in the previous section presents the components of the ButterflySDK software. Before starting the implementation of a new system, even before selecting a programming language, one must identify the logical entities that compose the system. These entities are usually called 'Abstract Data Types'. We say that an Abstract Data Type 'encapsulates' or models the behaviour or the structure of an actual component. Choosing the appropriate abstractions is the first task that needs to be performed in the design phase. I like to think the concept of the Abstract Data Types as the application of the 'Divide and Conquer' principle in software design. Splitting a system in Abstract Data Types permits you to think only one part of the system at one time. Instead of solving one big problem of great complexity, you only need to solve many simple problems.

In ButterflySDK an Abstract Data Type is implemented by a single C file and its corresponding header file. For example, the LCD component is implemented in the lcd.c and lcd.h files. Everything that has to do with the LCD is to be found in these two files. Most importantly, these two files include nothing else but LCD code. This modular design facilitates, among others, the re-use of the code. Since the LCD code is not coupled with any other functionality, it is very easy to re-use it in a future project.

In object oriented languages there is a constructor and a destructor associated with every object. In embedded software you almost never need a destructor but a constructor is always required. The constructor is the routine that initialises the module. In ButterflySDK I create an initialisation routine for every module. This routine is named by the module's acronym and the 'Init' affix and plays the role of the constructor (e.g. `LcdInit`, `ButtonInit`, `SoundInit`). I also impose the requirement to the users of the module, to call this routine before start using the module's functionality. Inconvenient as this may be, there is no way in C to automatically call a constructor. You may notice that static and global variables are initialised in these constructor-like routines and not when they are defined. This helps me to group all initialisations commands and processes in a single place. I actually believe, that in embedded software is a good habit to not initialise static variables in their definitions. This can help you to achieve shorter system start-up times and will make your code more portable.

Data hiding

Another key concept of the Object Oriented design is that the Abstract Data Types hide from the external world the implementation details and only expose hooks, which the caller of the Abstract Data Type manipulates to make use of the offered functionality. This mechanism is called 'Data Hiding' or 'Information Hiding'. In object oriented languages it is implemented by defining the members of a class to be either *public* or *private*.

Data hiding helps both in design and implementation level. In design level it helps the developer to think beforehand, what functions an Abstract Data Type should perform and what the users of the Abstract Data Type will need. In implementation level, it allows a module to be internally modified without affecting the external world. Another advantage is, that it prevents a user of the Abstract Data Type to accidentally misuse it. These greatly promote software re-usability and maintainability.

In ButterflySDK a private variable is implemented by a *static* variable. I prefix these static variables with the name of the module they belong to. E.g. `lcdTimer`, `rtcSecond`, `buttonDebounceCounter` are all 'private' variables. A static variable can be used from any routine in the C file it is defined, including the Interrupt Service Routines, but it is inaccessible from any other code. Therefore it fits perfectly with the private member concept.

To make a function public, I simply add its declaration to the module's header file. All functions that are not declared in the header file are private members. Generally I only include in the header file the definitions of structures, macros, constants and the declarations of functions that the users of the module need. Everything else is defined in the C file. It may seem strange at first to see typdef and macro definitions in a C file, but I feel this promotes the data-hiding concept. If this is undesired, one could create two header files, the first with the public and the second with the private definitions.

Although every Object Oriented language I know permits the definition of public variables, good practice dictates public variables to be avoided. One could correspond public variables in an Object Oriented language with global variables in C programming language. I will discuss global variables shortly.

Inheritance - Polymorphism

Inheritance is a means for extending an existent class in order to enhance its functionality. The functionality of the 'base class' (also called parent class) is re-used from the derived class. The new class can be used in place of the base class.

Of-course you can't implement inheritance in C. However it is still beneficial to think in terms of Object Oriented Design in the design phase. In ButterflySDK I want for example to support two types of songs: songs stored in program memory and songs stored in dataflash. The user downloads the songs in the dataflash and these are the songs that will be mainly used. If the dataflash is however empty or it gets corrupted, then the system will revert to program memory songs. These two Abstract Data Types share a great deal of functionality. They can be both derived from a single 'base class', that defines the representation of the song and the functions that need to be implemented. This base class is partly implemented in songs.h file. The program memory and dataflash songs modules use this header file and they both define and export a common set of functions (`SongSelect`, `SongGetNote`).

Both types of songs are played by the Sound module. The user selects the song to be played, either a program memory or a dataflash song. It would be very convenient for the Sound module to have a single routine for playing songs and determine which method to call in run-time. This is called polymorphism and is implemented in Object Oriented languages by *virtual* functions. In ButterflySDK I use an if-clause to call either `SongGetNote` or `DfSongGetNote`.

Global Variables

As mentioned previously the use of public member variables is avoided in Object Oriented languages. Instead the class provides 'getters' and 'setters' public methods, which the callers of the class can use to modify and read the variables. In embedded software design the global variables of a module can be considered to play the role of public variables. However use of 'getters' and 'setters' is in this case usually avoided, because they introduce both a code size and time performance penalty. The most common thing done, is to add a declaration of the global variable in a header file and access it directly from external code. However this is far from a good solution. Global variables increase the coupling between logical independent modules and make the code more difficult to understand and maintain. They also introduce the risk of inadvertently setting an invalid value to a variable. On the contrary private variables are only accessed internally. The module that owns the variable is aware of its allowed value range and can guarantee that if a variable modification requires some extra processing, this extra processing will be actually performed. All these have to do with the implementation of the module, for which the user of the module supposedly knows nothing about. Therefore the user of the module can easily misunderstand the use of a variable. Or perhaps the variable is used correctly at first but then the implementation changes and the software stops working.

Another reason that 'getters' and 'setters' are a better solution than global variables, is that we normally use verb phrases for functions names and can therefore clearly self-document what the function does. For example if the modification of a variable requires resetting a module, we can have a function named `ResetModule(char variable)`. It's obvious from the function's name that we don't simple set a variable, but we also perform a reset. 'Setters' functions can also return 'False', if the value is out of range. All these can't be achieved by simply handling global variables.

In ButterflySDK I use both 'getters' and 'setters' functions and global variables. One example of the 'getters' and 'setters' concept can be found in the RTC module:

```
void RtcSetMonth(uint8_t month)
{
	if (rtcMonth < 12)
		rtcMonth = month;
}

uint8_t RtcGetMonth(void)
{
	return rtcMonth;
}
```

As you can see the setter function not just sets the month variable but also performs a range test. For such simple functions the size and time overhead is small. When the source is compiled with the size optimisation flag, the GCC AVR compiler generates no prologue or epilogue blocks for the above functions and the size overhead is negligible. The time overhead we need to pay for using this function stems from the CALL and RET instructions, which add 7 clock cycles.

There are situations, where these overheads, small as they may be, are unacceptable. In ButterflySDK I use some global variables, not because I have strict size and time requirements, but because I want to demonstrate how one should deal with them. These global variables are mainly interrupt flags, i.e. variables that the ISR set to indicate to the outer world that an interrupt has occurred. The modules that own these global variables read from and write to them freely. However the external modules are only accessing them through macros. The declaration of the variable is still present at the header file of the module, but the macro clearly documents its use. When someone else looks at the code (including the original developer after some time), it would be clear to him or her, how the variable should be used.

I just want to add a final note about global variables. Global variables should not be confused with 'Singleton' objects. 'Singleton' objects are declared the same way as global variables, but they are actually design abstractions and do not belong to a specific module. They represent a

unique system object - there is only one object of this type in the system. One such example in ButterflySDK is the song selected by the user. This is defined in sound.c file. If that helps, you can move it to a C file of its own.

In software engineering a general repeatable solution to a commonly occurring design problem is called a 'Design Pattern'. One such 'Design Pattern' is the 'Singleton'. RTC module makes use of the 'Observer Design Pattern'. Design Patterns are very popular in Object Oriented Design and it is a concept that every software developer should be aware of.

Code Conventions

Everyone in software development world agrees that there is a need of applying a set of specific rules to the way we write code. These rules are usually referred to as code conventions and they deal with the code layout, the naming conventions, the comment technique and others. The need for these rules stems from the observation that *"Code is read must more often than is being written"*. We actually write the code once and then we read it many times, in order to understand what the code does and how it does it. Therefore it is important to make the code as 'readable' as possible. By the word 'readable' we mean code that is both visually pleasant to look at and easy to understand.

For ButterflySDK I have adopted my own personal favourite code conventions. These are by no means standard and many developers would have used a different set of rules. The important thing is however, that the code uses specific rules and is homogeneous in style. The following is a brief description of the conventions I've used.

- *Indentation*: I use tabs of 4 spaces size for indentation. Many developers will prefer spaces, but I find that it is easier to move back and forth the indentation levels, when tabs are used. Anyway, any smart text editor can convert one style to another instantly.
- *Code layout*: I use the first of the following code layouts:

Curly braces in their own line	More compact layout

`if (expression)`
`{`
`  do something`
`  for(i=0; i<10; i++)`
`  {`
`  process`
`  }`
`}`
	`if (expression) {`
	`  do something`
	`  for(i=0; i<10; i++) {`
	`  process`
	`  }`
	`}`

The choice between these two styles is perhaps the most controversial topic in code style discussions.

- *Whitespace*: I usually leave an empty line between two logically separate blocks of code
- *Line size*: I try to keep the lines of code shorter than 80 characters.
- *Name conventions*: I use 'Pascal Case' for naming functions (`DoSomething, ResetValue, StepAndShoot`). As mentioned before, function names are always verb phrases that describe, what the function does. To make evident the module to which the function belongs, I prefix public methods with the name of the module (`LcdInit, RtcSetYear, SongGetNote`). For variables I use 'Camel Case' (`lcdTimer, songsCount, rxIndex`). The names of structures are always nouns and the first letter is capitalised (`Alarm, SongInfo`).
- *Strings in flash*: Strings in program memory and the routines that take them as arguments are affixed with `_F`. I normally don't like this kind of notation (underscores, use of the variable's type in its name) but I believe this exception to the rule is justified.
- *Macros and constants*: All constants and macro names are fully capitalised words, separated with underscores. I do not extend this rule to constant flash strings. This rule is the most ubiquitous one, adopted by code conventions for nearly every programming language. However ATMEL's code violates it by naming non-constant variables with all letters in uppercase. I find this inexplicably confusing.
- *Comments*: I believe that the code should be self-documented. Comments are generally not needed and should only be used, when a code block performs a tricky or a not so obvious operation. Commenting the obvious, like:

  ```
  i++; //Increment i
  stage = 0; // Set stage to 0
  ```

  is irritating at least. However in embedded code I make heavy use of comments in code that accesses the hardware:

  ```
  // Set Clock Prescaler Change Enable
  CLKPR = (1<<CLKPCE);

  // Set prescaler = 8, Inter RC 8Mhz / 8 = 1Mhz
  CLKPR = (1<<CLKPS1) | (1<<CLKPS0);

  // Disable OCIE2A and TOIE2
  TIMSK2 = 0;
  ```

  I do so, because I want the reader of the code to have an idea of how the code works, without having to refer to the hardware manual.

- *Documentation comments*: I use Doxygen comments to automatically generate documentation from the sources.

Software Modules

Documentation for the ButterflySDK software modules has been produced with the aid of 'Doxygen' tool. In this section I will focus on the differences - enhancements over the ATMEL's original Butterfly code.

LCD module

LCD is an important module of the system. The use of the megaAVR's internal LCD controller gives the board a technological and pricing advantage. The code supplied by ATMEL for the LCD module is over complicated. Not only that, the use of the LCD functions is tricky. It seems that for the LCD to operate, you need to call the mysterious function `LCD_UpdateRequired` from the user's code. Anyway, the ButterflySDK LCD code is much easier to use. You basically only use the `LcdWriteString` routine and include the `LcdTask` in the main routine's infinite loop.

To be able to explain the software, a few words about the logic behind the code are required first. Writing to the LCD is performed in three steps. First we write the text in a text buffer. In this step we don't care about scrolling. This will be handled automatically later on, based on the length of the text. Flashing is controlled by the most significant bit of the text characters. The basic operations are performed from the `LcdTask`. This task takes the text characters and translates them to LCD segment codes. It puts the result of the processing in a mirror of the LCD registers in RAM. This task is responsible for producing scrolling and flashing effects. Then, when the LCD Start Of Frame interrupt is run, the RAM contents will be copied to the LCD registers. Actually the `LcdTask` will only run after a LCD Start Of Frame interrupt. We don't need to worry about this reversed order however, because it is not necessary to update the LCD registers in every Start Of Frame cycle.

This architecture has numerous advantages:

- It has a very compact and fast LCD Start Of Frame ISR.
- The caller of the code only uses simple printf-like functions.
- It separates basic LCD functions from scrolling and flashing effects. These effects can be easily modified or removed without affecting all the code.
- All the code is included in a single file. This file includes nothing but LCD code.

This last thing has to do with the ATMEL's code including button debouncing functionality in the LCD Start Of Frame ISR. It does so, because this ISR provides the 'tick' of the system. When in Power-Save mode, the micro-controller will wake from sleep to run this ISR. Interrupts from Timer0 will

not wake the processor and interrupts from Timer1 are 1 second apart. Therefore the only way to implement debouncing is through this interrupt. We don't need however to put the code directly in the ISR. Button debouncing functions are not time-critical and can be easily served by a 'ServeInterrupt' task.

Sensors module

The sensors module performs temperature, light and voltage measurements. Before going on I want to confess, that I don't know how an ADC value is translated to a light measurement and how the light sensor affects the voltage reference (as implied by the ATMEL's code). Perhaps a hardware developer could explain these two points to me.

What I did differently from ATMEL was first to separate the ADC functionality, the sensors logic and the measurement display routines. The ADC routines deserve a module of their own, which can be re-used in many future projects. The sensors logic (processing required to get the sensor measurement from the ADC reading) was put in one routine and the measurement display in another. This technique, i.e. using different routines for logical different operations, is followed throughout the ButterflySDK code. Why this is a good thing, should be obvious to you. If not, you should spend some time thinking about it.

The other enhancement over the ATMEL's code is the exclusion of floating point operations. In almost every embedded software book you will read that floating point arithmetic operations should be avoided, as they introduce big size and time overheads for little accuracy. Indeed, operations involving `float` data types can lead to unexpected results, because of the approximations involved. One must be aware of the basic principles of arithmetic analysis and only use them with extreme care. `double` data types alleviate the problem to a great extent, but it is almost impossible to use them with an 8-bit processor.

Anyway floating-point operations are only needed in rare occasions. Most times you can get away by using fixed-point arithmetics and pre-calculations. The first thing you should do however is to think if you can solve the same problem with integers. In voltage measurements for example I need to measure volts from 0 to 5 Volts in steps of 0.1 Volt. I can easily represent this value with an integer in the range 0..50. Not even fixed-point numbers are necessary.

Another thing you can do is to pre-calculate some values. In voltage reading routine I have to look up a value in a table, multiply it by the ADC value and then divide it by 6. I can however create my look-up table with the division by 6 already performed. Because even integer arithmetic operations are time consuming in 8-bit processors, the more you can pre-

calculate the better. Keep in my mind however, that in software calculations the order of the processing is important. `(a*b)/c` is not the same with `a*(b/c)`. Also beware of the overflows. The maximum value that a 16-bit integer can hold is 65535. You can easily overcome this value by multiplying two integers.

RTC module

The RTC module features an interesting implementation of the 'Observer Design Pattern'. The design problem we need to solve, is to be able to inform more than one modules that an RTC event has occurred. These modules are called 'listeners'. The 'Observer Design Pattern' dictates that every 'listener' subscribes to the 'publisher' of the events, namely the RTC module. This is achieved with the following code:

```
uint8_t listenerId = RtcRegisterListener();
```

Then a listener queries the RTC module about new events using its acquired ID:

```
BOOL newEvent = RtcNewEvent(listenerId);
```

A new event occurs every one second. Therefore the above function will return TRUE once a second. Upon returning TRUE the internal state of the RTC module for the specific listener will be cleared and the function will not return TRUE again for one second. However if we call the function with a different listener's ID, it will return TRUE. In this way we can have Clock, Date and Alarm modules all monitoring RTC events without one interfering with another.

Songs module

I use a slightly different song representation from ATMEL. I basically bundle the duration and the tone of a note in two bytes to save space. I also use macros to perform the bundling and un-bundling, so that the songs remain human-editable and the song reading code is not affected much. You can compare ATMEL's and ButterflySDK song representations in the following code blocks.

ATMEL Song

```
const int FurElise[] PROGMEM=
{
  3,
  8,e2, 8,xd2, 8,e2, 8,xd2, 8,e2, 8,b1, 8,d2, 8,c2, 4,a1, 8,p,
  8,c1, 8,e1, 8,a1, 4,b1, 8,p, 8,e1, 8,xg1, 8,b1, 4,c2, 8,p, 8,e1,
  8,e2, 8,xd2, 8,e2, 8,xd2, 8,e2, 8,b1, 8,d2, 8,c2, 4,a1, 8,p, 8,c1,
  8,e1, 8,a1, 4,b1, 8,p, 8,e1, 8,c2, 8,b1, 4,a1,
  0, 1
};
```

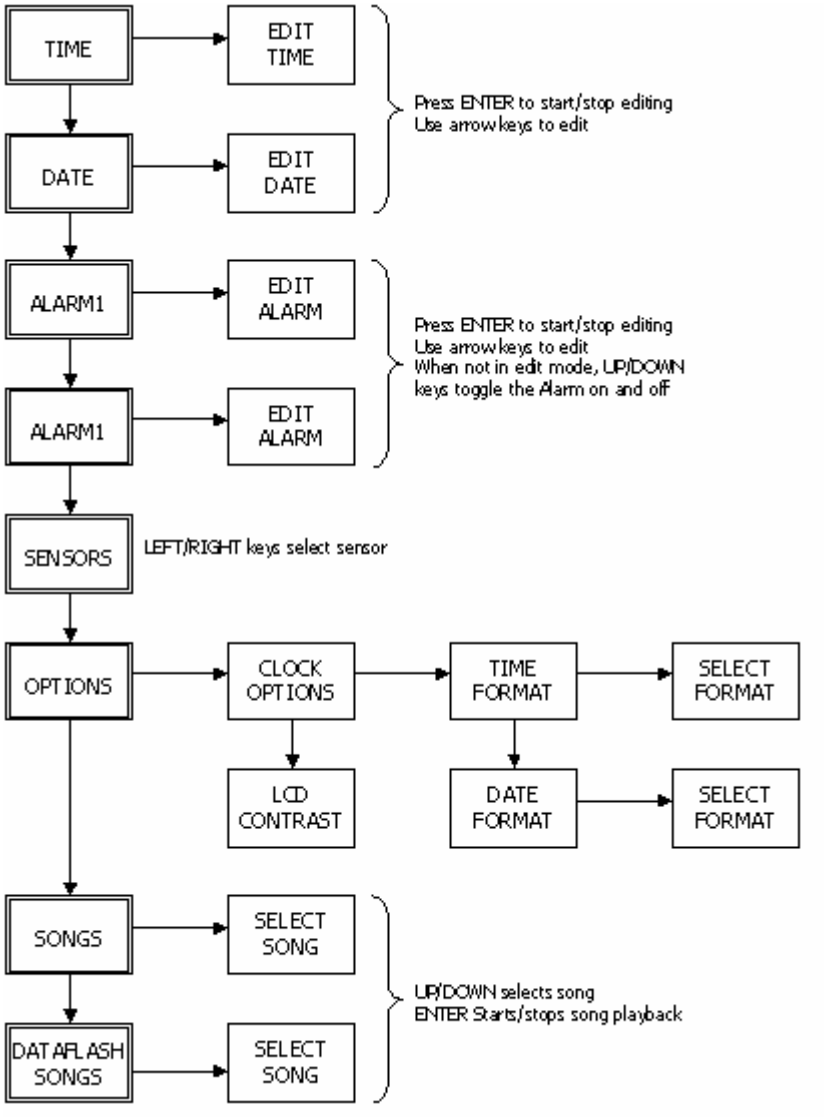
ButterflySDK Song

```
const uint8_t FurElise[] PROGMEM =
{
  'F', 'u', 'r' , ' ', 'E', 'l', 'i' ,'s', 'e', '\0', '\0', '\0',
  SONG_SETTINGS(3, 0, 40),
  NOTE(D_8,e2 ), NOTE(D_8,xd2), NOTE(D_8,e2 ), NOTE(D_8,xd2), NOTE(D_8,e2 ),
  NOTE(D_8,b1 ), NOTE(D_8,d2 ), NOTE(D_8,c2 ), NOTE(D_4,a1 ), NOTE(D_8,p  ),
  NOTE(D_8,c1 ), NOTE(D_8,e1 ), NOTE(D_8,a1 ), NOTE(D_4,b1 ), NOTE(D_8,p  ),
  NOTE(D_8,e1 ), NOTE(D_8,xg1), NOTE(D_8,b1 ), NOTE(D_4,c2 ), NOTE(D_8,p  ),
  NOTE(D_8,e1 ), NOTE(D_8,e2 ), NOTE(D_8,xd2), NOTE(D_8,e2 ), NOTE(D_8,xd2),
  NOTE(D_8,e2 ), NOTE(D_8,b1 ), NOTE(D_8,d2 ), NOTE(D_8,c2 ), NOTE(D_4,a1 ),
  NOTE(D_8,p  ), NOTE(D_8,c1 ), NOTE(D_8,e1 ), NOTE(D_8,a1 ), NOTE(D_4,b1 ),
  NOTE(D_8,p  ), NOTE(D_8,e1 ), NOTE(D_8,c2 ), NOTE(D_8,b1 ), NOTE(D_4,a1 ),
};
```

ButterflyAlarm

ButterflyAlarm is the fun application I promised to build with ButterflySDK. It allows you to use the Butterfly board as a personal alarm clock with downloadable ring-tones. A PC application, named ButterflyConnect, is used to connect to the Butterfly and manage its song book. ButterflyConnect is developed with wxPython and can run in a multiple of platforms.

ButterflyAlarm stores songs not only in Program Memory but also in the dataflash. This means that new songs can be downloaded without firmware modifications. ButterflyConnect caters for the songs downloading.



ButterflyAlarm menu is different from that in the original application. The above figure depicts how the various operations are accessed.

Besides its use as an alarm clock, ButterflyAlarm can be used to demonstrate the features of the Butterfly board (thus replacing the ATMEL's original application). ButterflyAlarm has the following improvements over the ATMEL's application:

- There are more features in the same program size
- Button click sound works
- The menu is accessible even when a song is playing

Please note that is strongly recommended not to operate the Butterfly with the coin cell battery as power supply, when connected to a PC. Not only the RS-232 transceiver will drain the battery out quickly, but also the dataflash write operations require currents that the battery can't supply.

A subset of the operations of the ButterflyAlarm is emulated by a Javascript Applet I wrote. You can find it at http://users.forthnet.gr/ath/kgiannak/.

Emulating an embedded system using a PC programming environment is not only fun, but can also be very helpful. At first, you could demonstrate the idea of your design early in the development cycle. An emulation also helps you to test the usability of your user interface (what kind of a display is needed, how many buttons are needed). Finally emulation also helps in debugging. They are parts in software that have nothing to do with the hardware. Any code that performs a complex logic operation, like a sorting algorithm for example, is much easier to be tested in a PC than in a embedded system. When the C programming language is used, one can easily write the code in such a way, that in runs both in a PC and a micro-controller system. This technique can help you spot logical errors faster.

Summary

Tips

This is a summary of my recommendations for embedded software development.

Design

- Do a structural design first. Identify the Abstract Data Types of the system.
- Write the main routine's infinite loop in pseudo code. Identify the system's tasks and analyse their timing requirements. Based on the result of the analysis determine whether 'Round-Robin with interrupts' architecture is suitable or an RTOS is needed.
- Follow the 'Divide and Conquer' principle. Have a modular design.
- Hide the implementation details of the modules from the external world.

Implementation

- Implement a module with a .C and a .H file.
- Use the `static` keyword for all the module's internal variables.
- Avoid using global variables. Use getters and setters instead.
- Use macros to modify/read global variables.
- Use a different routine for logically different operations.
- Avoid floating point operations. Use integer or fixed-point operations and pre-calculations instead.
- Do not initialise static and global variables in their definitions.
- Do not comment the obvious. Do comment however tricky operations and code that accesses the hardware.
- Define and adopt a set of code style conventions.

Tools

This is a collection of tools every developer should be aware of. You may use none of them, but you still need to know that they exist and what they are good for.

- Doxygen: Documentation auto generation tool
- CVS: Source-control tool
- SVN: Source-control tool. It is considered to be better than CVS.
- WinMerge: Tool for comparing and merging files.
- CTAGS: Generates an index (or tag) file of C language objects found in C source and header files that allows these items to be quickly and easily located by a text editor.

- mirkes.de Tiny Hexer: Useful and free hex editor.

The above list is by no means complete. Many alternatives to the above programs exist, both commercial and free. The list doesn't include a text editor. However this is the most important tool in the developer's arsenal. I recommend to select your editor of choice, preferably one that can be easily extended and use it for all your development needs, both embedded and not. Do not rely on the editors supplied by compiler vendors or hardware manufacturers. I also recommend investing some time in at least one scripting language, like Perl or Python, or a scripting shell. This will greatly help you automate your tasks.

Further Reading

- *Code Complete Second Edition - Steve McConnel - Microsoft Press*. Everyone writing an article or a book about practices for high-quality code references this book. It is considered a definite read for all software developers.
- *An Embedded Software Primer - David E. Simon - Addison-Wesley*. This is a good book I've read about embedded software. They are many more embedded software books available.
- *Design Patterns - Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides - Addison-Wesley Professional Computing Series*. This is the book that introduced the idea of the Design Patterns.
- The Humble Programmer: An essay by the great Edsger W. Dijkstra. Although written in 1972, it still remains useful and educational. It presents many interesting ideas. The one that I find more fascinating is that in software design high quality code doesn't imply a higher code. On the contrary by practicing techniques that make the code better and more reliable, you also achieve to reduce costs!
- NASA C Style Guide
- JAVA Code Conventions by SUN: Most of these conventions can also be applied in C language. You can use it as an example of style conventions.