Application Report
SLAA688–March 2016

PGA411-Q1 Step-by-Step Initialization With Any Host System

Ankur Verma
Fei Xu
Jiri Panacek

Mixed Signal Analog

ABSTRACT

The purpose of this application report is to help with powering up and running the PGA411-Q1 device with any microcontroller or any host system. This report includes the initialization sequence and code snippets for the PGA411-Q1 device. This guide also describes how to interface with the PGA411-Q1 resolver-to-digital converter.

The sections in this document list the recommended sequence to communicate with the PGA411-Q1 device, the step-by-step configuration, and the code snippets for basic functionality.

Contents

List of Figures

1 Introduction

The contents of this application report are provided under the assumption that the setup shown in Figure 1 to interface the PGA411-Q1 device with the microcontroller (MCU) is complete and that the schematic and layout recommendations listed in the PGA411-Q1 data sheet (SLASE76) and the application report, *PGA411-Q1 PCB Design Guidelines* (SLAA697), are followed.

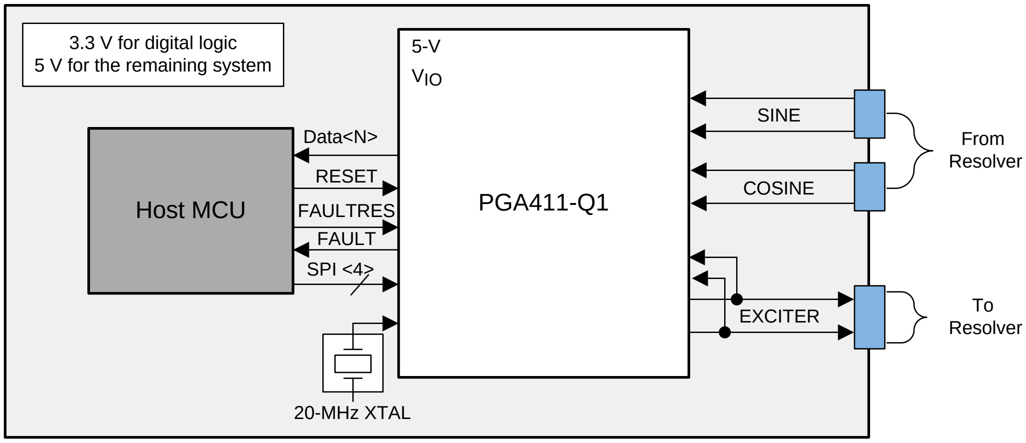


Figure 1. System Block Diagram

2 Step-by-Step Board Initialization

To ensure the custom PGA411-Q1 board functions and is set up properly, the faults are ignored in the first part of the testing process

The second part of the testing process describes how to read the faults in the system.

Use the following command sequence:

Step 1. Apply power (ensure V_{CC}, QVCC, and VIO voltages are in range according to the PGA411-Q1 data sheet).

Step 2. Ensure that the NRESET signal has a low-to-high transition.

Step 3. Wait for the start-up time (approximately 5.2 ms because of the diagnostics checks, ABIST and LBIST, at start-up). See Appendix A for a detailed timing diagram.

Step 4. Check the AOUT signal to ensure that the digital core turns on. See Appendix B for more information about the AOUT pin.

Step 5. Set the FAULTRES pin logic low.

NOTE: An EEPROM configuration is already stored in the PGA411-Q1 device. The default configuration is sufficient to test the initial working of the PGA411-Q1 device, however, for final production, customer-specific EEPROM configuration should be programmed.

If the setup is compatible with the default EEPROM settings (4 V_{RMS} output mode, a 10-kHz excitation signal), no additional register configuration is required to obtain the preliminary angle and velocity data.

In the first part of the testing process, the FAULTRES pin can be held low to keep the exciter output enabled even during fault conditions. Otherwise, if any faults occur in the system, the PGA411-Q1 device enters the FAULT state and the FAULT pin goes into the High-Z state. Running the system for the first time has a high chance for some faults to occur in the system and therefore TI recommends to ignore the fault conditions. Ensure that no high current short circuits occur on the exciter amplifier. See the following NOTE.

NOTE: Toggling the FAULTRES pin with a fault condition still present causes the PGA411-Q1 device to go into normal operation, which may cause damage to the PGA411-Q1 device. This is most likely to occur with high-current short circuits on the exciter amplifier. Ignoring the faults is only recommended for initial evaluation.

For a list of the faults and troubleshooting steps to help remove all fault conditions, see the application report, *Troubleshooting Guide for PGA411-Q1* (SLAA687).

3 Detailed Explanation of SPI Command Sequence

The following list includes the important items to check in the SPI command sequence while communicating with the PGA411-Q1 device:

- The NCS pin is kept low when sending a 32-bit command. One byte is loaded to the SPI shift registers and then sent to the slave device, which occurs 4 times to maintain the 32-bit time frame.
- The clock must be within the specified frequency range (< 8 MHz).
- The SPI does not support back-to-back SPI frame operation. After each SPI transfer, the NCS pin must go from low to high to low before the next SPI transfer can start.
- Each SPI transfer is considered as a 32-bit frame.
- The minimum time, t_{w_cs}, between two SPI commands during which the NCS pin must remain high is 200 ns.

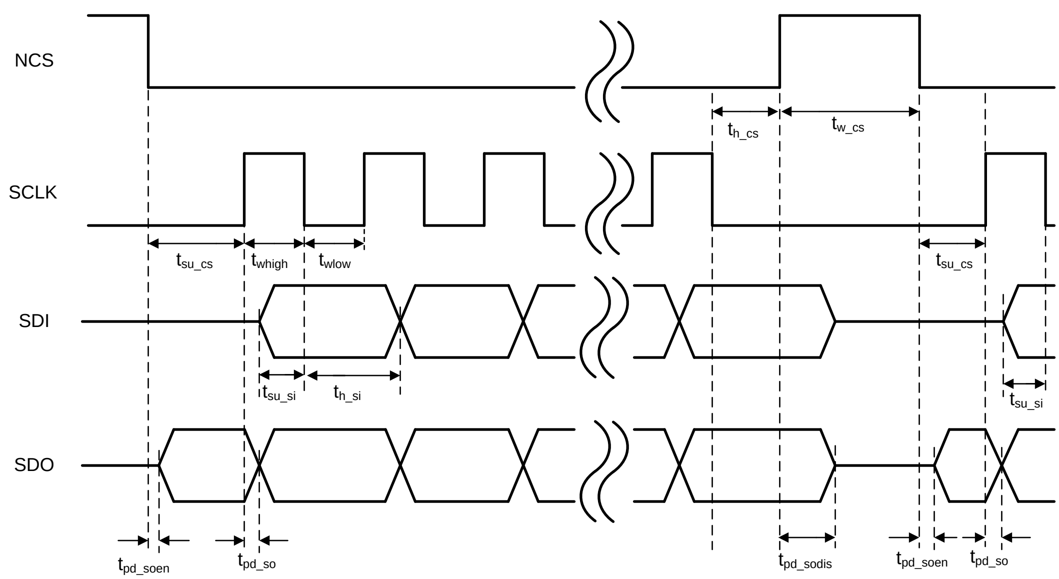
The PGA41x-Q1 SPI includes a four-wire SPI using the following pins:

- The NCS pin which is the SPI chip select (active low) pin.
- The SCLK pin which is the SPI clock pin.
- The SDI pin which is the SPI slave in and master out (SIMO) pin.
- The SDO pin which is the SPI slave out and SPI master in (SOMI, tri-state output) pin.

In this case the resolver-to-digital converter is always configured as a slave device.

The SPI should follow the timing shown in Figure 2.

The SPI frame size is 32 bits, with an MSB-first alignment using the assignments shown in Figure 2.



For timing requirements, see the PGA411-Q1 data sheet (SLASE76).

Figure 2. SPI Timing Diagram

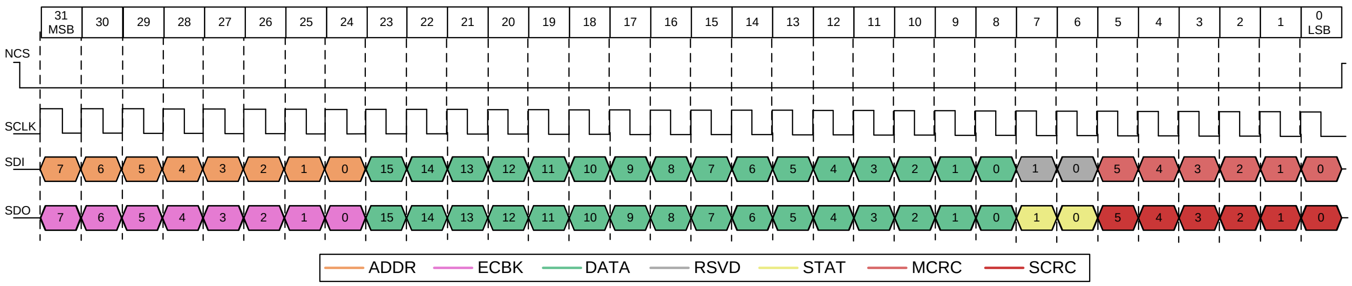


Figure 3. 32-Bit SPI Frame

3.1 SPI Read and Write Communication Timing Examples

Figure 4 shows a practical example of a SPI transfer which represents writing a value of 0x8FC0 to the DEV_OVUV1 register. The entire 32-bit packet corresponds to 0x878FC019 where 0x87 is the DEV_OVUV1 write address, 0x8FC0 is the data to be written, and 0x19 is the CRC-6. Bytes are transferred from the most significant to the least significant. The PGA411-Q1 device returns the content of the previously accessed on the SDO (SOMI) pin. This content is typically discarded after a write operation unless a safety check during a bulk write is implemented.

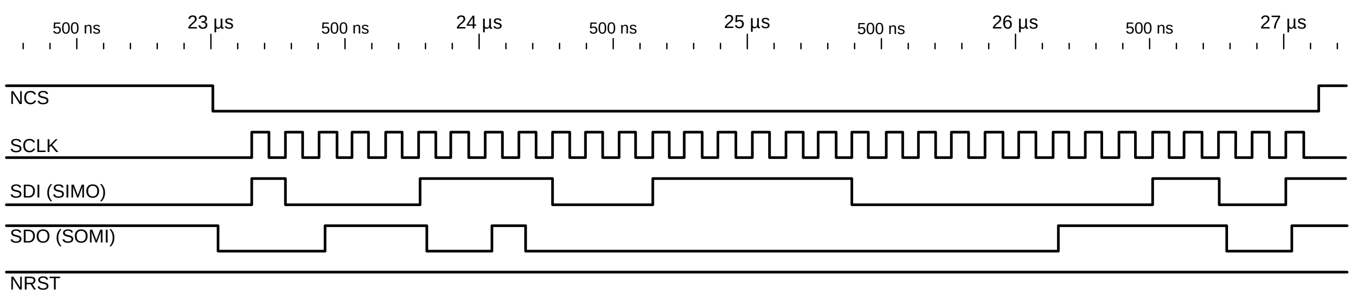


Figure 4. Example SPI Write Operation

The read operation occurs in two phases. The first phase is a dummy read which sets the internal register pointer of the PGA411-Q1 device to the desired DEV_OVUV2 register (see Figure 5). The entire 32-bit packet corresponds to 0x6B00F000 where 0x6B is the DEV_OVUV2 read address, 0x00F0 is the random dummy content of the SPI, and 0x00 is the CRC-6. The return data from this read corresponds to the register that was previously read and can be discarded.

The second phase of the read operation, shown in Figure 6, returns the content of the desired register. The entire 32-bit packet is identical with the previous read. During the second read phase, the PGA411-Q1 device returns the content of the desired DEV_OVUV2 register because this location was accessed in the previous access.

NOTE: Some registers must be unlocked before accessing. Refer to the data sheet for further details.

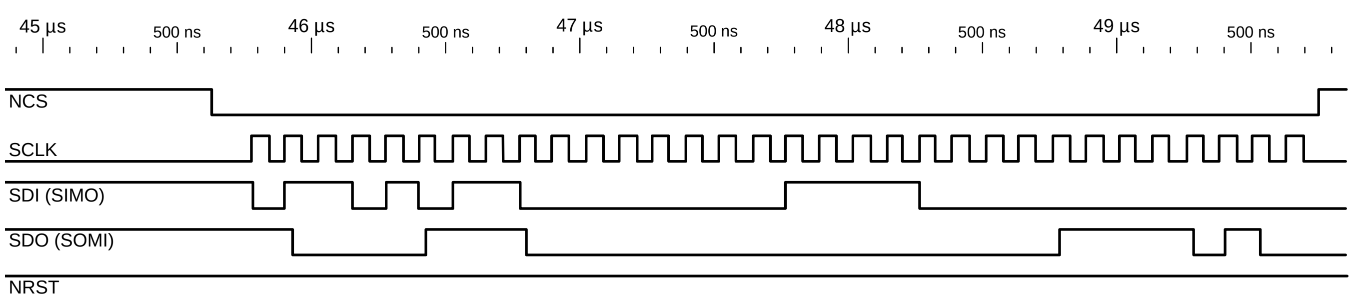


Figure 5. Dummy Read Access

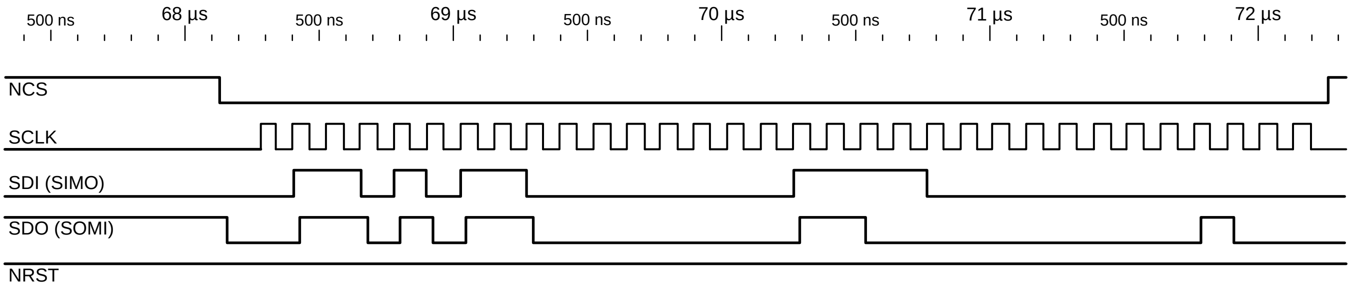


Figure 6. Second Phase of the Read Operation

3.2 SPI Troubleshooting Tips

Use the following steps to troubleshoot the SPI:

1. Verify that the SPI data is clocked in on the falling edge, and the SPI clock polarity and idle value (the clock idle state is low and the slave reacts on a falling edge). The corresponding SPI configuration is CPOL = 0 and CPHA = 1.
2. Check that the CRC-6 code is computed correctly by the MCU. Use the examples provided in the PGA411-Q1 data sheet (SLASE76), troubleshooting application report (SLAA687), and the code snippets listed in Section 4.
3. Composing the packet byte-by-byte using structures can also cause endianness or a structure packing problem. Verify endianness using a scope to ensure the correct byte order (the most significant byte must be transferred first: for example transferring 0x012345678 results in 0x01 as the first byte sent).
4. Verify the correct order of bits (the most significant bit must be transferred first).
5. Ensure that the NRESET signal is logic high.
6. Ensure that the NCS pin is logic low during transmission.
7. Check if the MCU periphery is set to allow 32-bit access. The NCS pin must remain low for the entire 32-bit transfer. If the SPI periphery does not allow a 32-bit wide access to occur, the NCS pin must be controlled manually by software.

4 Example Code Snippets in the C Programming Language

The following sections of code were used together with TI's F28069 Piccolo™ controlSTICK for the C2000™ MCU and TI compiler. Other compilers may require different directives, such as the *const* (flash) directive.

Knowledge of the C programming language is assumed. The following code snippets describe the basic functionality for interfacing with the PGA411-Q1 device. Figure 7 shows the flow for the system reset. The FAULT pin can be tied to an interrupt routine in the microcontroller (not shown in Figure 7). Refer to *Safety Manual for PGA411-Q1 Resolver Sensor Interface* (SLAA684) for additional information on the FAULT usage.

NOTE: All provided code snippets are provided without warranty and should only be used as a starting point.

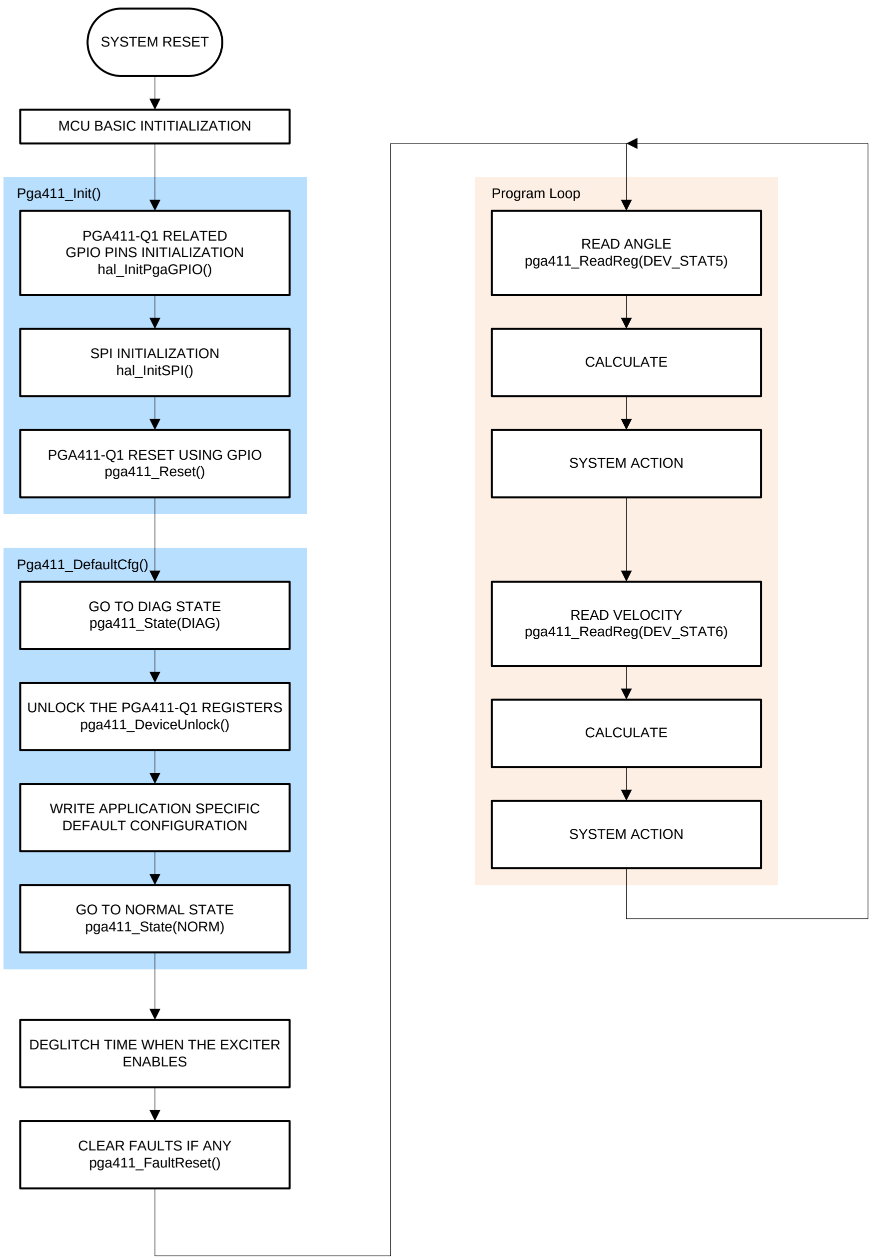


Figure 7. System Reset Flow for TIDA-00796

4.1 Data Types

Defining data types can occur in the header file or in the C file depending on the required level of module encapsulation. The correct file location to start in is a C file. This type of data preparation helps to create a complex structure in the C file which contains all important PGA411-Q1 related data.

4.1.1 PGA411-Q1_SPI_FRAME_T (Header or C File)

This type of definition helps to access individual fields in the frames as well as the entire 32-bit data block. The same structure can be used for both the outgoing and incoming frame, using the small trick with union directive.

```
typedef union
{
    /* outgoing data frame (from master to slave) */
    struct
    {
        /* reverse order in bitfields, starting from bit 0 */
        Uint32    mcrc:6;       /* master's CRC for data, bits 0..5 */
        Uint32    reserved:2;   /* reserved, always 0, bits 6..7  */
        Uint32    dataout:16;   /* data, MSB first, bits 8..23 */
        Uint32    addr:8;       /* register address, bits 24..31 */
    };
    /* incomming data frame (to master from slave */
    struct
    {
        Uint32    scrc:6;       /* slaves's CRC for data, bits 0..5 */
        Uint32    stat:2;       /* status of SPI communication, bits 6..7 */
        Uint32    datain:16;    /* data, MSB first, bits 8..23 */
        Uint32    ecbk:8;       /* register address, bits 24..31 */
    };
    /* and finaly the whole frame */
    struct
    {
        Uint32    frame;
    };
} PGA411_spi_frame_t;
```

4.1.2 PGA411-Q1_REG_T (Header or C File)

This type of definition helps to create a structure of registers containing read and write addresses, real-time values, and default-content values. Mandatory SPI read and write address are placed in permanent memory (FLASH) using the const directive. Using the default value and real-time value depends on the implementation.

```
typedef struct
{
    /* registers definition */
    const Uint16 read_add;   /* SPI read address */
    const Uint16 write_add;  /* SPI write address */
    const Uint16 def_val;    /* default PGA value */
    Uint16 real_val;         /* realtime register content */

} PGA411_reg_t;
```

4.1.3 Macros, Enums (Header File)

These macros store the relative address offset of the PGA411-Q1 registers. Refer to Appendix C or the register maps in the PGA411-Q1 data sheet (SLASE76) for more information on registers.

```
/* PGA411 registers and their offsets */
#define     DEV_OVUV1       0x00
#define     DEV_OVUV2       0x01
#define     DEV_OVUV3       0x02
#define     DEV_OVUV4       0x03
..
..
..
```

These enums can be replaced by macros. The benefit of the enumerated type is auto numbering.

```
enum {READ, WRITE}; /* Enmus selects SPI operation */
enum {DIAG, NORMAL}; /* Operational states of PGA */
```

4.2 Variables

4.2.1 PGA411_REGS (C file)

This array of the register-structure data type is defined as follows. The declaration also defines (initializes) the SPI read and write address for the PGA411-Q1 registers. Refer to Appendix C or the register maps in the PGA411-Q1 data sheet (SLASE76) for more information on registers.

```
PGA411_reg_t PGA411_regs[PGA411_REG_COUNT] =
{
    /* memory constants init */
    [DEV_OVUV1]     = {.read_add = 0x53, .write_add = 0x87, .def_val = 0xBFC0},
    [DEV_OVUV2]     = {.read_add = 0x6B, .write_add = 0x26, .def_val = 0x00C0},
    [DEV_OVUV3]     = {.read_add = 0x65, .write_add = 0x17, .def_val = 0xFCFF},
    ..
    ..
    ..
```

4.3 Functions

4.3.1 CRC Calculation

This CRC calculation routine is optimized for the PGA411-Q1 example code running on a 32-bit C2000 MCU little endian. The input of the function corresponds to an entire 32-bit frame but only the higher 24 bits are used for the calculation. TI recommends to check the output of the routine with the SPI CRC examples listed in the PGA411-Q1 data sheet.

```
/* CRC6 calculation - optimized for PGA411 */
Uint16 PGA411_crc2 (Uint32 datin)
{
    Uint16 byte_idx, bit_idx, crc= (CRC_INITSEED «2);

    /* byte by byte starting from most significant (3-2-1) */
    for(byte_idx = CRC_BYTECOUNT; byte_idx >= 1; byte_idx--)
    {
        /* XOR-in new byte from left to right */
        crc ^= ((datin » (byte_idx«3)) & 0x000000FF);
        /* bit by bit for each byte */
        for(bit_idx = 0; bit_idx < 8; bit_idx++)
        {
            crc = crc « 1 ^ (crc & 0x80 ? (CRC_POLYNOM«2) : 0);
        }
    }
    return(crc » 2 & CRC_INITSEED); /*restore two bit offset */
```

4.3.2 SPI Communication Trigger

Calling this function triggers SPI communication. The *hal_Xmit4BSPI* and *hal_assert* functions are functions defined in the hardware abstraction layer. Depending on the *dir* parameter, the appropriate SPI address is selected. In case the CRC calculation for the incoming SPI frame fails, program termination is called.

```
/* Xmit data to PGA over SPI, for reg use defined macros */
PGA411_spi_frame_t PGA411_XmitSPI(Uint16 dir, Uint16 reg, Uint16 wdata)
{
    PGA411_spi_frame_t out, in;

    /* do we read data ? */
    if(dir == READ) out.addr = PGA411_regs[reg].read_add; /* read address */
    /* or write data ? */
    else out.addr = PGA411_regs[reg].write_add; /* write address */
    /* compose the rest of the frame */
    out.dataout = wdata; /* real data (W) or dummy data (R) */
    out.reserved = 0x00; /* always zero */
    out.mcrc = PGA411_crc2(out.frame); /* calculate TX CRC */
    in.frame = hal_Xmit4BSPI(out.frame);

    /* check RX CRC */
    if(PGA411_crc2(in.frame) != in.scrc)
        {
        hal_assert(); /* if eeror -> terminate */
        }
    return(in);
}
```

4.3.3 Reading Data from Register

This function makes reading data from a register easier.

```
/* Read data from defined */
Uint16 PGA411_ReadReg(Uint16 reg)
{
    /* first read returns whatever */
    PGA411_XmitSPI(READ, reg, SPI_DUMMY);
    /* second read returns requested data */
    return (PGA411_XmitSPI(READ, reg, SPI_DUMMY).datain);
}
```

Example use:

```
i = PGA411_ReadReg (DEV_OVUV1);
```

4.3.4 Writing Data to a Register

This function makes writing data to a register easier.

```
/* Write data to defined register */
void PGA411_WriteReg(Uint16 reg, Uint16 data)
{
    /* here we just making it nice, can be macro too */
    PGA411_XmitSPI(WRITE, reg, data);
}
```

Example use:

```
PGA411_WriteReg (DEV_OVUV1, 0x8FC0);
```

4.3.5 Switching Between Diagnostics and Normal State

This function changes between the DIAGNOSTICS state and the NORMAL state based on the state parameter (defined by the enum).

NOTE: After NRESET, the transition from the DIAGNOSTICS state to the NORMAL state is automatic (if no fault is present).

When the DIAGNOSTICS state is requested through SPI, then the automatic transition does not happen and the PGA411-Q1 device waits until the SPI transaction sets the DIAGEXIT bit to take the device back to the NORMAL state.

```
/* Change state diagnostic/normal */
void PGA411_State (Uint16 state)
{
    Uint16 temp;
    /* Enter Diagnostic state */
    if(state == DIAG)
    {
        /* read content of  DEV_CONTROL3 register */
        temp = PGA411_ReadReg(DEV_CONTROL3);
        temp |= 0x0004; /* set bit SPIDIAG */
        PGA411_WriteReg(DEV_CONTROL3, temp); /* finish R-M-W */
    }
    /* Go back to normal state */
    else
    {
        /* read content of  DEV_CONTROL1 register */
        temp = PGA411_ReadReg(DEV_CONTROL1);
        temp |= 0x0001; /* set bit DIAGEXIT */
        PGA411_WriteReg(DEV_CONTROL1, temp); /* finish R-M-W */
    }
}
```

4.3.6 Unlocking the Device

This function performs the device unlocking procedure. Refer to the PGA411-Q1 data sheet for more information.

NOTE: The PGA411-Q1 device must be in the DIAGNOSTICS state and the complete unlock procedure must be completed within 10 ms.

```
/* Device unlock (must be in diagnostic state) */
void PGA411_DeviceUnlock (void)
{
    PGA411_WriteReg(DEV_UNLK_CTRL1, 0x000F);
    PGA411_WriteReg(DEV_UNLK_CTRL1, 0x0055);
    PGA411_WriteReg(DEV_UNLK_CTRL1, 0x00AA);
    PGA411_WriteReg(DEV_UNLK_CTRL1, 0x00F0);
}
```

4.3.7 Unlocking EEPROM

This function unlocks the EEPROM registers. Refer to the PGA411-Q1 data sheet for more information.

NOTE: The PGA411-Q1 device must be in the DIAGNOSTICS state and the complete unlock procedure must be completed within 10 ms.

```
/* EEPROM unlock (must be in diagnostic state) */
void PGA411_EEPROMUnlock (void)
{
    PGA411_WriteReg(DEV_EE_CTRL4, 0x000F);
    PGA411_WriteReg(DEV_EE_CTRL4, 0x0055);
    PGA411_WriteReg(DEV_EE_CTRL4, 0x00AA);
    PGA411_WriteReg(DEV_EE_CTRL4, 0x00F0);
}
```

4.4 Reading Angle and Velocity

The angle and velocity data are also available in the register memory space and can be polled through the SPI. The corresponding locations are the ORDANGLE bit in the DEV_STAT5 register for angle data value and the ORDVELOCITY bit in the DEV_STAT6 register for velocity data value.

Use the following equations to convert the PGA411-Q1 parallel output or SPI data into meaningful angle and velocity values:

- 10-bit angle:

$$\varphi\ (\text{degrees}) = 360 \times \frac{\text{ORDx}}{2^{10}} \tag{1}$$

- 12-bit angle:

$$\varphi\ (\text{degrees}) = 360 \times \frac{\text{ORDx}}{2^{12}} \tag{2}$$

- 10-bit velocity:

$$\vartheta\ (\text{RPM}) = 60 \times \frac{f_{\text{clk}} \times (\text{ORDx} + 1)}{2^{21}}$$

where

- where f_{clk} is the device clock frequency (typically 20 MHz) (3)

- 12-bit velocity

$$\vartheta\ (\text{RPM}) = 60 \times \frac{f_{\text{clk}} \times (\text{ORDx} + 1)}{2^{25}} \tag{4}$$

The angle data read from PGA411-Q1 device is in hexadecimal format and is an unsigned 16-bit integer. Only the lower 12 bits contain the angle information. A value of 0 represents 0 degrees. Therefore a value of 4096 represents 360 degrees. Use Equation 5 to calculate the angle in degrees.

$$\text{Angle (degrees)} = 360 \times \frac{\text{Angle (hexadecimal)}}{4096} \tag{5}$$

4.4.1 Angle Calculation Example

The following code snippet reads angle data from the PGA411-Q1 device, calculates the corresponding angle and prints the result over a serial port to the host system.

The provided example uses Texas Instruments' IQmathLib library (processors.wiki.ti.com/index.php/IQmath_Library_for_C28x). An alternate approach uses the traditional float or fixed point (Q number format) arithmetic. The final approach should be considered based on the hardware architecture.

```
/* angle calculation for the PGA411 using IQmathLib library */
/* _iq19 stores -4096 to 4095.999998093 with 0.000001907 resolution */
{
    Uint16 angle_raw; /* raw data from the PGA411 register */
    _iq angle; /* IQ math angle */

    /* read data from the PGA411 */
    angle_raw = pga411_ReadReg(DEV_STAT5);
    angle_raw &= 0x1FFF; /* preserve only ORDANGLE bits */

    /* multiply minimal step (LSB) by ORDANGLE value */
    angle = _IQ19mpy(_IQ19(360.0/4096), _IQ19(angle_raw)); /* 12b example */

    /* convert IQ variable to string */
    _IQ19toa(outbuf, "%4.5f", angle);

    /* print-out the buffer */
    hal_PutsUART(outbuf);
}
```

4.4.2 Velocity Calculation Example

Similarly to the angle calculation example, the next code snippet reads velocity data from the PGA411-Q1 device, calculates velocity and prints the result over a serial port.

The velocity calculation is more complex than the angle calculation. Velocity data (10 or 12 bit) from the PGA411-Q1 are stored in 2's-complement format. However, because of different word size, the data must be first converted. This conversion occurs as follows:

Step 1. Check the sign (bit 11).

Step 2. Convert to the positive integer.

Step 3. Preserve only the ORDVELOCITY bits.

Step 4. Cast to float format.

Step 5. Multiply by –1 if the sign is negative

The IQmathLib library is used for the final velocity calculation and string conversion. The final implementation should consider target architecture, especially float and negative integer data representation.

```
       /* velocity calculation for the PGA411 using IQmathLib library */
    /* _iq14 stores -131072 to 131071.999938965 with 0.000061035 resolution */
    {
        Int16 velocity_raw; /* raw data from the PGA411 register */
        float velocity_float; /* temp for 2nd complement to float conversion */
        _iq velocity; /* IQ math velocity */

        /* read data from the PGA411 */
        velocity_raw = pga411_ReadReg(DEV_STAT6);

        /* convert 2nd complement to float */
        if(velocity_raw & 0x0800) /* negative number ? */
        {
            /* convert to positive number first */
            velocity_float = (((~velocity_raw) +1) & 0x07FF);
            velocity_float *= -1; /* and make it negative */
        }
        else
        {
            /* positive number, preserve only needed bits */
            velocity_float = (velocity_raw & 0x07FF);
        }

        velocity = _IQ14mpy(_IQ14(60*20000000/33554432),_IQ14(velocity_float + 1)); /* 12b
example */

        /* convert IQ variable to string */
        _IQ14toa(outbuf, "%7.5f", velocity);

        /* print-out the buffer */
        hal_PutsUART(outbuf);
    }
```

Additional Troubleshooting Tips

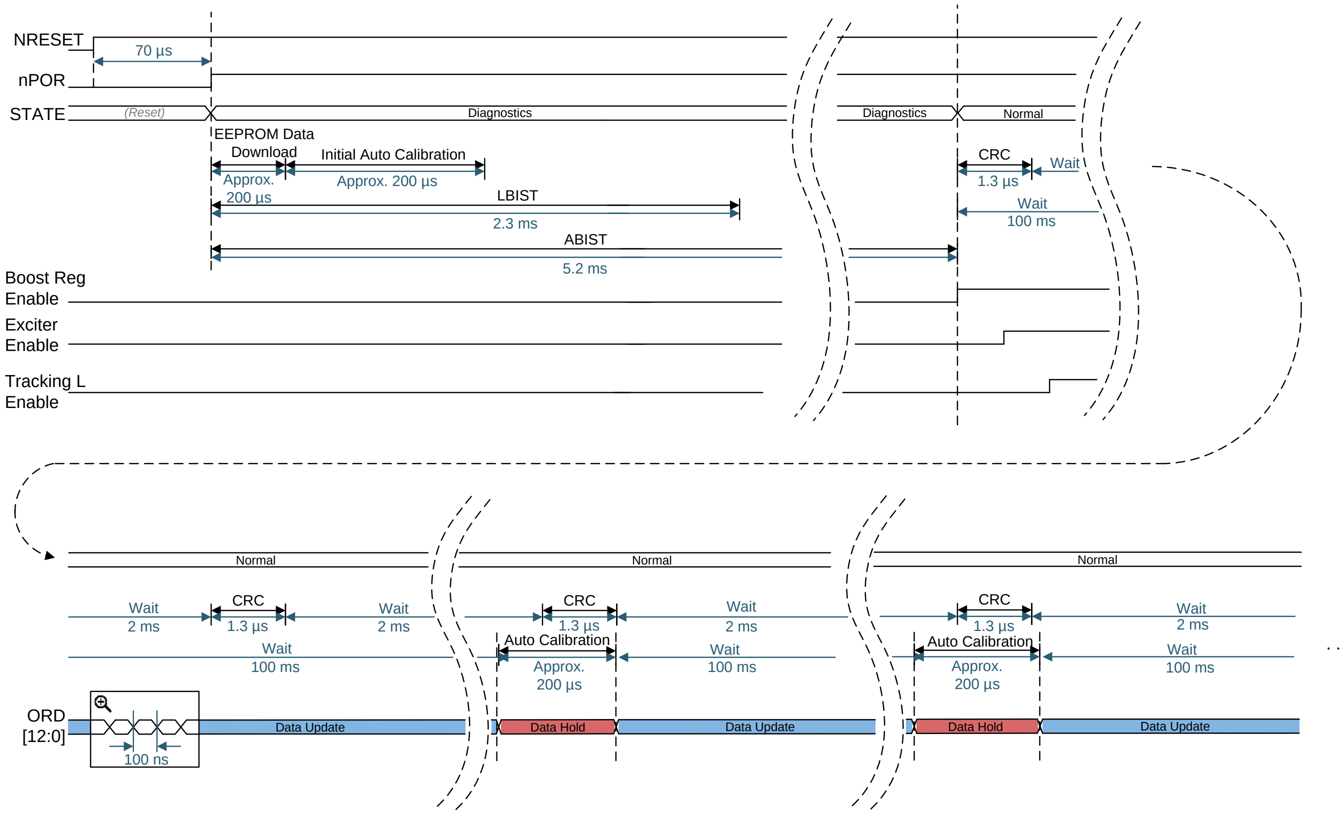


Figure 8. Power-Up Timing Diagram

AOUT Pin

The AOUT pin represents the analog version of the angle output. The PGA411-Q1 device has internal 10-bit DAC to convert the digital parallel output to an analog value between 0.5 V and 4.5 V. This feature is intended to be used only for initial evaluation and debug. Figure 9 shows the analog voltage representation of the angle. Figure 10 shows the AOUT performance when the RDC is working correctly and the resolver sensor is rotating with constant velocity. The waveform is the same as the one shown in Figure 10 even if the tracking loop is trying to lock. The AOUT has a fixed angular position when the resolver is not moving. Figure 11 shows the OE1, OE2, and AOUT pins for a complete picture of the system function.

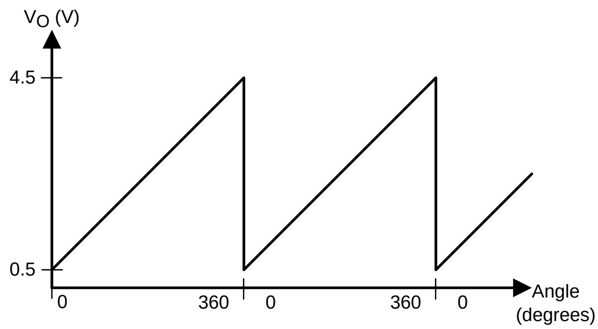


Figure 9. Analog Angle Output

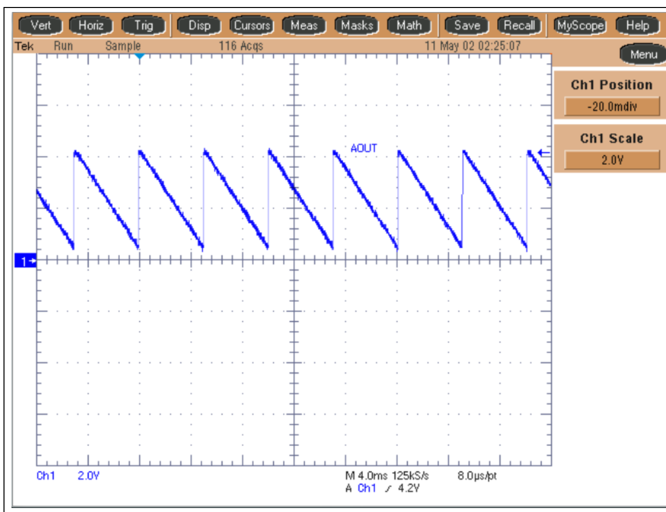


Figure 10. Test Output of Analog DAC

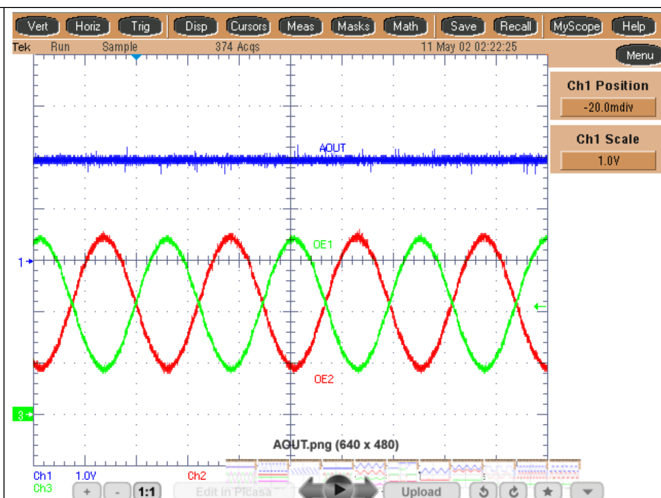


Figure 11. Tracking Loop Locked (AOUT Output is a Fixed Value)

SPI Memory Map

Table 1 provides an overview of each register located in the PGA411-Q1 SPI memory map.

Table 1. SPI Memory Map

Register Location	Register Name	SPI Read Address	SPI Write Address	Write State	Configuration CRC
0x00	DEV_OVUV1	0x53	0x87	DIAG	YES
0x01	DEV_OVUV2	0x6B	0x26		Yes
0x02	DEV_OVUV3	0x65	0x17		Yes
0x03	DEV_OVUV4	0xEC	0x39		Yes
0x04	DEV_OVUV5	0x52	0x75		Yes
0x05	DEV_OVUV6	0xE9	0x83		Yes
0x06	DEV_TLOOP_CFG	0 xA6	0x42		Yes
0x07	DEV_AFE_CFG	0xC2	0x91		Yes
0x08	DEV_PHASE_CFG	0 x57	0x85		Yes
0x09	DEV_CONFIG1	0xBE	0xEB		Yes
0x0A	DEV_CONTROL1[(1)]	0x90	0x0D		Yes
0x0B	DEV_CONTROL2[(1)]	0x63	0x38		Yes
0x0C	DEV_CONTROL3	0xDD	0xAE	All	No
0x0D	DEV_STAT1	0x81	N/A (Read-only register)		No
0x0E	DEV_STAT2	0x4D			No
0x0F	DEV_STAT3	0x84			No
0x10	DEV_STAT4	0x1F			No
0x11	DEV_STAT5	0x41			No
0x12	DEV_STAT6	0x6F			No
0x13	DEV_STAT7	0xE1			No
0x14	DEV_CLCRC	0x4F	0xFC	DIAG	No
0x15	DEV_CRC	0x0F	0xE7		No
0x16	CRCCALC	0xD9	N/A (Read-only register)		No
0x17	DEV_EE_CTRL1[(2)]	0xE3	0x6E	DIAG	No
0x18	DEV_CRC_CTRL1	0x7A	0xB6		No
0x19	DEV_EE_CTRL4	0xBA	0x56		No
0x1A	DEV_UNLK_CTRL1	0x64	0x95		No

NOTE: The configuration registers that are accessible only in the DIAGNOSTICS state cannot be written in case the PGA411-Q1 device is in the NORMAL operating state. Any attempt to do so causes the DEV_STAT1 register STAT[1:0] = 0b11 response (such as *Address Not Found*). However, these registers can be read in any state.

(1) Device unlock sequence required in order to access register.
(2) EEPROM space unlock sequence required in order to access register.