MCU Coding Tips

Please factor in the following information when writing your MCU code:

If you need code examples that use the deprecated C++ afLib, download the archive file (.zip).

Don't Forget to Call af_lib_loop()

This is very basic, but critical: when you write MCU code using afLib2, you **must** call `af_lib_loop()` to give afLib2's state machine time to execute. Without this, afLib2 won't run, and your code will not interact properly with the Afero Platform components.

You should definitely make a call to `af_lib_loop()` in your code's `loop()` method; and you can call it elsewhere, as well, if you need to ensure that afLib2 is executing frequently enough to support your application.

```
#include <SPI.h>
#include "af_lib.h"
#include "arduino_spi.h"
```

```
// Pin Defines assume Arduino Uno
#define CS_PIN                    10
#define INT_PIN                   2

boolean shouldGetAttr = true;
af_lib_t *af_lib;

bool attrSetHandler(uint8_t requestId, const uint16_t attributeId, const uint16_t valueLen,
const uint8_t *value) {
  // Any application-specific actions
  return true;
}

void attrNotifyHandler(const uint8_t requestId, const uint16_t attributeId, const uint16_t
valueLen, const uint8_t *value) {
  // Any application-specific actions
}

void setup() {
  Serial.begin(115200);

  // Initialize afLib
  af_transport_t* arduinoSPI = arduino_transport_create_spi(CS_PIN);
  af_lib = af_lib_create(attrSetHandler, attrNotifyHandler, arduinoSPI);
  arduino_spi_setup_interrupts(af_lib, digitalPinToInterrupt(INT_PIN));
}

void loop() {
  // Any application-specific actions

  // CRITICAL: Give afLib2 processing time by calling af_lib_loop() in sketch's loop()
  af_lib_loop(af_lib);
}
```

Notes:

- afLib2 requires *time slices* to process attribute writes.
- `af_lib_loop()` performs small amounts of work every time it's called. A multi-threaded process is emulated on a single-threaded MCU.

- Call `af_lib_loop()` as often as possible in your code. If afLib2 has little to do, it returns quickly. Some complex calls, like `setAttribute`, require multiple `loop()` calls to finish.
- IMPORTANT! Calling `af_lib_loop()` from within callbacks such as `attrSetHandler` and `attrNotifyHandler` is NOT supported and will likely result in undesired behavior.
- When afLib2 is finished with an important task, you'll get a notify event:
  - `attrSetHandler` - afLib2 has completed sending data to ASR.
  - `attrNotifyHandler` - afLib2 has completed receiving data from ASR.
- Make sure afLib2 has time to complete its work. When doing multiple `setAttribute` calls, ideally wait for `attrSetHandler` callback before doing additional writes.
- If you write faster than afLib2 can process, you can fill its queue and the return code will be: `AF_ERROR_QUEUE_OVERFLOW  -4  // Queue is full`
- Always check the return code from afLib2 calls. Check for return of `AF_SUCCESS`.

Adjust the Request Queue Size If Necessary

You may notice that the return value from some afLib2 calls in your MCU code indicates a full request queue (`AF_ERROR_QUEUE_OVERFLOW = -4`). If you observe this, you should consider increasing the size of the request queue. This value is defined as the `REQUEST_QUEUE_SIZE` in `afLib.h`. The default value is 10, but you can safely change that value within the bounds of available memory on your platform.

Robust af_lib_set_attribute*() Calls

All forms of `af_lib_set_attribute()` provide a return value to indicate whether the set request has been successfully enqueued. Checking this return value and re-trying can make your use of `af_lib_set_attribute*()` more robust. Here's a code snippet from the afBlink example sketch that demonstrates some forms of this:

```
// snip //
bool rebootPending = false;
// snip //
```

```
void attrNotifyHandler(const uint8_t requestId, const uint16_t attributeId, const uint16_t
valueLen, const uint8_t *value) {

    switch (attributeId) {
        // Update the state of the LED based on the actual attribute value.
        case AF_MODULO_LED:
            moduloLEDIsOn = (*value == 0);
            break;

            // Allow the button on Modulo to control our blinking state.
        case AF_MODULO_BUTTON: {
            uint16_t *buttonValue = (uint16_t *) value;
            if (moduleButtonValue != *buttonValue) {
                moduleButtonValue = *buttonValue;
                blinking = !blinking;
// EXAMPLE 1:
                if (af_lib_set_attribute_bool(af_lib, AF_BLINK, blinking) != AF_SUCCESS) {
                    Serial.println("Could not set BLINK");
                }
            }
        }
            break;

        case AF_SYSTEM_ASR_STATE:
            Serial.print("ASR state: ");
            switch (value[0]) {
                case AF_MODULE_STATE_REBOOTED:
                    Serial.println("Rebooted");
                    break;

                case AF_MODULE_STATE_LINKED:
                    Serial.println("Linked");
                    break;

                case AF_MODULE_STATE_UPDATING:
                    Serial.println("Updating");
                    break;

                case AF_MODULE_STATE_UPDATE_READY:
                    Serial.println("Update ready - need to reboot");
// EXAMPLE 2:
                    rebootPending = true;
                    break;

                default:
```

```
                    break;
                }
                break;

            default:
                break;
        }
}

void setModuloLED(bool on) {
  if (moduloLEDIsOn != on) {
    int16_t attrVal = on ? LED_ON : LED_OFF; // Modulo LED is active low

// EXAMPLE 3:
    int timeout = 0;
    while (af_lib_set_attribute_16(af_lib, AF_MODULO_LED, attrVal) != AF_SUCCESS) {
      delay(10);
      af_lib_loop(af_lib);
      timeout++;
      if (timeout > 500) {
        // If we haven't been successful after 5 sec (500 tries, each after 10 msec delay),
request reboot
        reboot_pending = true;
        return;
      }
    }

    moduloLEDIsOn = on;
  }
}

void loop() {

// snip //

    if (rebootPending) {
        retVal = af_lib_set_attribute_32(af_lib, AF_SYSTEM_COMMAND, AF_MODULE_COMMAND_REBOOT);
        rebootPending = (retVal != 0);
    }

// snip //


    // Give the afLib state machine some time.
    af_lib_loop(af_lib);
```

```
}
// snip //
```

Notes:

- In all examples, the author compares the return value from `af_lib_set_attribute()` to `AF_SUCCESS`.
- The first two instances occur within the scope of the `attrNotifyHandler()` callback. You should avoid calling `afLib->loop()` within such handlers; these examples show two ways of handling potential failures of a `af_lib_set_attribute()` call within a callback:
  - In the first example, a non-zero return value simply triggers some debugging output. This is suitable for situations in which you want to be warned of, but can tolerate, failing `af_lib_set_attribute()` calls.
  - In the second example, we set a global variable to indicate a `af_lib_set_attribute()` call is required, and then handle that in the main `loop()`, retrying as needed until success. This pattern works well if successful execution of the command is critical.
- In the third instance, the `af_lib_set_attribute()` call retries, waiting for `AF_SUCCESS`. There is a timeout, so that re-trying does not go on indefinitely in the face of some fatal error; if the timeout is exceeded, we set the global variable requesting a reboot, and stop trying. Note also that the author calls `af_lib_loop()` within the re-try loop, to ensure that afLib2 is getting adequate processing time regardless of any unexpected delays. This pattern is robust, but again: do not use this pattern within afLib2 callbacks.

Handle Reboot Requests

Your MCU code will often use the `attrNotifyHandler()` callback to take action when an attribute value has been changed by a call to `af_lib_set_attribute()`. Your primary concern will be changes to attributes you define, but your code must also watch for changes to the system attribute AF_SYSTEM_ASR_STATE. This attribute can have one of four values:

0 = Rebooted

1 = Linked

2 = Updating

3 = Update Ready to Apply (Reboot Requested)

AF_SYSTEM_ASR_STATE will have value 3 (Reboot Requested) whenever ASR receives an Over-the-Air (OTA) firmware update. When this happens, ASR sends an update message with that attribute ID and value, and your MCU code is responsible for recognizing this case and triggering a reboot. You code should do this calling `af_lib_set_attribute()` for the attribute AF_SYSTEM_COMMAND, with value 1. Here's a snippet:

```
#define AF_MODULE_STATE_REBOOTED       0
#define AF_MODULE_STATE_LINKED         1
#define AF_MODULE_STATE_UPDATING       2
#define AF_MODULE_STATE_UPDATE_READY   3

#define AF_MODULE_COMMAND_REBOOT       1

void attrNotifyHandler(const uint8_t requestId, const uint16_t attributeId, const uint16_t
valueLen, const uint8_t *value) {

    switch (attributeId) {
        // snip //

        case AF_SYSTEM_ASR_STATE:
            Serial.print("ASR state: ");
            switch (value[0]) {
                case AF_MODULE_STATE_REBOOTED:
                    Serial.println("Rebooted");
                    break;

                case AF_MODULE_STATE_LINKED:
                    Serial.println("Linked");
                    break;

                case AF_MODULE_STATE_UPDATING:
                    Serial.println("Updating");
                    break;

                case AF_MODULE_STATE_UPDATE_READY:
                    Serial.println("Update ready - rebooting");
                    af_lib_set_attribute_32(af_lib, AF_SYSTEM_COMMAND,
AF_MODULE_COMMAND_REBOOT);
```

```
        }
                    break;

                default:
                    break;
            }
            break;

        default:
            break;
    }
}
```

Our `attrNotifyHandler()` checks the supplied attribute ID, looking for AF_SYSTEM_ASR_STATE. If we have received an update message for that attribute ID, we check the attribute value. For values 0-2, this example just prints debug information, but if the value is 3 (AF_MODULE_STATE_UPDATE_READY), then we trigger a reboot of ASR by calling `af_lib_set_attribute_32()` for the AF_SYSTEM_COMMAND attribute, with value AF_MODULE_COMMAND_REBOOT. Note that in this case, the reboot was triggered as soon as requested, but in a more complicated project you might wait until you've completed an ongoing operation, or are in some idle state.

Useful Debugging Methods

This isn't a tip so much as a couple of useful methods that you can copy/paste into your code. You can see these methods used in the afBlink example.

```
#define ATTR_PRINT_HEADER_LEN     60
#define ATTR_PRINT_MAX_VALUE_LEN  512   // Each byte is 2 ASCII characters in HEX.
#define ATTR_PRINT_BUFFER_LEN     (ATTR_PRINT_HEADER_LEN + ATTR_PRINT_MAX_VALUE_LEN)

char attr_print_buffer[ATTR_PRINT_BUFFER_LEN];

void getPrintAttrHeader(const char *sourceLabel, const char *attrLabel, const uint16_t
attributeId, const uint16_t valueLen) {
    memset(attr_print_buffer, 0, ATTR_PRINT_BUFFER_LEN);
    snprintf(attr_print_buffer, ATTR_PRINT_BUFFER_LEN, "%s id: %s len: %05d value: ",
sourceLabel, attrLabel, valueLen);
}
```

```
void printAttrBool(const char *sourceLabel, const char *attrLabel, const uint16_t attributeId,
const uint16_t valueLen, const uint8_t *value) {
    getPrintAttrHeader(sourceLabel, attrLabel, attributeId, valueLen);
    if (valueLen > 0) {
        strcat(attr_print_buffer, *value == 1 ? "true" : "false");
    }
    af_logger_println_buffer(attr_print_buffer);
}

void printAttr8(const char *sourceLabel, const char *attrLabel, const uint8_t attributeId,
const uint16_t valueLen, const uint8_t *value) {
    getPrintAttrHeader(sourceLabel, attrLabel, attributeId, valueLen);
    if (valueLen > 0) {
        char intStr[6];
        strcat(attr_print_buffer, itoa(*((int8_t *)value), intStr, 10));
    }
    af_logger_println_buffer(attr_print_buffer);
}

void printAttr16(const char *sourceLabel, const char *attrLabel, const uint16_t attributeId,
const uint16_t valueLen, const uint8_t *value) {
    getPrintAttrHeader(sourceLabel, attrLabel, attributeId, valueLen);
    if (valueLen > 0) {
        char intStr[6];
        strcat(attr_print_buffer, itoa(*((int16_t *)value), intStr, 10));
    }
    af_logger_println_buffer(attr_print_buffer);
}

void printAttr32(const char *sourceLabel, const char *attrLabel, const uint16_t attributeId,
const uint16_t valueLen, const uint8_t *value) {
    getPrintAttrHeader(sourceLabel, attrLabel, attributeId, valueLen);
    if (valueLen > 0) {
        char intStr[11];
        strcat(attr_print_buffer, itoa(*((int32_t *)value), intStr, 10));
    }
    af_logger_println_buffer(attr_print_buffer);
}

void printAttrHex(const char *sourceLabel, const char *attrLabel, const uint16_t attributeId,
const uint16_t valueLen, const uint8_t *value) {
    getPrintAttrHeader(sourceLabel, attrLabel, attributeId, valueLen);
    for (int i = 0; i < valueLen; i++) {
        strcat(attr_print_buffer, String(value[i], HEX).c_str());
    }
```

```
    af_logger_println_buffer(attr_print_buffer);
}

void printAttrStr(const char *sourceLabel, const char *attrLabel, const uint16_t attributeId,
const uint16_t valueLen, const uint8_t *value) {
    getPrintAttrHeader(sourceLabel, attrLabel, attributeId, valueLen);
    int len = strlen(attr_print_buffer);
    for (int i = 0; i < valueLen; i++) {
        attr_print_buffer[len + i] = (char)value[i];
    }
    af_logger_println_buffer(attr_print_buffer);
}


void printAttribute(const char *label, const uint16_t attributeId, const uint16_t valueLen,
const uint8_t *value) {
    switch (attributeId) {
        case AF_BLINK:
            printAttrBool(label, "AF_BLINK", attributeId, valueLen, value);
            break;

        case AF_MODULO_LED:
            printAttr16(label, "AF_MODULO_LED", attributeId, valueLen, value);
            break;

        case AF_GPIO_0_CONFIGURATION:
            printAttrHex(label, "AF_GPIO_0_CONFIGURATION", attributeId, valueLen, value);
            break;

        case AF_MODULO_BUTTON:
            printAttr16(label, "AF_MODULO_BUTTON", attributeId, valueLen, value);
            break;

        case AF_GPIO_3_CONFIGURATION:
            printAttrHex(label, "AF_GPIO_3_CONFIGURATION", attributeId, valueLen, value);
            break;

        case AF_BOOTLOADER_VERSION:
            printAttrHex(label, "AF_BOOTLOADER_VERSION", attributeId, valueLen, value);
            break;

        case AF_SOFTDEVICE_VERSION:
            printAttrHex(label, "AF_SOFTDEVICE_VERSION", attributeId, valueLen, value);
            break;
```

```
        case AF_APPLICATION_VERSION:
            printAttrHex(label, "AF_APPLICATION_VERSION", attributeId, valueLen, value);
            break;

        case AF_PROFILE_VERSION:
            printAttrHex(label, "AF_PROFILE_VERSION", attributeId, valueLen, value);
            break;

        case AF_SYSTEM_ASR_STATE:
            printAttr8(label, "AF_SYSTEM_ASR_STATE", attributeId, valueLen, value);
            break;

        case AF_SYSTEM_LOW_POWER_WARN:
            printAttr8(label, "AF_ATTRIBUTE_LOW_POWER_WARN", attributeId, valueLen, value);
            break;

        case AF_SYSTEM_REBOOT_REASON:
            printAttrStr(label, "AF_REBOOT_REASON", attributeId, valueLen, value);
            break;

        case AF_SYSTEM_MCU_INTERFACE:
            printAttr8(label, "AF_SYSTEM_MCU_INTERFACE", attributeId, valueLen, value);
            break;

        case AF_SYSTEM_LINKED_TIMESTAMP:
            printAttr32(label, "AF_SYSTEM_LINKED_TIMESTAMP", attributeId, valueLen, value);
            break;
    }
}
```

Here's a brief example usage snippet:

```
void attrNotifyHandler(const uint8_t requestId, const uint16_t attributeId, const uint16_t
valueLen, const uint8_t *value) {
    printAttribute("attrNotifyHandler", attributeId, valueLen, value);
    // snip //
}
```

Watch Your Memory Usage

If you've worked much with Arduino, this tip is probably not news to you: Memory is limited, and if your MCU code exceeds available memory, your output can be quite puzzling. We have often encountered bizarre – but seemingly error-free – behavior from a sketch running on Uno, only to find that the same code runs entirely as expected on Teensy. This is particularly useful to remember in the context of the Useful Debugging Methods provided above; use of those methods requires significant memory, and will be difficult using Uno.

See Also

Tutorial - Lesson 3

Next: *Setting Time on the MCU*

Updated June 6, 2018