Embedded Systems Essentials with Arm: Getting Started

Module 2

KV3 (2): High-level and low-level programming

High-level programming hides some of the details of the target platform from the developer via a level of abstraction. This often includes automatic memory management, which allows the developer to focus strictly on implementing the desired functionality.

The benefits of high-level programming include shorter development time, portability across devices, code that is easier to read and maintain, code that is reusable, and rapid prototyping of applications.

Drawbacks include less optimized code, additional translation time for source-to-machine code, and an additional level of abstraction.

Low-level programming typically involves coding with a limited instruction set that maps closely to the processor instructions. A compiler may not be required to convert the code into machine language.

The benefits of low-level programming are better optimized code, increased memory efficiency, less translation time for source-to-machine code, and the ability for the program to 'talk' directly to the hardware.

Drawbacks are that code is less portable from one device to another, that the resulting code is harder for others to read, reuse, and maintain, and that code may take longer to develop.

Let's look at an example of low-level programming: blinking an LED with MCU Registers.

Obviously, assembly language is typically what we associate with low-level programming, in which the code corresponds more directly with the underlying hardware. But this code is the closest approach to assembly level within C or C++: everything is coded with addresses, pointers, and registers. Note that the first line, including the mbed.h, can be ignored. In this example it is used for readability. This code encapsulates the necessary port initialization, so it can focus on the 'blinky'-function as the main routine.

In line 7, pin 18 is coded in a variable by left-shifting a 1 eighteen times. This sets the bit 18 of mask_pin18.

Lines 9 and 10 define the addresses the ports   that can be set and reset. In this example, it's port 1.

In lines 13 and 16, the pin 18 at port 1 is toggled by setting and resetting it respectively.

The code is short and efficient, but is nearly unreadable if you don't know the purpose of the addresses in lines 9 and 10, and if the variable names are unclear.

Now let's look at an example of high-level programming: blinking an LED with an Mbed API.

Here you can see an example of its code implementation with the full C++ based abstraction, including operator functions.

With the support of the Mbed API, the same 'blinky' example can be programmed in a much simpler and more intuitive way: the declaration 'led1' constructs and assigns itself to the 'LED1' part of the evaluation board, with the correct ports and addresses. The assignment of 1 in line 7, and of 0 in line 10, lets the program toggle the LED on and off by writing the corresponding values to the port.

In this code, the use of addresses is completely encapsulated and invisible.

In this example, the code is written using the Mbed API. Note that the Mbed API is programmed using the object-oriented language C++. C++ originated from C language, which uses object-oriented features such as classes. Deep knowledge of C++ is not required to use the Mbed API, but there are many tutorials and books available to help you learn it.