Serial Modem Documentation

Nordic Semiconductor

latest



Contents

1 Getting started **2**
- 1.1 Initialization . . . 2
- 1.2 Building and running . . . 3

2 Serial Modem application **5**
- 2.1 Building and running . . . 6
- 2.2 Logging and tracing . . . 11
- 2.3 Configuration . . . 18
- 2.4 Cellular PPP modem . . . 25
- 2.5 Releasing Serial Modem . . . 36
- 2.6 Extending the application . . . 39
- 2.7 Testing scenarios . . . 40

3 Serial Modem AT commands **52**
- 3.1 AT command syntax . . . 52
- 3.2 Generic AT commands . . . 52
- 3.3 CMUX AT commands . . . 68
- 3.4 CoAP client AT commands . . . 74
- 3.5 DFU AT commands . . . 84
- 3.6 FOTA AT commands . . . 96
- 3.7 GNSS AT commands . . . 101
- 3.8 HTTP client AT commands . . . 109
- 3.9 ICMP AT commands . . . 119
- 3.10 LwM2M carrier library AT commands . . . 121
- 3.11 MQTT client AT commands . . . 134
- 3.12 nRF Cloud AT commands . . . 145
- 3.13 nRF Provisioning AT commands . . . 164
- 3.14 PPP AT commands . . . 167
- 3.15 SMS AT commands . . . 173
- 3.16 Socket AT commands . . . 175
- 3.17 Trace AT commands . . . 216
- 3.18 Data mode . . . 220

4 UART configuration **225**
- 4.1 UART interface . . . 225
- 4.2 Board-specific pin mapping . . . 226
- 4.3 Controlling the UART power state . . . 231
- 4.4 UART security considerations . . . 236

5 Host **238**
- 5.1 AT Client Shell sample . . . 239

5.2 PPP shell sample 240
5.3 Serial Modem AT Client library 246
5.4 Serial Modem XDFU library 253

6 Releases **256**
6.1 Release notes 256
6.2 Release artifacts 258
6.3 Migration notes 259

Index **279**

The Serial Modem contains the Serial Modem application for nRF91-based devices, and is built on nRF Connect SDK. It includes host examples in the form of the *AT Client Shell sample* and the *PPP shell sample*.

If you are looking for the nRF91M1 AT commands, see the nRF91M1 AT Commands Reference Guide documentation.

1 Getting started

Before getting started, make sure you have a proper nRF Connect SDK development environment. Follow the official Getting started guide.

There are two options for setting up the project, depending on your preferred development environment:

- Using nRF Connect for VS Code (recommended)
- Using the command line

1.1 Initialization

This section represents alternative approaches for initializing the workspace.

nRF Connect for VS Code

1. To install the nRF Connect SDK and its toolchain using nRF Connect for VS Code, follow extension documentation or Installing nRF Connect SDK and VS Code exercise on Nordic Developer Academy.
2. Open the nRF Connect extension in Visual Studio Code by clicking its icon in the *Activity Bar*.
3. Click *Create New Application*.
4. Select the **Browse nRF Connect SDK add-on Index** option.
5. Search for the **Serial Modem** application and create the project.

Command line

Add Serial Modem into existing nRF Connect SDK workspace

Assume you have an existing nRF Connect SDK workspace in the `ncs` folder.

1. Navigate to the workspace folder:

```
cd ncs
```

2. Clone Serial Modem repository:

```
git clone https://github.com/nrfconnect/ncs-serial-modem.git
```

3. Set manifest path to the Serial Modem:

```
west config manifest.path ncs-serial-modem
```

4. Update the nRF Connect SDK modules:

```
west update
```

Depending on the current state of your nRF Connect SDK modules, this may take several minutes as it fetches all nRF Connect SDK modules according to the requirements of the Serial Modem.

Initialize workspace and add Serial Modem from scratch

Complete the following steps for initialization:

1. To initialize the workspace:

```
west init -m https://github.com/nrfconnect/ncs-serial-modem --mr
main ncs
```

2. Navigate to the workspace folder:

```
cd ncs
```

3. Update the nRF Connect SDK modules:

```
west update
```

This may take several minutes as it fetches all nRF Connect SDK modules according to the requirements of the Serial Modem.

1.2 Building and running

Before building and running the application, you need to connect the development kit to your PC using a USB cable and power on the development kit. Complete the following steps for building and running:

nRF Connect for VS Code

1. Complete the steps in the How to build an application section in the extension documentation to build the application.

   You can also specify the additional configuration variables (Kconfig, DTC overlays, and extra Kconfig options) while setting up a build configuration during building the application. See Providing CMake options for more information on setting the additional configuration variables.

2. Open the nRF Connect extension in Visual Studio Code by clicking its icon in the *Activity Bar*.
3. In the extension's Actions View, click on *Flash*.

The application is now built and flashed to the device. You can open a serial terminal to use the application. The default baud rate is 115200.

Command line

1. From the workspace folder, navigate to the `ncs-serial-modem` folder:

   ```
   cd ncs-serial-modem/app
   ```

2. To build the application, run the following command:

   ```
   west build -p -b nrf9151dk/nrf9151/ns
   ```

   To build the application with nRF91M1 configuration, run the following command, which utilizes the configuration from the `app/sample.yaml` file:

   ```
   west build -b nrf9151dk/nrf9151/ns -p -T ./serial_modem.nrf91m1
   ```

   To build the application with Kconfig, DTC overlays, and extra Kconfig options, the following command gives an example of how they are passed as build arguments:

   ```
   west build -p -b nrf9151dk/nrf9151/ns -- -DEXTRA_CONF_FILE="ppp.
   conf;cmux.conf" -DEXTRA_DTC_OVERLAY_FILE="external-mcu.overlay"
   -DCONFIG_SM_LOG_LEVEL_DBG=y
   ```

3. To program the application, run the following command:

   ```
   west flash
   ```

The application is now built and flashed to the device. You can open a serial terminal to use the application. The default baud rate is 115200.

Note:

When building the Serial Modem application, the manifest path must point to the `ncs-serial-modem` folder. Otherwise linking will fail as drivers and libraries from Serial Modem will not be found.

To check your current manifest path, run the following command anywhere in your workspace:

```
west config manifest.path
```

To set the manifest path for Serial Modem, run the following command:

```
west config manifest.path ncs-serial-modem
```

2 Serial Modem application

The Serial Modem (SM) application can be used to emulate a stand-alone modem on an nRF91 Series device. This allows you to run your application on a separate host MCU. The application accepts both the modem specific AT commands and proprietary AT commands. The AT commands are documented in the following guides:

- Modem specific AT commands
  - `mfw_nrf91x1` - nRF91x1 AT Commands Reference Guide
  - `mfw_nrf9151-ntn` - nRF91x1 NTN AT Commands Reference Guide
- Proprietary AT commands - *Serial Modem AT commands*

The application supports the following modem firmware:

- `mfw_nrf91x1` v2.0.4 or later
- `mfw_nrf9151-ntn`

The application supports the following development kits:

Hardware platforms	PCA	Board name	Board target	SW maturity level
nRF9151 DK	PCA10171	nrf9151dk	`nrf9151dk/nrf9151/ns`	Supported
Thingy:91 X	PCA20065	thingy91x	`thingy91x/nrf9151/ns`	Experimental

For more security, the application must use the `*/ns` variant of the board target, which is required when using the nRF91 Series Modem library. When built for this variant, the sample is configured to compile and run as a non-secure application using security by separation. Therefore, it automatically includes Trusted Firmware-M that prepares the required peripherals and secure services to be available for the application.

This application has the following dependencies:

- This application uses the following nRF Connect SDK libraries:
  - AT parser
  - AT monitor
  - Modem library integration layer
  - Modem JWT

  - SMS
  - FOTA download
  - Downloader
  - nRF Cloud
  - nRF Cloud A-GNSS
  - nRF Cloud P-GPS
  - nRF Cloud location
  - nRF Cloud device provisioning
- It uses the following sdk-nrfxlib libraries:
  - Modem library
- In addition, it uses the following secure firmware component:
  - Trusted Firmware-M

See the subpages for how to use the application, how to extend it, and information on the supported AT commands.

2.1 Building and running

The Serial Modem application can be found under `app` in the Serial Modem folder structure.

For more security, it is recommended to use the `*/ns` variant of the board target (see the Requirements section above.) When built for this variant, the sample is configured to compile and run as a non-secure application using security by separation. Therefore, it automatically includes Trusted Firmware-M that prepares the required peripherals and secure services to be available for the application.

To build the sample, follow the instructions in Building an application for your preferred building environment. See also Programming an application for programming steps and Testing and optimization for general information about testing and debugging in the nRF Connect SDK.

Note:

When building repository applications in the SDK repositories, building with sysbuild is enabled by default. If you work with out-of-tree freestanding applications, you need to manually pass the `--sysbuild` parameter to every build command or configure west to always use it.

2.1.1 Communicating with the modem on an nRF91 Series DK

In this scenario, an nRF91 Series DK running the Serial Modem application serves as the host. You can use either a PC or an external MCU as a client.

Connecting with a PC

To connect to an nRF91 Series DK with a PC:

1. Verify that `UART_0` is selected in the application. It is defined in the default configuration.
2. Use the Serial Terminal app to connect to the development kit. See Testing and optimization for instructions. You can also use the *Open Serial Terminal* option of the Cellular Monitor app to open the Serial Terminal app. Using the Cellular Monitor app in combination with the Serial Terminal app shows how the modem responds to the different modem commands. You can then use this connection to send or receive AT commands over UART, and to see the log output of the development kit.

   Instead of using the Serial Terminal app, you can use PuTTY to establish a terminal connection to the development kit, using the default serial port connection settings.

   Note:

   The default AT command terminator is a carriage return (`\r`). The Serial Terminal app, PuTTY and many terminal emulators support this format by default. However, make sure that the configured AT command terminator corresponds to the line terminator of your terminal. See *Configuration options* for more details on AT command terminator choices.

Connecting with an external MCU

Note:

This section does not apply to Thingy:91 X.

If you run your user application on an external MCU (for example, an nRF54 Series development kit), you can control the Serial Modem application on an nRF91 Series device directly from the application. See the *AT Client Shell sample* for a sample implementation of such an application.

To connect with an external MCU using UART_2, include the `external-mcu.overlay` devicetree overlay in your build. This overlay configures the UART_2 pins, DTR pin, and RI pin for the nRF9151 DK.

If you use a different setup, you can customize the `external-mcu.overlay` file to match your hardware configuration in (for example) the following ways:

- Change the highlighted UART baud rate or DTR and RI pins:

```
&uart2 {
   compatible = "nordic,nrf-uarte";
   current-speed = <115200>;
   hw-flow-control;
   status = "okay";

   dtr_uart2: dtr-uart {
```

```
      compatible = "nordic,dtr-uart";
      dtr-gpios = <&gpio0 31 (GPIO_PULL_UP | GPIO_ACTIVE_
LOW)>;
      ri-gpios = <&gpio0 30 GPIO_ACTIVE_LOW>;
      status = "okay";
   };
   pinctrl-0 = <&uart2_default_alt>;
   pinctrl-1 = <&uart2_sleep_alt>;
   pinctrl-names = "default", "sleep";
};
```

- Change the highlighted UART pins:

```
&pinctrl {
   uart2_default_alt: uart2_default_alt {
      group1 {
         psels = <NRF_PSEL(UART_RX, 0, 3)>,
                 <NRF_PSEL(UART_CTS, 0, 7)>;
         bias-pull-up;
      };
      group2 {
         psels = <NRF_PSEL(UART_TX, 0, 2)>,
                 <NRF_PSEL(UART_RTS, 0, 6)>;
      };
   };

   uart2_sleep_alt: uart2_sleep_alt {
      group1 {
         psels = <NRF_PSEL(UART_TX, 0, 2)>,
                 <NRF_PSEL(UART_RX, 0, 3)>,
                 <NRF_PSEL(UART_RTS, 0, 6)>,
                 <NRF_PSEL(UART_CTS, 0, 7)>;
         low-power-enable;
      };
   };
};
```

The following table shows how to connect selected development kit to an nRF91 Series development kit to be able to communicate through UART:

nRF54L15 DK	nRF91 Series DK
UART TX P0.00	UART RX P0.03
UART RX P0.01	UART TX P0.02
UART CTS P0.03	UART RTS P0.06
UART RTS P0.02	UART CTS P0.07
DTR OUT P1.11	DTR IN P0.31
RI IN P1.12	RI OUT P0.30
GPIO GND	GPIO GND

Note:

You must disable the VCOM0 on the nRF54L15 DK to release the UART GPIO pins to use it with the *AT Client Shell sample*.

- For nRF54L15 DK, you can use the Board Configurator app to disable the `Connect port VCOM0` setting.

Note:

The GPIO output levels on the nRF91 Series device and nRF54L15 DK must be the same.

- You can set the VDD voltages for both devices with the Board Configurator app.

Use the following UART devices:

- nRF54 Series DK - UART30
- nRF91 Series DK - UART2

The UART configuration must match on both sides. By default the Serial Modem application and *AT Client Shell sample* use the following settings:

- Hardware flow control: enabled
- Baud rate: 115200
- Parity bit: no

2.1.2 Communicating with the modem on Thingy:91 X

In this scenario, Thingy:91 X running the Serial Modem application serves as the host. You can use a PC as a client.

Connecting with a PC

The nRF5340 SoC of Thingy:91 X is pre-programmed with the Connectivity bridge application. To update the Connectivity bridge application, see the Updating the Thingy:91 X firmware using nRF Util documentation. The Connectivity bridge application routes `UART_0` to `USB_CDC0` on Thingy:91 X. By enabling the `CONFIG_BRIDGE_BLE_ENABLE` Kconfig option in the Connectivity bridge application, you can also use Serial Modem over Nordic UART Service (NUS).

To connect to a Thingy:91 X with a PC:

1. Verify that `UART_0` is selected in the application. It is defined in the default configuration.
2. Use the Serial Terminal app to connect to the development kit. See Testing and optimization for instructions. You can also use the *Open Serial Terminal* option of the Cellular Monitor app to open the Serial Terminal app. Using the Cellular Monitor app in combination with the Serial Terminal app shows how the modem responds to the different modem commands. You can then use this connection to send or receive AT commands over UART, and to see the log output of the development kit.

Instead of using the Serial Terminal app, you can use PuTTY to establish a terminal connection to the development kit, using the default serial port connection settings.

Note:

The default AT command terminator is a carriage return (`\r`). The Serial Terminal app, PuTTY and many terminal emulators support this format by default. However, make sure that the configured AT command terminator corresponds to the line terminator of your terminal. See *Configuration options* for more details on AT command terminator choices.

2.1.3 Testing

The following testing instructions focus on testing the application with a PC client.

After programming the sample to your development kit, complete the following steps to test it:

1. Connect the kit to the computer using a USB cable. The kit is assigned a serial port. Serial ports are referred to as COM ports on Windows, /dev/ttyACM devices on Linux, and /dev/tty devices on macOS. To list Nordic Semiconductor devices connected to your computer together with their serial ports, open a terminal and run the `nrfutil device list` command. Alternatively, check your operating system's device manager or its equivalent.
2. Connect to the kit with the Serial Terminal app. You can also use the *Open Serial Terminal* option of the Cellular Monitor app to open the Serial Terminal app. If you want to use a different terminal emulator, see *Connecting with a PC*.
3. Reset the kit.
4. Observe that the development kit sends a `Ready\r\n` message on UART.
5. Send AT commands and observe the responses from the development kit.

See *Testing scenarios* for typical test cases.

Note:

If the initialization of Serial Modem fails, the application sends an `INIT ERROR\r\n` message on UART if it has managed to enable UART. See the logs for more information about the error. The logs are written to RTT by default.

2.1.4 Using the LwM2M carrier library

The application supports the nRF Connect SDK LwM2M carrier library that you can use to connect to the operator's device management platform. See the library's documentation for more information and configuration options.

To enable the LwM2M carrier library, add the following parameters to your build command:

```
-DEXTRA_CONF_FILE=carrier.conf
-DEXTRA_DTC_OVERLAY_FILE=carrier-nrf9151.overlay
```

The CA root certificates that are needed for modem FOTA are not provisioned in the Serial Modem application. You can flash the Cellular: LwM2M carrier sample to write the certificates to modem before flashing the Serial Modem application, or use the Cellular: AT Client sample as explained in preparing the Cellular: LwM2M Client sample for production. It is also possible to modify the Serial Modem application project itself to include the certificate provisioning, as demonstrated in the Cellular: LwM2M carrier sample.

```
int lwm2m_carrier_event_handler(const lwm2m_carrier_event_t *event)
{
        switch (event->type) {
        case LWM2M_CARRIER_EVENT_INIT:
                carrier_cert_provision();
        ...
```

The certificate provisioning can also be done directly in the Serial Modem application by using the same AT commands as described for the Cellular: AT Client sample.

When the LwM2M carrier library is in use, by default the application will auto-connect to the network on startup. This behavior can be changed by disabling the *CONFIG_SM_AUTO_CONNECT* option.

2.1.5 Building without the immutable B0 bootloader

By default, the Serial Modem application uses an immutable bootloader B0 (NSIB) and an upgradeable MCUboot bootloader.

Version 1.x.x of the Serial Modem application uses only an upgradeable MCUboot bootloader. Building without the B0 bootloader is offered as backwards compatibility option if you want to maintain the same partition layout as in Serial Modem version 1.x.x.

To build without B0 and with a single, non-updatable MCUboot slot, include `disable-b0-nrf9151.overlay` in both the application and MCUboot images and disable the secure boot appcore in the sysbuild configuration:

```
west build -p -b nrf9151dk/nrf9151/ns -- -DEXTRA_DTC_OVERLAY_
FILE=disable-b0-nrf9151.overlay -Dmcuboot_EXTRA_DTC_OVERLAY_FILE=
<path-to-app>/disable-b0-nrf9151.overlay -DSB_CONFIG_SECURE_BOOT_
APPCORE=n
```

Where `<path-to-app>` is the absolute path to the `app` directory.

This configuration removes the `b0` and `s1` partitions, giving MCUboot a single 64 kB slot.

2.2 Logging and tracing

The Serial Modem and the modem produce separate logs with their own methods. The logs of the Serial Modem are called application logs, and the logs of the modem are called

modem traces.

2.2.1 Application logging

The Serial Modem outputs B0, MCUboot, and application logs over `UART1` (VCOM1 on the nRF9151 DK) during boot. After the application is initialized, the log backend is disabled, and the UART is suspended. Use `AT#XLOG=1` or `AT#XLOG=2` to enable application logs and resume the UART. This avoids UART power overhead when logs are not needed during normal operation. See *Trace AT commands* for the full command reference.

Nordic Semiconductor support might ask you for application logs when investigating an issue.

The `zephyr,console` and `ncs,sm-uart` chosen nodes must point to different UARTs. The Serial Modem enables asynchronous reception on the AT command UART and cannot suspend it. The build fails if both nodes point to the same UART.

TF-M logging must use the same UART as the application. For more details, see shared TF-M logging.

Note:

The negative error codes that are visible in logs are *errno* codes defined in nrf_errno.h.

Log levels

The Serial Modem uses the following log levels, from the most severe to the most verbose:

Level	Usage
Error (ERR)	An operation failed and the Serial Modem cannot complete it.
Warning (WRN)	Something unexpected happened, but the Serial Modem can continue.
Information (INF)	Normal operation. Includes the AT commands, responses, and notifications as text, with sensitive payloads redacted.
Debug (DBG)	Detailed internal information for debugging. Includes the AT commands, responses, and notifications as raw hex dumps.

The Serial Modem shows a log message only if its level is enabled. Enabling a level also enables all the levels above it in the list. The log level is set at build time separately for each software component.

By default, the Serial Modem logs at the debug level, which is required for `AT#XLOG=2`. To change the level, set one of the `CONFIG_SM_LOG_LEVEL_ERR`, `CONFIG_SM_LOG_LEVEL_WRN`, `CONFIG_SM_LOG_LEVEL_INF`, or `CONFIG_SM_LOG_LEVEL_DBG` Kconfig options. The other components, such as Zephyr and the nRF Connect SDK libraries, log at the information level unless their level is set explicitly. To change the level of a component, set its own Kconfig option, for example, the `CONFIG_MODEM_CMUX_LOG_LEVEL_DBG` Kconfig option. The `CONFIG_LOG_DEFAULT_LEVEL` Kconfig option sets the default level for the components that do not set their own.

You select what the Serial Modem outputs on the log UART at runtime with the AT#XLOG command:

AT#XLOG mode	Log UART	AT traffic logged as
0	Suspended	Not output
1	Active	Information level text, with sensitive payloads redacted
2	Active	Debug level hex dump

Use mode 1 for normal troubleshooting. Use mode 2 only when you need the exact bytes of the AT traffic, for example when the redaction hides the information that you need. Mode 2 outputs sensitive payloads, such as credentials, so do not share these logs publicly.

Enabling RTT logs

RTT logging is disabled by default. To switch from UART1 back to SEGGER RTT, add the following to your `prj.conf`:

```
# Disable UART logging
CONFIG_UART_CONSOLE=n
CONFIG_LOG_BACKEND_UART=n

# Enable Segger RTT
CONFIG_USE_SEGGER_RTT=y
CONFIG_RTT_CONSOLE=y
CONFIG_LOG_BACKEND_RTT=y
# Optional: increase the buffer so that boot logs are not lost
before RTT Viewer connects.
CONFIG_SEGGER_RTT_BUFFER_SIZE_UP=2048
```

You can view the RTT logs with an RTT client such as `J-Link RTT Viewer`. See Testing and optimization for instructions.

Note:

nRF9151 anomaly 36 locks the debug port when the application reaches a low power state (<3 uA current consumption). This takes place when DTR is deasserted and the RTT client, such as J-Link RTT Viewer, is not connected. Since the RTT backend relies on the debug port, the RTT client must be connected before the application enters a low power state to avoid this issue.

Note:

Modem traces captured through UART are corrupted if application logs through RTT are simultaneously captured. When capturing modem traces through UART with the Cellular Monitor app and simultaneously capturing RTT logs, for example, with J-Link RTT Viewer, the modem trace misses packets, and captured packets might have incorrect information.

If you need to capture modem traces and RTT logs at the same time, enable HW flow control for modem trace UART. Otherwise, you can choose not to capture RTT logs. Having only RTT logs enabled does not cause this issue.

2.2.2 Modem traces

The modem outputs modem traces that you can use to analyze the behavior of the modem. Collect the modem traces for the following purposes:

- Inspecting the IP-level traffic during development.
- Analyzing network issues.
- Investigating modem crashes and suspected modem failures with the crash dump.

Nordic Semiconductor support might ask you for modem traces when investigating an issue.

Enable the modem traces before the problem occurs, because they cannot be collected afterwards. If the Serial Modem reports a modem fault with the *#XMODEM: FAULT* notification, or if you suspect another modem failure, repeat the scenario with the modem traces enabled.

Modem traces through UART

The Serial Modem outputs modem traces over UART1 (VCOM1 on the nRF9151 DK). The UART is shared with the application logs. The baud rate of the shared UART is set to 1000000 to support the high data rate required for modem traces.

Configuration Application log output over UART1 is included in the default build.

To also enable the modem trace backend (AT#XTRACE), build with the Kconfig overlay and the devicetree overlay that defines the modem trace SRAM partition:

```
west build -p -b nrf9151dk/nrf9151/ns -- -DEXTRA_CONF_FILE="trace-
backend-uart.conf" -DEXTRA_DTC_OVERLAY_FILE="trace-backend-nrf9151.
overlay"
```

After the application is initialized, the UART is suspended. Use AT#XLOG=1 to activate application logs and AT#XTRACE=1 to activate modem traces. See *Trace AT commands* for the full command reference.

Note:

Attempting to enable one backend while the other is active returns ERROR. Always disable the active backend before enabling the other.

Note:

When measuring power consumption, make sure both backends are disabled (`AT#XLOG=0` and `AT#XTRACE=0`) so the UART stays suspended. An active trace or log backend adds approximately 700 uA overhead.

Collecting traces The modem traces cannot be enabled while the application logs are enabled, so send `AT#XLOG=0` first.

To collect modem traces using the shared UART backend:

1. Capture the trace data from `UART1` (VCOM1 on the nRF9151 DK) at 1000000 baud rate, for example, with the Cellular Monitor app. See the Cellular Monitor app documentation for how to capture and save the trace.
2. Enable modem trace collection with the AT command UART:

```
AT#XTRACE=1
OK
```

3. Activate the modem to start generating trace data:

```
AT+CFUN=1
OK
```

4. Wait for the modem to register on the network and collect the traces.
5. When done, disable the modem trace backend:

```
AT#XTRACE=0
OK
```

Modem traces through CMUX

The Serial Modem application supports collecting modem traces through the CMUX multiplexer. When enabled, modem traces are sent through a dedicated CMUX channel, allowing simultaneous AT commands, PPP data, and trace collection over the same serial port. The trace CMUX channel is the first channel after the AT command channel and the PPP channel.

Note:

Unlike the *shared UART backend*, the CMUX trace backend does not use `AT#XTRACE`. When CMUX is started with `AT#XCMUX`, the trace channel is attached automatically. When CMUX is started with `AT+CMUX`, or to move the trace output to a different CMUX channel, use `AT#XCMUXTRACE`.

Configuration To use the CMUX trace backend, build the Serial Modem application with the trace backend configuration overlay in addition to the PPP and CMUX overlays, and the devicetree overlay that defines the modem trace SRAM partition:

```
west build -p -b nrf9151dk/nrf9151/ns -- -DEXTRA_CONF_FILE="ppp.
conf;cmux.conf;trace-backend-cmux.conf" -DEXTRA_DTC_OVERLAY_FILE=
"trace-backend-nrf9151.overlay"
```

For optimal throughput and to minimize trace data loss, configure the UART to run at maximum speed:

1. Set the UART speed in your devicetree configuration (for example, 1000000 baud).
2. Use the `-b` parameter with the script to match this speed (for example, `-b 1000000`).

Note:

Some trace data will be dropped. The amount depends on the UART speed, ongoing modem operations, and the trace level set with `AT%XMODEMTRACE`.

Setting trace level Configure the modem trace level using the AT%XMODEMTRACE command.

The modem trace subsystem automatically sends the `AT%XMODEMTRACE=1,2` command at startup. This provides the most trace data and includes the crash dump collection. Depending on the drop rate you observe, you might need to select a different `<set_id>` to reduce the amount of trace data generated.

Collecting traces through CMUX on Linux The `sm_start_ppp.sh` script for Linux host includes support for collecting modem traces. Use the `-T` flag to enable trace collection. Traces are saved to the `/var/log/nrf91-modem-trace.bin` file. The trace collection starts after the CMUX channel is established and continues until you stop the connection with the `sm_stop_ppp.sh` script.

The stop script automatically terminates trace collection and preserves the trace file for later analysis.

```
# Start PPP connection with trace collection with baud rate matching
the devicetree setting
$ sudo scripts/sm_start_ppp.sh -b 1000000 -T
Trace file: /var/log/nrf91-modem-trace.bin
Connect and wait for PPP link...
PPP link started

# Check that the trace is being collected
$ ls -la /var/log/nrf91-modem-trace.bin
-rw-r--r-- 1 root root 3467306 Jan 16 12:13 /var/log/nrf91-modem-
trace.bin
```

(continues on next page)

(continued from previous page)

```
# Stop PPP connection (also stops trace collection)
$ sudo scripts/sm_stop_ppp.sh
Stopping PPP link...
Waiting for Shutdown script to complete...
Stopping trace collection...
```

Analyzing traces

You can analyze the captured trace file in the Cellular Monitor app to inspect AT commands, network events, and IP-level details of the communication between the modem and the cellular network.

If you suspect a modem failure, or if the modem crashes and the trace collection was enabled, you can send the trace file to Nordic Semiconductor support for further analysis.

2.2.3 nRF Cloud observability

The Serial Modem collects diagnostic data about its operation. The host can ask the Serial Modem to upload the data to nRF Cloud, so you can use it to investigate crashes of devices in the field.

The Serial Modem collects the following data:

- Events, such as the reset reason after a restart and a modem fault with its reason and program counter.
- Metrics, which the Serial Modem collects into a heartbeat every hour. The metrics include the LTE metrics and connection statistics. See *#XNRFCLOUDOBSHEARTBEAT* for more information.
- Logs, which are the most recent information, warning, and error messages, with sensitive payloads redacted. The Serial Modem captures logs at these levels even when `AT#XLOG=0` disables UART logging. With `AT#XLOG=2`, the Serial Modem logs AT traffic only as debug-level hex dumps and does not include it in the observability logs. After a crash, the logs from before the crash are part of the core dump.
- A core dump, which the Serial Modem stores in flash when it crashes.

The data does not leave the device on its own. The host controls when it is sent, and the data reaches the cloud as follows:

1. The Serial Modem records the data on the device. When the Serial Modem crashes, it stores a core dump in flash and restarts. The core dump stays in flash over the restart until it is uploaded. The Serial Modem keeps the other data in RAM, so the data is lost on reset or power loss.
2. The host connects the Serial Modem to nRF Cloud with the *#XNRFCLOUD* command. You must onboard and claim the device first. See *nRF Provisioning AT commands*.
3. The host uploads the buffered data with one of the following commands:
   - *#XNRFCLOUDOBSUPLOAD* uploads the data when you issue it.
   - *#XNRFCLOUDOBSAUTO* uploads the data periodically. It is disabled by default.

The upload includes the buffered events, metrics, and logs. A stored core dump is included only when core dump upload is enabled with *#XNRFCLOUDOBSCOREDUMP*, which is the default.

4. nRF Cloud receives the data. By default, the data goes to the project that nRF Cloud associates with your device. To send the data to a specific project, specify the project key of that project in the upload command. You can analyze the events, metrics, logs, and core dumps in nRF Cloud.

Note:

The core dump storage holds one core dump. While a core dump is stored, subsequent crashes do not capture a new one until the stored one has been uploaded.

The core dump covers the crashes of the Serial Modem. The Serial Modem records a modem fault only as an event. The modem fault also drops the network connection, so the host cannot upload the event until the Serial Modem is connected to nRF Cloud again. If the host resets the Serial Modem before the upload, the event is lost. To investigate a modem fault, collect the modem traces as described in *Modem traces*.

The nRF Cloud observability AT commands require the *CONFIG_SM_NRF_CLOUD_OBSERVABILITY* Kconfig option, which is enabled by default. The `CONFIG_MEMFAULT_METRICS_HEARTBEAT_INTERVAL_SECS` Kconfig option sets the heartbeat interval. The default build stores the core dump in flash.

See *Observability #XNRFCLOUDOBS** for the full command reference.

2.3 Configuration

See Configuring and building in the nRF Connect SDK documentation for information about how to permanently or temporarily change the configuration.

2.3.1 Configuration options

CONFIG_SM_CUSTOMER_VERSION - Customer version string
Version string defined by the customer after customizing the application. When defined, this version is reported with the baseline versions by the AT#XSMVER AT command.

CONFIG_SM_CMUX - Enable CMUX functionality
This option is enabled by the CMUX overlay. It adds support for CMUX. See *CMUX AT commands* for more information.

CONFIG_SM_PPP - Enable PPP functionality
This option is enabled by the PPP overlay. It adds support for PPP. PPP can be used in conjunction with *CMUX* in order to use a single UART for both AT data and PPP. When CMUX is also enabled, PPP is usable only through a CMUX channel. See *PPP AT commands* for more information.

CONFIG_SM_EXTERNAL_XTAL - Use external XTAL for UARTE
This option configures the application to use an external XTAL for UARTE. For more

information, see the UARTE - Universal asynchronous receiver/transmitter with Easy-DMA section of the nRF9151 Product Specification documentation.

CONFIG_SM_AUTO_CONNECT - Connect to the network at start-up or reset
This option enables connecting to the network at start-up or reset using a defined PDN configuration. This option is enabled by the LwM2M Carrier overlay, but is otherwise disabled by default.

When enabled, the following sub-options are available for configuration:

CONFIG_SM_AUTO_CONNECT_SYSTEM_MODE - System mode for automatic network attach
This option defines the system mode to use for automatic network attach. The format is a comma-separated string of four single-digit integers: `"X,Y,Z,W"`

Where:

- X = LTE-M support (0=disabled, 1=enabled)
- Y = NB-IoT support (0=disabled, 1=enabled)
- Z = GNSS support (0=disabled, 1=enabled)
- `W` = LTE preference (0=no preference, 1=LTE-M, 2=NB-IoT, 3=network selection, 4=LTE-M then NB-IoT)

See the %XSYSTEMMODE command in the AT command Reference Guide for more details. The default value is `"1,1,1,0"` (enable LTE-M, NB-IoT, and GNSS with no preference).

CONFIG_SM_AUTO_CONNECT_PDN_CONFIG - Enable PDN configuration
This option enables sending of PDN configuration commands during automatic network attach. When enabled, the following sub-options become available:

CONFIG_SM_AUTO_CONNECT_PDN_APN - Access Point Name (APN)
This option specifies the APN to use for automatic network attach. If not set (default), the APN configured in the modem will be used.

CONFIG_SM_AUTO_CONNECT_PDN_FAMILY - PDN family (IP type)
This choice option selects the PDN family (IP address type) to use for automatic network attach. See the +CGDCONT command in the AT command Reference Guide for more details.

Available options:

- `CONFIG_SM_AUTO_CONNECT_PDN_FAMILY_IP` - Use IPv4 only
- `CONFIG_SM_AUTO_CONNECT_PDN_FAMILY_IPV6` - Use IPv6 only
- `CONFIG_SM_AUTO_CONNECT_PDN_FAMILY_IPV4V6` - Use both IPv4 and IPv6 (dual stack, default)
- `CONFIG_SM_AUTO_CONNECT_PDN_FAMILY_NON_IP` - Use Non-IP PDN type for data transmission without IP headers

CONFIG_SM_AUTO_CONNECT_PDN_AUTH - PDN authentication type
This option specifies the PDN authentication protocol to use for automatic network attach. See the +CGAUTH command in the AT command Reference Guide for more details.

Valid values:

- `0` - None (no authentication, default)
- `1` - PAP (Password Authentication Protocol)
- `2` - CHAP (Challenge Handshake Authentication Protocol)

CONFIG_SM_AUTO_CONNECT_PDN_USERNAME - PDN authentication username
This option specifies the username to use for PDN authentication during automatic network attach. Do not set the Kconfig option (default) if no username is required. Use only when *CONFIG_SM_AUTO_CONNECT_PDN_AUTH* is set to 1 (PAP) or 2 (CHAP).

CONFIG_SM_AUTO_CONNECT_PDN_PASSWORD - PDN authentication password
This option specifies the password to use for PDN authentication during automatic network attach. Leave empty (default) if no password is required. Use only when *CONFIG_SM_AUTO_CONNECT_PDN_AUTH* is set to 1 (PAP) or 2 (CHAP).

Example configuration overlay for NB-IoT with Non-IP PDN:

```
CONFIG_SM_AUTO_CONNECT=y
CONFIG_SM_AUTO_CONNECT_SYSTEM_MODE="0,1,0,0"
CONFIG_SM_AUTO_CONNECT_PDN_CONFIG=y
CONFIG_SM_AUTO_CONNECT_PDN_FAMILY_NON_IP=y
```

Example configuration overlay for LTE-M with custom APN and PAP authentication:

```
CONFIG_SM_AUTO_CONNECT=y
CONFIG_SM_AUTO_CONNECT_SYSTEM_MODE="1,0,0,0"
CONFIG_SM_AUTO_CONNECT_PDN_CONFIG=y
CONFIG_SM_AUTO_CONNECT_PDN_APN="internet"
CONFIG_SM_AUTO_CONNECT_PDN_FAMILY_IPV4V6=y
CONFIG_SM_AUTO_CONNECT_PDN_AUTH=1
CONFIG_SM_AUTO_CONNECT_PDN_USERNAME="myuser"
CONFIG_SM_AUTO_CONNECT_PDN_PASSWORD="mypass"
```

CONFIG_SM_CR_TERMINATION - CR termination
This option configures the application to accept AT commands ending with a carriage return. This is the default AT command terminator.

Select this option if you want to connect to the development kit using PuTTY. See Testing and optimization for instructions.

CONFIG_SM_LF_TERMINATION - LF termination
This option configures the application to accept AT commands ending with a line feed.

CONFIG_SM_CR_LF_TERMINATION - CR+LF termination
This option configures the application to accept AT commands ending with a carriage return followed by a line feed.

CONFIG_SM_SMS - SMS support in Serial Modem
This option enables additional AT commands for using the SMS service.

CONFIG_SM_GNSS - GNSS support in Serial Modem
This option enables additional AT commands for using the GNSS service.

CONFIG_SM_ICMP - ICMP ping support in Serial Modem
This option enables AT command for sending ICMP echo requests.

CONFIG_SM_NRF_CLOUD - nRF Cloud support in Serial Modem
This option enables additional AT commands for using the nRF Cloud service.

CONFIG_SM_NRF_CLOUD_LOCATION - nRF Cloud Location support
This option enables the nRF Cloud Location Services for cloud-assisted geolocation. Supports cellular and Wi-Fi positioning. This requires *CONFIG_SM_NRF_CLOUD* to be enabled.

CONFIG_SM_NRF_CLOUD_OBSERVABILITY - nRF Cloud observability support
This option enables the `#XNRFCLOUDOBS*` commands, which control the Memfault data upload over the nRF Cloud CoAP transport. This requires *CONFIG_SM_NRF_CLOUD*. See *Observability #XNRFCLOUDOBS** for more information.

CONFIG_SM_NRF_CLOUD_OBSERVABILITY_AUTO_INTERVAL_SECONDS - Default interval of the automatic upload
This option sets the interval, in seconds, that `#XNRFCLOUDOBSAUTO` uses when the host enables the automatic upload without giving one, and no interval has been stored yet. It defaults to one hour. The interval the host gives is persistent, so this value only applies until the host configures one. This requires *CONFIG_SM_NRF_CLOUD_OBSERVABILITY* to be enabled.

CONFIG_SM_NRF_CLOUD_OBSERVABILITY_DEBUG - Debug observability commands
This option enables the `#XNRFCLOUDOBSDEVINFO`, `#XNRFCLOUDOBSCRASH` and `#XNRFCLOUDOBSEXPORT` commands, which return the Memfault device information, force a crash for testing the coredump capture, and print the buffered Memfault chunks to the AT interface. It is disabled by default, because `#XNRFCLOUDOBSCRASH` deliberately crashes the application. Enable it only in development builds. This requires *CONFIG_SM_NRF_CLOUD_OBSERVABILITY* to be enabled.

CONFIG_SM_NRF_CLOUD_OBSERVABILITY_LTE_METRICS - Additional LTE metrics for Memfault
This option collects the cell ID, the tracking area code, the LTE mode, the PSM active time and periodic TAU, and the eDRX interval and paging time window on each Memfault heartbeat. It also tracks connection statistics between heartbeats, including the time to connect to the network, the number of connection losses, and the time the LTE link has been active. It is enabled by default. This requires *CONFIG_SM_NRF_CLOUD_OBSERVABILITY*.

CONFIG_SM_NRF_CLOUD_FOTA - nRF Cloud FOTA AT commands
This option enables the `#XNRFCLOUDFOTA*` commands, which check for and download application and modem firmware updates using Memfault release management, delivered over the same nRF Cloud CoAP transport as `#XNRFCLOUDOBS*`. This requires *CONFIG_SM_NRF_CLOUD*. See *nRF Cloud FOTA #XNRFCLOUDFOTA* for more information.

CONFIG_SM_COAPC - CoAP client support in Serial Modem
This option enables the CoAP client AT commands for making CoAP requests. See *CoAP client AT commands* for more information.

CONFIG_SM_MQTTC - MQTT client support in Serial Modem
This option enables additional AT commands for using the MQTT client service.

CONFIG_SM_MQTTC_MESSAGE_BUFFER_LEN - Size of the buffer for the MQTT library
This option specifies the maximum message size which can be transmitted or received through MQTT (excluding PUBLISH payload). The default value is 512, meaning 512 bytes for TX and RX, respectively.

CONFIG_SM_HTTPC - HTTP client support in Serial Modem
This option enables the HTTP client AT commands for making HTTP/HTTPS requests. See *HTTP client AT commands* for more information.

When enabled, the following sub-option is available:

CONFIG_SM_HTTPC_RESPONSE_TIMEOUT_MS - HTTP client idle timeout (ms)
Idle timeout for an active HTTP request in milliseconds. The timer is a sliding window that resets each time data is sent or received (request headers, body upload chunks, and response body chunks). If no activity occurs within this window the request is aborted, and `#XHTTPCSTAT: <fd>,-1,<bytes>` is reported. The default value is 30000 (30 seconds).

CONFIG_SM_UART_RX_BUF_COUNT - Receive buffers for UART.
This option defines the number of buffers for receiving (RX) UART traffic. The default value is 3.

CONFIG_SM_UART_RX_BUF_SIZE - Receive buffer size for UART.
This option defines the size of a single buffer for receiving (RX) UART traffic. The default value is 256.

CONFIG_SM_UART_TX_BUF_SIZE - Send buffer size for UART.
This option defines the size of the buffer for sending (TX) UART traffic. The default value is 256.

CONFIG_SM_URC_BUFFER_SIZE - URC buffer size.
Buffer, in which unsolicited result codes (URC) are stored before being sent to the host. The buffer has to be large enough to hold the largest URC that can be sent by the modem. Result codes longer than this size will get dropped. The default and maximum value is 4096.

CONFIG_SM_PPP_FALLBACK_MTU - Control the MTU used by PPP.
This option controls the MTU used by PPP. PPP tries to retrieve the cellular link MTU from the modem (using `AT+CGCONTRDP`). If MTU is not returned by the modem, this value will be used as a fallback. The MTU will be used for sending and receiving data on both the PPP and cellular links. The default value is 1280.

CONFIG_SM_CARRIER_AUTO_STARTUP - Enable automatic startup on boot.
This option enables automatic startup of the library on device boot. If this configuration is disabled, automatic startup is controlled through a dedicated AT command.

CONFIG_SM_PGPS_INJECT_FIX_DATA - Injects the data obtained when acquiring a fix.
This option, if enabled, when a fix is acquired the current location and time will be passed to the P-GPS subsystem. It allows you to speed up the time it takes to acquire the next fix when A-GNSS is disabled or unavailable. This can be detrimental to short TTFF if the device is expected to move distances longer than a few dozen kilometers between fix attempts. In that case, this option should be disabled. The default value is y.

CONFIG_SM_DFU_MODEM_FULL - Enable full modem DFU
This option enables support for full modem firmware updates using the

AT#XDFUINIT, AT#XDFUWRITE, and AT#XDFUAPPLY commands. See the *DFU AT commands* for more information.

CONFIG_SM_DEBUG_STATS_HEAP - Enable heap memory statistics
This option enables the AT#XDBGSTATSMEM command for printing heap memory statistics. See the *Memory statistics #XDBGSTATSMEM* for more information.

CONFIG_SM_DEBUG_STATS_THREAD_STACK - Enable thread stack memory statistics
This option enables thread stack memory statistics in the AT#XDBGSTATSMEM command. See the *Memory statistics #XDBGSTATSMEM* for more information.

2.3.2 Configuration files

You can find the configuration files in the app directory.

Configuration files have a .conf or .overlay extension for Kconfig or devicetree overlays, respectively. Board-specific configuration files are named <BOARD> with a .conf or .overlay extension and are located in the boards directory. When the name of the board-specific configuration file matches the board target, the overlay is automatically included by the build system.

See Build and configuration system for more information on the nRF Connect SDK configuration system.

Important:

When adding Kconfig fragments and devicetree overlays, make sure to use the **EXTRA_CONF_FILE** and **EXTRA_DTC_OVERLAY_FILE** CMake options, respectively. Otherwise, if **CONF_FILE** or **DTC_OVERLAY_FILE** is used, all the configuration files that normally get picked up automatically will have to be included explicitly.

The following configuration files are provided:

- prj.conf - This configuration file contains the standard configuration for the Serial Modem application and is included by default by the build system.
- external-mcu.overlay - This configures the Serial Modem application to communicate with external MCU over uart2, using specific pins for UART, DTR, and RI. The overlay is pin compatible with nRF9151DK. For other setups, you can customize the overlay to fit your configuration.
- nrf91m1.overlay - Devicetree overlay for use with the nRF91M1 SiP. Configures uart0 and uart1 with pins suitable for nRF9151 DK usage and for connecting to an external MCU. See *nRF91M1 pre-programmed Serial Modem application* for pin details.
- interposer-nrf9151-mk-bg95.overlay - Devicetree overlay for an nRF9151 SiP on an nRF9151-MK-BG95 interposer board. Configures the uart0 pins to match the interposer footprint and removes hw-flow-control, as the interposer does not use the RTS and CTS signals for flow control. See *Interposer boards* for pin details.
- interposer-nrf9151-mk-gm02s.overlay - Devicetree overlay for an nRF9151 SiP on an nRF9151-MK-GM02S interposer board. Configures the uart0 pins to match the interposer footprint and moves the DTR pin off the UART0 TX pin. See *Interposer boards* for pin details.

- `carrier.conf` - Configuration file that adds nRF Connect SDK LwM2M carrier support. Used in conjunction with `carrier-nrf9151.overlay`. See *Using the LwM2M carrier library* for more information on how to connect to an operator's device management platform.
- `carrier-nrf9151.overlay` - Devicetree overlay that adds a dedicated flash partition for the LwM2M carrier library NVS storage. Used in conjunction with `carrier.conf`. It does not reserve a `memfault_coredump_partition`, so the Memfault coredump falls back to RAM in this configuration.
- `carrier-softbank.conf` and `sysbuild-softbank.conf` - Configuration files that add SoftBank configurations needed by the carrier library. Used in conjunction with `carrier.conf` and `carrier-nrf9151.overlay`. For more information, see the Carrier-specific dependencies section of the LwM2M carrier documentation.
- `carrier-lgu.conf` - This configuration file adds LG U+ configurations needed by the carrier library. Used in conjunction with `carrier.conf` and `carrier-nrf9151.overlay`. For more information, see the Carrier-specific dependencies section of the LwM2M carrier documentation.
- `cmux.conf` - Configuration file that adds support for the CMUX protocol. See *CMUX AT commands* for more information.
- `ppp.conf` - Configuration file that adds support for the Point-to-Point Protocol (PPP). This disables most of the IP-based protocols available through AT commands (such as MQTT) as it is expected that the controlling chip's own IP stack is used instead. See *CONFIG_SM_PPP* and *PPP AT commands* for more information.
- `trace-backend-cmux.conf` - Configuration file that enables CMUX modem trace backend. When enabled, modem traces are transmitted on a dedicated CMUX channel. Must be combined with `trace-backend-nrf9151.overlay`. See the *Modem traces through CMUX* documentation for more information.
- `trace-backend-uart.conf` - Configuration file that enables the UART modem trace backend. Backend starts disabled and is activated at runtime using AT#XTRACE. Must be combined with `trace-backend-nrf9151.overlay`. See the *Modem traces through UART* documentation for more information.
- `trace-backend-nrf9151.overlay` - Devicetree overlay that defines the SRAM partition used by the modem trace backend. Required when using `trace-backend-uart.conf` or `trace-backend-cmux.conf`.
- `full-fota.conf` - Configuration file that adds full modem FOTA support. Must be combined with `full-fota-nrf9151.overlay`. See *FOTA AT commands* for more information on how to use full modem FOTA functionality.
- `full-fota-nrf9151.overlay` - Devicetree overlay that adds a dedicated flash partition for modem firmware storage in external flash. Must be combined with `full-fota.conf`.
- `pgps-nrf9151.overlay` - Devicetree overlay that adds a dedicated flash partition for P-GPS prediction data storage. The overlay resizes the MCUboot primary and secondary slots to free up space for the `pgps_partition`. Required when `CONFIG_NRF_CLOUD_PGPS` is enabled. Must also be passed to the `mcuboot` image using **`mcuboot_EXTRA_DTC_OVERLAY_FILE`** so that MCUboot operates with the same partition layout. The absolute path to the overlay file must be provided. It does not

reserve a `memfault_coredump_partition`, so the Memfault coredump falls back to RAM in this configuration. See *GNSS AT commands* for more information.

- `disable-b0-nrf9151.overlay` - Devicetree overlay for a build configuration without the NSIB (B0) immutable bootloader. This is intended only for upgrading v1.x.x devices to v2.x.x firmware. Must be combined with `SB_CONFIG_SECURE_BOOT_APPCORE=n` in the sysbuild configuration and passed to the `mcuboot` image using **`mcuboot_EXTRA_DTC_OVERLAY_FILE`** (using the absolute path). It does not reserve a `memfault_coredump_partition`, so the Memfault coredump falls back to RAM in this configuration.
- `disable-dtr.overlay` - Devicetree overlay that disables the DTR and RI pins and related functionality. This overlay can be used if your setup does not have the need or means for managing the power externally. Modify the overlay to fit your configuration.
- `nrf-device-provisioning.conf` - Configuration file that enables the nRF Cloud device provisioning client and its AT command interface. See *nRF Provisioning AT commands* for more information.

The board-specific devicetree overlays (`boards/*.overlay`) set up configurations that are specific to each supported development kit. All of them configure the DTR to be deasserted from a button and RI to drive an LED.

2.3.3 Flash and SRAM partition layouts

The Serial Modem application uses DTS-defined partitions instead of the nRF Connect SDK Partition Manager. The base partition layout for the nRF9151 DK is defined in `boards/nrf9151dk_nrf9151_ns.overlay`. Several DTS overlay files extend or replace parts of the layout for specific build configurations.

2.4 Cellular PPP modem

2.4.1 Overview

Serial Modem can provide a Point-to-Point Protocol (PPP) link over a UART so that a host can use the nRF9151 SiP as a cellular dial-up modem. This is typically used in one of the following ways:

- **Linux host** - The host runs a standard PPP daemon (`pppd`) and establishes a PPP network interface (`ppp0`). Optionally, you can use CMUX to keep an AT command channel available while PPP is running.
- **MCU host (controlling chip)** - The host runs embedded RTOS, for example, Zephyr, and uses its cellular modem driver. The host application uses the IP stack on the RTOS, while Serial Modem provides the modem control, CMUX, and PPP transport.

PPP and CMUX are controlled through AT commands. For details, see *PPP AT commands* and *CMUX AT commands*.

2.4.2 PPP and CMUX

Serial Modem supports PPP both with or without CMUX (GSM 07.10 multiplexing). This allows you to choose the setup that best fits your use case. The recommended setup

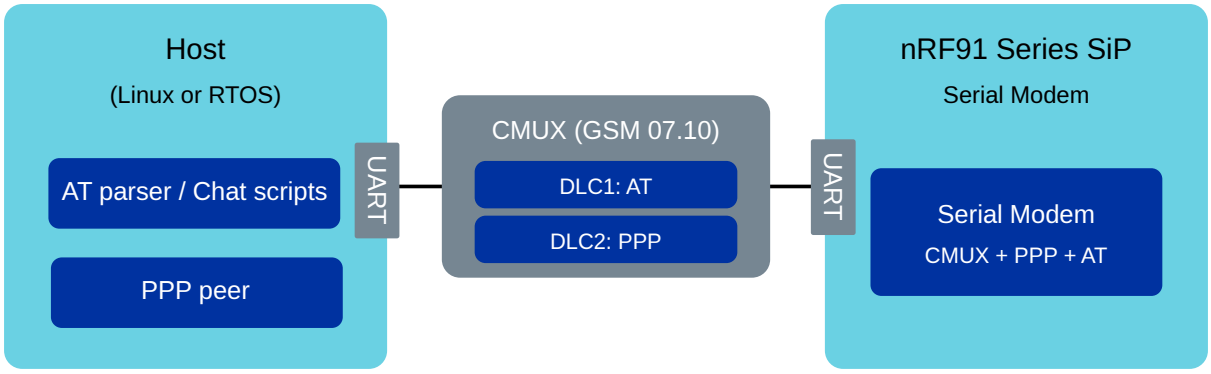


Fig. 1: CMUX multiplexes AT and PPP over one UART using separate DLC channels.

for most modem use cases is to use CMUX, as it keeps AT control available while PPP is running.

PPP without CMUX

Without CMUX, PPP runs on the current UART. Send `AT+CFUN=1` to activate the modem, then `AT+CGDATA` to start PPP. The modem responds with `CONNECT` and the UART immediately enters PPP data mode. You cannot use the UART for AT commands until PPP is terminated by LCP negotiation. No AT notifications are delivered over the UART while PPP is active.

Example Switching UART from AT mode to PPP mode and back:

PPP with CMUX

With CMUX, AT traffic and PPP traffic can share a single UART by using separate CMUX channels.

Start CMUX with `AT+CMUX=0`, then issue `AT+CGDATA` on the chosen DLC channel to start PPP on that channel. The following example uses DLC channel 1 for PPP and DLC channel 2 for AT commands, with `AT#XCMUXURC=2` directing URCs to DLC channel 2.

Example Send `AT#XCMUXURC=2` to route URCs to DLC channel 2, start CMUX with `AT+CMUX=0`, then start PPP on DLC channel 1 using `AT+CGDATA`:

Note:

The `AT#XCMUXURC=2` is optional, but it is recommended to route URCs to a static channel. When the URC channel is not selected, URCs are sent to the first open AT channel, which is not in data mode.

Note:

The sequence diagrams in this section show `+CEREG:` and `+CGEV:` URCs on the AT channel for illustration. By default, Serial Modem does not forward these URCs to the host.

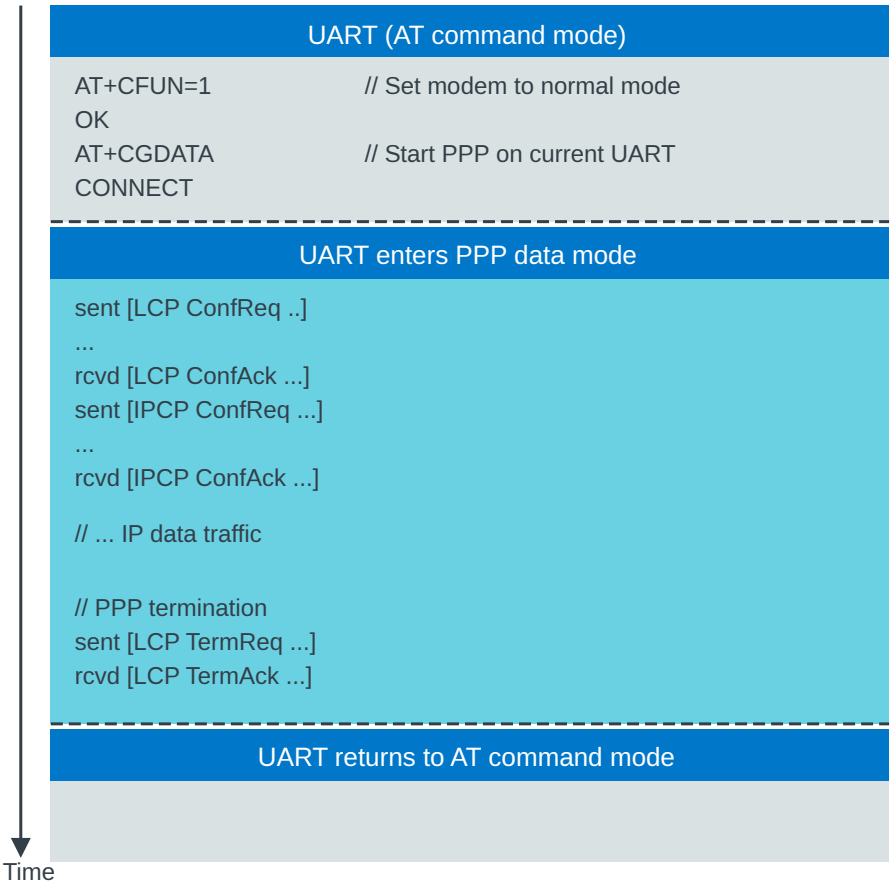


Fig. 2: PPP session without CMUX.

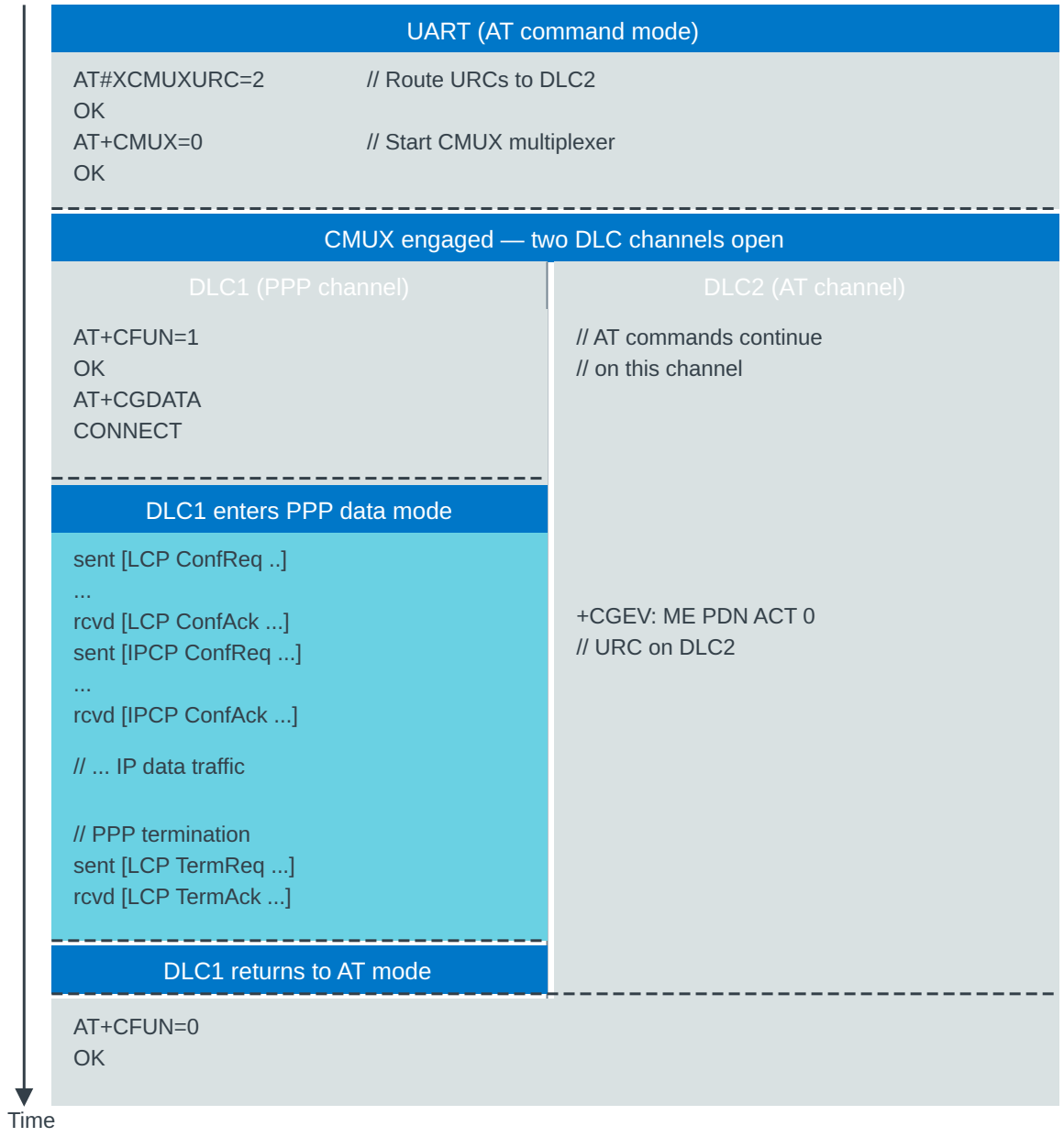


Fig. 3: PPP on DLC1, AT commands and URCs on DLC2.

2.4.3 Use case: Linux host

This section describes how to use an nRF9151 SiP running Serial Modem as a PPP modem for a Linux device.

Prerequisites

Install the following packages on the Linux host:

- `pppd` (PPP daemon)
- `ldattach` (from util-linux, needed only for the CMUX-based setup)

These packages are available on common Linux distributions.

Build and flash Serial Modem

Build and program Serial Modem with the PPP overlay enabled. For the CMUX-based setup, enable both PPP and CMUX:

- `ppp.conf`
- `cmux.conf`

If you change the UART baud rate in Serial Modem's devicetree overlay, set the same baud rate in the host scripts or `pppd` command line.

Note:

When using the standard Linux `ldattach` utility, CMUX MRU/MTU is set to 127 bytes and cannot be changed. In that case, make sure Serial Modem uses `CONFIG_MODEM_CMUX_MTU=127` (this is already configured in `cmux.conf`).

Option A: PPP over CMUX (recommended)

Serial Modem provides start and stop scripts for Linux in the application's `scripts` directory. The `sm2_start_ppp.sh` script uses standard 3GPP AT commands (`AT+CMUX` and `AT+CGDATA`) and is the recommended way to establish a PPP connection over CMUX. If needed, adjust the serial port settings with command line parameters. Run `sm2_start_ppp.sh -h` to see all available options. Default settings assume that the modem is available at `/dev/ttyACM0` with a baud rate of `115200`.

1. Start the connection (requires superuser privileges for PPPD and CMUX):

```
$ sudo scripts/sm2_start_ppp.sh [-s serial_port] [-b baud_rate]
[-t timeout]
```

2. Verify that the PPP interface is up (for example, with `ip addr show ppp0`) and test connectivity.
3. You may observe log messages similar to the following in `/var/log/syslog`:

```
pppd[xxxx]: sent [LCP ConfReq id=0x1 <options>]
pppd[xxxx]: rcvd [LCP ConfAck id=0x1 <options>]
...
pppd[xxxx]: local  IP address <IP_address>
pppd[xxxx]: remote IP address <IP_address>
```

You can now use the PPP connection for network communication.

4. Stop the connection:

```
$ sudo poff

# OR

$ sudo scripts/sm_stop_ppp.sh
```

Note:

The scripts do not manage DNS settings. Consult your distribution documentation if you need to configure DNS (for example, by using `usepeerdns` with `pppd` or by integrating with Network Manager or systemd-resolved).

Option B: PPP without CMUX (simple, but no AT control)

You can also run PPP directly on the UART without CMUX. This setup is useful for quick testing, but the UART cannot be used for AT commands while PPP is active.

```
$ sudo pppd noauth <UART_dev> <baud_rate> local crtscts
debug noipdefault connect "/usr/sbin/chat -v -t60 ''
AT+CFUN=1 OK AT+CGDATA CONNECT" nodetach
```

Replace `<UART_dev>` by the device file assigned to the Serial Modem's UART and `<baud_rate>` by the baud rate of the UART. Typically, the device file assigned to it is `/dev/ttyACM0` for an nRF9151 DK. To run PPPD in background, remove the `nodetach` option and observe the logs in `/var/log/syslog`.

1. After the PPP link negotiation has completed successfully, you should see log messages similar to the following:

```
sent [LCP ConfReq id=0x1 <options>]
rcvd [LCP ConfAck id=0x1 <options>]
...
local  IP address <IP_address>
remote IP address <IP_address>
```

You can now use the PPP connection for network communication.

2. Terminate the PPP connection with CTRL+C in the terminal where `pppd` is running or `sudo poff` in another terminal.

Option C: PPP over CMUX with NTN support

When using NTN (Non-Terrestrial Network) cellular profiles, PPP cannot function while the modem is in NTN mode. It must be stopped and restarted on each mode transition between TN (Terrestrial Network) and NTN.

Option A manages CMUX and PPP as a single lifecycle: stopping PPP also issues `AT+CFUN=0` and tears down CMUX. For NTN scenarios, use the separate CMUX management scripts so that CMUX stays alive across PPP restarts:

- `sm2_start_cmux.sh` - Starts CMUX only, then exits. The `gsmtty` channels remain open.
- `sm2_stop_cmux.sh` - Stops CMUX and releases the serial port.
- `sm2_start_ppp.sh` with the `-C` flag - attaches PPP to an already-running CMUX (DLC 1). When PPP stops, the modem and CMUX are left untouched.

When PPP is not running, all CMUX channels (`gsmtty1` and `gsmtty2`) are available for AT commands. This allows the host to reconfigure the modem (for example, switch profiles) through the AT channel while PPP is inactive.

Note:

The `-C` flag prevents `AT+CFUN=0` and `AT#XCMUXCLD` from being issued when PPP stops, so the modem and CMUX remain active.

Workflow:

1. Start CMUX (once, before the first PPP session):

```
$ sudo scripts/sm2_start_cmux.sh [-s serial_port] [-b baud_rate]
[-B new_speed]
```

2. Start PPP in TN mode:

```
$ sudo scripts/sm2_start_ppp.sh -C [-s serial_port] [-a APN]
```

3. When switching to NTN mode, stop PPP without tearing down CMUX:

```
$ sudo poff
```

   After PPP stops, both `gsmtty1` and `gsmtty2` are available for AT commands. Use them to reconfigure the modem or monitor URCs while in NTN mode.

4. When returning to TN mode, start PPP again:

```
$ sudo scripts/sm2_start_ppp.sh -C [-s serial_port] [-a APN]
```

5. When done with all PPP sessions, stop CMUX:

```
$ sudo scripts/sm2_stop_cmux.sh [-s serial_port]
```

2.4.4 Use case: Zephyr host

This section describes how to use Serial Modem as a modem that is controlled by Zephyr's cellular modem driver. The host runs a Zephyr application, which uses Zephyr's native IP stack.

Note:

Zephyr's cellular modem driver support has some limitations when used with an nRF9151 SiP:

- GNSS functionality is not available through the driver.
- eDRX or PSM configuration through the driver is not supported. Requires manual changes of the modem settings through AT commands.
- Power saving feature requires UART with DTR and RI pins connected between the host and the nRF9151 SiP. See *UART configuration* for more information.
- System mode is not configured. See the %XSYSTEMMODE command in the AT command Reference Guide for more details.

User pipes for AT commands

Zephyr's cellular modem driver provides **user pipes** for sending AT commands to the modem while PPP is active. User pipes are additional CMUX channels (DLC channels 3 and 4) that the host application can use for modem configuration, DFU/FOTA, or other AT command operations.

User pipe availability User pipes are available only when the modem driver has all CMUX channels open. This occurs after the following conditions are met:

- CMUX has been started with `AT+CMUX=0`.
- The network interface is activated with `net_if_up()`, which opens the CMUX channels, connects to the network, and establishes the PPP connection.

When the network interface is deactivated with `net_if_down()`, the PPP connection is closed and user pipes become unavailable.

Important:

User pipes are not available when PPP is closed. The modem driver is designed so that `net_if_up()` opens all CMUX channels, including user pipes, and `net_if_down()`

closes them.

The DLC channel callback notifies the host application when a user pipe opens or closes.

Use cases for user pipes User pipes allow the host application to send AT commands while maintaining an active PPP connection. Common use cases include:

- Configuring eDRX and PSM settings (`AT+CEDRXS`, `AT+CPSMS`).
- Performing modem firmware updates using `AT#XFOTA` or DFU commands.
- Querying modem status and diagnostics.
- Managing SMS or other modem features not exposed through the PPP interface.
- Temporarily pausing PPP for maintenance operations.

Note:

When using the `nordic,nrf91-sm-v2` cellular driver, you can temporarily pause the PPP connection by issuing `AT+CFUN=4` on a user pipe. This command puts the modem into flight mode, which causes the cellular modem driver to wait for a network registration URC message. The PPP connection remains paused until you restore normal operation with `AT+CFUN=1`, at which point the modem re-registers with the network and the PPP connection resumes. This is useful for performing maintenance operations that require the modem to be offline temporarily.

Example

The Zephyr modem driver sends `AT+CMUX=0` to start the CMUX, then uses DLC channel 1 for AT commands and DLC channel 2 for PPP. User pipes on DLC channels 3 and 4 become available for host application AT commands once the network interface is up. The dial script issues `AT+CGDATA` on DLC channel 2, which starts PPP on that channel and leaves DLC channel 1 free for AT commands and URCs.

Setup

Both sides must be compiled with matching configuration:

nRF9151 SiP running Serial Modem Include the following configuration overlays:

- `cmux.conf` - Enable CMUX.
- `ppp.conf` - Enable PPP.
- `external-mcu.overlay` - Configure UART (pins, baud rate), DTR, and RI pins used between the MCU and the nRF9151 SiP.

Host running Zephyr Start from the configuration files in the *PPP shell sample* sample. In particular:

- `prj.conf` enables the required Zephyr subsystems.

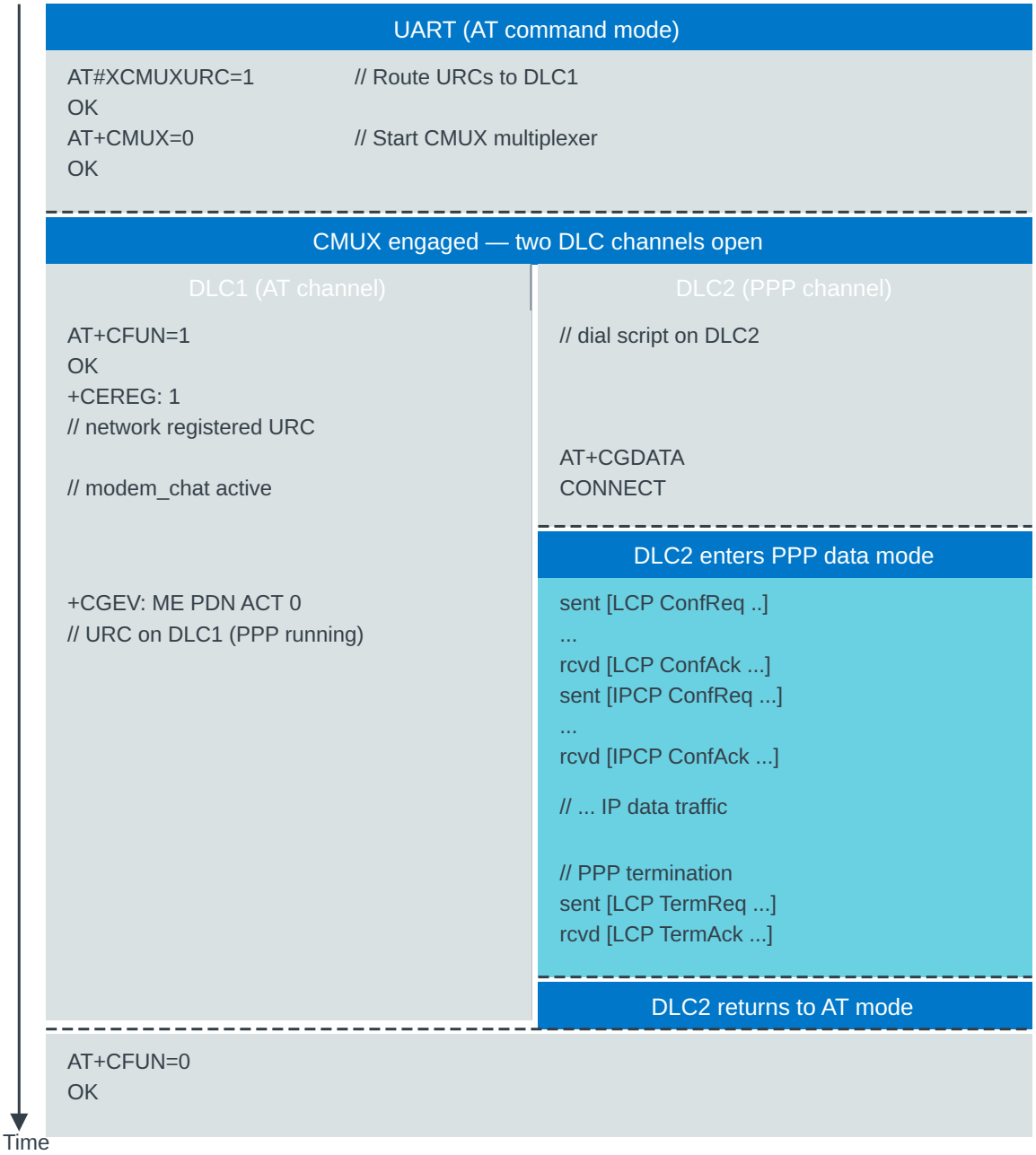


Fig. 4: Zephyr modem driver uses DLC1 as the AT command channel and starts PPP on DLC2 using AT+CGDATA.

- `boards/nrf54l15dk_nrf54l15_cpuapp.conf.conf` Board-specific configuration.
- `boards/nrf54l15dk_nrf54l15_cpuapp.overlay` - Configure UART (pins, baud rate), DTR and RI pins used between the MCU and the nRF9151 SiP.

Depending on your hardware, a devicetree overlay is needed to describe the modem UART and power control pins. The modem node in the overlay must use `compatible = "nordic,nrf91-sm-v2"` to select the Serial Modem modem driver:

```
&uart0 {
    status = "okay";
    current-speed = <115200>;
    hw-flow-control;

    modem: modem {
        compatible = "nordic,nrf91-sm-v2";
        status = "okay";
        mdm-ring-gpios = <&gpio0 0 (GPIO_ACTIVE_LOW | GPIO_PULL_UP)>
;
        mdm-dtr-gpios  = <&gpio0 1 GPIO_ACTIVE_LOW>;
        cmux-enable-runtime-power-save;
        cmux-close-pipe-on-power-save;
        cmux-idle-timeout-ms = <5000>;
        autostarts;
    };
};
```

The `nordic,nrf91-sm-v2` binding is compatible with Serial Modem application v2.0.0 and later. Devicetree settings also control the CMUX power-saving behavior.

Before using a Zephyr host to control the nRF9151 SiP, make sure the modem system mode is set to your desired configuration including PSM and eDRX settings. When the Zephyr's modem driver starts up, it will perform the following steps:

- Enable the UART interface.
- Enable registration URC forwarding with `AT+CEREG=1` and query modem information.
- Configure URCs to DLC channel 1 with `AT#XCMUXURC=1`.
- Enable CMUX with `AT+CMUX=0`.
- Set the modem to normal mode with `AT+CFUN=1`.
- Start PPP on DLC channel 2 with `AT+CGDATA`.

The Zephyr host can then use standard Zephyr networking APIs to use the PPP link for network communication. Only exception for normal network interfaces is that the PPP interface cannot be automatically started at boot. You must start the PPP interface manually when required.

```
struct net_if *ppp_iface = net_if_get_default();
net_if_up(ppp_iface);
```

Using legacy nrf91-slm Zephyr driver

Zephyr provides an older upstream cellular modem driver that uses `compatible = "nordic,nrf91-slm"`. It was developed for the Serial LTE Modem (SLM) sample application and is not recommended for new designs.

Unlike the `nordic,nrf91-sm-v2` driver, the legacy `nrf91-slm` driver uses the proprietary `AT#XCMUX=2` command to switch the AT command channel to DLC channel 2 at runtime and `AT#XPPP=1` to allow PPP to start automatically on the secondary DLC channel. This approach is unreliable in various recovery scenarios, because PPP startup and AT channel switching might run out of sync, leading to a loss of AT control.

2.4.5 Operational notes

- The nRF9151 SiP automatically activates the default PDP context. Manually configuring the APN is typically not required.
- The nRF9151 SiP automatically restarts a network scan when connectivity is lost. Manual dial-up scripts are typically not required for LTE connection recovery.
- When you use the proprietary `AT#XCMUX=1` and `AT#XPPP=1` commands, CMUX channel allocation is static (by default, allocating DLC channel 1 to PPP and DLC channel 2 to AT). See *CMUX AT commands* and *PPP AT commands* for details.
- If you enable packet domain URC forwarding with `AT+CGEREP=1`, you might see a packet domain event (`+CGEV: IPV6 FAIL 0`). This event indicates a failure to obtain an IPv6 address. This is normal and you can ignore it, because the modem does not require an IPv6 address when you use PPP. The host's IP stack handles IPv6 addressing.

2.5 Releasing Serial Modem

Before each official release or pre-release, update the following version metadata. Each published application image must carry a higher MCUboot sign version than the previous one on the same product line. Increment `CONFIG_FW_INFO_FIRMWARE_VERSION` only when the MCUboot bootloader image changes.

2.5.1 Application - app/VERSION

`app/VERSION` is the single source of truth for:

- The `major.minor.patchlevel` prefix in `AT#XSMVER <sm_version>` (Git-describe style: `v` + that triplet + optional `-N-g<hash>` from Git).
- The MCUboot application sign version (`CONFIG_MCUBOOT_IMGTOOL_SIGN_VERSION`, `major.minor.patchlevel+VERSION_TWEAK`), used for application downgrade prevention during DFU.

See Zephyr application VERSION.

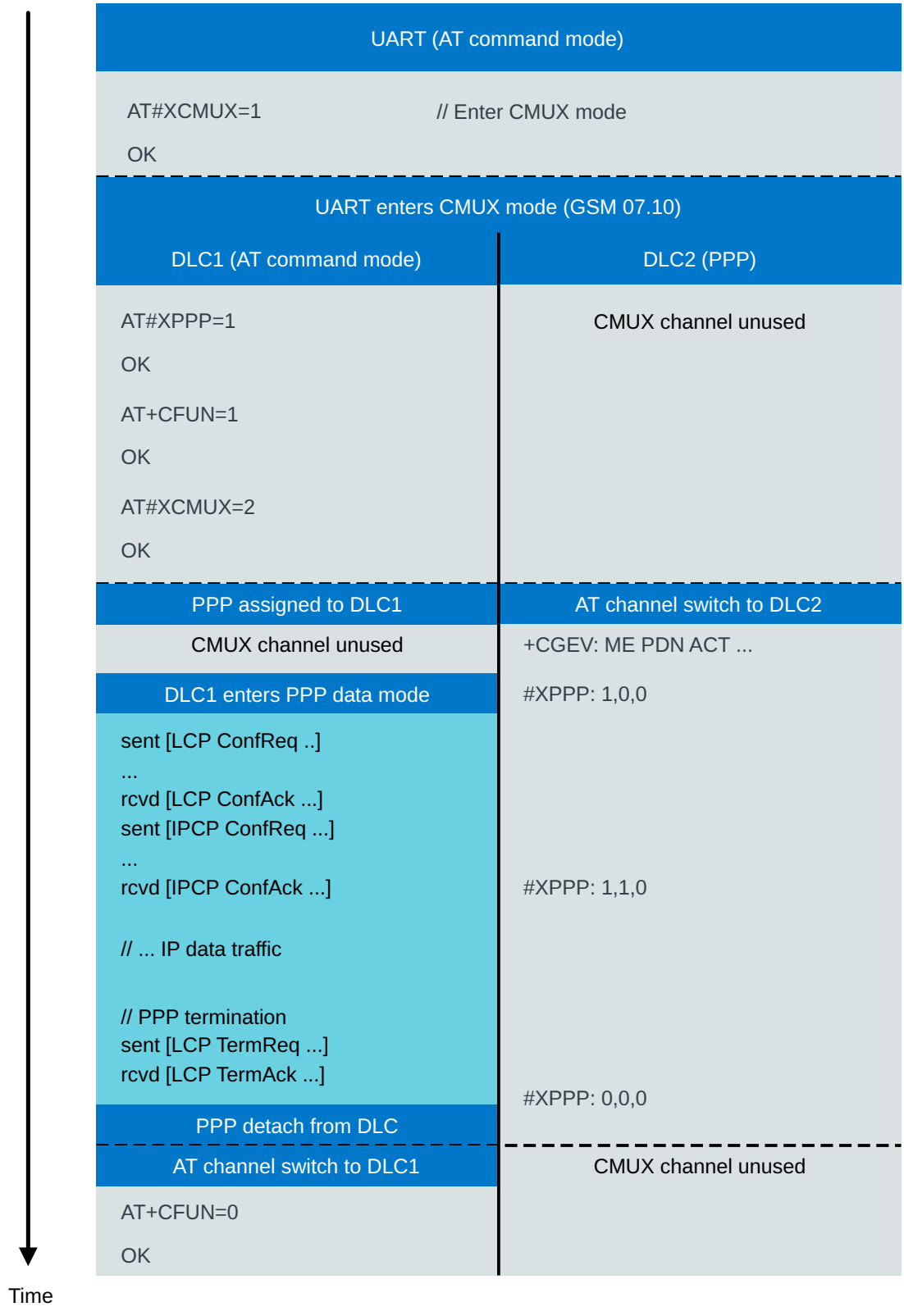


Fig. 5: Legacy `nrf91-slm` driver starts CMUX using `AT#XCMUX=1`, then uses `AT#XCMUX=2` to switch the AT channel to DLC2 and starts PPP on DLC1.

DFU and AT#XSMVER

For application firmware update, a host can read AT#XSMVER before and after DFU and compare the `major.minor.patchlevel` prefix of `<sm_version>` (the part from `app/VERSION`, before any `-N-g<hash>` Git suffix). That value must increase when moving to a newer published image. MCUboot enforces the same ordering using the sign version derived from `app/VERSION`. The Git suffix is informational only and is not used for downgrade checks.

Pre-release numbering

Before General Availability (GA) for a target `X.Y.Z`, use a staging line below `X.Y.Z` and increment `PATCHLEVEL` for each published pre-release. Keep `VERSION_TWEAK = 0` unless you have a specific reason to use it after GA.

v2.0.0 GA The first v2.0.0 pre-release uses `1.99.0`, the second `1.99.1`, and so on. At GA, set the product version to `2.0.0`.

```
# First v2.0.0 pre-release
VERSION_MAJOR = 1
VERSION_MINOR = 99
PATCHLEVEL = 0
VERSION_TWEAK = 0
```

```
# Second v2.0.0 pre-release
VERSION_MAJOR = 1
VERSION_MINOR = 99
PATCHLEVEL = 1
VERSION_TWEAK = 0
```

```
# v2.0.0 release
VERSION_MAJOR = 2
VERSION_MINOR = 0
PATCHLEVEL = 0
VERSION_TWEAK = 0
```

`2.0.0+0` must be greater than every shipped `1.99.N+0` pre-release.

Future v2.1.0 GA Pre-releases before v2.1.0 GA, use the `2.0.99` line (for example `2.0.99`, then `2.0.100`), incrementing `PATCHLEVEL` for each published pre-release, following the same pattern as `1.99.0`, `1.99.1` before v2.0.0 GA. At v2.1.0 GA, set `VERSION_MAJOR`, `VERSION_MINOR`, and `PATCHLEVEL` to 2, 1, and 0.

General Availability and later

The following guidelines describe how to manage versioning for General Availability and later:

- Set the product version and `VERSION_TWEAK = 0` at GA.
- After GA, increment `PATCHLEVEL` for each new published application build on that line.
- Do not update version fields for unpublished local or CI builds.
- Record the mapping between Git tag and `app/VERSION` in the release notes,
- Confirm the version with `AT#XSMVER`.

2.5.2 MCUboot — `CONFIG_FW_INFO_FIRMWARE_VERSION`

In `app/sysbuild/mcuboot/prj.conf`, increment the value only when the MCUboot bootloader image changes. The value is used by NSIB (B0) for MCUboot slot downgrade prevention and is independent of `app/VERSION`.

After an MCUboot update, `AT#XBOOTINFO=0` reports the active slot value.

See *Release notes* for more information.

2.6 Extending the application

The AT commands in the Serial Modem are implemented with the Custom AT commands library. You can extend the Serial Modem application by adding your own AT commands.

2.6.1 Adding AT commands

If you want to implement a custom AT command in Serial Modem, you will need to follow the instructions in the Custom AT commands library documentation. Setting the AT command filter and callback with the `AT_CMD_CUSTOM` macro is enough for a custom AT command that does not require setup or teardown.

As a preferred alternative to the `AT_CMD_CUSTOM` macro, Serial Modem defines the `SM_AT_CMD_CUSTOM` macro, which is a wrapper around the `AT_CMD_CUSTOM` macro. The `SM_AT_CMD_CUSTOM` macro pre-processes the AT command parameters for the AT command callback and sets the default `OK` response if the callback returns successfully.

If your custom AT command requires setup or teardown, you will need to perform the following steps:

1. Create a header file in `app/src/`. The file must expose the following function declarations:
   - `*_init()` - Your setup function for the custom AT command.
   - `*_uninit()` - Your teardown function for the custom AT command.
2. Create a corresponding `.c` file in `app/src/`. The file must implement the following functions:
   - Your AT command callback, which is declared with the `AT_CMD_CUSTOM` macro.
   - `*_init()` - Your setup function for the custom AT command.
   - `*_uninit()` - Your teardown function for the custom AT command. Make sure that the teardown function exits successfully. It will be called before entering power saving states or shutting down.

3. Edit `app/src/sm_at_commands.c` and make the following changes:
   a. Include your header file.
   b. In the `sm_at_init()` function, add a call to your setup function.
   c. In the `sm_at_uninit()` function, add a call to your teardown function.

2.7 Testing scenarios

The following testing scenarios give detailed instructions for testing specific use cases. They list the required AT commands and the expected responses.

Some scenarios are generic and should work out of the box, while others require you to set up a server that you can test against.

See *Building and running* for instructions on how to build and run the Serial Modem application. *Testing* describes how to turn on the modem and conduct the tests.

2.7.1 Generic AT commands

Complete the following steps to test the functionality provided by the *Generic AT commands*:

1. Retrieve the version of the Serial Modem application.

```
AT#XSMVER
#XSMVER: "v1.99.0-67-ga9e396bc3d58","3.3.99"
OK
```

2. Retrieve a list of all supported proprietary AT commands.

```
AT#XCLAC
AT#XSMVER
AT#XSLEEP
AT#XCLAC
AT#XSOCKET
AT#XSOCKETOPT
AT#XBIND
[...]
OK
```

3. Check the supported values for the sleep command, then put the kit in sleep mode.

```
AT#XSLEEP=?
#XSLEEP: (1,2)
OK
```

2.7.2 TCP/UDP AT commands

The following sections show how to test the functionalities provided by the *Socket AT commands*.

Network connection is required for these tests, so insert a SIM card and connect to the network:

```
AT+CFUN=1

OK
```

TCP connection

The following steps assume that you have a TCP echo server available. You can connect to public HTTP servers (port 80), but you cannot send and receive data.

1. Test the TCP connection with a TCP socket:
   a. Check the available values for the AT#XSOCKET command.

      ```
      AT#XSOCKET=?

      #XSOCKET: <handle>,(1,2),(1,2,3),(0,1),<cid>

      OK
      ```

   b. Open a TCP socket, read the information (handle, family, role, type and cid) about the open socket, and set the receive timeout of the open socket to 30 seconds.

      ```
      AT#XSOCKET=1,1,0

      #XSOCKET: 0,1,6

      OK

      AT#XSOCKET?

      #XSOCKET: 0,1,0,1,0

      OK

      AT#XSOCKETOPT=0,1,20,30

      OK
      ```

   c. Replace *example.com* with the hostname or IPv4 address of the TCP echo server, and *1234* with the corresponding port. 1 indicates that the connection is established.

      ```
      AT#XCONNECT=0,"example.com",1234

      #XCONNECT: 0,1

      OK
      ```

d. Send plaintext data to the TCP server and retrieve the response.

```
AT#XSEND=0,0,0,"Test TCP"

#XSEND: 0,0,8

OK

AT#XRECV=0,0,0,0

#XRECV: 0,0,8
Test TCP
OK
```

e. Close the socket and confirm its state.

```
AT#XCLOSE=0

#XCLOSE: 0,0

OK

AT#XSOCKET?

OK
```

UDP connection

The following steps assume that you have a UDP echo server available.

1. Test the UDP connection with a UDP socket using `AT#XSENDTO`:

   a. Open a UDP socket and read the information (handle, family, role, type and cid) about the open socket.

```
AT#XSOCKET=1,2,0

#XSOCKET: 0,2,17

OK

AT#XSOCKET?

#XSOCKET: 0,1,0,2,0

OK
```

   b. Send plaintext data to a UDP echo server on a specified port. Replace *example.com* with the hostname or IPv4 address of a UDP server, and *1234* with the corresponding port. Then retrieve the response.

```
AT#XSENDTO=0,0,0,"example.com",1234,"Test UDP"

#XSENDTO: 0,0,8

OK

AT#XRECVFROM=0,0,0,0

#XRECVFROM: 0,0,8,"<IP address>",<port>
Test UDP
OK
```

c. Close the socket.

```
AT#XCLOSE=0

#XCLOSE: 0,0

OK
```

2. Test the UDP connection with a UDP socket, using `AT#XCONNECT`:

   a. Open a UDP socket and set connection to UDP server. Replace *example.com* with the hostname or IPv4 address of a UDP server, and *1234* with the corresponding port.

```
AT#XSOCKET=1,2,0

#XSOCKET: 0,2,17

OK

AT#XCONNECT=0,"example.com",1234

#XCONNECT: 0,1

OK
```

   b. Send plaintext data to the UDP server and retrieve the response.

```
AT#XSEND=0,0,0,"Test UDP"

#XSEND: 0,0,8

OK

AT#XRECV=0,0,0,0

#XRECV: 0,0,8
Test UDP
```

```
OK
```

c. Close the socket.

```
AT#XCLOSE=0

#XCLOSE: 0,0

OK
```

TLS connection

The following steps assume that you have a TLS echo server available. You can connect to public HTTPS servers (port 443), but you cannot send and receive the data.

A TLS connection requires a valid certificate for the TLS server.

Update your TLS (root) certificate in PEM format with your selected security tag (in this example, 1000), and start the modem:

Note:

Sending multi-line text to Serial Modem requires the terminal to be configured to use <CR><LF> as the line ending.

```
AT+CFUN=0

OK

AT%CMNG=0,1000,0,"-----BEGIN CERTIFICATE-----
MIIFazCCA1OgAwIBAgIRAIIQz7DSQONZRGPgu2OCiwAwDQYJKoZIhvcN
AQELBQAwTzELMAkGA1UEBhMCVVMxKTAnBgNVBAoTIEludGVybmV0IFNl
Y3VyaXR5IFJlc2VhcmNoIEdyb3VwMRUwEwYDVQQDEwxJU1JHIFJvb3Qg
WDEwHhcNMTUwNjA0MTEwNDM4WhcNMzUwNjA0MTEwNDM4WjBPMQswCQYD
VQQGEwJVUzEpMCcGA1UEChMgSW50ZXJuZXQgU2VjdXJpdHkgUmVzZWFy
Y2ggR3JvdXAxFTATBgNVBAMTDElTUkcgUm9vdCBYMTCCAiIwDQYJKoZI
hvcNAQEBBQADggIPADCCAgoCggIBAK3oJHP0FDfzm54rVygch77ct984
kIxuPOZXoHj3dcKi/vVqbvYATyjb3miGbESTtrFj/RQSa78f0uoxmyF+
0TM8ukj13Xnfs7j/EvEhmkvBioZxaUpmZmyPfjxwv60pIgbz5MDmgK7i
S4+3mX6UA5/TR5d8mUgjU+g4rk8Kb4Mu0UlXjIB0ttov0DiNewNwIRt1
8jA8+o+u3dpjq+sWT8KOEUt+zwvo/7V3LvSye0rgTBIlDHCNAymg4VMk
7BPZ7hm/ELNKjD+Jo2FR3qyHB5T0Y3HsLuJvW5iB4YlcNHlsdu87kGJ5
5tukmi8mxdAQ4Q7e2RCOFvu396j3x+UCB5iPNgiV5+I3lg02dZ77DnKx
HZu8A/lJBdiB3QW0KtZB6awBdpUKD9jf1b0SHzUvKBds0pjBqAlkd25H
N7rOrFleaJ1/ctaJxQZBKT5ZPt0m9STJEadao0xAH0ahmbWnOlFuhjue
fXKnEgV4We0+UXgVCwOPjdAvBbI+e0ocS3MFEvzG6uBQE3xDk3SzynTn
jh8BCNAw1FtxNrQHusEwMFxIt4I7mKZ9YIqioymCzLq9gwQbooMDQaHW
BfEbwrbwqHyGO0aoSCqI3Haadr8faqU9GY/rOPNk3sgrDQoo//fb4hVC
```

```
1CLQJ13hef4Y53CIrU7m2Ys6xt0nUW7/vGT1M0NPAgMBAAGjQjBAMA4G
A1UdDwEB/wQEAwIBBjAPBgNVHRMBAf8EBTADAQH/MB0GA1UdDgQWBBR5
tFnme7bl5AFzgAiIyBpY9umbbjANBgkqhkiG9w0BAQsFAAOCAgEAVR9Y
qbyyqFDQDLHYGmkgJykIrGF1XIpu+ILlaS/V9lZLubhzEFnTIZd+50xx
+7LSYK05qAvqFyFWhfFQDlnrzuBZ6brJFe+GnY+EgPbk6ZGQ3BebYhtF
8GaV0nxvwuo77x/Py9auJ/GpsMiu/X1+mvoiBOv/2X/qkSsisRcOj/KK
NFtY2PwByVS5uCbMiogziUwthDyC3+6WVwW6LLv3xLfHTjuCvjHIInNz
ktHCgKQ5ORAzI4JMPJ+GslWYHb4phowim57iaztXOoJwTdwJx4nLCgdN
bOhdjsnvzqvHu7UrTkXWStAmzOVyyghqpZXjFaH3pO3JLF+l+/+sKAIu
vtd7u+Nxe5AW0wdeRlN8NwdCjNPElpzVmbUq4JUagEiuTDkHzsxHpFKV
K7q4+63SM1N95R1NbdWhscdCb+ZAJzVcoyi3B43njTOQ5yOf+1CceWxG
1bQVs5ZufpsMljq4Ui0/1lvh+wjChP4kqKOJ2qxq4RgqsahDYVvTH9w7
jXbyLeiNdd8XM2w9U/t7y0Ff/9yi0GE44Za4rF2LN9d11TPAmRGunUHB
cnWEvgJBQl9nJEiU0Zsnvgc/ubhPgXRR4Xq37Z0j4r7g1SgEEzwxA57d
emyPxgcYxn/eR44/KJ4EBs+lVDR3veyJm+kXQ99b21/+jh5Xos1AnX5i
ItreGCc=
-----END CERTIFICATE-----"

OK

AT+CFUN=1

OK
```

1. Test the TLS connection with a TLS socket:

   a. Open a TLS socket that uses the security tag 1000 and connect to a TLS server on a specified port. Replace *example.com* with the hostname or IPv4 address of a TLS server and *1234* with the corresponding port.

   ```
   AT#XSSOCKET=1,1,0,1000

   #XSSOCKET: 0,1,258

   OK

   AT#XCONNECT=0,"example.com",1234

   #XCONNECT: 0,1

   OK
   ```

   b. Send plaintext data to the TLS server and retrieve the response.

   ```
   AT#XSEND=0,0,0,"Test TLS"

   #XSEND: 0,0,8

   OK
   ```

```
AT#XRECV=0,0,0,0

#XRECV: 0,0,8
Test TLS
OK
```

c. Close the socket.

```
AT#XCLOSE=0

#XCLOSE: 0,0

OK
```

DTLS connection

The following steps assume that you have a DTLS echo server available with pre-shared key (PSK) authentication.

Update your hex-encoded PSK and the PSK identity to be used for the DTLS connection in the modem, with your selected security tag (in this example, 1001):

```
AT+CFUN=0

OK

AT%CMNG=0,1001,3,"6e7266393174657374"

OK

AT%CMNG=0,1001,4,"nrf91test"

OK

AT+CFUN=1

OK
```

1. Test the DTLS connection with a DTLS socket:

   a. Open a DTLS socket that uses the security tag 1001 and connect to a DTLS server on a specified port. Replace *example.com* with the hostname or IPv4 address of a DTLS server and *1234* with the corresponding port.

```
AT#XSSOCKET=1,2,0,1001

#XSSOCKET: 0,2,273

OK
```

```
AT#XCONNECT=0,"example.com",1234

#XCONNECT: 0,1

OK
```

1. Send plaintext data to the DTLS server and retrieve the returned data.

```
AT#XSEND=0,0,0,"Test DTLS"

#XSEND: 0,0,9

OK

AT#XRECV=0,0,0,0

#XRECV: 0,0,9
Test DTLS
OK
```

b. Close the socket.

```
AT#XCLOSE=0

#XCLOSE: 0,0

OK
```

UDP server

To act as a UDP server, the nRF91 Series DK must have a public IP address and the radio network must be configured to route incoming IP packets to the nRF91 Series DK. These depend on the network and SIM card used.

To check your current setup, use the `AT+CGDCONT?` command to check if the IP address allocated by the network is a reserved IPv4 private address of class A, B, or C (see Private addresses). If it is, the device is not reachable from the public network with this IPv4 address and you should try with an IPv6 address instead. Generally, IPv6 addresses are more likely to be reachable from the public network.

To test the UDP server functionality, complete the following steps:

1. Create a Python script `client_udp.py` that acts as a UDP client. See the following sample code (make sure to use the correct IP addresses and port):

```
import socket
import time

host_addr = '000.000.000.00'
```

(continues on next page)

(continued from previous page)

```
host_port = 1234
host = (host_addr, host_port)
local_addr = '9.999.999.99'
local_port = 1234
local = (local_addr, local_port)
s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
s.bind(local)
print("Sending: 'Hello, UDP#1!")
s.sendto(b"Hello, UDP#1!", host)
time.sleep(1)
print("Sending: 'Hello, UDP#2!")
s.sendto(b"Hello, UDP#2!", host)
data, address = s.recvfrom(1024)
print(data)
print(address)

print("Sending: 'Hello, UDP#3!")
s.sendto(b"Hello, UDP#3!", host)
time.sleep(1)
print("Sending: 'Hello, UDP#4!")
s.sendto(b"Hello, UDP#4!", host)
time.sleep(1)
print("Sending: 'Hello, UDP#5!")
s.sendto(b"Hello, UDP#5!", host)
data, address = s.recvfrom(1024)
print(data)
print(address)

print("Closing connection")
s.close()
```

2. Establish and test a UDP connection:

   a. Open a UDP socket and bind it to the UDP port that you want to use. Replace *1234* with the correct port number.

   ```
   AT#XSOCKET=1,2,1
   #XSOCKET: 0,2,17
   OK

   AT#XBIND=0,1234
   OK
   ```

   b. Run the `client_udp.py` script to start sending data to the server.

   c. Start receiving and acknowledging the data. Replace *example.com* with the host-name or IPv4 address of the UDP client and *1234* with the corresponding port. A more complete example would use `AT#XAPOLL` to see when data has arrived and then read it with `AT#XRECVFROM`.

```
AT#XRECVFROM=0,0,0,0
#XRECVFROM: 0,0,13,"<IP address>",<port>
Hello, UDP#1!
OK

AT#XRECVFROM=0,0,0,0
#XRECVFROM: 0,0,13,"<IP address>",<port>
Hello, UDP#2!
OK

AT#XSENDTO=0,0,0,"example.com",1234,"UDP1/2 received"
#XSENDTO: 0,0,15
OK

AT#XRECVFROM=0,0,0,0
#XRECVFROM: 0,0,13,"<IP address>",<port>
Hello, UDP#3!
OK

AT#XRECVFROM=0,0,0,0
#XRECVFROM: 0,0,13,"<IP address>",<port>
Hello, UDP#4!
OK

AT#XRECVFROM=0,0,0,0
#XRECVFROM: 0,0,13,"<IP address>",<port>
Hello, UDP#5!
OK

AT#XSENDTO=0,0,0,"example.com",1234,"UDP3/4/5 received"
#XSENDTO: 0,0,17
OK
```

d. Observe the output of the Python script:

```
$ python client_udp.py

Sending: 'Hello, UDP#1!
Sending: 'Hello, UDP#2!
b'UDP1/2 received'
('000.000.000.00', 1234, 0, 0)
Sending: 'Hello, UDP#3!
Sending: 'Hello, UDP#4!
Sending: 'Hello, UDP#5!
b'UDP3/4/5 received'
('000.000.000.00', 1234, 0, 0)
Closing connection
```

e. Close the socket.

```
AT#XCLOSE=0
#XCLOSE: 0,0
OK
```

DNS lookup

1. Look up the IP address for a hostname.

```
AT#XGETADDRINFO="www.google.com"
#XGETADDRINFO: "172.217.174.100"
OK

AT#XGETADDRINFO="ipv6.google.com"
#XGETADDRINFO: "2404:6800:4006:80e::200e"
OK

AT#XGETADDRINFO="172.217.174.100"
#XGETADDRINFO: "172.217.174.100"
OK

AT#XGETADDRINFO="2404:6800:4006:80e::200e"
#XGETADDRINFO: "2404:6800:4006:80e::200e"
OK
```

Socket options

After opening a client-role socket, you can configure various options.

1. Check the available values for the XSOCKETOPT command.

```
AT#XSOCKETOPT=?
#XSOCKETOPT: <handle>,(0,1),<name>,<value>
OK
```

2. Open a client socket.

```
AT#XSOCKET=1,1,0
#XSOCKET: 0,1,6
OK
```

3. Test to set and get socket options. Note that not all options are supported.

```
AT#XSOCKETOPT=0,1,20,30
OK
```

2.7.3 ICMP AT commands

Complete the following steps to test the functionality provided by the *ICMP AT commands*:

1. Ping a remote host, for example, *www.google.com*.

```
AT#XPING="www.google.com",45,5000,5,1000
OK
#XPING: 0.637 seconds
#XPING: 0.585 seconds
#XPING: 0.598 seconds
#XPING: 0.598 seconds
#XPING: 0.599 seconds
#XPING: average 0.603 seconds

AT#XPING="ipv6.google.com",45,5000,5,1000
OK
#XPING: 0.140 seconds
#XPING: 0.109 seconds
#XPING: 0.113 seconds
#XPING: 0.118 seconds
#XPING: 0.112 seconds
#XPING: average 0.118 seconds
```

2. Ping a remote IP address, for example, 172.217.174.100.

```
AT#XPING="172.217.174.100",45,5000,5,1000
OK
#XPING: 0.873 seconds
#XPING: 0.576 seconds
#XPING: 0.599 seconds
#XPING: 0.623 seconds
#XPING: 0.577 seconds
#XPING: average 0.650 seconds
```

3 Serial Modem AT commands

3.1 AT command syntax

The AT Commands have standardized syntax rules.

Words enclosed in <angle brackets> are references to syntactical elements. Words enclosed in [square brackets] represent optional items that can be left out of the command line at the specified point. The brackets are not used when the words appear in the command line.

<CR>, <LF> and <CR><LF> are allowed in an AT command sent by an application.

A string type parameter input must be enclosed between quotation marks (`"string"`).

There are 3 types of AT commands:

- Set command `<CMD>[=...]`. Set commands set values or perform actions.
- Read command `<CMD>?`. Read commands check the current values of subparameters.
- Test command `<CMD>=?`. Test commands test the existence of the command and provide information about the type of its subparameters. Some test commands can also have other functionality.

AT responds to all commands with a final response.

The maximum AT command length is 8190 bytes, including the terminator character.

3.2 Generic AT commands

This page describes generic AT commands.

3.2.1 UART baud rate AT+IPR

The `AT+IPR` command sets the UART baud rate.

Set command

The set command sets the UART baud rate. Serial Modem does not support automatic baud rate detection. When the device resets, the baud rate is set to the default value set in the devicetree.

Syntax

```
AT+IPR=<baud_rate>
```

The parameters and their defined values are the following:

<baud_rate>
Integer. The desired baud rate.

Note:

The baud rate change takes effect after the modem responds with OK. The host must switch to the new baud rate to continue communication.

Example Set the baud rate to 115200.

```
AT+IPR=115200
OK
```

Read command

The read command reads the current UART baud rate.

Syntax

```
AT+IPR?
```

Response syntax

```
+IPR: <baud_rate>
```

Example

```
AT+IPR?
+IPR: 115200
OK
```

Test command

The test command lists the supported baud rates.

Syntax

```
AT+IPR=?
```

Response syntax

```
+IPR: (list of supported autodetectable baud rates)[,(list of fixed-
only baud rates)]
```

Example

```
AT+IPR=?
+IPR: (),(115200,230400,460800,921600,1000000)
OK
```

3.2.2 Echo E0/E1

The E command enables or disables the AT command echo feature of the Serial Modem.

Set command

The set command enables or disables the AT command echo feature. While enabled, the Serial Modem echoes back all the characters that are received in the AT command mode.

Note:

When working with a terminal, you should disable local echo to avoid double echoing of characters.

Syntax

```
ATE1 - Enable AT command echo.
ATE0 - Disable AT command echo (default).
```

Example Set the AT command echo:

```
ATE1
OK

AT
AT    // Echoed command
OK

ATE0
ATE0  // Echoed command
OK

AT
OK
```

Read command

The read command is not supported.

Test command

The test command is not supported.

3.2.3 Serial Modem version #XSMVER

The #XSMVER command returns the versions of the Serial Modem application firmware and nRF Connect SDK in which the application firmware is built.

Set command

The set command returns the versions of the Serial Modem application firmware and nRF Connect SDK.

Syntax

```
AT#XSMVER
```

Response syntax

```
#XSMVER: "<sm_version>","<ncs_version>"[,"<customer_version>"]
```

The parameters and their defined values are the following:

<sm_version>
String.

The version of the Serial Modem application firmware in `v<major>.<minor>.<patch>[<git-describe-style>]` format.

The `v<major>.<minor>.<patch>` prefix comes from `app/VERSION`.

The optional `<git-describe-style>` suffix in `-N-g<hash>` format (and `-dirty`) comes from `git describe` in the application tree.

Example: `"v2.99.0-68-gb2d3dde4dfa2-dirty"` uses `v2.99.0` from `app/VERSION` and `-68-gb2d3dde4dfa2` from `git describe`.

Hosts can use the `v<major>.<minor>.<patch>` prefix to verify that application DFU or FOTA moved to a newer image. To check, compare the AT#XSMVER response before and after the update.

Application downgrade prevention uses the same `major.minor.patch` version. An application update with a lower version than the running application is rejected. The Git suffix and the `VERSION_TWEAK` value in `app/VERSION` are not taken into account.

See *Releasing Serial Modem* for release version updates.

<ncs_version>
String. The version of the nRF Connect SDK in `<major>.<minor>.<patch>` format.

<customer_version>
String. The *CONFIG_SM_CUSTOMER_VERSION* string, if defined.

Example The following command examples read the versions:

```
// v2.0.0 release
AT#XSMVER
#XSMVER: "v2.0.0","3.4.1"
OK
```

```
// FOTA test package for v2.0.0 release
AT#XSMVER
#XSMVER: "v2.0.0","3.4.1","FOTA-TEST"
OK
```

```
// First v3.0.0 pre-release
AT#XSMVER
#XSMVER: "v2.99.1","3.4.99"
OK
```

```
// Pre-release with customer version
AT#XSMVER
#XSMVER: "v2.99.1","3.4.99","SomeCustomer_5.0"
OK
```

```
// Development build
AT#XSMVER
#XSMVER: "v2.99.0-67-ga9e396bc3d58","3.4.99"
OK
```

Read command

The read command is not supported.

Test command

The test command is not supported.

3.2.4 SM proprietary command list #XCLAC

The #XCLAC command requests the list of the proprietary Serial Modem commands.

Set command

The set command requests the list of the proprietary Serial Modem commands. It is an add-on for AT+CLAC, which lists all modem AT commands.

Syntax

```
AT#XCLAC
```

Response syntax

```
<command list>
```

The parameters and their defined values are the following:

<command list>
String. A list of values representing all the #X* commands followed by <CR><LF>.

Example

```
AT#XCLAC
AT#XSMVER
AT#XSLEEP
AT#XCLAC
AT#XSOCKET
AT#XBIND
AT#XCONNECT
AT#XSEND
AT#XRECV
AT#XSENDTO
AT#XRECVFROM
AT#XPING
AT#XGNSS
OK
```

Read command

The read command is not supported.

Test command

The test command is not supported.

3.2.5 Power saving #XSLEEP

The #XSLEEP command makes the Serial Modem to enter idle or sleep mode.

Note:

The #XSLEEP command is intended for experimentation and power consumption measurements and must not be used in production.

You can use the DTR (Data Terminal Ready) and RI (Ring Indicator) signals to control the power state of the UART between the Serial Modem and the host. See the *UART configuration* section for more information about DTR and RI.

Note:

If you want to do power measurements on the nRF91 Series development kit while running the Serial Modem, remember to disable unused peripherals.

Set command

The set command makes the Serial Modem enter either Idle or Sleep mode.

Syntax

```
AT#XSLEEP=<sleep_mode>
```

The parameters and their defined values are the following:

<sleep_mode>

- 1 - Enter Sleep. In this mode, both the Serial Modem service and the LTE connection are terminated.

  Serial Modem can be woken up using the DTR pin.

- 2 - Enter Idle. In this mode, both the Serial Modem service and the LTE connection are maintained, but the UART is disabled to save power. Received data is buffered and sent to the host after idle mode is exited.

  Serial Modem can exit the idle mode using the DTR pin. When the Serial Modem is in idle mode, and there is data to be read by the host, the RI pin is asserted for a short period of time to notify the host. The host can then deassert and assert DTR to exit idle mode and read the data.

The DTR pin is defined either in the `boards/*_ns.overlay` overlay file matching your board or in the `external-mcu.overlay` overlay file, if it is included.

Note:

- If the modem is on, entering Sleep mode (by issuing `AT#XSLEEP=1`) sends a `+CFUN=0` command to the modem, which causes a write to non-volatile memory (NVM). Take the NVM wear into account, or put the modem in flight mode by issuing `AT+CFUN=4` before Sleep mode.

Examples

```
AT#XSLEEP=0
ERROR
```

```
AT#XSLEEP=1
OK
```

See the following for an example of when the modem is on:

```
AT+CFUN=4
OK

AT#XSLEEP=1
OK
```

```
AT#XSLEEP=2
OK
```

Read command

The read command is not supported.

Test command

The test command tests the existence of the AT command and provides information about the type of its subparameters.

Syntax

```
AT#XSLEEP=?
```

Response syntax

```
#XSLEEP: <list of shutdown_mode>
```

Example

```
AT#XSLEEP=?
#XSLEEP: (1,2)
OK
```

3.2.6 Power off #XSHUTDOWN

The #XSHUTDOWN command makes the nRF91 Series SiP enter System OFF mode, which is the deepest power saving mode.

Set command

The set command makes the nRF91 Series SiP enter System OFF mode.

Syntax

```
AT#XSHUTDOWN
```

Note:

In this case the nRF91 Series SiP cannot be woken up using the DTR pin.

Example

```
AT#XSHUTDOWN
OK
```

Read command

The read command is not supported.

Test command

The test command is not supported.

3.2.7 Reset #XRESET

The #XRESET command performs a soft reset of the nRF91 Series SiP.

Set command

The set command resets the nRF91 Series SiP.

Syntax

```
AT#XRESET
```

Example

```
AT#XRESET
OK
Ready
```

Read command

The read command is not supported.

Test command

The test command is not supported.

3.2.8 Modem reset #XMODEMRESET

The `#XMODEMRESET` command performs a reset of the modem.

The modem is set to minimal function mode (via `+CFUN=0`) before being reset. The Serial Modem is not restarted. After the command returns, the modem will be in minimal function mode.

Set command

The set command resets the modem.

Syntax

```
AT#XMODEMRESET
```

Response syntax

```
#XMODEMRESET: <result>[,<error_code>]
```

The parameters and their defined values are the following:

<result>

- `0` - Success.
- *Positive value* - On failure, indicates the step that failed.

<error_code>
Integer. Only printed when the modem reset was not successful. It is the error code indicating the reason for the failure.

Example

```
AT#XMODEMRESET

#XMODEMRESET: 0

OK
```

Read command

The read command is not supported.

Test command

The test command is not supported.

3.2.9 Device UUID #XUUID

The `#XUUID` command requests the device UUID.

Set command

The set command returns the device UUID.

Syntax

```
AT#XUUID
```

Response syntax

```
#XUUID: <device-uuid>
```

The parameters and their defined values are the following:

<device-uuid>
String. The UUID of the device.

Example

```
AT#XUUID

#XUUID: 50503041-3633-4261-803d-1e2b8f70111a

OK
```

Read command

The read command is not supported.

Test command

The test command is not supported.

3.2.10 MCUboot bootloader info #XBOOTINFO

Query information about the MCUboot second-stage bootloader.

This command is only available when the firmware is built with the NSIB (B0) and MCUboot as a second-stage bootloader.

Set command

```
AT#XBOOTINFO=<op>
```

The parameters and their defined values are the following:

<op>

- `0` - Query the firmware version of the active MCUboot slot.
- `1` - Query which MCUboot slot is currently active.

Response syntax

```
#XBOOTINFO: <value>
```

The parameters and their defined values are the following:

<value>
Integer.

- When `<op>` is `0`, it is an unsigned integer representing the firmware version of the active MCUboot slot, used by NSIB as a downgrade-protection counter.

  The firmware version is set at build time using `CONFIG_FW_INFO_FIRMWARE_VERSION`.

- When `<op>` is `1`, it is `0` if slot `s0` is active and `1` if slot `s1` is active.

Examples Query the firmware version of the active MCUboot slot:

```
AT#XBOOTINFO=0

#XBOOTINFO: 1

OK
```

Query the active MCUboot slot:

```
AT#XBOOTINFO=1

#XBOOTINFO: 0

OK
```

Test command

```
AT#XBOOTINFO=?
```

Response syntax

```
#XBOOTINFO: (0,1)
```

Example

```
AT#XBOOTINFO=?
#XBOOTINFO: (0,1)
OK
```

3.2.11 Modem fault #XMODEM

The application monitors the modem status. When the application detects a *modem fault*, it sends the #XMODEM unsolicited notification.

Unsolicited notification

The application sends the following unsolicited notification when it detects a modem fault:

```
#XMODEM: FAULT,<reason>,<program_count>
```

The parameters and their defined values are the following:

<reason>
Integer. A hexadecimal integer indicating the reason of the modem fault.

<program_count>
Integer. A hexadecimal integer indicating the address of the modem fault.

The application sends the following unsolicited notification when it shuts down libmodem:

```
#XMODEM: SHUTDOWN,<result>
```

The parameters and their defined values are the following:

<result>
Integer. The result of the shutdown of libmodem.

The application sends the following unsolicited notification when it re-initializes libmodem:

```
#XMODEM: INIT,<result>
```

The parameters and their defined values are the following:

<result>
Integer. The result of the re-initialization of libmodem.

Note:

After libmodem is re-initialized, the MCU side must restart the current active service as follows:

1. Stopping the service. For example, disconnecting the TCP connection and closing the socket.
2. Connecting again using LTE.
3. Restarting the service. For example, opening the socket and re-establishing the TCP connection.

3.2.12 Memory statistics #XDBGSTATSMEM

Note:

This AT command is Experimental.

Print heap and thread stack memory statistics to the log at the `INF` level. Since this is using the logging as an output, you must enable logging first by issuing the `AT#XLOG=1` command.

The following statistics are printed:

- System heap (`malloc` heap) - Free bytes, allocated bytes, and peak allocated bytes.
- Kernel heap (`k_malloc` heap) - Free bytes, allocated bytes, and peak allocated bytes.
- Kernel heap block information - This is only available when the `CONFIG_SYS_HEAP_INFO` Kconfig option is enabled.
- Thread stack usage statistics:
  - This is only available when the *CONFIG_SM_DEBUG_STATS_THREAD_STACK* Kconfig option is enabled.
  - Setting the `CONFIG_LOG_BUFFER_SIZE` Kconfig option to 2048 or more is required to avoid dropping some of the log entries showing stack usage statistics.

This command is available when the firmware is built with the *CONFIG_SM_DEBUG_STATS_HEAP* Kconfig option enabled. In addition, the `CONFIG_LOG` Kconfig

option and the CONFIG_SM_LOG_LEVEL_DBG or CONFIG_SM_LOG_LEVEL_INF Kconfig option must be enabled.

Set command

```
AT#XDBGSTATSMEM
```

The command prints memory statistics to the log and returns OK. No memory information is returned in the AT response.

Read command

The read command is not supported.

Test command

The test command is not supported.

Example

Query memory statistics:

```
AT#XLOG=1
OK
AT#XDBGSTATSMEM
OK
```

These commands will print the following output in the log:

```
[00:00:36.270,965] <inf> sm_at: System heap stats:
[00:00:36.270,996] <inf> sm_at:   free:            45844
[00:00:36.270,996] <inf> sm_at:   allocated:         472
[00:00:36.271,026] <inf> sm_at:   max. allocated:    472
[00:00:36.271,026] <inf> sm_at: Kernel heap stats:
[00:00:36.271,057] <inf> sm_at:   free:             1892
[00:00:36.271,057] <inf> sm_at:   allocated:           0
[00:00:36.271,057] <inf> sm_at:   max. allocated:   1280
```

The output appears as follows when *CONFIG_SM_DEBUG_STATS_THREAD_STACK* and CONFIG_SYS_HEAP_INFO Kconfig options are enabled.

```
[00:01:16.852,478] <inf> sm_at: System heap stats:
[00:01:16.853,393] <inf> sm_at:   free:            48292
[00:01:16.854,431] <inf> sm_at:   allocated:         472
[00:01:16.855,438] <inf> sm_at:   max. allocated:   1344
[00:01:16.856,445] <inf> sm_at: Kernel heap stats:
[00:01:16.857,360] <inf> sm_at:   free:             1892
```

(continues on next page)

(continued from previous page)

```
[00:01:16.858,398] <inf> sm_at:   allocated:           0
[00:01:16.859,405] <inf> sm_at:   max. allocated:   1280
[00:01:16.860,412] <inf> sm_at: Kernel heap block information:
Heap at 0x20028d40 contains 245 units in 8 buckets

bucket#    min units        total      largest      largest
           threshold       chunks      (units)      (bytes)
-------------------------------------------------------------
      7          128            1          237         1896

Chunk dump:
chunk    0: [*] size=8    left=0    right=8
chunk    8: [-] size=237  left=0    right=245
chunk  245: [*] size=0    left=8    right=245

1892 free bytes, 0 allocated bytes, overhead = 72 bytes (3.7%)
[00:01:16.869,781] <inf> sm_at: Thread stack stats:
Thread analyze:
 ppp_tx              : STACK: unused 796 usage 228 / 1024 (22 %);
CPU: 0 %
                     : Total CPU cycles used: 2
 trace_thread        : STACK: unused 460 usage 308 / 768 (40 %);
CPU: 0 %
                     : Total CPU cycles used: 3
 coap_client_recv_thread: STACK: unused 1276 usage 772 / 2048 (37
%); CPU: 0 %
                     : Total CPU cycles used: 67
 nrf_provisioning_work_q: STACK: unused 1812 usage 236 / 2048 (11
%); CPU: 0 %
                     : Total CPU cycles used: 2
 date_time_work_q    : STACK: unused 492 usage 788 / 1280 (61 %);
CPU: 0 %
                     : Total CPU cycles used: 32
 ppp_data_passing    : STACK: unused 1564 usage 484 / 2048 (23 %);
CPU: 0 %
                     : Total CPU cycles used: 2
 downloader          : STACK: unused 3844 usage 252 / 4096 (6 %);
CPU: 0 %
                     : Total CPU cycles used: 2
 net_mgmt            : STACK: unused 596 usage 204 / 800 (25 %);
CPU: 0 %
                     : Total CPU cycles used: 1
 sysworkq            : STACK: unused 3180 usage 916 / 4096 (22 %);
CPU: 0 %
                     : Total CPU cycles used: 475
 idle                : STACK: unused 252 usage 68 / 320 (21 %); CPU:
99 %
                     : Total CPU cycles used: 2506299
```

(continues on next page)

(continued from previous page)

```
 sm_work_q           : STACK: unused 812 usage 3284 / 4096 (80 %);
CPU: 0 %
                     : Total CPU cycles used: 4467
 ISR0                : STACK: unused 1588 usage 460 / 2048 (22 %);
CPU: 0 %
                     : Total CPU cycles used: 0
```

3.3 CMUX AT commands

This page describes CMUX-related AT commands.

The GSM 0710 multiplexer protocol (CMUX) enables multiplexing multiple data streams through a single serial link, setting up one channel per data stream. For example, it can be used to exchange AT data and have a *Point-to-Point Protocol (PPP)* link up at the same time on a single UART. Serial Modem implements the basic option of the CMUX protocol with only UIH frames as described in the 3GPP TS 27.010 specification.

The maximum length of the information field in UIH frames is configurable using the `CONFIG_MODEM_CMUX_MTU` Kconfig option, which defaults to 127 bytes.

Note:

To use the nRF91 Series SiP as a cellular dial-up PPP modem, see *Cellular PPP modem*.

CMUX is enabled in Serial Modem by compiling it with the appropriate configuration files, depending on your use case. See the *Configuration files* section for more information.

Note:

- Only basic mode (mode 0) is supported.
- Only UIH frames are used.
- The speed used is the configured baud rate of Serial Modem's UART.
- No system parameters (N1, T1, T2, T3, k, m) are configurable.

3.3.1 CMUX setup +CMUX

The `AT+CMUX` command starts the CMUX multiplexer. It is defined in 3GPP TS 27.007 (section 5.7) and the underlying protocol is specified in 3GPP TS 27.010.

Only basic mode (`<mode>=0`) with subset `0` is supported. All other parameter values are rejected.

Set command

The set command allows you to start the CMUX multiplexer.

An OK response is sent before CMUX is started, after which only CMUX framing is accepted on the serial link.

Syntax

```
AT+CMUX=<mode>[,<subset>]
```

The parameters and their defined values are the following:

<mode>
Integer. Selects the operation mode. Only 0 (basic mode) is supported.

<subset>
Integer. Selects the subset of mode 0. Only 0 is supported. Default value is 0.

Read command

The read command returns the current multiplexer configuration.

Syntax

```
AT+CMUX?
```

Response syntax

```
+CMUX: <mode>,<subset>
```

The parameters and their defined values are the following:

<mode>
Integer. Always 0 (basic mode).

<subset>
Integer. Always 0.

Test command

The test command returns the supported parameter ranges.

Syntax

```
AT+CMUX=?
```

Response syntax

```
+CMUX: (0),(0)
```

Example

```
AT+CMUX=?

+CMUX: (0),(0)

OK
AT+CMUX?

+CMUX: 0,0

OK
AT+CMUX=0

OK
// CMUX is now started. Open the CMUX channels to continue
communication.
AT+CMUX=0,0

OK
// Equivalent to AT+CMUX=0.
```

3.3.2 CMUX setup #XCMUX

Note:

AT#XCMUX is a compatibility command for Serial Modem v1.x.x and is not recommended for new designs. Use the standard AT+CMUX=0 command instead.

The #XCMUX command manages the configuration of CMUX over the serial link.

Set command

The set command allows you to start CMUX and assign the CMUX channels.

The CMUX link is closed down automatically when the remote end sends the CMUX Multiplexer Close Down (CLD) sequence. The CMUX link can be closed manually with the AT#XCMUXCLD command.

Syntax

```
AT#XCMUX[=<AT_channel>]
```

The parameters and their defined values are the following:

<AT_channel>
Integer. The address of the AT channel. The AT channel denotes the CMUX channel where AT data (commands, responses, notifications) is exchanged. If specified, it must be 1, unless *PPP* is enabled. If PPP is enabled, it can also be 2 (to allocate the

first CMUX channel to PPP). If not specified, the previously used address is used. If no address has been previously specified, the default address is 1.

Note:

If there is more than one CMUX channel (such as when using *PPP*), the non-AT channels will automatically get assigned to addresses other than the one used for the AT channel. For example, if PPP is enabled and CMUX is started with the AT#XCMUX=2 command, the AT channel will be assigned to address 2 and the PPP channel to address 1.

An OK response is sent if the command is accepted, after which CMUX is started. This means that after successfully running this command, you must set up the CMUX link and open the channels appropriately. The AT channel will be available at the configured address. You can send an empty AT command to make sure that the protocol is set up properly.

Caution:

Using AT#XCMUX to switch the AT command channel at runtime is error-prone and not recommended. The channel switching might run out of sync with automatic PPP startup and recovery enabled by AT#XPPP=1.

Read command

The read command allows you to read the address of the AT channel and the total number of channels.

Syntax

```
AT#XCMUX?
```

Response syntax

```
#XCMUX: <AT_channel>,<channel_count>
```

The parameters and their defined values are the following:

<AT_channel>
Integer. The address of the AT channel. It is between 1 and <channel_count>.

<channel_count>
Integer. The total number of CMUX channels. It depends on what features are enabled (for example, *PPP*).

Example

Without PPP:

```
AT#XCMUX?

#XCMUX: 1,1

OK
AT#XCMUX

OK
// Here, CMUX is started and communication can now happen only
through it (until a reset).
// Open the AT channel, which is the only one, to continue
exchanging AT data.
AT

OK
```

With PPP:

```
AT#XCMUX?

#XCMUX: 1,2

OK
AT#XCMUX=2

OK
// Start up CMUX and open the channels. The AT channel is now at
address 2.
AT#XCMUX?

#XCMUX: 2,2

OK
```

3.3.3 CMUX close down #XCMUXCLD

The #XCMUXCLD command closes down the CMUX service.

This command can be used on host devices that do not support sending the CMUX Multiplexer close-down sequence. Once CMUX is closed down, the serial link returns to AT command mode.

Set command

The set command allows you to close down the CMUX link.

Syntax

```
AT#XCMUXCLD
```

An OK response is sent in the CMUX channel if the command is accepted, after which CMUX is closed down.

Read command

The read command is not supported.

3.3.4 URC channel #XCMUXURC

The #XCMUXURC command configures the CMUX DLC channel to which URCs are routed.

By default (channel 0), URCs are sent to the first open AT channel that is not in data mode. Routing URCs to a fixed channel is recommended when multiple CMUX channels are in use, for example, when one channel is occupied by a PPP session.

Set command

The set command sets the CMUX channel used for URC delivery.

Syntax

```
AT#XCMUXURC=<channel>
```

The parameters and their defined values are the following:

<channel>

- 0 - Auto-select: URCs are sent to the first open AT channel not in data mode.
- 99 - All channels: URCs are sent to all open AT channels not in data mode.
- Any other value routes URCs to the specified DLC channel.

Read command

The read command returns the currently configured URC channel.

Syntax

```
AT#XCMUXURC?
```

Response syntax

```
#XCMUXURC: <channel>
```

The parameters and their defined values are the following:

<channel>
Integer. The currently configured URC channel (see the set command).

Test command

The test command returns the parameter description.

Syntax

```
AT#XCMUXURC=?
```

Response syntax

```
#XCMUXURC=<channel>
```

Example

Route URCs to DLC channel 2 before starting CMUX:

```
AT#XCMUXURC=2

OK
AT#XCMUXURC?

#XCMUXURC: 2

OK
AT+CMUX=0

OK
// CMUX is now started. URCs are delivered on DLC channel 2.
```

3.4 CoAP client AT commands

This page describes AT commands for the CoAP client. The CoAP client operates on sockets managed by the *Socket AT commands*. You can perform the following using the Socket AT commands:

- Create a UDP socket with `AT#XSOCKET` or `AT#XSSOCKET`.
- Connect it with `AT#XCONNECT`.
- Set socket options with `AT#XSOCKETOPT` or `AT#XSSOCKETOPT`.
- Close with `AT#XCLOSE`.

Only one CoAP request might be active at a time across all sockets.

3.4.1 CoAP request #XCOAPCREQ

The `#XCOAPCREQ` command starts an asynchronous CoAP request on a connected UDP socket.

Set command

The set command sends a CoAP request and returns OK immediately. The server's response is delivered through unsolicited #XCOAPCDATA and #XCOAPCSTAT notifications.

Syntax

```
AT#XCOAPCREQ=<handle>,<path>,<method>[,<auto_reception>[,<format>[,
<confirmable>[,<content_format>[,<payload_len>[,<opt_num_1>,<opt_
val_1>[,<opt_num_2>,<opt_val_2>[...]]]]]]]]
```

The parameters and their defined values are the following:

<handle>
Integer. Identifies the connected socket handle returned by AT#XSOCKET and connected by AT#XCONNECT.

<path>
String. The URI-path of the resource, for example "/sensors/temperature".

<method>

- 1 - GET.
- 2 - POST.
- 3 - PUT.
- 4 - DELETE.
- 5 - FETCH.
- 6 - PATCH.
- 7 - iPATCH.

<auto_reception>
Integer. Optional. When omitted, automatic reception is used.

- 1 - Automatic mode (default). Response bytes are forwarded to the host automatically through #XCOAPCDATA URCs.
- 0 - Manual mode. The firmware buffers each response block and emits #XCOAPCHEAD. The host must then retrieve the block using AT#XCOAPCDATA.

<format>
Integer. Optional. Controls the encoding used to deliver response payload bytes to the host.

- 0 - Binary (default). Response payload bytes are forwarded to the host as raw binary.
- 1 - Hex string. Response payload bytes are encoded as a lowercase ASCII hex string before delivery. Each raw byte becomes two hex characters. The <length> field in #XCOAPCDATA URCs and responses always reports the raw byte count. The actual data transmitted to the host is 2×<length> ASCII hex characters.

<confirmable>
Integer. Optional.

- `0` - Non-confirmable (NON) message (default).
- `1` - Confirmable (CON) message.

<content_format>
Integer. Optional. The CoAP Content-Format option number for the request payload (default `0` = `text/plain`). Common values:

- `0` - text/plain.
- `42` - application/octet-stream.
- `50` - application/json.
- `60` - application/cbor.

<payload_len>
Integer. Optional. When `0` or omitted and no option pairs follow, no request payload is sent and the command returns `OK` immediately. A positive integer specifies the total payload length in bytes and the command then returns `OK` and enters data mode.

> **Note:**
>
> When option pairs follow, `<payload_len>` must be written explicitly. Use `0` if no payload is needed. Omitting it shifts the option pairs one position left, which always causes the command to return `ERROR`.

The host must send exactly this many bytes as the request payload. Data mode exits and `#XDATAMODE: 0` is reported when all bytes have been consumed. Data mode is the only mechanism for supplying the payload - there is no inline parameter alternative.

The exact sending behavior depends on payload size:

- Small payload (≤ `CONFIG_COAP_CLIENT_BLOCK_SIZE`, default 512 bytes) - All bytes are buffered until data mode exits, then the CoAP request is sent as a single packet.
- Large payload (> `CONFIG_COAP_CLIENT_BLOCK_SIZE`) - The CoAP request starts transmitting while data mode is still active. As soon as the first full block of `CONFIG_COAP_CLIENT_BLOCK_SIZE` bytes has arrived, the first CoAP Block1 packet is sent to the server. Each subsequent block is sent as more data arrives, so the server exchange runs concurrently with the host's serial upload.

<opt_num_X>,<opt_val_X>
Optional CoAP option pairs.

- When present, `<payload_len>` must also be present as the positional separator before the first pair.
- `<opt_num_X>` is a decimal integer option number.
- `<opt_val_X>` is a quoted string.
- A value prefixed with `0x` or `0X` is decoded as raw hex bytes (for example, `"0x3c"` → byte `0x3c`).

- The hex part after the prefix must be non-empty and even-length, otherwise, the command returns ERROR.
- Any value without the 0x prefix is used verbatim, including strings that look like hex (for example "abcd" → 4 ASCII bytes).
- Common option numbers:
  - 6 - Observe ("0x00" = register, "0x01" = deregister).
  - 17 - Accept (content format the client will accept, hex-encoded).
  - 35 - Proxy-Uri (full URI passed to a proxy, verbatim string).

Examples:

```
17,"0x00"                 Accept: text/plain (0x00)
17,"0x3c"                 Accept: application/cbor (0x3c = 60)
35,"coap://proxy/res"     Proxy-Uri verbatim string
6,"0x00",17,"0x3c"        Observe register + Accept cbor
```

Note:

The Observe option (option 6) with the register value "0x00" starts a persistent subscription. The request stays active and the firmware delivers each server notification through #XCOAPCDATA (automatic mode) or #XCOAPCHEAD (manual mode). No #XCOAPCSTAT is emitted between notifications. To stop the subscription, use AT#XCOAPCCANCEL. You cannot send a separate deregister request (6,"0x01") while the subscription is active, because only one CoAP request can be active at a time.

Unsolicited notification #XCOAPCDATA is emitted in automatic mode for each received response block.

```
#XCOAPCDATA: <handle>,<offset>,<length>
```

The notification line is terminated with \r\n, the response data follows immediately, and a \r\n terminator follows the data bytes. In binary mode (default), the data is <length> raw bytes. In hex mode (<format>=1), the data is 2×<length> lower-case ASCII hex characters.

The parameters and their defined values are the following:

<handle>
Integer. Identifies the socket.

<offset>
Integer. The number of response bytes already delivered before this block.

<length>
Integer. The raw byte count of the response data in this block. In hex mode (<format>=1), the payload following the header line is 2×<length> ASCII hex characters.

#XCOAPCHEAD is emitted in manual mode for each received response block.

```
#XCOAPCHEAD: <handle>,<code>,<block_len>
```

The parameters and their defined values are the following:

<handle>
Integer. Identifies the socket.

<code>
Integer. The CoAP response code for this block.

<block_len>
Integer. The raw byte count of the buffered block. In hex mode (<format>=1), AT#XCOAPCDATA will deliver 2×<block_len> ASCII hex characters. The host must call AT#XCOAPCDATA to retrieve this block before the next one arrives.

#XCOAPCSTAT is emitted when the request completes, fails, or is cancelled.

```
#XCOAPCSTAT: <handle>,<status_code>,<total_bytes>
```

The parameters and their defined values are the following:

<handle>
Integer. Identifies the socket.

<status_code>
Integer. The CoAP response code on success, or -1 on failure or cancel. CoAP response codes are encoded as (class << 5) | detail. For example, 2.05 Content is encoded as 69 and 2.04 Changed as 68.

<total_bytes>
Integer. The total number of raw response payload bytes received. In hex mode (<format>=1), this is the raw byte count, not the number of hex characters delivered.

Note:

Block2 response transfer (server-side fragmentation of large responses) is handled automatically by the CoAP client library. All blocks are forwarded to the host in sequence. #XCOAPCSTAT is emitted only after the final block is received and delivered.

Examples CoAP GET (automatic mode, binary):

```
AT#XSOCKET=1,2,0
#XSOCKET: 0,2,17
OK

AT#XCONNECT=0,"coap.example.com",5683
#XCONNECT: 0,1
OK
```

(continues on next page)

(continued from previous page)

```
AT#XCOAPCREQ=0,"/sensors/temperature",1
OK

#XCOAPCDATA: 0,0,12
23.5 Celsius

#XCOAPCSTAT: 0,69,12

AT#XCLOSE=0
OK
```

Note:

In automatic mode, a `\r\n` terminator is appended after the data bytes of each `#XCOAPCDATA` notification.

CoAP GET (automatic mode, hex-encoded response, `format=1`):

```
AT#XCOAPCREQ=0,"/sensors/temperature",1,1,1
OK

#XCOAPCDATA: 0,0,12
32332e352043656c73697573

#XCOAPCSTAT: 0,69,12
```

Note:

The hex string and the line break after it represent the `2×<length>` hex characters followed by the `\r\n` terminator.

CoAP GET (manual mode):

```
AT#XCOAPCREQ=0,"/sensors/temperature",1,0
OK

#XCOAPCHEAD: 0,69,12

AT#XCOAPCDATA=0
#XCOAPCDATA: 0,0,12
23.5 Celsius
OK
```

(continues on next page)

(continued from previous page)

```
#XCOAPCSTAT: 0,69,12
```

CoAP POST with JSON payload (`content_format=50` = `application/json`):

```
AT#XCOAPCREQ=0,"/data",2,1,0,0,50,13
OK
{"value": 42}
#XDATAMODE: 0

#XCOAPCSTAT: 0,68,0
```

CoAP GET through proxy with Proxy-URI option (option 35, value is a plain URI string):

```
AT#XCOAPCREQ=0,"proxy",1,1,0,0,0,0,35,"coap://remote-host/resource"
OK

#XCOAPCDATA: 0,0,128
<128 bytes>

#XCOAPCSTAT: 0,69,128
```

CoAP GET with Observe and Accept options (option 6 = Observe register `"0x00"`, option 17 = Accept `"0x3c"` = 0x3c = application/cbor):

```
AT#XCOAPCREQ=0,"/obs",1,1,0,1,0,0,6,"0x00",17,"0x3c"
OK

#XCOAPCDATA: 0,0,18
<18 bytes CBOR>

#XCOAPCDATA: 0,0,18
<18 bytes CBOR>

AT#XCOAPCCANCEL=0
OK

#XCOAPCSTAT: 0,-1,0
```

The subscription stays active and delivers a `#XCOAPCDATA` notification for each server update until you cancel it with `AT#XCOAPCCANCEL`. The `#XCOAPCSTAT` notification is emitted only when the subscription ends.

Test command

The test command tests the existence of the command and provides information about the type of its subparameters.

Syntax

```
AT#XCOAPCREQ=?
```

Response syntax

```
#XCOAPCREQ: <handle>,<path>,<method>[,<auto_reception>[,<format>[,
<confirmable>[,<content_format>[,<payload_len>[,<opt_num>,<opt_val>.
..]]]]]]
```

Example

```
AT#XCOAPCREQ=?
#XCOAPCREQ: <handle>,<path>,<method>[,<auto_reception>[,<format>[,
<confirmable>[,<content_format>[,<payload_len>[,<opt_num>,<opt_val>.
..]]]]]]
OK
```

3.4.2 CoAP data pull #XCOAPCDATA

The #XCOAPCDATA command pulls a buffered response block for a manual-mode CoAP request.

Set command

The set command delivers one buffered response block to the host and unblocks the CoAP client to request the next Block2 block, if any.

Syntax

```
AT#XCOAPCDATA=<handle>[,<length>]
```

The parameters and their defined values are the following:

<handle>
Integer. Identifies the socket used when starting the manual-mode request.

<length>
Integer. Optional. The maximum number of bytes to return in this pull.

When omitted, the full block buffer size (CONFIG_COAP_CLIENT_BLOCK_SIZE, default 512 bytes) is used.

Response syntax When a block is available:

```
#XCOAPCDATA: <handle>,<offset>,<length>
<data>
OK
```

In binary mode (default), <data> is <length> raw bytes. In hex mode (<format>=1), <data> is 2×<length> lower-case ASCII hex characters.

The #XCOAPCDATA: line is terminated with \r\n. <length> raw response bytes follow immediately with no additional separator. OK follows on its own line after the response bytes.

When no block has been buffered yet (the firmware is still waiting for the server response).

```
#XCOAPCDATA: <handle>,<offset>,<length>
OK
```

The parameters and their defined values are the following:

<handle>
Integer. Identifies the socket.

<offset>
Integer. The number of response bytes already delivered before this pull.

<length>
Integer. The number of response bytes delivered in this pull. A value of 0 means no block is ready yet.

After the last block is drained, #XCOAPCSTAT is emitted as a URC after the final OK.

Note:

If AT#XCOAPCDATA is not called within 30 seconds of a #XCOAPCHEAD notification, the request is aborted and #XCOAPCSTAT: <handle>,-1,<total_bytes> is emitted.

Example

```
AT#XCOAPCREQ=0,"/large",1,0
OK

#XCOAPCHEAD: 0,69,512

AT#XCOAPCDATA=0,512
#XCOAPCDATA: 0,0,512
<512 bytes>
OK

#XCOAPCHEAD: 0,69,512

AT#XCOAPCDATA=0,512
#XCOAPCDATA: 0,512,512
<512 bytes>
OK

#XCOAPCSTAT: 0,69,1024
```

Test command

The test command tests the existence of the command and provides information about the type of its subparameters.

Syntax

```
AT#XCOAPCDATA=?
```

Response syntax

```
#XCOAPCDATA: <handle>[,<length>]
```

Example

```
AT#XCOAPCDATA=?
#XCOAPCDATA: <handle>[,<length>]
OK
```

3.4.3 CoAP request cancel #XCOAPCCANCEL

The #XCOAPCCANCEL command cancels the active CoAP request. It is also the mechanism for stopping a CoAP Observe subscription started with the Observe register option (`6,"0x00"`).

Set command

The set command cancels the currently active CoAP request.

Syntax

```
AT#XCOAPCCANCEL=<handle>
```

The parameters and their defined values are the following:

<handle>
: Integer. Must match the socket handle that was passed to AT#XCOAPCREQ when the request was started. This acts as an ownership assertion: the command returns ERROR if no request is active or if the active request belongs to a different handle.

Note:

Only one CoAP request can be active at a time. The <handle> check is an ownership guard, not a per-socket filter. Cancellation always applies to the single in-flight request.

An unsolicited #XCOAPCSTAT: <handle>,-1,<total_bytes> notification is emitted after cancellation, where <total_bytes> is the number of response bytes already delivered to the host.

Example

```
AT#XCOAPCCANCEL=0
OK
#XCOAPCSTAT: 0,-1,0
```

Test command

The test command tests the existence of the command and provides information about the type of its subparameters.

Syntax

```
AT#XCOAPCCANCEL=?
```

Response syntax

```
#XCOAPCCANCEL: <handle>
```

Example

```
AT#XCOAPCCANCEL=?
#XCOAPCCANCEL: <handle>
OK
```

3.5 DFU AT commands

This page describes AT commands related to Device Firmware Update (DFU) operations. These commands allow you to update the application firmware, delta modem firmware, full modem firmware, or the MCUboot second-stage bootloader through UART.

Note:

The DFU commands use data mode to receive firmware data. See the *Data mode* section for more information about data mode.

3.5.1 DFU types

The following DFU image types are supported:

- `0` - Application firmware (MCUboot)
- 1 - Delta modem firmware
- 2 - Full modem firmware (requires *CONFIG_SM_DFU_MODEM_FULL*)

- 3 - MCUboot second-stage bootloader self-update. Only available when the device uses the NSIB (B0) and MCUboot as a second-stage bootloader. The host must stream the signed image for the **inactive** `s0`/`s1` slot.

Caution:

Full modem firmware DFU is disabled by default.

If the full modem firmware update fails after firmware data has been programmed to modem flash, the modem remains unusable until a successful update is completed.

3.5.2 DFU initialize #XDFUINIT

The `#XDFUINIT` command initializes the DFU target for a specific image type.

Set command

The set command initializes the DFU target.

Syntax

```
AT#XDFUINIT=<type>[,<size>]
```

The parameters and their defined values are the following:

<type>
Integer. The DFU image type as specified in the *DFU types* section.

<size>
Integer. The total firmware image size in bytes. Required for application (type `0`), delta modem firmware (type `1`), and MCUboot bootloader (type `3`) updates.

For full modem firmware (type 2), the command triggers an immediate reboot into bootloader mode.

The `#XDFUINIT`, `#XDFUWRITE`, and `#XDFUAPPLY` commands use the same modem DFU area as the `#XFOTA` and `#XNRFCLOUDFOTA` commands. For delta modem firmware (type `1`), they return `ERROR` in one of the following cases:

- A FOTA update started with the `#XFOTA` or `#XNRFCLOUDFOTA` command is in progress.
- Such an update is downloaded, but not yet activated.

When one of these commands succeeds, it reserves the modem DFU area for direct DFU until the modem is reset, for example with the `#XMODEMRESET` or `#XRESET` command. While the area is reserved, the `#XFOTA` and `#XNRFCLOUDFOTA` commands cannot start a modem firmware update or erase the modem DFU area.

Example

```
AT#XDFUINIT=0,123456
OK

AT#XDFUINIT=2
// device reboots into bootloader mode
Bootloader mode ready
```

Read command

The read command is not supported.

Test command

The test command tests the existence of the command and provides information about the type of its subparameters.

Syntax

```
AT#XDFUINIT=?
```

Response syntax

```
#XDFUINIT: <list of supported types>,<size>
```

Example

```
AT#XDFUINIT=?

#XDFUINIT: (0,1,2,3),<size>

OK
```

3.5.3 DFU write #XDFUWRITE

The #XDFUWRITE command writes firmware data to the DFU target. The command enters data mode to receive the firmware chunk.

Set command

The set command initiates a firmware data write operation.

Syntax

```
AT#XDFUWRITE=<type>,<addr>,<len>
```

The parameters and their defined values are the following:

<type>
Integer. The DFU image type as specified in the *DFU types* section.

<addr>
Integer. The address offset for the data.

<len>
Integer. The length of the data chunk to write.

After the command returns `OK`, the Serial Modem enters data mode to receive exactly `<len>` bytes of firmware data. When the data has been received and written, the Serial Modem sends a `#XDFU` unsolicited notification with the status. For delta modem firmware (type `1`), the command returns `ERROR` under the same conditions as the `#XDFUINIT` command.

Example

```
AT#XDFUWRITE=0,0,4096
OK
// 4096 bytes of firmware data
#XDATAMODE: 0
#XDFU: 0,1,0
```

Read command

The read command is not supported.

Test command

The test command tests the existence of the command and provides information about the type of its subparameters.

Syntax

```
AT#XDFUWRITE=?
```

Response syntax

```
#XDFUWRITE: <list of supported types>,<addr>,<len>
```

Example

```
AT#XDFUWRITE=?

#XDFUWRITE: (0,1,2,3),<addr>,<len>

OK
```

3.5.4 DFU apply #XDFUAPPLY

The #XDFUAPPLY command finalizes the firmware update and schedules it for activation.

Set command

The set command applies the firmware update.

Syntax

```
AT#XDFUAPPLY=<type>
```

The parameters and their defined values are the following:

<type>
Integer. The DFU image type as specified in the *DFU types* section.

For application (type 0), delta modem firmware (type 1), and MCUboot bootloader (type 3), the update is scheduled and will be activated on the next reset, which can be done with the AT#XRESET command. For full modem firmware (type 2), the command applies the current segment (bootloader or firmware) and triggers a reboot if needed. For delta modem firmware (type 1), the command returns ERROR under the same conditions as the #XDFUINIT command.

Example

```
AT#XDFUAPPLY=0
OK
#XDFU: 0,2,0

AT#XRESET
OK
Ready
```

Read command

The read command is not supported.

Test command

The test command tests the existence of the command and provides information about the type of its subparameters.

Syntax

```
AT#XDFUAPPLY=?
```

Response syntax

```
#XDFUAPPLY: <list of supported types>
```

Example

```
AT#XDFUAPPLY=?

#XDFUAPPLY: (0,1,2,3)

OK
```

3.5.5 DFU unsolicited notification #XDFU

The Serial Modem sends the #XDFU unsolicited notification to report the status of DFU write and apply operations.

Unsolicited notification

```
#XDFU: <type>,<operation>,<status>
```

The parameters and their defined values are the following:

<type>
Integer. The DFU image type as specified in the *DFU types* section.

<operation>

- 1 - Data write completed.
- 2 - Apply update completed.

<status>

- 0 - Success.
- -1 - Failure.

3.5.6 Complete DFU examples

The following sections showcase a complete example of application, delta modem, and full modem firmware update.

Application firmware update

The following example shows a complete application firmware update:

```
// Note the current application version (major.minor.patch from app/
VERSION)
AT#XSMVER
```

(continues on next page)

(continued from previous page)

```
#XSMVER: "v1.99.0-7-gdf00f46-dirty","3.3.0-rc1"
OK

// Initialize DFU for application firmware (total size 123456 bytes)
AT#XDFUINIT=0,123456
OK

// Write first chunk (4096 bytes at offset 0)
AT#XDFUWRITE=0,0,4096
OK
// 4096 bytes of firmware data
#XDATAMODE: 0
#XDFU: 0,1,0

// Write second chunk (4096 bytes at offset 4096)
AT#XDFUWRITE=0,4096,4096
OK
// 4096 bytes of firmware data
#XDATAMODE: 0
#XDFU: 0,1,0

// ... continue writing chunks ...

// Apply the update
AT#XDFUAPPLY=0
OK
#XDFU: 0,2,0

// Reset to activate the new firmware
AT#XRESET
OK
Ready

// Verify the VERSION prefix increased (here 1.99.0 -> 1.99.1)
AT#XSMVER
#XSMVER: "v1.99.1-8-gf0fd0f46-dirty","3.3.0-rc1"
OK
```

Delta modem firmware update

The following example shows a complete delta modem firmware update:

```
// Note the current modem firmware
AT%SHORTSWVER
%SHORTSWVER: nrf91x1_2.0.0
OK
```

(continues on next page)

(continued from previous page)

```
// Initialize DFU for delta modem firmware (total size 65536 bytes)
AT#XDFUINIT=1,65536
OK

// Write first chunk (4096 bytes at offset 0)
AT#XDFUWRITE=1,0,4096
OK
// 4096 bytes of firmware data
#XDATAMODE: 0
#XDFU: 1,1,0

// Write second chunk (4096 bytes at offset 4096)
AT#XDFUWRITE=1,4096,4096
OK
// 4096 bytes of firmware data
#XDATAMODE: 0
#XDFU: 1,1,0

// ... continue writing chunks ...

// Apply the update
AT#XDFUAPPLY=1
OK
#XDFU: 1,2,0

// Reset the modem to activate the new firmware
AT#XMODEMRESET
#XMODEMRESET: 0
OK

// Verify that the modem is running the new firmware
AT%SHORTSWVER
%SHORTSWVER: nrf91x1_2.0.4
OK
```

Full modem firmware update

The following example shows a complete full modem firmware update.

For full modem firmware update, enable the *CONFIG_SM_DFU_MODEM_FULL* Kconfig option.

Important:

If the update fails after the first 8 kB block of firmware data has been programmed, the modem remains unusable until a successful update is performed. The bootloader remains intact for retries. Ensure a reliable connection between the host and Serial Modem, and stable power supply before starting the update.

The full modem update consists of two phases:

1. Bootloader segment - Writes the bootloader data.
2. Firmware segment - Writes the firmware data.

Note:

Firmware segment data is buffered and programmed to modem flash in 8 kB blocks. A `#XDFU: 2,1,0` notification means the data was accepted, not that it was programmed. With contiguous writes of 8 kB or less, the first block is programmed during the write that brings the total amount of firmware data to 8 kB or more. If that write fails, the modem rejected the firmware, for example, because it is an older version. In that case, the existing modem firmware is left untouched, and you can return to normal mode with the `AT#XRESET` command. If a later write or `AT#XDFUAPPLY` fails, the modem remains unusable until a full modem firmware update is completed successfully.

```
// Note the current firmware version of the modem
AT%SHORTSWVER
%SHORTSWVER: nrf91x1_2.0.0
OK

// Initialize DFU for full modem firmware
AT#XDFUINIT=2
// device reboots
Bootloader mode ready

// Phase 1: Write bootloader segment
AT#XDFUWRITE=2,0,4096
OK
// 4096 bytes of bootloader data
#XDATAMODE: 0
#XDFU: 2,1,0

// ... continue writing bootloader chunks ...

// Apply bootloader segment (switches to firmware segment mode)
AT#XDFUAPPLY=2
OK
#XDFU: 2,2,0

// Phase 2: Write firmware segments
// Warning: Data is programmed to modem flash in 8 kB blocks. A
failure after the first block is programmed leaves the modem
unusable until the update is completed successfully.
AT#XDFUWRITE=2,0,4096
OK
// 4096 bytes of firmware data
#XDATAMODE: 0
```

(continues on next page)

(continued from previous page)

```
#XDFU: 2,1,0

// ... continue writing firmware chunks ...

// Apply firmware segment (triggers reboot)
AT#XDFUAPPLY=2
OK
#XDFU: 2,2,0
// device reboots
Ready

// Verify that the modem is running the new firmware
AT%SHORTSWVER
%SHORTSWVER: nrf91x1_2.0.4
OK
```

MCUboot firmware update

Important:

NSIB (B0) uses monotonic counter values to prevent rollback of the MCUboot bootloader. When performing an MCUboot DFU, the host must ensure that the new MCUboot image has a strictly higher monotonic counter value than the currently running MCUboot to avoid update failure.

This means that updates for MCUboot images are limited to the number of slots defined in the `SB_CONFIG_SECURE_BOOT_NUM_VER_COUNTER_SLOTS` Kconfig option. In Serial Modem, it is set to `40` by default.

The following example shows a complete MCUboot firmware update:

```
// Query the current slot for MCUboot image
AT#XBOOTINFO=1
#XBOOTINFO: 0
OK

// Download the correct MCUboot image for the inactive slot.
// The default NCS image names for MCUboot are:
// - Slot 0: signed_by_mcuboot_and_b0_mcuboot.bin
// - Slot 1: signed_by_mcuboot_and_b0_s1_image.bin

// In this example, the device is currently running slot 0, so the
host should download the signed image for slot 1, which is signed_
by_mcuboot_and_b0_s1_image.bin.

// Note the current monotonic counter value for MCUboot
```

(continues on next page)

(continued from previous page)

```
AT#XBOOTINFO=0
#XBOOTINFO: 1
OK

// Initialize DFU for MCUboot firmware (total size 23456 bytes)
AT#XDFUINIT=3,23456
OK

// Write first chunk (4096 bytes at offset 0)
AT#XDFUWRITE=3,0,4096
OK
// 4096 bytes of firmware data
#XDATAMODE: 0
#XDFU: 3,1,0

// Write second chunk (4096 bytes at offset 4096)
AT#XDFUWRITE=3,4096,4096
OK
// 4096 bytes of firmware data
#XDATAMODE: 0
#XDFU: 3,1,0

// ... continue writing chunks ...

// Apply the update
AT#XDFUAPPLY=3
OK
#XDFU: 3,2,0

// Reset to activate the new firmware
AT#XRESET
OK
Ready

// Verify that the monotonic counter value for MCUboot was updated
AT#XBOOTINFO=0
#XBOOTINFO: 2
OK

// (Optional) Verify that the MCUboot slot was switched
AT#XBOOTINFO=1
#XBOOTINFO: 1
OK
```

3.5.7 DFU host example

See `app/scripts/sm_dfu_host.py` for an example of how to implement a DFU host. The script demonstrates the DFU AT command flow and can be used as a starting point

for custom implementations.

Install dependencies:

```
pip install -r app/scripts/requirements.txt
```

Note:

cbor2 5.8.0 does not work with the `sm_dfu_host.py` script due to a known bug introduced in that version. Versions older and newer than that work. See cbor2 bug for more information.

Usage

```
python sm_dfu_host.py --port <serial_port> --baudrate <baud> --file
<path-to-firmware> --type <dfu_type>
```

Where `<dfu_type>` is one of: `application`, `modem-delta`, or `modem-full`.

After the DFU transfer completes:

Note:

Applying application and delta modem firmware can take some time.

- For application updates, reset the device to activate the new firmware:

  ```
  python sm_dfu_host.py --port <serial_port> --baudrate <baud> --
  reset
  ```

- For delta modem updates, reset the modem to apply the update:

  ```
  python sm_dfu_host.py --port <serial_port> --baudrate <baud> --
  modem-reset
  ```

- Full modem updates handle the reset automatically.

To check if the device is operating in normal mode:

```
python sm_dfu_host.py --port <serial_port> --baudrate <baud> --ping
```

Note:

Ping will return an error if the device is in bootloader mode.

3.6 FOTA AT commands

This page describes AT commands related to Firmware Over-The-Air (FOTA) updates.

3.6.1 FOTA request #XFOTA

The #XFOTA command sends various types of FOTA requests, based on specific operation codes.

Set command

The set command performs FOTA-related operations.

Syntax

```
AT#XFOTA=<op>[,<file_url>[,<sec_tag>[,<pdn_id>]]]
```

The parameters and their defined values are the following:

<op>

- 0 - Stop FOTA (during download only). Can be used as a way to pause and resume the download, by issuing the same FOTA start command to resume.
- 1 - Start FOTA for application update.
- 2 - Start FOTA for modem delta update.
- 3 - Start FOTA for full modem update. Can only be used when the `full-fota.conf` configuration file and `full-fota-nrf9151.overlay` devicetree overlay are used.
- 5 - Start FOTA for MCUboot second-stage bootloader update.

  Only available when the device uses the NSIB (B0) and MCUboot as a second-stage bootloader.

  Not supported on the Thingy:91 X.

- 7 - Read modem DFU area size and firmware image offset (for modem delta update).
- 9 - Erase modem DFU area (for modem delta update).

<file_url>

String. The full HTTP or HTTPS path of the target image to download. Must be provided to the start operations. For the MCUboot bootloader update, the target image must be the signed image for the **inactive** `s0/s1` slot. You can use the AT#XBOOTINFO=1 command to query which slot is active.

<sec_tag>

Integer. Indicates to the modem the credential of the security tag used for establishing a secure connection for downloading the image. It is associated with the certificate or PSK. Must be provided to the start operations when using HTTPS.

<pdn_id>
Integer. The Packet Data Network (PDN) ID that can be used instead of the default PDN for downloading. Can be provided to the start operations.

> **Note:**
>
> For historical reasons, this is the PDN ID and not the `cid` of the `+CGDCONT` command, which most other commands use. You can get the PDN ID of a `cid` with the `AT%XGETPDNID=<cid>` command.

> **Note:**
>
> Raw sockets must not use the PDN connection at the same time. See *Socket AT commands* for more information.

After the `#XFOTA: 4,0` notification indicating that the FOTA update is downloaded, it must be activated by resetting either the whole device with the `AT#XRESET` command, or in case of modem update only the modem with the `AT#XMODEMRESET` command.

> **Note:**
>
> During a delta update to the modem, the modem DFU area is automatically erased as needed as part of the update process. Erasing this area manually is optional.
>
> However, this leads to the command starting the update taking longer to complete, and also leaves the connection to the FOTA server idle while the area is being erased, which can provoke issues.

> **Note:**
>
> The firmware image is stored in external flash memory during a full modem update. The external flash is erased automatically after a new firmware activation.
>
> Activating the new modem firmware is done identically to a modem delta update, by resetting either the whole device or only the modem.

> **Important:**
>
> For the MCUboot bootloader update, NSIB (B0) uses monotonic counter values to prevent rollback. When performing an MCUboot FOTA, the host must ensure that the new MCUboot image has a strictly higher monotonic counter value than the currently running MCUboot to avoid update failure.
>
> This means that updates for MCUboot images are limited to the number of slots defined in the `SB_CONFIG_SECURE_BOOT_NUM_VER_COUNTER_SLOTS` Kconfig option. In Serial Modem, it is set to `40` by default.

Response syntax

```
AT#XFOTA=7
#XFOTA: <size>,<offset>
```

The parameters and their defined values are the following:

<size>
Integer. The size of the modem DFU area in bytes.

<offset>
Integer. The offset of the firmware image in the modem DFU area. It is `0` if no image is in the modem DFU area, and `2621440` (`NRF_MODEM_DELTA_DFU_OFFSET_DIRTY`) if the modem DFU area needs to be erased before a new firmware update can be received.

Example

```
application firmware download and update
AT#XFOTA=1,"http://remote.host/fota/sm_app_update.bin"
OK
#XFOTA: 1,0,0
...
#XFOTA: 4,0
AT#XRESET
OK
Ready
#XFOTA: 5,0

read modem DFU info
AT#XFOTA=7
#XFOTA: 294912,0
OK

modem firmware download and update
AT#XFOTA=2,"http://remote.host/fota/mfw_nrf9151_update_from_2.0.2_
to_2.0.3.bin"
OK
#XFOTA: 1,0,0
...
#XFOTA: 4,0
AT#XMODEMRESET
#XFOTA: 5,0
#XMODEMRESET: 0
OK

read modem DFU info
AT#XFOTA=7
#XFOTA: 294912,2621440
OK
```

(continues on next page)

(continued from previous page)

```
erase modem DFU area for next modem delta update (optional)
AT#XFOTA=9
OK

MCUboot bootloader update
AT#XBOOTINFO=1
#XBOOTINFO: 0
OK
AT#XFOTA=5,"https://remote.host/fota/signed_by_mcuboot_and_b0_s1_
image.bin",42
OK
#XFOTA: 1,0,0
...
#XFOTA: 4,0
AT#XRESET
OK
Ready
#XFOTA: 5,0
```

Unsolicited notification

```
#XFOTA: <fota_stage>,<fota_status>[,<fota_info>]
```

The parameters and their defined values are the following:

<fota_stage>

- `0` - Init.
- `1` - Download.
- `2` - Download, erase pending (modem delta update only).
- `3` - Download, erase complete (modem delta update only).
- `4` - Downloaded, to be activated.
- `5` - Complete.

<fota_status>

- `0` - OK.
- `1` - Error.
- `2` - Cancelled.
- `3` - Reverted (application FOTA only).

<fota_info>

Integer. Its value can have different meanings based on the values returned by `<fota_stage>` and `<fota_status>`.

See the following table:

<fota_stage>	<fota_status>	<fota_info>
1 (namely *Download*)	0 (namely *OK*)	Percentage of the download
1 (namely *Download*)	1 (namely *ERROR*)	Error Code
1 (namely *Download*)	2 (namely *CANCELLED*)	0 - Downloading is cancelled before completion
5 (namely *Complete*)	1 (namely *ERROR*)	Error Code

The error codes can be the following:

- 1 - Download failed
- 2 - Update image rejected (for example modem firmware version error)
- 3 - Update image mismatch (for example <op> is 1 but <file_url> points to a modem image)

For modem FOTA, the error codes can be the following:

- 71303169 (NRF_MODEM_DFU_RESULT_INTERNAL_ERROR) - The modem encountered a fatal internal error during firmware update.
- 71303170 (NRF_MODEM_DFU_RESULT_HARDWARE_ERROR) - The modem encountered a fatal hardware error during firmware update.
- 71303171 (NRF_MODEM_DFU_RESULT_AUTH_ERROR) - Modem firmware update failed due to an authentication error.
- 71303172 (NRF_MODEM_DFU_RESULT_UUID_ERROR) - Modem firmware update failed due to UUID mismatch.
- 71303173 (NRF_MODEM_DFU_RESULT_VOLTAGE_LOW) - Modem firmware update not executed due to low voltage. The modem will retry the update on reboot.

Read command

The read command is not supported.

Test command

The test command tests the existence of the command and provides information about the type of its subparameters.

Syntax

```
AT#XFOTA=?
```

Response syntax

```
#XFOTA: <list of op value>,<file_url>,<sec_tag>,<pdn_id>
```

Examples

```
AT#XFOTA=?

#XFOTA: (0,1,2,3,5,7,9),<file_url>,<sec_tag>,<pdn_id>

OK
```

3.7 GNSS AT commands

This page describes GNSS-related AT commands.

3.7.1 Control GNSS #XGNSS

The #XGNSS command controls the GNSS.

Set command

The set command allows you to start and stop the GNSS module.

The nRF Cloud A-GNSS service can be used with the module to reduce the time it takes for the GNSS module to estimate device position.

Both the nRF Cloud A-GNSS service and the nRF Cloud P-GPS service can be used with the module, either together or independently of each other. Using them reduces the time it takes for the GNSS module to estimate device position.

To use either of them, the device must be connected to nRF Cloud (using the *#XNRFCLOUD* AT command) when starting the GNSS module. In addition, the following Kconfig options must be enabled:

- `CONFIG_NRF_CLOUD_AGNSS` to use A-GNSS.
- `CONFIG_NRF_CLOUD_PGPS` to use P-GPS. This also requires applying the `pgps-nrf9151.overlay` devicetree overlay to both the application and the `mcuboot` image (using the absolute path for the latter).

If both assistive services were enabled during compilation, you cannot choose to use only one of them at run time.

Syntax

```
AT#XGNSS=<op>,<cloud_assistance>,<interval>[,<timeout>]
```

The parameters and their defined values are the following:

<op>

- `0` - Stop the GNSS module. In this case, no other parameter is allowed.
- `1` - Start the GNSS module.

<cloud_assistance>

- `0` - Disabled.
- `1` - Enabled.

<interval>

Integer. The GNSS fix interval in seconds. It can have one of the following values:

- `0` - Single-fix navigation mode.
- `1` - Continuous navigation mode. The fix interval is set to 1 second.
- Ranging from `10` to `65535` - Periodic navigation mode. The fix interval is set to the specified value.

<timeout>

Integer. The maximum time in seconds that the GNSS receiver is allowed to run while trying to produce a valid Position, Velocity, and Time (PVT) estimate. Can only be specified in single-fix and periodic navigation modes, and must be omitted in continuous navigation mode. If omitted in single-fix or periodic navigation modes, the default value is 60 seconds.

- `0` - The GNSS receiver runs indefinitely until a valid PVT estimate is produced.
- Any positive integer - The GNSS receiver is turned off after the specified time is up, even if a valid PVT estimate was not produced.

In periodic navigation mode, the `<interval>` and `<timeout>` parameters are temporarily ignored during the first fix.

Note:

When `<cloud_assistance>` is disabled, no request is made to nRF Cloud for assistance data. However, if it has been previously enabled and used, such data may remain locally available and will be used if still valid.

Note:

Make sure that the GNSS antenna is configured properly. You may need to issue the `AT%XMAGPIO` and `AT%XCOEX0` AT commands to configure the Low-Noise Amplifier (LNA), depending on the hardware (HW).

If you want to use the nRF9151 DK with an onboard antenna, use the `AT%XCOEX0=1,1,1565,1586` command.

Tip:

When the LTE link is enabled, make sure to have either Power Saving Mode (PSM) or extended Discontinuous Reception (eDRX) enabled to give the GNSS receiver the time

it needs to acquire fixes.

Tip:

The Serial Modem logs NMEA and PVT data when trying to acquire fixes, which can help with troubleshooting. These logs are enabled if the `CONFIG_SM_LOG_LEVEL_DBG` Kconfig option is set and logs have been enabled with `AT#XLOG`.

Note:

See the documentation for the nRF Cloud A-GNSS and nRF Cloud P-GPS libraries for information on how to best configure and use A-GNSS and P-GPS, respectively.

Note:

When using P-GPS, make sure that the value of the *CONFIG_SM_PGPS_INJECT_FIX_DATA* Kconfig option matches your use case. It is enabled by default but should be disabled if the device is expected to move distances longer than a few dozen kilometers between fix attempts.

Note:

When using P-GPS, apply the `pgps-nrf9151.overlay` devicetree overlay to both the application and the `mcuboot` image to provide a dedicated flash partition for P-GPS prediction data storage and to ensure MCUboot operates with the same partition layout. The absolute path to the overlay file must be provided when passing it to the `mcuboot` image using **`mcuboot_EXTRA_DTC_OVERLAY_FILE`**. See *Configuration files* for more information.

As an alternative to GNSS-based positioning, see *#XNRFCLOUDPOS* for cellular and Wi-Fi® positioning.

Unsolicited notification `#XGNSSPOS` notification indicates that a new position fix has been acquired.

```
#XGNSSPOS: <latitude>,<longitude>,<altitude>,<accuracy>,<speed>,
<heading>,<datetime>
```

The parameters and their defined values are the following:

<latitude>
Float. The latitude in degrees.

<longitude>
Float. The longitude in degrees.

<altitude>
Float. The altitude above the WGS-84 ellipsoid in meters.

<accuracy>
Float. The accuracy (2D 1-sigma) in meters.

<speed>
Float. The horizontal speed in meters per second.

<heading>
Float. The heading of the movement of the user in degrees.

<datetime>
String. The UTC date-time.

`#XGNSS` notification indicates GNSS status.

```
#XGNSS: <gnss_service>,<gnss_status>
```

The parameters and their defined values are the following:

<gnss_service>
Integer. When it has the value `1`, it means that GNSS is supported in `%XSYSTEMMODE` and activated in `+CFUN`.

<gnss_status>

- `0` - GNSS is stopped.
- `1` - GNSS is started.
- `2` - GNSS wakes up in periodic mode.
- `3` - GNSS enters sleep because of timeout.
- `4` - GNSS enters sleep because a fix is acquired.

Example Get GNSS fix in GNSS only mode:

```
AT%XSYSTEMMODE=0,0,1,0

OK
AT+CFUN=31

OK
AT#XGNSS=1,0,0,0

#XGNSS: 1,1

OK

#XGNSSPOS: 35.457576,139.625090,121.473785,22.199919,0.442868,0.
```

(continues on next page)

(continued from previous page)

```
000000,"2021-06-02 06:25:48"

#XGNSS: 1,4
```

Set PSM and eDRX configurations, use GNSS assistance from nRF Cloud and get GNSS fixes in continuous navigation mode:

```
AT%XSYSTEMMODE=1,0,1,0

OK
AT+CPSMS=1,,,"00000001","00000011"

OK
AT+CEDRXS=2,4,"0011"

OK
AT+CFUN=1

OK

+CEDRXP: 4,"0011","0011","0011"

AT#XNRFCLOUD=1

OK
#XNRFCLOUD: 1,0
AT#XGNSS=1,1,1

#XGNSS: 1,1

OK

#XGNSSPOS: 35.457417,139.625211,162.850952,15.621976,1.418092,0.
000000,"2021-06-02 05:21:31"

#XGNSSPOS: 35.457435,139.625348,176.104797,14.245458,1.598184,69.
148659,"2021-06-02 05:21:32"

#XGNSSPOS: 35.457417,139.625415,179.132980,13.318132,1.235241,69.
148659,"2021-06-02 05:21:33"

#XGNSSPOS: 35.457410,139.625469,181.223541,12.667312,0.803951,69.
148659,"2021-06-02 05:21:34"
AT#XGNSS=0
OK

#XGNSS: 1,0
```

Set PSM and eDRX configurations, use GNSS assistance from nRF Cloud and get GNSS fixes in periodic navigation mode every 10 seconds:

```
AT%XSYSTEMMODE=1,0,1,0

OK
AT+CPSMS=1,,,"00000001","00000011"

OK
AT+CEDRXS=2,4,"0011"

OK
AT+CFUN=1

OK
AT#XNRFCLOUD=1

OK
#XNRFCLOUD: 1,0
AT#XGNSS=1,1,10

#XGNSS: 1,1

OK

#XGNSS: 1,2

#XGNSSPOS: 35.457243,139.625435,149.005020,28.184258,10.431827,281.
446014,"2021-06-24 04:35:52"

#XGNSS: 1,4

#XGNSS: 1,2

#XGNSSPOS: 35.457189,139.625602,176.811203,43.015198,0.601837,281.
446014,"2021-06-24 04:36:32"

#XGNSS: 1,4

#XGNSS: 1,2

#XGNSSPOS: 35.457498,139.625422,168.243591,31.753956,0.191195,281.
446014,"2021-06-24 04:37:12"

#XGNSS: 1,4

#XGNSS: 1,2

#XGNSSPOS: 35.457524,139.624667,100.745979,25.324850,6.347160,94.
699837,"2021-06-24 04:37:52"
```

(continues on next page)

(continued from previous page)

```
#XGNSS: 1,4
AT#XGNSS=0

#XGNSS: 1,0

OK
```

Read command

The read command allows you to check GNSS support and its status.

Syntax

```
AT#XGNSS?
```

Response syntax

```
#XGNSS: <gnss_service>,<gnss_status>
```

The parameters and their defined values are the following:

<gnss_service>
Integer. When it has the value `1`, it means that GNSS is supported in `%XSYSTEMMODE` and activated in `+CFUN`.

<gnss_status>

- `0` - GNSS is stopped.
- `1` - GNSS is started.
- `2` - GNSS wakes up in periodic mode.
- `3` - GNSS enters sleep because of timeout.
- `4` - GNSS enters sleep because a fix is acquired.

Example

```
AT#XGNSS?

#XGNSS: 1,1

OK
```

Test command

The test command tests the existence of the command and provides information about the type of its subparameters.

Syntax

```
AT#XGNSS=?
```

Example

```
AT#XGNSS=?

#XGNSS: (0,1),(0,1),<interval>,<timeout>

OK
```

3.7.2 Delete GNSS data #XGNSSDEL

The #XGNSSDEL command deletes GNSS data from non-volatile memory. This command should be issued when GNSS is activated but not started yet.

Note:

This is a debug feature, and is not supposed to be used in production code.

Set command

The set command allows you to delete cached GNSS data. Using this command does not trigger A-GNSS nor P-GPS data request events. The execution of this command may delay the full functionality of A-GNSS and P-GPS until the next periodic data request has been received.

Syntax

```
AT#XGNSSDEL=<mask>
```

The parameters and their defined values are the following:

<mask>
Integer. A bitwise OR of the following bitmasks:

- `0x001` - Ephemerides.
- `0x002` - Almanacs (excluding leap second and ionospheric correction).
- `0x004` - Ionospheric correction parameters.
- `0x008` - Last good fix (the last position).
- `0x010` - GPS time-of-week (TOW).
- `0x020` - GPS week number.
- `0x040` - Leap second (UTC parameters).
- `0x100` - Precision estimate of GPS time-of-week (TOW).

- `0x400` - Extended Kalman Filter (EKF) state (last position/velocity/time solution).

For example, the integer value with all of the above bits enabled will be `1407` (`0x57f`).

Example

```
AT%XSYSTEMMODE=0,0,1,0
OK
AT+CFUN=31
OK
AT#XGNSSDEL=1407
OK
AT+CFUN=0
OK
```

Read command

The read command is not supported.

Test command

The test command tests the existence of the command and provides information about the type of its subparameters.

Syntax

```
AT#XGNSSDEL=?
```

Example

```
AT#XGNSSDEL=?

#XGNSSDEL: <mask>

OK
```

3.8 HTTP client AT commands

This page describes AT commands for the HTTP client. The HTTP client operates on sockets managed by the *Socket AT commands*. You can perform the following using the Socket AT commands:

- Create a socket with `AT#XSOCKET` or `AT#XSSOCKET`.
- Connect it with `AT#XCONNECT`.
- Set socket options with `AT#XSOCKETOPT` or `AT#XSSOCKETOPT`.
- Close with `AT#XCLOSE`.

Request idle timeout

Every active HTTP request has a sliding idle timeout controlled by the *CONFIG_SM_HTTPC_RESPONSE_TIMEOUT_MS* Kconfig option (default 30 seconds).

The timer resets each time data is sent or received:

- Sending request headers or body upload chunks (POST/PUT data mode).
- Receiving response headers or body bytes.
- Pulling a body chunk in manual mode (AT#XHTTPCDATA).

If no such activity occurs within the configured window, the request is aborted and `#XHTTPCSTAT: <handle>,-1,<total_bytes>,<connection_close>` is emitted. The timeout is enforced by a background timer that fires independently of normal socket poll events, so a server that stalls silently (no TCP RST or FIN) is also detected.

3.8.1 HTTP request #XHTTPCREQ

The #XHTTPCREQ command starts an asynchronous HTTP or HTTPS request on a socket connected with AT#XCONNECT.

Set command

The set command starts an HTTP or HTTPS request.

Syntax

```
AT#XHTTPCREQ=<handle>,<url>,<method>[,<auto_reception>[,<format>[,
<body_len>[,<header 1>[,<header 2>[...]]]]]]
```

The parameters and their defined values are the following:

<handle>
Integer. Identifies the connected socket handle returned by AT#XSOCKET or AT#XSSOCKET, and connected by AT#XCONNECT.

<url>
String. The full URL of the request, for example, `http://host/path` or `https://host/path`.

<method>

- 0 - GET.
- 1 - POST.
- 2 - PUT.
- 3 - DELETE.
- 4 - HEAD.

<auto_reception>
Integer. Optional. When omitted, automatic reception is used.

- `1` - Automatic mode (default). Response body bytes are received and forwarded to the host automatically.
- `0` - Manual mode. Response body reception is paused after the headers are parsed and `#XHTTPCHEAD` is emitted. The host must then retrieve body data in chunks using `AT#XHTTPCDATA` commands.

<format>
Integer. Optional. Controls the encoding used to deliver response body bytes to the host.

- `0` - Binary (default). Response body bytes are forwarded to the host as raw binary.
- `1` - Hex string. Response body bytes are encoded as a lower-case ASCII hex string before delivery. Each raw byte becomes two hex characters. The `<length>` field in `#XHTTPCDATA` always reports the raw byte count. The actual data delivered is `2×<length>` ASCII hex characters. Applies to both automatic and manual reception modes.

<body_len>
Integer. Optional. Required as a placeholder when `<header>` parameters follow.

- `0` - No request body.
- Positive integer - Length of the request body in bytes. Valid for POST and PUT only. When set, the HTTP request headers are sent to the server immediately when the AT command is processed. The command then responds with `OK` and enters data mode. Body bytes are forwarded to the server as they are received from the host in data mode. The host must send exactly this many bytes as the request body. Data mode exits and `#XDATAMODE: 0` is reported when all bytes have been processed.

<header X>
String. Optional. An additional HTTP request header. Empty parameters between headers (`<header 1>`, `<header 2>`,, `<header 3>`, ...) are ignored.

Note:

The HTTP client uses HTTP/1.1 and relies on the protocol default of persistent (keep-alive) connections. Multiple sequential requests can therefore be made on the same connected socket without reconnecting. To close the connection after a response, pass `"Connection: close"` as a custom header.

Response syntax

```
#XHTTPCREQ: <handle>
OK
```

The parameters and their defined values are the following:

<handle>
Integer. Identifies the socket.

Unsolicited notification #XHTTPCHEAD is emitted when response headers have been parsed:

```
#XHTTPCHEAD: <handle>,<status_code>,<content_length>
```

The parameters and their defined values are the following:

<handle>
Integer. Identifies the socket.

<status_code>
Integer. The HTTP status code returned by the server.

<content_length>
Integer. The value of the `Content-Length` response header, or `-1` when the server uses chunked transfer encoding or does not provide a content length.

#XHTTPCDATA is emitted in automatic mode for each received body chunk:

```
#XHTTPCDATA: <handle>,<offset>,<length>
<data>
```

The notification line is terminated with `\r\n`, the response data follows immediately, and a `\r\n` terminator follows the data bytes. In binary mode (`<format>=0`, default), the data is `<length>` raw bytes. In hex mode (`<format>=1`), the data is `2×<length>` lower-case ASCII hex characters.

The parameters and their defined values are the following:

<handle>
Integer. Identifies the socket.

<offset>
Integer. The number of body bytes already delivered before this chunk.

<length>
Integer. The number of body bytes in this chunk.

Note:

When the server uses `Transfer-Encoding: chunked`, the body bytes delivered via #XHTTPCDATA include the raw chunked framing. Each chunk consists of a size line (hexadecimal length followed by `\r\n`), the chunk data, and a trailing `\r\n`. The final zero-length chunk `0\r\n\r\n` is also forwarded. The host strips this framing to recover the original body content. The `<total_bytes>` field in #XHTTPCSTAT reflects the raw wire byte count, including chunked framing, not the decoded body length.

#XHTTPCSTAT is emitted when the request completes, fails, or is cancelled:

```
#XHTTPCSTAT: <handle>,<status_code>,<total_bytes>,<connection_close>
```

The parameters and their defined values are the following:

<handle>
Integer. Identifies the socket.

<status_code>
Integer. The HTTP status code on success, or `-1` on failure, cancel, or timeout.

<total_bytes>
Integer. On successful completion, failure, or timeout, it contains the total number of response body bytes received by the HTTP client. For chunked transfer encoding this includes the raw framing bytes (chunk-size lines, `\r\n` separators, and the final `0\r\n\r\n` terminator). On cancel (`status_code=-1` from `AT#XHTTPCCANCEL` or `AT#XCLOSE`), or timeout, it contains the number of response body bytes already delivered to the host.

<connection_close>

- `0` - Keep-alive connection.
- `1` - The server includes a `Connection: close` header, indicating that the TCP connection will be closed after this response. The host must close the socket with `AT#XCLOSE` and open a new connection before issuing the next request.

Note:

For responses that carry no body by definition — `HEAD` requests, `1xx`, `204 No Content`, and `304 Not Modified` — `#XHTTPCHEAD` is emitted immediately followed by `#XHTTPCSTAT`. No `#XHTTPCDATA` notification is generated.

Examples The following examples demonstrate how to perform an HTTP GET request and receive the response in automatic or manual mode on a connected socket.

HTTP GET (automatic mode):

```
AT#XHTTPCREQ=0,<url>,0
#XHTTPCREQ: 0
OK

#XHTTPCHEAD: 0,200,261

#XHTTPCDATA: 0,0,261
<261 bytes>

#XHTTPCSTAT: 0,200,261,0
```

Note:

In automatic mode, a `\r\n` terminator is appended after the data bytes of each `#XHTTPCDATA` notification.

HTTP GET (manual mode):

```
AT#XHTTPCREQ=0,<url>,0,0
#XHTTPCREQ: 0
OK

#XHTTPCHEAD: 0,200,261

AT#XHTTPCDATA=0
#XHTTPCDATA: 0,0,261
<261 bytes>
OK

#XHTTPCSTAT: 0,200,261,0
```

HTTP GET with hex-encoded response body (`format=1`, automatic mode):

```
AT#XHTTPCREQ=0,<url>,0,1,1
#XHTTPCREQ: 0
OK

#XHTTPCHEAD: 0,200,12

#XHTTPCDATA: 0,0,12
48656c6c6f20576f726c6421

#XHTTPCSTAT: 0,200,12,0
```

The hex string and the line break after it represent the `2×<length>` hex characters followed by the `\r\n` terminator.

HTTP POST with JSON body and custom header:

```
AT#XHTTPCREQ=0,<url>,1,1,0,15,"Content-Type: application/json"
#XHTTPCREQ: 0
OK
{"key":"value"}
#XDATAMODE: 0

#XHTTPCHEAD: 0,200,432

#XHTTPCDATA: 0,0,432
<432 bytes>

#XHTTPCSTAT: 0,200,432,0
```

HTTP GET data in binary format with Range header (`body_len=0` is required as a placeholder when `<header>` parameters follow):

In this example the server returns `connection_close=1`, so the host must close and reopen the socket before the next request.

```
AT#XHTTPCREQ=0,<url>,0,1,0,0,"Range: bytes=0-127"
#XHTTPCREQ: 0
OK

#XHTTPCHEAD: 0,206,128

#XHTTPCDATA: 0,0,128
<128 bytes>

#XHTTPCSTAT: 0,206,128,0

AT#XHTTPCREQ=0,<url>,0,1,0,0,"Range: bytes=128-255"
#XHTTPCREQ: 0
OK

#XHTTPCHEAD: 0,206,128

#XHTTPCDATA: 0,0,128
<128 bytes>

#XHTTPCSTAT: 0,206,128,1

AT#XCLOSE=0
#XCLOSE: 0,0
OK
```

HTTP HEAD (no body — #XHTTPCSTAT follows immediately after #XHTTPCHEAD):

```
AT#XHTTPCREQ=0,<url>,4
#XHTTPCREQ: 0
OK

#XHTTPCHEAD: 0,200,261

#XHTTPCSTAT: 0,200,0,0
```

HTTP POST with chunked response (`content_length=-1`):

```
AT#XHTTPCREQ=0,<url>,1,1,0,1024
#XHTTPCREQ: 0
OK
<1024 bytes payload>
#XDATAMODE: 0

#XHTTPCHEAD: 0,200,-1

#XHTTPCDATA: 0,0,1132
```

(continues on next page)

(continued from previous page)

```
460\r\n{"args":{},"data":"<1024 bytes>","url":"..."}\r\n0\r\n\r\n

#XHTTPCSTAT: 0,200,1132,0
```

Note:

The 1132 raw bytes break down as chunked framing: `460\r\n` (chunk-size 1120 decimal in hex, 5 bytes), 1120 bytes of JSON body, `\r\n` (chunk trailer, 2 bytes), and `0\r\n\r\n` (final zero-length chunk, 5 bytes). Strip this framing to recover the 1120-byte JSON body.

Test command

The test command tests the existence of the command and provides information about the type of its subparameters.

Syntax

```
AT#XHTTPCREQ=?
```

Response syntax

```
#XHTTPCREQ: <handle>,<url>,<method>[,<auto_reception>[,<format>[,
<body_len>[,<header>]...]]]
```

Example

```
AT#XHTTPCREQ=?
#XHTTPCREQ: <handle>,<url>,<method>[,<auto_reception>[,<format>[,
<body_len>[,<header>]...]]]
OK
```

3.8.2 HTTP data pull #XHTTPCDATA

The #XHTTPCDATA command pulls the next body chunk for a manual-mode HTTP request.

Set command

The set command performs one non-blocking receive on the socket of a manual-mode request and forwards the available data to the host.

Syntax

```
AT#XHTTPCDATA=<handle>[,<length>]
```

The parameters and their defined values are the following:

<handle>
Integer. Identifies the socket used when starting the manual-mode request.

<length>
Integer. Optional. The maximum number of bytes to receive in this pull. When omitted, the full internal receive buffer size is used.

Response syntax When body data is available:

```
#XHTTPCDATA: <handle>,<offset>,<length>
<length bytes of raw body data>
OK
```

The `#XHTTPCDATA:` line is terminated with `\r\n`. `<length>` raw body bytes follow immediately with no additional separator. `OK` follows on its own line after the body bytes.

When the socket buffer is temporarily empty (EAGAIN):

```
#XHTTPCDATA: <handle>,<offset>,0
OK
```

The parameters and their defined values are the following:

<handle>
Integer. Identifies the socket.

<offset>
Integer. The number of body bytes already delivered before this pull.

<length>
Integer. The number of body bytes delivered in this pull. A value of `0` means the socket buffer is currently empty.

When all body bytes have been delivered, `#XHTTPCSTAT` is sent as a URC after the final `OK`. This happens either when the server closes the connection, when the `Content-Length` bytes have all been forwarded, or when the chunked transfer `0\r\n\r\n` terminator is received.

Note:

When the server uses `Transfer-Encoding: chunked` (`content_length=-1` in `#XHTTPCHEAD`), the bytes returned by `#XHTTPCDATA` include raw chunked framing. See the note under `#XHTTPCDATA` in the `AT#XHTTPCREQ` section.

Note:

Retry the pull after a brief delay when `#XHTTPCDATA: <handle>,<offset>,0` is received. Use `AT#XHTTPCCANCEL=<handle>` to cancel if no more data is needed.

Example

```
AT#XHTTPCDATA=0,256
#XHTTPCDATA: 0,0,256
<256 bytes>
OK

AT#XHTTPCDATA=0,256
#XHTTPCDATA: 0,256,256
<256 bytes>
OK

#XHTTPCSTAT: 0,200,512,0
```

Test command

The test command tests the existence of the command and provides information about the type of its subparameters.

Syntax

```
AT#XHTTPCDATA=?
```

Response syntax

```
#XHTTPCDATA: <handle>[,<length>]
```

Example

```
AT#XHTTPCDATA=?
#XHTTPCDATA: <handle>[,<length>]
OK
```

3.8.3 HTTP request cancel #XHTTPCCANCEL

The #XHTTPCCANCEL command cancels an active HTTP request.

Set command

The set command cancels an active request.

Syntax

```
AT#XHTTPCCANCEL=<handle>
```

The parameters and their defined values are the following:

<handle>
Integer. Identifies the socket of the request to cancel.

An unsolicited `#XHTTPCSTAT: <handle>,-1,<total_bytes>,<connection_close>` notification is emitted after cancellation, where `<total_bytes>` is the number of response body bytes already delivered to the host.

Note:

Closing the socket with `AT#XCLOSE` while a request is active also cancels the request and emits `#XHTTPCSTAT: <handle>,-1,<total_bytes>,<connection_close>` before the socket is freed.

Example

```
AT#XHTTPCCANCEL=0
OK
#XHTTPCSTAT: 0,-1,0,0
```

Test command

The test command tests the existence of the command and provides information about the type of its subparameters.

Syntax

```
AT#XHTTPCCANCEL=?
```

Response syntax

```
#XHTTPCCANCEL: <handle>
```

Example

```
AT#XHTTPCCANCEL=?
#XHTTPCCANCEL: <handle>
OK
```

3.9 ICMP AT commands

This page describes ICMP-related AT commands.

3.9.1 ICMP echo request #XPING

The `#XPING` command sends an ICMP Echo Request, also known as *Ping*.

Only one ping request can be active at a time.

Set command

The set command allows you to send an ICMP Echo Request.

Syntax

```
AT#XPING=<url>,<length>,<timeout>[,<count>[,<interval>[,<pdn>]]]
```

The parameters and their defined values are the following:

<url>
String. The hostname, the IPv4, or the IPv6 address of the target host.

<length>
Integer. The length of the ICMP payload, in bytes. The value range is `0` to `1472` for IPv4 and `0` to `1452` for IPv6. These limits come from the MTU of 1500 bytes minus the IP and ICMP headers.

<timeout>
Integer. The time to wait for each reply, in milliseconds. The value range is `0` to `4294967295`.

<count>
Integer. The number of echo requests to send. The default value is `1`. The value range is `1` to `65535`.

<interval>
Integer. The time to wait for sending the next echo request, in milliseconds. The default value is `1000`. The value range is `0` to `4294967295`.

<pdn>
Integer. Represents `cid` in the `+CGDCONT` command. The default value is `0`.

> **Note:**
>
> Other sockets cannot use the same PDN connection. See *Socket AT commands* for more information.

Unsolicited notification

```
#XPING: <response time> seconds
```

The parameters and their defined values are the following:

<response time>
Float. The elapsed time, in seconds, between the echo requests and the echo replies.

Example

```
AT#XPING="5.189.130.26",45,5000,5,1000
OK
```

(continues on next page)

(continued from previous page)

```
#XPING: 0.386 seconds
#XPING: 0.341 seconds
#XPING: 0.353 seconds
#XPING: 0.313 seconds
#XPING: 0.313 seconds
#XPING: average 0.341 seconds
AT#XGETADDRINFO="ipv6.google.com"
#XGETADDRINFO: "2404:6800:4006:80e::200e"
OK
AT#XPING="ipv6.google.com",45,5000,5,1000
OK
#XPING: 0.286 seconds
#XPING: 0.077 seconds
#XPING: 0.110 seconds
#XPING: 0.037 seconds
#XPING: 0.106 seconds
#XPING: average 0.123 seconds
AT#XPING="5.189.130.26",45,5000,5,1000,1
OK
#XPING: 1.612 seconds
#XPING: 0.349 seconds
#XPING: 0.334 seconds
#XPING: 0.278 seconds
#XPING: 0.278 seconds
#XPING: average 0.570 seconds
```

Read command

The read command is not supported.

Test command

The test command is not supported.

3.10 LwM2M carrier library AT commands

Note:

These AT commands are Experimental.

This page describes AT commands related to the LwM2M carrier library.

3.10.1 Carrier event #XCARRIEREVT

The #XCARRIEREVT is an unsolicited notification that indicates the event of the LwM2M carrier library.

Unsolicited notification

It indicates the event of the LwM2M carrier library.

Syntax

```
#XCARRIEREVT: <evt_type>,<param1>[,<param2>[,<param3>]..]
```

The parameters and their defined values are the following:

<evt_type>
Integer. The type of the event, followed by a variable number of parameters depending on the event type. These may be the following:

- `#XCARRIEREVT: 1,<status>`

  Request to use the `AT+CFUN=1` AT command to set the modem to full functional mode. `<status>` returns two possible values:

  - `0` - Request handling is fulfilled automatically.
  - `-1` - Request handling is deferred and must be fulfilled by the application at its earliest convenience.

- `#XCARRIEREVT: 2,<status>`

  Request to use the `AT+CFUN=4` AT command to set the modem to flight functional mode. `<status>` returns two possible values:

  - `0` - Request handling is fulfilled automatically.
  - `-1` - Request handling is deferred and must be fulfilled by the application at its earliest convenience.

- `#XCARRIEREVT: 3,<status>`

  Request to use the `AT+CFUN=0` AT command to set the modem to minimum functional mode. `<status>` returns two possible values:

  - `0` - Request handling is fulfilled automatically.
  - `-1` - Request handling is deferred and must be fulfilled by the application at its earliest convenience.

- `#XCARRIEREVT: 4,0`

  Bootstrap sequence complete.

- `#XCARRIEREVT: 5,0`

  Device registered successfully to the device management servers.

- `#XCARRIEREVT: 6,0`

  Device deregistered successfully from the device management servers.

- `#XCARRIEREVT: 7,<reason>,<timeout>`

  Client operation has been deferred.

  `<reason>` indicates the reason for the connection deferral and can return the following values:

  - 0 - LWM2M_CARRIER_DEFERRED_NO_REASON
  - 1 - LWM2M_CARRIER_DEFERRED_PDN_ACTIVATE
  - 2 - LWM2M_CARRIER_DEFERRED_BOOTSTRAP_NO_ROUTE
  - 3 - LWM2M_CARRIER_DEFERRED_BOOTSTRAP_CONNECT
  - 4 - LWM2M_CARRIER_DEFERRED_BOOTSTRAP_SEQUENCE
  - 5 - LWM2M_CARRIER_DEFERRED_SERVER_NO_ROUTE
  - 6 - LWM2M_CARRIER_DEFERRED_SERVER_CONNECT
  - 7 - LWM2M_CARRIER_DEFERRED_SERVER_REGISTRATION
  - 8 - LWM2M_CARRIER_DEFERRED_SERVICE_UNAVAILABLE
  - 9 - LWM2M_CARRIER_DEFERRED_SIM_MSISDN

  <timeout> indicates the time in seconds before the operation is resumed.

- #XCARRIEREVT: 8,0

  Firmware update started.

- #XCARRIEREVT: 9,0

  Firmware updated successfully.

- #XCARRIEREVT: 10,<status>

  Request to perform an application reboot, for example using the AT#XRESET AT command.

  <status> returns two possible values:

  - 0 - Request handling is fulfilled automatically.
  - -1 - Request handling is deferred and must be fulfilled by the application at its earliest convenience.

- #XCARRIEREVT: 11,0

  Modem domain event received.

- #XCARRIEREVT: 12,<type>,<URI>[,<data_length>\r\n<data>]

  Operation performed on the Binary App Data Container object (ID:19) or the App Data Container object (ID: 10250).

  <type> indicates the type of operation performed by the device management server:

  - 0 - A write request.
  - 1 - An observation start request.
  - 2 - An observation stop request.

  <URI> is a plain-text string in double quotes that describes the URI path that was targeted by the operation.

  <data_length> and <data> parameters are only applicable to notifications of <type> 0 (write).

<data_length> indicates the length in bytes of <data>.

<data> is a hexadecimal string in double quotes that contains the data written by the device management server to the indicated <URI> path.

- #XCARRIEREVT: 13,0

  Request to initialize the modem.

- #XCARRIEREVT: 14,0

  Request to shut down the modem.

- #XCARRIEREVT: 15,0

  The device error codes have been reset by the server.

- #XCARRIEREVT: 20,<type>,<value>

  LwM2M carrier library error occurred.

  <type> indicates the type of error and can return the following values:

  - 0 - LWM2M_CARRIER_ERROR_NO_ERROR
  - 1 - LWM2M_CARRIER_ERROR_LTE_LINK_UP_FAIL
  - 2 - LWM2M_CARRIER_ERROR_LTE_LINK_DOWN_FAIL
  - 3 - LWM2M_CARRIER_ERROR_BOOTSTRAP
  - 4 - LWM2M_CARRIER_ERROR_FOTA_FAIL
  - 5 - LWM2M_CARRIER_ERROR_CONFIGURATION
  - 6 - LWM2M_CARRIER_ERROR_INIT
  - 8 - LWM2M_CARRIER_ERROR_CONNECT

  <value> indicates the error value returned in this event.

3.10.2 LwM2M Carrier library #XCARRIER

The #XCARRIER command allows you to send LwM2M carrier library commands.

Set command

The set command allows you to send LwM2M carrier library commands.

Syntax

```
AT#XCARRIER=<cmd>[,<param1>[,<param2>]..]
```

The parameters and their defined values are the following:

<cmd>
String. The carrier library command to execute. Can be used as follows:

- AT#XCARRIER="app_data_create",<obj_inst_id>,<res_inst_id>

  Create an empty resource instance of the Data resource (ID: 0) of the Binary App Data Container object (ID: 19).

`<obj_inst_id>` indicates the target object instance.

`<res_inst_id>` indicates the target resource instance.

- `AT#XCARRIER="app_data_set"[,<data>][,<obj_inst_id>,<res_inst_id>]`

  Put the value in `<data>` into the indicated path. `<data>` must be a hexadecimal string in double quotes, unless *Data mode* is enabled.

  - If `<obj_inst_id>` and `<res_inst_id>` are specified, the data is set in an instance of the Data resource (ID: 0) of the Binary App Data Container object (ID: 19). The URI path of the resource instance is indicated as `/19/<obj_inst_id>/0/<res_inst_id>`.
  - If `<obj_inst_id>` and `<res_inst_id>` are not present, the data is set in the Uplink Data resource (ID: 0) of the App Data Container object (ID: 10250). The URI path of the resource instance is indicated as `/10250/0/0`.
  - If `<data>` is not present, Serial Modem enters *Data mode* and the data is set in the Uplink Data resource (ID: 0) of the App Data Container object (ID: 10250). The URI path of the resource instance is indicated as `/10250/0/0`.

- `AT#XCARRIER="battery_level",<battery_level>`

  Put the value in `<battery_level>` into the Battery Level resource (ID: 9) of the Device object (ID :3). `<battery_level>` must be an integer value between `0` and `100`.

- `AT#XCARRIER="battery_status",<battery_status>`

  Set the Battery Status resource (ID: 20) of the Device object (ID: 3). `<battery_status>` must be an integer value as defined in the OMA LwM2M specification.

- `AT#XCARRIER="current",<power_source>,<current>`

  Put the value in `<current>` into the Power Source Current resource (ID: 8) instance corresponding to one of the Available Power Sources resource (ID: 6) instances of the Device object (ID: 3). Refer to the `AT#XCARRIER="power_sources"` command for information regarding the supported power sources. `<current>` must be an integer value specified in milliamperes (mA).

- `AT#XCARRIER="dereg"`

  Request to send a Deregister request to the server.

- `AT#XCARRIER="error","add|remove",<error>`

  Update the Error Code resource (ID: 11) of the Device Object (ID: 3) by adding or removing an individual error. `<error>` must be an integer value as defined in the OMA LwM2M specification.

  - `AT#XCARRIER="error","add",<error>` adds a resource instance with the indicated error if one with that error is not present already.
  - `AT#XCARRIER="error","remove",<error>` removes the resource instance with the indicated error if it exists.

- `AT#XCARRIER="link_down"`

  Request to set the modem to flight mode.

- `AT#XCARRIER="link_up"`

  Request to set the modem to full functionality.

- `AT#XCARRIER="log_data",<data>`

  Put the value in `<data>` into the LogData resource (ID: 4014) of the default EventLog object (ID: 20) instance. `<data>` must be a hexadecimal string in double quotes.

- `AT#XCARRIER="memory_free","read|write"[,<memory>]`

  Read or write the Memory Free resource (ID: 10) of the Device object (ID: 3).

  - `AT#XCARRIER="memory_free","read"` returns the current value expressed in kilobytes.
  - `AT#XCARRIER="memory_free","write",<memory>` puts the value in `<memory>` into the resource. `<memory>` must be an integer value specified in kilobytes.

- `AT#XCARRIER="memory_total",<memory>`

  Put the value in `<memory>` into the Memory Total resource (ID: 21) of the Device object (ID: 3). `<memory>` must be an integer value specified in kilobytes.

- `AT#XCARRIER="portfolio","create|read|write",<obj_inst_id>[,<res_inst_id>[,<identity>]]`

  Create an instance of the Portfolio object (ID: 16), or read or write into the Identity (ID: 0) resource of the Portfolio object (ID: 16).

  - `AT#XCARRIER="portfolio","create",<obj_inst_id>` creates an instance of the object, where the URI path is specified as `/16/<obj_inst_id>`.
  - `AT#XCARRIER="portfolio","read",<obj_inst_id>,<res_inst_id>` returns the current value of the indicated resource instance, where the URI path is specified as `/16/<obj_inst_id>/0/<res_inst_id>`.
  - `AT#XCARRIER="portfolio","write",<obj_inst_id>,<res_inst_id>,<identity>` puts the value in `<identity>` into the indicated resource instance, where the URI path is specified as `/16/<obj_inst_id>/0/<res_inst_id>`. `<identity>` must be a string in double quotes.

- `AT#XCARRIER="power_sources"[,<source1>[,<source2>[,...[,<source8>]]]]`

  Set one or more sources specified in `<source>` parameters into the Available Power Sources resource (ID: 6) of the Device object (ID: 3). Each `<source>` parameter must be an integer value as defined in the OMA LwM2M specification.

- `AT#XCARRIER="position",<latitude>,<longitude>,<altitude>,<timestamp>,<uncertainty>`

Put location telemetry values into the corresponding resources of the Location object (ID: 6).

- `<latitude>` specified in the decimal notation of latitude (WGS1984) is put into the Latitude resource (ID: 0). Must be a double type value in double quotes.
- `<longitude>` specified in the decimal notation of latitude (WGS1984) is put into the Longitude resource (ID: 1). Must be a double type value in double quotes.
- `<altitude>` specified in meters is put into the Altitude resource (ID: 2). Must be a float type value in double quotes.
- `<timestamp>` is put into the Timestamp resource (ID: 5). Must be an integer value specified in UNIX time.
- `<uncertainty>` specified in meters is put into the Radius resource (ID: 3). Must be a float type value in double quotes.

- `AT#XCARRIER="reboot"`

  Request to reboot the device. This allows the library to perform any necessary cleanup before the application resets the device.

- `AT#XCARRIER="regup"`

  Request to send a Register request (or Registration Update, as dictated by the lifetime) to the server.

- `AT#XCARRIER="send",<obj_id>,<obj_inst_id>,<res_id>[,<res_inst_id>]`

  Perform a Send operation to send the currently stored data in the indicated resource or resource instance to the server. This operation is currently only supported for readable opaque resources. The URI path of the resource or resource instance is indicated as `/<obj_id>/<obj_inst_id>/<res_id>/<res_inst_id>`.

- `AT#XCARRIER="time"`

  Read the time reported by the device, including the UTC time, the UTC offset and the timezone. See examples for response syntax.

- `AT#XCARRIER="timezone","read|write"[,<timezone>]`

  Read or write the value reported in the Timezone (ID: 15) resource of the Device object (ID: 3) in IANA Timezone (TZ) database format.

  - `AT#XCARRIER="timezone","read"` returns the timezone currently stored by the device.
  - `AT#XCARRIER="timezone","write",<timezone>` puts the value in `<timezone>` into the resource. `<timezone>` must be a string in double quotes.

- `AT#XCARRIER="utc_offset","read|write"[,<utc_offset>]`

  Read or write the UTC Offset resource (ID: 14) of the Device object (ID: 3).

  - `AT#XCARRIER="utc_offset","read"` returns the UTC offset currently in effect for the device as per ISO 8601.
  - `AT#XCARRIER="utc_offset","write",<utc_offset>` puts the value in `<utc_offset>` into the resource. `<utc_offset>` must be an integer value specified in minutes.

- `AT#XCARRIER="utc_time","read|write"[,<utc_time>]`

  Read or write the Current Time resource (ID: 13) of the Device object (ID: 3).

  - `AT#XCARRIER="utc_time","read"` returns the current UNIX time of the device.
  - `AT#XCARRIER="utc_time","write",<utc_time>` puts the value in `<utc_time>` into the resource. `<utc_time>` must be an integer value specified in UNIX time.

- `AT#XCARRIER="velocity",<heading>,<speed_h>,<speed_v>,<uncertainty_h>,<uncertainty_v>`

  Set or update the latest velocity information that will be mapped into the Velocity resource (ID: 4) and Speed resource (ID: 6) of the Location object (ID: 6).

  - `<heading>` is the horizontal direction of movement in degrees clockwise from North. Must be an integer value between `0` and `359`.
  - `<speed_h>` is the horizontal non-negative speed in meters per second. Must be a float type value in double quotes.
  - `<speed_v>` is the vertical speed in meters per second. A positive value indicates upward motion, while a negative value indicates downward motion. Must be a float type value in double quotes.
  - `<uncertainty_h>` is the horizontal uncertainty speed in meters per second. Must be a non-negative float type value in double quotes.
  - `<uncertainty_v>` is the vertical uncertainty speed in meters per second. Must be a non-negative float type value in double quotes.

- `AT#XCARRIER="voltage",<power_source>,<voltage>`

  Put the value in `<voltage>` into the Power Source Voltage resource (ID: 7) instance corresponding to one of the Available Power Sources resource (ID: 6) instances of the Device object (ID: 3). Refer to the `AT#XCARRIER="power_sources"` command for information regarding the supported power sources. `<voltage>` must be an integer value specified in millivolts.

Response syntax The response syntax depends on the commands used.

Examples

```
AT#XCARRIER="time","read"
#XCARRIER: UTC_TIME: 2022-12-30T14:56:46Z, UTC_OFFSET: 60, TIMEZONE:
Europe/Paris
OK
```

```
AT#XCARRIER="error","add",5
OK

AT#XCARRIER="error","remove",5
OK
```

```
AT#XCARRIER="power_sources",1,2,6
OK
```

```
AT#XCARRIER="portfolio","read",2,3
#XCARRIER: LwM2M carrier 3.1.0
OK
```

```
AT#XCARRIER="reboot"
OK
```

```
AT#XCARRIER="position","63.43","10.47","48",1708684683,"30.5"
OK
```

```
AT#XCARRIER="send",19,0,0,0
OK
```

Read command

The read command is not supported.

Test command

The test command is not supported.

3.10.3 LwM2M Carrier library configuration #XCARRIERCFG

The `#XCARRIERCFG` command allows you to configure the LwM2M carrier library. The settings are stored and applied persistently.

Note:

To use `#XCARRIERCFG`, the `CONFIG_LWM2M_CARRIER_SETTINGS` Kconfig option must be enabled. For more details on LwM2M carrier library configuration, see the Configuration section of the library's documentation.

Set command

The set command allows you to configure the LwM2M carrier library.

Syntax

```
AT#XCARRIERCFG=<cmd>[,<param1>[,<param2>]..]
```

The parameters and their defined values are the following:

<cmd>
String. The carrier library configuration command to execute. Can be used as follows:

- `AT#XCARRIERCFG="apn"[,<apn>]`

  Configure the LwM2M carrier library to use a custom APN specified in `<apn>` when connecting to the device management network. `<apn>` must be a string in double quotes. For details, see the `CONFIG_LWM2M_CARRIER_CUSTOM_APN` Kconfig option.

- `AT#XCARRIERCFG="auto_register"[,<0|1>]`

  Set a flag to automatically register to the device management server when attaching to the network. When this configuration is disabled, the user must trigger the registration manually through the `AT#XCARRIER="regup"` command after recovering network coverage. This command accepts two possible input parameters: `0` to disable or `1` to enable.

- `AT#XCARRIERCFG="auto_startup"[,<0|1>]`

  Set a flag to automatically apply the enabled settings to the LwM2M carrier library configuration and connect to the device management network. This command accepts two possible input parameters: `0` to disable or `1` to enable. This command is not available when the *CONFIG_SM_CARRIER_AUTO_STARTUP* Kconfig option is enabled.

- `AT#XCARRIERCFG="carriers"[,"all"|<carrier1>[,<carrier2>[,...[,<carrier6>]]]]`

  Configure the networks in which the LwM2M carrier library will apply (see the General options section of the library's documentation). By default, any network is allowed. The input parameters are mapped as follows:

  - `all` - Any network allowed.
  - `0` - `CONFIG_LWM2M_CARRIER_GENERIC`.
  - `1` - `CONFIG_LWM2M_CARRIER_VERIZON`.
  - `2` - `CONFIG_LWM2M_CARRIER_BELL_CA`.
  - `3` - `CONFIG_LWM2M_CARRIER_LG_UPLUS`.
  - `4` - `CONFIG_LWM2M_CARRIER_T_MOBILE`.
  - `5` - `CONFIG_LWM2M_CARRIER_SOFTBANK`.

- `AT#XCARRIERCFG="coap_con_interval"[,<interval>]`

Configure how often the LwM2M carrier library is to send the Notify operation as a CoAP Confirmable message instead of a Non-Confirmable message. `<interval>` must be an integer value specified in seconds. Two special values may also be used: `-1` to always use Confirmable notifications, or `0` to use the default interval of 86400 seconds. For details, see the `CONFIG_LWM2M_CARRIER_COAP_CON_INTERVAL` Kconfig option.

- `AT#XCARRIERCFG="download_timeout"[,<timeout>]`

  Configure the time allowed for a single firmware image download before it is aborted. This configuration is only supported for Push delivery method of firmware images. `<timeout>` must be an integer value specified in minutes. For details, see the `CONFIG_LWM2M_CARRIER_FIRMWARE_DOWNLOAD_TIMEOUT` Kconfig option.

- `AT#XCARRIERCFG="config_enable"[,<0|1>]`

  Set flag to apply the stored settings to the general Kconfig options (see the General options section of the library's documentation).

- `AT#XCARRIERCFG="device_enable"[,<0|1>]`

  Set flag to apply the stored settings to the device Kconfig options (see the Device options section of the library's documentation).

- `AT#XCARRIERCFG="device_type"[,<device_type>]`

  Configure the value in `<device_type>` to be put into the Device Type resource (ID: 17) of the Device object (ID: 3). `<device_type>` must be a string in double quotes. For details, see the `CONFIG_LWM2M_CARRIER_DEVICE_TYPE` Kconfig option.

- `AT#XCARRIERCFG="hardware_version"[,<version>]`

  Configure the value in `<version>` to be put into the Hardware Version resource (ID: 18) of the Device object (ID: 3). `<version>` must be a string in double quotes. For details, see the `CONFIG_LWM2M_CARRIER_DEVICE_HARDWARE_VERSION` Kconfig option.

- `AT#XCARRIERCFG="manufacturer"[,<manufacturer>]`

  Configure the value in `<manufacturer>` to be put into the Manufacturer resource (ID: 0) of the Device object (ID: 3). `<manufacturer>` must be a string in double quotes. For details, see the `CONFIG_LWM2M_CARRIER_DEVICE_MANUFACTURER` Kconfig option.

- `AT#XCARRIERCFG="model_number"[,<model_number>]`

  Configure the value in `<model_number>` to be put into the Model Number resource (ID: 1) of the Device object (ID: 3). `<model_number>` must be a string in double quotes. For details, see the `CONFIG_LWM2M_CARRIER_DEVICE_MODEL_NUMBER` Kconfig option.

- `AT#XCARRIERCFG="software_version"[,<version>]`

  Configure the value in `<version>` to be put into the Software Version resource (ID: 19) of the Device object (ID: 3). `<version>` must be a string in double quotes. For details, see the `CONFIG_LWM2M_CARRIER_DEVICE_SOFTWARE_VERSION` Kconfig option.

- `AT#XCARRIERCFG="device_serial_no_type"[,<device_serial_no_type>]`

  Configure the Device Serial Number type to be used in LG U+ network. The `<device_serial_no_type>` must be an integer value. It accepts the following values:

  - `0` - `CONFIG_LWM2M_CARRIER_LG_UPLUS_IMEI`.
  - `1` - `CONFIG_LWM2M_CARRIER_LG_UPLUS_2DID`.

  For details, see the `CONFIG_LWM2M_CARRIER_LG_UPLUS_DEVICE_SERIAL_NUMBER` Kconfig option.

- `AT#XCARRIERCFG="service_code"[,<service_code>]`

  Configure the Service Code registered for this device with LG U+. `<service_code>` must be a string in double quotes. For details, see the `CONFIG_LWM2M_CARRIER_LG_UPLUS_SERVICE_CODE` Kconfig option.

- `AT#XCARRIERCFG="pdn_type"[,<pdn_type>]`

  Configure the PDN type of the custom APN configured through the `AT#XCARRIERCFG="apn"` command. `<pdn_type>` must be an integer value. It accepts the following values:

  - `0` - `LWM2M_CARRIER_PDN_TYPE_IPV4V6`.
  - `1` - `LWM2M_CARRIER_PDN_TYPE_IPV4`.
  - `2` - `LWM2M_CARRIER_PDN_TYPE_IPV6`.
  - `3` - `LWM2M_CARRIER_PDN_TYPE_NONIP`.

  For details, see the `CONFIG_LWM2M_CARRIER_PDN_TYPE` Kconfig option.

- `AT#XCARRIERCFG="queue_mode"[,<0|1>]`

  Configure whether the LwM2M carrier library is to inform the server that it may be disconnected for an extended period of time. This configuration corresponds to the Queue Mode Operation as defined in the OMA LwM2M specification. This command accepts two possible input parameters: `0` to disable or `1` to enable. For details, see the `CONFIG_LWM2M_CARRIER_QUEUE_MODE` Kconfig option.

- `AT#XCARRIERCFG="binding"[,<binding>]`

  Configure the binding over which the LwM2M carrier library is to connect to the device management network. `<binding>` must be a string in double quotes. It accepts any combination of the following values:

  - `"U"` - `CONFIG_LWM2M_CARRIER_SERVER_BINDING_UDP`.
  - `"N"` - `CONFIG_LWM2M_CARRIER_SERVER_BINDING_NONIP`.

  Additionally, an empty `<binding>` resets the configuration to default setting (factory configuration).

- `AT#XCARRIERCFG="is_bootstrap"[,<0|1>]`

  Indicate whether the custom server configured through the `AT#XCARRIERCFG="uri"` command is a bootstrap server. This command accepts two possible input parameters: `0` if it is not a boot-

strap server or `1` if it is a bootstrap server. For details, see the `CONFIG_LWM2M_CARRIER_IS_BOOTSTRAP_SERVER` Kconfig option.

- `AT#XCARRIERCFG="lifetime"[,<lifetime>]`

  Configure the lifetime of the custom server configured through the `AT#XCARRIERCFG="uri"` command. `<lifetime>` must an integer value specified in seconds. This configuration is ignored if the custom server is a bootstrap server. For details, see the `CONFIG_LWM2M_CARRIER_SERVER_LIFETIME` Kconfig option.

- `AT#XCARRIERCFG="sec_tag"[,<sec_tag>]`

  Configure the security tag that stores the credentials to be used to set up the DTLS session with the custom server configured through the `AT#XCARRIERCFG="uri"` command. `<sec_tag>` must be an integer value specifying the security tag number to be used. This configuration is ignored for non-secure connections. For details, see the `CONFIG_LWM2M_CARRIER_SERVER_SEC_TAG` Kconfig option.

- `AT#XCARRIERCFG="uri"[,<uri>]`

  Configure the URI of a custom server that the library is to connect to. `<uri>` must be a string in double quotes. For details, see the `CONFIG_LWM2M_CARRIER_CUSTOM_URI` Kconfig option.

- `AT#XCARRIERCFG="session_idle_timeout"[,<session_idle_timeout>]`

  Configure how long a DTLS session used by the library can be idle before it is closed. `<session_idle_timeout>` must be an integer value specified in seconds. Two special values may also be used: `-1` to disable the session idle timeout, or `0` to use the default interval of 60 seconds. For details, see the `CONFIG_LWM2M_CARRIER_SESSION_IDLE_TIMEOUT` Kconfig option.

If a valid command string is used without any parameter, the current value of the corresponding configuration will be returned in the response, as in the examples shown below.

Response syntax The response syntax depends on the commands used.

Examples

```
AT#XCARRIERCFG="auto_startup"
#XCARRIERCFG: 0
OK

AT#XCARRIERCFG="auto_startup",1
OK

AT#XCARRIERCFG="auto_startup"
#XCARRIERCFG: 1
OK
```

```
AT#XCARRIERCFG="manufacturer","Nordic Semiconductor ASA"
OK

AT#XCARRIERCFG="manufacturer"
#XCARRIERCFG: Nordic Semiconductor ASA
OK
```

```
AT#XCARRIERCFG="apn","custom.APN"
OK

AT#XCARRIERCFG="apn"
#XCARRIERCFG: custom.APN
OK
```

```
AT#XCARRIERCFG="carriers","all"
OK

AT#XCARRIERCFG="carriers"
#XCARRIERCFG: all

AT#XCARRIERCFG="carriers",1,3
OK

AT#XCARRIERCFG="carriers"
#XCARRIERCFG: 1, 3
OK
```

Read command

The read command is not supported.

Test command

The test command is not supported.

3.11 MQTT client AT commands

This page describes the AT commands used to operate the MQTT client.

Note:

The MQTT client holds no internal state information for publish and subscript. It will not retransmit packets or guarantee packet delivery according to QoS classes. This deviates from MQTT v3.1.1.

3.11.1 MQTT event notification #XMQTTEVT

MQTT client AT commands report protocol events as unsolicited notifications:

Unsolicited notification

```
#XMQTTEVT: <evt_type>,<result>
```

The parameters and their defined values are the following:

<evt_type>
Integer. The MQTT event type:

- `0` - Connection acknowledgment (CONNACK).
- `1` - Disconnection (DISCONNECT). The MQTT client is disconnected from the MQTT broker when the Serial Modem sends this event. If the host receives this event without a preceding `AT#XMQTTCON=0` command, the host must issue `AT#XMQTTCON=0` before connecting again.
- 2 - Incoming publish (PUBLISH). A subscribed topic received a message. The Serial Modem delivers the payload in a separate `#XMQTTMSG` notification. See *Incoming message notification #XMQTTMSG*.
- 3 - Publish acknowledgment (PUBACK), QoS 1.
- 4 - Publish received (PUBREC), QoS 2.
- 5 - Publish release (PUBREL), QoS 2.
- 6 - Publish complete (PUBCOMP), QoS 2.
- 7 - Subscribe acknowledgment (SUBACK).
- 8 - Unsubscribe acknowledgment (UNSUBACK).
- 9 - Ping response (PINGRESP). The Serial Modem can send this event at any time while connected, when the keep alive interval expires and the client pings the broker.

<result>

- `0` - Success.
- *Negative value* - Error code indicating the reason for the failure.

The commands describe which `<evt_type>` values each command can trigger. All notifications use the syntax documented in this section.

3.11.2 MQTT configure #XMQTTCFG

The `#XMQTTCFG` command allows you to configure the MQTT client before connecting to a broker.

Set command

The set command allows you to configure the MQTT client.

Syntax

```
AT#XMQTTCFG=<client_id>[,<keep_alive>[,<clean_session>]]
```

The parameters and their defined values are the following:

<client_id>
String. The MQTT Client ID. If this command is not issued, Serial Modem uses the default value of `sm_default_client_id`.

<keep_alive>
Integer. The maximum Keep Alive time in seconds for MQTT. The default Keep Alive time is 60 seconds.

<clean_session>

- `0` - Connect to a MQTT broker using a persistent session.
- `1` - Connect to a MQTT broker using a clean session.

The default is using a persistent session.

Examples

```
AT#XMQTTCFG="MyMQTT-Client-ID",300,1
OK
```

Read command

The read command shows MQTT client configuration information.

Syntax

```
AT#XMQTTCFG?
```

Response syntax

```
#XMQTTCFG: <client_id>,<keep_alive>,<clean_session>
```

The parameters and their defined values are the following:

<client_id>
String. The MQTT Client ID.

<keep_alive>
Integer. The maximum Keep Alive time in seconds for MQTT.

<clean_session>

- `0` - Connect to a MQTT broker using a persistent session.
- `1` - Connect to a MQTT broker using a clean session.

Examples

```
AT#XMQTTCFG?
#XMQTTCFG: "MyMQTT-Client-ID",60,0
OK
```

Test command

The test command tests the existence of the command and provides information about the type of its subparameters.

Syntax

```
AT#XMQTTCFG=?
```

Response syntax

```
#XMQTTCFG: <client_id>,<keep_alive>,<clean_session>
```

Examples

```
AT#XMQTTCFG=?
#XMQTTCFG: <client_id>,,<keep_alive>,<clean_session>
OK
```

3.11.3 MQTT connect #XMQTTCON

The `#XMQTTCON` command allows you to connect to and disconnect from the MQTT broker.

This command can send `#XMQTTEVT` with the following `<evt_type>` values:

- `0` - After a successful connect request (`<op>` `1` or `2`).
- `1` - After a disconnect request (`<op>` `0`), when the broker closes the connection, or when the client disconnects for other reasons.
- `9` - While connected, when the broker responds to a keep alive ping.

See *MQTT event notification #XMQTTEVT* for the notification syntax and parameter descriptions.

Set command

The set command allows you to connect to and disconnect from the MQTT broker.

Note:

The `#XMQTTCON` command uses default PDN connection with ID `0`. Raw sockets must not use the PDN connection at the same time. See *Socket AT commands* for more information.

Syntax

```
AT#XMQTTCON=<op>[,<username>,<password>,<url>,<port>[,<sec_tag>]]
```

The parameters and their defined values are the following:

<op>

- `0` - Disconnect from the MQTT broker.
- `1` - Connect to the MQTT broker using IP protocol family version 4.
- `2` - Connect to the MQTT broker using IP protocol family version 6.

<username>
String. The MQTT client username.

<password>
String. The MQTT client password in cleartext.

<url>
String. The MQTT broker hostname.

<port>
Integer. An unsigned 16-bit integer (0 - 65535) indicating the MQTT broker port.

<sec_tag>
Integer. The credential of the security tag used for establishing a secure connection.

Examples

```
AT#XMQTTCFG="MyMQTT-Client-ID",300,1
OK

AT#XMQTTCON=1,"","","mqtt.server.com",1883
OK
#XMQTTEVT: 0,0

Keep alive expires and broker responds to our ping:
#XMQTTEVT: 9,0
```

```
AT#XMQTTCON=0
OK
#XMQTTEVT: 1,0
```

When the MQTT broker closes the connection:

```
#XMQTTEVT: 1,0

AT#XMQTTCON=0
OK
```

Read command

The read command shows MQTT client information.

Syntax

```
AT#XMQTTCON?
```

Response syntax

```
#XMQTTCON: <status>[,<client_id>,<url>,<port>[,<sec_tag>]]
```

The parameters and their defined values are the following:

<status>

- `0` - MQTT is not connected.
- `1` - MQTT is connected.

<client_id>
String. The MQTT client ID.

<url>
String. The MQTT broker hostname. Present only when `<status>` is `1`.

<port>
Integer. An unsigned 16-bit integer (0 - 65535) indicating the MQTT broker port. Present only when `<status>` is `1`.

<sec_tag>
Integer. The credential of the security tag used for establishing a secure connection. Present only when `<status>` is `1`.

Examples

```
AT#XMQTTCON?
#XMQTTCON: 1,"","","mqtt.server.com",1883
OK
```

Test command

The test command tests the existence of the command and provides information about the type of its subparameters.

Syntax

```
#XMQTTCON=?
```

Response syntax

```
#XMQTTCON: (list of op),<username>,<password>,<url>,<port>,<sec_tag>
```

Examples

```
AT#XMQTTCON=?
#XMQTTCON: (0,1,2),<username>,<password>,<url>,<port>,<sec_tag>
OK
```

3.11.4 Incoming message notification #XMQTTMSG

When the MQTT client has successfully subscribed to a topic using *MQTT subscribe #XMQTTSUB* and a message is published on that topic, the Serial Modem sends #XMQTTMSG unsolicited notification.

Unsolicited notification

```
#XMQTTMSG: <topic_length>,<message_length>
<topic_received>
<message>
```

The parameters and their defined values are the following:

<topic_length>
Integer. The length of the <topic_received> field.

<message_length>
Integer. The length of the <message> field.

<topic_received>
String. The topic that received the message.

<message>
Raw payload bytes. The message received from the topic, forwarded without conversion or escaping. Read exactly <message_length> bytes, as the payload can contain NUL bytes and line breaks.

3.11.5 MQTT subscribe #XMQTTSUB

The #XMQTTSUB command allows you to subscribe to an MQTT topic.

On successful subscription, the Serial Modem sends #XMQTTEVT with <evt_type> 7 (SUBACK).

When a message is published on a subscribed topic, the Serial Modem sends `#XMQTTMSG` and `#XMQTTEVT` with `<evt_type>` 2 (PUBLISH). For incoming QoS 2 messages, the Serial Modem also sends `#XMQTTEVT` with `<evt_type>` 5 (PUBREL).

See *MQTT event notification #XMQTTEVT* and *Incoming message notification #XMQTTMSG* for the notification syntax and parameter descriptions.

Set command

The set command allows you to subscribe to an MQTT topic.

Syntax

```
AT#XMQTTSUB=<topic>,<qos>
```

The parameters and their defined values are the following:

<topic>
String. The topic to subscribe to.

<qos>
The MQTT Quality of Service type.

- `0` - Lowest Quality of Service. No acknowledgment of the reception is needed for the published message.
- `1` - Medium Quality of Service. If the acknowledgment of the reception is expected for the published message, publishing duplicate messages is permitted.
- `2` - Highest Quality of Service. The acknowledgment of the reception is expected and the message should be published only once.

Examples

```
AT#XMQTTSUB="nrf91/sm/mqtt/topic0",0
OK
#XMQTTEVT: 7,0

Message with QoS0 is received:
#XMQTTMSG: 21,7
nrf91/sm/mqtt/topic0
message
#XMQTTEVT: 2,0
```

```
AT#XMQTTSUB="nrf91/sm/mqtt/topic1",1
OK
#XMQTTEVT: 7,0

Message with QoS1 is received:
#XMQTTMSG: 21,7
nrf91/sm/mqtt/topic1
```

(continues on next page)

(continued from previous page)

```
message

#XMQTTEVT: 2,0
```

```
AT#XMQTTSUB="nrf91/sm/mqtt/topic2",2
OK
#XMQTTEVT: 7,0

Message with QoS2 is received:
#XMQTTMSG: 21,7
nrf91/sm/mqtt/topic2
message

#XMQTTEVT: 2,0

#XMQTTEVT: 5,0
```

Read command

The read command is not supported.

Test command

The test command is not supported.

3.11.6 MQTT unsubscribe #XMQTTUNSUB

The `#XMQTTUNSUB` command allows you to unsubscribe from an MQTT topic.

On successful unsubscription, the Serial Modem sends `#XMQTTEVT` with `<evt_type>` 8 (UNSUBACK).

See *MQTT event notification #XMQTTEVT* for the notification syntax and parameter descriptions.

Set command

The set command allows you to unsubscribe from an MQTT topic.

Syntax

```
AT#XMQTTUNSUB=<topic>
```

The parameters and their defined values are the following:

<topic>
String. The topic to unsubscribe from.

Examples

```
AT#XMQTTUNSUB="nrf91/sm/mqtt/topic0"
OK
#XMQTTEVT: 8,0
```

Read command

The read command is not supported.

Test command

The test command is not supported.

3.11.7 MQTT publish #XMQTTPUB

The #XMQTTPUB command allows you to publish messages on MQTT topics.

Depending on the QoS level, this command can send #XMQTTEVT with the following <evt_type> values:

- 3 - QoS 1 (PUBACK).
- 4 and 6 - QoS 2 (PUBREC, then PUBCOMP).

QoS 0 does not produce an #XMQTTEVT notification.

See *MQTT event notification #XMQTTEVT* for the notification syntax and parameter descriptions.

Set command

The set command allows you to publish messages on MQTT topics.

Syntax

```
AT#XMQTTPUB=<topic>[,<msg>[,<qos>[,<retain>[,<data_len>]]]]
```

The parameters and their defined values are the following:

<topic>
String. The topic on which data is published.

<msg>
String. The payload on the topic being published. If the payload is empty (for example, ""), Serial Modem enters *Data mode*.

<qos>

- 0 - Lowest Quality of Service (default value). No acknowledgment of the reception is needed for the published message.
- 1 - Medium Quality of Service. If the acknowledgment of the reception is expected for the published message, publishing duplicate messages is permitted.

- 2 - Highest Quality of Service. The acknowledgment of the reception is expected and the message should be published only once.

<retain>
Integer. Its default value is `0`. When `1`, it indicates that the broker should store the message persistently.

<data_len>
Integer. Optional, only used when <msg> is empty (data mode). Sets the number of bytes of payload to publish in data mode. When the required number of bytes are received, the payload is published and the data mode is exited.

The termination command *CONFIG_SM_DATAMODE_TERMINATOR* is not used in this case. The value must not exceed the value configured in the *CONFIG_SM_DATAMODE_BUF_SIZE* Kconfig option, as the payload must fit within the data mode buffer to be published as a single message.

The value `0` is equivalent to omitting the parameter. Specifying a non-zero <data_len> together with a non-empty <msg> results in an error.

Examples

```
AT#XMQTTPUB="nrf91/sm/mqtt/topic0","Test message with QoS 0",0,0
OK
```

```
AT#XMQTTPUB="nrf91/sm/mqtt/topic0"
OK
{"msg":"Test Json publish"}+++
#XDATAMODE: 0
```

```
AT#XMQTTPUB="nrf91/sm/mqtt/topic0","",0,0,23
OK
Test message, 23 bytes.
#XDATAMODE: 0
```

```
AT#XMQTTPUB="nrf91/sm/mqtt/topic1","Test message with QoS 1",1,0
OK
#XMQTTEVT: 3,0
```

```
AT#XMQTTPUB="nrf91/sm/mqtt/topic2","",2,0
OK
Test message with QoS 2+++
#XDATAMODE: 0
#XMQTTEVT: 4,0
#XMQTTEVT: 6,0
```

Read command

The read command is not supported.

Test command

The test command is not supported.

3.12 nRF Cloud AT commands

The page describes the following nRF Cloud-related AT commands:

- nRF Cloud connection (see *nRF Cloud access #XNRFCLOUD*)
- nRF Cloud location (see *nRF Cloud location #XNRFCLOUDPOS*)
- nRF Cloud FOTA (see *nRF Cloud FOTA #XNRFCLOUDFOTA*)
- nRF Cloud observability (see *Observability #XNRFCLOUDOBS** and commands starting with `#XNRFCLOUDOBS`)

Note:

Before using nRF Cloud AT commands, you must onboard and claim the device in nRF Cloud. See *nRF Provisioning AT commands* for information about device provisioning and claiming.

Observability #XNRFCLOUDOBS*

The `#XNRFCLOUDOBS*` commands control the Memfault data that the device collects (metrics, events, logs, and, in builds that include it, a coredump) and its upload to nRF Cloud over the CoAP transport. See *nRF Cloud observability* for how the data is collected and how it ends up in nRF Cloud.

To use the `#XNRFCLOUDOBS*` commands that access the network (`#XNRFCLOUDOBSUPLOAD` and `#XNRFCLOUDOBSFORWARD`), the device must be connected to nRF Cloud. See *nRF Cloud access #XNRFCLOUD*.

`#XNRFCLOUDOBSDEVINFO`, `#XNRFCLOUDOBSCRASH`, and `#XNRFCLOUDOBSEXPORT` additionally require the *CONFIG_SM_NRF_CLOUD_OBSERVABILITY_DEBUG* Kconfig option, which is disabled by default.

Upload is host-driven. Automatic uploads are disabled by default. The host enables them with `AT#XNRFCLOUDOBSAUTO=1` or initiates uploads on demand with `AT#XNRFCLOUDOBSUPLOAD`.

The `<project_key>` parameter, used by several commands, is a string. It is a 32-character Memfault project key. When it is present and not empty, it overrides the server-side project-key routing, sending the data to the specified Memfault project. For more information about Memfault project keys, see Memfault Project Keys. Find your project key in Memfault Project Settings.

3.12.1 nRF Cloud access #XNRFCLOUD

The #XNRFCLOUD command controls the access to the nRF Cloud service.

Note:

To use #XNRFCLOUD, the following preconditions apply:

- You must first onboard the device to nRF Cloud, using the device-specific UUID as the device ID. See nRF Cloud Preconnect onboarding for more information.
- The *CONFIG_SM_NRF_CLOUD* Kconfig option must be enabled.
- The device must have access to nRF Cloud through the LTE network.

Set command

The set command allows you to access the nRF Cloud service.

Note:

The #XNRFCLOUD command uses default PDN connection with ID 0. Raw sockets must not use the PDN connection at the same time. See *Socket AT commands* for more information.

Syntax

```
AT#XNRFCLOUD=<op>[,<send_location>]
```

The parameters and their defined values are the following:

<op>

- 0 - Disconnect from the nRF Cloud service.
- 1 - Connect to the nRF Cloud service.
- 2 - Send a message in the JSON format to the nRF Cloud service.

When <op> is 2, Serial Modem enters *Data mode*.

<send_location>

Used only when the value of <op> is 1.

- 0 - The device location is not sent to nRF Cloud. This is the default behavior if the parameter is omitted.
- 1 - The device location is sent to nRF Cloud.

Note:

The location is sent to the nRF Cloud whenever a fix is produced by the GNSS module. You must use the *#XGNSS* AT command to start GNSS either in single-

fix or periodic navigation mode. The interval between fixes must be at least 5 seconds.

Unsolicited notification

```
#XNRFCLOUD: <ready>,<send_location>
```

The parameters and their defined values are the following:

<ready>
Integer. Indicates whether the connection to nRF Cloud is established or not.

<send_location>
Integer. Indicates whether the device location will be sent to nRF Cloud or not.

Example

```
// Connect to nRF Cloud without sending location.
AT#XNRFCLOUD=1

OK

#XNRFCLOUD: 1,0
// Send a message to nRF Cloud.
AT#XNRFCLOUD=2

OK
{"msg":"Hello, nRF Cloud"}+++

#XDATAMODE: 0
// Disconnect from nRF Cloud.
AT#XNRFCLOUD=0

OK

#XNRFCLOUD: 0,0
// Connect to nRF Cloud and send location.
AT#XNRFCLOUD=1,1

OK

#XNRFCLOUD: 1,1
AT#XNRFCLOUD=0

OK

#XNRFCLOUD: 0,1
```

Read command

The read command checks whether the connection to nRF Cloud is established or not.

Syntax

```
AT#XNRFCLOUD?
```

Response syntax

```
#XNRFCLOUD: <ready>,<send_location>,<sec_tag>,<device_id>
```

The parameters and their defined values are the following:

<ready>
Integer. Indicates whether the connection to nRF Cloud is established or not.

<send_location>
Integer. Indicates whether the device location will be sent to nRF Cloud or not.

<sec_tag>
Integer. The `sec_tag` used for accessing nRF Cloud.

<device_id>
String. The device ID used for accessing nRF Cloud.

Example

```
AT#XNRFCLOUD?

#XNRFCLOUD: 1,0,16842753,"50503041-3633-4261-803d-1e2b8f70111a"

OK
```

Test command

The test command tests the existence of the command and provides information about the type of its subparameters.

Syntax

```
AT#XNRFCLOUD=?
```

Example

```
AT#XNRFCLOUD=?

#XNRFCLOUD: (0,1,2),<send_location>
```

(continues on next page)

(continued from previous page)

```
OK
```

3.12.2 nRF Cloud location #XNRFCLOUDPOS

The #XNRFCLOUDPOS command sends a request to nRF Cloud to determine the device's location. The request uses information from the cellular network, Wi-Fi® access points, or both.

Note:

To use #XNRFCLOUDPOS, the device must be connected to nRF Cloud using *#XNRFCLOUD*.

Also, the *CONFIG_SM_NRF_CLOUD_LOCATION* Kconfig option must be enabled.

Set command

The set command allows sending a location request to nRF Cloud.

Syntax

```
AT#XNRFCLOUDPOS=<cell_count>,<wifi_pos>[,<MAC 1>[,<RSSI 1>],<MAC 2>
[,<RSSI 2>][,<MAC 3>[...]]]
```

The parameters and their defined values are the following:

<cell_count>
Integer. The number of cells to include in the location request. The value range is 0 to 15. For cellular positioning, a recommended value is 4. 0 means that no cellular network information will be included in the location request. The Serial Modem uses the AT%NCELLMEAS command to retrieve the cellular network information, and depending on the value of <cell_count>, the command might be executed multiple times.

Note:

Since the Serial Modem uses the AT%NCELLMEAS command internally, the host must not use the AT%NCELLMEAS command during #XNRFCLOUDPOS command execution. You may still use AT%NCELLMEAS command before or after #XNRFCLOUDPOS command execution for your own purposes. You will also see %NCELLMEAS notifications, which you can ignore, during the #XNRFCLOUDPOS command execution.

<wifi_pos>

- 0 - Do not include Wi-Fi access point information in the location request.

- `1` - Use Wi-Fi access point information. The access points must be given as additional parameters to the command. The minimum number of access points to provide is two (`NRF_CLOUD_LOCATION_WIFI_AP_CNT_MIN`), and the maximum is limited by the maximum size of the AT command, which is 8190 bytes, including the terminator character.

<MAC x>
String. The MAC address of a Wi-Fi access point. Must be formatted as `%02x:%02x:%02x:%02x:%02x:%02x` (`WIFI_MAC_ADDR_TEMPLATE`).

<RSSI x>
Integer. Optional. The signal strength of a Wi-Fi access point in dBm, between `-128` and `0`. If provided, it must follow the MAC address parameter of the access point. Providing the RSSI parameters helps improve the accuracy of the Wi-Fi location.

Unsolicited notification

```
#XNRFCLOUDPOS: <status>[,<type>,<latitude>,<longitude>,<uncertainty>
]
```

The notification is sent when the location request completes.

The parameters and their defined values are the following:

<status>

- `0` - Successful request. Other parameters are also present.
- `-1` - Location request failed.
- `<positive integer>` - Requesting location from the cloud failed with cloud error as defined in `nrf_cloud_error` values.

<type>
The service used to fulfill the location request.

- `0` (`LOCATION_TYPE_SINGLE_CELL`) - Single-cell cellular location.
- `1` (`LOCATION_TYPE_MULTI_CELL`) - Multi-cell cellular location.
- `2` (`LOCATION_TYPE_WIFI`) - Wi-Fi location.

<latitude>
Float. The latitude in degrees.

<longitude>
Float. The longitude in degrees.

<uncertainty>
Integer. The radius of the uncertainty circle around the location in meters, also known as Horizontal Positioning Error (HPE).

Example

```
AT%XSYSTEMMODE=1,0,0,0
```

(continues on next page)

(continued from previous page)

```
OK
AT+CFUN=1

OK
AT#XNRFCLOUD=1

OK

#XNRFCLOUD: 1,0
AT#XNRFCLOUDPOS=1,0

OK

#XNRFCLOUDPOS: 0,0,35.455833,139.626111,1094

AT#XNRFCLOUDPOS=5,0

OK

%NCELLMEAS: 0,"0199F10A","44020","107E",65535,3750,5,49,27,107504,
3750,251,33,4,0,475,107,26,14,25,475,58,26,17,25,475,277,24,9,25,
475,51,18,1,25

%NCELLMEAS: 0,"01234567","44020","0340",50,175456,3400,34,5,24,
1775066,1,0,"00143FAE","44020","0140",65535,0,6200,47,40,14,1775066,
0,0

%NCELLMEAS: 0,"00987654","44020","0240",50,1754746,5500,44,4,4,
1463457,1,0,"002F4344","44020","0140",65535,0,6200,47,40,14,1775066,
0,0,"001C0502","44013","5A00",65535,0,6400,130,29,18,1775124,0,0,
"00136107","44013","5A00",65535,0,3600,202,26,13,234533,0,0

#XNRFCLOUDPOS: 0,1,35.455833,139.626111,1094
AT#XNRFCLOUDPOS=0,1,"40:9b:cd:c1:5a:40","00:90:fe:eb:4f:42"

OK

#XNRFCLOUDPOS: 0,2,35.457335,139.624443,60
AT#XNRFCLOUDPOS=0,1,"40:9b:cd:c1:5a:40",-40,"00:90:fe:eb:4f:42",-69

OK

#XNRFCLOUDPOS: 0,2,35.457346,139.624449,20
```

Read command

The read command is not supported.

Test command

The test command is not supported.

3.12.3 nRF Cloud FOTA #XNRFCLOUDFOTA

The #XNRFCLOUDFOTA command checks for and downloads an application or modem firmware update using Memfault release management, delivered over the same nRF Cloud CoAP transport as #XNRFCLOUDOBS*.

Requires the *CONFIG_SM_NRF_CLOUD_FOTA* Kconfig option.

An application update is staged the same way as AT#XFOTA=1, that is, the host activates it with the AT#XRESET AT command. A modem update is staged the same way as AT#XFOTA=2, that is, the host activates it with the AT#XMODEMRESET AT command. Both share their FOTA session with #XFOTA, but progress and completion are reported over the #XNRFCLOUDFOTA notification instead, using the same <fota_stage>, <fota_status> and <fota_info> tuple as #XFOTA (see *FOTA AT commands* for download and activation semantics). Only one FOTA session can be ongoing at a time, regardless of which command initiated it.

Note:

Unlike AT#XFOTA, #XNRFCLOUDFOTA does not support MCUboot bootloader updates. This is expected to be a rare use case. The support is planned for a future release.

Note:

<op>=2 uses a dedicated Memfault project key for modem firmware, obtained from the General section of the modem firmware project's Project Settings, which is separate from the application project.

Set it with the CONFIG_MEMFAULT_FOTA_MODEM_PROJECT_KEY Kconfig option, or override it at runtime with the <project_key> parameter.

When neither is set, the request targets the default project where the device was claimed.

Set command

The set command starts a FOTA check, and the download if requested and an update is available.

Note:

The #XNRFCLOUDFOTA command uses default PDN connection with ID 0. Raw sockets must not use the PDN connection at the same time. See *Socket AT commands* for more information.

Syntax

```
AT#XNRFCLOUDFOTA=<op>[,<project_key>]
```

The parameters and their defined values are the following:

<op>

- 0 - Cancel an ongoing download. This is effective only after a download has started, that is, after the first #XNRFCLOUDFOTA progress notification.
- 1 - Check for and download an application update.
- 2 - Check for and download a modem firmware update. If an interrupted download is detected, it is continued.
- 4 - Check for an application update, without downloading it.
- 6 - Check for a modem firmware update, without downloading it.

<project_key>

String. Project key used for the FOTA operation.

For <op>=1 and <op>=4, it overrides the application project key (CONFIG_MEMFAULT_PROJECT_KEY), and for <op>=2 and <op>=6 it overrides CONFIG_MEMFAULT_FOTA_MODEM_PROJECT_KEY. If this option is not provided and the corresponding Kconfig setting is unset, the request targets the default project where the device was claimed.

The command returns OK immediately and the check runs asynchronously. When it completes, an unsolicited notification is sent.

Unsolicited notification

```
#XNRFCLOUDFOTA: <fota_stage>,<fota_status>[,<fota_info>]
```

The <fota_stage>, <fota_status>, and <fota_info> parameters use the same values as the #XFOTA notification:

<fota_stage>

- 0 - Init.
- 1 - Download.
- 2 - Download, erase pending (modem delta update only).
- 3 - Download, erase complete (modem delta update only).
- 4 - Downloaded, to be activated.
- 5 - Complete.

<fota_status>

- 0 - OK.
- 1 - Error.
- 2 - Cancelled.
- 3 - Reverted (application FOTA only).

<fota_info>

Integer. Its value can have different meanings based on the values returned by <fota_stage> and <fota_status>.

See the following table:

<fota_stage>	<fota_status>	<fota_info>
1 (namely *Download*)	0 (namely *OK*)	Percentage of the download
1 (namely *Download*)	1 (namely *ERROR*)	Error Code
1 (namely *Download*)	2 (namely *CANCELLED*)	0 - Downloading is cancelled before completion
5 (namely *Complete*)	1 (namely *ERROR*)	Error Code

The error codes can be the following:

- 1 - Download failed
- 2 - Update image rejected (for example modem firmware version error)
- 3 - Update image mismatch (for example <op> is 1 but <file_url> points to a modem image)

For modem FOTA, the error codes can be the following:

- 71303169 (NRF_MODEM_DFU_RESULT_INTERNAL_ERROR) - The modem encountered a fatal internal error during firmware update.
- 71303170 (NRF_MODEM_DFU_RESULT_HARDWARE_ERROR) - The modem encountered a fatal hardware error during firmware update.
- 71303171 (NRF_MODEM_DFU_RESULT_AUTH_ERROR) - Modem firmware update failed due to an authentication error.
- 71303172 (NRF_MODEM_DFU_RESULT_UUID_ERROR) - Modem firmware update failed due to UUID mismatch.
- 71303173 (NRF_MODEM_DFU_RESULT_VOLTAGE_LOW) - Modem firmware update not executed due to low voltage. The modem will retry the update on reboot.

When the FOTA check finishes without starting a download, <fota_stage> is 0 (*Init*) and <fota_info> indicates the result. This form is used by #XNRFCLOUDFOTA only (not #XFOTA) and specified as follows:

<fota_stage>	<fota_status>	<fota_info>
0 (namely *Init*)	0 (namely *OK*)	0 - No update is available
0 (namely *Init*)	0 (namely *OK*)	**1 - An update is available (sent only for** <op>=4 and <op>=6
0 (namely *Init*)	1 (namely *ERROR*)	Error code

Examples

```
AT#XNRFCLOUDFOTA=1

OK

#XNRFCLOUDFOTA: 1,0,45

#XNRFCLOUDFOTA: 1,0,100

#XNRFCLOUDFOTA: 4,0
AT#XRESET
```

The following example checks whether an application update is available, without downloading it:

```
AT#XNRFCLOUDFOTA=4

OK

#XNRFCLOUDFOTA: 0,0,1
```

Read command

The read command is not supported.

Test command

The test command returns the supported syntax.

Syntax

```
AT#XNRFCLOUDFOTA=?
```

Response

```
#XNRFCLOUDFOTA: (0,1,2,4,6)[,<project_key>]
```

3.12.4 On-demand upload #XNRFCLOUDOBSUPLOAD

The `#XNRFCLOUDOBSUPLOAD` command uploads the buffered observability data to nRF Cloud.

The captured logs are collected before the upload so that they are included.

Set command

The set command uploads the buffered data.

Syntax

```
AT#XNRFCLOUDOBSUPLOAD[=<project_key>]
```

The command returns `OK` immediately and the upload runs asynchronously. When it completes, an unsolicited notification is sent.

Unsolicited notification

```
#XNRFCLOUDOBSUPLOAD: <result>[,<bytes>]
```

The parameters and their defined values are the following:

<result>

- `0` - Success. The `<bytes>` parameter follows and indicates the number of bytes uploaded, which is `0` when there was nothing buffered.
- `-1` - Failure. The error code is shown in the log.

Example

```
AT#XNRFCLOUDOBSUPLOAD

OK

#XNRFCLOUDOBSUPLOAD: 0,108
```

Read command

The read command is not supported.

Test command

The test command returns the supported syntax.

Syntax

```
AT#XNRFCLOUDOBSUPLOAD=?
```

Response

```
#XNRFCLOUDOBSUPLOAD: <project_key>
```

3.12.5 Metrics heartbeat #XNRFCLOUDOBSHEARTBEAT

The #XNRFCLOUDOBSHEARTBEAT command collects and finalizes a metrics heartbeat.

The data is buffered on the device until it is uploaded, so the command does not require a connection to nRF Cloud.

The heartbeat carries the LTE metrics that the modem reports, such as the modem firmware version, the network operator, RSRP, SNR, the current band, and the transmitted and received data in kilobytes.

When *CONFIG_SM_NRF_CLOUD_OBSERVABILITY_LTE_METRICS* is enabled, the heartbeat also carries the cell ID, the tracking area code, the LTE mode, the PSM active time and periodic TAU, and the eDRX interval and paging time window.

The heartbeat additionally carries connection statistics that are tracked between heartbeats, including the time to connect to the network, the number of connection losses, and the time the LTE link has been active. Values not reported by the modem, such as when it is deactivated, are excluded from the heartbeat.

Set command

The set command collects a heartbeat.

Syntax

```
AT#XNRFCLOUDOBSHEARTBEAT
```

Example

```
AT#XNRFCLOUDOBSHEARTBEAT

OK
```

Read command

The read command is not supported.

Test command

The test command is not supported.

3.12.6 Chunk forwarding #XNRFCLOUDOBSFORWARD

The `#XNRFCLOUDOBSFORWARD` command forwards a Memfault chunk produced by the host to nRF Cloud, using the Serial Modem connection.

Note:

nRF Cloud attributes the chunk to the authenticated device, so the observability data of the host is reported under the device serial of the Serial Modem device by default. To report it under a device serial of the host instead, build the host firmware with `MEMFAULT_EVENT_INCLUDE_DEVICE_SERIAL` set to `1` in its `memfault_platform_config.h`. Every chunk then carries the device serial of the host, which Memfault uses to attribute the data.

Set command

The set command forwards a chunk.

Syntax

```
AT#XNRFCLOUDOBSFORWARD=<base64_chunk>[,<project_key>]
```

The parameters and their defined values are the following:

<base64_chunk>
String. The base64-encoded Memfault chunk to forward.

The command returns `OK` immediately and the chunk is posted asynchronously. When the post completes, an unsolicited notification is sent.

Unsolicited notification

```
#XNRFCLOUDOBSFORWARD: <result>
```

The parameters and their defined values are the following:

<result>

- `0` - Success.
- `-1` - Failure. The chunk is not buffered, so the host must send it again to retry.

Example Forward a chunk of the host, sent to a specific Memfault project:

```
AT#XNRFCLOUDOBSFORWARD="CAKnAgIDAQpqdGVzdHNlcmlhbA==","<project_key>
"

OK

#XNRFCLOUDOBSFORWARD: 0
```

Read command

The read command is not supported.

Test command

The test command returns the supported syntax.

Syntax

```
AT#XNRFCLOUDOBSFORWARD=?
```

Response

```
#XNRFCLOUDOBSFORWARD: <base64_chunk>,<project_key>
```

3.12.7 Core dump upload #XNRFCLOUDOBSCOREDUMP

The `#XNRFCLOUDOBSCOREDUMP` command controls whether a stored core dump is included in an upload.

When disabled, `#XNRFCLOUDOBSUPLOAD` and the automatic upload send the buffered events, logs, and CDRs but leave the core dump in storage, so it can be uploaded later by enabling this again. Core dump upload is enabled by default, and the setting is persisted.

Note:

Core dump storage holds a single core dump. While a core dump is stored, subsequent crashes do not capture a new core dump until the stored one has been uploaded. Keeping the upload disabled prevents new core dumps from being captured.

Set command

The set command enables or disables the core dump upload.

Syntax

```
AT#XNRFCLOUDOBSCOREDUMP=<enable>
```

The parameters and their defined values are the following:

<enable>

- `0` - Exclude the stored core dump from uploads.
- `1` - Include the stored core dump in uploads.

Read command

The read command returns the current setting.

Syntax

```
AT#XNRFCLOUDOBSCOREDUMP?
```

Response

```
#XNRFCLOUDOBSCOREDUMP: <enable>
```

Test command

The test command returns the supported syntax.

Syntax

```
AT#XNRFCLOUDOBSCOREDUMP=?
```

Response

```
#XNRFCLOUDOBSCOREDUMP: (0,1)
```

3.12.8 Automatic upload #XNRFCLOUDOBSAUTO

The `#XNRFCLOUDOBSAUTO` command configures the automatic upload of the buffered observability data.

The configuration is persistent, so it survives a reboot.

Set command

The set command enables or disables the automatic upload and configures its interval.

Syntax

```
AT#XNRFCLOUDOBSAUTO=<enable>[,<interval_seconds>][,<project_key>]
```

The parameters and their defined values are the following:

<enable>

- `0` - Disable the automatic upload.
- `1` - Enable the automatic upload.

<interval_seconds>
Integer. The interval between two uploads, from `60` to `86400` seconds. When omitted, the stored interval is kept.

The initial value is set by the *CONFIG_SM_NRF_CLOUD_OBSERVABILITY_AUTO_INTERVAL_SECONDS* Kconfig option.

<project_key>
String. When omitted, the stored project key is kept, and an empty string clears it.

The first upload runs when the interval expires, not when the automatic upload is enabled. An upload that falls while there is no connection to nRF Cloud is skipped, and the next one is scheduled as usual.

The automatic upload is silent and does not send any unsolicited notifications.

Response The command returns OK, also when the configuration could not be stored, in which case only its persistence is lost.

Read command

The read command returns the configuration of the automatic upload.

Syntax

```
AT#XNRFCLOUDOBSAUTO?
```

Response

```
#XNRFCLOUDOBSAUTO: <enable>,<interval_seconds>,<project_key>
```

Example

```
AT#XNRFCLOUDOBSAUTO=1,600

OK
AT#XNRFCLOUDOBSAUTO?

#XNRFCLOUDOBSAUTO: 1,600,""

OK
```

Test command

The test command returns the supported syntax.

Syntax

```
AT#XNRFCLOUDOBSAUTO=?
```

Response

```
#XNRFCLOUDOBSAUTO: (0,1),(60-86400),<project_key>
```

3.12.9 Memfault device information #XNRFCLOUDOBSDEVINFO

The `#XNRFCLOUDOBSDEVINFO` command returns the Memfault device information, which identifies the device and the firmware in the Memfault project.

This command requires the *CONFIG_SM_NRF_CLOUD_OBSERVABILITY_DEBUG* Kconfig option.

Set command

The set command returns the device information.

Syntax

```
AT#XNRFCLOUDOBSDEVINFO
```

Response

```
#XNRFCLOUDOBSDEVINFO: <device_serial>,<software_type>,<software_
version>,<hardware_version>
```

The parameters and their defined values are the following:

<device_serial>
String. The device serial.

<software_type>
String. The software type.

<software_version>
String. The software version.

<hardware_version>
String. The hardware version.

Example

```
AT#XNRFCLOUDOBSDEVINFO

#XNRFCLOUDOBSDEVINFO: "50344654-3037-409f-802d-2206917f23d2","app",
"2.0.0+0","nrf9151dk"

OK
```

Read command

The read command is not supported.

Test command

The test command is not supported.

3.12.10 Forced crash #XNRFCLOUDOBSCRASH

The #XNRFCLOUDOBSCRASH command forces a crash, so that the coredump capture and upload can be tested.

This command requires the *CONFIG_SM_NRF_CLOUD_OBSERVABILITY_DEBUG* Kconfig option.

Set command

The set command crashes the application.

Syntax

```
AT#XNRFCLOUDOBSCRASH[=<type>]
```

The parameters and their defined values are the following:

<type>
Integer. An integer from `0` to `4`. It defaults to `0`, an assertion failure.

The device crashes and no response is returned, unless the crash type is invalid.

Read command

The read command is not supported.

Test command

The test command returns the supported syntax.

Syntax

```
AT#XNRFCLOUDOBSCRASH=?
```

Response

```
#XNRFCLOUDOBSCRASH: <type>
```

3.12.11 Chunk export #XNRFCLOUDOBSEXPORT

The #XNRFCLOUDOBSEXPORT command prints the buffered Memfault chunks to the AT interface instead of uploading them in the Memfault chunk export format, which the Memfault tooling can parse.

This command requires the *CONFIG_SM_NRF_CLOUD_OBSERVABILITY_DEBUG* Kconfig option.

Note:

The command consumes the chunks. They are no longer available for #XNRFCLOUDOBSUPLOAD and the automatic upload.

Set command

The set command prints the buffered chunks.

Syntax

```
AT#XNRFCLOUDOBSEXPORT
```

Response The buffered chunks are returned, one per line:

```
MC:<base64_chunk>:
```

Example

```
AT#XNRFCLOUDOBSHEARTBEAT

OK
AT#XNRFCLOUDOBSEXPORT

MC:CAKnAgIDAQpqdGVzdHNlcmlhbA==:

OK
```

Read command

The read command is not supported.

Test command

The test command is not supported.

3.13 nRF Provisioning AT commands

This page describes AT commands for the nRF Device Provisioning service. The provisioning client connects to the nRF Cloud Provisioning Service over CoAP or DTLS and applies device configuration commands (credentials, settings, and firmware updates) issued from the cloud.

To use the provisioning service with the Serial Modem, you must first claim the device using its attestation token:

1. Retrieve the attestation token with the `AT%ATTESTTOKEN` command.
2. Enter the token as described in the nRF Cloud Provisioning Service documentation.
3. Run the `AT#XNRFPROV` command to trigger provisioning, as described in *Trigger provisioning #XNRFPROV*. During the session, the host must respond to the `#XNRFPROV:`

1 and #XNRFPROV: 2 notifications. The host takes the modem offline to write credentials, and then brings it back online to reconnect to the provisioning server. See the *unsolicited notifications* section for details.

These commands are only available when the application is built with the `nrf-device-provisioning.conf` overlay, which enables the provisioning client and its AT command interface.

3.13.1 Trigger provisioning #XNRFPROV

The #XNRFPROV command triggers an immediate provisioning attempt.

Set command

The set command starts a provisioning session immediately, independently of the configured schedule interval.

Syntax

```
AT#XNRFPROV
```

Response syntax

```
OK
```

Unsolicited notifications One of the following #XNRFPROV notifications is emitted when the provisioning session completes, an event occurs, or host action is required:

```
#XNRFPROV: <status>
```

The parameters and their defined values are the following:

<status>

- 0 - Provisioning successful, or no pending commands on the server. The two cases are not distinguished.
- 1 - Host action required: deactivate LTE. The host must take the modem offline (for example AT+CFUN=4) so that credentials can be written safely.

  The provisioning client waits until the modem reports offline functional mode before continuing or times out after CONFIG_NRF_PROVISIONING_MODEM_STATE_WAIT_TIMEOUT_SECONDS seconds.
- 2 - Host action required: activate LTE. The host must bring the modem back online (for example AT+CFUN=1) to reconnect to the provisioning server.

  The provisioning client waits until the modem reports LTE registration before continuing or times out after CONFIG_NRF_PROVISIONING_MODEM_STATE_WAIT_TIMEOUT_SECONDS seconds.

- `-1` - Provisioning failed. The host can retry provisioning by triggering the `#XNRFPROV` command again.
- `-2` - Device not claimed on nRF Cloud. The device must be claimed using its attestation token. Refer to the nRF Cloud Provisioning Service documentation for more information.
- `-3` - Wrong root CA certificate. Provision the correct nRF Cloud root CA certificate.
- `-4` - No valid datetime reference. The modem must have a valid time before provisioning can proceed.
- `-5` - Too many commands received from the server. Increase `CONFIG_NRF_PROVISIONING_CBOR_RECORDS`.
- `-6` - Fatal error: the provisioning client encountered an irrecoverable error.

Example The provisioning client requests the host to manage the LTE connection during the session. The host must respond to the `#XNRFPROV: 1` and `#XNRFPROV: 2` notifications accordingly.

```
AT#XNRFPROV
OK

#XNRFPROV: 1

AT+CFUN=4
OK

#XNRFPROV: 2

AT+CFUN=1
OK

#XNRFPROV: 0
```

Test command

The test command returns the command syntax.

Syntax

```
AT#XNRFPROV=?
```

Response syntax

```
#XNRFPROV
```

3.14 PPP AT commands

This page describes AT commands related to the Point-to-Point Protocol (PPP).

PPP is enabled in Serial Modem by compiling it with the appropriate configuration files, depending on your use case (with or without CMUX). See the *Configuration files* section for more information.

3.14.1 Enter data state +CGDATA

The AT+CGDATA command enters data state and starts PPP, as specified in 3GPP TS 27.007 (section 10.1.12).

Set command

The set command starts PPP on the current channel, optionally specifying the layer 2 protocol and PDN connection.

Syntax

```
AT+CGDATA[=<L2P>[,<cid>]]
```

The parameters and their defined values are the following:

<L2P>
: String. The layer 2 protocol to use. The only supported value is "PPP". If omitted, "PPP" is assumed.

<cid>
: Integer. The PDN connection to use. Its default value is 0, which represents the default PDN connection.

Response syntax

```
CONNECT
```

Examples With default parameters:

```
AT+CGDATA

CONNECT
```

Request PPP to use specified PDN connection:

```
AT+CGDATA="PPP",1

CONNECT
```

Test command

The test command lists the supported layer 2 protocols.

Syntax

```
AT+CGDATA=?
```

Response syntax

```
+CGDATA: (<L2P_values>)
```

Example

```
AT+CGDATA=?

+CGDATA: ("PPP")

OK
```

3.14.2 Control PPP #XPPP

Note:

AT#XPPP is a compatibility command for Serial Modem v1.x.x and is not recommended for new designs. It depends on the channel assignment configured by the AT#XCMUX command. Use the standard AT+CGDATA command instead.

Set command

The set command request activation or deactivation of PPP link, and optionally define the PDN connection used for PPP. PPP link is established on the second channel (DLC channel 2) when CMUX is used. If the PPP link is preferred on the first channel (DLC channel 1), you must use the AT#XCMUX=2 command to switch the AT command channel to DLC channel 2 before starting PPP.

If PPP is started without CMUX, the current UART switches to PPP mode.

Note:

When a PPP start has been issued, the PPP connection is automatically activated and deactivated when the PDN connection requested for PPP is established and lost, respectively. This will continue until a PPP stop is issued by either the user by the AT#XPPP=0 command or by the remote peer disconnecting the PPP using LCP termination.

Note:

When PPP is started without CMUX, the current UART cannot be used for AT commands until PPP is stopped by LCP termination.

Syntax

```
AT#XPPP=<op>[,<cid>]
```

The parameters and their defined values are the following:

<op>

- `0` - Stop PPP.
- `1` - Start PPP.

<cid>

Integer. The PDN connection to be used for PPP. It represents `cid` in the `+CGDCONT` command. Its default value is `0`, which represents the default PDN connection.

Note:

Other sockets cannot use the same PDN connection. See *Socket AT commands* for more information.

Unsolicited notification

```
#XPPP: <running>,<peer_connected>,<cid>
```

The parameters and their defined values are the following:

<running>

- `0` - PPP is stopped.
- `1` - PPP is running.

<peer_connected>

- `0` - No peer is connected.
- `1` - A peer is connected.

<cid>

Integer. The PDN connection used for PPP.

When you activate a PDN connection used for PPP, the `#XPPP: 1,0,<cid>` notification is sent and the PPP process starts and takes ownership of the associated serial channel. If you use the CMUX , the PPP connection starts at the alternative DLC channel, which can be controlled by the `AT#XCMUX` command. Without CMUX, the PPP starts on the current UART, and AT commands cannot be used until PPP is stopped by the LCP termination message. Similarly, no more AT notifications are sent over the same UART and all URC messages are lost until PPP is stopped.

Note:

PPP auto-start uses `+CGEV:` URCs inside Serial Modem. These URCs are not forwarded to the host unless you enable them with `AT+CGEREP=1`. The host receives PPP status through the `#XPPP` notification.

PPP Link termination

PPP link termination behaves differently in various scenarios.

When the PPP is started on the UART without CMUX, the PPP link termination always returns the channel back to AT command mode.

When the PPP is started over CMUX, the behavior depends on the termination reason and channel configuration:

Table 1: PPP link termination reason

Reason	Description
PDN connection lost when PPP is running on DLC channel 2	The PPP link is terminated using the LCP Terminate-Request message. The PPP module keeps waiting for the PDN connection to be re-established to restart PPP automatically. The Remote peer might wait for LCP Config-Requests to re-establish the link.
PDN connection lost when PPP is running on DLC channel 1	The PPP link is terminated using the LCP Terminate-Request message. The PPP module stops without trying to restart. DLC channel 1 is returned to AT command mode. Traffic on DLC channel 2 is ignored.
Peer disconnection using the LCP Terminate-Request	The PPP module stops PPP and releases the CMUX channel. It does not attempt to restart.
PPP stopped by AT command	The PPP module stops PPP and releases the CMUX channel. It does not attempt to restart.

As the `` `AT#XPPP=1` `` command is designed to restart the PPP connection automatically when the DLC channel used for PPP is the second channel (DLC channel 2). This is to allow the PPP connection to recover from temporary network losses without your intervention while still allowing you to stop PPP when needed. This behavior differs from the standard `ATD*99#` command, which is not implemented on Serial Modem.

Examples

The following examples assume CMUX is already established so that the AT channel is usable while PPP is running on a separate CMUX channel.

PPP with default PDN connection:

```
// Start PPP.
AT#XPPP=1
```

(continues on next page)

(continued from previous page)

```
OK

AT+CFUN=1

OK

// PPP is started and waits for a peer to connect.
#XPPP: 1,0,0

// Peer connects to |SM|'s PPP.
#XPPP: 1,1,0

// Peer disconnects.
#XPPP: 1,0,0

// |SM| stops PPP when a peer disconnects.
#XPPP: 0,0,0

AT+CFUN=4

OK
```

PPP with non-default PDN connection:

```
// Exemplary PDN connection creation.
// Note: APN depends on operator and additional APNs may not be
supported by the operator.
AT+CGDCONT=1,"IP","internet2"

OK

// Start PPP with the created PDN connection.
AT#XPPP=1,1

OK

AT+CFUN=1

OK

// Activate the created PDN connection.
AT+CGACT=1,1

// PPP is automatically started when the PDN connection set for PPP
has been activated.
#XPPP: 1,0,1
```

(continues on next page)

(continued from previous page)

```
// Peer connects to |SM|'s PPP.
#XPPP: 1,1,1
```

Connection recovery for network loss. This requires the PPP on the peer side to keep retrying or waiting for LCP Config-Requests.

```
// Simulate connection loss by activating the flight mode.
AT+CFUN=4

OK

// PPP is automatically stopped when the PDN connection has been
deactivated.
#XPPP: 0,0,0

// Reactivate the connection.
AT+CFUN=1

OK

// PPP is automatically restarted when the PDN connection has been
reactivated.
#XPPP: 1,0,0

// Peer connects to |SM|'s PPP.
#XPPP: 1,1,0
```

Read command

The read command allows you to get the status of PPP.

Syntax

```
AT#XPPP?
```

Response syntax

```
#XPPP: <running>,<peer_connected>,<cid>
```

The parameters and their defined values are the following:

<running>

- `0` - PPP is stopped.
- `1` - PPP is running.

<peer_connected>

- `0` - No peer is connected.
- `1` - A peer is connected.

<cid>
Integer. The PDN connection used for PPP.

When you activate a PDN connection used for PPP, the `#XPPP: 1,0,<cid>` notification is sent and the PPP process starts and takes ownership of the associated serial channel. If you use the CMUX , the PPP connection starts at the alternative DLC channel, which can be controlled by the `AT#XCMUX` command. Without CMUX, the PPP starts on the current UART, and AT commands cannot be used until PPP is stopped by the LCP termination message. Similarly, no more AT notifications are sent over the same UART and all URC messages are lost until PPP is stopped.

Note:

PPP auto-start uses `+CGEV:` URCs inside Serial Modem. These URCs are not forwarded to the host unless you enable them with `AT+CGEREP=1`. The host receives PPP status through the `#XPPP` notification.

3.15 SMS AT commands

This page describes SMS-related AT commands.

Note:

Serial Modem's SMS implementation is based on the SMS library, which uses the following AT commands: `AT+CMGS`, `AT+CNMI`, `AT+CNMA`. These modem AT commands should not be used in conjunction with the `#XSMS` command by the host in order to avoid unexpected behavior. The implementation always acknowledges incoming `+CMT` and `+CDS` notifications with `AT+CNMA` command.

3.15.1 SMS support #XSMS

The `#XSMS` command supports functionalities for sending and receiving SMS messages.

Set command

The set command allows you to start or stop SMS, as well as send SMS text. Only GSM-7 encoding is supported, 8-bit and UCS2 encodings are not supported.

Syntax

```
AT#XSMS=<op>[,<number>,<text>]
```

The parameters and their defined values are the following:

<op>

- `0` - Stop SMS.
- `1` - Start SMS, ready to receive.
- `2` - Send SMS.

<number>
String. The SMS recipient's phone number, including the country code (for example `+81xxxxxxx`).

<text>
String. The SMS text to be sent.

Unsolicited notification This is the notification syntax when an SMS message is received:

```
#XSMS: <datetime>,<number>,<text>
```

The parameters and their defined values are the following:

<datetime>
String. The time when the SMS is received. It has a format of `YY-MM-DD HH:MM:SS UTC[+|-]HH:MM`.

<number>
String. The SMS sender's phone number.

<text>
String. The SMS text that has been received.

When receiving concatenated SMS messages, there will be only one notification.

Example

```
at#xsms=1

OK
at#xsms=2,"+8190xxxxxxxx","SM test"

OK

#XSMS: "21-05-24 11:58:22 UTC+09:30","090xxxxxxx","Tested OK"

at#xsms=2,"+8190xxxxxxxx","0123456789abcdefghijklmnopqrstuvwxyzABCDE
FGHIJKLMNOPQRSTUVWXYZ0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJK
LMNOPQRSTUVWXYZ0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQ
RSTUVWXYZ"

OK

#XSMS: "21-05-24 13:29:47 UTC-09:00","090xxxxxxx","0123456789abcdef
```

(continues on next page)

(continued from previous page)

```
ghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789abcdefghijkl
mnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789abcdefghijklmnopqr
stuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ"
```

Read command

The read command is not supported.

Test command

The test command tests the existence of the command and provides information about the type of its subparameters.

Syntax

```
AT#XSMS=?
```

Response syntax

```
#XSMS: <list of op value>,<number>,<text>
```

Examples

```
at#xsms=?
#XSMS: (0,1,2),<number>,<text>
OK
```

3.16 Socket AT commands

This page describes socket-related AT commands. The application can open up to 8 sockets.

For more information on the networking services, see the Zephyr Network APIs.

Note:

A socket cannot be created on a PDN connection that is already in use by a raw socket. A raw socket cannot be created on a PDN connection that is already in use by another socket. The reason is that a raw socket cannot keep its data separate from the data of another IP socket when both are operating on the same PDN. When a raw socket is active, it intercepts all downlink data intended for other sockets on the same PDN, which disrupts normal socket operations. Raw sockets are used internally by the following AT commands: `AT#XPPP` and `AT#XPING`.

3.16.1 Socket #XSOCKET

The #XSOCKET command allows you to open a socket and to check the socket handle.

Set command

The set command allows you to open a socket.

Syntax

```
AT#XSOCKET=<family>,<type>,<role>[,<cid>]
```

The parameters and their defined values are the following:

<family>

- 1 - IP protocol family version 4.
- 2 - IP protocol family version 6.
- 3 - Raw packet family. The `<type>` parameter must be 3 and the `<role>` parameter must be `0`.

<type>

- 1 - Set `SOCK_STREAM` for the stream socket type using the TCP protocol.
- 2 - Set `SOCK_DGRAM` for the datagram socket type using the UDP protocol.
- 3 - Set `SOCK_RAW` for the raw socket type using a generic packet protocol. The `<family>` parameter must be 3 and the `<role>` parameter must be `0`.

<role>

- `0` - Client.
- 1 - Server.

<cid>

Integer. Indicates the used PDN connection. It represents `cid` in the `+CGDCONT` command. Its default value is `0`.

Response syntax

```
#XSOCKET: <handle>,<type>,<protocol>
```

The parameters and their defined values are the following:

<handle>

Integer. Socket handle (ID) that can be interpreted as follows:

- Positive or `0` - The socket opened successfully.
- Negative - The socket failed to open.

<type>

- 1 - Set `SOCK_STREAM` for the stream socket type using the TCP protocol.

- 2 - Set SOCK_DGRAM for the datagram socket type using the UDP protocol.
- 3 - Set SOCK_RAW for the raw socket type using a generic IP protocol.

<protocol>

- 0 - IPPROTO_IP.
- 6 - IPPROTO_TCP.
- 17 - IPPROTO_UDP.

Examples

```
AT#XSOCKET=1,1,0
#XSOCKET: 0,1,6
OK
AT#XSOCKET=1,2,0
#XSOCKET: 1,2,17
OK
AT#XSOCKET=2,1,0
#XSOCKET: 2,1,6
OK
AT#XSOCKET=1,3,0
#XSOCKET: 3,3,0
OK
```

Read command

The read command allows you to list all sockets that have been opened.

Syntax

```
AT#XSOCKET?
```

Response syntax

```
#XSOCKET: <handle>,<family>,<role>,<type>,<cid>
```

The parameters and their defined values are the following:

<handle>
Integer. When positive or 0, the socket is valid.

<family>
Present only in the response to a request to open the socket.

- 1 - IP protocol family version 4.
- 2 - IP protocol family version 6.
- 3 - Packet family.

<role>

- `0` - Client.
- `1` - Server.

<type>

- `1` - Set `SOCK_STREAM` for the stream socket type using the TCP protocol.
- `2` - Set `SOCK_DGRAM` for the datagram socket type using the UDP protocol.
- `3` - Set `SOCK_RAW` for the raw socket type using a generic packet protocol.

<cid>

Integer. Indicates the used PDN connection. It represents `cid` in the `+CGDCONT` command.

Example

```
AT#XSOCKET?
#XSOCKET: 0,1,0,1,0
#XSOCKET: 1,2,0,1,0
#XSOCKET: 4,1,0,2,1
OK
```

Test command

The test command tests the existence of the command and provides information about the type of its subparameters.

Syntax

```
AT#XSOCKET=?
```

Response syntax

```
#XSOCKET: <handle>,<list of ops>,<list of types>,<list of roles>,
<cid>
```

Example

```
AT#XSOCKET=?
#XSOCKET: <handle>,(1,2),(1,2,3),(0,1),<cid>
OK
```

3.16.2 Secure socket #XSSOCKET

The #XSSOCKET command allows you to open a secure socket and to check the socket handle.

Note:

TLS and DTLS servers are currently not supported.

Note:

A secure socket cannot be created on a PDN connection that is already in use by a raw socket. See *Socket AT commands* for more information.

Set command

The set command allows you to open a secure socket.

Syntax

```
AT#XSSOCKET=<family>,<type>,<role>,<sec_tag>[,<peer_verify>[,<cid>]]
```

The parameters and their defined values are the following:

<family>

- 1 - IP protocol family version 4.
- 2 - IP protocol family version 6.

<type>

- 1 - Set SOCK_STREAM for the stream socket type using the TLS 1.2 protocol.
- 2 - Set SOCK_DGRAM for the datagram socket type using the DTLS 1.2 protocol.

<role>

- 0 - Client.
- 1 - Server.

<sec_tag>

Integer. Indicates to the modem the credential of the security tag to be used for establishing a secure connection. It is associated with a credential, that is, a certificate or PSK. The credential must be stored on the modem side beforehand.

Note:

When <role> has a value of 1, <sec_tag> is not supported.

<peer_verify>

- 0 - None (default for server role).
- 1 - Optional.
- 2 - Required (default for client role).

<cid>
Integer. Indicates the used PDN connection. It represents `cid` in the `+CGDCONT` command. Its default value is `0`.

Response syntax

```
#XSSOCKET: <handle>,<type>,<protocol>
```

The parameters and their defined values are the following:

<handle>
Integer. Socket handle (ID) that can be interpreted as follows:

- Positive or `0` - The socket opened successfully.
- Negative - The socket failed to open.

<type>

- `1` - `SOCK_STREAM` for the stream socket type using the TLS 1.2 protocol.
- `2` - `SOCK_DGRAM` for the datagram socket type using the DTLS 1.2 protocol.

<protocol>

- `258` - IPPROTO_TLS_1_2.
- `273` - IPPROTO_DTLS_1_2.

Examples

```
AT#XSSOCKET=1,1,0,16842753,2
#XSSOCKET: 0,1,258
OK
AT#XSSOCKET=1,2,0,16842753
#XSSOCKET: 1,2,273
OK
```

Read command

The read command allows you to check the secure socket handle.

Syntax

```
AT#XSSOCKET?
```

Response syntax

```
#XSSOCKET: <handle>,<family>,<role>,<type>,<sec_tag>,<cid>
```

The parameters and their defined values are the following:

<handle>
Integer. When positive or `0`, the socket is valid.

<family>

- `1` - IP protocol family version 4.
- `2` - IP protocol family version 6.

<role>

- `0` - Client.
- `1` - Server.

<type>

- `1` - `SOCK_STREAM` for the stream socket type using the TLS 1.2 protocol.
- `2` - `SOCK_DGRAM` for the datagram socket type using the DTLS 1.2 protocol.

<sec_tag>
Integer. Indicates to the modem the credential of the security tag to be used for establishing a secure connection.

<cid>
Integer. Indicates the used PDN connection. It represents `cid` in the `+CGDCONT` command.

Example

```
AT#XSSOCKET?
#XSSOCKET: 0,1,0,1,16842753,0
OK
```

Test command

The test command tests the existence of the command and provides information about the type of its subparameters.

Syntax

```
AT#XSSOCKET=?
```

Response syntax

```
#XSSOCKET: <handle>,<list of ops>,<list of types>,<list of roles>,
<sec_tag>,<peer_verify>,<cid>
```

Example

```
AT#XSSOCKET=?
#XSSOCKET: <handle>,(1,2),(1,2),(0,1),<sec_tag>,<peer_verify>,<cid>
OK
```

3.16.3 Close socket #XCLOSE

The #XCLOSE command allows you to close one or all sockets.

Set command

The set command allows you to close a specific socket or all open sockets.

Syntax

```
AT#XCLOSE[=<handle>]
```

The parameters and their defined values are the following:

<handle>
Integer. Optional. Specifies the socket handle to close. This is the handle value returned from #XSOCKET or #XSSOCKET commands. If omitted, all open sockets will be closed.

Response syntax

```
#XCLOSE: <handle>,<result>
```

The parameters and their defined values are the following:

<handle>
Integer. The handle of the closed socket.

<result>
Integer. The result of closing the socket. When `0`, the socket was closed successfully.

When closing all sockets (no handle parameter provided), multiple responses will be sent, one for each socket that was closed.

Examples Close a specific socket:

```
AT#XCLOSE=0
#XCLOSE: 0,0
OK
```

Close all open sockets:

```
AT#XCLOSE
#XCLOSE: 0,0
#XCLOSE: 1,0
#XCLOSE: 2,0
OK
```

Read command

The read command is not supported.

Test command

The test command is not supported.

3.16.4 Socket options #XSOCKETOPT

The #XSOCKETOPT command allows you to get and set socket options.

Set command

The set command allows you to get and set socket options.

Syntax

```
AT#XSOCKETOPT=<handle>,<op>,<name>[,<value>]
```

The parameters and their defined values are the following:

<handle>
Integer. Specifies the socket handle returned from #XSOCKET or #XSSOCKET commands.

<op>

- 0 - Get.
- 1 - Set.

<name>

- 2 - AT_SO_REUSEADDR (set-only).
  - <value> is an integer that indicates whether the reuse of local addresses is enabled. It is 0 for disabled or 1 for enabled.
- 20 - AT_SO_RCVTIMEO.
  - <value> is an integer that indicates the receive timeout in seconds.
- 21 - AT_SO_SNDTIMEO.
  - <value> is an integer that indicates the send timeout in seconds.
- 30 - AT_SO_SILENCE_ALL.

  - <value> is an integer that indicates whether ICMP echo replies for IPv4 and IPv6 are disabled. It is 0 for allowing ICMP echo replies or 1 for disabling them.

- 31 - AT_SO_IP_ECHO_REPLY.
  - <value> is an integer that indicates whether ICMP echo replies for IPv4 are enabled. It is 0 for disabled or 1 for enabled.
- 32 - AT_SO_IPV6_ECHO_REPLY.
  - <value> is an integer that indicates whether ICMP echo replies for IPv6 are enabled. It is 0 for disabled or 1 for enabled.
- 34 - AT_SO_KEEPOPEN.
  - <value> is an integer that indicates whether socket is kept open when the modem enters the flight mode (+CFUN=4). It is 0 for disabled or 1 for enabled.
- 40 - AT_SO_BINDTOPDN (set-only).
  - <value> is an integer that indicates the packet data network ID to bind to.
- 55 - AT_SO_TCP_SRV_SESSTIMEO.
  - <value> is an integer that indicates the TCP server session inactivity timeout for a socket. It accepts values from the range 0 to 135, where 0 is no timeout and 135 is 2 hours, 15 minutes.
- 61 - AT_SO_RAI (set-only). Release Assistance Indication (RAI).
  - <value> The option accepts an integer, indicating the type of RAI. Accepted values for the option are:
    * 1 - RAI_NO_DATA. Indicates that the application does not intend to send more data. This socket option applies immediately and lets the modem exit connected mode more quickly.
    * 2 - RAI_LAST. Indicates that the application does not intend to send more data after the next call to send() or sendto(). This lets the modem exit connected mode more quickly after sending the data.
    * 3 - RAI_ONE_RESP. Indicates that the application is expecting to receive just one data packet after the next call to send() or sendto(). This lets the modem exit connected mode more quickly after having received the data.
    * 4 - RAI_ONGOING. Indicates that the application is expecting to receive just one data packet after the next call to send() or sendto(). This lets the modem exit connected mode more quickly after having received the data.
    * 5 - RAI_WAIT_MORE. Indicates that the socket is in active use by a server application. This lets the modem stay in connected mode longer.
- 62 - AT_SO_IPV6_DELAYED_ADDR_REFRESH.
  - <value> is an integer that indicates whether delayed IPv6 address refresh is enabled. It is 0 for disabled or 1 for enabled.

See nRF socket options for explanation of the supported options.

Examples

```
AT#XSOCKETOPT=0,1,20,30
OK
```

```
AT#XSOCKETOPT=0,0,20
#XSOCKETOPT: 0,30
OK
```

Read command

The read command is not supported.

Test command

The test command tests the existence of the command and provides information about the type of its subparameters.

Syntax

```
AT#XSOCKETOPT=?
```

Response syntax

```
#XSOCKETOPT: <handle>,<list of ops>,<name>,<value>
```

Example

```
AT#XSOCKETOPT=?
#XSOCKETOPT: <handle>,(0,1),<name>,<value>
OK
```

3.16.5 Secure socket options #XSSOCKETOPT

The #XSSOCKETOPT command allows you to get and set socket options for secure sockets.

Set command

The set command allows you to get and set socket options for secure sockets.

Syntax

```
AT#XSSOCKETOPT=<handle>,<op>,<name>[,<value>]
```

The parameters and their defined values are the following:

<handle>
Integer. Specifies the socket handle returned from #XSSOCKET command.

<op>

- 0 - Get.
- 1 - Set.

<name>

- 2 - AT_TLS_HOSTNAME.
  - <value> is a string that indicates the hostname to check against during TLS handshakes. It can be NULL to disable hostname verification.
- 4 - AT_TLS_CIPHERSUITE_USED (get-only). The TLS cipher suite is chosen during the TLS handshake.
- 5 - AT_TLS_PEER_VERIFY.
  - <value> is an integer that indicates the peer verification level. It is 0 for none, 1 for optional, or 2 for required.
- 12 - AT_TLS_SESSION_CACHE.
  - <value> is an integer that indicates whether to use TLS session caching. It is 0 for disabled or 1 for enabled.
- 13 - AT_TLS_SESSION_CACHE_PURGE (set-only). Indicates that the TLS session cache must be deleted.
  - <value> can be any integer value.
- 14 - AT_TLS_DTLS_CID (set-only).
  - <value> is an integer that indicates the DTLS connection identifier setting. It can be one of the following values:
    * 0 - TLS_DTLS_CID_DISABLED.
    * 1 - TLS_DTLS_CID_SUPPORTED.
    * 2 - TLS_DTLS_CID_ENABLED.

  See NRF_SO_SEC_DTLS_CID for more details regarding the allowed values.
- 15 - AT_TLS_DTLS_CID_STATUS (get-only). It is the DTLS connection identifier status. It can be retrieved after the DTLS handshake. See NRF_SO_SEC_DTLS_CID_STATUS for more details regarding the returned values.
- 18 - AT_TLS_DTLS_HANDSHAKE_TIMEO.
  - <value> is an integer that indicates the DTLS handshake timeout in seconds. It can be one of the following values: 1, 3, 7, 15, 31, 63, 123.
- 19 - AT_TLS_DTLS_CONN_SAVE (set-only). Write-only socket option to save DTLS connection.

  - <value> can be any integer value, which will be ignored.

  After this option is successfully called, you must call AT_TLS_DTLS_CONN_LOAD before continuing to communicate on the socket.

- 20 - AT_TLS_DTLS_CONN_LOAD (set-only). Write-only socket option to load DTLS connection.
  - <value> can be any integer value, which will be ignored.
- 22 - AT_TLS_DTLS_FRAG_EXT.
  - <value> is an integer that indicates the use of the DTLS fragmentation extension specified in RFC 6066. It can be one of the following values:
    * 0 - DTLS_FRAG_EXT_DISABLED.
    * 1 - DTLS_FRAG_EXT_512_ENABLED.
    * 2 - DTLS_FRAG_EXT_1024_ENABLED.

See nRF socket options for explanation of the supported options.

Example

```
AT#XSSOCKETOPT=0,1,5,2
OK
```

Read command

The read command is not supported.

Test command

The test command tests the existence of the command and provides information about the type of its subparameters.

Syntax

```
AT#XSSOCKETOPT=?
```

Response syntax

```
#XSSOCKETOPT: <handle>,<list of ops>,<name>,<value>
```

Example

```
AT#XSSOCKETOPT=?
#XSSOCKETOPT: <handle>,(0,1),<name>,<value>
OK
```

3.16.6 Socket binding #XBIND

The #XBIND command allows you to bind a socket with a local port.

You can use this command with TCP and UDP, and it is needed for incoming UDP data, where the remote end targets a particular port.

Set command

The set command allows you to bind a socket with a local port.

Syntax

```
AT#XBIND=<handle>,<port>
```

The parameters and their defined values are the following:

<handle>
Integer. Specifies the socket handle returned from #XSOCKET or #XSSOCKET commands.

<port>
Integer. An unsigned 16-bit integer (0 - 65535) representing the specific port to use for binding the socket. 0 is a special value that indicates that the modem should choose the port automatically.

Example

```
AT#XBIND=0,1234
OK
```

Read command

The read command is not supported.

Test command

The test command is not supported.

3.16.7 Connection #XCONNECT

The #XCONNECT command allows you to connect to a server and to check the connection status.

This command is for TCP and UDP client sockets.

Set command

The set command allows you to connect to a TCP or UDP server.

Syntax

```
AT#XCONNECT=<handle>,<url>,<port>
```

The parameters and their defined values are the following:

<handle>
Integer. Specifies the socket handle returned from #XSOCKET or #XSSOCKET commands.

<url>
String. The hostname or the IP address of the server. The maximum supported size of the hostname is 128 bytes. When using IP addresses, it supports both IPv4 and IPv6.

<port>
Integer. An unsigned 16-bit integer (0 - 65535) representing the port of the TCP or UDP service on the remote server.

Response syntax

```
#XCONNECT: <handle>,<status>
```

The parameters and their defined values are the following:

<handle>
Integer. The socket handle.

<status>

- `0` - Disconnected.
- `1` - Connected.

Examples

```
AT#XCONNECT=0,"test.server.com",1234
#XCONNECT: 0,1
OK
```

```
AT#XCONNECT=1,"192.168.0.1",1234
#XCONNECT: 1,1
OK
```

```
AT#XCONNECT=2,"2a02:c207:2051:8976::1",4567
#XCONNECT: 2,1
OK
```

Read command

The read command is not supported.

Test command

The test command is not supported.

3.16.8 Send data #XSEND

The #XSEND command allows you to send data over TCP, UDP, and raw sockets.

Set command

The set command allows you to send data over the connection.

Syntax

```
AT#XSEND=<handle>,<mode>,<flags>,<data> when ``<mode>`` is ``0`` or
``1``

AT#XSEND=<handle>,<mode>,<flags>[,<data_len>] when ``<mode>`` is
``2``
```

The parameters and their defined values are the following:

<handle>
Integer. Specifies the socket handle returned from #XSOCKET or #XSSOCKET commands.

<mode>

- 0 - String mode. Data is provided directly in the command as the <data> parameter.
- 1 - Hex string mode. Data is provided as a hexadecimal string in the <data> parameter.
- 2 - Data mode. Serial Modem enters *Data mode* for data input.

<flags>
Sending behavior based on the BSD socket definition.

- 0 - No flags set. The request is complete when the data is pushed to the modem buffer.
- 512 - Blocks send operation until the request is acknowledged by the network. The request will not return until the data is pushed to the network, or acknowledged by the network (for TCP), or the timeout given by the AT_SO_SNDTIMEO socket option is reached. Valid timeout values are 1 to 600 seconds.
- 8192 - Send unsolicited #XSENDNTF notification when the request is acknowledged by the network. Unsolicited notification will be sent when the data is

pushed to the network, or acknowledged by the network (for TCP), or the timeout given by the `AT_SO_SNDTIMEO` socket option is reached. Valid timeout values are 1 to 600 seconds. Further sends for the socket are blocked until the unsolicited notification is received.

This is only supported by the following modem firmware:

- mfw_nrf9151-ntn

<data>
String. Required when `<mode>` is `0` (string mode) or `1` (hex string mode). For string mode (`0`), it is a string that contains the data to be sent. For hex string mode (`1`), it is a hexadecimal string representation of the data to be sent. The maximum payload size in hexadecimal string mode is up to 2800 characters (1400 bytes). Not used when `<mode>` is 2 (data mode).

<data_len>
Integer. Optional, only used when `<mode>` is 2 (data mode). Sets the number of bytes to send in data mode. When the required number of bytes are sent, the data mode is exited.

The termination command *CONFIG_SM_DATAMODE_TERMINATOR* is not used in this case.

Note:

UDP packets that exceed the Maximum Transmission Unit (MTU) of any network segment along their path might be dropped or fragmented, increasing the risk of packet loss. Ethernet networks have an MTU of 1500 bytes, which allows a maximum UDP payload of 1472 bytes for IPv4 and 1452 bytes for IPv6. With DTLS sockets, the usable payload size is further reduced due to encryption overhead.

The cellular network MTU can be queried with the `AT+CGCONTRDP` command, but some networks might still drop packets smaller than the reported MTU.

A UDP payload size of 1200 bytes is commonly recommended, especially for IPv6, as it ensures the total packet size remains below the IPv6 minimum MTU of 1280 bytes after accounting for headers and DTLS overhead. Keeping UDP packet sizes well below the theoretical maximum increases the likelihood of successful transmission. You can even use 1024 bytes as a safe size for UDP packets.

Response syntax

```
#XSEND: <handle>,<result_type>,<size>
```

The parameters and their defined values are the following:

<handle>
Integer. The socket handle.

<result_type>

- `0` - Indicates that there are no further notifications.

- `1` - Indicates that an unsolicited notification will be sent when the network acknowledged send is completed.

<size>
Integer. The actual number of bytes that has been sent.

Unsolicited notification For network acknowledged sends (when the `8192` flag is used), an unsolicited notification is sent when the send operation is completed.

This is only supported by the following modem firmware:

- mfw_nrf9151-ntn

```
#XSENDNTF: <handle>,<status>,<size>
```

The parameters and their defined values are the following:

<handle>
Integer. The socket handle.

<status>
The status of the send acknowledged by the network.

- `0` - Success.
- `-1` - Failure.

<size>
Integer. The size of the data sent.

Example

```
AT#XSEND=0,0,0,"Test TCP"
#XSEND: 0,0,8
OK

AT#XSEND=0,1,0,"48656C6C6F"
#XSEND: 0,0,5
OK

AT#XSEND=1,0,8192,"Test notification"
#XSEND: 1,1,17
OK
#XSENDNTF: 1,0,17

AT#XSEND=1,2,8192
OK
Test datamode with flags+++
#XDATAMODE: 0
#XSENDNTF: 1,0,24
```

Read command

The read command is not supported.

Test command

The test command is not supported.

3.16.9 Receive data #XRECV

The #XRECV command allows you to receive data over TCP, UDP, and raw sockets.

Set command

The set command allows you to receive data over the connection.

Syntax

```
AT#XRECV=<handle>,<mode>,<flags>,<timeout>[,<data_len>]
```

The parameters and their defined values are the following:

<handle>
Integer. Specifies the socket handle returned from #XSOCKET or #XSSOCKET commands.

<mode>

- `0` - Binary mode. Data is received as binary data.
- `1` - Hex string mode. Data is received as a hexadecimal string representation.

<flags>
Receiving behavior based on the BSD socket definition.

- `0` - No flags set.
- `2` - Read data without removing it from the socket input queue.
- `64` - Override the operation to non-blocking.
- `256` (TCP only) - Block until the full amount of data can be returned.

<timeout>
Integer. The timeout value in seconds. When `0`, it means no timeout, and it makes this request block indefinitely.

<data_len>
Integer. Optional. The maximum number of bytes to receive. The maximum value is 2048 bytes, which is also the default value when the parameter is omitted.

Response syntax

```
#XRECV: <handle>,<mode>,<size>
<data>
```

The parameters and their defined values are the following:

<handle>
Integer. The socket handle.

<mode>
Integer. The receive mode used.

<size>
Integer. The actual number of bytes received. In case of hex string mode, it represents the number of bytes before conversion to hexadecimal format.

<data>
String. The data being received.

Example

```
AT#XRECV=0,0,0,10
#XRECV: 0,0,7
Test OK
OK

AT#XRECV=0,1,0,10
#XRECV: 0,1,5
74205a6f63
OK
```

Read command

The read command is not supported.

Test command

The test command is not supported.

3.16.10 UDP send data #XSENDTO

The #XSENDTO command allows you to send data over UDP.

Set command

The set command allows you to send data over UDP.

Syntax

```
AT#XSENDTO=<handle>,<mode>,<flags>,<url>,<port>,<data> when ``<mode>
`` is ``0`` or ``1``

AT#XSENDTO=<handle>,<mode>,<flags>,<url>,<port>[,<data_len>] when ``
<mode>`` is ``2``
```

The parameters and their defined values are the following:

<handle>
Integer. Specifies the socket handle returned from #XSOCKET or #XSSOCKET commands.

<mode>

- `0` - String mode. Data is provided directly in the command as the `<data>` parameter.
- `1` - Hex string mode. Data is provided as a hexadecimal string in the `<data>` parameter.
- 2 - Data mode. Serial Modem enters *Data mode* for data input.

<flags>
Sending behavior based on the BSD socket definition.

- `0` - No flags set. The request is complete when the data is pushed to the modem buffer.
- `512` - Blocks send operation until the request is acknowledged by network. The request will not return until the data is pushed to the network, or acknowledged by network (for TCP), or the timeout given by the AT_SO_SNDTIMEO socket option, is reached. Valid timeout values are 1 to 600 seconds.
- `8192` - Send unsolicited `#XSENDNTF` notification when the request is acknowledged by network. Unsolicited notification will be sent when the data is pushed to the network, or acknowledged by network (for TCP), or the timeout given by the AT_SO_SNDTIMEO socket option, is reached. Valid timeout values are 1 to 600 seconds. Further sends for the socket are blocked until the unsolicited notification is received.

  This is only supported by the following modem firmware:

  - mfw_nrf9151-ntn

<url>
String. The hostname or the IP address of the remote peer. The maximum size of the hostname is 128 bytes. When using IP addresses, it supports both IPv4 and IPv6.

<port>
Integer. An unsigned 16-bit integer (0 - 65535) representing the port of the UDP service on the remote peer.

<data>
String. Required when `<mode>` is `0` (string mode) or `1` (hex string mode). For string mode (`0`), it is a string that contains the data to be sent. For hex string mode (`1`), it is a hexadecimal string representation of the data to be sent. The maximum payload size in hexadecimal string mode is up to 2800 characters (1400 bytes). For large packets, it is recommended to use data mode (2) since AT parser's memory limits the maximum size of data that can be sent in string or hex string modes. Not used when `<mode>` is 2 (data mode).

<data_len>
Integer. Optional, only used when `<mode>` is 2 (data mode). Sets the number of bytes to send in data mode. When the required number of bytes are sent, the data mode is exited.

The termination command *CONFIG_SM_DATAMODE_TERMINATOR* is not used in this case.

Note:

UDP packets that exceed the Maximum Transmission Unit (MTU) of any network segment along their path may be dropped or fragmented, increasing the risk of packet loss. Ethernet networks have an MTU of 1500 bytes, which allows a maximum UDP payload of 1472 bytes for IPv4 and 1452 bytes for IPv6. With DTLS sockets, the usable payload size is further reduced due to encryption overhead.

The cellular network MTU can be queried with the `AT+CGCONTRDP` command, but some networks may still drop packets smaller than the reported MTU.

A UDP payload size of 1200 bytes is commonly recommended, especially for IPv6, as it ensures the total packet size remains below the IPv6 minimum MTU of 1280 bytes, after accounting for headers and DTLS overhead. Keeping UDP packet sizes well below the theoretical maximum increases the likelihood of successful transmission. Even 1024 bytes could be used as a safe size for UDP packets.

Note:

With DTLS connections, the connection cannot be established with the `#XSENDTO` command. Instead, it must be established using the `#XCONNECT` command. This is a limitation in the nRF91 modem firmware.

Response syntax

```
#XSENDTO: <handle>,<result_type>,<size>
```

The parameters and their defined values are the following:

<handle>
Integer. The socket handle.

<result_type>

- `0` - Indicates that there are no further notifications.
- `1` - Indicates that an unsolicited notification will be sent when the network acknowledged send is completed.

<size>
Integer. The actual number of bytes that have been sent.

Unsolicited notification For network acknowledged sends (when the 8192 flag is used), an unsolicited notification is sent when the send operation is completed.

This is only supported by the following modem firmware:

- mfw_nrf9151-ntn

```
#XSENDNTF: <handle>,<status>,<size>
```

The parameters and their defined values are the following:

<handle>
Integer. The socket handle.

<status>
The status of the send acknowledged by the network.

- `0` - Success.
- `-1` - Failure.

<size>
Integer. The size of the data sent.

Example

```
AT#XSENDTO=0,0,0,"test.server.com",1234,"Test UDP"
#XSENDTO: 0,0,8
OK

AT#XSENDTO=0,1,0,"test.server.com",1234,"48656C6C6F"
#XSENDTO: 0,0,5
OK

AT#XSENDTO=0,0,8192,"test.server.com",1234,"Test notification"
#XSENDTO: 0,1,17
OK
#XSENDNTF: 0,0,17
```

Read command

The read command is not supported.

Test command

The test command is not supported.

3.16.11 UDP receive data #XRECVFROM

The #XRECVFROM command allows you to receive data over UDP.

Set command

The set command allows you to receive data over UDP.

Syntax

```
AT#XRECVFROM=<handle>,<mode>,<flags>,<timeout>[,<data_len>]
```

The parameters and their defined values are the following:

<handle>
Integer. Specifies the socket handle returned from #XSOCKET or #XSSOCKET commands.

<mode>

- `0` - Binary mode. Data is received as binary data.
- `1` - Hex string mode. Data is received as a hexadecimal string representation.

<flags>
Receiving behavior based on the BSD socket definition.

- `0` - No flags set.
- `2` - Read data without removing it from the socket input queue.
- `64` - Override the operation to non-blocking.

<timeout>
Integer. The timeout value in seconds. When `0`, it means no timeout, and it makes this request block indefinitely.

<data_len>
Integer. Optional. The maximum number of bytes to receive. The maximum value is 2048 bytes, which is also the default value when the parameter is omitted.

Response syntax

```
#XRECVFROM: <handle>,<mode>,<size>,"<ip_addr>",<port>
<data>
```

The parameters and their defined values are the following:

<handle>
Integer. The socket handle.

<mode>
Integer. The receive mode used.

<size>
Integer. The actual number of bytes received. In the case of hex string mode, it represents the number of bytes before conversion to hexadecimal format.

<ip_addr>
String. The IPv4 or IPv6 address of the remote peer.

<port>
Integer. The UDP port of the remote peer.

<data>
String. The data being received.

Example

```
AT#XRECVFROM=0,0,0,10
#XRECVFROM: 0,0,7,"192.168.1.100",24210
Test OK
OK

AT#XRECVFROM=0,1,0,10
#XRECVFROM: 0,1,7,"192.168.1.100",24210
54657374205a4d
OK
```

Read command

The read command is not supported.

Test command

The test command is not supported.

3.16.12 Asynchronous socket polling #XAPOLL

The #XAPOLL command allows you to receive Unsolicited Result Code (URC) notifications for poll events on sockets.

Set command

The set command allows you to activate or deactivate asynchronous polling for sockets.

Syntax

```
AT#XAPOLL=[<handle>],<op>,[<events>]
```

The parameters and their defined values are the following:

<handle>
Integer. Optional. Sets the socket handle to poll. Handles are sent in the AT#XSOCKET or AT#XSSOCKET responses. Handles can also be obtained using the AT#XSOCKET? or AT#XSSOCKET? command. If the handle is omitted, the operation applies to all open sockets and to any new sockets that are created.

<op>

- 0 - Stop asynchronous polling.
- 1 - Start asynchronous polling.

<events>
Integer. Optional. Interpreted as a bit field representing the events to poll for. Permanent error and closure events (POLLERR, POLLHUP, and POLLNVAL) are always polled. The value can be any combination of the following values summed up:

- 0 - Poll the default events.

- 1 - Read events (POLLIN) are polled, in addition to the default events.
- 4 - Write events (POLLOUT) are polled, in addition to the default events.

Response syntax When the asynchronous socket events are enabled, Serial Modem sends events as URC notifications.

- For POLLIN events, the URC notification is sent only for the first incoming data on the socket. AT#XRECV or AT#XRECVFROM command will re-enable the URC notification for the next incoming data.
- For POLLOUT events, the URC notification is sent only for the first time when the socket is ready for writing. AT#XSEND or AT#XSENDTO command will re-enable the URC notification for the next time when the socket is ready for writing.
- For POLLERR, POLLHUP, and POLLNVAL events, the URC notification is sent only once for each socket. No further URC notifications will be sent for the same socket.

Note:

When closing the socket with #XCLOSE, no closure event will be sent.

```
#XAPOLL: <handle>,<revents>
```

The parameters and their defined values are the following:

<handle>
Integer. The handle of the socket that has events.

<revents>
Integer. Interpreted as a bit field representing the returned events as a combination of POLLIN (1), POLLOUT (4), POLLERR (8), POLLHUP (16), and POLLNVAL (32) summed up. Hexadecimal representation is avoided to support AT command parsers that do not support hexadecimal values.

Example

```
// Start asynchronous polling for all sockets with POLLIN and
POLLOUT events.
AT#XAPOLL=,1,5

OK

// Create a TCP socket and connect to the test server.
AT#XSOCKET=1,1,0

#XSOCKET: 0,1,6

OK
```

(continues on next page)

(continued from previous page)

```
AT#XCONNECT=0,"test.server.com",1234

#XCONNECT: 0,1

OK

#XAPOLL: 0,4

// Send data to the test server, which will echo it back.
AT#XSEND=0,0,0,"echo"

#XSEND: 0,0,4

OK

#XAPOLL: 0,4

// Test server sends the data back and closes the connection. POLLIN
and POLLHUP events are received.
#XAPOLL: 0,17

AT#XRECV=0,0,0,1

#XRECV: 0,0,4
echo
OK

// Close the TCP socket.
AT#XCLOSE=0

#XCLOSE: 0,0

OK

// Create a UDP socket and send network acknowledge data to another
test server, which does not echo.
AT#XSOCKET=2,2,0

#XSOCKET: 0,2,17

OK

#XAPOLL: 0,4

AT#XCONNECT=0,"no_echo.test.server.com",1234

#XCONNECT: 0,1
```

(continues on next page)

(continued from previous page)

```
OK

// Send data to the test server with network acknowledged send flag.
AT#XSEND=0,0,8192,"no echo"

#XSEND: 0,1,7

OK

// Unsolicited notification for network acknowledged send.
#XSENDNTF: 0,0,7

#XAPOLL: 0,4

// Close the UDP socket.
AT#XCLOSE=0

#XCLOSE: 0,0

OK

// Stop asynchronous polling for all sockets.
AT#XAPOLL=,0

OK
```

Read command

The read command lists the socket handles with the events that are being polled.

Syntax

```
AT#XAPOLL?
```

Response syntax

```
#XAPOLL: <handle>,<events>
```

The parameters and their defined values are the following:

<handle>
Integer. The handle of the socket that is being polled.

<events>
Integer. Interpreted as a bit field representing the events that are being polled. Permanent error and closure events (`POLLERR`, `POLLHUP`, and `POLLNVAL`) are always polled. The value can be any combination of the following values:

- `0` - Poll the default events.
- `1` - Poll read events (`POLLIN`) in addition to the default events.
- `4` - Poll write events (`POLLOUT`) in addition to the default events.

Example

```
// Start asynchronous polling for all sockets with POLLIN event.
AT#XAPOLL=,1,1

OK

// Create socket 0.
AT#XSOCKET=1,1,0

#XSOCKET: 0,1,6

OK

// Create socket 1 to show that poll events can be configured per
socket.
AT#XSOCKET=1,1,0

#XSOCKET: 1,1,6

OK

// Activate only POLLOUT event polling for socket 1.
AT#XAPOLL=1,1,4

OK

// Create socket 2 to show that POLLIN for all sockets is still in
effect.
AT#XSOCKET=1,1,0

#XSOCKET: 2,1,6

OK

// Read the current poll settings.
AT#XAPOLL?

#XAPOLL: 0,1

#XAPOLL: 1,4

#XAPOLL: 2,1

OK
```

(continues on next page)

(continued from previous page)

```
// Stop asynchronous polling for all sockets.
AT#XAPOLL=,0

OK

// Close all sockets.
AT#XCLOSE

#XCLOSE: 0,0

#XCLOSE: 1,0

#XCLOSE: 2,0

OK
```

Test command

The test command provides information about the command and its parameters.

Syntax

```
AT#XAPOLL=?
```

Response syntax

```
#XAPOLL: <handle>,(stop/start),(events)
```

Example

```
AT#XAPOLL=?

#XAPOLL: <handle>,(0,1),(0,1,4,5)

OK
```

3.16.13 Configure socket receive #XRECVCFG

The socket receive configuration command allows you to configure the following aspects of a socket:

- Automatic data reception
- Automatic data reception in hex format

Set command

The set command allows you to configure the socket receive configuration of all sockets or a specific socket.

Syntax

```
AT#XRECVCFG=[<handle>],<auto_reception_flags>[,<hex_format>]
```

The parameters and their defined values are the following:

<handle>
Integer. Identifies the socket handle. If omitted, the command applies to all opened sockets, whether already open or opened in the future.

<auto_reception_flags>
Integer. Specifies the automatic reception flags. Can be a combination of the following values summed up:

- `0` - No automatic data reception.
- `1` - Automatic data reception in AT-command mode.
- `2` - Automatic data reception in data mode.

<hex_format>
Integer. Specifies the hex format for automatically received data. Applies only when automatic data reception is enabled.

- `0` - Data is received in binary format (default).
- `1` - Data is received in hex string format (supported only in AT-command mode).

Response syntax When in the AT-command mode and the automatic data reception is enabled, Serial Modem sends `#XRECV` and `#XRECVFROM` responses as URC notifications when data is received. `#XRECV` is used for TCP, raw, and connected UDP sockets, while `#XRECVFROM` is used for unconnected UDP sockets.

```
#XRECV: <handle>,<mode>,<size>
<data>
```

The parameters and their defined values are the following:

<handle>
Integer. The socket handle.

<mode>
Integer. The receive mode used.

<size>
Integer. The actual number of bytes received. In case of hex string mode, it represents the number of bytes before conversion to hexadecimal format.

<data>
String. The data being received.

```
#XRECVFROM: <handle>,<mode>,<size>,"<ip_addr>",<port>
<data>
```

The parameters and their defined values are the following:

<handle>
Integer. The socket handle.

<mode>
Integer. The receive mode used.

<size>
Integer. The actual number of bytes received. In the case of hex string mode, it represents the number of bytes before conversion to hexadecimal format.

<ip_addr>
String. The IPv4 or IPv6 address of the remote peer.

<port>
Integer. The UDP port of the remote peer.

<data>
String. The data being received.

Note:

<CR><LF> is sent after <data>. This differs from the standard #XRECV and #XRECVFROM responses where <CR><LF> is sent before OK.

When in data mode and the automatic data reception is enabled, Serial Modem sends the received data as is, without any additional headers or formatting.

Example

```
// Enable automatic data reception in AT-command mode in binary
format for all sockets.
AT#XRECVCFG=,1,0

OK

AT#XSOCKET=1,1,0

#XSOCKET: 0,1,6

OK

AT#XCONNECT=0,"test.server.com",1234

#XCONNECT: 0,1
```

(continues on next page)

(continued from previous page)

```
OK

// Send data to the test server, which will echo it back.
AT#XSEND=0,0,0,"Test"

#XSEND: 0,0,4

OK

// Data is automatically received.
#XRECV: 0,0,4
Test

// Enable automatic data reception in AT-command mode in hex string
format for socket 0.
AT#XRECVCFG=0,1,1

OK

// Send data to the test server, which will echo it back.
AT#XSEND=0,0,0,"Test hex"

#XSEND: 0,0,8

OK

// Data is automatically received in hex string format.
#XRECV: 0,1,8
5465737420686578

// Enable automatic reception of data in data mode.
AT#XRECVCFG=0,2,0

OK

// Enter data mode and send data to the test server, which will echo
it back.
AT#XSEND=0,2,0

OK
DATA TEST
DATA TEST
+++
#XDATAMODE: 0

// Create a new UDP socket. Data reception in AT-command mode in
binary format is enabled for it.
AT#XSOCKET=1,2,0
```

(continues on next page)

(continued from previous page)

```
#XSOCKET: 1,2,17

OK

// Send data with unconnected UDP socket to the test server, which
will echo it back with a delay.
AT#XSENDTO=1,0,0,"test.server.com",1235,"Delayed UDP data"

#XSENDTO: 1,0,16

OK

// Enter data mode with socket 0.
AT#XSEND=0,2,0

OK

// Long operations during data mode.

// Exiting the data mode allows the delayed UDP data to be received.
+++
#XDATAMODE: 0

// Unconnected UDP socket automatically receives the delayed data
with RECVFROM.
#XRECVFROM: 1,0,16,"111.112.113.114",1235
Delayed UDP data

// Disable automatic data reception for all sockets.
AT#XRECVCFG=,0

OK

AT#XCLOSE

#XCLOSE: 0,0

#XCLOSE: 1,0

OK
```

Read command

The read command allows you to check the receive configuration settings of sockets.

Syntax

```
AT#XRECVCFG?
```

Response syntax

```
#XRECVCFG: <handle>,<auto_reception_flags>,<hex_format>
```

The parameters and their defined values are the following:

<handle>
Integer. Identifies the socket handle.

<auto_reception_flags>
Integer. Specifies the automatic reception flags. A combination of the following values summed up:

- `0` - No automatic data reception.
- `1` - Automatic data reception in AT-command mode.
- `2` - Automatic data reception in data mode.

<hex_format>
Integer. Specifies the hex format for automatically received data.

- `0` - Data is received in binary format.
- `1` - Data is received in hex string format (supported only in AT-command mode).

Example

```
AT#XRECVCFG=,1,0

OK
AT#XSOCKET=1,1,0

#XSOCKET: 0,1,6

OK
AT#XSOCKET=1,1,0

#XSOCKET: 1,1,6

OK
// Enable automatic data reception in data mode for socket 1.
AT#XRECVCFG=1,2

OK
AT#XRECVCFG?

#XRECVCFG: 0,1,0
```

(continues on next page)

(continued from previous page)

```
#XRECVCFG: 1,2,0

OK
// Disable automatic reception for all sockets.
AT#XRECVCFG=,0

OK
AT#XRECVCFG?

OK
```

Test command

The test command provides information about the command and its parameters.

Syntax

```
AT#XRECVCFG=?
```

Response syntax

```
#XRECVCFG: <handle>,(0,1,2,3),(0,1)
```

Example

```
AT#XRECVCFG=?

#XRECVCFG: <handle>,(0,1,2,3),(0,1)

OK
```

3.16.14 Resolve hostname #XGETADDRINFO

The `#XGETADDRINFO` command allows you to resolve hostnames to IPv4 and IPv6 addresses.

Set command

The set command allows you to resolve hostnames to IPv4 and IPv6 addresses.

Syntax

```
AT#XGETADDRINFO=<hostname>[,<address_family>]
```

The parameters and their defined values are the following:

<hostname>
String. The hostname to resolve.

<address_family>
Integer. Optional. Gives a hint for DNS query on address family.

- `0` - Unspecified address family.
- `1` - IPv4 address family.
- `2` - IPv6 address family.

If not specified, there will be no hint given for the DNS query.

Response syntax

```
#XGETADDRINFO: "<ip_addresses>"
```

The parameters and their defined values are the following:

<ip_addresses>
String. The IPv4 or IPv6 address of the resolved hostname.

Example

```
AT#XGETADDRINFO="google.com"
#XGETADDRINFO: "142.251.42.142"
OK
AT#XGETADDRINFO="google.com",0
#XGETADDRINFO: "172.217.31.142"
OK
AT#XGETADDRINFO="google.com",1
#XGETADDRINFO: "142.251.42.142"
OK
AT#XGETADDRINFO="ipv6.google.com",2
#XGETADDRINFO: "2404:6800:4004:824::200e"
OK
```

Read command

The read command is not supported.

Test command

The test command is not supported.

3.16.15 Socket listen #XLISTEN

The `#XLISTEN` command allows you to put a TCP server socket in listening mode to accept incoming connections.

The socket must be a stream socket (TCP) that has been bound to a non-zero local port using the `#XBIND` command before it can be put in listening mode.

Note:

The nRF91 modem does not support TLS server sockets, so you cannot use the #XLISTEN command with TLS sockets created with the #XSSOCKET command.

Set command

The set command allows you to put a socket in listening mode.

Syntax

```
AT#XLISTEN=<handle>
```

The parameters and their defined values are the following:

<handle>
Integer. Specifies the socket handle returned from #XSOCKET command. The socket must be a stream socket (TCP) that has been bound to a non-zero local port with #XBIND.

Example

```
AT#XLISTEN=0
OK
```

Read command

The read command allows you to list all sockets that are in listening mode.

Syntax

```
AT#XLISTEN?
```

Response syntax

```
#XLISTEN: <handle>,<cid>,<local_port>
```

The parameters and their defined values are the following:

<handle>
Integer. The socket handle.

<cid>
Integer. The PDN connection ID.

<local_port>
Integer. An unsigned 16-bit integer (0 - 65535) representing the local port the socket is bound to and listening on.

Example

```
AT#XLISTEN?
#XLISTEN: 0,0,1000

OK
```

Test command

The test command tests the existence of the command and provides information about the type of its subparameters.

Syntax

```
AT#XLISTEN=?
```

Response syntax

```
#XLISTEN: <handle>
```

Example

```
AT#XLISTEN=?
#XLISTEN: <handle>
OK
```

3.16.16 Socket accept #XACCEPT

The #XACCEPT command allows you to accept an incoming connection on a listening socket.

This command is used with TCP server sockets that have been put in listening mode with the #XLISTEN command. When a connection is accepted, a new socket is created to handle the connection, and the original listening socket continues to listen for additional connections.

Note:

The listening socket is set to non-blocking mode when the #XLISTEN command is executed. If there are no incoming connections when the #XACCEPT command is executed, it returns an error. You can use the #XAPOLL command to receive asynchronous notifications (POLLIN) for incoming connections on the listening socket.

Set command

The set command allows you to accept an incoming connection on a listening socket.

Syntax

```
AT#XACCEPT=<handle>
```

The parameters and their defined values are the following:

<handle>
Integer. Specifies the socket handle that was used for the AT#XLISTEN command.

Response syntax

```
#XACCEPT: <handle>,<cid>,"<peer_addr>",<peer_port>
```

The parameters and their defined values are the following:

<handle>
Integer. The new socket handle created for the accepted connection. You can use this new socket to send and receive data with the connected client.

<cid>
Integer. The PDN connection ID.

<peer_addr>
String. The IP address of the remote peer that connected. Supports both IPv4 and IPv6 addresses.

<peer_port>
Integer. An unsigned 16-bit integer (0 - 65535) representing the port number of the remote peer.

Example

```
AT#XSOCKET=2,1,0
#XSOCKET: 0,1,6

OK

// Enable asynchronous polling for all the sockets with POLLIN
event.
AT#XAPOLL=,1,1
OK

AT#XBIND=0,1000
OK

AT#XLISTEN=0
OK

// POLLIN for listening socket is received when a client connects.
#XAPOLL: 0,1
```

(continues on next page)

(continued from previous page)

```
// Accept the incoming connection.
AT#XACCEPT=0
#XACCEPT: 1,0,"1111:2222:3333:4444::1",1234

OK

// Once the connection is accepted, data can be received and sent
with the new socket handle.
#XAPOLL: 1,1

AT#XRECV=1,0,0,1
#XRECV: 1,0,17
Hello from client
OK

AT#XSEND=1,0,0,"Hello from server"
#XSEND: 1,0,17

OK

#XAPOLL: 1,1

AT#XRECV=1,0,0,1
#XRECV: 1,0,15
Bye from client
OK

// Client closes the connection. POLLIN event is always received
with POLLHUP, if POLLIN is enabled for polling.
#XAPOLL: 1,17
AT#XRECV=1,0,0,1
OK

AT#XCLOSE=1
#XCLOSE: 1,0

OK

AT#XCLOSE
#XCLOSE: 0,0

OK
```

Read command

The read command is not supported.

Test command

The test command tests the existence of the command and provides information about the type of its subparameters.

Syntax

```
AT#XACCEPT=?
```

Response syntax

```
#XACCEPT: <handle>
```

Example

```
AT#XACCEPT=?
#XACCEPT: <handle>
OK
```

3.17 Trace AT commands

This page describes the AT commands for controlling the shared UART trace backend.

The AT#XLOG command is always available in the default build. The AT#XTRACE command requires building with the `trace-backend-uart.conf` configuration overlay and the `trace-backend-nrf9151.overlay` devicetree overlay. See *Modem traces through UART* for a full description of the feature. The CMUX modem trace backend (`trace-backend-cmux.conf`) does not use this command. See *Modem traces through CMUX*.

The Zephyr application log backend (AT#XLOG) and the modem trace backend (AT#XTRACE) share a single UART. They are mutually exclusive: enabling one while the other is active returns an error. During boot, the UART outputs logs from B0, MCUboot, and the application. After initialization, the UART is suspended when no backend is active and resumed when at least one backend is enabled.

The trace UART is configured at 1000000 baud rate to support the high data rate required for the modem traces. Trace data is available through UART1 (VCOM1 on the nRF9151 DK).

3.17.1 Application log AT#XLOG

The AT#XLOG command enables or disables the Zephyr application log backend and the shared UART.

Note:

Regardless of the logging mode, nRF Cloud Observability collects information, warning, and error level logs (with sensitive payloads redacted) on crash, even when the UART

is silent. You can only view these logs once uploaded to nRF Cloud. See *Observability #XNRFCLOUDOBS**.

Set command

The set command enables or disables the Zephyr application log backend.

Syntax

```
AT#XLOG=<mode>
```

The parameters and their defined values are the following:

<mode>

- `0` - Suspend the UART and disable the application log backend, no logs are shown. Logs are still collected by nRF Cloud Observability.
- `1` - Resume the UART and enable the application log backend. AT commands, responses, and URCs are logged as strings at information log level, with sensitive payloads redacted.
- `2` - Resume the UART and enable the application log backend. AT commands, responses, and URCs are logged as raw hex dumps at debug level only. These hex dumps are not collected by nRF Cloud Observability.

  Requires the `CONFIG_SM_LOG_LEVEL_DBG` Kconfig option.

Note:

Returns `ERROR` if `AT#XTRACE=1` has been issued.

Read command

The read command returns the current state of the application log backend.

Syntax

```
AT#XLOG?
```

Response syntax

```
#XLOG: <mode>
```

The parameters and their defined values are the following:

<mode>
The current state.

- `0` - Disabled.

- 1 - Logging with sensitive AT command payloads redacted.
- 2 - Logging with sensitive AT command and response payloads logged as DBG hex dump only.

  Only available if the `CONFIG_SM_LOG_LEVEL_DBG` Kconfig option is enabled.

Test command

The test command returns the supported parameter range.

Syntax

```
AT#XLOG=?
```

Response syntax

```
#XLOG: (0,1,2)
```

The 2 option is only listed if the `CONFIG_SM_LOG_LEVEL_DBG` Kconfig option is enabled. Otherwise, the response is the following:

```
#XLOG: (0,1)
```

Example

```
AT#XLOG=1
OK

AT#XLOG?
#XLOG: 1
OK

AT#XLOG=0
OK
```

3.17.2 Modem trace AT#XTRACE

The `AT#XTRACE` command enables or disables the modem trace backend on the shared UART.

When enabled, the UART is resumed and the modem is instructed to generate full-level traces (`AT%XMODEMTRACE=1,2`). When disabled, the modem is instructed to stop generating traces (`AT%XMODEMTRACE=0`) and the UART is suspended.

Set command

The set command enables or disables the modem trace backend.

Syntax

```
AT#XTRACE=<mode>
```

The parameters and their defined values are the following:

<mode>

- `0` - Disable modem traces and suspend the UART.
- `1` - Resume the UART and enable the modem trace backend.

Note:

Returns ERROR if `AT#XLOG=1` has been issued.

Read command

The read command returns the current state of the modem trace backend.

Syntax

```
AT#XTRACE?
```

Response syntax

```
#XTRACE: <mode>
```

The parameters and their defined values are the following:

<mode>

The current state.

- `0` - Disabled.
- `1` - Enabled.

Test command

The test command returns the supported parameter range.

Syntax

```
AT#XTRACE=?
```

Response syntax

```
#XTRACE: (0,1)
```

Example

```
AT#XTRACE=1
OK

AT#XTRACE?
#XTRACE: 1
OK

AT#XTRACE=0
OK
```

3.18 Data mode

The Serial Modem can run in the two operation modes defined by the AT command set: AT-command mode and data mode.

When running in data mode, the application does the following:

- It considers all the data received from the MCU over the UART bus as arbitrary data to be streamed through the LTE network by various service modules.
- It buffers the URCs received from modem and threads and sends them to the MCU only after exiting data mode.
- For the socket that is in data mode with automatic data reception, the data streamed from a remote service is considered binary data to be sent to the MCU over the UART.

3.18.1 Overview

You can manually switch between AT-command mode and data mode. However, the Serial Modem data mode is applied automatically when any of the following AT commands are issued:

- Socket `AT#XSEND` and `AT#XSENDTO`
- MQTT publish `AT#XMQTTPUB`
- nRF Cloud send message `AT#XNRFCLOUD`
- DFU write `AT#XDFUWRITE`
- LwM2M carrier library app data send `AT#XCARRIER`

Entering data mode

The Serial Modem enters data mode when an AT command to send data out does not carry the payload. See the following examples:

- `AT#XSEND=0,2,0` makes Serial Modem enter data mode to receive arbitrary data to transmit for socket 0.
- `AT#XSEND=0,2,0,<data_len>` makes Serial Modem enter data mode to receive arbitrary data of the specified length to transmit for socket 0.

- `AT#XSEND=0,0,0,"data"` makes Serial Modem transmit data in normal AT Command mode for socket 0.

Note:

If the data contains either `,` or `"` as characters, it can only be sent in data mode. A typical use case is to send JSON messages.

Other examples:

- `AT#XMQTTPUB=<topic>,"",<qos>,<retain>`
- `AT#XNRFCLOUD=2`
- `AT#XDFUWRITE=0,0,4096`
- `AT#XCARRIER="app_data_set"`

The Serial Modem sends an *OK* response when it successfully enters data mode.

Sending data in data mode

Any arbitrary data received from the MCU is sent to LTE network *as-is*.

Note:

If sending fails while in data mode, the Serial Modem drops any further data received from the UART until data mode is exited, as described in the *Exiting data mode* section. When data mode was entered with `<data_len>`, no termination string is sent. The MCU receives `#XDATAMODE` at the expected byte boundary instead.

When the data mode was entered without `<data_len>`, the Serial Modem also sends the termination string, set by the *CONFIG_SM_DATAMODE_TERMINATOR* Kconfig option, to the MCU to signal the error.

Exiting data mode

To exit the data mode without the specification of `<data_len>`, the MCU sends the termination command set by the *CONFIG_SM_DATAMODE_TERMINATOR* configuration option over UART.

The pattern string could be sent alone or as an affix to the data. The pattern string must be sent in full.

If `<data_len>` is specified in the AT command and the specified data length is reached, the Serial Modem exits data mode. The termination command is not used in this case.

If a send operation fails while in data mode, such as when the socket in data mode encounters an error, the same exit conditions apply. The MCU must send the termination command if `<data_len>` was not specified. If `<data_len>` was specified, the MCU must send the remaining bytes required to reach the specified length.

When exiting the data mode, the Serial Modem sends the `#XDATAMODE` unsolicited notification.

After exiting the data mode, the Serial Modem returns to the AT command mode.

Triggering the transmission

The Serial Modem buffers all the arbitrary data received from the UART bus before initiating the transmission.

The transmission of the buffered data to the LTE network is triggered in the following scenarios:

- Time limit when the defined inactivity timer times out.
- Reception of the termination string.
- Filling of the data mode buffer.

If there is no time limit configured, the minimum required value applies. For more information, see the *Data mode control #XDATACTRL* command.

Flow control in data mode

When Serial Modem fills its UART receive buffers, it disables UART reception. If `hw-flow-control` is enabled for the UART, hardware flow control is imposed. Without hardware flow control, the Serial Modem will drop incoming data while the UART reception is disabled. Serial Modem reenables UART reception when the data has been moved to the data mode buffer. If the data mode buffer fills, the data are transmitted to the LTE network.

Note:

There is no unsolicited notification defined for this event. UART hardware flow control is responsible for imposing and revoking flow control.

Note:

The whole buffer is sent in a single operation. When transmitting UDP packets, only one complete packet must reside in the data mode buffer at any time.

The data mode buffer size is controlled by *CONFIG_SM_DATAMODE_BUF_SIZE*.

3.18.2 Data mode control #XDATACTRL

The #XDATACTRL command allows you to configure the time limit used to trigger data transmissions. It can be applied only after entering data mode.

When the time limit is configured, small-size packets will be sent only after the timeout.

Set command

The set command allows you to configure the time limit for the data mode.

Syntax

```
AT#XDATACTRL=<time_limit>
```

The parameters and their defined values are the following:

<time_limit>
: Integer. The timeout value in milliseconds. The default value is the minimum required value, based on the configured UART baud rate.

  This value must be long enough to allow for a DMA transmission of an UART receive (RX) buffer (*CONFIG_SM_UART_RX_BUF_SIZE*).

Read command

The read command allows you to check the current time limit configuration and the minimum value required, based on the configured UART baud rate.

Syntax

```
AT#XDATACTRL?
```

Response syntax

```
#XDATACTRL: <current_time_limit>,<minimal_time_limit>
```

Test command

The test command tests the existence of the command and provides information about the type of its subparameters.

Syntax

```
AT#XDATACTRL=?
```

Response syntax

```
#XDATACTRL=<time_limit>
```

3.18.3 Exit data mode notification #XDATAMODE

When the application receives the termination command *CONFIG_SM_DATAMODE_TERMINATOR* in data mode, it sends the `#XDATAMODE` unsolicited notification.

Unsolicited notification

```
#XDATAMODE: <status>
```

The parameters and their defined values are the following:

<status>

- `0` - Success.
- `-1` - Failure.

Example

```
AT#XSEND=0,2,0

OK
Test datamode
+++
#XDATAMODE: 0
```

3.18.4 Configuration options

Check and configure the following configuration options for data mode:

CONFIG_SM_DATAMODE_TERMINATOR - Pattern string to terminate data mode
This option specifies a pattern string to terminate data mode. The default pattern string is +++.

CONFIG_SM_DATAMODE_BUF_SIZE - Buffer size for data mode
This option defines the buffer size for the data mode. The default value is 4096.

4 UART configuration

This section describes the UART configuration for the Serial Modem application.

4.1 UART interface

A typical Serial Modem application configuration consists of a host processor connected to an nRF91 Series SiP through a UART interface. The UART connection serves as the physical transport layer, carrying both AT commands and PPP data frames for network communication.

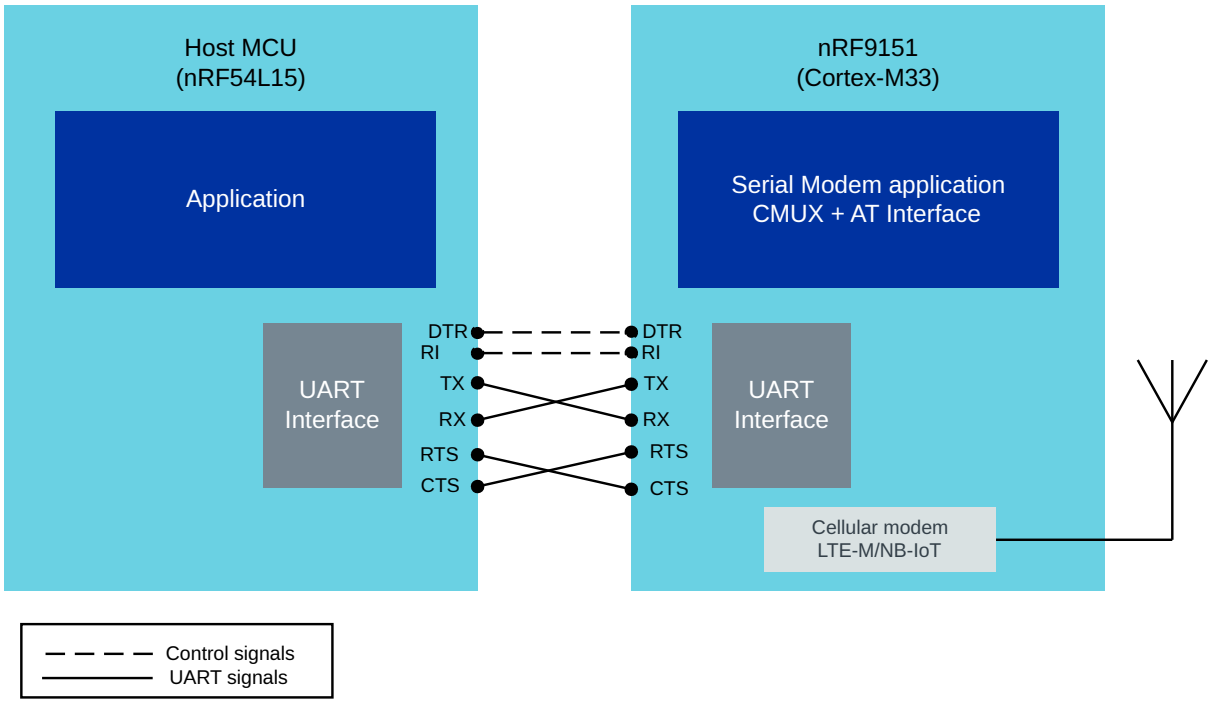


Fig. 1: UART interface setup

The following UART signals are used:

Signal	Direction	Purpose
TX/RX	Bidirectional (crossed)	Data transmission
RTS/CTS	Bidirectional (crossed)	Hardware flow control
DTR	Host Modem	*Data Terminal Ready* - Host signals UART is powered on
RI	Modem Host	*Ring Indicator* - Modem wakes host for incoming data

The Data Terminal Ready (DTR) and Ring Indicator (RI) signals are the **control signals** used for UART power management, while TX/RX/RTS/CTS are the standard UART signals.

Important:

The use of Serial Modem without hardware flow control (RTS/CTS) is experimental and strongly not recommended. See *Operating without hardware flow control* for more information.

4.2 Board-specific pin mapping

The UART configuration varies depending on the board and whether it is running the Serial Modem application (nRF9151) or a host application (for example, nRF54L15).

UART configuration is done in the devicetree as a build time configuration.

4.2.1 Serial Modem application (nRF9151)

The following tables shows how to connect the UART pins to the corresponding pins on the nRF91 Series SiP:

USB UART with PC host

UART Signal	nRF9151 Pin
TX	P0.27
RX	P0.26 (pull-up)
RTS	P0.14
CTS	P0.15 (pull-up)
DTR	P0.08 (Button 1, pull-up, active high)
RI	P0.00 (LED1)

- UART instance: UART0 (VCOM0 on the interface chip)
- Baud rate: 115200
- Hardware flow control: Enabled

The DTR pin defaults to pull-up with an active high level, so the UART is always powered on. This pin is shared with **Button 1**, which you can use to manually toggle the DTR state. When the button is pressed, DTR is pulled low (inactive), powering down the UART.

The RI pin is connected to **LED1**, as it is not routed to the host PC.

This setup is default when building for nRF9151 DK without any overlay files.

external MCU as host

UART Signal	nRF9151 Pin
TX	P0.02
RX	P0.03 (pull-up)
RTS	P0.06
CTS	P0.07 (pull-up)
DTR	P0.31 (active low, pull-up)
RI	P0.30 (active low)

- UART instance: UART2
- Baud rate: 115200
- Hardware flow control: Enabled

On the nrf9151dk board, UART data and flow control pins (TX, RX, RTS, CTS) are on the **P4** connector, while control signals (DTR, RI) are on the **P3** connector.

The DTR pin defaults to pull-up, so the external MCU must assert DTR to power on the UART. When UART power management is not used, the DTR pin should be pulled to active state.

When hardware flow control is not used, remove `hw-flow-control;` from the `uart2` node and remove the `bias-pull-up` from the CTS pin in the overlay. With hardware flow control disabled, the UART peripheral does not monitor the CTS pin, so Serial Modem transmits unconditionally. The RTS and CTS pins can be left unconnected. See *Operating without hardware flow control* for further considerations.

This setup is provided in the `app/external-mcu.overlay` overlay file.

Note:

DTR and RI pins use the **P0.31** and **P0.30** pins, which are shared with the I2C2 instance on the nrf9151dk board. I2C2 must stay disabled to use these pins for DTR and RI signals.

Thingy:91 X with PC host

UART Signal	nRF9151 Pin
TX	P0.00
RX	P0.01
DTR	P0.26 (Button 0, pull-up, active high)
RI	P0.30 (blue LED)

- UART instance: UART0 (VCOM0 on the interface chip)
- Baud rate: 115200
- Hardware flow control: Disabled

Hardware flow control is disabled, as there are no RTS and CTS pins on the Thingy:91 X interface chip.

The DTR pin defaults to pull-up with the active high level, so the UART is always powered on. This pin is shared with **Button 0**, which you can use to manually toggle the DTR state. When the button is pressed, DTR is pulled low (inactive), powering down the UART.

The RI pin is connected to the blue LED, which is used to indicate incoming data.

4.2.2 Power and reset pins

The following pins supply, power on, and reset the nRF9151 SiP:

Signal	LGA pin
ENABLE	10 (active high)
nRESET	9 (active low, external pull-up not allowed)
VDD	14
VDD_GPIO	65

The Serial Modem firmware has no watchdog. The host must therefore be able to reset the nRF9151 SiP to a known state, either by power cycling it with **ENABLE** or by using **nRESET**.

Note:

The pins listed here are not the only pins that can be useful with the nRF9151 SiP. For the complete pin list, see the nRF9151 LGA pin assignments. For hardware integration guidance, see the nRF9151 Hardware Integration page.

4.2.3 nRF91M1 pre-programmed Serial Modem application

nRF91M1 is a pre-programmed nRF9151 SiP that runs the Serial Modem application out of the box.

For the UART configuration of the nRF91M1, see the nRF91M1 UART configuration appendix in the nRF91M1 AT Commands documentation.

This setup is provided in the `app/nrf91m1.overlay` devicetree overlay file.

4.2.4 Interposer boards

An interposer board adapts the nRF9151 SiP to the footprint of another cellular module. The interposer wiring determines which nRF9151 pins carry the UART signals, so each interposer needs its own devicetree overlay.

Both overlays target the nrf9151dk board. Each one assigns its own pinctrl states and points `uart0` at them, so the pin mapping does not depend on the states the board files happen to define.

nRF9151-MK-BG95 interposer

UART Signal	nRF9151 Pin
TX	P0.15
RX	P0.14 (pull-up)
RTS	P0.18
CTS	P0.19 (pull-up)
DTR	P0.08 (Button 1, pull-up, active high)
RI	P0.00 (LED1)

- UART instance: UART0
- Baud rate: 115200
- Hardware flow control: Disabled

The overlay remaps the `uart0` pins to match the interposer footprint and removes `hw-flow-control`. DTR and RI keep the nrf9151dk board defaults.

The RTS and CTS pins are still assigned in the pinctrl states but are not used for flow control, and both can be left unconnected. See *Operating without hardware flow control* for further considerations.

This setup is provided in the `app/interposer-nrf9151-mk-bg95.overlay` devicetree overlay file.

nRF9151-MK-GM02S interposer

UART Signal	nRF9151 Pin
TX	P0.08
RX	P0.07 (pull-up)
RTS	P0.05
CTS	P0.04 (pull-up)
DTR	P0.15 (pull-up, active high)
RI	P0.00 (LED1)

- UART instance: UART0
- Baud rate: 115200
- Hardware flow control: Enabled

The interposer routes UART0 TX to **P0.08**, which the nrf9151dk board overlay uses for DTR. The overlay therefore moves DTR to **P0.15**, which UART0 CTS no longer occupies.

Note:

RI, together with the UART0 pins, overlaps **LED 1** (**P0.00**), **LED 3** (**P0.04**), **LED 4** (**P0.05**) and **Button 1** (**P0.08**) on the nrf9151dk board. The overlay disables the LED, PWM LED and button nodes so that nothing else can drive those pins.

This setup is provided in the `app/interposer-nrf9151-mk-gm02s.overlay` devicetree overlay file.

4.2.5 Host application

PC serial terminal

When using a PC serial terminal application (for example, PuTTY, Tera Term, minicom), build the Serial Modem application with the PC host. This is the default configuration when building for nRF9151 DK:

- UART instance: VCOM0
- Baud rate: 115200
- Hardware flow control: RTS/CTS Enabled
- Line terminator: CRLF
- AT command terminator: CR

When configuring the serial terminal application, ensure that return key sends **CR** (Carriage Return) character as the line terminator. For incoming serial output from the modem, both **CR** and **LF** (Line Feed) characters are used. This is the most common configuration for AT command terminals.

This setup is the default for Tera Tearm, Putty, Minicom, and Picocom serial terminal applications.

nRF54L15DK

The nRF54L15DK host applications use UART30 for Serial Modem communication:

UART Signal	nRF54L15 Pin
TX	P0.00
RX	P0.01
RTS	P0.02
CTS	P0.03
DTR	P1.11
RI	P1.12

UART0 on the P0 connector is the `uart30` instance in the devicetree and is routed to VCOM0 on the interface chip. This must be disabled in the Board Configurator app to release the UART pins for external use.

UART1 is routed to VCOM1 on the interface chip. The VCOM1 is used for console/shell.

RI and DTR signals are located on the P1 connector.

Note:

Disable VCOM0 from Board Configurator app to release UART30 for Serial Modem.

Native simulator

For native simulator testing, UART2 is configured to use a host serial port:

```
uart2: uart2 {
    status = "okay";
    compatible = "zephyr,native-tty-uart";
    serial-port = "/dev/ttyACM0";
    current-speed = <115200>;
    hw-flow-control;

    modem: modem {
        compatible = "nordic,nrf91-slm";
        status = "okay";
        zephyr,pm-device-runtime-auto;
    };
};
```

Build the Serial Modem application with the PC host. The UART0/VCOM0 from Serial Modem is assumed to be connected to `/dev/ttyACM0` on the Linux host.

See the *PPP shell sample* for more information.

4.3 Controlling the UART power state

The UART power state of the Serial Modem is controlled by the **DTR** signal. When DTR is asserted by the host, the UART is enabled and ready for communication. When DTR is deasserted, the UART is powered down to save power.

When the UART is powered down, it releases its high-speed clocks, which are then automatically stopped by the system's power management. This achieves the lowest possible power consumption, as the UART is one of the primary power consumers in Serial Modem. Power consumption can be reduced from hundreds of microamps to just a few microamps when the UART is powered down.

UART power management is completely independent of the modem power management (CFUN, PSM, eDRX, and so on). By actively managing the UART power state through DTR control, the nRF9151 SiP achieves ultra-low power consumption of approximately 2 µA when the cellular modem enters Power Saving Mode (PSM) and the UART is powered down. This is a reduction of over 99% compared to the 400-450 µA consumed when the UART remains active during idle periods. The nRF9151 SiP retains all the RAM content while in any of the power saving modes, so the application can resume from where it left off when the UART is powered up. Full network context is maintained, including socket connections, TLS/DTLS sessions, and so on. LTE connection is automatically resumed when there is outgoing data to be sent. No other GPIO control is required to utilize the PSM or eDRX modes.

4.3.1 Incoming data wake-up

When the UART is powered down and there is incoming data, the Serial Modem asserts the **RI** pin to notify the host. The host must handle the RI signal by powering up its UART device and asserting the DTR line, signaling to the Serial Modem that the UART is powered on and ready to receive data. The RI signal remains asserted until the UART is activated and ready to transmit data.

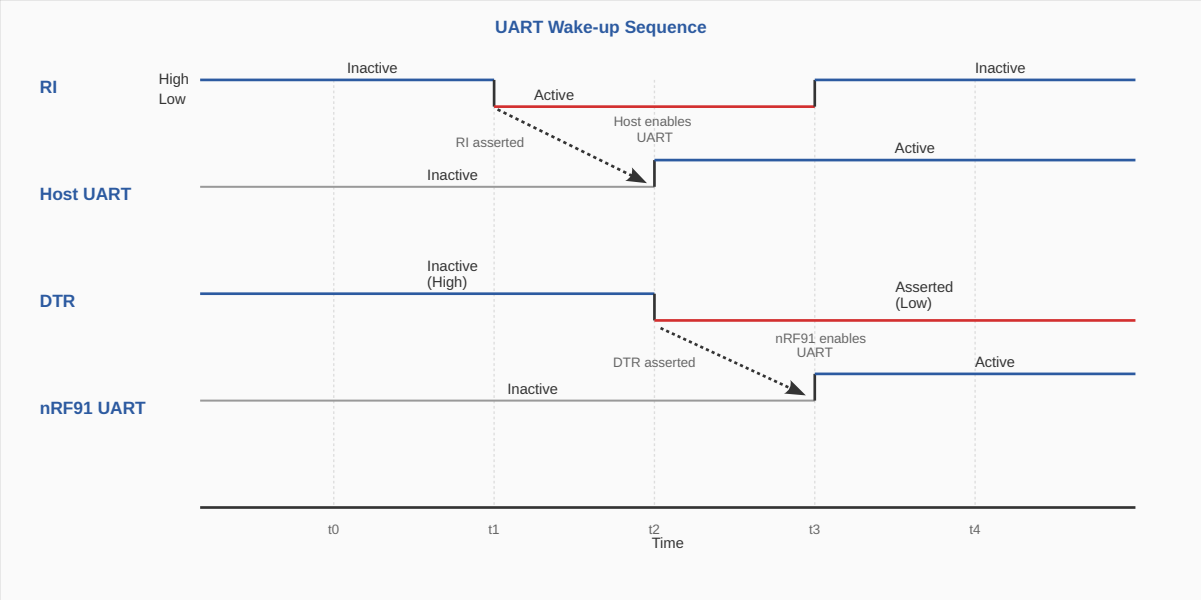


Fig. 2: Sequence diagram showing the wake-up sequence when the UART is powered down

This sequence diagram illustrates the wake-up sequence when the UART is powered down. The RI signal triggers the host to power up its UART and assert DTR. Once the host UART is ready and DTR is asserted, the Serial Modem enables its UART interface. The RI signal remains active throughout this process and only returns to inactive once the Serial Modem UART is active. This coordinated wake-up ensures both ends of the communication link are ready before data transmission begins.

With hardware flow control - The host can observe its CTS pin (connected to Serial Modem's RTS) to know when Serial Modem is ready to receive data. When Serial Modem starts driving its RTS pin, the host knows that Serial Modem is ready to receive. The host must also have a pull-up on its CTS pin.

Without hardware flow control- The host must monitor the RI signal, which is de-asserted when the Serial Modem UART becomes ready to receive data. Alternatively, the host can implement a sufficient timeout between asserting DTR and initiating its first TX operation. See *Operating without hardware flow control* for how to configure Serial Modem to transmit without hardware flow control.

4.3.2 Host-initiated wake-up

When the host processor needs to send data, it first powers up the UART device, then asserts the DTR signal to indicate that the UART is active. The modem detects the DTR assertion and powers up its UART interface, allowing communication to resume.

4.3.3 Boot-up scenario

To boot-up the Serial Modem, the host asserts **ENABLE**, **DTR**, and enables its UART (driving Serial Modem **CTS** low). The Serial Modem firmware initializes within approximately 2000 ms, during which it activates its UART and asserts **RTS** to signal readiness. Serial Modem then transmits a `"Ready"` message on **TX**, confirming that the link is operational.

The **ENABLE** pin (pin **10** on the nRF9151 SiP) is a high-impedance control input for the internal power management unit (PMU) of the nRF9151 SiP. Driving it to a logic high powers

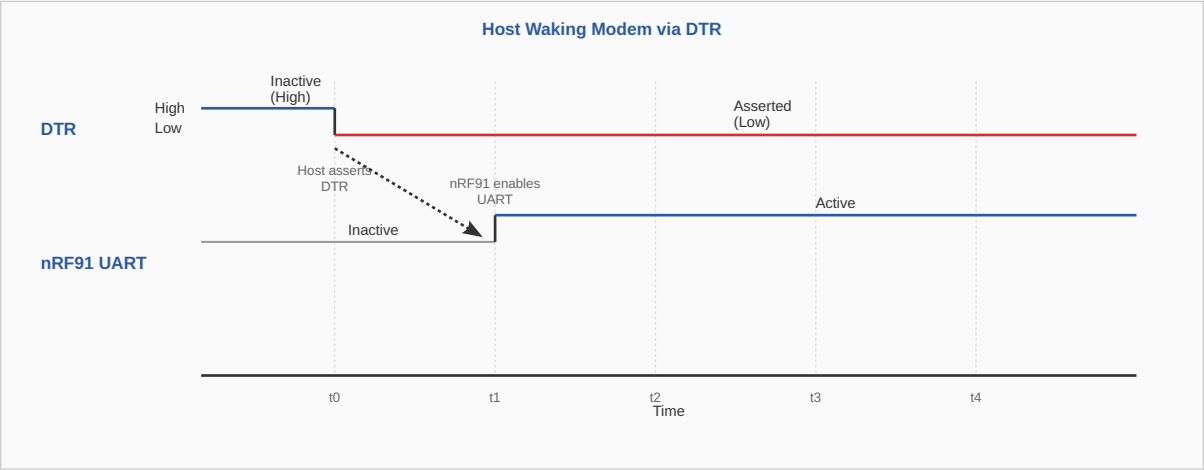


Fig. 3: Sequence diagram showing host-initiated wake-up of the modem

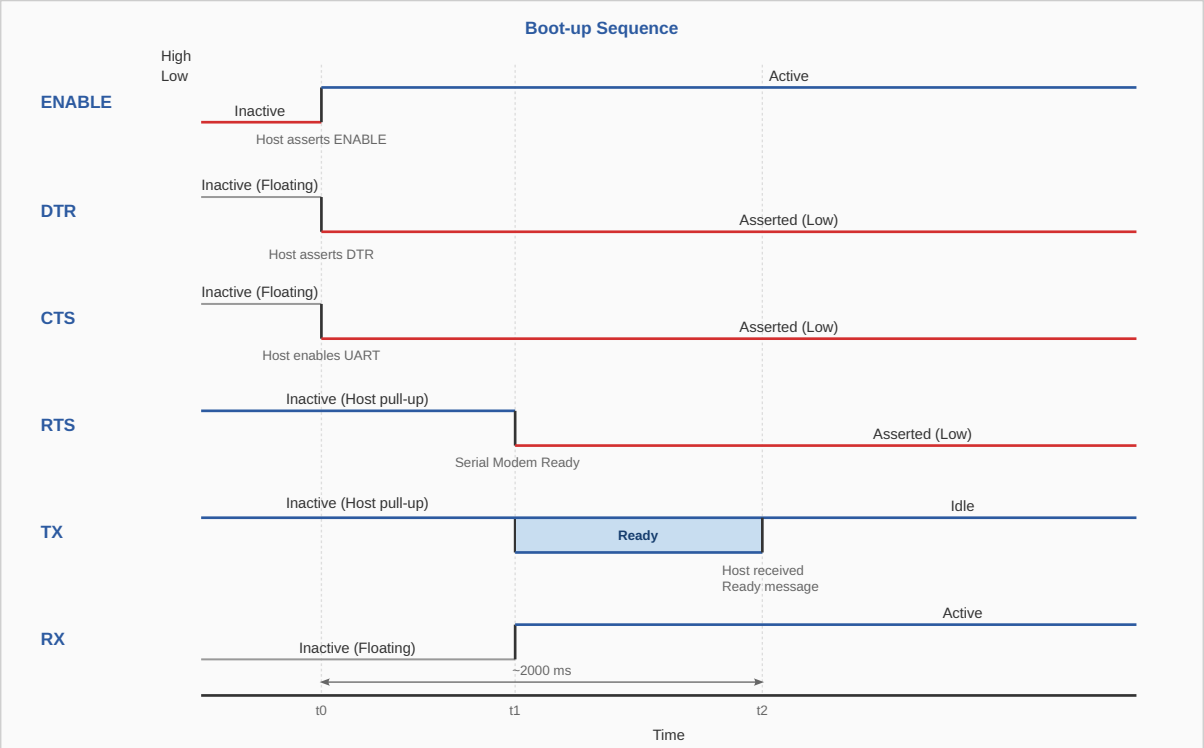


Fig. 4: Signal sequence during boot-up

on the nRF9151 SiP. Driving it to a logic low brings the device into an extremely low-current state. Any internal state not retained in non-volatile memory is lost, and a full boot-up sequence is required when **ENABLE** is reasserted. See nRF9151 Hardware Design Guidelines — ENABLE for more information about the ENABLE pin.

4.3.4 Reset scenario

To reset the nRF9151 SiP to a known state, do one of the following:

- Drive **nRESET** low for at least 5 µs.
- Power cycle the nRF9151 SiP with **ENABLE**. Keep **ENABLE** low for at least 1 s.

After the reset, the startup sequence is the same as in the *Boot-up scenario*.

4.3.5 Operating without hardware flow control

Important:

The use of Serial Modem without hardware flow control (RTS/CTS) is experimental and strongly not recommended. Hardware flow control ensures reliable communication and prevents timing-related issues. Without hardware flow control, there is a risk of data loss in UART communication, especially in high-throughput scenarios.

The approach for a custom Serial Modem build is the following:

- **Remove** `hw-flow-control;` **from the UART node in the overlay** — With hardware flow control disabled, the UART peripheral does not monitor CTS and Serial Modem transmits unconditionally.
- **Remove the** `bias-pull-up` **from the CTS pinctrl group in the overlay** — This avoids a pull resistor drawing current against any external signal.
- **Leave RTS and CTS unconnected** — Neither pin is used and both can be left floating.

Key requirements:

- The host must enable its own UART before asserting DTR.
- To detect when Serial Modem is ready to receive, monitor the RI signal (de-asserted when ready) or use a sufficient timeout after DTR assertion.
- See *Incoming data wake-up* for detailed timing information.
- In boot-up scenarios, the host must wait for the `"Ready"` message from Serial Modem before sending any data.

Buffer configuration:

Increase *CONFIG_SM_UART_RX_BUF_SIZE* and possibly *CONFIG_SM_UART_RX_BUF_COUNT* to provide sufficient buffer space for your use case.

4.3.6 Automatic UART power management using Zephyr's Cellular Modem driver

As the Serial Modem's UART interface power state is controlled by the DTR signal from the host, it is possible to automate the power management using Zephyr's cellular modem driver with Zephyr's CMUX (3GPP TS 27.010) module. This allows host applications to manage the UART power state automatically. See the Zephyr CMUX Power Saving documentation for more information. This configuration is completely done in the host. Serial Modem does not require any changes as it already follows the DTR signal state for UART power management.

You can configure the Zephyr's cellular modem driver to automatically power on and off the UART device based on the data traffic on the CMUX DLCI channels. This is done by setting the `cmux-enable-runtime-power-save` and `cmux-close-pipe-on-power-save` configuration options. When PPP is connected, the UART is automatically powered down when the PPP connection is idle for a certain period of time (configurable with the `cmux-idle-timeout-ms` configuration option).

The following example shows how to configure automatic UART power management for the nRF54L15DK host application:

```
/* Zephyr modem UART <-> nRF91 Serial Modem UART */
&uart30 {
    status = "okay";
    current-speed = <115200>;
    hw-flow-control;
    zephyr,pm-device-runtime-auto;

    /* Nordic nRF91 Serial Modem configuration */
    modem: modem {
        compatible = "nordic,nrf91-slm";
        status = "okay";
        mdm-ring-gpios = <&gpio1 12 (GPIO_ACTIVE_LOW | GPIO_PULL_
UP)>;
        mdm-dtr-gpios = <&gpio1 11 GPIO_ACTIVE_LOW>;
        zephyr,pm-device-runtime-auto;
        cmux-enable-runtime-power-save;
        cmux-close-pipe-on-power-save;
        cmux-idle-timeout-ms = <5000>;
    };
};
```

Key devicetree options:

- `zephyr,pm-device-runtime-auto`: Enables automatic runtime power management for the UART device
- `cmux-enable-runtime-power-save`: Enables CMUX power saving mechanism
- `cmux-close-pipe-on-power-save`: Closes the pipe when entering power save mode
- `cmux-idle-timeout-ms`: Timeout in milliseconds before entering idle state (5000 ms in this example)

The setup above also required the `CONFIG_PM_DEVICE_RUNTIME` and `CONFIG_PM_DEVICE_RUNTIME_ASYNC` Kconfig options to be enabled.

When the cellular modem driver enters the idle state and there is no data to be sent or received on any DLCI channel, the UART device is powered down. When the cellular modem driver exits the idle state and there is data to be sent or received on any DLCI channel, the UART device is powered up.

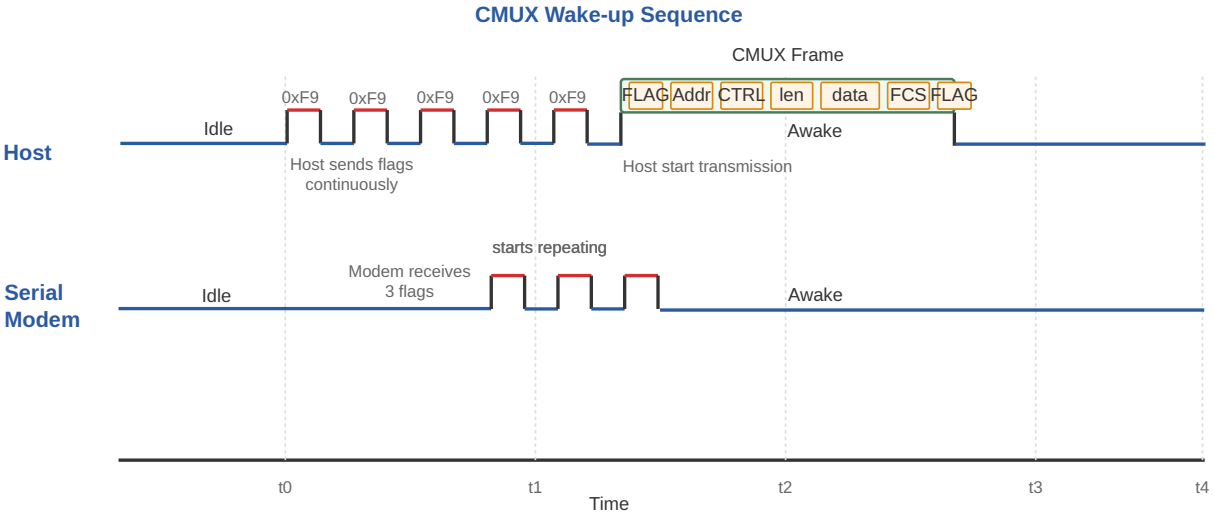


Fig. 5: CMUX wake-up sequence showing flag character exchange

The CMUX wake-up sequence consists of the peer repeatedly sending flag characters (`0xF9`) on the UART. The remote peer, upon detecting these flags, responds with its own flag character to indicate it is awake and ready to receive data. This handshake ensures both ends of the communication link are synchronized before data transmission resumes. As the flag characters carry no information, there is no issue of losing data when waking up. When the peer that initiates the wake-up receives a flag character, it can continue by sending a valid frame. Either end of the CMUX channel can initiate the wake-up procedure.

See the *PPP shell sample* for an example of how to configure the CMUX module to automatically power on and off the UART device based on the data traffic on the DLCI channels.

4.4 UART security considerations

UART is a plain serial link with no built-in encryption or authentication, so data that the host and the Serial Modem exchange over the UART is therefore not protected. You can reduce the exposure by keeping the UART lines hidden inside the physical security boundary of your product. When you need stronger confidentiality, terminate secure protocols on the host instead of in the modem.

The following data is sent over the UART in cleartext:

- Application data sent and received over modem-terminated TLS or DTLS sockets.
- AT commands, responses, and notifications exchanged with the Serial Modem that may contain sensitive information.

Choose one of the following security models based on the physical security of your product.

Aspect	Model A: Trusted UART	Model B: Untrusted UART
When to use	The UART lines are inside the physical security boundary of the product.	The UART lines are accessible outside the physical security boundary of the product, or the product requires stronger confidentiality.
Credential storage	Credentials are stored in the modem with `AT%CMNG`. Provision them in a controlled environment, such as the production line.	Credentials are stored on the host. No credentials for internet services are written to the modem.
Secure channel	Secure protocols, such as TLS, DTLS, MQTT, and HTTP, are terminated in the modem and controlled with AT commands.	All internet protocols, such as TLS, DTLS, MQTT, HTTP, and CoAP, are terminated on the host. The host uses *PPP* over *CMUX* and the modem only forwards IP packets.
Data on the UART	Application data and AT commands are in cleartext.	Application data is encrypted by the host. AT commands are still in cleartext. Commands that configure the modem, such as the SIM PIN, PDN authentication credentials, and SMS content, can carry sensitive data. Apply these settings in a controlled environment where possible. Commands that are needed in the field remain visible on the UART.

Note:

Firmware integrity does not depend on the UART. Firmware updates are signature-verified on the device, regardless of the security model.

5 Host

The Serial Modem includes a library and sample applications that demonstrate how to interact with the *Serial Modem application* running on an nRF91 Series device. These samples showcase different integration approaches for controlling the nRF91 Series SiP as a cellular modem.

Serial Modem host sample setup

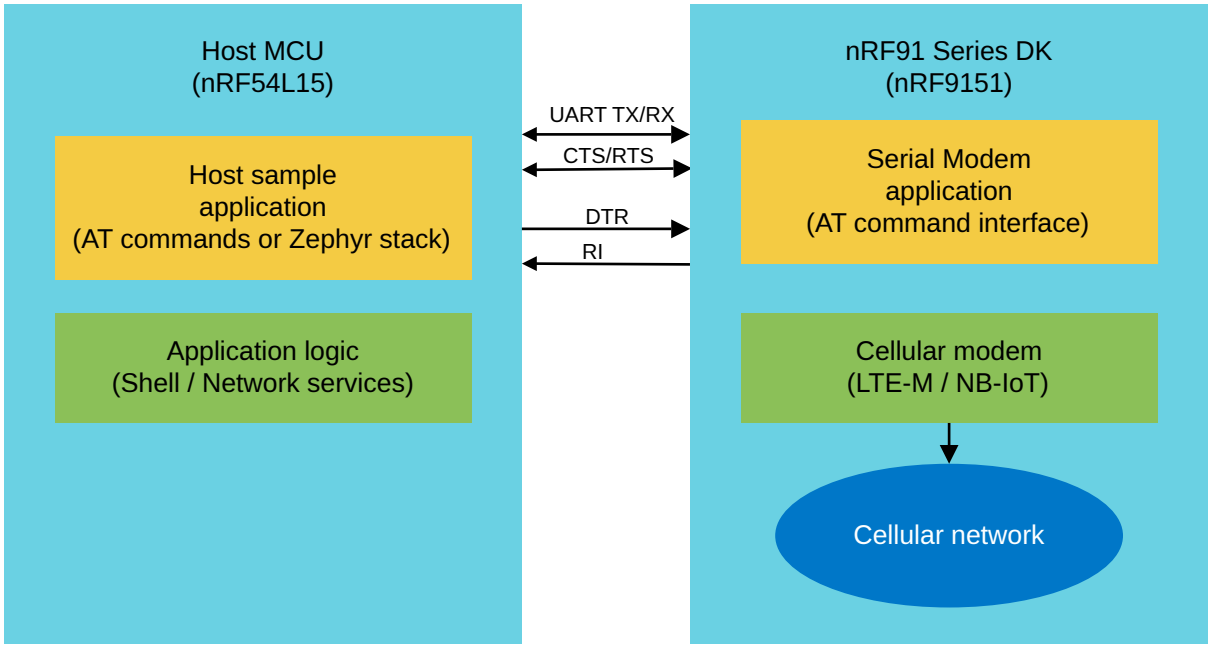


Fig. 1: Serial Modem running on nRF91 Series DK with a host sample application on an external MCU

In addition, the Serial Modem includes the *Serial Modem AT Client library*, which exposes the AT command interface of the *Serial Modem application* for external devices over a serial interface.

Use the *AT Client Shell sample* for one or more of the following use cases:

- You need direct AT command control.
- Your application uses offloaded sockets.
- You want to use the Serial Modem Host library.
- You are working with resource-constrained MCUs.
- You need to send both standard and proprietary AT commands.

Use the *PPP shell sample* for one or more of the following use cases:

- You want to use Zephyr's native networking stack.
- You prefer standard POSIX socket APIs.
- You need to test network performance.
- You are developing on Linux with the native simulator.
- You want seamless integration with Zephyr network services.

Both samples require the Serial Modem application to be running on an nRF91 Series device.

5.1 AT Client Shell sample

The Serial Modem AT Client Shell sample demonstrates how to send AT commands to the modem through the *Serial Modem application* running on an nRF91 Series SiP. This sample enables an external MCU to send modem and Serial Modem proprietary AT commands for LTE connection and IP services. See more information on the functionality of this sample from the *Serial Modem AT Client library*, which provides the core functionality for this sample.

5.1.1 Requirements

The Serial Modem application should be configured with `app/external-mcu.overlay` on the nRF91 Series DK side.

The sample supports the following development kit:

Hardware platforms	PCA	Board name	Board target
nRF54L15 DK	PCA10156	nrf54l15dk	`nrf54l15dk/nrf54l15/cpuapp`

Connect the DK pins that are defined in the board-specific overlay files `samples/sm_at_client_shell/boards/*.overlay` to the corresponding pins in the nRF91 Series DK, which are defined in `app/external-mcu.overlay`.

The following table shows how to connect UART, DTR and RI pins of the DK to the corresponding pins in the nRF91 Series DKs:

nRF54L15 DK	nRF91 Series DK
UART TX P0.00	UART RX P0.03
UART RX P0.01	UART TX P0.02
UART CTS P0.03	UART RTS P0.06
UART RTS P0.02	UART CTS P0.07
DTR OUT P1.11	DTR IN P0.31
RI IN P1.12	RI OUT P0.30
GPIO GND	GPIO GND

Note:

You must disable the VCOM0 on the nRF54L15 DK to release the UART GPIO pins to use it with the *AT Client Shell sample*.

- For nRF54L15 DK, you can use the Board Configurator app to disable the `Connect port VCOM0` setting.

Note:

The GPIO output levels on the nRF91 Series device and nRF54L15 DK must be the same.

- You can set the VDD voltages for both devices with the Board Configurator app.

5.1.2 References

- nRF91x1 AT Commands Reference Guide
- *Serial Modem AT commands*

5.1.3 Dependencies

This sample uses the following nRF Connect SDK libraries:

- *Serial Modem AT Client library*

5.2 PPP shell sample

The Serial Modem PPP shell sample demonstrates how to use the nRF91 Series SiP as a Zephyr-compatible cellular modem through a Point-to-Point Protocol (PPP) interface. This sample enables an external MCU or Linux host to use Zephyr's native networking stack and shell commands to control the cellular modem, test network connectivity, and measure network performance.

5.2.1 Requirements

The Serial Modem application must be built with the following overlays:

- `app/cmux.conf` - To enable CMUX support
- `app/ppp.conf` - To enable PPP support

If using an external MCU as the controlling device, also include:

- `app/external-mcu.overlay` - To configure UART pins, DTR, and RI signals

The sample supports the following development kits:

Hardware platforms	PCA	Board name	Board target
nRF54L15 DK	PCA10156	nrf54l15dk	`nrf54l15dk/nrf54l15/cpuapp`
Native simulator	N/A	`native_sim`	`native_sim`

Hardware setup for nRF54L15 DK

When using the nRF54L15 DK as the controlling device, connect the DK pins to the corresponding pins on the nRF91 Series DK as follows:

nRF54L15 DK	nRF91 Series DK
UART TX P0.00	UART RX P0.03
UART RX P0.01	UART TX P0.02
UART CTS P0.03	UART RTS P0.06
UART RTS P0.02	UART CTS P0.07
DTR OUT P1.11	DTR IN P0.31
RI IN P1.12	RI OUT P0.30
GPIO GND	GPIO GND

Note:

You must disable the VCOM0 on the nRF54L15 DK to release the UART GPIO pins for communication with the Serial Modem.

- Use the Board Configurator app to disable the `Connect port VCOM0` setting.

Note:

The GPIO output levels on the nRF91 Series device and nRF54L15 DK must be the same.

- Set the VDD voltages for both devices using the Board Configurator app.

Native simulator setup

When running on the native simulator, the sample expects the Serial Modem to be available at `/dev/ttyACM0` on the Linux host system. Connect the nRF91 Series DK running Serial Modem to your Linux host through USB.

5.2.2 Overview

This sample showcases the integration between Zephyr's cellular modem driver and the Serial Modem application running on an nRF91 Series SiP. Instead of using AT commands directly, the controlling device can use Zephyr's standard networking APIs and shell commands to interact with the modem.

The sample provides:

- Full Zephyr networking stack (IPv4 and IPv6 support).
- Network shell commands for interface management.
- Network performance testing using zperf.
- DNS resolution support.
- Automatic modem connection management.
- CMUX (GSM 07.10) multiplexing support with user pipes for AT command access.
- Power management with runtime power saving.

For more information about using the nRF91 Series SiP as a Zephyr-compatible modem, including how to use user pipes for AT commands alongside PPP, see *Cellular PPP modem*.

5.2.3 Configuration

The sample is configured with the following key features:

- PPP Layer 2 networking
- IPv4 and IPv6 support
- TCP and UDP protocols
- DNS resolver with automatic DNS server configuration from PPP
- Network connection manager for automatic reconnection
- Zephyr cellular modem driver with CMUX support
- Runtime power management with automatic power saving
- Network shell commands
- Zperf for network performance testing

The CMUX configuration enables:

- Power save mode that closes CMUX pipes when idle
- 5 second idle timeout before entering power save
- MTU size of 127 bytes for optimal performance
- User pipes on DLC channels 3 and 4 for AT command access (available when the network interface is up)

5.2.4 Building and running

This sample can be found under `samples/sm_ppp_shell`.

nRF54L15 DK

To build the sample for the nRF54L15 DK:

```
west build -b nrf54l15dk/nrf54l15/cpuapp
```

Flash the sample to your board and open a serial terminal to view the shell interface.

For nRF54L15 DK, the shell is available on UART1 (VCOM1).

Native simulator

To build the sample for the native simulator:

```
west build -b native_sim
```

For native simulator, run the executable with the `-attach_uart` option:

```
build/zephyr/zephyr.exe -attach_uart
```

5.2.5 Testing

After the sample starts, you will see the Zephyr shell prompt. The PPP network interface is configured to start the modem automatically when requested.

Bringing up the network interface

To start the cellular modem and establish a network connection:

```
uart:~$ net iface up 1
```

The modem begins the network registration process. You can monitor the connection status with:

```
uart:~$ net iface
```

The network management event monitor (enabled by default) will also display connection events automatically.

Once connected, the interface will show as “UP” and you will see an assigned IP address.

Testing network connectivity

After the network interface is up, you can test basic connectivity.

To ping a (DNS) server:

```
uart:~$ net ping 8.8.8.8
```

To perform DNS resolution:

```
uart:~$ net dns google.com
```

Performance testing with zperf

The sample includes the zperf tool for network performance measurements:

To run a UDP upload test:

```
uart:~$ zperf udp upload <server-ip> <port> <duration> <packet-size>
<rate>
```

See the zperf: Network Traffic Generator for more details on using zperf.

Send AT commands through user pipes

When the network interface is up, you can access the modem's AT command interface through the user pipes on CMUX DLC channels 3 and 4. By default, the shell sample is configured to use the user pipe on DLC channel 3.

To send an AT command:

```
uart:~$ modem at AT+CFUN=4
EVENT: L3 [1] IPv4 address del 100.92.136.41
EVENT: L4 [1] disconnected
EVENT: L4 [1] IPv4 connectivity lost
EVENT: L2 [1] interface down
OK
[00:14:52.706,554] <dbg> modem_cellular: modem_cellular_log_state_
changed: switch from registered to await PPP dead
[00:14:52.706,738] <dbg> modem_cellular: modem_cellular_log_event:
event ppp dead
[00:14:53.095,303] <dbg> modem_cellular: modem_cellular_log_event:
event deregistered
[00:14:54.706,661] <dbg> modem_cellular: modem_cellular_log_event:
event timeout
[00:14:54.706,720] <dbg> modem_cellular: modem_cellular_log_state_
changed: switch from await PPP dead to await registered
[00:14:56.706,792] <dbg> modem_cellular: modem_cellular_log_event:
event timeout
uart:~$ modem at AT+CFUN?
+CFUN: 4
OK
uart:~$ modem at AT+CFUN=1
OK
[00:15:16.308,209] <dbg> modem_cellular: modem_cellular_log_event:
event deregistered
EVENT: L2 [1] interface up
[00:15:18.072,762] <dbg> modem_cellular: modem_cellular_log_event:
event registered
[00:15:18.072,803] <dbg> modem_cellular: modem_cellular_log_state_
changed: switch from await registered to run dial script
[00:15:18.072,861] <dbg> modem_cellular: modem_cellular_log_event:
```

(continues on next page)

(continued from previous page)

```
event timeout
[00:15:18.100,659] <dbg> modem_cellular: modem_cellular_log_event:
event script success
[00:15:18.100,740] <dbg> modem_cellular: modem_cellular_log_state_
changed: switch from run dial script to registered
EVENT: L3 [1] IPv6 address add fe80::3330:3539:3831:3936
EVENT: L3 [1] IPv6 neighbor add fe80::5eff:fe00:533a
EVENT: L3 [1] IPv4 address add 100.92.136.41
EVENT: L4 [1] connected
EVENT: L4 [1] IPv4 connectivity available
uart:~$
```

Shutting down the network interface

To shut down the cellular connection and power off the modem:

```
uart:~$ net iface down 1
```

5.2.6 Dependencies

The sample enables the following key features:

- Networking stack (`CONFIG_NETWORKING`)
- PPP L2 layer (`CONFIG_NET_L2_PPP`)
- Cellular modem driver (`CONFIG_MODEM_CELLULAR`)
- Network shell (`CONFIG_NET_SHELL`)
- Zperf (`CONFIG_NET_ZPERF`)
- Power management (`CONFIG_PM_DEVICE`)
- AT command shell (`CONFIG_MODEM_AT_SHELL`)

5.2.7 References

- *Cellular PPP modem*
- *Serial Modem application*
- Zephyr Networking
- Zephyr Network APIs
- Zephyr Modem modules

5.3 Serial Modem AT Client library

Note:

This library is Experimental.

The Serial Modem AT Client library exposes the AT command interface of the *Serial Modem application* for external devices over a serial interface. This library is intended for applications running on an external MCU that are connected to an nRF91 Series SiP through UART.

5.3.1 Overview

The Serial Modem AT Client library allows you to perform the following functions:

- Send modem or Serial Modem proprietary AT commands, receive responses and notifications, similar to the AT Host library. Received AT responses or notifications can be parsed by the AT parser library.
- Send raw data in Serial Modem data mode. Refer to *Data mode*.
- Manage the UART power state of the host.
- Manage the UART power state of the Serial Modem with DTR (Data Terminal Ready) and RI (Ring Indicator) pins. See *UART configuration* for more information.
- Monitor AT notifications with registered callbacks, similar to the AT monitor library.
- Send AT commands through UART or RTT shell, similar to the AT shell library.

5.3.2 Configuration

Configure the following Kconfig option to enable the library:

- `CONFIG_SM_AT_CLIENT` - Enables the Serial Modem AT Client library.

Optionally configure the following Kconfig options based on need:

- `CONFIG_SM_AT_CLIENT_SHELL` - Enables the shell function in the Serial Modem AT Client library, which is disabled by default.
- `CONFIG_SM_AT_CLIENT_AT_CMD_RESP_MAX_SIZE` - Configures the size of the AT command response buffer. The default size is 2100 bytes, which is aligned with Serial Modem.
- `CONFIG_SM_AT_CLIENT_UART_RX_BUF_COUNT` - Configures the number of RX buffers for the UART device. The default value is 3.
- `CONFIG_SM_AT_CLIENT_UART_RX_BUF_SIZE` - Configures the size of the RX buffer for the UART device. The default value is 256 bytes.
- `CONFIG_SM_AT_CLIENT_UART_TX_BUF_SIZE` - Configures the size of the TX buffer for the UART device. The default value is 256 bytes.

Use one of the following options to select the termination character:

- `CONFIG_SM_AT_CLIENT_CR_TERMINATION` - Enables <CR> as the termination character, which is selected by default.
- `CONFIG_SM_AT_CLIENT_LF_TERMINATION` - Enables <LF> as the termination character.
- `CONFIG_SM_AT_CLIENT_CR_LF_TERMINATION` - Enables <CR+LF> as the termination character.

You must configure the same termination character as that configured in Serial Modem on the nRF91 Series SiP. The library sends the termination character automatically after an AT command.

The application must configure the devicetree nodes for the UART, UART pins, and DTR GPIOs.

The following is an example overlay for configuring UART and DTR/RI GPIOs with the nrf54l15dk:

```
/ {
  chosen {
    ncs,sm-uart = &uart30;
  };
};

/* Serial Modem AT Client <-> Serial Modem UART */
&uart30 {
  compatible = "nordic,nrf-uarte";
  current-speed = <115200>;
  status = "okay";
  hw-flow-control;
};

/* DTR gpios for uart30 */
/ {
  dte_dtr: dte_dtr {
    compatible = "nordic,dte-dtr";
    dtr-gpios = <&gpio1 11 GPIO_ACTIVE_LOW>;
    ri-gpios = <&gpio1 12 (GPIO_ACTIVE_LOW | GPIO_PULL_UP)>;
  };
};
```

UART baud rate, hardware flow control, and parity must match the Serial Modem application configuration. UART pins must be wired correctly (TX to RX, RTS to CTS) to the Serial Modem application UART pins. DTR and RI GPIO must be wired to the corresponding DTR and RI pins of the Serial Modem application.

5.3.3 Shell usage

Serial Modem AT commands

Send AT commands for Serial Modem in shell:

```
uart:~$ sm AT%XPTW=4,\"0001\"

OK

uart:~$ sm at%ptw?

%XPTW: 4,"0001"
%XPTW: 5,"0011"

OK
```

Serial Modem accepts AT command characters in upper, lower, or mixed case.

Host commands

Use `smsh` command to see commands for the Serial Modem AT Client library functions.

```
uart:~$ smsh
smsh - Commands handled in Serial Modem AT Client shell
device
Subcommands:
  uart  : Enable/Disable DTR UART.

uart:~$ smsh uart
uart - Enable/Disable DTR UART.
Subcommands:
  auto     : [<inactivity_period>]
             (Default) Automatically enable DTR UART from RI.
Disable DTR UART
             after inactivity period (default value is
100ms).
  enable   : Enable DTR UART. Disable automatic handling.
  disable  : Disable DTR UART. Disable automatic handling.
```

Enable or disable host UART and command Serial Modem to do the same with DTR:

```
uart:~$ smsh uart enable
Enable DTR UART.

uart:~$ smsh uart disable
Disable DTR UART.
```

Set the automatic UART handling to 1000 ms inactivity period:

```
uart:~$ smsh uart auto 1000
Automatic DTR UART. Inactivity timeout 1000 ms
```

When automatic UART and DTR handling is enabled, the UART's will be suspended after the inactivity period. UARTs are resumed when there is an RI signal from the Serial Modem or when the host sends data.

5.3.4 Serial Modem AT Client Monitor usage

The Serial Modem AT Client Monitor has similar functions to the AT monitor library, except "Direct dispatching".

```
SM_MONITOR(network, "\r\n+CEREG:", cereg_mon);
```

5.3.5 API documentation

Header file: `include/sm_at_client.h`
Source file: `lib/sm_at_client/sm_at_client.c`
Source file: `lib/sm_at_client/sm_at_client_monitor.c`

group **Serial Modem AT Client library**

Public APIs for the Serial Modem AT Client library.

Defines

SM_AT_CMD_RESPONSE_MAX_LEN

Max size of AT command response is 2100 bytes.

MON_ANY

Wildcard. Match any notifications.

MON_PAUSED

Monitor is paused.

MON_ACTIVE

Monitor is active, default

SM_MONITOR(name, _filter, _handler, ...)

Define an Serial Modem monitor to receive notifications in the system workqueue thread.

Parameters

- **name** – The monitor's name.
- **_filter** – The filter for AT notification the monitor should receive, or MON_ANY to receive all notifications.
- **_handler** – The monitor callback.

- `...` – Optional monitor initial state (`MON_PAUSED` or `MON_ACTIVE`). The default initial state of a monitor is `MON_ACTIVE`.

Typedefs

typedef void (***sm_data_handler_t**)(const uint8_t *data, size_t datalen)

Handler to handle data received from Serial Modem, which could be AT response, AT notification or simply raw data (for example DFU image).

Note:

The handler runs from uart callback. It must not call *sm_at_client_send_cmd*. The data should be copied out by the application as soon as called.

Param data
Data received from Serial Modem.

Param datalen
Length of the data received.

typedef void (***sm_ri_handler_t**)(void)

Handler to handle the Ring Indicate (RI) signal from Serial Modem.

typedef void (***sm_monitor_handler_t**)(const char *notif)

Serial Modem monitor callback.

Param notif
The AT notification callback.

Enums

enum **at_cmd_state**

AT command result codes.

Values:

enumerator **AT_CMD_OK**

enumerator **AT_CMD_ERROR**

enumerator **AT_CMD_ERROR_CMS**

enumerator **AT_CMD_ERROR_CME**

enumerator **AT_CMD_PENDING**

Functions

int **sm_at_client_init**(*sm_data_handler_t* handler, bool automatic_uart, k_timeout_t inactivity_timeout)

Initialize Serial Modem AT Client library.

Parameters

- **handler** – Pointer to a handler function of type *sm_data_handler_t*.
- **automatic_uart** – If true, DTR and UART are automatically managed by the library.
- **inactivity_timeout** – Inactivity timeout for DTR and UART disablement. Only used if `automatic` is true.

Returns

Zero on success, non-zero otherwise.

int **sm_at_client_uninit**(void)

Un-initialize Serial Modem AT Client.

int **sm_at_client_register_ri_handler**(*sm_ri_handler_t* handler)

Register callback for Ring Indicate (RI) pin.

Parameters

handler – Pointer to a handler function of type *sm_ri_handler_t*.

Return values

Zero – on success. Otherwise, a (negative) error code is returned.

void **sm_at_client_automatic_dtr_uart**(k_timeout_t inactivity)

Configure automatic DTR UART handling.

If automatic DTR UART handling is enabled, the library will enable DTR UART when RI signal is detected, and disable it after inactivity timeout.

Parameters

inactivity – Inactivity timeout for DTR UART disablement.

void **sm_at_client_disable_dtr_uart**(void)

Disable DTR UART.

Disables DTR UART. Disables automatic DTR UART handling.

void **sm_at_client_enable_dtr_uart**(void)

Enable DTR UART.

Enables DTR UART. Disables automatic DTR UART handling.

int **sm_at_client_send_cmd**(const char *const command, uint32_t timeout)

Function to send an AT command in Serial Modem command mode.

This function wait until command result is received. The response of the AT command is received through the sm_ind_handler_t registered in *sm_at_client_init*.

Parameters

- **command** – Pointer to null terminated AT command string without command terminator

- **timeout** – Response timeout for the command in seconds, Zero means infinite wait

Return values

- **state** – *at_cmd_state* if command execution succeeds.
- **-EAGAIN** – if command execution times out.
- **other** – if command execution fails.

int **sm_at_client_send_data**(const uint8_t *const data, size_t datalen)

Function to send raw data in Serial Modem data mode.

Parameters

- **data** – Raw data to send
- **datalen** – Length of the raw data

Returns

Zero on success, non-zero otherwise.

static inline void **sm_monitor_pause**(struct *sm_monitor_entry* *mon)

Pause monitor.

Pause monitor `mon` from receiving notifications.

Parameters

mon – The monitor to pause.

static inline void **sm_monitor_resume**(struct *sm_monitor_entry* *mon)

Resume monitor.

Resume forwarding notifications to monitor `mon`.

Parameters

mon – The monitor to resume.

struct **sm_monitor_entry**

#include <sm_at_client.h> Serial Modem monitor entry.

Public Members

const char ***filter**

The filter for this monitor.

const *sm_monitor_handler_t* **handler**

Monitor callback.

uint8_t **paused**

Monitor is paused.

5.4 Serial Modem XDFU library

Note:

This library is Experimental.

The Serial Modem XDFU library implements the host side of the `AT#XDFU` update sequence, so that a host MCU can update the nRF91 Series SiP that runs the *Serial Modem application*. This library is intended for applications running on an external MCU that use Zephyr's `modem_cellular` driver to control the Serial Modem device over UART, for example, the *PPP shell sample*.

5.4.1 Overview

The Serial Modem XDFU library drives the full DFU exchange described in *DFU AT commands* on behalf of the application:

- Sends `AT#XDFUINIT`, `AT#XDFUWRITE`, and `AT#XDFUAPPLY` for one of the following image types: application firmware, delta modem firmware, full modem firmware, or the MCUboot bootloader.
- Streams the update file from a mounted file system to the Serial Modem device in chunks, using the `modem_cellular` driver's UART pipe directly. This means the pipe must not be otherwise in use by the driver's own chat scripts while the update runs.
- For full modem firmware updates, decodes the CBOR-encoded update package (as produced for nRF Connect SDK full modem firmware updates) and writes each firmware segment to the address indicated in the package.
- Resets the Serial Modem device with `AT#XRESET` after the update (except after a successful full modem firmware update, where the final `AT#XDFUAPPLY` already reboots the modem into the new firmware) and waits for it to become ready again.

The library exposes a single blocking call, `sm_xdfu_run()`, that runs the whole sequence for one file and returns only once the update has completed, failed, or timed out.

5.4.2 Configuration

Configure the following Kconfig option to enable the library:

- `CONFIG_SM_XDFU_LIB` - Enables the Serial Modem XDFU library.

The library also depends on the following features, which must be enabled by the application:

- `CONFIG_MODEM_CELLULAR` - Provides the modem device and UART pipe that the library attaches to.
- `CONFIG_FILE_SYSTEM` - Provides access to the update file.
- `CONFIG_ZCBOR` - Decodes the CBOR-encoded full modem firmware update package.
- `CONFIG_COMMON_LIBC_MALLOC` - Provides dynamic memory for the library's internal chat instance and CBOR metadata buffer.

5.4.3 Usage

Call `sm_xdfu_run()` with the `modem_cellular`-compatible device for the Serial Modem device, the image type to update, and the path to the update file on a mounted file system:

```
#include <sm_xdfu.h>

const struct device *modem = DEVICE_DT_GET_ONE(nordic_nrf91_sm_v2);

int ret = sm_xdfu_run(modem, SM_XDFU_TYPE_APP, "/lfs1/app_update.bin
");

if (ret < 0) {
    /* Handle error, see sm_xdfu_run() return values */
}
```

Before calling `sm_xdfu_run()`, ensure that the modem device and its underlying UART are not in use, for example, by suspending networking on the `modem_cellular` device. The function returns `-EBUSY` if either is in use.

The *PPP shell sample* sample wraps this call in an `xdfu` shell command, see its source for a complete example, including how to report the different error codes in your application.

5.4.4 API documentation

Header file: `include/sm_xdfu.h`
Source file: `lib/sm_xdfu/sm_xdfu.c`

group **Serial Modem XDFU library**

Public APIs for the Serial Modem XDFU library.

Enums

enum **sm_xdfu_image_type**

Image type accepted by the modem's AT#XDFU command.

Values:

enumerator **SM_XDFU_TYPE_APP**

enumerator **SM_XDFU_TYPE_DELTA_MFW**

enumerator **SM_XDFU_TYPE_FULL_MFW**

enumerator **SM_XDFU_TYPE_MCUBOOT_BL**

Functions

int **sm_xdfu_run**(const struct device *modem, enum *sm_xdfu_image_type* type, const char *file)

Run an AT#XDFU update on the given cellular modem.

Attach the modem's UART pipe, transfers `file` to the modem using AT#XDFU commands, and resets the modem with AT#XRESET afterwards.

Parameters

- **modem** – Cellular modem device (compatible with the Zephyr modem_cellular driver).
- **type** – Image type to update.
- **file** – Path to the update file (for example, on a mounted filesystem).

Return values

- **0** – on success.
- **-EINVAL** – Invalid `type`, or `file` is missing/empty.
- **-ENODEV** – Modem or its UART device is not ready.
- **-EBUSY** – Modem or its UART is currently in use.
- **-ETIMEDOUT** – Timed out waiting for the modem to respond.
- **-EIO** – Transfer or apply failed.

6 Releases

This page shows the information for the Serial Modem releases.

6.1 Release notes

For a list of changes for a specific Serial Modem release, refer to the following sections. See the releases on GitHub for the complete release history, more detailed changelogs, and downloadable assets.

6.1.1 Release notes for Serial Modem v2.0.0

This page tracks changes and updates as compared to the latest official release (v1.0.1). For more information refer to the following section.

Changelog

This release is based on the nRF Connect SDK v3.4.1 release. It is also based on the Memfault SDK v1.45.0 release, which is updated from the v1.40.1 in the nRF Connect SDK v3.4.1 release.

- Added:
  - Updatable bootloaders, included by default.
  - HTTP client through `AT#XHTTPC*` commands.
  - CoAP client through `AT#XCOAPC*` commands.
  - nRF Cloud provisioning through `AT#XNRFPROV` command.
  - nRF Cloud FOTA through `AT#XNRFCLOUDFOTA` command.
  - nRF Cloud observability (Memfault) through `AT#XNRFCLOUDOBS*` commands.
  - TCP server support through `AT#XLISTEN` and `AT#XACCEPT` commands.
  - More standard and dynamic handling of CMUX channels with `AT+CMUX` and `AT+CGDATA`.
  - Modem driver (`nordic,nrf91-sm-v2`) for Zephyr host using the `AT+CMUX` and `AT+CGDATA` commands.
  - `AT#XCMUXURC` for configuring the URC channel.
  - Support for using AT commands and receiving URCs from the Zephyr host. DFU support was also added to the SM PPP shell sample.

  - Shared UART for application log and modem trace. `AT#XLOG` and `AT#XTRACE` select which output to use.
  - Redacting of sensitive data from Serial Modem logs. See `AT#XLOG=1` vs. `AT#XLOG=2`.
  - nRF91M1 build configuration.
  - The `VERSION` file, which changes how version information is reported (`AT#XSMVER`).
  - Support for printing heap statistics to the log with `AT#XDBGSTATSMEM`.
  - nRF91M1 AT commands documentation.
- Updated:
  - nRF Cloud AT commands to use CoAP instead of MQTT.
  - UART to be the default application log output instead of RTT.
  - The default Serial Modem log level from INF to DBG.
  - Major refactoring of the internal AT host implementation.
  - Memory partitions from Partition Manager to devicetree (Partition Manager is deprecated in nRF Connect SDK).

6.1.2 Release notes for Serial Modem v1.0.1

This page tracks changes and updates as compared to the latest official release. For more information refer to the following section.

Changelog

This release is based on the nRF Connect SDK v3.2.4 release.

- Updated:
  - Improved pipe handling of CMUX modem trace module.
  - RI from pulse (100 ms) to level triggered, meaning RI stays asserted until the host asserts DTR.

6.1.3 Release notes for Serial Modem v1.0.0

This page tracks changes and updates as compared to the latest official release. For more information refer to the following section.

Changelog

This is the first official release of the Serial Modem. It is based on the nRF Connect SDK v3.2.2 release.

- Added:
  - DFU AT commands.
  - Runtime setting for UART speed (`AT+IPR`).

  - Modem trace support for a CMUX channel.
- Updated:
  - PPP starts on AT UART when used without CMUX. Option to use secondary UART removed.
  - UART pins for external MCU use case.
  - Marked the following as experimental:
    * Thingy:91 X board target.
    * nRF Cloud AT commands
    * MQTT AT commands
    * LwM2M carrier lib
    * Memfault
    * *Serial Modem AT Client library*

6.2 Release artifacts

The Serial Modem repository contains the release artifacts that you can download from the Assets section of the release on the release page. The following release artifacts are available:

Release artifact	Description
`serial_modem_{VERSION}_nrf9`	nRF91M1 content for nRF9151 DK.
`sm_at_client_shell_{VERSI`	AT client shell build for nRF54L15 DK host that can be used as an external MCU. See *UART configuration* for pin wiring.
`sm_ppp_shell_{VERSION}_nr`	PPP shell build for nRF54L15 DK host that can be used as an external MCU. See *UART configuration* for pin wiring.
`serial_modem_{VERSION}_inte`	Adapts the nRF9151 SiP to the BG95 module footprint through the nRF9151-MK-BG95 interposer, using the nRF91M1 content.
`serial_modem_{VERSION}_inte`	Adapts the nRF9151 SiP to the GM02S module footprint through the nRF9151-MK-GM02S interposer, using the nRF91M1 content.

The artifacts are zipped and contain build files such as:

Release artifact	Description
`.hex`	The unsigned application binary (MCUboot + application) to be flashed with Programmer or nRF Util.
`.signed.bin`	Signed application only binary to be used in FOTA/DFU.
`_dfu_applicati zip`	Signed application only binary zipped with metadata used by nRF Cloud for FOTA/DFU.
`.elf`	Application image symbols for debugging purposes.
`.dts`	Devicetree.
`.config`	Application image Kconfig options.

For more details about the build artifacts, see the build output files documentation.

6.3 Migration notes

Migration notes for the Serial Modem to assist user's transition from the previous releases.

6.3.1 Migration notes for Serial Modem v2.0.0

This document provides guidance for migrating from Serial Modem v1.x.x to v2.0.0. The general principle has been to keep v2.0.0 backwards compatible. However, a small number of changes have been introduced to reduce both maintenance and flash usage. Some of these modifications might not require any updates on the host side, even if corresponding features are in use.

Required changes

The following changes are mandatory to make your application work in the same way as in previous releases.

- Application logging backend changed from RTT to UART — The default application log backend has changed from SEGGER RTT to UART1 (VCOM1 on the nRF9151 DK). The UART is suspended at startup and activated at runtime with `AT#XLOG=1`. See *Trace AT commands* for the full command reference. If you cannot move to UART logs, see *Enabling RTT logs* for how to re-enable RTT logs.
- When compiling, remove `overlay-` prefix from Kconfig fragments and devicetree overlays as follows:
  - `overlay-*.conf` to `*.conf`
  - `overlay-*.overlay` to `*.overlay`
- Full FOTA - When compiling, rename `overlay-full_fota.conf` to `full-fota.conf` and add `full-fota-nrf9151.overlay` to the build configuration. See *FOTA AT commands* for more information.
- `#XGNSS` notification had two meanings with alternative syntaxes as follows:

```
#XGNSS: <gnss_service>,<gnss_status>
#XGNSS: <latitude>,<longitude>,<altitude>,<accuracy>,
<speed>,<heading>,<datetime>
```

The latter is now renamed to `#XGNSSPOS` as follows:

```
#XGNSSPOS: <latitude>,<longitude>,<altitude>,<accuracy>,
<speed>,<heading>,<datetime>
```

The former syntax remains unchanged.

- `AT#XNRFCLOUDPOS`:
  - Changed `<cell_pos>` parameter to `<cell_count>`. The meaning changes from no cell positioning, single-cell or multi-cell to the number of cells to be included in the location request. `0` means that cellular positioning is not requested at all.
  - The `AT#XNRFCLOUDPOS` command has been updated to use the `AT%NCELLMEAS` command internally, so the host must not use it anymore.
  - `#XNRFCLOUDPOS` notification now includes the status of the location request. The syntax has changed from:

    ```
    #XNRFCLOUDPOS: <error>
    #XNRFCLOUDPOS: <type>,<latitude>,<longitude>,<uncertainty>
    ```

    to:

    ```
    #XNRFCLOUDPOS: <status>[,<type>,<latitude>,<longitude>,
    <uncertainty>]
    ```

- The `CONFIG_SM_AT_BUF_SIZE` Kconfig option has been removed. The maximum size of the AT command is 8190 bytes, which includes the terminator character.
- The `CONFIG_SM_LOG_LEVEL_DBG` Kconfig option has been set by default. This is required for `AT#XLOG=2`. Set the `CONFIG_SM_LOG_LEVEL_INF` Kconfig option if you do not need debug logging.

Partition Manager removal The Serial Modem no longer uses the nRF Connect SDK Partition Manager.

Custom static partition layout migration All flash and SRAM partitions are now defined in devicetree overlays instead of `pm_static_*.yml` files.

If you maintained a custom `pm_static_*.yml` file, recreate the partition layout as a devicetree overlay, using the files in `app/boards/` and `app/*.overlay` as a reference. For a general guide on migrating from Partition Manager to DTS, see the nRF Connect SDK PM to DTS migration page.

If you are planning to upgrade the applications in field from v1.x.x to v2.0.0 and were using the default partition layout, see *Building without the immutable B0 bootloader* for how to disable the B0 partition, which is used by default in v2.0.0.

Merged hex generation Previously, Partition Manager automatically generated a `merged.hex` file in the build directory. This file is no longer generated automatically. After running `west build`, run the following command to generate the combined hex file:

```
west ncs-mergehex --no-rebuild
```

The output file name is now board-specific instead of the generic `merged.hex`:

- For `nrf9151dk/nrf9151/ns`, use `build/merged_nrf9151dk.hex`.
- For `thingy91x/nrf9151/ns`, use `build/merged_thingy91x.hex`.
- For `nrf54l15dk/nrf54l15/cpuapp`, use `build/merged_nrf54l15dk.hex`.

Informational changes

The following changes are listed for informational purposes, and many hosts will work without any changes.

- On a fatal error (such as a CPU fault or stack overflow), the device now resets instead of halting, using the `CONFIG_RESET_ON_FATAL_ERROR` Kconfig option.
- Application rollback prevention - Application firmware DFU now uses MCUboot downgrade prevention (`CONFIG_MCUBOOT_DOWNGRADE_PREVENTION`). The device rejects signed application images older than the version currently running. Set `app/VERSION` before each published build; see *Releasing Serial Modem*.
- `AT#XSMVER` - No change to the response layout for hosts. `<sm_version>` is still a quoted Git-describe-style string (for example `"v2.0.0-18-g2c85d9224fca"`). Existing parsers can keep using it as in v1.x.x. The `major.minor.patch` part now follows `app/VERSION` (same value used for rollback prevention), the `-N-g<hash>` suffix still comes from Git.
- Ring Indication (RI) - Change RI from pulse (100 ms) to level triggered, meaning RI stays asserted until the host asserts DTR. After the Serial Modem has enabled UART, RI will be deasserted.
- HTTP client has been added and it's enabled by default. Use CONFIG_SM_HTTPC=n if you do not need it and want to save flash.
- nRF Cloud transport has been changed from MQTT to CoAP.
- `CONFIG_SM_NRF_CLOUD_LOCATION` is enabled by default whenever `CONFIG_SM_NRF_CLOUD` is enabled. Use `CONFIG_SM_NRF_CLOUD_LOCATION=n` if you do not need it and want to save flash.
- The `sm_ppp_shell` sample no longer supports the `nrf54l15dk/nrf54l15/cpuapp/ns` (TF-M) board target. Use `nrf54l15dk/nrf54l15/cpuapp` instead.

6.3.2 Migration notes from nRF Connect SDK v3.1.x SLM

This migration note helps you to move from nRF Connect SDK v3.1.x Serial LTE modem (SLM) to the Serial Modem.

There are several breaking changes between Serial Modem and nRF Connect SDK SLM, such as renaming, file movement, AT command changes, overlay file changes, and so on. The following sections cover all the changes that must be taken into account.

Background

The base of the Serial Modem repository is a copy of nRF Connect SDK SLM and related components from the nRF Connect SDK main branch commit `437f372b37849fe215243f8de48847d578976c13`, which is in practice a bit after the nRF Connect SDK release v3.1.0.

The following nRF Connect SDK files were copied into this repository:

- `applications/serial_lte_modem` to `app`
- `lib/modem_slm` to `lib/sm_at_client`
- `samples/cellular/slm_shell` to `samples/sm_at_client_shell`
- `include/modem/modm_slm.h` to `include/sm_at_client.h`
- `doc/nrf/libraries/modem/modem_slm.rst` to `doc/lib/sm_at_client.rst`

Required changes

The following changes are mandatory to make your application work in the same way as in previous releases.

This section gives instructions on how to migrate from the nRF Connect SDK v3.1.x SLM to the Serial Modem:

- Rename the following Kconfig options:
  - `CONFIG_SLM_*` to `CONFIG_SM_*`
  - `CONFIG_MODEM_SLM_*` to `CONFIG_SM_AT_CLIENT_*`
  - `CONFIG_SLM_CMUX_TX_BUFFER_SIZE` to `CONFIG_SM_URC_BUFFER_SIZE`
- When compiling, remove `overlay-` prefix from Kconfig fragments and devicetree overlays as follows:
  - `overlay-*.conf` to `*.conf`
  - `overlay-*.overlay` to `*.overlay`
- Code patches:
  - Renamed the file names from `slm_` to `sm_` and `modem_slm` to `sm_at_client`.
  - Functions and other symbols in the code have been renamed accordingly making automatic patching to likely fail.
- Changed the default AT command terminator from `\r\n` (`CONFIG_SM_CR_LF_TERMINATION` and `CONFIG_SM_AT_CLIENT_CR_LF_TERMINATION`) to `\r` (`CONFIG_SM_CR_TERMINATION` and `CONFIG_SM_AT_CLIENT_CR_TERMINATION`).

- Application logging backend changed from RTT to UART — The default application log backend has changed from SEGGER RTT to UART1 (VCOM1 on the nRF9151 DK). The UART is suspended at startup and activated at runtime with `AT#XLOG=1`. See *Trace AT commands* for the full command reference. If you cannot move to UART logs, see *Enabling RTT logs* for how to re-enable RTT logs.
- Replaced the use of `CONFIG_NRF_CLOUD_LOCATION` to Serial Modem-specific new Kconfig option *CONFIG_SM_NRF_CLOUD_LOCATION*. You can now use this option to enable the nRF Cloud Location Services for cloud-assisted geolocation, which supports cellular and Wi-Fi positioning.
- Rename the following AT commands:
  - `AT#XGPS` to `AT#XGNSS`
  - `AT#XGPSDEL` to `AT#XGNSSDEL`
  - `AT#XSLMVER` to `AT#XSMVER`

DTR and RI GPIOs replace Power and Indicate pins The Serial Modem application uses DTR (Data Terminal Ready) and RI (Ring Indicator) pins to manage the UART power state instead of the Power and Indicate pins used in the nRF Connect SDK SLM.

- Removed:
  - The Power pin, which was an active low input, expected a short pulse and was configured with `CONFIG_SLM_POWER_PIN`.
  - The Indicate pin, which was active low output, sent a pulse configured with `CONFIG_SLM_INDICATE_TIME` and was configured with `CONFIG_SLM_INDICATE_PIN`.
- Added:
  - DTR pin, which is a level-based input that is configured in the devicetree with the `dtr-gpios` property.
  - RI pin, which is a level-based output that is configured in the devicetree with the `ri-gpios` property.

See *UART configuration* for more information on how DTR and RI pins work in the Serial Modem application. See *Cellular PPP modem* for information on how to configure DTR and RI pins when using the Serial Modem application as a Zephyr modem.

Partition Manager removal The Serial Modem no longer uses the nRF Connect SDK Partition Manager.

Custom static partition layout migration All flash and SRAM partitions are now defined in devicetree overlays instead of `pm_static_*.yml` files.

If you maintained a custom `pm_static_*.yml` file, recreate the partition layout as a devicetree overlay, using the files in `app/boards/` and `app/*.overlay` as a reference. For a general guide on migrating from Partition Manager to DTS, see the nRF Connect SDK PM to DTS migration page.

Merged hex generation Previously, Partition Manager automatically generated a `merged.hex` file in the build directory. This file is no longer generated automatically. After running `west build`, run the following command to generate the combined hex file:

```
west ncs-mergehex --no-rebuild
```

The output file name is now board-specific instead of the generic `merged.hex`:

- For `nrf9151dk/nrf9151/ns`, use `build/merged_nrf9151dk.hex`.
- For `thingy91x/nrf9151/ns`, use `build/merged_thingy91x.hex`.
- For `nrf54l15dk/nrf54l15/cpuapp`, use `build/merged_nrf54l15dk.hex`.

Socket AT command changes The socket AT commands have been updated to use a handle-based approach instead of socket selection `AT#XSOCKETSELECT`. This provides more flexibility and clearer socket management by directly referencing socket handles in all operations. There are also other changes to the socket AT commands to improve functionality and usability. Especially the `AT#XSEND`/`AT#XSENDTO` and `AT#XRECV`/`AT#XRECVFROM` commands have been updated significantly.

The following is the list of changes:

- Added socket closing:
  - `AT#XCLOSE` - New command to close individual sockets or all sockets at once.
  - Syntax: `AT#XCLOSE[=<handle>]` (handle is optional - omit to close all sockets).
- Updated socket creation with `AT#XSOCKET` and `AT#XSSOCKET`:
  - No longer supports closing sockets (`op=0` removed). Only creates sockets and returns a handle.
  - The `<op>` parameter has been renamed to `<family>` since socket closing is no longer supported.
  - In `AT#XSOCKET`, a new value 3 has been added for the `<family>` parameter to represent the packet family to be used with raw sockets. This value is not valid for secure sockets in `AT#XSSOCKET`.
- Removed commands:
  - `AT#XSOCKETSELECT` - Socket selection is no longer needed. Each command now directly specifies the socket handle.
- `AT#XSEND` command parameter changes:
  - Added `<handle>`, `<mode>` parameters, and optional `<data_len>` parameter (for data mode) to the `AT#XSEND` command. Changed parameter order.
    - Old syntax - `AT#XSEND[=<data>][,<flags>]`
    - New syntax for string and hex string modes - `AT#XSEND=<handle>,<mode>,<flags>,<url>,<port>,<data>`
    - New syntax for data mode - `AT#XSEND=<handle>,<mode>,<flags>,<url>,<port>[,<data_len>]`
  - Added result type to the `#XSEND` response.

    - Old response: `#XSEND: <size>`
    - New response: `#XSEND: <handle>,<result_type>,<size>`
  - A new `#XSENDNTF` notification will be sent when the network acknowledges the send operation. This notification is requested with the `<flags>` parameter in the `AT#XSEND` or `AT#XSENDTO` commands.
    - Syntax: `#XSENDNTF: <handle>,<status>,<size>`
- `AT#XSENDTO` parameter changes:
  - Added `<handle>`, `<mode>` parameters, and optional `<data_len>` parameter (for data mode) to the `AT#XSENDTO` command. Changed parameter order.
    - Old syntax: `AT#XSENDTO=<url>,<port>[,<data>][,<flags>]`
    - New syntax for string and hex string modes: `AT#XSENDTO=<handle>,<mode>,<flags>,<url>,<port>,<data>`
    - New syntax for data mode: `AT#XSENDTO=<handle>,<mode>,<flags>,<url>,<port>[,<data_len>]`
- `AT#XRECV` parameter changes:
  - Added `<handle>`, `<mode>` parameters, and optional `<data_len>` parameter to the `AT#XRECV` command. Changed parameter order.
    - Old syntax: `AT#XRECV=<timeout>[,<flags>]`
    - New syntax: `AT#XRECV=<handle>,<mode>,<flags>,<timeout>[,<data_len>]`
- `AT#XRECVFROM` parameter changes:
  - Added `<handle>`, `<mode>` parameters, and optional `<data_len>` parameter to the `AT#XRECVFROM` command. Changed parameter order.
    - Old syntax: `AT#XRECVFROM=<timeout>[,<flags>]`
    - New syntax: `AT#XRECVFROM=<handle>,<mode>,<flags>,<timeout>[,<data_len>]`
  - Added the new `<mode>` parameter for send or receive commands:
    - For send commands (AT#XSEND, AT#XSENDTO):
      - `0` - String mode. Data provided directly as a string parameter.
      - `1` - Hex string mode. Data provided as a hexadecimal string representation.
      - `2` - Data mode. Enter data input mode for binary data.
    - For receive commands (AT#XRECV, AT#XRECVFROM):
      - `0` - Binary mode. Data received as binary data.
      - `1` - Hex string mode. Data received as a hexadecimal string representation.
- `AT#XAPOLL` parameter changes:

  - Added <handle> as the first parameter to the AT#XAPOLL command. If a handle is provided, the operation applies only to that specific socket. If no handle is provided, operation applies to all open sockets. Removed support for multiple socket handles in a single command.
    * Old syntax: AT#XAPOLL=<op>[,<events>[,<handle1>[,<handle2> ...<handle8>]]
    * New syntax: AT#XAPOLL=[<handle>],<op>[,<events>]
  - Other socket operations now require handle parameter:
    * AT#XSOCKETOPT=<handle>,<op>,<name>[,<value>] (handle parameter added)
    * AT#XSSOCKETOPT=<handle>,<op>,<name>[,<value>] (handle parameter added)
    * AT#XBIND=<handle>,<port> (handle parameter added)
    * AT#XCONNECT=<handle>,<url>,<port> (handle parameter added)
- AT#XLISTEN parameter changes:
  - Added <handle> parameter to the AT#XLISTEN command.
    * Old syntax: AT#XLISTEN
    * New syntax: AT#XLISTEN=<handle>
- AT#XACCEPT parameter changes:
  - Removed <timeout> parameter and added <handle> parameter to the AT#XACCEPT command. The command is now non-blocking and returns immediately with an error if there is no incoming connection to accept.
    * Old syntax: AT#XACCEPT=<timeout>
    * New syntax: AT#XACCEPT=<handle>
  - Response format now includes CID and port: #XACCEPT: <handle>,<cid>,"<peer_addr>",<peer_port> (previously #XACCEPT: <handle>,"<ip_addr>")
- Response format changes:
  - AT#XSOCKETOPT - Response to get options now includes socket handle: #XSOCKETOPT: <handle>,<value> (previously just #XSOCKETOPT: <value>)
  - AT#XSSOCKETOPT - Response to get options now includes socket handle: #XSSOCKETOPT: <handle>,<value> (previously just #XSSOCKETOPT: <value>)
  - AT#XCONNECT - Response now includes socket handle: #XCONNECT: <handle>,<status> (previously just #XCONNECT: <status>)
  - AT#XSEND - Response now includes socket handle: #XSEND: <handle>,<size> (previously just #XSEND: <size>)
  - AT#XRECV - Response now includes socket handle and mode: #XRECV: <handle>,<mode>,<size> (previously just #XRECV: <size>)

- `AT#XSENDTO` - Response now includes socket handle: `#XSENDTO: <handle>,<size>` (previously just `#XSENDTO: <size>`)
- `AT#XRECVFROM` - Response now includes socket handle and mode: `#XRECVFROM: <handle>,<mode>,<size>,"<ip_addr>",<port>` (previously just `#XRECVFROM: <size>,"<ip_addr>",<port>`)

Migration example:

Old approach (nRF Connect SDK SLM):

```
AT#XSOCKET=1,1,0          // Open socket, returns handle 1
AT#XCONNECT="server",80   // Connect socket handle 1
AT#XSEND="data"           // Send on socket handle 1
AT#XSOCKET=1,1,0          // Open socket, returns handle 2
AT#XCONNECT="server",80   // Connect socket handle 2
AT#XRECV=10               // Receive data from socket handle
2 with 10s timeout, no flags
AT#XSOCKETSELECT=1        // Select socket handle 1
AT#XSOCKET=0              // Close selected socket handle 1
```

New approach (Serial Modem):

```
AT#XSOCKET=1,1,0          // Open socket, returns handle 1
AT#XCONNECT=1,"server",80 // Connect socket handle 1
AT#XSEND=1,0,0,"data"     // Send on socket handle 1
AT#XSOCKET=1,1,0          // Open socket, returns handle 2
AT#XCONNECT=2,"server",80 // Connect socket handle 2
AT#XRECV=2,0,0,10         // Receive data from socket handle
2 with mode 0, no flags, 10s timeout
AT#XCLOSE=1               // Close socket handle 1
```

HTTP client changes The HTTP client has been redesigned from a dedicated connection-oriented interface to a socket-based interface. The old nRF Connect SDK SLM HTTP client kept its own HTTP connection state with `AT#XHTTPCCON`, sent requests with `AT#XHTTPCREQ`, and delivered the raw HTTP response stream followed by `#XHTTPCRSP` notifications. In Serial Modem, the HTTP client reuses Socket AT commands for transport setup and provides dedicated HTTP AT commands for requests and notifications, including headers, body data, and completion status.

The following is the list of changes:

- Removed `AT#XHTTPCCON`.
  - Create a plain TCP socket with `AT#XSOCKET` or a TLS socket with `AT#XSSOCKET`.
  - Configure TLS options such as security tag and peer verification with `AT#XSSOCKETOPT`.
  - Establish the transport connection with `AT#XCONNECT=<handle>,<host>,<port>`.
  - Close the connection with `AT#XCLOSE` when no more requests are needed.

- Updated AT#XHTTPCREQ to operate on an already connected socket.
  - Old syntax: AT#XHTTPCREQ=<method>,<resource>[,<headers>[,<content_type>,<content_length>[,<chunked_transfer>]]]
  - New syntax: AT#XHTTPCREQ=<handle>,<url>,<method>[,<auto_reception>[,<body_len>[,<header 1>[,<header 2>[...]]]]]
- Changed request parameter model.
  - <method> changed from a string such as "GET" or "POST" to an integer:
    - 0 - GET
    - 1 - POST
    - 2 - PUT
    - 3 - DELETE
    - 4 - HEAD
  - <resource> is replaced by a full <url> parameter.
  - Optional headers are no longer passed as one CRLF-delimited string. Each header is now passed as a separate optional parameter.
  - For POST and PUT, the body upload length is given with <body_len>. The command then enters data mode and the host must send exactly that many bytes.
  - For GET or HEAD requests with extra headers, set <body_len> to 0 as a placeholder before the header parameters.
- Changed response delivery model.
  - Removed the old raw response plus #XHTTPCRSP: <received_byte_count>,<state> framing.
  - Response headers are now reported with #XHTTPCHEAD: <handle>,<status_code>,<content_length>.
  - In automatic mode, response body chunks are reported with #XHTTPCDATA: <handle>,<offset>,<length> and the raw body bytes follow immediately.
  - In manual mode, the host pulls body chunks explicitly with AT#XHTTPCDATA=<handle>[,<length>].
  - Request completion including failure, timeout, or cancel is reported with #XHTTPCSTAT: <handle>,<status_code>,<total_bytes>.
- Added request cancellation with AT#XHTTPCCANCEL=<handle>.

Migration example:

Old approach (nRF Connect SDK SLM):

```
AT#XHTTPCCON=1,"postman-echo.com",80
#XHTTPCCON: 1
OK

AT#XHTTPCREQ="GET","/get?foo1=bar1&foo2=bar2"
```

(continues on next page)

(continued from previous page)

```
OK
#XHTTPCREQ: 0

HTTP/1.1 200 OK
<headers>

#XHTTPCRSP: 244,1
<244 bytes of body>
```

New approach (Serial Modem):

```
AT#XSOCKET=1,1,0
#XSOCKET: 0,1,6
OK

AT#XCONNECT=0,"postman-echo.com",80
#XCONNECT: 0,1
OK

AT#XHTTPCREQ=0,"http://postman-echo.com/get?foo1=bar1&
foo2=bar2",0
#XHTTPCREQ: 0
OK

#XHTTPCHEAD: 0,200,244

#XHTTPCDATA: 0,0,244
<244 bytes of body>

#XHTTPCSTAT: 0,200,244
```

For full details of the new interface, see *HTTP client AT commands* and *Socket AT commands*.

PPP connection management changes

- To start the PPP connection, run the `AT#XPPP=1` command when the modem is put into online mode using the `AT+CFUN=1` command. The `AT#XPPP=1` command can be run before or after the `AT+CFUN=1` command. So PPP connection is not started automatically anymore when the `AT+CFUN=1` command is run. After the `AT#XPPP=1` command is run, the PPP connection is started when the `AT+CFUN=1` command is run and stopped when the network is lost (for example, with `AT+CFUN=4`, `AT+CFUN=0`, or bad reception). When the network is regained (for example, with `AT+CFUN=1`), the PPP connection is started again automatically. To permanently stop the PPP connection, either the remote peer must disconnect the PPP using LCP termination or the `AT#XPPP=0` command must be run. If PPP is terminated using LCP termination or the `AT#XPPP=0` command, the PPP connection can be started again with the `AT#XPPP=1` command.
- The default behavior of CMUX channels has changed if DLCI 1 is used for PPP. Now

when PPP is shut down, the CMUX channel 1 switches to AT command mode, and channel 2 is not used for AT commands anymore.

- Removed:
  - The `overlay-zephyr-modem.conf` file as the default behavior of the Serial Modem application is compatible with the Zephyr modem driver.
  - The `overlay-ppp-cmux-linux.conf` Kconfig fragment file. Use the `ppp.conf` and `cmux.conf` files instead.

Other changes

- On a fatal error (such as a CPU fault or stack overflow), the device now resets instead of halting, using the `CONFIG_RESET_ON_FATAL_ERROR` Kconfig option.
- `#XGNSS` notification had two meanings with alternative syntaxes as follows:

```
#XGNSS: <gnss_service>,<gnss_status>
#XGNSS: <latitude>,<longitude>,<altitude>,<accuracy>,<speed>,
<heading>,<datetime>
```

  The latter is now renamed to `#XGNSSPOS` as follows:

```
#XGNSSPOS: <latitude>,<longitude>,<altitude>,<accuracy>,<speed>,
<heading>,<datetime>
```

  The former syntax remains unchanged.

- `AT#XNRFCLOUDPOS`:
  - Changed `<cell_pos>` parameter to `<cell_count>`. The meaning changes from no cell positioning, single-cell or multi-cell to the number of cells to be included in the location request. `0` means that cellular positioning is not requested at all.
  - The `AT#XNRFCLOUDPOS` command has been updated to use the `AT%NCELLMEAS` command internally, so the host must not use it anymore.
  - `#XNRFCLOUDPOS` notification now includes the status of the location request. The syntax has changed from:

```
#XNRFCLOUDPOS: <error>
#XNRFCLOUDPOS: <type>,<latitude>,<longitude>,<uncertainty>
```

    to:

```
#XNRFCLOUDPOS: <status>[,<type>,<latitude>,<longitude>,
<uncertainty>]
```

Removed features

This section lists features that have been removed from the Serial Modem compared to the nRF Connect SDK v3.1.x Serial LTE modem (SLM). If you need any of those features with this Serial Modem, please contact customer support and describe your use case.

- Removed:
  - Support for the `nrf9161dk`, `nrf9160dk`, `thingy91`, and `nrf9131ek` boards.
    - Use `nrf9151dk` instead.
  - Support for the `nrf5340dk`, `nrf52840dk`, and `nrf7002dk` boards from the *AT Client Shell sample*.
    - Use `nrf54l15dk` instead.
  - Native TLS support including `overlay-native_tls.conf`.
  - TCP and UDP clients. This includes the removal of the following AT commands:
    - `AT#XTCPCLI`
    - `AT#XTCPSEND`
    - `AT#XUDPCLI`
    - `AT#XUDPSEND`

    The following URC notifications have also been removed:
    - `#XTCPDATA`
    - `#XUDPDATA`

    You can replace this functionality by using the socket AT commands.

    Migration examples:
    - TCP IPv4 client

      nRF Connect SDK SLM approach:

```
AT#XTCPCLI=1,"test.server.com",1234

#XTCPCLI: 0,"connected"

OK

AT#XTCPSEND="echo this"

#XTCPSEND: 9

OK

#XTCPDATA: 9
echo this

AT#XTCPCLI=0
```

(continues on next page)

(continued from previous page)

```
OK

#XTCPCLI: 0,"disconnected"
```

Serial Modem approach:

```
AT#XSOCKET=1,1,0

#XSOCKET: 0,1,6

OK

AT#XRECVCFG=0,3

OK

AT#XCONNECT=0,"test.server.com",1234

#XCONNECT: 0,1

OK

AT#XSEND=0,0,0,"echo this"

#XSEND: 0,0,9

OK

#XRECV: 0,0,9
echo this

AT#XCLOSE

#XCLOSE: 0,0

OK
```

* DTLS IPv6 client

  nRF Connect SDK SLM approach:

```
 AT#XUDPCLI=2,"test.server.com",1235,1000

 #XUDPCLI: 0,"connected"

 OK
```

(continues on next page)

(continued from previous page)

```
 AT#XUDPSEND="echo this"

 #XUDPSEND: 9

 OK

 #XUDPDATA: 9,"::",0
 echo this

AT#XUDPCLI=0

OK
```

Serial Modem approach:

```
AT#XSSOCKET=2,2,0,1000

 #XSSOCKET: 0,2,273

 OK

 AT#XRECVCFG=0,3

 OK

 AT#XCONNECT=0,"test.server.com",1235

 #XCONNECT: 0,1

 OK

 AT#XSEND=0,0,0,"echo this"

 #XSEND: 0,0,9

 OK

 #XRECV: 0,0,9
 echo this

 AT#XCLOSE=0

 #XCLOSE: 0,0

 OK
```

You can set the parameters such as `<hostname_verify>` and

<use_dtls_cid> using the AT#XSSOCKETOPT command.

- TCP and UDP servers. The following AT commands have been removed:
  * AT#XTCPSVR
  * AT#XTCPHANGUP
  * AT#XUDPSVR

  The AT#XLISTEN and AT#XACCEPT commands have been reintroduced and you can use them to implement TCP server functionality using the socket AT commands. However, there is no support for TLS or DTLS servers, as the nRF91 modem does not support TLS server sockets.

  You can replace this functionality by using the socket AT commands.

  Migration examples:

  * TCP IPv4 server

    nRF Connect SDK SLM approach:

```
AT#XTCPSVR=1,1000

#XTCPSVR: 0,"started"

OK

#XTCPSVR: "192.0.2.1","connected"

#XTCPDATA: 9
echo this

AT#XTCPSEND="echo this"

#XTCPSEND: 9

OK

AT#XTCPSVR?

#XTCPSVR: 0,2,1

OK

AT#XTCPHANGUP=2

#XTCPSVR: 0,"disconnected"

OK

AT#XTCPSVR=0
```

(continues on next page)

(continued from previous page)

```
#XTCPSVR: 0,"stopped"

OK
```

Serial Modem approach:

```
AT#XSOCKET=1,1,0

#XSOCKET: 0,1,6

OK

AT#XAPOLL=,1,1

OK

AT#XBIND=0,1000

OK

AT#XLISTEN=0

OK

#XAPOLL: 0,1

AT#XACCEPT=0

#XACCEPT: 1,0,"192.0.2.1",54321

OK

#XAPOLL: 1,1

AT#XRECV=1,0,0,1

#XRECV: 1,0,9
echo this

OK

AT#XSEND=1,0,0,"echo this"

#XSEND: 1,0,9

OK
```

(continues on next page)

(continued from previous page)

```
AT#XCLOSE=1

#XCLOSE: 1,0

OK

AT#XCLOSE=0

#XCLOSE: 0,0

OK
```

* UDP IPv6 server

  nRF Connect SDK SLM approach:

```
AT#XUDPSVR=2,1235

#XUDPSVR: 0,"started"

OK

#XUDPDATA: 9,"2001:db8::1",54321
echo this

AT#XUDPSEND="echo this"

#XUDPSEND: 9

OK

AT#XUDPSVR=0

#XUDPSVR: 0,"stopped"

OK
```

  Serial Modem approach:

```
AT#XSOCKET=2,2,0

#XSOCKET: 0,2,17

OK

AT#XAPOLL=0,1,1
```

(continues on next page)

(continued from previous page)

```
OK

AT#XBIND=0,1235

OK

#XAPOLL: 0,1

AT#XRECVFROM=0,0,0,1

#XRECVFROM: 0,0,9,"2001:db8::1",54321
echo this

OK

AT#XSENDTO=0,0,0,"2001:db8::1",54321,"echo this"

#XSENDTO: 0,9

OK

AT#XCLOSE=0

#XCLOSE: 0,0

OK
```

- FTP and TFTP clients, including `AT#XFTP` and `AT#XTFTP` commands.
- The `AT#XGPIO` AT command.
- The `AT#XPOLL` command. Use `AT#XAPOLL` instead.
- The `CONFIG_SLM_DATAMODE_URC` Kconfig option.
- The `CONFIG_SLM_START_SLEEP` Kconfig option.
- The `CONFIG_SLM_AT_BUF_SIZE` Kconfig option. The maximum size of the AT command is 8190 bytes, which includes the terminator character.
- The `overlay-zephyr-modem.conf` file as the default behavior of the Serial Modem application is compatible with the Zephyr modem driver.
- The `overlay-ppp-cmux-linux.conf` Kconfig fragment file. Use the `ppp.conf` and `cmux.conf` files instead.
- The `sm_auto_connect.h` header file. Use the `CONFIG_SM_AUTO_CONNECT*` Kconfig options to configure automatic network attach.
- The `CONFIG_SM_SKIP_READY_MSG` Kconfig option. The `Ready\r\n` message is always sent when the Serial Modem application is ready to accept AT commands.
- The `CONFIG_SM_AT_MAX_PARAM` Kconfig option. This Kconfig option has not been relevant in the latest versions of nRF Connect SDK SLM, as the AT parser

library is now used.

- The `CONFIG_SM_GNSS_OUTPUT_NMEA_ON_CMUX_CHANNEL` Kconfig option. You can see the NMEA messages in debug logs with the `CONFIG_SM_GNSS_OUTPUT_NMEA_SATELLITES` Kconfig option, which is enabled by default if the `CONFIG_SM_LOG_LEVEL_DBG` Kconfig option is set.

Index

A

at_cmd_state (*C++ enum*), 250
at_cmd_state::AT_CMD_ERROR (*C++ enumerator*), 250
at_cmd_state::AT_CMD_ERROR_CME (*C++ enumerator*), 250
at_cmd_state::AT_CMD_ERROR_CMS (*C++ enumerator*), 250
at_cmd_state::AT_CMD_OK (*C++ enumerator*), 250
at_cmd_state::AT_CMD_PENDING (*C++ enumerator*), 250

M

MON_ACTIVE (*C macro*), 249
MON_ANY (*C macro*), 249
MON_PAUSED (*C macro*), 249

S

sm_at_client_automatic_dtr_uart (*C++ function*), 251
sm_at_client_disable_dtr_uart (*C++ function*), 251
sm_at_client_enable_dtr_uart (*C++ function*), 251
sm_at_client_init (*C++ function*), 251
sm_at_client_register_ri_handler (*C++ function*), 251
sm_at_client_send_cmd (*C++ function*), 251
sm_at_client_send_data (*C++ function*), 252
sm_at_client_uninit (*C++ function*), 251
SM_AT_CMD_RESPONSE_MAX_LEN (*C macro*), 249
sm_data_handler_t (*C++ type*), 250
SM_MONITOR (*C macro*), 249
sm_monitor_entry (*C++ struct*), 252
sm_monitor_entry::filter (*C++ member*), 252
sm_monitor_entry::handler (*C++ member*), 252
sm_monitor_entry::paused (*C++ member*), 252
sm_monitor_handler_t (*C++ type*), 250
sm_monitor_pause (*C++ function*), 252
sm_monitor_resume (*C++ function*), 252
sm_ri_handler_t (*C++ type*), 250
sm_xdfu_image_type (*C++ enum*), 254
sm_xdfu_image_type::SM_XDFU_TYPE_APP (*C++ enumerator*), 254
sm_xdfu_image_type::SM_XDFU_TYPE_DELTA_MFW (*C++ enumerator*), 254
sm_xdfu_image_type::SM_XDFU_TYPE_FULL_MFW (*C++ enumerator*), 254
sm_xdfu_image_type::SM_XDFU_TYPE_MCUBOOT_BL (*C++ enumerator*), 254
sm_xdfu_run (*C++ function*), 255