WHITE PAPER

Software design for self-sustaining embedded systems

Designing and implementing software for embedded systems with limited power resources is a challenging task. This paper focuses on software development problems in systems powered by harvested ambient energy. Software modularity and platform independence are critical design aspects of such systems. The first part of the paper is a short introduction to embedded systems with limited power resources. The second part discusses methods of efficient software development using C language for energy-critical systems.

Attila Strba, Research & Development, EnOcean GmbH

1. INTRODUCTION

Significant technology advances in microelectronics made the increasing miniaturization of embedded systems possible. This trend, which is expected to continue, leads to the development of tiny and smart embedded systems, integrated into more and more everyday objects. Intelligent systems surrounding us can offer a number of new possibilities [1]. Imagine an intelligent home where the heating can communicate with a window handle to find out if the window is open or closed. Or a light that could determine the presence of a person in a room and send this information back to the heating. Such intelligent devices need a wireless interface to transmit information between each other. They also need some source of power enabling them to function. Power transmitted to them through cables or using batteries is not a solution. Devices embedded in the environment have to be self-sustaining. So the only powering possibility is to gather energy from this environment by what is called energy harvesting. Pressing a button with 5 N force, a temperature difference of 5 K or just 400 lux of light generate enough energy to power a module equipped with a microcontroller and an RF transmitter. This energy is sufficient to transmit a wireless signal over a distance of up to 300 meters.

Is this a utopian vision? No, it is reality. Since 2003 there have been products on the market that are powered by energy harvesting. An example of such a product is shown in Figure 1. It is a transmitter device named PTM 200 from EnOcean GmbH. This module enables the implementation of wireless remote control without batteries. Power is provided through a built-in electrodynamic energy converter.

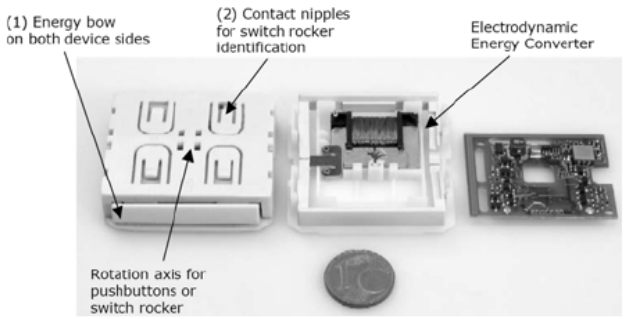

Figure 1 Electrodynamically powered radio transmitter device [3]

A specific state-of-the-art design approach is needed for devices powered by energy generators in both hardware and software development. For example, the mentioned electrodynamic generator

produces an energy impulse for 3 ms. The impulse charges a capacitor, which supplies the module with power for 5 to 10 ms. In this critical, short amount of time the sensor values need to be read, the protocol stack has to be activated and a valid packet with encapsulated information has to be transmitted to the air. Such an example illustrates that developing software for such a system is a real challenge. The remainder of this paper deals with the software design problems of energy-autonomous systems. A specific design approach is needed for embedded systems satisfying the concept of ambient intelligence.

2. ASSEMBLER VERSUS C

When it comes to software development for a system of limited resources where time is a critical constraint, developers would certainly opt for assembler. There is a general belief among embedded software developers that goes: "If you need to write time-critical and space-optimized software code, you should write it in assembler and forget about C compilers". Basically this is true. On the other hand, how many software developers are able to write assembler code that really gets the most out of a system? Writing efficient and optimized C code is not an easy task. It takes several years of coding experience, analyzing different source code until a developer's coding style is efficient enough. It probably takes the same amount of time with assembler. When a project developed in assembler is migrated to a new platform the developer basically has to start gaining experience from the beginning.

That is not the only drawback of using assembler code. Assembler unlike C is not a structured language. It is a symbolic representation of hardware-platform-specific machine code. It is very difficult for one developer to read another's poorly commented assembler code. A number of tools that make software development more efficient, like Doxygen (documentation generator from C source code), Lint (C source code checker) as well as the model-based design based on UML language, cannot be used with assembler. C programming language offers platform independence and flexibility. Developers using C can focus more on the actual problem to be implemented.

3. PLATFORM INDEPENDENCE

Why would software for a tiny, self-powered embedded system need platform independence? Consider a task such as designing software for a heating system control as in Figure 2. The system consists of several wireless, energy-autonomous modules: window handle sensor, heating regulator, occupancy sensor and central device. The heating regulator is mounted on the radiator. It is able to adjust the thermostat valve and has a built in temperature sensor. The module is powered by a thermo converter that harvests energy from the temperature difference between the radiator and its surroundings. The window handle is attached to the window and can detect when it is open and closed. It is powered by the opening and closing motion of the handle using an electrodynamic energy harvester. The occupancy sensor is fitted in the ceiling. It is powered by a solar cell and can detect motion and changes in temperature. The central device is also solar-powered. It is attached to the wall of the room. It collects information from the other modules and sends commands to the heating regulator about valve adjustment. For example, if this central device receives information from the occupancy sensor that there is no longer anybody in the room, it will send a command to the heating regulator to reduce the level of the heating. Or if the central device detects that a window is open, it commands the regulator to turn off the heating.

Figure 2 Heating system control with energy-autonomous devices

Even though we are talking about energy-autonomous devices, the system is complex enough to use high-level software methodology approaches. Different sensor modules are running on different microcontroller platforms adjusted for the energy requirements of the energy harvesters. The common communication protocol requires that each module have the same protocol stack implemented. From this example you see that a platform-independent software language is required for the protocol stack. C is an appropriate multi-platform language for embedded software. The next section of the paper looks at the problem of using C language effectively for energy-autonomous devices.

4. EFFECTIVE USE OF C COMPILER

A significant increase in performance can be achieved by optimizing routines written in C. Several optimization algorithms are implemented in embedded C compilers [3]. Using these, the code of a program can be reduced so that execution of the program will be less, meaning energy saving. Most compiler optimization methods focus on low-level code optimization, i.e. generating the most effective code for the particular hardware platform. Compared to low-level optimization techniques, only a few high-level approaches are proposed that are applicable to generic C source code [2].

Apart from compiler optimization, efficiency of software execution can be achieved by manual code reorganization. This chapter looks at manual source code optimization methods in C. All the methods described in this chapter are based on the fact that C is a very flexible language, and the same routine can be written in many different ways. The following part of the chapter introduces some examples, showing how reorganization of C code can lead to effective assembler results. The assembler code for the examples was generated with the Keil C compiler for the 8051 platform.

Using pointers

Accessing more dimension arrays or complex structures can be very time-consuming, especially when implementing a loop where an operation is necessary for every element of a complex array. Consider the piece of code shown in Figure 3 where multiple elements of multiple arrays are accessed for a checksum calculation. The resulting checksum is written as the last element of the array. The selected array is indexed with the u8b variable, the elements of the array are indexed with u8i.

```
for (u8i=0; u8i < (MAXLENGTH-1); u8i++)
        u8Chk += u8Buff[u8b][u8i];

u8Buff[u8b][u8i] = u8Chk;
```

Figure 3 Loop, indexing elements of an array

If you look at the assembler listing in Figure 4, you see that every time the u8Buff variable is accessed, the array indexing variable u8b is loaded and the absolute address of the actual array item is calculated with a multiplication. After the checksum loop is finished, the u8b variable is loaded again for the checksum saving operation in the array. Calculating the u8b absolute address with the multiplication command is a time-consuming operation. Frequent repetition of this operation in the loop wastes a lot of time. The reason why the compiler generated this code is that it cannot detect that the u8b variable does not change during the loop.

```
?C0007?TEST:   ;checksum calculating loop
MOV      A,u8b ;the u8b is loaded every time
MOV      B,#01EH
MUL      AB    ;the item addr. calculation
ADD      A,#LOW u8Buff
ADD      A,u8i
MOV      R0,A
MOV      A,@R0
ADD      A,u8Chk
MOV      u8Chk,A
INC      u8i
MOV      A,u8i
CJNE     A,#09H,?C0007?TEST

;storing the checksum result to the array
MOV      A,u8b
MOV      B,#0AH
MUL      AB
ADD      A,#LOW u8Buff
ADD      A,u8i
MOV      R0,A
MOV      @R0,u8Chk
```

Figure 4 Assembler listing of loop, indexing elements of an array

Execution of the code above takes 179 µs . By pointers the time to execute the previous code can be reduced as shown in Figure 5.

```
ptr = u8Buff[u8b];
for (u8i=0; u8i < (MAXLENGTH-1); u8i++)
      u8Chk += *(ptr + u8i);

*(ptr + u8i) = u8Chk;
```

Figure 5 Optimized loop execution with pointers

The absolute address of the array *u8Buff* indexed with *u8b* is loaded to a pointer variable called *ptr*. This *ptr* address is incremented with the loop variable *u8i*. The assembler code shows that the *mul* instructions are removed, and the *ptr* address is loaded only once to the *R7* register. Storing the checksum value is also faster because the *ptr* remains in the *R7* register.

```
;loading ptr variable to R7 register
MOV      A,u8b
MOV      B,#01EH
MUL      AB
ADD      A,#LOW u8Buff
MOV      R7,A
;loop label
?C0007?TEST:
MOV      A,R7
ADD      A,u8i
MOV      R0,A
MOV      A,@R0
ADD      A,u8Chk
MOV      u8Chk,A
INC      u8i
MOV      A,u8i
CJNE     A,#01DH,?C0007?TEST
;storing the checksum result to the array
MOV      A,R7
ADD      A,u8i
MOV      R0,A
MOV      @R0,u8Chk
```

Figure 6 Assembler listing of optimized loop execution

The code execution of assembler source in Figure 6 takes 102 µs. With a larger array (tested with 230 elements) the speedup is more significant and several *µs* of execution time can be saved.

It is good practice to put calculations with constants before or after the loop. The following example in Figure 7 shows the increase of each checksum value by 3. Doing it after the loop can save a lot more time even if multiplication is needed.

```
for (u8i=0; u8i < (MAXLENGTH-1); u8i++)
        u8Chk = u8Buff[u8b][u8i] + 3;   //instead here

u8Chk += u8i*3;  //do it rather here
```

Figure 7 Constant calculation after the loop

Structures and more dimensional arrays are very powerful tools in C. But when software is developed for a time- and energy-critical environment the compiler generated code should be checked often to avoid surprises with ineffective code.

Using union with structures

When developing software for embedded systems there is often need to create a 16 bit value from two 8 bit variables. A common way of solving this problem is to use mask operations as shown in the example in Figure 8.

```
#define getLowByte()    0xAA
#define getHighByte()   0xBB

int u16Value = 0;
u16Value = (int)(getHighByte() << 8);
u16Value |= (unsigned char)( getLowByte() );
```

Figure 8 Calculation of 16 bit variables

Shifting is a very time-consuming operation. The code shown in Figure 8 is executed in 30 µs. Using unions the same functionality can be reduced to just 0.75 µs execution time as shown in Figure 9 .

```
typedef union
{
      unsigned char u16Byte;
      struct
      {
            unsigned char u8HighByte;
            unsigned char u8LowByte;
      } r;
} union_t;

union_t u16Value;
u16Value.r.u8HighByte = getHighByte();
u16Value.r.u8LowByte = getLowByte();
```

Figure 9 Calculation of 16 bit variables using unions

The advantage of this solution is that the whole 16 bit variable or just the 8 bit parts can be accessed through the union parameters. The compiled assembler code is very simple as shown in Figure 10.

```
MOV   u16Value,#0BBH
MOV   u16Value+01H,#0AA
```

Figure 10 Assembler listing of calculation of 16 bit variables using unions

Another way to access separate bytes of a 16 bit variable is to use pointers as shown in Figure 11. When using this solution we have to know if the compiler stores variable in Big or Little Endian format.

```
int u16Value = 0;
*(((uint8*)&u16Value)+1) = u8LowByte;
*((uint8*)& u16Value) = u8HighByte;
```

Figure 11 Assembler listing of calculation of 16 bit variables using unions

The power of preprocessor

The preprocessor built into the C compiler processes the source text of a source file before it is actually compiled into machine language and object code [1]. Perhaps the most useful aspect of the C preprocessor is the ability to create and use macros. Assume there is a need to create a function that calculates the timer period based on the oscillator frequency (OF), time slice in milliseconds (u16Msec) and prescaler (u8Presc) settings. An example of such a function is shown in Figure 12.

```
uint16 CalcTPeriode(uint16 u16msec, uint8 u8Presc)
{
   return (uint16)( msec / (float32) ( OF * u8Presc);
}
```

Figure 12 Timer period calculation function

Execution of the function can take several hundred microseconds caused by the 32 bit division with floating point number. The timer period calculation is usually needed only once per software life cycle - only the constant value to be written to the timer register has to be calculated. This is an ideal situation where the preprocessor and macros should be used in the following way as shown in Figure 13.

```
#define CalcTPeriode(u16msec, u8Presc)  \
   ((uint16)( msec / (float32) ( OF * u8Presc));
```

Figure 13 Timer period calculation using macro

The code shown in Figure 13 generates only two assembler instructions. The calculation is done by the preprocessor and the result is a 16 bit constant value. Using a preprocessor and advanced macros can bring modularity to the software without loss of performance.

Switch-case statement

The execution time switch-case statement can vary according to the definition of the case constants. Consider the following switch case statement consisting of at least six arguments shown in Figure 14.

```
switch (u8CurState)
{
   case 4:  u8Res = 'A'; break;
   case 21: u8Res = 'S'; break;
   case 11: u8Res = 'D'; break;
  //… further case elements…
}
```

Figure 14 Switch case statement

The compiler generates assembler code where the switch statement is broken up to test and branch instructions as shown in Figure 15.

```
MOV   A,u8CurState
ADD         A,#0F5H
JZ          ?C0010?TEST ;evaluation of case 11
ADD         A,#0F6H
JZ          ?C0009?TEST ;evaluation of case 21
ADD         A,#011H
JNZ         ?C0011?TEST ;evaluation of case 4
;… further jumps …
```

Figure 15 Assembler listing of switch case statement

Such code is not effective for energy-critical systems because in the worst situation all branch instructions have to be evaluated before jumping to the correct statement code – which with six case statements means 6 x 2 = 12 instructions. Instead, using random case arguments constants 4,21,11… numbers in increasing order should be used (if possible). Assuming the case arguments are 0,1,2,3…, the compiler generates a jump table where the destination address for each case is calculated in six instruction time as shown in Figure 16.

```
MOV   A,u8CurState
CJNE  A,#07H,?C0017?TEST
?C0017?TEST:        ;calculating jump address
JNC   ?C0015?TEST
MOV   DPTR,#03FH
ADD   A,ACC
JMP   @A+DPTR
?C0018?TEST:      ;jump table
AJMP  ?C0008?TEST
AJMP  ?C0009?TEST
….
```

Figure 16 Optimized assembler listing of switch case statement

As you can see from assembler code, such a solution is effective if you use several case arguments. On the other hand, if the number of case arguments is below six, the branch and jump method generated assembler code is faster.

5. SUMMARY

There is rapid growth of new microcontrollers on the market with ultra-low power consumption. To be able to migrate energy-autonomous systems from one microcontroller platform to another, a key requirement is to design and implement the software in platform-independent language. The use of C language for self-sustaining embedded systems can highly improve portability of the code to new platforms. Nevertheless, when writing software for energy-autonomous applications using C language, for optimal performance the compiler-generated assembler needs to be checked.

References

[1] Keil Cx51 Compiler User's Guide

[2] Eui-Young Chung, L. Benini, G. de Micheli: Energy Efficient Source Code Transformation based on Value Profiling
In: Proc. International Workshop on Compilers and Operating Systems for Low Power, 2000

[3] Advanced Compiler Optimization Techniques
Last access 28.11.2008
http://www.embedded.com/192200575?_requestid=279140