<< BACK

Low-level device drivers

Magnus Unemyr
IAR Systems

Class #207

Embedded Systems Conference West 2001
San Francisco

Introduction

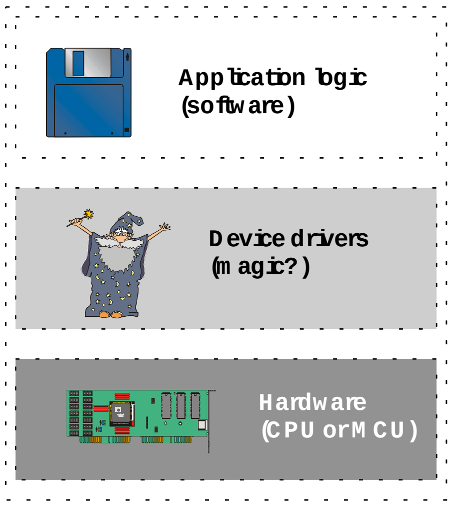
Developing device drivers for a modern, highly integrated microcontroller can be daunting, partly due to the sheer complexity of the device, but also due to some other difficulties.

This paper will give an overview of device driver development for microcontrollers, and will discuss some common problems and solutions to them. Topics include controlling the silicon device, accessing the hardware from the C language as well as interrupt handling with its associated problems.

Device drivers for real time operating systems (RTOS'es) are not covered.

Device drivers

Before going into any technical details, we need to define what a device driver is and what its responsibilities are. The embedded system can be organized in a layered architecture:



The responsibilities of the different layers are explained in the table below:

Application logic	What? When?
Device drivers	How?
Hardware	Do it!

The application software decides what to do, and when. The device drivers know how to do it, and the hardware performs the desired action.

An example might be an application program that decides to write an error message to an LCD display when a counter is decremented to zero. The application calls the device driver function *WriteLCD(char* Message)*, which then instructs the LCD controller to display the message. The LCD display will finally perform the action by displaying the message.

A low-level device driver is normally organized as one or more files that contain a number of functions for controlling the silicon device. Drivers normally contains 3 types of functions:

- Initialization (setup baudrate, timer periods etc)

- Run-time control (send/receive characters, start/stop timers or DMA transfers etc)
- Interrupt handlers (respond to and handle hardware events)

Low-level device drivers normally handle the following in a microcontroller:

- Chip setup (bus controller & interrupt controller)
- Peripheral modules (UART, Timers, DMA, A/D, D/A, LCD, etc.)

Chip setup

When a new electronic board is available, software must be written to handle system start-up. This is usually done by responding to a reset interrupt or jumping to a fixed address. Basic initialization of stack pointer, compiler environment and bus controller settings are done during this phase. The CPU will most often not interface to external hardware unless some low-level configurations are made:

- Bus interface (address- and data buses, chip select signals)
- Memory configuration (DRAM refresh, wait-states, handshaking)
- Interrupt system (IRQs, interrupt priorities and masks)

Once low-level configuration has been performed, execution normally continues in the application program's main() function. At this point, the application and peripheral module device drivers can start execution.

Peripheral modules

Device drivers provide a software interface for accessing hardware from software. The aim is to write a driver library that can be used by the application program to access hardware services from peripheral modules (UARTs, Timers, A/D or D/A converters, CAN or DMA controllers, etc.).

Driver logic is implemented by modifying or testing special function register (SFR) control and status bits in a suitable order. A modern high-end microcontroller can have several thousand SFR bits, each of which must be carefully initialized and manipulated in proper sequence at run-time.

Accessing the hardware

In order to control the silicon device, we somehow need to talk to it. This is done by reading and/or writing bits from SFR registers. The hardware manual defines the memory addresses for certain SFR registers, and the meaning of the bits in the register.

In some hypothetical microcontroller device, the 8-bit register UCR (UART Control Register) is located at the address 0x00FF4022 in memory. Furthermore, assume that bit 3 is a PARE bit (Parity Error), and that bit 4 is a DBIT bit (Data bits).

0x00FF4022: UART Control Register (UCR)

Unused	Unused	Unused	DBIT 0=7 data bits 1=8 data bits	PARE 0=No error 1=Parity error	Unused	Unused	Unused
Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0

By reading bit 3 in UCR, the device driver can determine if there is a parity error detected in the UART (PARE = 0 for no parity error, PARE = 1 for parity error).

By writing to bit 4 in UCR, the device driver can configure the number of data bits to use during serial communications (DBIT = 0 for 7 data bits, DBIT = 1 for 8 data bits).

It might not be very easy to understand how the bits should be manipulated to achieve the desired operation, as (complex) protocols often define the sequence and timing of bit access. In particular, the driver may need to follow strict timing requirements.

This might be achieved by waiting for a status bit to be updated, but might also have to be implemented by software delays. Do not use empty loops for delays, as optimizing compilers remove code with no functional side effects.

A typical way of "overloading" an SFR variable at a specific memory address is the following code sample. Once the SFR is defined as the value at the desired address, the SFR can be accessed much like any other variable:

```
#define UCSR (*(volatile unsigned char *)(0x00FF4024))
…
UCSR = 0xFF;
```

Note that we must define SFR variables as *volatile*, as the value can be changed in the background by the hardware. Without the *volatile* keyword, the compiler might optimize the code by caching the value in CPU registers, and erroneous operation would be the result if the hardware updates the value between read accesses. For writes, *volatile* ensures that all changes are written to the actual hardware and not just to a temporary register.

Once we can access the SFR register, we need a way of accessing individual SFR bits. This is commonly performed by defining bit masks that represent the fields in the SFR, and by masking bits using logical OR and logical AND.

```
/* Define bits */
#define ERROR       0x08        /* bit 3 - error flag */
#define SENDBYTE    0x20        /* bit 5 - send data byte */

/* Clear bit (error flag) */
UCSR &= ~ERROR;                 /* sfr = sfr AND NOT(bit): Clear error flag */

/* Set bit (send byte) */
UCSR |= SENDBYTE;               /* sfr = sfr OR bit: Set send data byte flag */

/* Query bits */
while( (UCSR & ERROR) != 0 )    /* Wait for error release */
        ;
```

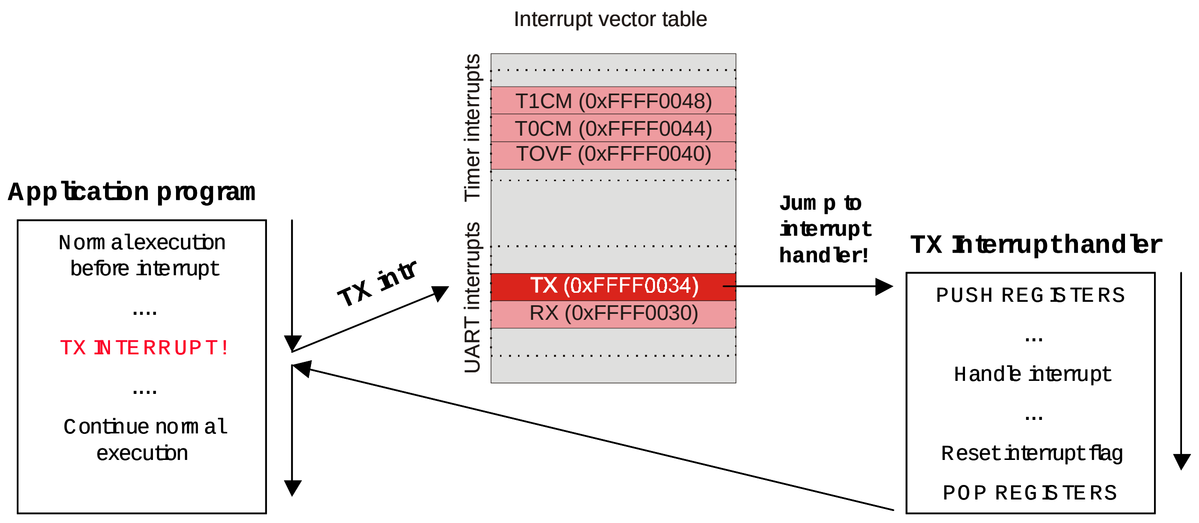
A fairly common way of accessing SFR bits is using the bitfield syntax available in the C language. They provide a nice syntax for bit manipulation, but the bit order is undefined according to the ANSI C standard, effectively making this solution compiler dependent and thus non-portable.

Interrupts

The device drivers need to be informed about hardware events that may occur at any (unknown) time, such as the arrival of a new character on a UART channel, or an ended DMA transfer.

The hardware detects such events, interrupts execution of the application program, and uses the interrupt vector table to lookup the address of the interrupt handler function that should take care of the event. The interrupt handler is executed, and the application program is then resumed.

The interrupt handler must be connected at the appropriate entry in the interrupt vector table, which in turn must be linked to the proper memory address (defined in the microcontroller hardware manual).



Interrupt handler functions require a somewhat modified assembler code implementation (using the “return from interrupt” instruction rather than the “return from subroutine” instruction) compared to normal functions. Most embedded compilers have specific (non-portable) keywords or #pragmas for defining interrupt handlers that is implemented using the “return from interrupt” instruction.

Once the interrupt handler is hooked into the interrupt vector table, we need to write code that handles the event and responds in an intelligent way. During this process, we might introduce errors that appear to occur at random times during run-time.

The reason for this is that the interrupt handler might interrupt the application and start executing at any time, and if we do not take careful precautions, we will end up with some very hard-to-find bugs indeed.

The most common errors are introduced by:

- Reentrancy problems
- Race conditions

- Register caching problems

Reentrancy

A function is reentrant if it can be interrupted and another copy of the same function can be executed from the interrupt handler without unexpected results. In other words, an interrupt handler can safely break the execution of the function and start executing a second copy of the same function. A non-reentrant function must never be interrupted by code that starts execution of another instance of the function.

A function that only uses local variables and no global resources is normally reentrant (but be careful with compilers that use “static overlay” memory allocation algorithms). But if the function uses global resources (such as file handles, global variables etc), the function may become non-reentrant and we need to protect the execution of the function.

This is done by making the function calls in the application program a “critical section”, which guarantees that only one instance of the function can execute at the same time. The critical section will typically disable interrupts before the call, safely perform the call, and then enable the interrupts again. The code inside the critical section becomes “unbreakable” and cannot be interrupted by an interrupt handler. Some embedded compilers have explicit support for making functions that should be run without interrupts enabled.

The granularity of the critical sections can be controlled using either global or local critical sections.

- Global critical sections switch off all interrupts in the processor, thus giving very safe critical sections as no interrupts of any type can occur. The disadvantage is that also unrelated (safe) interrupts are disabled, thus introducing a performance penalty by reducing interrupt response times.

- Local critical sections work on a more detailed level, as only the interrupt(s) that causes the reentrancy problem is disabled. Local critical sections do not suspend unrelated interrupts, whereby improving overall system performance.

Race conditions

Race conditions is similar to the reentrancy problem described above, but applies to global variables instead of function calls. The problem is that C statements are not atomic; a computation involving a certain variable might be interrupted by an interrupt handler that alters the variable at the same time. The result is random calculation errors that are very difficult to find.

The solution to this problem is making all modifications of the variable atomic in the application program. This is done by inserting “critical sections” around the variable calculations in the application program, usually by switching off interrupts during computation. Since no interrupts can occur during computation, the interrupt handler cannot tamper with the variable at unsafe times.

Register caching

Problems occur when optimizing compilers reduce memory accesses by caching variable values in CPU registers. If the application program makes two accesses to a variable, and an interrupt comes between and modifies the value, the second access in the application program will not use the updated value, as it uses the old cached value in the register instead of the new value in memory.

This problem is handled by declaring variables that can be accessed both from the application program and the interrupt handler as “volatile”, in which case the compiler are not allowed to optimize memory accesses for such variables.

Special considerations

The hardware may impose specific restrictions that may cause problems when writing low-level device drivers using the C language.

In some cases, the microcontroller requires certain assembler instructions to be used for accessing some features, such as using the BSET (bit set) instruction rather than logical OR instruction for setting bits in some registers. Whether or not such accesses can be performed from a C program or not is compiler dependent, and thus not portable. You may have to write assembler code to work around the compiler’s code generation algorithm in such cases.

In other cases, pointers must be of specific (and un-natural) sizes. An architecture that uses 32-bit pointers may require 16-bit pointers in certain situations. This have to be solved using compiler specific #pragmas or using assembler code.

Debugging

Debugging can prove to be particularly tricky, as the drivers may not work without external hardware, and may need to conform to timing constraints or other resource limitations. They may need to run at full speed and can sometimes not be single stepped in a debugger. Interrupt handlers can be very hard to debug due to timing or triggering requirements.

In practice, a fairly complete hardware environment with a debug monitor or emulator is needed, as are additional laboratory instruments such as an oscilloscope or a logic analyzer. A debug simulator (running on the host PC) can seldom be used.

A good way of getting familiar with the device behavior before writing the device driver source code is to manually read/write to the SFR registers of the device using the memory commands in a debug monitor or emulator. In many cases, you can initiate many advanced hardware features by this simple method.

Where are the tools?

To summarize traditional device driver development, the following tasks have to be done more or less manually:

- Reading the hardware manual and learning the chip internals
- Understanding the electronic board design
- Designing and implementing the device driver library
- Debugging the device driver library

- Test and integration

Using modern device driver development tools now available from several tool vendors, the developer can configure the microcontroller using graphical dialog boxes, and let the tools generate highly target-specific device driver source code automatically.

While you still have to know what features you want to use in the microcontroller, you do not have to know exactly how to implement those features, as the source code is generated automatically from the device driver development tool.

Conclusion

Writing device drivers is often considered a rather tedious task that requires detailed knowledge of the silicon device. While this is true, device driver development provides some very interesting programming challenges to the interested programmer.

With some basic understanding of the hardware device, the C language and your compiler, you can quite easily make the device perform advanced tasks.

On the other hand, without a good understanding of low-level programming from the C language, you may end up with a mess of random errors, hardware that doesn’t seem to respond to communications and unreliable code.

Using a new breed of device driver development tools, development efficiency and code quality can be enhanced.