ARL-TR-9463 • MAY 2022

DEVCOM ARMY RESEARCH LABORATORY

Integration of the ADIS16465 Tactical-Grade Inertial Sensor with the Open-Source Pixhawk Autopilot and PX4 Flight Stack

by Sangjin Han and Paul Sabbagh

NOTICES

Disclaimers

The findings in this report are not to be construed as an official Department of the Army position unless so designated by other authorized documents.

Citation of manufacturer's or trade names does not constitute an official endorsement or approval of the use thereof.

Destroy this report when it is no longer needed. Do not return it to the originator.

ARL-TR-9463 ● MAY 2022

Integration of the ADIS16465 Tactical-Grade Inertial Sensor with the Open-Source Pixhawk Autopilot and PX4 Flight Stack

by Sangjin Han
Booz Allen Hamilton, Inc.

Paul Sabbagh
DEVCOM Army Research Laboratory

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED (From - To)
May 2022	Technical Report	March 2022–April 2022

4. TITLE AND SUBTITLE
Integration of the ADIS16465 Tactical-Grade Inertial Sensor with the Open-Source Pixhawk Autopilot and PX4 Flight Stack

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER
0602184A

6. AUTHOR(S)
Sangjin Han and Paul Sabbagh

5d. PROJECT NUMBER
CN2

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
DEVCOM Army Research Laboratory
ATTN: FCDD-RLS-SI
Adelphi Laboratory Center, MD 20783

8. PERFORMING ORGANIZATION REPORT NUMBER
ARL-TR-9463

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES
primary author's email: <han_sangjin@bah.com>.

14. ABSTRACT
A tactical-grade inertial measurement unit (IMU) provides more stable measurements over low-cost IMUs in accuracy and precision. We developed a PX4 Autopilot device driver for a tactical-grade IMU, ADIS 16465, and a daughterboard for the Pixracer R15, a Pixhawk hardware flight controller. The daughterboard communicates over the serial peripheral interface (SPI) to receive IMU measurement data from the ADIS16465 and sends the data to the Pixracer R15. By adding the components in a compact way, we demonstrate that the proposed method replaces low-cost IMUs with a high-performance IMU while guaranteeing the clearance for the other connections that the Pixracer R15 offers. The entire source code that we modified to the legacy source tree is documented, and we validated the connection and functionality of the IMU. Readers who want to use tactical-grade IMUs, in particular the ADIS 16465 with a Pixracer R15, could reproduce our result in this technical report with minimal effort.

15. SUBJECT TERMS
PX4 autopilot, IMU, SPI, ADIS16465, Pixracer R15, device driver, Photonics, Electronics, and Quantum Sciences

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Paul Sabbagh
Unclassified	Unclassified	Unclassified	UU	55	19b. TELEPHONE NUMBER (Include area code) 301-394-0201

Standard Form 298 (Rev. 8/98)
Prescribed by ANSI Std. Z39.18

Contents

List of Figures

List of Tables

1. Introduction

An unmanned aerial vehicle (UAV) is an aircraft that is remotely controlled and features autonomous operations. The autopilot, the "brain" of the UAV,[1] is a flight-control software stack, and the PX4 Autopilot is one implementation of an open source autopilot. We use "PX4" and "PX4 Autopilot" interchangeably. The flight controller is the hardware that runs an PX4 Autopilot on the NuttX operating system (OS). NuttX is a real-time operating system (RTOS) for microcontrollers.

The flight controller uses sensors as control inputs to the system. The sensor measurement data are used to determine the state of the vehicle and are essential for vehicle stabilization and autonomous flight control. In particular, the gyroscope and accelerometer are of critical importance to determine the orientation of the vehicle in its own body frame. Most commercial off-the-shelf flight controllers employ low-cost micro-electromechanical system (MEMS) inertial measurement unit (IMU) sensors that are integrated into the printed circuit board (PCB) of the flight controller. Due to spacial tightness, the inertial sensors are inevitably located next to other integrated chips, such as the microcontroller and other "internal sensors", so that they are exposed to heat and magnetic fields generated by the other components during operation.

Tactical-grade IMU sensors (Angular Random Walk $< 0.2^\circ/\sqrt{hr}$ and Gyroscope Noise $\leq 2^\circ/hr$) are known to be highly stable due to factory calibrations that characterize the sensor for sensitivity, bias, and alignment among other factors, and provide high precision and reliability in a range of harsh environments. The sensors send their measurements to the microcontroller via the serial peripheral interface (SPI) communication protocol. SPI is recommended to be used for only communications between integrated circuits (e.g., the internal sensors) or PCBs with short cables.

We implemented an auxiliary daughterboard for a tactical-grade IMU that plugs into the flight controller using board-to-board connectors. We modified the current firmware for the PX4 Autopilot to communicate over SPI using the daughterboard. Our approach enables the tactical-grade IMU to replace the default internal IMU(s) of the flight controller.

2. Background

The PX4 Autopilot is an open-source flight control software. The source code for the firmware is maintained in GitHub,[2] and the latest version can be downloaded and modified under the BSD license. As of this writing, our current implementation is based on the version 1.13.0-alpha1, released on July 30, 2021. The flight controller hardware of interest is called a Pixracer R15. It is part of the Pixhawk series[1] manufactured by mRobotics. We chose the ADIS 16465 from Analog Devices as our tactical-grade IMU. The ADIS 16465 supports the SPI protocol for data communication. The following sections outline the necessary steps for a Pixracer R15 and an ADIS 16465 to transmit and receive data via SPI.

2.1 Preliminary Setup for External SPI Connection

The Pixracer R15 does not have a designated connector for an external SPI. Pixhawk,[3] however, uses the same microprocessor chip and has the external SPI connector. By cross-referencing with the Pixhawk configuration, candidate pins for external SPI in the Pixracer R15 can be identified.

Consider the schematic of Pixhawk in Fig.1.

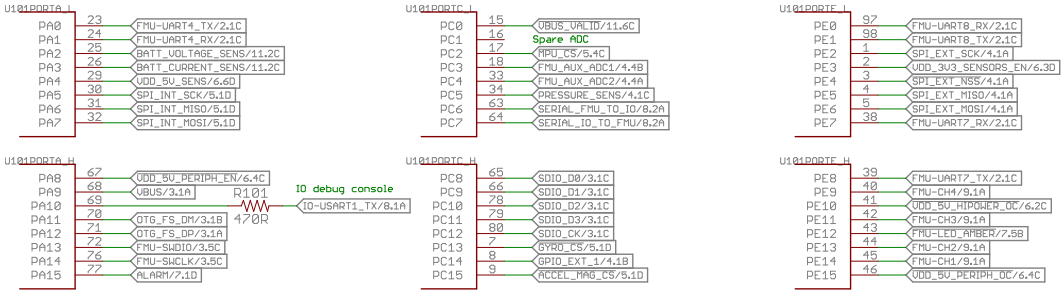

Fig. 1 STM32F427 pinout for 3DR Pixhawk 1

We see the external SPI pinouts in Table 1. Now, consider the schematic of Pixracer R15 in Fig. 2. By inspection, we located the pinouts that are of importance in Table 2. From Table 2, it is obvious that we need to repurpose the ESP8266 connection in the Pixracer for the external SPI connection. Thus, we used the ESP8266 connector information in Fig. 3.

Table 1 External SPI pinouts

Pin (Connector)	Signal	Voltage (V)	STM32F427 Pin
1	VCC	+5	PA8
2	SPI_EXT_SCK	+3.3	PE2
3	SPI_EXT_MISO	+3.3	PE5
4	SPI_EXT_MOSI	+3.3	PE6
5	!SPI_EXT_NSS	+3.3	PE4
6	!GPIO_EXT	+3.3	PC14
7	GND	GND	GND

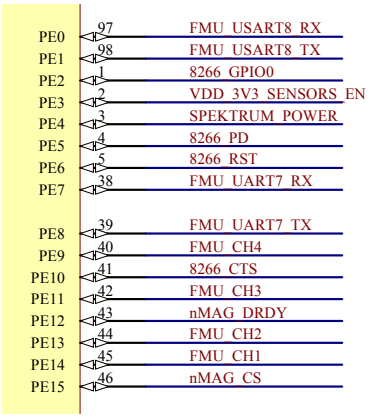

Fig. 2 STM32F427 pinout (left) for Pixracer

Table 2 Pixracer candidate pins for external SPI connection

Pin (Connector)	Signal	Voltage (V)	STM32F427 Pin
6	8266 GPIO0	+3.3	PE2
3	8266 PD	+3.3	PE5
9	8266 RST	+3.3	PE6

In the hardware of the 3DR Pixhawk 1 autopilot, the external SPI is labeled interchangeably as SPI4. The relevant firmware setup for SPI4 is mainly done in `/PX4-Autopilot/boards/px4/fmu-v2/`. In `/PX4-Autopilot/boards/px4/fmu-v2/nuttx-config/include/board_dma_map.h`, the Direct Memory Access (DMA) channels are set. In `/PX4-Autopilot/boards/px4/fmu-v2/nuttx-config/nsh/defconfig`, the chip specific parameter definitions for

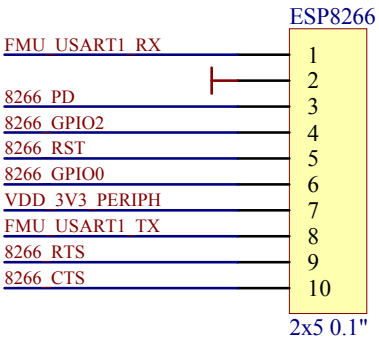

Fig. 3 ESP8266 connection for external SPI connection

SPI4 and its DMA are set. In `/PX4-Autopilot/boards/px4/fmu-v2/src/spi.cpp`, the external SPI Bus is initialized with a specific chip select (CS) pin.

2.2 Tactical-Grade IMU Pinout

Analog Digital provides the datasheet for ADIS 16465[4] where we can find the pinout information. From Table 5 in the datasheet, we identify the following pins in Table 3.

Table 3 ADIS 16465 pin descriptions

Pin	Mnemonic	Description
1	DR	Data Ready Indicator
3	SCLK	SPI Serial Clock
4	DOUT	SPI Data Output
5	DIN	SPI Data Input
6	$\overline{\text{CS}}$	SPI Chip Select
11	VDD	Power Supply (3.3 V)
13	GND	Power Ground

`DR` is a data ready pin and connected to a GPIO pin in Pixracer. `SCLK` is a clock pin and connected to `SCK`. `DOUT` is a data output pin and connected to `MISO`. `DIN` is a data input pin and connected to `MOSI`. $\overline{\text{CS}}$ is a chip select pin and connected to `NSS` which is read by 'negative slave select'. `VDD` is a power supply pin and connected to a 3.3-V power. `GND` is a ground pin and connected to a ground.

2.3 IMU Device Driver

The IMU device driver source codes are located in `/PX4-Autopilot/src/drivers/imu/`. The latest firmware source code includes the device driver for ADIS 16470 and it is similar to our choice of IMU. In the next section, we develop our device driver based on this driver. Here, the file structure of the ADIS 16470 is briefly introduced with a few remarks.

The ADIS 16470 device driver is located in `/PX4-Autopilot/src/drivers/imu/analog_devices/adis16470` and contains five files:
1) `CMakeLists.txt` specifies source files that a px4 IMU module comprises
2) `Analog_Devices_ADIS16470_registers.hpp` is a header file where the register names are defined with corresponding addresses
3) `ADSI16470.hpp` defines the class, `ADIS16470`, and lists functions and members
4) `ADIS16470.cpp` implements the aforementioned class
5) `adis16470_main.cpp` deals with the main function that runs on NuttX Shell, called `nsh`

On updating the IMU readings, the `RumImpl` function in ADIS16470 device driver executes the data fetch. Upon a successful data fetch from the IMU, it publishes uORB topics, called `sensor_accel` for accelerometer values and `sensor_gyro` for gyroscope values via the function called `update`. However, the Pixracer R15 has two internal IMU sensors, ICM 20602 and MPU 9250, and their device drivers use the function called `updateFIFO`. This publishes four uORB topics: `sensor_accel`, `sensor_gyro`, `sensor_accel_fifo` and `sensor_gyro_fifo`. Since the 'fifo' topics are preferred in other modules, we will use `updateFIFO` instead of `update` in our application. Discussion of the 'fifo' topics is beyond the scope of this report.

3. Results

In this section, the custom daughterboard is detailed for the application and the implementation of the source code is explained.

3.1 Daughterboard

A custom PCB is made to minimize the overall size of the addition and to efficiently manage the connection without using additional cables for the external SPI connection on the flight controller hardware, Pixracer R15. The general configuration is based on the breakout board ADIS16IMU4/PCBZ from Analog Devices.[5] The breakout board has a connector of 14-pin socket that mates to the IMU. We connect the pins from this connector to ESP8266 port of Pixracer R15 shown in Fig. 3 along with two bypass capacitors. The pin matching is listed in Table 4. The

Table 4 Daughterboard pin matchup

Pixracer R15	Pin	Pin	ADIS 16465
8266 GPIO2	4	1	DR
8266 GPIO0	6	3	SCLK
8266 PD	3	4	DOUT
8266 RST	5	5	DIN
8266 RTS	9	6	$\overline{\text{CS}}$
VDD 3V3 PERIPH	7	11	VDD
GND	2	13	GND

PCB is designed based on the pin connections in Table 4. The schematic layout of the PCB is shown in Fig. 4. In Fig. 4, `J1` represents the connector for the Pixracer R15, and `J2` represents the connector for the ADIS 16465 IMU. We see that the lines connecting `J1` and `J2` are consistent to the rows in Table 4.

The PCB layout is then drawn as shown in Fig. 5 The front and back of the PCB is depicted in Fig. 6 and Fig. 7, respectively.

The surface-mounted device (SMD) capacitors are soldered onto the PCB. The male header (`J1`) for the daughterboard is compatible with the ESP8266 female header on the Pixracer R15. The female header (`J2`) is for the ADIS 16465 IMU. The daughterboard is printed and capacitors and headers and soldered. The front and back of the daughterboard is shown in Fig. 8.

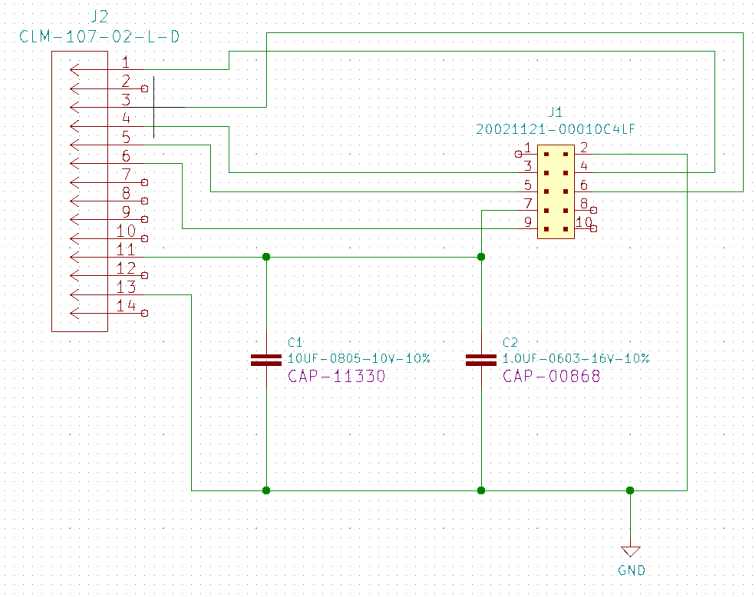

Fig. 4 Schematic for the daughterboard

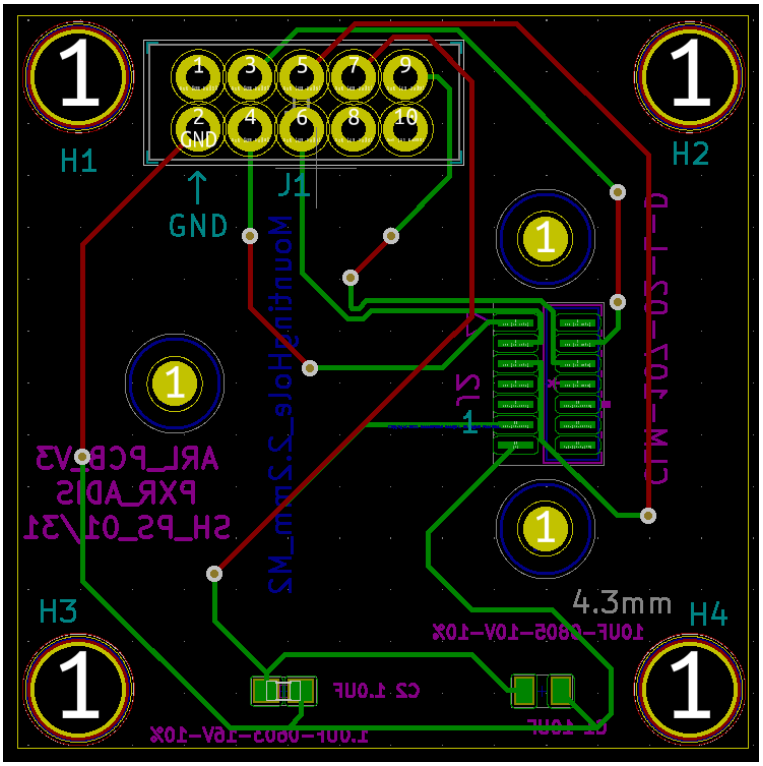

Fig. 5 PCB layout of the daughterboard

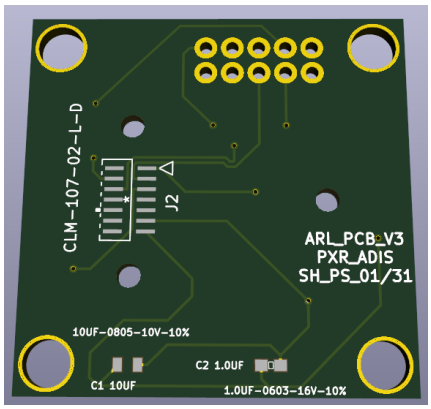

Fig. 6 Front 3D view of the PCB

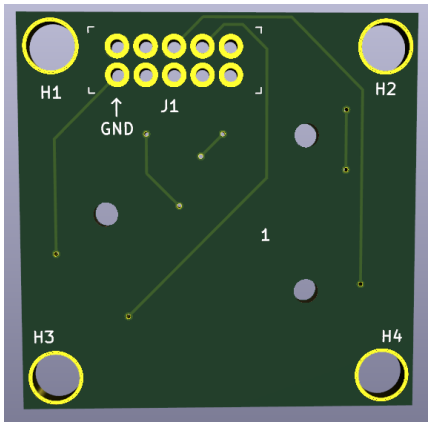

Fig. 7 Back 3D view of the PCB

Standoffs and screws are added to secure the daughterboard to the Pixracer R15, which is also shown in Fig. 8.

Fig. 9 shows that the assembly does not interfere with other cable connections. In the following section, the modifications of the open-source firmware of PX4-Autopilot are detailed so that the ADIS 16465 IMU communicates to the Pixracer R15.

Fig. 8 The daughterboard top (right), bottom (left), and assembled with Pixracer R15 (top)

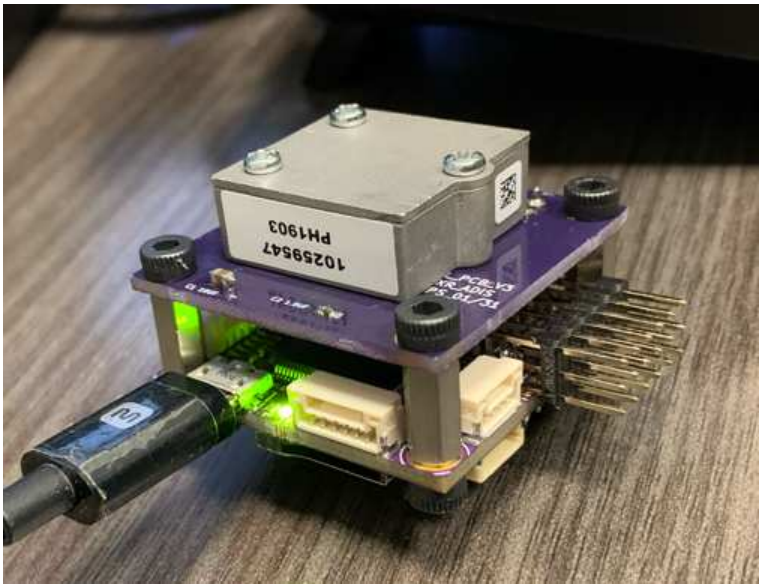

Fig. 9 ADIS16465 is assembled into the daughterboard that is mounted onto Pixracer R15

3.2 Implementation of the Device Driver

The external SPI connection is presented and the device driver for ADIS 16465 is introduced along with relevant setups.

3.2.1 Enabling External SPI Connection

In `PX4-Autopilot/boards/px4/fmu-v4/nuttx-config/nsh/defconfig` is the source code for SPI4 and its DMA settings.

```
CONFIG_STM32_SPI4=y
CONFIG_STM32_SPI_DMA=y
CONFIG_STM32_SPI_DMATHRESHOLD=8
```

This changes to the following:

```
CONFIG_STM32_SPI4=y
CONFIG_STM32_SPI4_DMA=y
CONFIG_STM32_SPI4_DMA_BUFFER=1024
CONFIG_STM32_SPI_DMA=y
CONFIG_STM32_SPI_DMATHRESHOLD=32
```

In `PX4-Autopilot/boards/px4/fmu-v4/nuttx-config/include/board_dma_map.h`, there is the following code for the DMA channels.

```
//  DMA2 Channel/Stream Selections
//------------------------------------------
//---------------------------//----------------
#define DMACHAN_SPI1_RX    DMAMAP_SPI1_RX_1
// DMA2, Stream 0, Channel 3    (SPI sensors RX)
#define DMAMAP_USART6_RX   DMAMAP_USART6_RX_1
// DMA2, Stream 1, Channel 4
#define DMAMAP_USART1_RX   DMAMAP_USART1_RX_1
// DMA2, Stream 2, Channel 4
#define DMACHAN_SPI1_TX    DMAMAP_SPI1_TX_1
// DMA2, Stream 3, Channel 3    (SPI sensors TX)
//      AVAILABLE
// DMA2, Stream 4
//      DMAMAP_TIM1_UP
// DMA2, Stream 5, Channel 6    (DSHOT)
#define DMAMAP_SDIO        DMAMAP_SDIO_2
// DMA2, Stream 6, Channel 4
```

This changes to the following:

```
//  DMA2 Channel/Stream Selections
//-------------------------------------------
//---------------------------//----------------
#define DMACHAN_SPI4_RX    DMAMAP_SPI4_RX_1
// DMA2, Stream 0, Channel 4    (SPI4 sensors RX)
#define DMAMAP_USART6_RX   DMAMAP_USART6_RX_1
// DMA2, Stream 1, Channel 4    (PX4IO RX)
#define DMACHAN_SPI1_RX    DMAMAP_SPI1_RX_2
// DMA2, Stream 2, Channel 3    (SPI1 sensors RX)
#define DMACHAN_SPI1_TX    DMAMAP_SPI1_TX_1
// DMA2, Stream 3, Channel 3    (SPI1 sensors TX)
#define DMACHAN_SPI4_TX    DMAMAP_SPI4_TX_2
// DMA2, Stream 4, Channel 5    (SPI4 sensors TX)
//      DMAMAP_TIM1_UP
// DMA2, Stream 5, Channel 6    (DSHOT)
#define DMAMAP_SDIO        DMAMAP_SDIO_2
// DMA2, Stream 6, Channel 4
#define DMAMAP_USART6_TX   DMAMAP_USART6_TX_2
// DMA2, Stream 7, Channel 5    (PX4IO TX)
```

In `PX4-Autopilot/boards/px4/fmu-v4/src/spi.cpp`, the initialization of the external SPI is implemented as follows:

```
initSPIBusExternal(SPI::Bus::SPI4, {
initSPIConfigExternal(SPI::CS{GPIO::PortA, GPIO::Pin8}),
}),
```

We added the port for the data ready pin (DRDY).

```
initSPIBusExternal(SPI::Bus::SPI4, {
initSPIConfigExternal(SPI::CS{GPIO::PortA, GPIO::Pin8},
SPI::DRDY{GPIO::PortB, GPIO::Pin4}),
}),
```

In `PX4-Autopilot/boards/px4/fmu-v4/src/init.c`, there are two places for probing the 8266 GPIO2 pin that we use for DRDY. First one is in the following code:

```
/* We have SPI4 is GPIO_8266_GPIO2 PB4 pin 3 Low */
if (stm32_gpioread(GPIO_8266_GPIO2) != 0) {
```

Since we used the pin for DRDY and it is not always low, we changed the code as follows:

```
/* We have SPI4 is GPIO_8266_GPIO2 PB4 pin 3 Low */
if (stm32_gpioread(GPIO_8266_GPIO2) != 0 && false) {
```

We could simply delete a block of the `if` clause, but a trivial `false` is rather added for minimal modification.

The second one is in the following code:

```
if (stm32_gpioread(GPIO_8266_GPIO2) == 0) {
```

We add a trivial `true` in this case.

```
if (stm32_gpioread(GPIO_8266_GPIO2) == 0 || true) {
```

The external SPI communication protocol is now enabled as `SPI4` with a DRDY port.

3.2.2 Creating a New Device Driver

We created a new folder `adis16465` in `PX4-Autopilot/src/drivers/imu/analog_devices` and created the `CMakeLists.txt` file with the following contents:

```
px4_add_module(
    MODULE drivers__imu__analog_devices__adis16465
    MAIN adis16465
```

```
    COMPILE_FLAGS
    SRCS
        ADIS16465.cpp
        ADIS16465.hpp
        adis16465_main.cpp
        Analog_Devices_ADIS16465_registers.hpp
    DEPENDS
        drivers_accelerometer
        drivers_gyroscope
        px4_work_queue
)
```

In the same folder, four more source files are added: `Analog_Devices_ADIS16465_registers.hpp`, `ADIS16465.hpp`, `ADIS16465.cpp`, and `adis16465_main.cpp` are created, and the codes are shown in Appendix A, Appendix B, Appendix C, Appendix D, respectively.

Now, there are four places that are modified so that the driver is executed at start-up. In `PX4-Autopilot/src/drivers/imu/analog_devices/CMakeLists.txt`, we have a list of these subdirectories:

```
add_subdirectory(adis16448)
add_subdirectory(adis16470)
```

We change this to the following:

```
add_subdirectory(adis16448)
add_subdirectory(adis16465)
```

In `PX4-Autopilot/src/drivers/drv_sensor.h`, we have the parameter definition as follows:

```
#define DRV_IMU_DEVTYPE_ADIS16470 0x58
```

We modified this to the following:

```
#define DRV_IMU_DEVTYPE_ADIS16465 0x58
```

In `PX4-Autopilot/boards/px4/fmu-v4/default.cmake`, we have a portion of the list of the included drivers as follows:

```
imu/analog_devices/adis16448
```

We change this to the following:

```
imu/analog_devices/adis16465
```

In `PX4-Autopilot/boards/px4/fmu-v4/init/rc.board_sensors`, we have the following start-up codes for peripheral sensors:

```
# ICM20602 internal SPI bus ICM-20608-G
if ! icm20602 -s -R 8 start
then
# ICM20608 internal SPI bus ICM-20602-G
icm20608g -s -R 8 start
fi

# For Teal One airframe
if param compare SYS_AUTOSTART 4250
then
mpu6500 -s -R 0 start
else
# mpu9250 internal SPI bus mpu9250
mpu9250 -s -R 8 start
fi
```

This needs to be changed to the following:

```
# ICM20602 internal SPI bus ICM-20608-G
# if ! icm20602 -s -R 8 start
```

```
# then
# # ICM20608 internal SPI bus ICM-20602-G
# icm20608g -s -R 8 start
# fi

# For Teal One airframe
if param compare SYS_AUTOSTART 4250
then
mpu6500 -s -R 0 start
else
# mpu9250 internal SPI bus mpu9250
# mpu9250 -s -R 8 start
adis16465 -S start
fi
```

3.3 Validation of Integration

We can validate the data from ADIS16465 using cthe NuttX Shell (`nsh`). The connection over external SPI can be shown by the following command, `adis16465 status`, in the shell:

```
NuttShell (NSH) NuttX-10.1.0
nsh> adis16465 status
INFO  [SPI_I2C] Running on SPI Bus 4
adis16465: reset: 2 events
adis16465: bad register: 0 events
adis16465: bad transfer: 0 events
adis16465: CRC16 bad: 0 events
adis16465: DRDY missed: 0 events
nsh>
```

Next, the data is validated by capturing recent uORB topics:

```
nsh> listener sensor_accel

TOPIC: sensor_accel
```

```
sensor_accel_s
timestamp: 443202918  (0.000556 seconds ago)
timestamp_sample: 443202624  (294 us before timestamp)
device_id: 5767202 (Type: 0x58, SPI:4 (0x00))
x: -23.7566
y: 0.3309
z: -42.8183
temperature: 31.9000
error_count: 0
clip_counter: [0, 0, 0]
samples: 1
nsh> listener sensor_gyro

TOPIC: sensor_gyro
sensor_gyro_s
timestamp: 452547679  (0.000553 seconds ago)
timestamp_sample: 452547337  (342 us before timestamp)
device_id: 5767202 (Type: 0x58, SPI:4 (0x00))
x: 0.0017
y: -0.0069
z: 0.0030
temperature: 32.3000
error_count: 0
samples: 1
nsh>
```

4. Conclusion

We developed a device driver of the tactical-grade IMU, ADIS16465 for a PX4 Autopilot, and a daughterboard for the specific flight controller hardware, Pixracer R15. Data from the IMU was validated by uORB topics via the NuttX Shell. The integration showed the possibility of replacing low-cost IMUs that are default to the commercial off-the-shelf flight controller with a tactical-grade IMU. This will provide significantly more-stable measurements than those from the default IMU sensors, and the feedback system of the flight controller will benefit the quality of the data. This report explained step-by-step procedures so that readers can reproduce this configuration with minimal effort.

5. References

1. Dronecode Project. PX4 Autopilot User Guide. n.d. [accessed 2022 Jan 1]. https://docs.px4.io/master/en.

2. Dronecode Project. PX4 Drone Autopilot. n.d. [accessed 2022 Jan 1]. https://github.com/PX4/PX4-Autopilot.

3. Dronecode Project. 3DR Pixhawk 1. n.d. [accessed 2022 Jan 1]. https://docs.px4.io/master/en/flight_controller/pixhawk.html.

4. Analog Devices. Precision MEMS IMU Module ADIS 16465 data sheet. n.d. [accessed 2022 Jan 1]. https://www.analog.com/media/en/technical-documentation/data-sheets/adis16465.pdf.

5. Analog Devices. ADIS16IMUZ/PCBZ data sheet. n.d. [accessed 2022 Feb 2]. https://www.analog.com/en/design-center/evaluation-hardware-and-software/evaluation-boards-kits/EVAL-ADIS16IMU4.html.

Appendix A. Analog_Devices_ADIS16465_registers.hpp

```
#pragma once

#include <cstdint>

// TODO: move to a central header
static constexpr uint16_t Bit0  = (1 << 0);
static constexpr uint16_t Bit1  = (1 << 1);
static constexpr uint16_t Bit2  = (1 << 2);
static constexpr uint16_t Bit3  = (1 << 3);
static constexpr uint16_t Bit4  = (1 << 4);
static constexpr uint16_t Bit5  = (1 << 5);
static constexpr uint16_t Bit6  = (1 << 6);
static constexpr uint16_t Bit7  = (1 << 7);
static constexpr uint16_t Bit8  = (1 << 8);
static constexpr uint16_t Bit9  = (1 << 9);
static constexpr uint16_t Bit10 = (1 << 10);
static constexpr uint16_t Bit11 = (1 << 11);
static constexpr uint16_t Bit12 = (1 << 12);
static constexpr uint16_t Bit13 = (1 << 13);
static constexpr uint16_t Bit14 = (1 << 14);
static constexpr uint16_t Bit15 = (1 << 15);

namespace Analog_Devices_ADIS16465
{
static constexpr uint32_t SPI_SPEED = 2 * 1000 * 1000;
// 2 MHz SPI serial interface

static constexpr uint32_t SPI_SPEED_BURST = 1 * 1000 * 1000;
// 1 MHz SPI serial interface for burst read

static constexpr uint32_t SPI_STALL_PERIOD = 16;
// 16 us Stall period between data

static constexpr uint16_t DIR_WRITE = 0x80;
```

```
static constexpr uint16_t Product_identification = 0x4051;
// 4051 in HEX is 16465 in decimal

static constexpr uint32_t SAMPLE_INTERVAL_US = 1000;
// 1000 Hz

enum class Register : uint16_t {
DIAG_STAT      = 0x02,

FILT_CTRL      = 0x5C,
RANG_MDL       = 0x5E,
MSC_CTRL       = 0x60,
DEC_RATE       = 0x64,
GLOB_CMD       = 0x68,

FIRM_REV       = 0x6C,
// Identification, firmware revision
FIRM_DM        = 0x6E,
// Identification, date code, day and month
FIRM_Y         = 0x70,
// Identification, date code, year
PROD_ID        = 0x72,
// Identification, part number
SERIAL_NUM     = 0x74,
// Identification, serial number

FLSHCNT_LOW    = 0x7C,
// Output, flash memory write cycle counter, lower word
FLSHCNT_HIGH   = 0x7E,
// Output, flash memory write cycle counter, upper word
};

// DIAG_STAT
enum DIAG_STAT_BIT : uint16_t {
Clock_error                = Bit7,
```

```
// A 1 indicates that the internal data sampling clock
Memory_failure              = Bit6,
// A 1 indicates a failure in the flash memory test
Sensor_failure              = Bit5,
// A 1 indicates failure of at least one sensor,
// at the conclusion of the self test
Standby_mode                = Bit4,
// A 1 indicates that the voltage across VDD and GND is <2.8 V,
// which causes data processing to stop
SPI_communication_error     = Bit3,
// A 1 indicates that the total number of SCLK cycles
// is not equal to an integer multiple of 16
Flash_memory_update_failure = Bit2,
// A 1 indicates that the most recent flash memory
// update failed
Data_path_overrun           = Bit1,
// A 1 indicates that one of the data paths have
// experienced an overrun condition
};

// FILT_CTRL
enum FILT_CTRL_BIT : uint16_t {

};

// MSC_CTRL
enum MSC_CTRL_BIT : uint16_t {
DR_polarity = Bit0, // 1 = active high when data is valid
};

// GLOB_CMD
enum GLOB_CMD_BIT : uint16_t {
Sensor_self_test  = Bit2,
Flash_memory_test = Bit4,
Software_reset    = Bit7,
```

```
};

} // namespace Analog_Devices_ADIS16465
```

Appendix B. ADIS16465.hpp

```
/**
* @file ADIS16465.hpp
*
* Driver for the Analog Devices ADIS16465 connected via
* external SPI.
*
*/

#pragma once

#include "Analog_Devices_ADIS16465_registers.hpp"

#include <drivers/drv_hrt.h>
#include <lib/drivers/accelerometer/PX4Accelerometer.hpp>
#include <lib/drivers/device/spi.h>
#include <lib/drivers/gyroscope/PX4Gyroscope.hpp>
#include <lib/geo/geo.h>
#include <lib/perf/perf_counter.h>
#include <px4_platform_common/atomic.h>
#include <px4_platform_common/i2c_spi_buses.h>

using namespace Analog_Devices_ADIS16465;

class ADIS16465 : public device::SPI,
public I2CSPIDriver<ADIS16465>
{
public:
ADIS16465(const I2CSPIDriverConfig &config);
~ADIS16465() override;

static void print_usage();

void RunImpl();

int init() override;
```

```
void print_status() override;

private:
void exit_and_cleanup() override;

struct register_config_t {
Register reg;
uint16_t set_bits{0};
uint16_t clear_bits{0};
};
struct BurstRead {
uint16_t cmd;
uint16_t DIAG_STAT;
int16_t X_GYRO_OUT;
int16_t Y_GYRO_OUT;
int16_t Z_GYRO_OUT;
int16_t X_ACCL_OUT;
int16_t Y_ACCL_OUT;
int16_t Z_ACCL_OUT;
int16_t TEMP_OUT;
uint16_t DATA_CNTR;
uint16_t checksum;
}buffer{};

int probe() override;

bool Reset();

bool Configure();

static int DataReadyInterruptCallback(int irq, void *context,
void *arg);
void DataReady();
bool DataReadyInterruptConfigure();
bool DataReadyInterruptDisable();
```

```
bool RegisterCheck(const register_config_t &reg_cfg);

uint16_t RegisterRead(Register reg);
void RegisterWrite(Register reg, uint16_t value);
void RegisterSetAndClearBits(Register reg, uint16_t setbits,
uint16_t clearbits);

void ProcessAccelGyro(const hrt_abstime &timestamp_sample,
const BurstRead bufdata);

const spi_drdy_gpio_t _drdy_gpio;

PX4Accelerometer _px4_accel;
PX4Gyroscope _px4_gyro;

perf_counter_t _reset_perf{perf_alloc(PC_COUNT,
MODULE_NAME": reset")};
perf_counter_t _bad_register_perf{perf_alloc(PC_COUNT,
MODULE_NAME": bad register")};
perf_counter_t _bad_transfer_perf{perf_alloc(PC_COUNT,
MODULE_NAME": bad transfer")};
perf_counter_t _perf_crc_bad{perf_counter_t(perf_alloc(PC_COUNT,
MODULE_NAME": CRC16 bad"))};
perf_counter_t _drdy_missed_perf{nullptr};

hrt_abstime _reset_timestamp{0};
hrt_abstime _last_config_check_timestamp{0};
int _failure_count{0};

px4::atomic<hrt_abstime> _drdy_timestamp_sample{0};
bool _data_ready_interrupt_enabled{false};

bool _self_test_passed{false};
```

```
enum class STATE : uint8_t {
RESET,
WAIT_FOR_RESET,
SELF_TEST_CHECK,
CONFIGURE,
READ,
} _state{STATE::RESET};

uint8_t _checked_register{0};
static constexpr uint8_t size_register_cfg{1};
register_config_t _register_cfg[size_register_cfg] {
// Register              | Set bits, Clear bits
{ Register::MSC_CTRL,    0, MSC_CTRL_BIT::DR_polarity },
};
};
```

Appendix C. ADIS16465.cpp

```
#include "ADIS16465.hpp"

using namespace time_literals;

static constexpr int16_t combine(uint8_t msb, uint8_t lsb)
{
return (msb << 8u) | lsb;
}

ADIS16465::ADIS16465(const I2CSPIDriverConfig &config) :
SPI(config),
I2CSPIDriver(config),
_drdy_gpio(config.drdy_gpio),
_px4_accel(get_device_id(), config.rotation),
_px4_gyro(get_device_id(), config.rotation)
{
if (_drdy_gpio != 0) {
_drdy_missed_perf = perf_alloc(PC_COUNT,
MODULE_NAME": DRDY missed");
}
}

ADIS16465::~ADIS16465()
{
perf_free(_reset_perf);
perf_free(_bad_register_perf);
perf_free(_bad_transfer_perf);
perf_free(_perf_crc_bad);
perf_free(_drdy_missed_perf);
}

int ADIS16465::init()
{
int ret = SPI::init();
```

```
if (ret != PX4_OK) {
DEVICE_DEBUG("SPI::init failed (%i)", ret);
return ret;
}

return Reset() ? 0 : -1;
}

bool ADIS16465::Reset()
{
_state = STATE::RESET;
DataReadyInterruptDisable();
ScheduleClear();
ScheduleNow();
return true;
}

void ADIS16465::exit_and_cleanup()
{
DataReadyInterruptDisable();
I2CSPIDriverBase::exit_and_cleanup();
}

void ADIS16465::print_status()
{
I2CSPIDriverBase::print_status();

perf_print_counter(_reset_perf);
perf_print_counter(_bad_register_perf);
perf_print_counter(_bad_transfer_perf);
perf_print_counter(_perf_crc_bad);
perf_print_counter(_drdy_missed_perf);
}

int ADIS16465::probe()
```

```
{
// Power-On Start-Up Time 259 ms
if (hrt_absolute_time() < 259_ms) {
PX4_WARN("Power-On Start-Up Time is 259 ms");
}

const uint16_t PROD_ID = RegisterRead(Register::PROD_ID);

if (PROD_ID != Product_identification) {
DEVICE_DEBUG("unexpected PROD_ID 0x%02x", PROD_ID);
return PX4_ERROR;
}

const uint16_t SERIAL_NUM = RegisterRead(Register::SERIAL_NUM);
const uint16_t FIRM_REV = RegisterRead(Register::FIRM_REV);
const uint16_t FIRM_DM = RegisterRead(Register::FIRM_DM);
const uint16_t FIRM_Y = RegisterRead(Register::FIRM_Y);

PX4_INFO("Serial Number: 0x%X,
Firmware revision: 0x%X Date: Y %X DM %X",
SERIAL_NUM, FIRM_REV, FIRM_Y, FIRM_DM);

return PX4_OK;
}

void ADIS16465::RunImpl()
{
const hrt_abstime now = hrt_absolute_time();

switch (_state) {
case STATE::RESET:
perf_count(_reset_perf);
// GLOB_CMD: software reset
RegisterWrite(Register::GLOB_CMD, GLOB_CMD_BIT::Software_reset);
_reset_timestamp = now;
```

```
_failure_count = 0;
_state = STATE::WAIT_FOR_RESET;
ScheduleDelayed(198_ms); // 198 ms Software Reset Recovery Time
break;

case STATE::WAIT_FOR_RESET:

if (_self_test_passed) {
if ((RegisterRead(Register::PROD_ID) == Product_identification)) {
// if reset succeeded then configure
_state = STATE::CONFIGURE;
ScheduleNow();

} else {
// RESET not complete
if (hrt_elapsed_time(&_reset_timestamp) > 1000_ms) {
PX4_DEBUG("Reset failed, retrying");
_state = STATE::RESET;
ScheduleDelayed(100_ms);

} else {
PX4_DEBUG("Reset not complete, check again in 100 ms");
ScheduleDelayed(100_ms);
}
}

} else {
RegisterWrite(Register::GLOB_CMD, GLOB_CMD_BIT::Sensor_self_test);
_state = STATE::SELF_TEST_CHECK;
ScheduleDelayed(14_ms); // Self Test Time
}

break;

case STATE::SELF_TEST_CHECK: {
```

```
// read DIAG_STAT to check result
const uint16_t DIAG_STAT = RegisterRead(Register::DIAG_STAT);

if (DIAG_STAT != 0) {
PX4_ERR("DIAG_STAT: %#X", DIAG_STAT);

} else {
PX4_DEBUG("self test passed");
_self_test_passed = true;
_state = STATE::RESET;
ScheduleNow();
}
}
break;

case STATE::CONFIGURE:
if (Configure()) {
// if configure succeeded then start reading
_state = STATE::READ;

if (DataReadyInterruptConfigure()) {
_data_ready_interrupt_enabled = true;

// backup schedule as a watchdog timeout
ScheduleDelayed(100_ms);

} else {
_data_ready_interrupt_enabled = false;
ScheduleOnInterval(SAMPLE_INTERVAL_US, SAMPLE_INTERVAL_US);
}

} else {
// CONFIGURE not complete
if (hrt_elapsed_time(&_reset_timestamp) > 1000_ms) {
PX4_DEBUG("Configure failed, resetting");
```

```
_state = STATE::RESET;

} else {
PX4_DEBUG("Configure failed, retrying");
}

ScheduleDelayed(100_ms);
}

break;

case STATE::READ: {
hrt_abstime timestamp_sample = now;

if (_data_ready_interrupt_enabled) {
// scheduled from interrupt if _drdy_timestamp_sample was set
// as expected
const hrt_abstime drdy_timestamp_sample =
_drdy_timestamp_sample.fetch_and(0);

if ((now - drdy_timestamp_sample) < SAMPLE_INTERVAL_US) {
timestamp_sample = drdy_timestamp_sample;

} else {
perf_count(_drdy_missed_perf);
}

// push backup schedule back
ScheduleDelayed(SAMPLE_INTERVAL_US * 2);
}

bool success = false;
/* moved to ADIS16465.hpp
struct BurstRead {
uint16_t cmd;
```

```
uint16_t DIAG_STAT;
int16_t X_GYRO_OUT;
int16_t Y_GYRO_OUT;
int16_t Z_GYRO_OUT;
int16_t X_ACCL_OUT;
int16_t Y_ACCL_OUT;
int16_t Z_ACCL_OUT;
int16_t TEMP_OUT;
uint16_t DATA_CNTR;
uint16_t checksum;
} buffer{}; */

// ADIS16465 burst report should be 176 bits
static_assert(sizeof(BurstRead) == (176 / 8),
"ADIS16465 report not 176 bits");

buffer.cmd = static_cast<uint16_t>(Register::GLOB_CMD) << 8;
set_frequency(SPI_SPEED_BURST);

if (transferhword((uint16_t *)&buffer, (uint16_t *)&buffer,
sizeof(buffer) / sizeof(uint16_t)) == PX4_OK) {

// Calculate checksum and compare

// Checksum = DIAG_STAT, Bits[15:8] + DIAG_STAT, Bits[7:0] +
//  X_GYRO_OUT, Bits[15:8] + X_GYRO_OUT, Bits[7:0] +
//  Y_GYRO_OUT, Bits[15:8] + Y_GYRO_OUT, Bits[7:0] +
//  Z_GYRO_OUT, Bits[15:8] + Z_GYRO_OUT, Bits[7:0] +
//  X_ACCL_OUT, Bits[15:8] + X_ACCL_OUT, Bits[7:0] +
//  Y_ACCL_OUT, Bits[15:8] + Y_ACCL_OUT, Bits[7:0] +
//  Z_ACCL_OUT, Bits[15:8] + Z_ACCL_OUT, Bits[7:0] +
//  TEMP_OUT, Bits[15:8] + TEMP_OUT, Bits[7:0] +
//  DATA_CNTR, Bits[15:8] + DATA_CNTR, Bits[7:0]
uint8_t *checksum_helper = (uint8_t *)&buffer.DIAG_STAT;
```

```
uint16_t checksum = 0;

for (int i = 0; i < 18; i++) {
checksum += checksum_helper[i];
}

if (buffer.checksum != checksum) {
// PX4_DEBUG("adis_report.checksum: %X vs calculated: %X",
// buffer.checksum, checksum);
perf_count(_bad_transfer_perf);
}

if (buffer.DIAG_STAT != DIAG_STAT_BIT::Data_path_overrun) {
// Data path overrun. A 1 indicates that one of the
// data paths have experienced an overrun condition.
// If this occurs, initiate a reset,

//Reset();
//return;
}

// Check all Status/Error Flag Indicators (DIAG_STAT)
if (buffer.DIAG_STAT != 0) {
perf_count(_bad_transfer_perf);
}

// temperature 1 LSB = 0.1°C
const float temperature = buffer.TEMP_OUT * 0.1f;
_px4_accel.set_temperature(temperature);
_px4_gyro.set_temperature(temperature);

ProcessAccelGyro(timestamp_sample, buffer); /* //
int16_t accel_x = buffer.X_ACCL_OUT;
int16_t accel_y = buffer.Y_ACCL_OUT;
int16_t accel_z = buffer.Z_ACCL_OUT;
```

```
// sensor's frame is +x forward, +y left, +z up
//  flip y & z to publish right handed with z down
// (x forward, y right, z down)
accel_y = (accel_y == INT16_MIN) ? INT16_MAX : -accel_y;
accel_z = (accel_z == INT16_MIN) ? INT16_MAX : -accel_z;

_px4_accel.update(timestamp_sample, accel_x, accel_y, accel_z);


int16_t gyro_x = buffer.X_GYRO_OUT;
int16_t gyro_y = buffer.Y_GYRO_OUT;
int16_t gyro_z = buffer.Z_GYRO_OUT;
// sensor's frame is +x forward, +y left, +z up
//  flip y & z to publish right handed with z down
// (x forward, y right, z down)
gyro_y = (gyro_y == INT16_MIN) ? INT16_MAX : -gyro_y;
gyro_z = (gyro_z == INT16_MIN) ? INT16_MAX : -gyro_z;
_px4_gyro.update(timestamp_sample, gyro_x, gyro_y, gyro_z); */

success = true;

if (_failure_count > 0) {
_failure_count--;
}

} else {
perf_count(_bad_transfer_perf);
}

if (!success) {
_failure_count++;

// full reset if things are failing consistently
if (_failure_count > 10) {
```

```
Reset();
return;
}
}

if (!success ||
hrt_elapsed_time(&_last_config_check_timestamp) > 100_ms) {
// check configuration registers periodically or immediately
// following any failure
if (RegisterCheck(_register_cfg[_checked_register])) {
_last_config_check_timestamp = now;
_checked_register = (_checked_register + 1) % size_register_cfg;

} else {
// register check failed, force reset
perf_count(_bad_register_perf);
Reset();
}
}
}

break;
}
}

bool ADIS16465::Configure()
{
// first set and clear all configured register bits
for (const auto &reg_cfg : _register_cfg) {
RegisterSetAndClearBits(reg_cfg.reg, reg_cfg.set_bits,
reg_cfg.clear_bits);
}
// Data rate 1000 Hz //
RegisterWrite(Register::DEC_RATE, 0x01);
```

```
// now check that all are configured
bool success = true;

for (const auto &reg_cfg : _register_cfg) {
if (!RegisterCheck(reg_cfg)) {
success = false;
}
}

_px4_accel.set_scale(0.25f * CONSTANTS_ONE_G / 1000.0f);
// accel 0.25 mg/LSB
_px4_accel.set_range(40.f * CONSTANTS_ONE_G);

_px4_gyro.set_scale(math::radians(0.1f)); // gyro 0.1 deg/sec/LSB
_px4_gyro.set_range(math::radians(2000.f));


return success;
}

int ADIS16465::DataReadyInterruptCallback(int irq, void *context,
void *arg)
{
static_cast<ADIS16465 *>(arg)->DataReady();
return 0;
}

void ADIS16465::DataReady()
{
_drdy_timestamp_sample.store(hrt_absolute_time());
ScheduleNow();
}

bool ADIS16465::DataReadyInterruptConfigure()
{
```

```
if (_drdy_gpio == 0) {
return false;
}

// Setup data ready on falling edge
return px4_arch_gpiosetevent(_drdy_gpio, false, true, false,
 &DataReadyInterruptCallback, this) == 0;
}

bool ADIS16465::DataReadyInterruptDisable()
{
if (_drdy_gpio == 0) {
return false;
}

return px4_arch_gpiosetevent(_drdy_gpio, false, false, false,
nullptr, nullptr) == 0;
}

bool ADIS16465::RegisterCheck(const register_config_t &reg_cfg)
{
bool success = true;

const uint16_t reg_value = RegisterRead(reg_cfg.reg);

if (reg_cfg.set_bits && ((reg_value & reg_cfg.set_bits) !=
 reg_cfg.set_bits)) {
PX4_DEBUG("0x%02hhX: 0x%02hhX (0x%02hhX not set)",
(uint8_t)reg_cfg.reg, reg_value, reg_cfg.set_bits);
success = false;
}

if (reg_cfg.clear_bits && ((reg_value & reg_cfg.clear_bits) !=
 0)) {
PX4_DEBUG("0x%02hhX: 0x%02hhX (0x%02hhX not cleared)",
```

```
(uint8_t)reg_cfg.reg, reg_value, reg_cfg.clear_bits);
success = false;
}

return success;
}

uint16_t ADIS16465::RegisterRead(Register reg)
{
set_frequency(SPI_SPEED);

uint16_t cmd[1];
cmd[0] = (static_cast<uint16_t>(reg) << 8);

transferhword(cmd, nullptr, 1);
px4_udelay(SPI_STALL_PERIOD);
transferhword(nullptr, cmd, 1);

return cmd[0];
}

void ADIS16465::RegisterWrite(Register reg, uint16_t value)
{
set_frequency(SPI_SPEED);

uint16_t cmd[2];
cmd[0] = (((static_cast<uint16_t>(reg))     | DIR_WRITE) << 8) |
((0x00FF & value));
cmd[1] = (((static_cast<uint16_t>(reg) + 1) | DIR_WRITE) << 8) |
((0xFF00 & value) >> 8);

transferhword(cmd, nullptr, 1);
px4_udelay(SPI_STALL_PERIOD);
transferhword(cmd + 1, nullptr, 1);
}
```

```
void ADIS16465::RegisterSetAndClearBits(Register reg,
uint16_t setbits, uint16_t clearbits)
{
const uint16_t orig_val = RegisterRead(reg);

uint16_t val = (orig_val & ~clearbits) | setbits;

if (orig_val != val) {
RegisterWrite(reg, val);
}
}

void ADIS16465::ProcessAccelGyro
(const hrt_abstime &timestamp_sample, const BurstRead bufdata)
{
sensor_accel_fifo_s accel{};
accel.timestamp_sample = timestamp_sample;
accel.samples = 1;
accel.dt = 1000.0; // 1000 Hz

sensor_gyro_fifo_s gyro{};
gyro.timestamp_sample = timestamp_sample;
gyro.samples = 1;
gyro.dt = 1000.0; // 1000 Hz

int16_t accel_x = bufdata.X_ACCL_OUT;
int16_t accel_y = bufdata.Y_ACCL_OUT;
int16_t accel_z = bufdata.Z_ACCL_OUT;
// sensor's frame is +x forward, +y left, +z up
//  flip y & z to publish right handed with z down
// (x forward, y right, z down)
accel.x[0] = accel_x;
accel.y[0] = (accel_y == INT16_MIN) ? INT16_MAX : -accel_y;
accel.z[0] = (accel_z == INT16_MIN) ? INT16_MAX : -accel_z;
```

```
_px4_accel.updateFIFO(accel);

int16_t gyro_x = bufdata.X_GYRO_OUT;
int16_t gyro_y = bufdata.Y_GYRO_OUT;
int16_t gyro_z = bufdata.Z_GYRO_OUT;
// sensor's frame is +x forward, +y left, +z up
//  flip y & z to publish right handed with z down
// (x forward, y right, z down)
gyro.x[0] = gyro_x;
gyro.y[0] = (gyro_y == INT16_MIN) ? INT16_MAX : -gyro_y;
gyro.z[0] = (gyro_z == INT16_MIN) ? INT16_MAX : -gyro_z;

_px4_gyro.updateFIFO(gyro);
}
```

Appendix D. adis16465_main.cpp

```
#include "ADIS16465.hpp"

#include <px4_platform_common/getopt.h>
#include <px4_platform_common/module.h>

void ADIS16465::print_usage()
{
PRINT_MODULE_USAGE_NAME("adis16465", "driver");
PRINT_MODULE_USAGE_SUBCATEGORY("imu");
PRINT_MODULE_USAGE_COMMAND("start");
PRINT_MODULE_USAGE_PARAMS_I2C_SPI_DRIVER(false, true);
PRINT_MODULE_USAGE_PARAM_INT('R', 0, 0, 35, "Rotation", true);
PRINT_MODULE_USAGE_DEFAULT_COMMANDS();
}

extern "C" int adis16465_main(int argc, char *argv[])
{
int ch;
using ThisDriver = ADIS16465;
BusCLIArguments cli{false, true};
cli.default_spi_frequency = SPI_SPEED;

while ((ch = cli.getOpt(argc, argv, "R:")) != EOF) {
switch (ch) {
case 'R':
cli.rotation = (enum Rotation)atoi(cli.optArg());
break;
}
}

const char *verb = cli.optArg();

if (!verb) {
ThisDriver::print_usage();
return -1;
```

```
}

BusInstanceIterator iterator(MODULE_NAME, cli,
DRV_IMU_DEVTYPE_ADIS16465);

if (!strcmp(verb, "start")) {
return ThisDriver::module_start(cli, iterator);
}

if (!strcmp(verb, "stop")) {
return ThisDriver::module_stop(iterator);
}

if (!strcmp(verb, "status")) {
return ThisDriver::module_status(iterator);
}

ThisDriver::print_usage();
return -1;
}
```

List of Symbols, Abbreviations, and Acronyms

3D	three-dimensional
BSD	Berkley Source Distribution
CS	chip select
DMA	Direct Memory Access
DRDY	data ready
IMU	inertial measurement unit
MEMS	micro-electromechanical system
OS	operating system
PCB	printed circuit board
RTOS	real-time operating system
SMD	surface-mounted device
SPI	serial peripheral interface
UAV	unmanned aerial vehicle
uORB	micro object request broker

1 (PDF)	DEFENSE TECHNICAL INFORMATION CTR DTIC OCA
1 (PDF)	DEVCOM ARL FCDD RLD DCI TECH LIB
2 (PDF)	BOOZ ALLEN HAMILTON M STINGER S HAN
3 (PDF)	DEVCOM ARL FCDD RLS SI J CONROY P SABBAGH FCDD RLH FB A TWEEDELL