ANALOG DEVICES
AHEAD OF WHAT'S POSSIBLE™

# Leveraging a Hardware Agnostic Approach to Ease Embedded Systems Design: The Basics

**Giacomo Paterniani**, Field Applications Engineer


## Abstract

This article demonstrates an approach that accelerates the prototyping phase of embedded system design. It will illustrate how to utilize a hardware agnostic driver in combination with a sensor to make component selection much easier for an entire embedded system. This article describes the components, the typical software structure of an embedded system, and the driver implementation. The subsequent article, "Leveraging a Hardware Agnostic Approach to Ease Embedded Systems Design: Driver Implementation," will further detail the execution.


## Introduction

Using a hardware agnostic driver allows designers to choose the type of microcontroller or processor to manage the sensor without a dependence on hardware. The benefit of this approach is offering the option to add software layers on top of the basic one provided by a supplier, as well as simplifying sensor integration. This article will use an inertial measurement unit (IMU) sensor as an example, but the approach is scalable to other sensors and components. The driver is configured using the C programming language and tested with a generic microcontroller.

## Component Selection

IMU sensors are mostly used for motion detection and to measure the intensity of movements through accelerations and rotational speeds. The ADIS16500 IMU sensor (Figure 1) was selected in this exercise as it allows for a simplified, cost-effective way to integrate accurate, multi-axis inertial sensing into industrial systems, compared with the complexity and investment associated with discrete designs.


*Figure 1. The ADIS16500 evaluation board.*

The main applications are:

- Navigation, stabilization, and instrumentation
- Unmanned and autonomous vehicles
- Smart agriculture and construction machinery
- Factory/industrial automation, robotics
- Virtual/augmented reality
- Internet of moving things

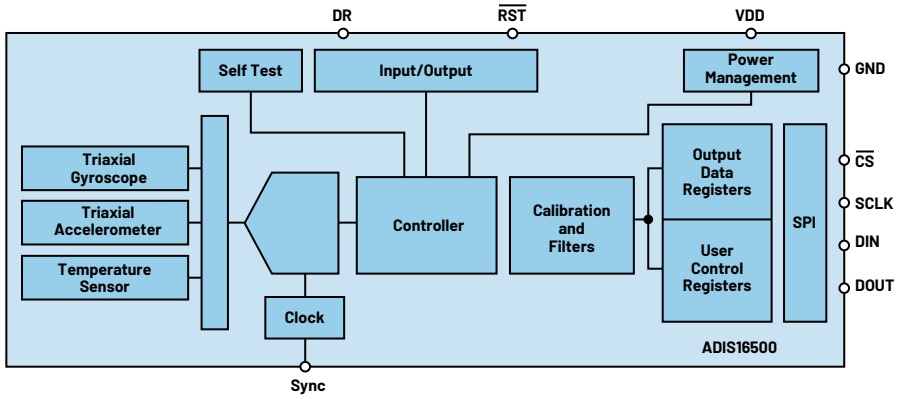


*Figure 2. The ADIS16500 block diagram.*

The ADIS16500 is a precision, miniature microelectromechanical system (MEMS) IMU that embeds a triaxial gyroscope, a triaxial accelerometer, and a temperature sensor. See Figure 2. It is factory calibrated for sensitivity, bias, alignment, linear acceleration (gyroscope bias), and point of percussion (accelerometer location). This means that the sensor measurements are accurate over a broad set of conditions.

This interface allows the microcontroller to write and read the user control registers, as well as read the output data registers from where the accelerometer, gyroscope, or temperature sensor data can be acquired. For that reason, all the software and firmware required to manage the interface has been developed. Figure 2 shows the data ready (DR) pin. This pin is a digital signal that indicates when new data is available to be read from the sensor. The DR pin can be easily managed by a microcontroller, as it can be considered as an input through a general-purpose input/output (GPIO) port.

From a hardware perspective, the IMU sensor and microcontroller will be connected using the SPI interface, which is a 4-wire interface consisting of the nCS, SCLK, $D_{IN}$, and $D_{OUT}$ pins. The DR pin should be connected to one of the microcontroller's GPIOs. The IMU sensor also needs a voltage supply that is between 3 V and 3.6 V, so 3.3 V is sufficient.

## Understanding the Typical Software Structure of an Embedded System

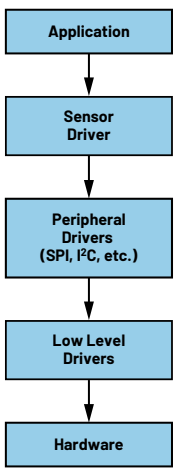


*Figure 3. An SW/FW structure of embedded systems.*

Understanding the generic software and firmware structure of an embedded system is essential to interfacing with a sensor driver. This will help the designer to build a software module that is flexible and easy enough to integrate into any project. Moreover, the driver must be implemented in a modular way, such that the designer can add higher level functions relying on existing ones.

The software structure of an embedded system is pictured in Figure 3. In Figure 3, the hierarchy begins with the application layer, which is where the application code is written. The application layer includes a main file, application modules that rely on the sensor, and modules that rely on peripheral drivers that manage processor configuration. Additionally, within the application layer, there are all the modules related to the tasks that the microcontroller has to process. For example, this includes all the software that manages a task with interrupt or polling, a state machine, and more. That layer level will be different depending on the type of project, so different projects have different codes implemented in it. Within the application layer, all the sensors of the system are initialized and configured in accordance with their data sheets. All the public functions offered by the sensor's drivers are invokable. For example, the read of a register from which data can be output, or a procedure that is writing a register that will change a setting/calibration.

Below the application layer is the sensor's driver layer, which has two types of interfaces. At this level, all functions invokable from the application layer are implemented. Moreover, the function's prototypes are inserted in the driver header file (.h). So, by looking into the header file of a sensor's driver, you can understand the driver's interface and so its invokable functions from higher levels. The lower level layers will be interfaced with peripheral drivers that are specific and dependent on the microcontroller that manages the sensor. The peripheral drivers include all the modules that manage the microcontroller's peripherals such as SPI, $I^2C$, UART, USB, CAN, SPORT, etc. or modules that manage processor internal blocks such as timers, memories, ADCs, etc. They can be called low level functions because they are strictly related to the hardware. For example, each SPI driver is different considering different microcontrollers. Let's look at the ADIS16500 as an example. The interface is the SPI, so its driver will be wrapped with the microcontroller's SPI driver. This will be the same for different sensors and different interfaces. For example, if another sensor has the $I^2C$ interface, then similarly the wrapping with the $I^2C$ driver of the microcontroller will take place in the sensor's initialization procedure.

Below the sensor's driver level are the peripheral drivers, which differ for each type of microcontroller. In Figure 3, there is a split between peripheral drivers and low level drivers. In essence, the peripheral drivers offer the functions of reading and writing through the available communication protocols. Because the low level driver will manage the physical layer of the signals, there's a strong dependence

on the hardware that the designer uses. Usually peripheral and low level driver layers are generated from the integrated development environment (IDE) of the microcontroller thorough the visual tools, depending on the evaluation board on which the microcontroller is mounted.

## Driver Implementation

A hardware agnostic approach enables the use of the same driver in different applications, and hence different microcontrollers or processors. This approach is dependent on how the driver is implemented. To understand the driver implementation, first, we will look at the interface, or the sensor's header file (*adis16500.h*) pictured in Figure 4.

The header file contains useful public macros. This includes register's addresses, SPI max speed, default output data rate (ODR), bitmasks, and the output sensitivity of the accelerometer, gyroscope, and temperature sensor, which are related to the number of bits (16 or 32) with which the data is represented. These macros are reported in Figure 4. Only a few register's addresses are shown to provide an example. The code the article is referring to is available in the appendix.

```
//------------------------------------------------------------//
// ADIS16500 registers
#define ADIS16500_PROD_ID          0x4074

#define ADIS16500_REG_DIAG_STAT    0x02
#define ADIS16500_REG_X_GYRO_L     0x04
#define ADIS16500_REG_X_GYRO_OUT   0x06
#define ADIS16500_REG_Y_GYRO_L     0x08
#define ADIS16500_REG_Y_GYRO_OUT   0x0A
#define ADIS16500_REG_Z_GYRO_L     0x0C
#define ADIS16500_REG_Z_GYRO_OUT   0x0E
#define ADIS16500_REG_X_ACCEL_L    0x10
#define ADIS16500_REG_X_ACCEL_OUT  0x12
#define ADIS16500_REG_Y_ACCEL_L    0x14
#define ADIS16500_REG_Y_ACCEL_OUT  0x16
#define ADIS16500_REG_Z_ACCEL_L    0x18
#define ADIS16500_REG_Z_ACCEL_OUT  0x1A
#define ADIS16500_REG_TEMP_OUT     0x1C
#define ADIS16500_REG_TIME_STAMP   0x1E
// spi max speed in brust mode
#define ADIS16500_BURST_MAX_SPEED  1000000

// default f_odr
#define ADIS16500_DEFAULT_F_ODR    2000 // Sample per second --> [hz]

// masks
#define ADIS16500_MASK_SYNC_MODE   0x000C
#define ADIS16500_MASK_DR_POL      0x0001

// output sensitivities (acceleration and gyroscope)
#define ADIS16500_ACC_SENSITIVITY_32b   5351254.0f    // [LSB/(m/sec^2)]
#define ADIS16500_GYRO_SENSITIVITY_16b  10.0f         // [LSB/(°/sec)]
#define ADIS16500_GYRO_SENSITIVITY_32b  655360.0f     // [LSB/(°/sec)]
#define ADIS16500_TEMP_SCALE_FACT_16b   0.1f          // [°C/LSB]
#define ADIS16500_TS_SCALE_FACT_16b     49.02f        // [usec/LSB]
```

*Figure 4. Macros displayed in the ADIS16500 header file (adis16500.h).*

Figure 3 in the appendix shows all the public variables and public type declarations that can be used by every module including the *adis16500.h*. Here, new types are defined to manage data more efficiently. To provide an example, the *ADIS16500_XL_OUT* type is defined as a structure containing three floats, one for each axis (x, y, and z). There is also an enumeration that allows the sensor to be configured in different ways, giving the designer the flexibility to choose the configuration that best suits their needs. The most interesting part here is the section that makes the driver hardware agnostic. At the beginning of the public variables part (Figure 3 in the appendix), there are three crucial type definitions: pointers to three fundamental functions, or SPI transmission and reception functions and the delay function needed between two SPI accesses to produce the right stall time. These code lines also show the prototype of the function that can be pointed to. The SPI transmission function takes a pointer to the value to be transmitted as input and it returns that can be checked to see if the transmission was successful. The same can be said for the SPI reception function that takes a pointer to a variable, as input, where the value read in reception will be stored. The delay function takes a float as input representing the number of microseconds that the designer wants to wait, and has no return (*void*). In that way, the designer can declare these three functions with these specific prototypes, at the application layer (in the main file for example). Once declared, they can assign the three functions to the fields of an *ADIS16500_INIT* private structure. To better understand this last step, an example is provided in Figure 2 in the appendix.

SPI transmitter, receiver functions, and delay function are declared as static in the main file, so at application level. They are dependent on peripheral driver functions, so the dependence on hardware is outside the sensor driver. The three functions are assigned to the fields of this variable that are pointers to functions. In this way, the designer can wrap the sensor and microcontroller without modifying the sensor driver code. If the designer changes the microcontroller, they only need to adjust the main file by substituting the low level functions inside the three static functions with the appropriate functions for the new microcontroller. This approach makes the driver hardware agnostic because the designer does not need to change the sensor's driver code. Low level functions like *spiSelect, spiReceive, spiUnselect, chThdSleepMicroseconds* etc., are usually already available from the IDE of the microcontroller. In that specific case, the microcontroller evaluation board used was SDP-K1, which embeds an STM32F469NIH6 Cortex®-M4 microcontroller. The IDE, indeed, was ChibiOS, which is a free Arm® development environment.

Figure 4 in the appendix shows prototypes of the invokable functions from the application level. Those prototypes are in the header file of the sensor's driver (*adis16500.h*), along with all the other software and firmware discussed in figures 2 and 3 in the appendix. First, there is the initialization function (*adis16500_init*) that takes a pointer to an *ADIS16500_INIT* structure as input and returns a status code indicating whether the initialization was successful. The implementation of the initialization function is done in the source file (*adis16500.c*) of the sensor's driver. Figure 5 in the appendix shows the code for the *adis16500_init* function. First a type called *ADIS16500_PRIV* is defined, which contains at least all the fields of *ADIS16500_INIT* structure, and then a private variable called *_adis16500_priv* of that type is declared. Within the initialization function all the fields of the *ADIS16500_INIT* structure passed by the application layer will be assigned to the private variable's fields *_adis16500_priv*. This means that any subsequent calls to the sensor driver will use the SPI write and read functions, and the processor delay function, that were passed in by the application layer. This is a key point because it is what makes the sensor driver hardware agnostic. If the designer wants to change the microcontroller, they only need to change the functions that they pass to the *adis16500_init* function. They do not need to modify the sensor driver code itself. At the beginning of the initialization function the initialized field of *_adis16500_priv* variable is set to false because the initialization process has not yet been completed. At the end of the function before the return it will be set to true. Every time the designer calls another public function (Figure 4 in the appendix) the following check is performed: if the *_adis16500_priv.initialized* is true it can proceed, if it is false it will immediately return an error called *ADIS16500_RET_VAL__ERROR*. This is to prevent users from calling a function without first initializing the sensor driver. Continuing with the initialization function discussion, the following steps are performed:

1. Check the product ID, which is known a priori, by reading the *ADIS16500_REG_PROD_ID* register.

2. Set the *Data Ready (DR)* pin polarity by writing the *ADIS16500_REG_MSC_CTRL* register in the appropriate bits field, with the value passed from the application layer (*main.c*).

3. Set the *sync* mode by writing the *ADIS16500_REG_MSC_CTRL* register in the appropriate bits field, with the value passed from the application layer (*main.c*).

4. Set the *decimation rate* by writing the *ADIS16500_REG_DEC_RATE* register, with the value passed from the application layer (*main.c*).

The initialization function depends on the read and write register functions (Figure 6 in the appendix). That is why the above four routines are done after the assignments to the *_adis16500_priv* variable. Otherwise, when the read or write register functions are called, they would not know which SPI transmitter, receiver, and processor delay functions to use.

Referring to Figure 4 in the appendix, there are other public functions that can be invoked after the initialization function. A description of the functionality of the implemented routines is given below, showing the low level ones. The second part of the article will go through details of other driver's implemented functions. All of the following functions must be called only after the initialization function. For this reason, a double check will be done at the beginning of each function to see if the sensor has been initialized or not. If the answer is no, then the procedure immediately returns an error.

- *adis16500_rd_reg_16*

This function is used to read a 16-bit register. Its implementation is available at Figure 6 in the appendix. The inputs are ad that is a *uint8_t* variable representing the address of the register to be read and **p_reg_val* that is a pointer to a variable of *uint16_t* type, that represents where the read value will be assigned. To do a read of a register through the SPI protocol, two SPI accesses are needed; the first to transmit the address, the second to read back the value of the addressed register. In between the two accesses a stall time is required, that is why a delay function is needed. During the first access we transmit the read/write bit, in that case is 1 (R = 1, W = 0), with the register address shifted by 8 bit plus 8 bit at 0, so the following sequence:

R/W | AD6 | AD5 | AD4 | AD3 | AD2 | AD1 | AD0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Where AD stands for address and R/W is the read/write bit.

After a delay, the function reads the value through the SPI and passes it to the input pointer. The registers of the ADIS16500 have a high address containing the high value (8 most significant bits) and a low address containing the low value (8 low significant bits). In order to get the entire value (low and high) of 16 bits it is sufficient to use the low address as ad, because the low and high addresses are consecutive.

- *adis16500_wr_reg_16*

This function is used to write a 16-bit register. Its implementation is available at Figure 6 in the appendix. The inputs are ad that is a *uint8_t* type variable representing the address of the register to be written and *reg_val* that is a *uint16_t* type variable, which represents the value to be written in the register. As for reading function low and high addresses and values are to be taken into consideration. For this reason, according to the data sheet, writing the ADIS16500's register requires two SPI accesses in transmission. The first will send the R/W bit equal to 0 followed by low register address followed by low value, so the sequence will be the following:

R/W | AD6 | AD5 | AD4 | AD3 | AD2 | AD1 | AD0 | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |

Where D stands for data.

The second SPI transmitter access will send the R/W bit equal to 0 followed by high register address followed by high value, so the sequence will be the following:

R/W | AD14 | AD13 | AD12 | AD11 | AD10 | AD9 | AD8 | D15 | D14 | D13 | D12 | D11 | D10 | D9 | D8 |.

The write and read register functions could actually also be defined as private and therefore not visible and invokable from outside the driver software module. The reason they are defined as public is to enable debugging. This allows the designer to quickly access any register in the sensor for reading or writing, which can be helpful for troubleshooting problems.

- *adis16500_rd_acc*

This function reads x, y, z acceleration data from the output data registers, and returns their values in [$m/sec^2$]. Its implementation is available at Figure 7 in the appendix. The input is a pointer to an *ADIS16500_XL_OUT* structure, which simply embeds three fields: x, y, z acceleration, expressed as a float type. The way the acceleration is read is the same for all the three axis, the only differences are the registers to be read. Each axis has its own: x-axis has to be read on x-acceleration output data register, y- and z- axis accordingly. The acceleration value will be represented with a 32-bit value, so the registers to be read are two. One for the most significant 16 bits and one for the least significant 16 bits. For this reason, having a look to the code, there are two register reading accesses with appropriate shift and OR bit operations. These operations allow the entire binary value to be stored on a private *int32_t* variable called *_temp*. At this point binary to two-complement conversion will take place. After the conversion the two-complement value is divided by the sensitivity expressed in [LSB/($m/sec^2$)] so that the final value will be the acceleration expressed in [$m/sec^2$]. This value will be registered in the x, y, or z field of the pointer to the structure that has been passed as input.

- *adis16500_rd_gyro*

The gyroscope reading function does exactly the same as the acceleration reading function. Obviously, it will read x, y, z gyroscope data expressed in [°/sec]. Its implementation is reported in Figure 8 in the appendix. The input of the function is, as for acceleration case, a pointer to an *ADIS16500_GYRO_OUT* structure embedding x, y, and z gyroscope data, expressed as float type. The registers read are the gyroscope output data registers. The binary value will be represented with 32 bits, and the same steps as for acceleration are required to reach the two-complement value. After the binary to two-complement conversion, the value will be divided by the sensitivity expressed in [LSB/(°/sec)], so that the final value will be expressed in [°/sec], and it will be registered in the x, y, or z field of the pointer to the structure that has been passed as input.

## Conclusion

In this article, a typical software/firmware stack of an embedded system has been illustrated. The IMU sensor's driver implementation was introduced. A hardware agnostic approach offers a repeatable method for various sensors or components, even if the interfaces (SPI, $I^2C$, UART, etc.) are different. The subsequent article, "Leveraging a Hardware Agnostic Approach to Ease Embedded Systems Design: Driver Implementation," explains the sensor driver implementation in further detail.

## About the Author

Giacomo Paterniani earned a biomedical engineering degree at University of Bologna. He completed his master's degree in electronics engineering at University of Modena and Reggio Emilia. After graduating, he spent a year as a research fellow at University of Modena and Reggio Emilia. In April 2022, he joined Analog Devices' graduate program as a graduate field applications engineer. In April 2023, he became an FAE.









TA24982-5/24