Extreme Asynchronous IoT Firmware Design

By Matthew Eshleman

This article was originally published in 2016 at embedded.com. The formatting of the original article has degraded, so I am re-offering this article as a downloadable PDF. Feel free to share, but please do not modify.

Abstract

Many Internet of Things (IoT) devices incorporate cellular modules to enable the device's internet access. Incorporating a self-contained cellular module into an IoT device's design has many advantages, yet the complexity and associated communication delays of the cellular network remain. If this inherent complexity is not properly addressed by the device's microcontroller firmware, then the end user may be exposed to erratic behavior induced by cellular network latency. Even if the device does not require user interactions, ignoring the impact of the cellular network may induce timing jitter in recorded sensor readings with the potential of invalidating the usefulness of the data. To fully solve these challenges, we must embrace "Extreme Asynchronous" event driven firmware design. This article was inspired by legacy code improvement projects where the author was requested to improve IoT devices exhibiting faults induced by firmware exposed to the delays inherit in the cellular system.

The Hardware

IoT devices incorporating a cellular module often follow a design pattern as shown in Figure 1.

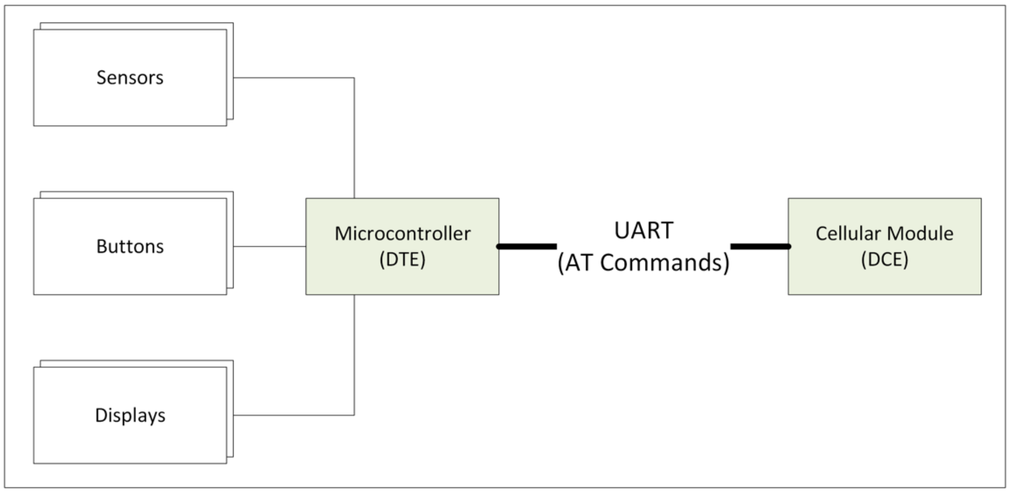

Figure 1: Generic IoT Device Hardware Diagram

The focus of this article is the UART interface between the microcontroller, acting as the Data Terminal Equipment (DTE) and the cellular module as the Data Communications Equipment (DCE). This serial interface is often controlled via “AT” style commands, a long lived legacy from Hayes modems and the 1980s. In the context of a cellular modem, some AT commands may incur substantial delays in delivering a response. These delays are often the root of erratic behavior in an IoT device.

The Cellular Network

There is no doubt that the modern cellular network is an amazing engineering and digital communications achievement. However, to better understand the core issue discussed in this article, a brief examination of some of the challenges involved with the cellular network is required. A device using the cell network for data communications must contend with:

- Device Registration
    - The device must be registered with the appropriate carrier. When activating, the device must contact the assigned carrier endpoint to join the network and receive an IP address.
    - Joining the network and receiving an IP address may take substantial time.
- Location and Movement
    - Whether the device is designed to be mobile or is installed in a fixed location, there is no guarantee that the location enjoys a solid data connection.
    - A fixed location device may still be exposed to variations in cellular performance as objects in the vicinity move, trees grow, or buildings are erected.
    - Variations in cell signal quality will cause variations in response times.
- Connection quality

- As other cellular devices traverse the area, the device may be exposed to varying levels of data connection quality as the active cell tower adjusts for varying loads.
- Changes in data rates or quality also impact response times.

The challenges listed above are just a few of the core issues faced by IoT firmware designers making use of a cellular network.

The Cellular Modems

There are many suppliers of embedded cellular modules with each providing various command interfaces allowing an external microcontroller or other device to control the modem. A common messaging and control interface is the "AT Commands" interface often delivered via a standard UART serial port or other serial data connection.

Most cellular modules adhere to the industry specification 3GPP 27.007 [3] which provides for a common "AT command set for User Equipment (UE)." This specification provides a base set of common AT commands and behavior among compliant cellular modules. Additionally, all cellular modules implement manufacturer specific proprietary commands to support their specific value-added features such as module assisted HTTP queries, FTP transfers, TLS support, and other commands as deemed necessary by the manufacturer and their target market.

It is fundamental to the design challenges called out in this article to emphasize that the AT interface is designed as a Command-Response style interface. The host controller (DTE) issues an AT command to the module and then waits for a response. No other commands are allowed until the cellular module responds. This restriction is verified with the following excerpt from [2]:

> "The chain Command -> Response shall always be respected and a new command must not be issued before the module has terminated all the sending of its response result code." – [2] *Section 3.2.5 – Command Issuing Timing*

This is not specific to the quoted manufacturer it is a core behavior among all cellular module AT command interfaces. Given that the DTE must wait for a response, how long might that wait be? Various datasheets again point us to an answer as shown in Table 1 where a small sample of commands and their specified max response delays are listed.

Source	Command	Description	Estimated Maximum Response Time (**seconds**)
[2] – Section 3.2.2	+COPS	Operator Selection	30
[2] – Section 3.2.2	+CGACT	Context Activation	150
[1] – Section B.4	+UPSDA	Context Activation	150
[1] – Section B.4	+USOCO	Connect Socket	20

Table 1: Sampling of AT Commands and Possible Response Delays

Yes, some of the commands in the above table may take **minutes** to respond. Not microseconds. Not milliseconds. Not seconds. Minutes. IoT system and firmware designers must not ignore this information in their module's datasheet. The author personally investigated a report involving legacy code where the device failed to respond to user interactions for over 2 minutes. This is the core reason for this article.

Firmware Architecture

Now that we are aware of the challenges involved with the AT command-response interface we can review various firmware architectures to determine which design is appropriate for various IoT device requirements.

Synchronous Messaging architecture

The Synchronous Messaging Architecture "hides" the asynchronous nature of the command interface and associated cellular network. In this architecture there is usually a module that provides APIs for sending and then other methods for waiting on an appropriate response. In our sample diagram shown in Figure 2, this is the "CELL" module. The user of this module is able to write code in a synchronous manner, which is natural and often times easier to understand and review.

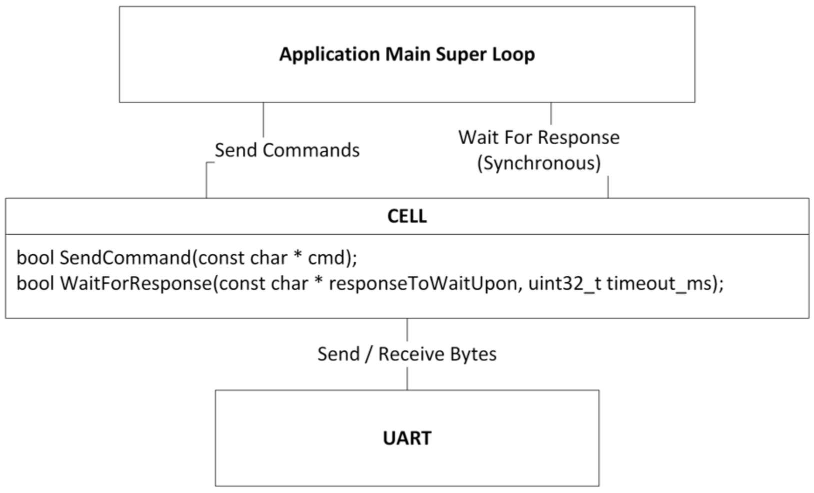
Application Main Super Loop
Send Commands
Wait For Response (Synchronous)
CELL
bool SendCommand(const char * cmd);
bool WaitForResponse(const char * responseToWaitUpon, uint32_t timeout_ms);
Send / Receive Bytes
UART

Figure 2: Synchronous Messaging Architecture

Legacy projects encountered by the author used this architecture and all suffered from erratic behavior induced by cellular network delays. How might this architecture appear in code? The code excerpt below is extracted from the publicly available Adafruit FONA class [4]. This class would be a design equivalent to the "CELL" module in Figure 2.

```
uint16_t Adafruit_FONA::TCPread(uint8_t *buff, uint8_t len) {
  uint16_t avail;
  mySerial->print(F("AT+CIPRXGET=2,"));
  mySerial->println(len);
  readline(); // BLOCKS UNTIL RESPONSE RECEIVED!
  if (! parseReply(F("+CIPRXGET: 2,"), &avail, ',', 0)) return false;

  readRaw(avail);
  memcpy(buff, replybuffer, avail);
  return avail;
}
```

The code listed above first sends an AT command and then waits for a response which is parsed appropriately for the particular AT command. The internals of `readline()` (equivalent to the "`CELL_WaitForResponse()`" API listed in Figure 2) are not critical to this investigation, suffice it to say the caller is blocked until a response message is correctly received or a timeout takes place. If the calling context is the main super loop, then all other activities directed by the super loop, such as servicing a user interface or reading sensor data, are also blocked. In locations where the cellular network is stable and responsive, the device will appear to be responsive and in good order, yet in areas of poor cellular coverage, the device may behave erratically and fail to meet its product requirements.

This architecture is simple, clean, and easy to use and may meet the needs of IoT projects lacking user interfaces or other time critical events managed by the super loop. If the firmware engineer selects this architecture for their IoT firmware, it is also critical that sensor data be collected appropriately in prioritized interrupt service routines. If the main loop also acquires the device's sensor data, then such data may be gathered erratically, potentially invalidating certain algorithms for later offline data analysis. For example, FFT or DFT analysis would suffer if the data is not acquired with a consistent regular sample rate.

Extreme Asynchronous Architecture

The Extreme Asynchronous Architecture is an event driven architecture where all application behavioral requirements, data acquisition, and user interface handling requirements are carefully separated from cellular AT command handling and processing. This architecture embraces what will be called "Extreme Asynchronous" design patterns as inspired by Samek in [5], where "event driven" and "run-to-completion" firmware behavior is required for a responsive embedded system. In the context of our IoT device, all software servicing the sensors, data acquisition, the user interface hardware, and the cellular module itself must be designed to be fully event driven and asynchronous. It is "extreme" because the response times are now potentially measured in minutes, not just seconds or milliseconds. Figure 3 provides an example illustration of this firmware architecture.

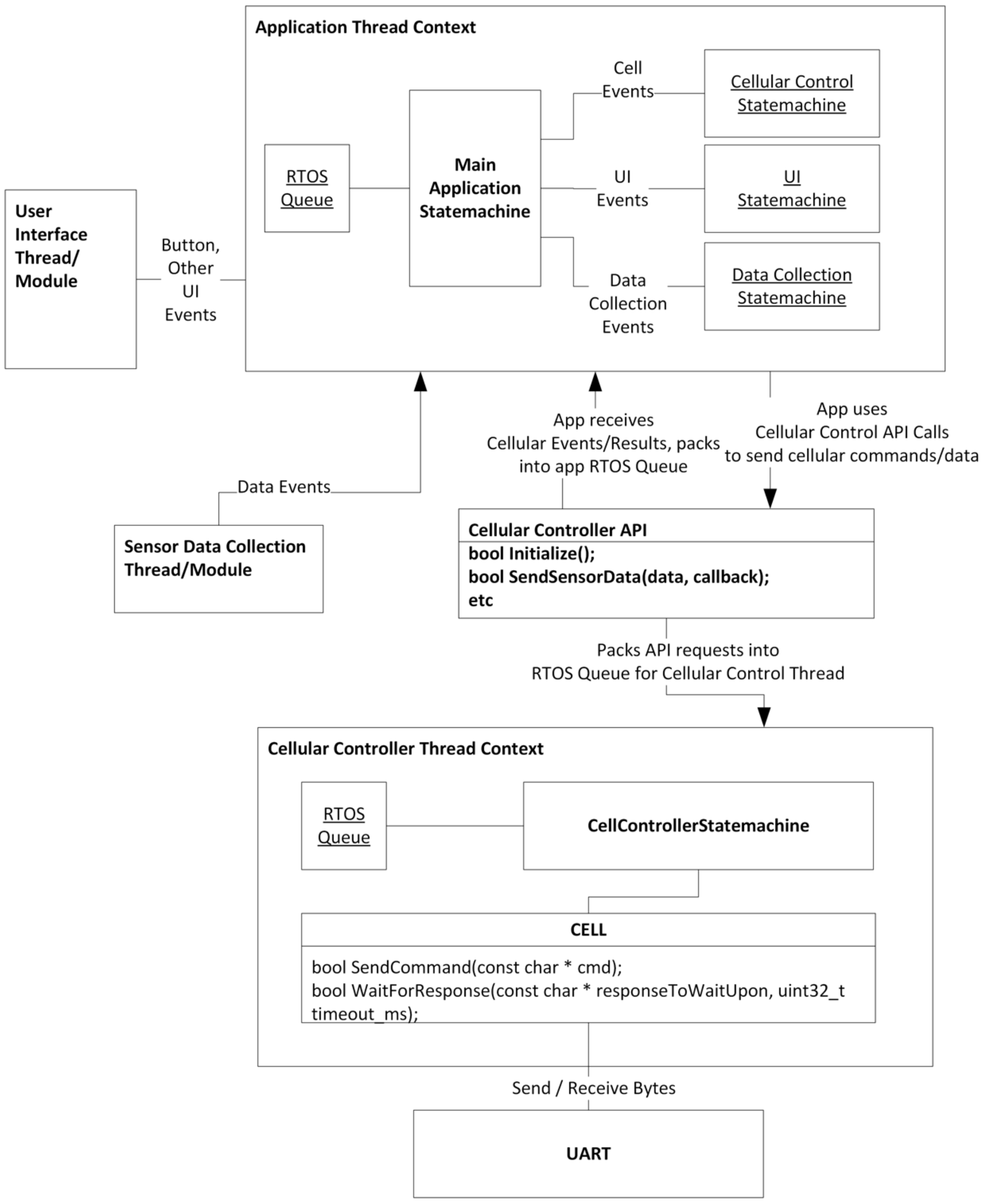
Application Thread Context
RTOS Queue
Main Application Statemachine
Cell Events
Cellular Control Statemachine
UI Events
UI Statemachine
Data Collection Events
Data Collection Statemachine
User Interface Thread/ Module
Button, Other UI Events
Data Events
Sensor Data Collection Thread/Module
App receives Cellular Events/Results, packs into app RTOS Queue
App uses Cellular Control API Calls to send cellular commands/data
Cellular Controller API
bool Initialize();
bool SendSensorData(data, callback);
etc
Packs API requests into RTOS Queue for Cellular Control Thread
Cellular Controller Thread Context
RTOS Queue
CellControllerStatemachine
CELL
bool SendCommand(const char * cmd);
bool WaitForResponse(const char * responseToWaitUpon, uint32_t timeout_ms);
Send / Receive Bytes
UART

Figure 3: Extreme Asynchronous Sample Architecture

There are a few key areas to examine in this architecture. The application thread as shown in the diagram is managing multiple parallel state machines. This is a design decision, there is nothing preventing the firmware designer from combining all application level device behavior into a single application state machine. However, the design would need to carefully consider the “extreme” cell response delays called out in this article. Parallel simple state machines or a single carefully designed hierarchical UML state chart approach are recommended for the main application thread state machine as being best suited to the highly asynchronous nature of the cellular system. Additionally, note that application events are delivered through a single application queue where queued events are processed only in the application thread context. This is consistent with the recommendations in [5] and active object design patterns.

This architecture does take a short cut with respect to the Cell Controller State Machine. This design assumes legacy code with an existing proven and tested “CELL” module, with the same behavioral pattern as discussed in the Synchronous Messaging Architecture. In this design the new Cell Controller module/thread provides a fully asynchronous API with application specific APIs and corresponding events. However, the internal Cell Controller State Machine is now violating the “run to completion” requirements of a well behaved active object. There is some risk in this approach, but ultimately this module is reflecting the design restrictions of the AT Command interface itself. If strict active object run-to-completion semantics are required, then the “CELL” and possibly the “UART” modules would need additional design changes to enable cellular message event handling. Regardless of which design approach is selected, the system must adhere to the core command-response requirement of the cellular module’s AT command interface.

Additional Cell Controller requirements not discussed in detail or shown in the diagram might include:

- Ongoing cell signal and network status monitoring
- Cell module unsolicited response code handling
- Error handling if the application requests cell commands while the Cell Controller is already busy/waiting on the previous application request

Other Design challenges

Although the “extreme asynchronous” cellular network response time is the primary focus of this article, it is not the only challenge an IoT firmware engineer will face. Some IoT devices may be required to handle SMS messages simultaneously with IP socket data connections or other cell network enabled features. Many IoT cell modules support a multiplexed AT Command interface, the so-called CMUX option as defined in [6]. When using the CMUX multiplexing layer, multiple AT Command “instances” are activated, allowing some level of parallel AT Command behavior. This multiplexed approach requires a more complicated data layer, but might be beneficial to certain IoT devices to further improve overall device responsiveness. Carefully check the target cell module’s datasheet for module specific restrictions when using the CMUX option. The firmware architecture design would also need to be expanded and enhanced to take full advantage of the additional flexibility when the CMUX option is enabled.

Not every AT command response will consist of simple ASCII strings or respond with a consistent data structure. Confirm the cell module’s response formats in the device’s datasheet and realize in advance that the cell module is a large and complicated system with its own shortfalls. The datasheet will not always match actual behavior and the firmware designer must be prepared to handle parsing many response formats. Test carefully, capture responses, and seriously consider a Test Driven Development environment to enable ongoing regression testing of any message parsing code.

Security is always a concern for IoT devices, especially given recent headlines regarding hacked IoT devices used to create large scale denial-of-service attacks. The project's system, hardware, and software engineers must consider the security of the firmware, the firmware update system, and even the AT Commands exchanged between the internal microcontroller and the cell module itself. Those AT commands will be in the clear and easily captured during a physical reverse engineering attack. Therefore engineers must carefully consider certificates or keys that might be exposed via the UART interface. Some cellular modules allow certificates or keys to be stored inside the cellular module's internal flash, avoiding repeated exposure via the AT command interface. But what happens if the cellular module itself has a bug and later corrupts the certificate? Consider error handling, security, and privacy concerns carefully during the device's early design stages.

Summary

This article has enumerated two possible IoT firmware architecture designs for handling the "extreme asynchronous" nature of the cellular network. More designs are certainly possible and the author can imagine achieving the asynchronous design pattern in a bare-metal no-RTOS environment with appropriate use of interrupt service routines and ISR safe event queues feeding events to a non-blocking main super loop.

With cellular command response delays that are sometimes measured as high as one to two minutes and often times in the five to twenty second range, firmware engineers tackling this domain must always be ready for "extreme asynchronous" behavior!

References

[1] u-blox Cellular Modules – AT Commands Manual - https://www.u-blox.com/sites/default/files/u-blox-ATCommands_Manual_%28UBX-13002752%29.pdf

[2] Telit AT Commands Reference Guide - http://www.telit.com/fileadmin/user_upload/products/Downloads/2G/Telit_AT_Commands_Reference_Guide_r23.pdf

[3] 3GPP TS 27.007 - AT command set for User Equipment (UE) - https://portal.3gpp.org/desktopmodules/Specifications/SpecificationDetails.aspx?specificationId=1515

[4] Adafruit FONA Software - https://github.com/adafruit/Adafruit_FONA

[5] Samek, Miro (2002). Practical Statecharts in C/C++ - Quantum Programming for Embedded Systems

[6] 3G TS 27.010 - Terminal Equipment to Mobile Station (TE-MS) multiplexer protocol - http://www.3gpp.org/ftp/tsg_t/tsg_t/tsgt_04/docs/pdfs/TP-99119.pdf