AT-based Modem Support in the Zephyr™ Project: Present and Future

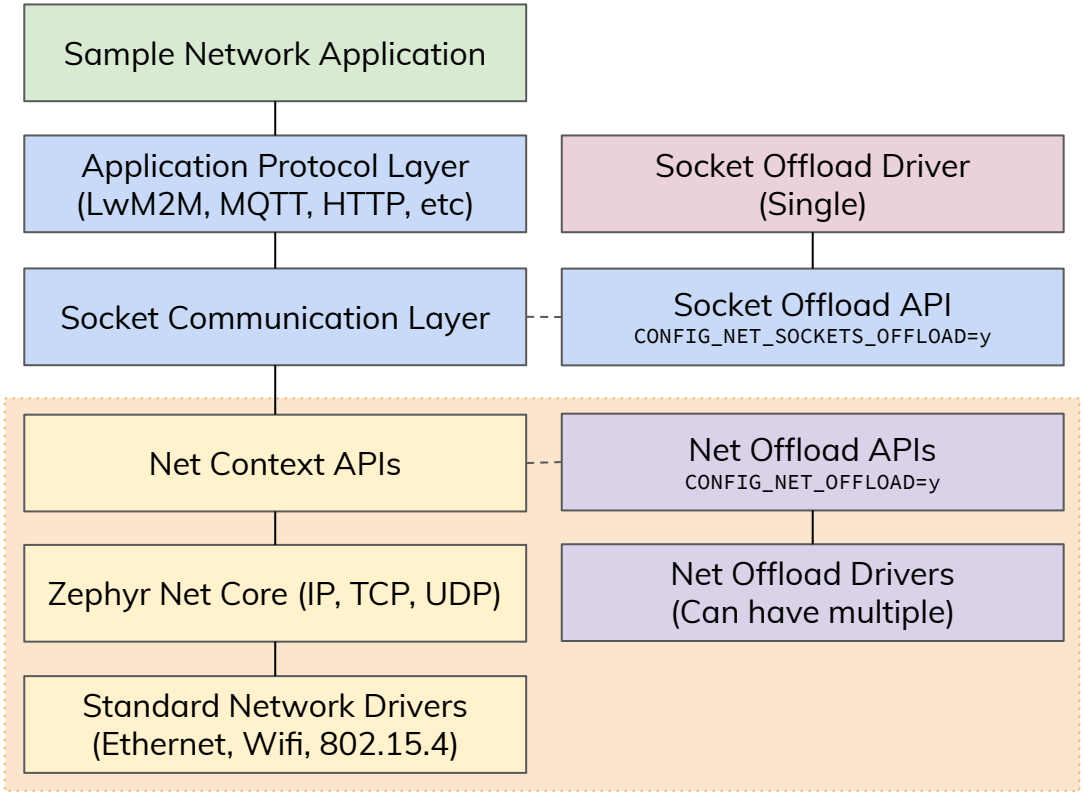
Michael Scott
mike@foundries.io

What can you expect during this presentation

- Brief description of the Zephyr™ networking stack and the offloaded options which allow modem drivers to intercept network handling
- Overview of the existing modem driver implementation model for v1.14
- New modem driver implementation model for v2.0
- “Quick” walk-through of how to implement your own modem driver
- Modems that are currently supported by mainline or Zephyr™-based SDKs
- Modems in products supported privately
- What’s next?
- Questions

Before we begin … Storytime!

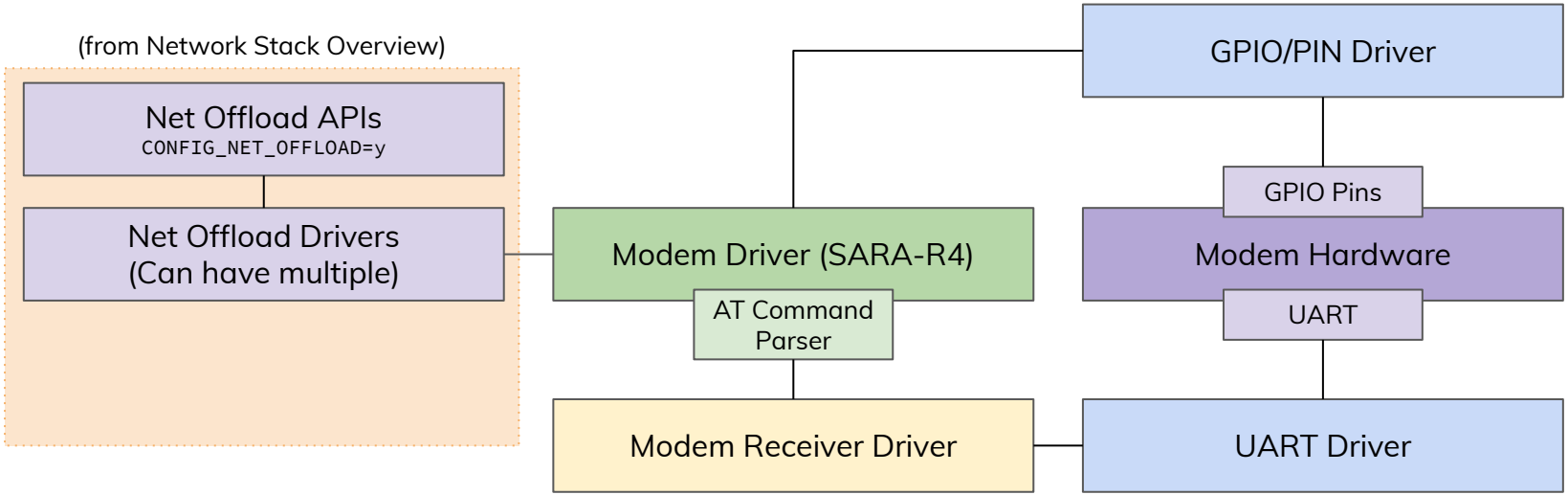
Zephyr™ RTOS Basic Network Stack Overview



Modem Driver Implementation Models

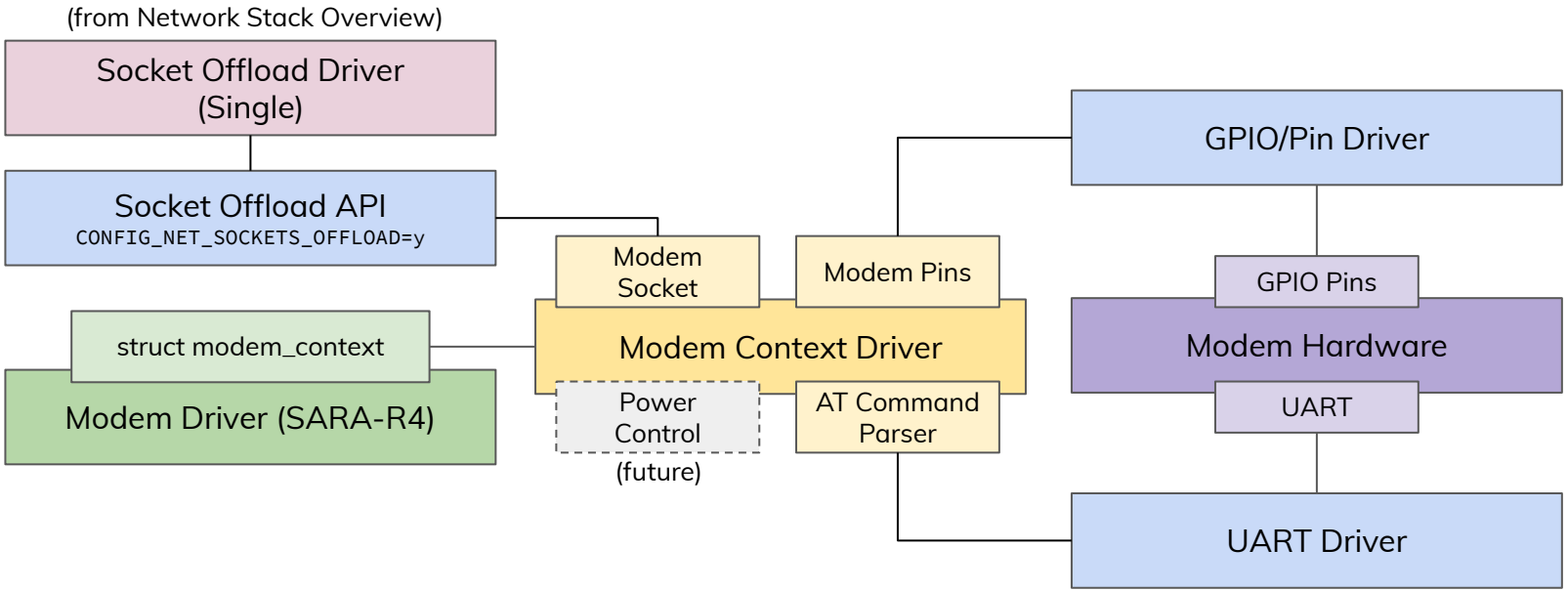
Modem Driver Implementation Model v1.14

Modem drivers have lots of duplicated code



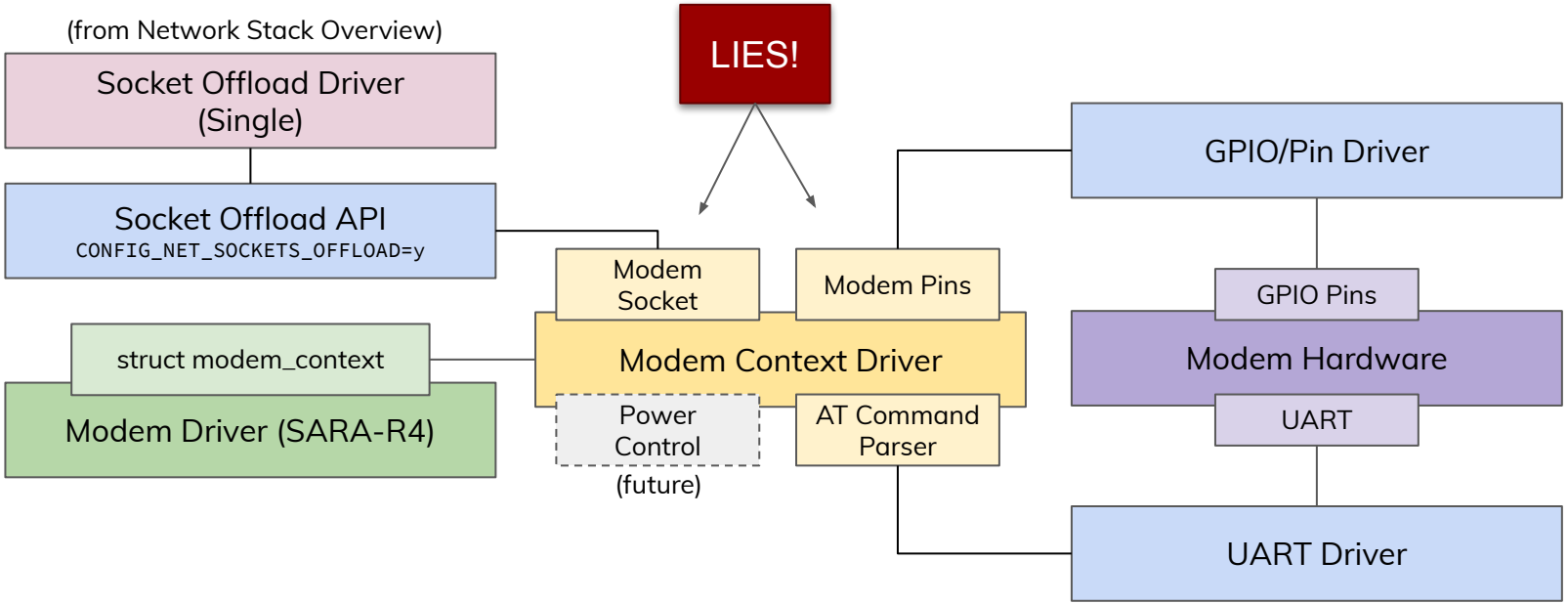
New Modem Driver Implementation Model for v2.0

Modem drivers have less duplicated code



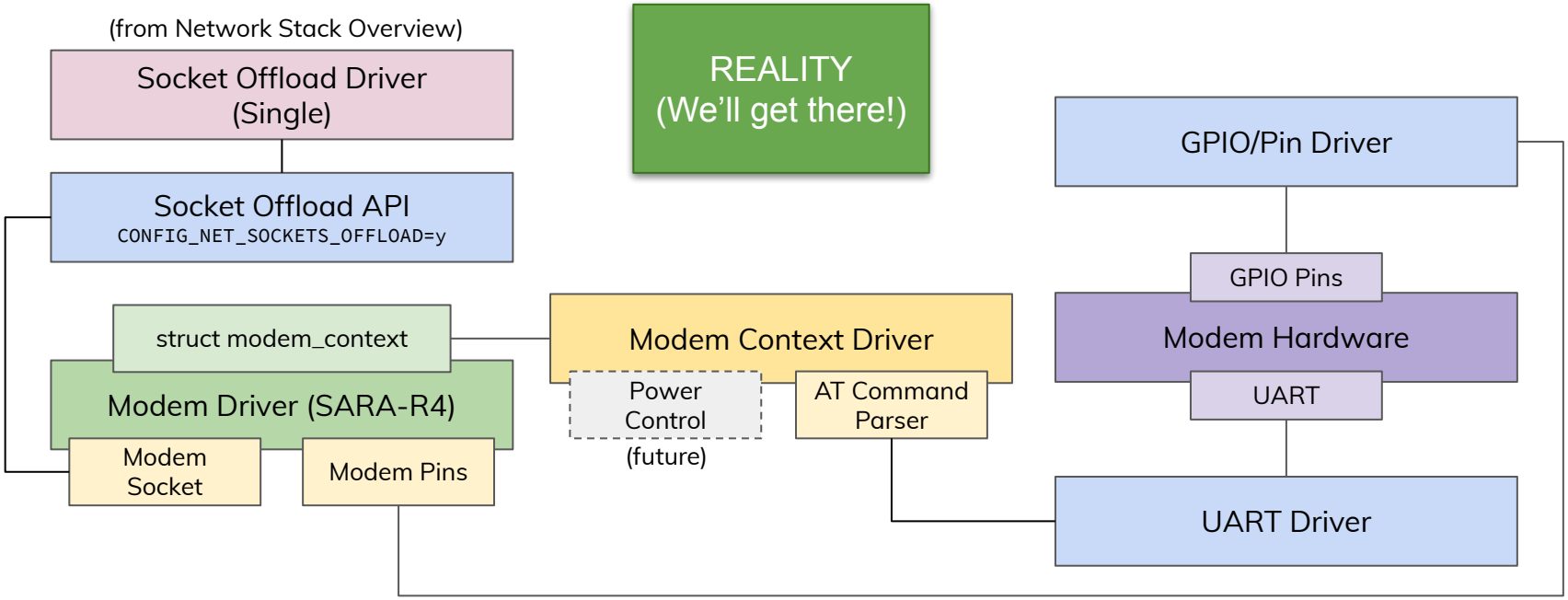
New Modem Driver Implementation Model for v2.0

Modem drivers have less duplicated code



New Modem Driver Implementation Model for v2.0

Modem drivers have less duplicated code



Net offload vs. Socket offload

Net offload API

Functions

- get, bind, listen, connect, accept, send, sendto, recv, put

Parameters

- struct net_context is the primary handle that a driver keeps track of for the connection lifecycle and the main network data structure is a net_pkt structure (meta data and a linked list of buffers).

The main functionality of this method of offloading network calls is for … handling network data in a very similar way as the core of Zephyr's network stack.

Socket offload API (POSIX-based)

Functions

- socket, close, accept, bind, listen, connect, poll, setsockopt, getsockopt, recv, recvfrom, send, sendto, getaddrinfo, freeaddrinfo and fcntl.

Parameters

- Integer-based file descriptor is the primary handle that a driver keeps track of as sockets are opened and closed. Network data is a char array buffer.

APIs make other functions such as security socket options and DNS available to the drivers to handle in their own way.

Many modems use a socket-based approach when designing the commands.

Anatomy of a modem helper layer

What the heck is a modem receiver?

August 2018: modem receiver helper introduced to Zephyr™ along with the Wistron M14A2A modem driver

- ISR triggers a function to pull data off the modem's UART interface and stuff it into a thread-safe ring buffer.
- Modem drivers then implement a loop that checks a "receive" function to see if there's data in the ring buffer. If there is, they empty the ring buffer into a "parsing" buffer and hand it off to the driver's internal parsing logic.
- Very specialized to the first and second drivers that were implemented
- Was a combination of UART-specific data and the semi-public interfaces that were needed by other parts of Zephyr™ to integrate with the modem layer such as the modem shell.

So … what's the problem?

- May 2019, I submit a new SARA-R4 modem driver
- Most of the code lives in each modem driver so there's a lot of copy/paste
- Community starts to get interested
- Bug fixes are hard to submit, because of copy / paste
- It's confusing where to contribute
- Hard to improve on the parser code, because each driver implements their own.

Let’s do something better

August 9th, the “modem context” is introduced to Zephyr™

Establish the semi-public APIs of 3 components needed by modem drivers (these are defined in drivers/modem/modem_context.h):

- Modem interface: read() and write() abstraction
- Command parser: process() abstraction
- Pin config: helper layer for setting up pins needed by modem drivers

Modem Interface vs. Modem Receiver

Why create a new “thing” when modem receiver already existed?

Modem interface is generic. The implementation lives in a different file.

The first interface supported is for handling UART, and that lives in drivers/modem/modem_iface_uart.c

The only thing exposed to the modem context a pointer to the read() and write() functions. These are set when calling modem_iface_uart_init() function which takes a modem context parameter.

We can add more implementations as needed, and those should still work with the existing command parser implementations, etc.

Command Response Parsing

Parsing the modem responses for matches which can be sent back to the modem drivers ended up being fairly large chunks of code. And as each modem driver was submitted, this was being duplicated.

Enter the command parser element of the modem context. The public process() function is a single point of entry where data can be sent after it's gathered from the interface. It's up to the implementation how it buffers that data and triggers command handlers defined by the drivers.

There will probably be different variations on command parsers implemented over time. These should be compatible with the interface implementations.

Example of a command handler

Parsing the modem responses for matches which can be sent back to the modem drivers ended up being fairly large chunks of code. And as each modem driver was submitted, this was being duplicated.

Enter the command parser element of the modem context. The public process() function is a single point of entry where data can be sent after it's gathered from the interface. It's up to the implementation how it buffers that data and triggers command handlers defined by the drivers.

There will probably be different variations on command parsers implemented over time. These should be compatible with the interface implementations.

Example of command handlers

Create your handlers with the MODEM_CMD_DEFINE() macro in 3 flavors: response handlers (OK, ERROR), unsolicited handlers (+CREG: 0) and actual command response handlers (create socket response):

```
MODEM_CMD_DEFINE(on_cmd_ok) { <code> }
MODEM_CMD_DEFINE(on_cmd_socknotifycreg) { <code> }
```

Attach these handlers to their trigger texts, set the # of parsed parameters and their delimiters. These parameters are returned as char **argv and u16_t argc to the handler functions:

```
static struct modem_cmd response_cmds[] = {
    MODEM_CMD("OK", on_cmd_ok, 0U, ""),
    MODEM_CMD("ERROR", on_cmd_error, 0U, ""),
};
static struct modem_cmd unsol_cmds[] = {
    MODEM_CMD("+CREG: ", on_cmd_socknotifycreg, 1U, ""),
};
```

Pin config helpers in a bit more detail

Define an enum such as this:

```
enum mdm_control_pins { MDM_POWER = 0, MDM_RESET };
```

Create an array of `struct modem_pin`. Use MODEM_PIN() macro to specify: gpio controller, pin # and flags.

```
static struct modem_pin modem_pins[] = {
      MODEM_PIN(DT_INST_0_UBLOX_SARA_R4_MDM_POWER_GPIOS_CONTROLLER,
              DT_INST_0_UBLOX_SARA_R4_MDM_POWER_GPIOS_PIN, GPIO_DIR_OUT), /* MDM_POWER */
      MODEM_PIN(DT_INST_0_UBLOX_SARA_R4_MDM_RESET_GPIOS_CONTROLLER,
              DT_INST_0_UBLOX_SARA_R4_MDM_RESET_GPIOS_PIN, GPIO_DIR_OUT), /* MDM_RESET */
};
```

DTS generated!

Create a pin_init() function. Functions like modem_pin_write() and modem_pin_read() make this easy.

```
static int pin_init(void)
{
      modem_pin_write(&mctx, MDM_RESET, 0); /* RESET no asserted */
      modem_pin_write(&mctx, MDM_POWER, 1); /* enable POWER */
}
```

mctx is a struct modem_context *

TIP: Use the pin_init() function in a place like modem_reset() or similar.

Socket offload ops in brief

Create your offloaded socket handling functions:

```
static int offload_socket(int family, int type, int proto) { <code> }
static int offload_close(int sock_fd) { <code> }
…
```

Connect these functions to the socket_offload structure:

```
static const struct socket_offload modem_socket_offload = {
     .socket = offload_socket,
     .close = offload_close,
…
```

Call the registration function in interface init function:

```
socket_offload_register(&modem_socket_offload);
```

Some helpful tips for writing a modem driver

Helpful tips

- Make sure you have the data sheet and AT-command reference for your hardware (**You're going to need it!**)
- Get familiar with an existing driver such as drivers/modem/ublox-sara-r4.c
- When testing be sure to enable the following CONFIG items:
  - CONFIG_MODEM_SHELL=y
  - CONFIG_MODEM_LOG_LEVEL_DBG=y
  - CONFIG_MODEM_CONTEXT_VERBOSE_DEBUG=y
- Start with a sample like hello_world. This let's you use the shell to test the modem. Example:

```
west build -b frdm_k64f -s zephyr/samples/hello_world -- -DSHIELD=sparkfun_sara_r4 \
 -DCONFIG_LOG=y -DCONFIG_NETWORKING=y -DCONFIG_NET_SOCKETS=y \
 -DCONFIG_MODEM_UBLOX_SARA_R4_APN=\"hologram\" \
 -DCONFIG_MODEM_UBLOX_SARA_R4_MANUAL_MCCMNO=\"310410\" \
 -DCONFIG_MODEM_SHELL=y -DCONFIG_MODEM_LOG_LEVEL_DBG=y \
 -DCONFIG_MODEM_CONTEXT_VERBOSE_DEBUG=y
```

- Get familiar with the SARA-R4 driver:
  https://github.com/zephyrproject-rtos/zephyr/blob/master/drivers/modem/ublox-sara-r4.c

Modem hardware supported by Zephyr™ RTOS

AT&T IoT Starter Kit

Avnet / AT&T

Hardware features:

- LTE-M / NB-IoT radio via Wistron WNC-M14A2A in an Arduino R3 shield form-factor. NOTE: pins are not standard for UART.
- Can be used on FRDM-K64F with Zephyr™ RTOS via "`-DSHIELD=wnc_m14a2a`" build argument.

Software:

- Basic mainline Zephyr™ RTOS support. You will need to edit the IP peer address in most samples. NOTE: This doesn't include things like power management, DNS, offloaded MQTT and other extended features.

http://cloudconnectkits.org/product/att-iot-starter-kit-powered-aws

LTE CAT M1/NB-IoT Shield: SARA-R4

SparkFun Electronics®

Hardware features:

- LTE-M / NB-IoT radio via SARA-R4 modem in an Arduino R3 shield form-factor.
- This means it can be used on several different boards with Zephyr™ RTOS via "`-DSHIELD=sparkfun_sara_r4`" build argument!

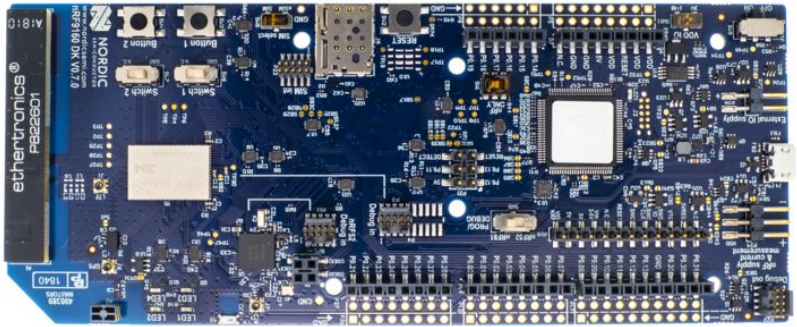
Software:

- Basic mainline Zephyr™ RTOS support. You will need to edit the IP peer address in most samples. NOTE: This doesn't include things like power management, DNS, offloaded MQTT and other extended features.

https://www.sparkfun.com/products/14997
https://www.u-blox.com/en/product/sara-r4-series

nRF9160 Development Kit

Nordic Semiconductor



Hardware features:

- nRF9160 SiP for LTE-M, NB-IoT and GPS
- Includes an nRF52840 board controller that for example can be used to build a Bluetooth Low Energy gateway

Software:

- Zephyr-based nRF Connect SDK includes everything needed to get started, application layer protocols, examples, peripheral drivers and more. The kit can easily be connected to Nordic's cloud solution, nRF Connect for Cloud.

https://www.nordicsemi.com/Software-and-Tools/Development-Kits/nRF9160-DK
https://www.nordicsemi.com/Software-and-Tools/Software/nRF-Connect-SDK

Pinnacle™ 100 modem

Laird Connectivity

Hardware features:

- LTE CAT M1 / NB-IoT radio via Sierra HL7800 modem
- MCU / Bluetooth 5 radio Nordic nRF52840

Software:

- Zephyr-based out-of-the-box demo app provided by Laird Connectivity allows you to commission your HW with a mobile app (BLE) to communicate with AWS. They also provide a demo website where the user can log in and see sensor data that is being reported to the cloud.
- Hopefully, an upstream driver for the Sierra HL7800 modem to be submitted soon.

https://www.lairdconnect.com/wireless-modules/cellular-solutions/pinnacle-100-modem

Next steps!

What's next for modem support in Zephyr™?

Or better known as "Sounds great! Where can I jump in?"

There are quite a few things to work on:

- eDRX/low power mode support (integrated w/ Zephyr™ power subsystem)
- 3GPP standard interface: out of the box support for most of the AT-based commands that are defined under 3GPP TS 27.007 (v16.1.0 as of Jun 14) http://www.3gpp.org/ftp/Specs/archive/27_series/27.007/27007-g10.zip
- Moving the offloaded socket "boiler-plate" code into modem_socket.c.
- "Dummy" modem driver + test suite to run on the modem helper driver for UART, socket communications and command handlers.
- Storage for modem configurations and / or APN lists, etc.
- And of course **more drivers**

Questions?

Thank you for listening!