AE483 Lab Manual: Appendix D
How to Change What is Logged

T. Bretl

September 9, 2019

1 The big picture

The drone (on-board code) sends data to the ground-station (off-board code) wirelessly, where it is logged (i.e., written to a file on disk). This appendix shows how to modify the on-board code and the off-board code in order to change what is sent and what is logged.

2 Change the on-board code

2.1 Working with global variables that are defined by default

We assume that you have already created an Eclipse project and already know how to compile and run the on-board code, starting from `AscTec_SDK_v3.0_LabXX.zip` (see Appendix A). Open the file `main.c`. In this file, near line 351, you should see the start of a function called `ACISDK()`. The code in this function determines what variables are being sent from the drone to the ground-station. In particular, you should see six calls to a function `aciPublishVariable` that each look like this:

```
aciPublishVariable(
    &RO_ALL_Data.angvel_pitch,      // pointer to global variable
    VARTYPE_INT32,                  // type
    0x0200,                         // index (must be unique!)
    "angvel_pitch",                 // name
    "Pitch angular velocity",       // description
    "0.0154 degree/s, bias free"    // units
);
```

Consider each argument:

- The first argument, `&RO_ALL_Data.angvel_pitch`, is a pointer to a global variable whose value will be sent, in this case a variable with the drone's angular velocity about the body-fixed y-axis (i.e., "pitch axis"). `RO_ALL_Data` is a `struct` with a bunch of useful variables that are defined by default.[1] You can find the definition of `RO_ALL_Data` in `sdk.h` on line 54. In particular, look at lines 94-107:

[1] https://en.wikipedia.org/wiki/Struct_(C_programming_language)

```
94  //angles derived by integration of gyro_outputs, drift compensated by data
        fusion; -90000..+90000 pitch(nick) and roll, 0..360000 yaw; 1000 = 1
        degree
95  int angle_pitch;
96  int angle_roll;
97  int angle_yaw;
98
99  //angular velocities, bias free, in 0.0154 deg/s (=> 64.8 = 1 deg/s)
100 int angvel_pitch;
101 int angvel_roll;
102 int angvel_yaw;
103
104 //acc-sensor outputs, calibrated: -10000..+10000 = -1g..+1g, body frame
        coordinate system
105 short acc_x;
106 short acc_y;
107 short acc_z;
```

`RO_ALL_Data.angvel_pitch` is the variable in this `struct` that has the angular velocity about the pitch axis. Prepending this with "`&`" gets you a pointer to this variable.[2] Replacing this first argument with `&RO_ALL_DATA.angle_pitch` would allow you to publish (a.k.a. "send") the pitch angle instead of the pitch angular velocity, and so forth.

- The second argument, `VARTYPE_INT32`, encodes the variable "type." `RO_ALL_Data.angvel_pitch` is an `int`. By default, an "`int`" in C is an integer that is stored in memory with 32 bits. ACI uses the macro `VARTYPE_INT32` to encode this information. Here is a list of common C variable types,[3] saying how to encode them in the second argument to `aciPublishVariable`:

To send a variable of this type...	...use this second argument to `aciPublishVariable`
`unsigned char`	`VARTYPE_UINT8`
`int`	`VARTYPE_INT32`
`short`	`VARTYPE_INT16`
`unsigned int`	`VARTYPE_UINT32`
`unsigned short`	`VARTYPE_UINT16`

Although it is possible to send floating-point variables, it is recommended that you stick to the integer variables that are in this list (e.g., converting from `float` to `int` before sending, and converting back after receiving). You can find a list of all types in `asctecDefinesOnboard.h`.

- The third argument, `0x0200`, is an index—in hexadecimal format[4]—that is used to identify the variable being sent. If the ground station sees data marked with `0x0200`, it will know that these data represent the pitch angular velocity. If the ground station sees data marked with `0x0201`, for example, it will know that these data represent something else. You get to choose this index yourself. The important thing is that it be unique. **You must not call `aciPublishVariable` twice with the same index!**

[2] https://www.geeksforgeeks.org/pointers-in-c-and-c-set-1-introduction-arithmetic-and-array/
[3] https://en.wikipedia.org/wiki/C_data_types
[4] https://en.wikipedia.org/wiki/Hexadecimal

- The fourth, fifth, and six arguments provide a name (`"angvel_pitch"`), a description (`"Pitch angular velocity"`), and a set of units (`"0.0154 degree/s, bias free"`) for the variables being sent. These arguments are there only for your own convenience—you can change them as you like, whatever is the most useful for you. (They will show up in the ACI Tool, for example, which is helpful when you are setting up your network configuration.)

There is one call to `aciPublishVariable` for each variable being sent. If you want to send a new variable, add a call to `aciPublishVariable`. If you want to stop sending a variable, remove the corresponding call to `aciPublishVariable`. For example, suppose you want to send **only** the pitch angle and the pitch angular velocity, nothing else. Then, modify `ACISDK` in `main.c` as follows:

```
355 #ifndef MATLAB
356     aciSetStartTxCallback(UARTWriteChar);
357 
358     // Variables
359     aciPublishVariable(&RO_ALL_Data.angvel_pitch, VARTYPE_INT32, 0x0200, "
        angvel_pitch", "Pitch angular velocity", "0.0154 degree/s, bias free");
360     aciPublishVariable(&RO_ALL_Data.angle_pitch, VARTYPE_INT32, 0x0201, "
        angle_pitch", "Pitch angle", "0.001 degree");
361 
362     // Commands
363 
364     // Parameters
365 
366 #else
```

2.2 Working with global variables that you define

The previous section showed how change the on-board code to send (or stop sending) global variables that are defined by default, e.g., as members of the `struct RO_ALL_DATA` in `sdk.h`. This section shows how to send global variables that you define. Take a look at `lab.c`:

```
8  #include "lab.h"
9  #include "main.h"
10 #include "sdk.h"
11 #include "math.h"
12 
13 /*———————— Global Variables ————————*/
14 
15 
16 /*———————— Main Control Loop ————————*/
17 void lab(void) {
18 
19 
20 
21 }
```

The function `lab()` gets called by the on-board code at a fixed rate (eventually, this is where you will implement your controller). Suppose you want to count the total number of times it has been

called, and to send this information from the drone to the ground-station. To count the number of times `lab()` has been called in `lab.c`, you would modify `lab.c` as follows:

```
8  #include "lab.h"
9  #include "main.h"
10 #include "sdk.h"
11 #include "math.h"
12
13 /*--------- Global Variables ----------*/
14 unsigned int counter = 0;       // define counter as an unsigned integer (i.e.,
15                                 // an integer that is always positive) with
16                                 // initial value zero
17
18 /*--------- Main Control Loop ---------*/
19 void lab(void) {
20     counter++;  // increment counter by one each time lab() is called - note
21                 // that this is equivalent to saying:
22                 //
23                 //  counter = counter + 1;
24                 //
25 }
```

To send this information to the ground station, you would add a call to `aciPublishVariable` in the function `ACISDK()` of `main.c`, just as in the previous section—maybe something like this, if you wanted to send **only** the pitch angle, the pitch angular velocity, and the counter:

```
355 #ifndef MATLAB
356     aciSetStartTxCallback(UARTWriteChar);
357
358     // Variables
359     aciPublishVariable(&RO_ALL_Data.angvel_pitch, VARTYPE_INT32, 0x0200, "
    angvel_pitch", "Pitch angular velocity", "0.0154 degree/s, bias free");
360     aciPublishVariable(&RO_ALL_Data.angle_pitch, VARTYPE_INT32, 0x0201, "
    angle_pitch", "Pitch angle", "0.001 degree");
361     aciPublishVariable(&counter, VARTYPE_UINT32, 0x0202, "counter", "Count of
    calls to lab()", "one per call");
362
363     // Commands
364
365     // Parameters
366
367 #else
```

Sadly, you will see an error if you try to compile this code. The reason is that, in C, global variables are (by default) specific to the source file in which they are defined. You defined `counter` in `lab.c`, so (by default) this variable can only be used in `lab.c`. To use this variable in `main.c` as well, you need to add one more line to `main.c`, near the top where you see `Global Variables`:

```
96 // HERE IS WHERE YOU ADD MORE GLOBAL VAIRABLES TO WRITE
97 extern unsigned int counter;
```

The keyword `extern` tells the compiler that `counter` is defined elsewhere.[5] It is important that the two definitions match: `extern unsigned int counter` in `main.c` and `unsigned int counter` in `lab.c`, for example.

2.3 Compile and flash

Do not forget to compile your on-board code (fixing any errors that arise) and to flash your executable to the drone after you are done changing it.

3 Change the network configuration

After you change the on-board code, you should run the ACI Tool and configure the network to receive all of the variables that you chose to send—this is an important check before proceeding to modify the ground-station.

4 Change the off-board code

We assume that you have already created a Visual Studio 2010 project and already know how to compile and run the off-board code, starting from `AE483GroundStation_LabXX.zip` (see Appendix C). Continuing our example from Section 2, suppose we want to modify the off-board code to receive and log the pitch angle, the pitch angular velocity, and the counter that we defined, along with all of the usual data from the motion capture system. Then, we would need to make the following changes to `main.c`:

- Define variables to receive data from the drone. Replace this code (near line 109)...

```
109 // Variables for receiving ACI data sent from on-board over XBee
110 int16_t accel_x;
111 int16_t accel_y;
112 int16_t accel_z;
113 int32_t gyro_x;
114 int32_t gyro_y;
115 int32_t gyro_z;
```

...with this code:

```
109 // Variables for receiving ACI data sent from on-board over XBee
110 int32_t gyro_y;
111 int32_t angle_pitch;
112 uint32_t onboard_counter;
```

[5] https://en.wikipedia.org/wiki/External_variable

Note the use of `int32_t` and `uint32_t` to define the variable types.[6] The point of doing this is to make really sure that the `int` and `unsigned int` we are defining here are stored in memory with 32 bits. We didn't have to do that in the on-board code, because that's running on known hardware. The off-board code could be running on anything, perhaps a computer with a different convention for how many bits are used to represent standard types.

The variables names are entirely our choice—we just need to be consistent throughout `main.c`. (We do **not** need to be consistent with the on-board code—the on-board and off-board code could use completely different names for things, so long as the hexadecimal index associated with each variable is the same.)

- Say what data should go into what variables. Replace this code (near line 338)...

```
338     // Define packet structure: aciAddContentToVarPacket(packet number, id,
        pointer to variable);
339     aciAddContentToVarPacket(1, 0x0203, &accel_x);
340     aciAddContentToVarPacket(1, 0x0204, &accel_y);
341     aciAddContentToVarPacket(1, 0x0205, &accel_z);
342     aciAddContentToVarPacket(1, 0x0201, &gyro_x);
343     aciAddContentToVarPacket(1, 0x0200, &gyro_y);
344     aciAddContentToVarPacket(1, 0x0202, &gyro_z);
```

...with this code:

```
338     // Define packet structure: aciAddContentToVarPacket(packet number, id,
        pointer to variable);
339     aciAddContentToVarPacket(1, 0x0200, &gyro_y);
340     aciAddContentToVarPacket(1, 0x0201, &angle_pitch);
341     aciAddContentToVarPacket(1, 0x0202, &onboard_counter);
```

This is the analog to `aciPublishVariable` in the on-board code (Section 2). For every variable you want to receive, you need a call to `aciAddContentToVarPacket`. The first argument is always "1" (it is the "packet number" that you set with the ACI tool). The second argument is the unique hexadecimal index that you associated with each variable. The third argument is a pointer to a global variable into which data will be copied.

- Change the header of the data file. Replace this code (near line 923)...

```
923 fprintf(FDR, "time (s), accel_x (m/s^2), accel_y (m/s^2), accel_z (m/s^2),
        gyro_x (rad/s), gyro_y (rad/s), gyro_z (rad/s), x (m), y (m), z (m), yaw
        (rad), pitch (rad), roll (rad)\n");
```

...with this code:

[6] https://www.gnu.org/software/libc/manual/html_node/Integers.html

```
923 fprintf(FDR, "time (s), gyro_y (rad/s), imu_pitch (rad), counter, x (m), y (m
        ), z (m), yaw (rad), pitch (rad), roll (rad)\n");
```

This is simply a comma-delimited list of the labels of all the variables you will write to the `.csv` file. Recall that `x`, `y`, `z`, `yaw`, `pitch`, and `roll` are all going to come from the motion capture system, and that `t` is a timestamp that is generated by the off-board code. The other variables come wirelessly from the drone. The labels can be whatever you want—it's important only that the number of them be consistent with the number of variables you will actually write, and that they be in the same order. In particular, note that we have chosen to use `imu_pitch` as the label for the pitch angle, and `counter` as the label for the counter, even though these things are stored in the global variables `angle_pitch` and `onboard_counter`.

- Change the format of the data file. Replace this code (near line 944)...

```
944 fprintf(FDR, "%0.4f, %0.4f, %0.4f, %0.4f, %0.4f, %0.4f, %0.4f, %0.4f, %0.4f,
        %0.4f, %0.4f, %0.4f, %0.4f\n",
945         timer,
946         ((float)accel_x)*9.80665 / 10000,
947         ((float)accel_y)*9.80665 / 10000,
948         ((float)accel_z)*9.80665 / 10000,
949         ((float)gyro_x)*0.0154*PI_f / 180.0,
950         ((float)gyro_y)*0.0154*PI_f / 180.0,
951         ((float)gyro_z)*0.0154*PI_f / 180.0,
952         QuadCurrent.Pos.x,
953         QuadCurrent.Pos.y,
954         QuadCurrent.Pos.z,
955         QuadCurrent.Ori.yaw,
956         QuadCurrent.Ori.pitch,
957         QuadCurrent.Ori.roll);
```

...with this code:

```
944 fprintf(FDR, "%0.4f, %0.4f, %0.4f, %d, %0.4f, %0.4f, %0.4f, %0.4f, %0.4f,
        %0.4f\n",
945         timer,
946         ((float)gyro_y)*0.0154*PI_f / 180.0,
947         ((float)angle_pitch)*0.001*PI_f / 180.0,
948         onboard_counter,
949         QuadCurrent.Pos.x,
950         QuadCurrent.Pos.y,
951         QuadCurrent.Pos.z,
952         QuadCurrent.Ori.yaw,
953         QuadCurrent.Ori.pitch,
954         QuadCurrent.Ori.roll);
```

This is an example of C string formatting.[7][8][9], etc. For us, the important rules are:

[7] https://www.gnu.org/software/libc/manual/html_node/Formatted-Output.html
[8] https://www.cprogramming.com/tutorial/printf-format-strings.html
[9] https://www.cypress.com/file/54441/download

- include one "`%`" tag for every variable you want to log;
- use "`%0.4f`" for floating-point numbers and `%d` for integers;
- do **not** get rid of "`\n`" (it is equivalent to hitting return at the end of a line of text);
- remember to convert variables, if necessary, from integers to floating-point numbers—for example, the pitch angle was an integer in units of 0.001 deg. To convert this to radians and print it as a floating-point number, we cast it as a `float`[10], multiply it by `0.001` to get degrees, then multiply it by `PI_f / 180.0` to get radians (where `PI_f` is defined in `planner.h`, just to be helpful). It is easy to get these conversions wrong, so be careful!

Note that it made no difference whether we were dealing with variables that were defined by default in the on-board code, and variables that we defined ourselves—in the off-board code, these two things are treated exactly the same way. At this point, you should be able to compile and run the ground-station, and to check that the resulting data file includes all of the variables that we wanted to log.

[10] https://en.wikipedia.org/wiki/Type_conversion