Piksi Integration Tutorial

This is a tutorial on how to integrate Piksi with a host system, receive SwiftNav Binary Protocol (SBP) messages from Piksi and decode position outputs.

We use an STM32F4 DISCOVERY Board (http://www.st.com/web/catalog/tools/FM116/SC959/SS1532/PF252419) as an example of a host system. We use the official STM toolchain and libraries to keep things as simple as possible. We use the CooCox IDE on Windows - you must use this if you wish to build the project in the Git repository without making any changes.

If anything in this guide is incorrect or unclear, please email our mailing list (https://groups.google.com/forum/#!forum/swiftnav-discuss).

Contents

Before you start

Please review the SwiftNav Binary Protocol documentation to familiarize yourself with the protocol.

> SwiftNav Binary Protocol

We also recommend a full run through of the Piksi User Getting Started Guide before starting work on integration.

> Piksi User Getting Started Guide

The Piksi Console will allow you to set Piksi in Simulation Mode, which will cause it to send out SBP messages for a simulated position solution. We recommend using Simulation Mode while working on SBP integration, as this will be far easier than sitting outside with your Piksi so you can get a real position fix. See the following for more info:

> Piksi User Getting Started Guide#Simulation Mode

Explanation of tutorial code

The code run on the STM32F4 DISCOVERY Board in this tutorial can be found at:

> https://github.com/swift-nav/sbp_tutorial.git

The SBP tutorial code consists of main.c and tutorial_implementation.c/tutorial_implementation.h.

The source files tutorial_implementation.c/tutorial_implementation.h contain implementation specific functions for setting up and interacting with peripherals on the STM32F4 DISCOVERY Board microcontroller.

The source file main.c contains the calls and instantiations of the SBP submodule that you will need to replicate in your code. These are documented in detail at:

> http://docs.swift-nav.com/libsbp/

Receiving bytes from Piksi

Bytes received from Piksi via USART1 are stored in a FIFO structure. This allows the host to receive multiple bytes from Piksi in between the SBP submodule processing them, without losing bytes. The following code is an interrupt that is called when the host's USART receives a byte from Piksi (in tutorial_implementation.c). It appends the received byte to the FIFO, toggles the LEDs once every 250 bytes received, and clears the interrupt.

```
void USART1_IRQHandler(void)
{
  fifo_write(USART1->DR);
  DO_EVERY(250,
    leds_toggle();
  );
  USART1->SR &= ~(USART_FLAG_RXNE);
}
```

Instantiating SBP structs

An `sbp_state_t` must be statically instantiated to keep the state of the SBP message parser. In this example we statically instantiate a matching message struct (`sbp_foo_t`) for each message type we want to receive from Piksi and decode. Each message type you wish to receive and decode must also have its own unique, statically allocated `sbp_msg_callbacks_node_t`. The `sbp_msg_callbacks_node_t` structs associate a specific message ID with a callback function - the SBP message parser searches through these to find the correct callback function for a specific message ID.

```
/*
 * State of the SBP message parser.
 * Must be statically allocated.
 */
sbp_state_t sbp_state;

/* SBP structs that messages from Piksi will feed. */
msg_pos_llh_t      pos_llh;
msg_baseline_ned_t baseline_ned;
msg_vel_ned_t      vel_ned;
msg_dops_t         dops;
msg_gps_time_t     gps_time;

/*
 * SBP callback nodes must be statically allocated. Each message ID / callback
 * pair must have a unique sbp_msg_callbacks_node_t associated with it.
 */
sbp_msg_callbacks_node_t pos_llh_node;
sbp_msg_callbacks_node_t baseline_ned_node;
sbp_msg_callbacks_node_t vel_ned_node;
sbp_msg_callbacks_node_t dops_node;
sbp_msg_callbacks_node_t gps_time_node;
```

Defining SBP callback functions

Every SBP message you wish to receive and decode must have a callback function associated with it. When the SBP message parser parses a valid SBP message, it searches through its list of `sbp_msg_callback_node_t`'s to find a callback function associated with the message's ID. It then passes the data associated with the message to the callback function. The callback function then interprets the

data. In the tutorial code, the callback functions are simple - they just cast the data from the SBP message to the associated message struct type, and assign it to the instantiated message struct. They can also implement more complex functionality, such as triggering events when a particular message is received.

```
/*
 * Callback functions to interpret SBP messages.
 * Every message ID has a callback associated with it to
 * receive and interpret the message payload.
 */
void sbp_pos_llh_callback(u16 sender_id, u8 len, u8 msg[], void *context)
{
  pos_llh = *(msg_pos_llh_t *)msg;
}
void sbp_baseline_ned_callback(u16 sender_id, u8 len, u8 msg[], void *context)
{
  baseline_ned = *(msg_baseline_ned_t *)msg;
}
void sbp_vel_ned_callback(u16 sender_id, u8 len, u8 msg[], void *context)
{
  vel_ned = *(msg_vel_ned_t *)msg;
}
void sbp_dops_callback(u16 sender_id, u8 len, u8 msg[], void *context)
{
  dops = *(msg_dops_t *)msg;
}
void sbp_gps_time_callback(u16 sender_id, u8 len, u8 msg[], void *context)
{
  gps_time = *(msg_gps_time_t *)msg;
}
```

Initializing SBP parser and registering callbacks

The SBP parser must be initialized via a call to `sbp_state_init` before any SBP messages can be parsed. Each message ID / callback pair must be associated with an `sbp_msg_callback_node_t` and registered with the SBP parser via a call to `sbp_register_callback`. When the SBP parser successfully parses an SBP message it will search through the list of `sbp_msg_callback_node_t`'s that have been registered with it to find the correct callback for the message's ID.

```
void sbp_setup(void)
{
  /* SBP parser state must be initialized before sbp_process is called. */
  sbp_state_init(&sbp_state);

  /* Register a node and callback, and associate them with a specific message ID. */
  sbp_register_callback(&sbp_state, SBP_MSG_GPS_TIME, &sbp_gps_time_callback,
                        NULL, &gps_time_node);
  sbp_register_callback(&sbp_state, SBP_MSG_POS_LLH, &sbp_pos_llh_callback,
                        NULL, &pos_llh_node);
  sbp_register_callback(&sbp_state, SBP_MSG_BASELINE_NED, &sbp_baseline_ned_callback,
                        NULL, &baseline_ned_node);
  sbp_register_callback(&sbp_state, SBP_MSG_VEL_NED, &sbp_vel_ned_callback,
                        NULL, &vel_ned_node);
  sbp_register_callback(&sbp_state, SBP_MSG_DOPS, &sbp_dops_callback,
                        NULL, &dops_node);
}
```

Parsing SBP messages

The function `sbp_process` must be called periodically in your program to parse the received bytes from Piksi. When it parses a valid SBP message and a callback function has been registered for this message's ID, it will call this function, passing it the data from the message. A returned value of less than zero indicates there was an error.

```
while (1) {
  s8 ret = sbp_process(&sbp_state, &fifo_read);
  if (ret < 0)
    printf("sbp_process error: %d\n", (int)ret);
}
```

`sbp_process` must be passed a pointer to a function that provides access to the received bytes from Piksi, that conforms to the definition:

```
u32 get_bytes(u8 *buff, u32 n, void *context)
```

Below is the function used in the tutorial, which reads bytes out of the FIFO:

```
u32 fifo_read(u8 *buff, u32 n, void *context) {
  int i;
  for (i=0; i<n; i++)
    if (!fifo_read_char((char *)(buff + i)))
      break;
  return i;
}
```

Installing tools

To build the code used in this tutorial, you must install the CooCox IDE, ARM-GCC, ST-Link, and configure CooCox IDE to use ARM-GCC.

CooCox IDE

Download and install the CooCox IDE (http://www.coocox.org/CoIDE/CoIDE_Updates.htm).

ARM-GCC

Download and install ARM-GCC (https://launchpad.net/gcc-arm-embedded/).

ST-Link

Download and install ST-Link (http://www.st.com/web/en/catalog/tools/PF258168).

Configure CooCox IDE to use ARM-GCC

After launching CooCox IDE, select Project->Select Toolchain Path.

Enter the path to the ARM-GCC binary, for example: `C:\Program Files\GNU Tools ARM Embedded\4.8 2014q1\bin`

Make sure to check what the actual path is on your system rather than just copying the above.

Setting up the hardware

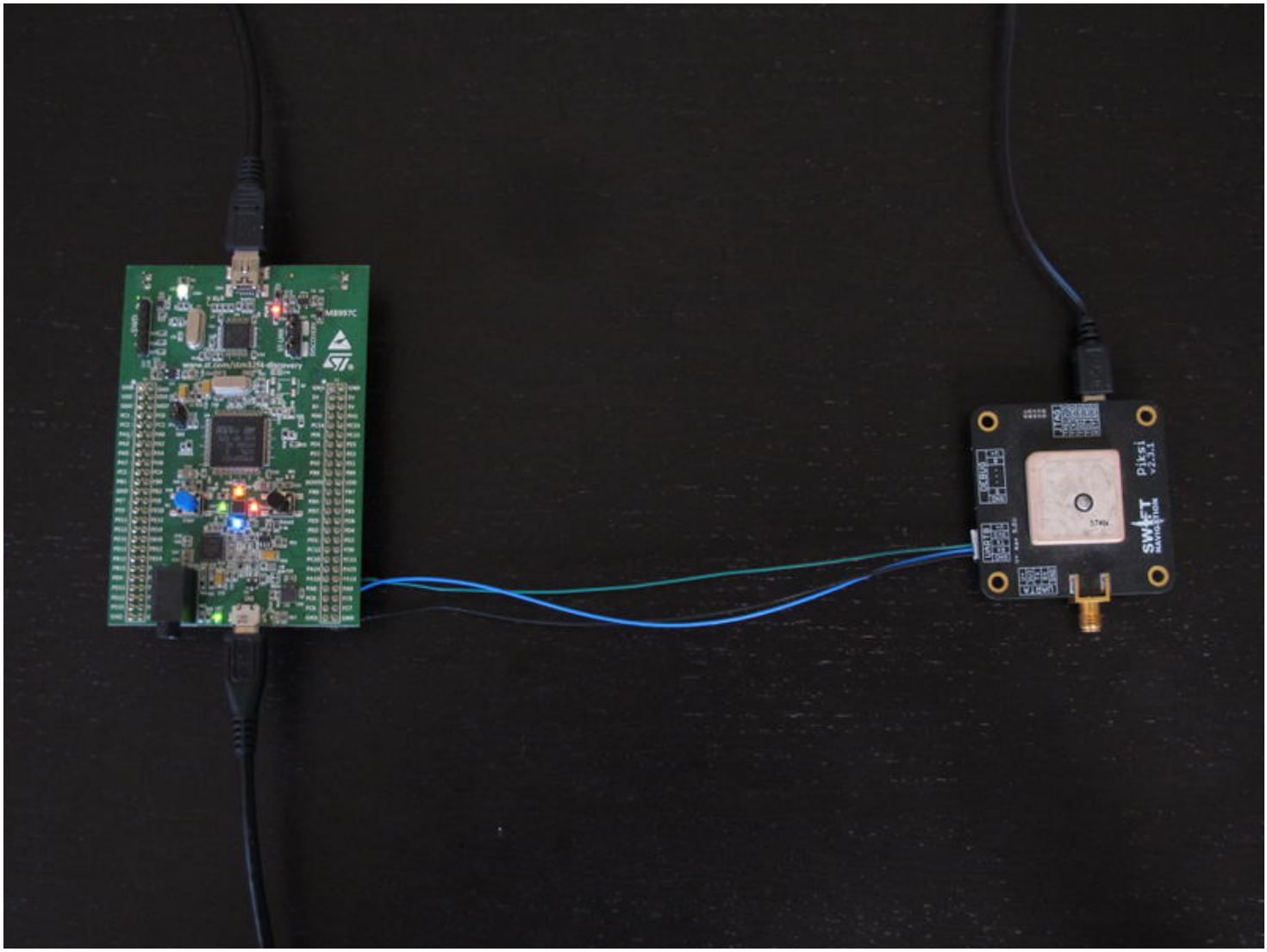
The following photo shows the test setup for the integration tutorial.

One of the pigtail Picoblade cables that ships with the Piksi RTK Kit (http://store.swift-nav.com/) is used to connect to one of the STM32F4 Discovery Board's USARTs.

The wiring for the Piksi USART to the STM32F4 Discovery Board USART is as follows:

STM32F4 Discovery Board Pin	Piksi UART B Pin
PA9 (USART1 TX)	UART Pin 2 (RX)
PA10 (USART1 RX)	UART Pin 3 (TX)
GND	UART Pin 1 (GND)

Two USB cables, a micro and a mini, are required to connect to the STM32F4 Discovery Board. A micro-USB cable is connected to Piksi. All three cables should be plugged into your PC.



Running the tutorial

Now that you have the hardware set up, we can run the tutorial.

Clone the tutorial git repository.

```
$ git clone https://github.com/swift-nav/sbp_tutorial.git
```

You'll also need to clone the libsbp submodule and update in order to build the project.

```
$ cd sbp_tutorial
$ git submodule init
$ git submodule update
```

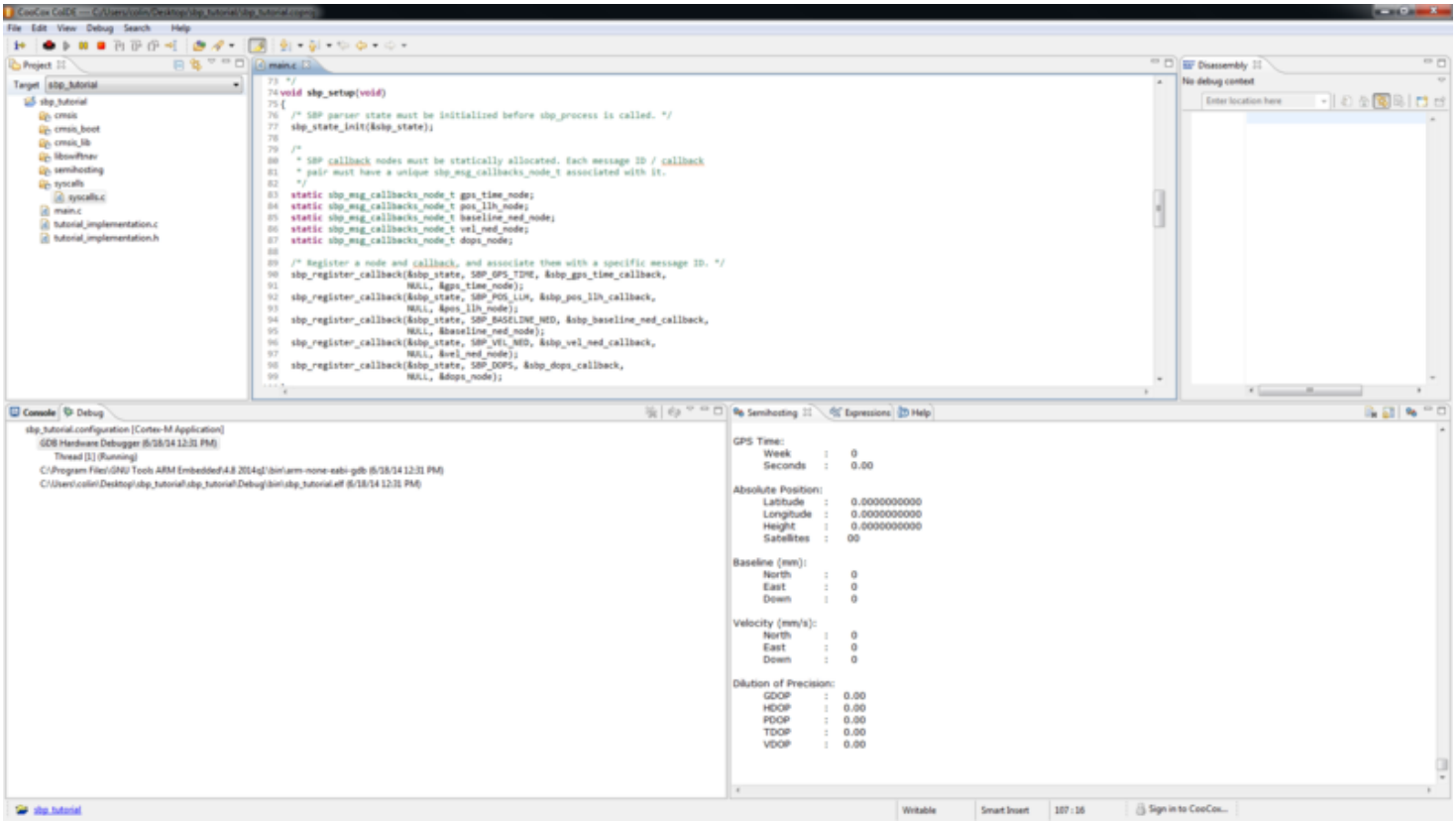
Open the CooCox IDE project file in the sbp_tutorial folder and build the firmware:

Project -> Build

Run the project via the Debug tab:

Debug -> Debug
Debug -> Run

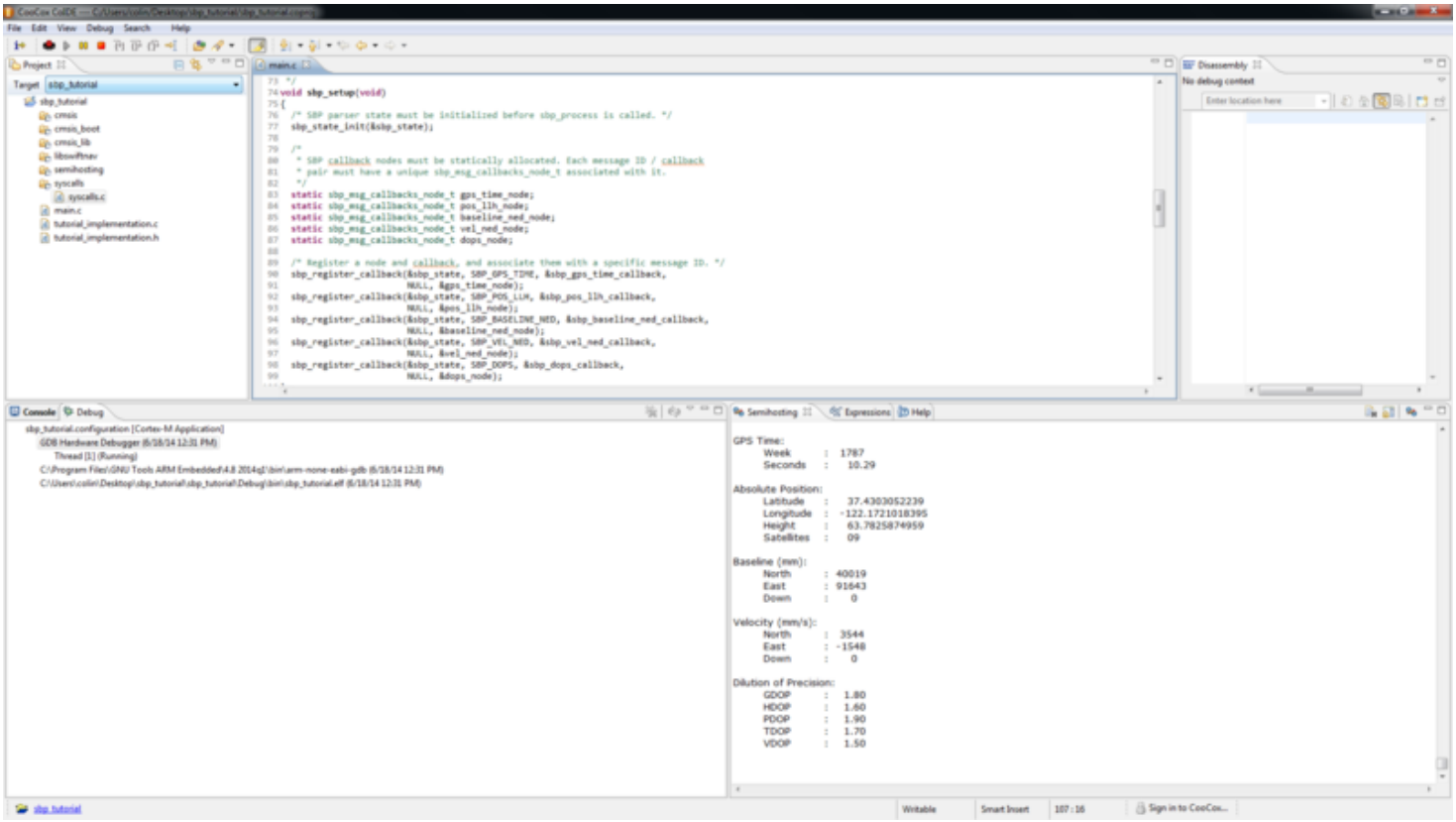
You should now see output being printed out through the Semihosting tab in the CooCox IDE, with all the fields being zero, as shown below.



Set Piksi into simulation mode, as outlined in the instructions in the User's Guide:

> Piksi User Getting Started Guide#Simulation_Mode

Now simulated output should be printed out in the Semihosting tab. If the STM32F4 Discovery Board is receiving data from Piksi over the UART, the four colored LEDs should be blinking.



Retrieved from "http://docs.swiftnav.com/w/index.php?title=Piksi_Integration_Tutorial&oldid=20348"

This page was last edited on 14 April 2015, at 23:15.