IoT Device Programming and Debugging

Tom Spink
Programming

Programming is the act of uploading code and data to an embedded device.

In general, it encompasses writing data into EEPROM and usually the configuration of the target device, via device configuration fuses.

Each microcontroller has its own method of programming, but often a bootloader will be used to enable a more standard or convenient programming method.
Programming

Configuration fuses are a non-volatile memory used to configure the fundamental behaviour of a microcontroller, e.g. the clock source, clock rate, watchdog timer, brown-out detect, etc.
Bootloader

A bootloader is a special (tiny) application that prepares a microcontroller for execution of the main program, by initialising memory and peripherals.

It is the first piece of code executed when the microcontroller comes out of reset.

It can also act as an interface to a more convenient programming method, i.e. expose functionality to enable writing to program and data memory through other means.
Debugging

Debugging an embedded system can be quite a complicated (and expensive!) task, in comparison to debugging a software application.

It’s an invaluable tool during development, as there may be a lot of peripherals to consider.

Ideally, debugging should be rapid so that you don’t have to wait for results.
Debugging

Using an IDE is the ideal approach for debugging an embedded application, as it should contain all the tools and functionality for inspecting the behaviour of the device.

In-circuit debugging allows you to debug the operation of a device, whilst in place on a circuit board - but the microcontroller must support this!

Simulation allows you to run a virtual version of a device, without having to use physical hardware.
Application-specific Debugging

Some applications may implement their own proprietary debugging routines, that are highly application-specific.

For example: implementing a command-line interface that can interrogate device memory.

But! Remember to disable this for production, or you may open up security holes!

Production of a device may leave debug headers available for factory programming, testing and QA, but they should be disabled afterwards.
Simulation

Simulation of a hardware platform is very useful, as you get to work with a virtual version of device that might not yet physically exist.

There is no need to deploy the application to hardware, which takes time, but also may degrade flash memory.

Simulation lets you work on the hardware design in parallel to software, or implement hardware/software co-design. Levels of simulation give different insights into the behaviour of the device:

- Functional Simulation       Fast
- Cycle-accurate Simulation  Slow
- RTL simulation              Very slow
Integrated Development Environments

IDEs are essential for working on large-scale embedded applications.

They can be used for the development of application software and the management of frameworks/libraries.

Most have the facility to program the target device, and some have the facility for in-circuit debugging.

Generic IDEs and manufacturer IDEs exist, with varying levels of support.
Application Frameworks

Frameworks introduce a level of maintainability to an application. By re-using common code, it reduces maintenance burden, improves code quality, and you benefit from (possibly) more scrutinised and tuned implementations.

Drawbacks are security issues, which when discovered in a framework will impact all devices using that framework, and obsolescence when a developer stops supporting it.

Also, performance can be an issue, if core functionality is implemented as function calls, e.g. a HAL which makes calls to toggle GPIO (Arduino’s `digitalWrite()`).
Standard Interfaces

Serial interfaces are typically found on Microcontrollers, e.g. UARTs. These can be used for application-specific communication, debugging or other purposes.

FTDI design and manufacture ICs for bridging a USB serial-port to UART, an extremely straightforward way to get USB connectivity to a device.

JTAG - Joint Test Action Group - is an industry standard for testing and debugging circuits after manufacture. It specifies a serial debug port that can be used to interrogate various chips, and even upload data to flash memory.
Joint Test Action Group (JTAG)

A JTAG interface is a standardised interface for programming and debugging. It connects between multiple chips on a bus, and allows commands and data to be sent to any chip on the bus.

Boundary scanning enables a direct view of the input/output pins on a chip, and thus interconnections between chips (that are difficult to probe) can be tested by interrogating the state of chips’ external pins.
JTAG

The JTAG state machine specifies how a device should interpret JTAG communications.

Holding TMS high for at most 5 TCK cycles will guarantee returning to the test logic reset state. (TRST is an optional signal)
Over-the-air Programming & Debugging

Over-the-air (OTA) programming can be convenient for rapid prototyping and testing of devices before or after deployment.

However, there is a danger of bricking the device, and if the device is physically inaccessible, then it is lost.

If programming is interrupted half-way through, or bad data is transmitted or received, it could cause undefined behaviour.

By assuming the worst (defensive programming), the bootloader/programmer should be developed to recover from an interruption.
Over-the-air Programming & Debugging

An on-chip bootloader may implement functionality to handle firmware updates from existing communication devices, enabling existing device infrastructure to be re-used.

The drawback is that if the update fails, and the bootloader becomes corrupt, the device is bricked.

A dedicated off-chip programmer may implement the required protocol for OTA firmware updating, being completely separate to the actual application.

The drawback is that this increases device size, complexity and cost.