Application Note

Firmware Guide for TCAN24xx and TCAN28xx Devices

TEXAS INSTRUMENTS

Parker Dodson

ABSTRACT

The TCAN28xx and TCAN24xx family of mid-range SBCs, are a great choice for many automotive applications. As with many SBC's there is a lot of variance in the potential configurations of the device that can be implemented through user configurable registers. This application note aims to explain how to communicate with the SBC's through a SPI, the register structure of the device, suggested firmware implementations, and finally an illustration of some more complex device configuration options.

Table of Contents

Trademarks

All trademarks are the property of their respective owners.

1 Introduction

The TCAN28xx and TCAN24xx families of SBC's offers the end designer various configuration options to be best designed for the needs of the system. With many options available to a designer approaching the firmware configuration of this device can seem daunting. However, with the understanding of a few basic concepts the configuration of the device can become clearer. This note aims to explain the underlying mechanisms of how to communicate with the device and configure this for a needed application use case. First, the note can start with how the SPI communication works on the TCAN28xx and TCAN24xx line of devices. Second, the note can dive into how the registers are setup inside of the TCAN28xx and TCAN24xx line of devices. Third, a more detailed look into suggested data structures to empower simple configurations. Finally, a brief overview of three varied configurations (device setup, partial networking, Watchdog timer) can be examined to give the designer a more holistic understanding of how to configure these devices.

2 SPI Communication

All device configuration takes place through the use of a 4-wire SPI. On the TCAN28xx and TCAN24xx families of devices the four SPI pins are SDO (Serial Data Out), SDI (Serial Data In), CLK (SPI Clock), and nCS (active low chip select line). These are analogous to common SPI signal naming SDO = CIPO (controller in, peripheral out), SDI = COPI (controller out, peripheral in), CLK = SCLK (SPI Clock), and nCS has the same naming convention – this assumes that the SBC is always a peripheral to a main system controller – which is true as these are mid-range SBCs that do not integrate a controller.

The SBCs supports SPI modes 0 through 3, but uses mode 0 as default. The mode changes the CPOL (clock polarity) and CPHA (clock phase) of the SPI communication. CPOL, or clock polarity, sets the default level of the SPI clock – in modes 0 and 1 this is low and in modes 2 and 3 this is set to high. CPHA, or clock phase, deals with when data is sampled and when this is shifted – in modes 0 and 2 data is sampled on rising edge and shifted on a falling edge where modes 1 and 3 can sample on the falling edge and shift on the rising edge.

Table 2-1. SPI Mode Description

Mode	CPOL	CPHA	Clock Phase
0	0	0	Data sampled on rising edge and shifted on falling edge
1	0	1	Data sampled on falling edge and shifted on rising edge
2	1	0	Data sampled on rising edge and shifted on falling edge
3	1	1	Data sampled on falling edge and shifted on rising edge

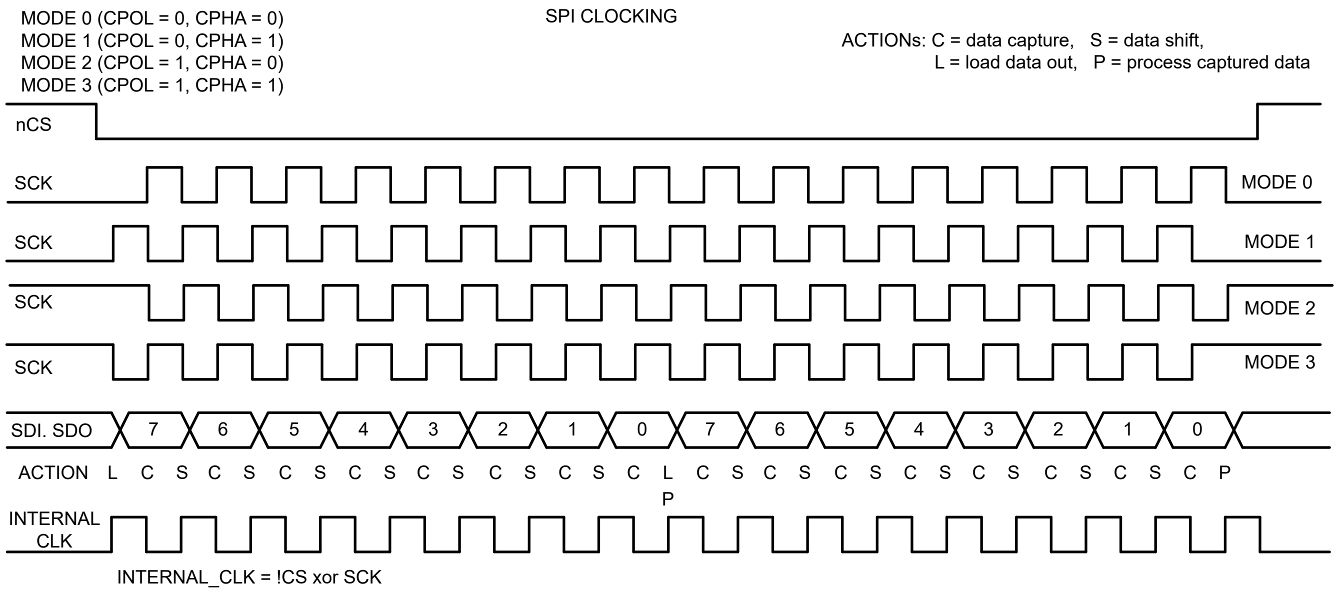

MODE 0 (CPOL = 0, CPHA = 0)
MODE 1 (CPOL = 0, CPHA = 1)
MODE 2 (CPOL = 1, CPHA = 0)
MODE 3 (CPOL = 1, CPHA = 1)
SPI CLOCKING
ACTIONs: C = data capture, S = data shift,
L = load data out, P = process captured data
nCS
SCK
MODE 0
SCK
MODE 1
SCK
MODE 2
SCK
MODE 3
SDI. SDO
7 6 5 4 3 2 1 0 7 6 5 4 3 2 1 0
ACTION L C S C S C S C S C S C S C S C L C S C S C S C S C S C S C S C P
P
INTERNAL CLK
INTERNAL_CLK = !CS xor SCK

Figure 2-1. SPI Clocking Vs. SPI Mode

The SBCs use 7-bit addresses for each register and in general unsigned 8-bit integers (uint8_t) need to be used when working with the SPI data on these devices. In general, to perform a write command the base address needs to be shifted left by 1 bit and the new bit in the zero position need to be set to one. If a read operation is to be performed the 7bit address need to be shifted left and the new bit in position 0 need to be set to 0 (if using unsigned integers, the bit shifted in can be 0). An example of a read and write address of a base address of 0x10 is shown in the following.

```
uint8_t baseAddress = 0x10;
uint8_t writeAddress = (baseAddress << 1) | 0x01;
uint8_t readAddress = (baseAddress <<1) & 0xFE;
```

The device supports either one byte or two-byte messages – but by default uses one-byte messages.

For one-byte communication – with no CRC included – there are two commands that can be used – a write and a read. The read command starts when nCS pin transitions from low to high and at this time the CLK pin need to see a clock signal on input (in accordance with SPI mode chosen). At the same time the modified read address can be sent to the SDI pin while simultaneously the SDO pin can output the global interrupt vector which gives a high-level view of device operation and potential errors. After the modified address has been sent the SDI line can go quiet and the SDO pin can transmit the contents of the requested register back to the controller. After the register has transmitted the contents to the controller the nCS pin can pull high and the transaction is complete.

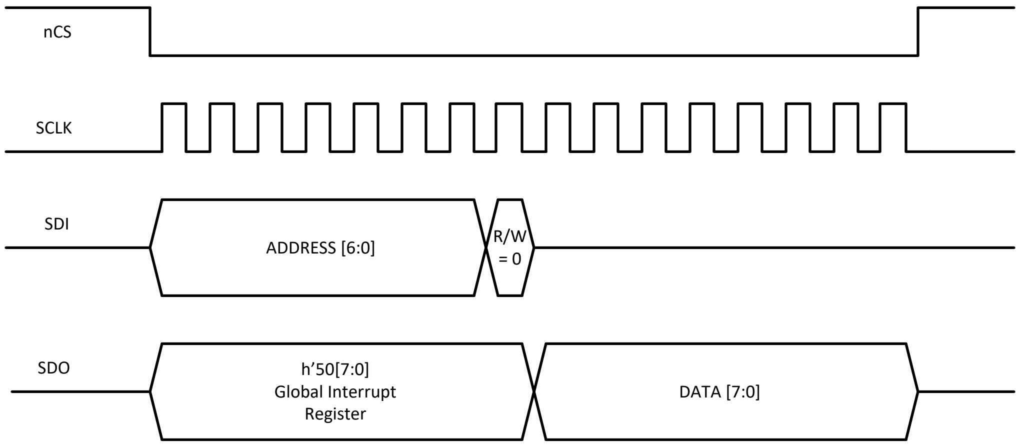


Figure 2-2. SPI Read - One Byte Mode

One byte write mode is very similar to the read except the 8th bit sent is 1 instead of 0 to indicate a write action and the data byte is actually sent from the controller to the SBC.

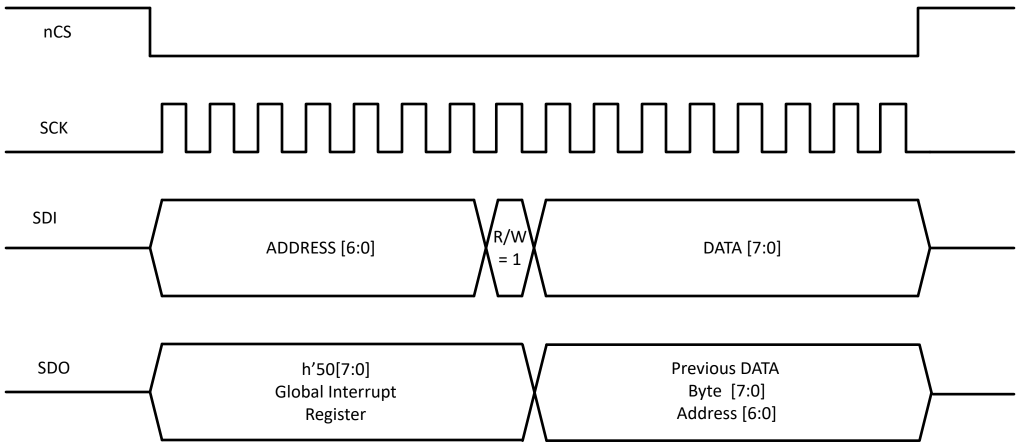

nCS
SCK
SDI
ADDRESS [6:0]
R/W = 1
DATA [7:0]
SDO
h'50[7:0] Global Interrupt Register
Previous DATA Byte [7:0] Address [6:0]

Figure 2-3. SPI Write - One Byte Mode

For two-byte communication the process is very similar to one-byte mode. First – to configure the device for two-byte mode the first SPI transaction can be in the default mode which supports one-byte transactions. To change to two-byte mode, after the device has been started and is in standby mode a write transaction to register 9h must be made with the bit in bit field #3 to be set to 0b1 instead of the 0b0 that is default. This can change the device to two-byte transactions. If other configurations of the SPI are needed, the configurations are mainly contained at register address 9h.

In two-byte mode the SPI communication is still very similar to one-byte mode. Some major differences are as follows. Register writing and reading must be done on sequential registers meaning that if you specify address 10h the two-byte transaction can apply to register 10h and 11h. The second major difference is that SPI speed is no longer limited to 4MHz max, but instead limited to 2MHz max. Each two-byte mode transaction contains 3 discrete byte periods. For two-byte mode read operations the SDI pin receives the 7-bit address and read or write bit for the first byte and is not used for the rest of the transaction; while the SDO pin can output the global interrupt register at the same time as the initial command has been placed then this can be followed by the contents of the register at the selected address further followed by the contents of the register at the selected address + 1. For example, this means that if you are reading register 10h and send command 0b00100000 the SDO pin can respond with register contents for address 10h and 11h while in two-byte mode. Write transactions are also fairly straightforward but the SDI pin can have an input for the entire 3-byte period where the first byte is address + command bit, followed by the data for selected address, finishing with a final byte for the selected address + 1. To exemplify this if you want to write 0xFF and 0xAA respectively to registers 10h and 11h the write command can have the first byte be 0b00100001, with the second byte 0b11111111, followed by 0b10101010.

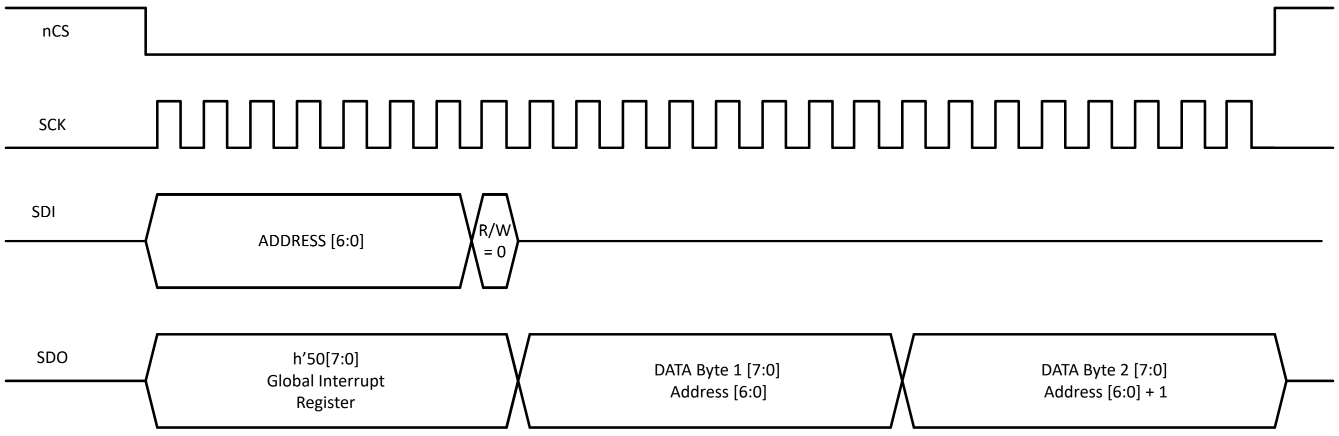

nCS
SCK
SDI
ADDRESS [6:0]
R/W = 0
SDO
h’50[7:0] Global Interrupt Register
DATA Byte 1 [7:0] Address [6:0]
DATA Byte 2 [7:0] Address [6:0] + 1

Figure 2-4. SPI Read - Two Byte Mode

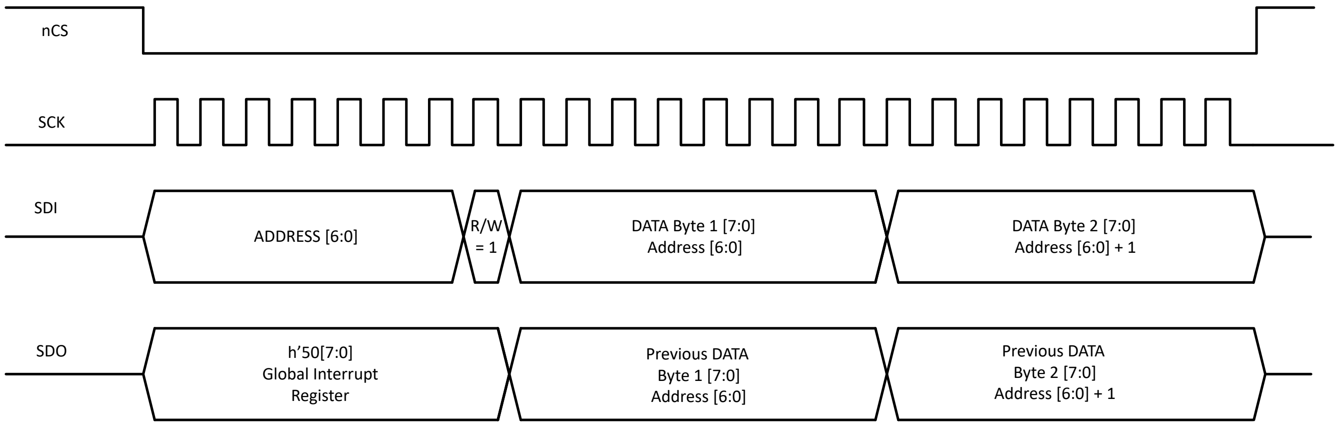

nCS
SCK
SDI
ADDRESS [6:0]
R/W = 1
DATA Byte 1 [7:0] Address [6:0]
DATA Byte 2 [7:0] Address [6:0] + 1
SDO
h’50[7:0] Global Interrupt Register
Previous DATA Byte 1 [7:0] Address [6:0]
Previous DATA Byte 2 [7:0] Address [6:0] + 1

Figure 2-5. SPI Write - Two Byte Mode

The final main transaction type allowed by the TCAN28xx and TCAN24xx lines of devices is one-byte mode + CRC which has transactions that take up a 3-byte time period. The device must be in one-byte mode with CRC enabled to allow for this configuration. The device, when SPI CRC is enabled, supports CRC8H2F, which takes the form X^8 + X^5 + X^3 + X^2 + X + 1 but can be configured to support CRC8 SAE J1850. For the one-byte CRC write operation this starts exactly the same as a typical one-byte write transaction, but after the data that was sent to the register has been transferred a 8-bit CRC of the address + R/W bit + data is input into the device where the SDO pin reads back the global interrupt vector, the previous data, and includes an 8-bit CRC of the global interrupt vector. The read transaction is a bit different – this is initiated in the same way as a one-byte SPI transaction, but instead of not expecting any input after the initial address and R/W bit have been transmitted a filler byte of 0x00 is sent followed by an 8-bit CRC of address + R/W bit + filler byte while the SDO pin can send the global interrupt vector, followed by the contents of selected register, finally followed by 8-bit CRC of the global interrupt vector and data of the selected register. This option can be needed in systems where data integrity on the SPI bus between controller and SBC is critical to system requirements and application success.

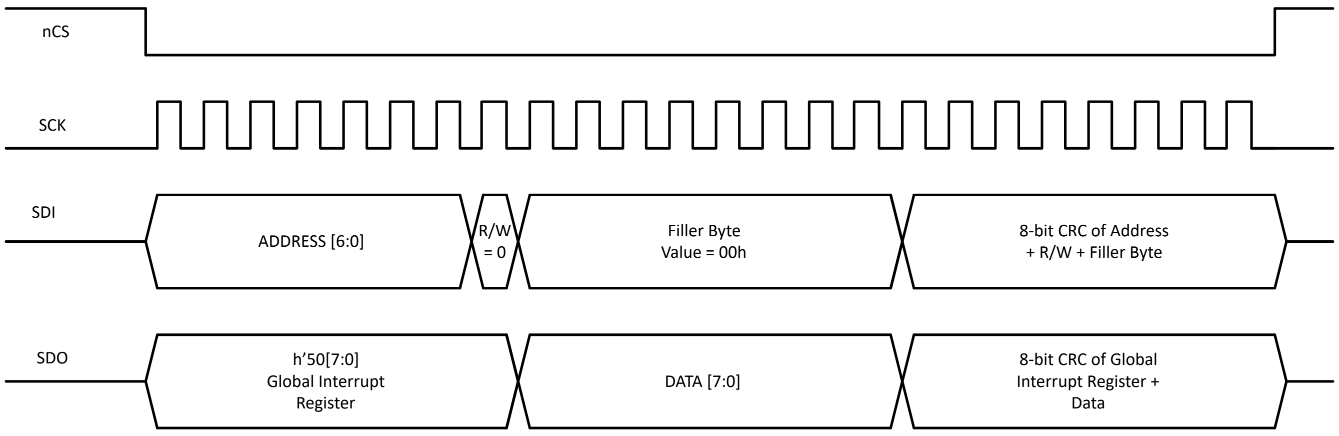

nCS
SCK
SDI
ADDRESS [6:0]
R/W = 0
Filler Byte Value = 00h
8-bit CRC of Address + R/W + Filler Byte
SDO
h'50[7:0] Global Interrupt Register
DATA [7:0]
8-bit CRC of Global Interrupt Register + Data

Figure 2-6. SPI Read - One Byte Mode + CRC

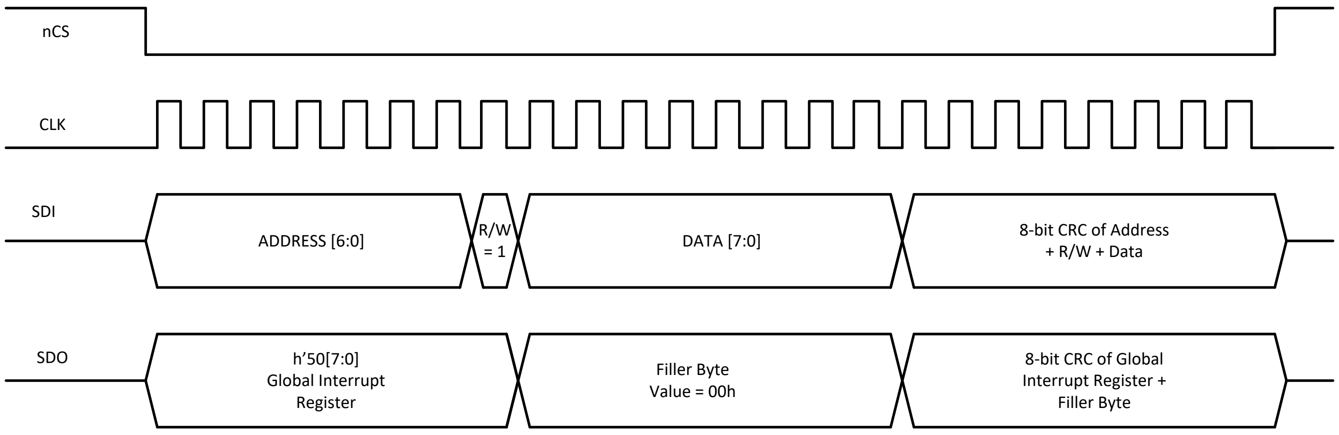

nCS
CLK
SDI
ADDRESS [6:0]
R/W = 1
DATA [7:0]
8-bit CRC of Address + R/W + Data
SDO
h'50[7:0] Global Interrupt Register
Filler Byte Value = 00h
8-bit CRC of Global Interrupt Register + Filler Byte

Figure 2-7. SPI Write - One Byte Mode + CRC

The type of transaction is not the only consideration when configuring SPI with the TCAN28xx and TCAN24xx lines of devices – as this device supports SPI modes 0, 1, 2, and 3 in addition to the variance of transaction type. The SPI mode is independent of the transaction type and this concerns the clock idle polarity (CPOL) as well as the clock phase (CPHA). By default, the SBC is in SPI mode 0, which is the most common form of the SPI bus, in this mode CPOL = 0 which means the clock has an idle state of *low* and CPHA = 0 meaning that the data is sampled on the rising edge of the clock signal. However, the other modes 2, 3, and 4 can vary the different combinations of CPOL and CPHA – where if CPOL = 1 then the idle state of *high* and if CPHA = 1 then the data is sampled on the falling edge of the clock.

That covers the basics of communicating with this line of SBCs through a 4-wire SPI bus – the next section covers the basic structure of the registers within device family.

3 Register Organization

The TCAN28xx and TCAN24xx lines of devices are configurable through the register stack located within device. For a detailed list of registers and the respective bit fields please see Device Registers section in the data sheet. Most control of the device is done through the SPI bus while interacting with the host controller.

Each register is comprised of 1 byte and there is one register per address within the device. There are 5 register access types within this family of device; two read access types and three write access types. Read access is denoted by “R” denoting that this bit field can be read while a code “RH” indicates that the bit field is set or cleared upon reception of hardware read. Write access has three types where basic write access is denoted by “W”, a hardware writes “H”, and a write 1 to clear “W1C” access types. Each bit field described in register tables from data sheet can also stipulate the access code.

The register stack is largely sectioned off into five main groups that are grouped based on what features the groups are connected to. The first group is the ID and revision registers of the device and exists between register address 0h through 8h and contains the device ID (which essentially the part number) and the major and minor revision ID (which needs to be 2.1 for production material). After that the next group can be considered major device registers which last from address 9h through address Eh. These registers control the SPI configuration, power regulation options, and SBC control options – which are options that need to be considered for every single application whereas the rest of the configuration registers are only used if application demands a deviation in default configuration. The third category is debug – there is only one register in this group and that is at address Fh – which has no influence on the rest of the SBC’s operation and only acts as a test register that can be read and written to. This register is mainly used to confirm SPI reads and writes are working as expected without any impact the overall SBCs functionality. The next group of registers is the largest and this can be thought of as optional feature configuration registers which start at address 10h and through 4Fh – this includes all other configuration options that are not in the first “major” device function group including but not limited to transceiver configuration, watch dog timer, selective wake, and HSS control to name a few. Most of the configuration options can be programmed into this part of the register stack. Finally, the last group of registers starts at address 50h and goes through address 62h and this is where the interrupts and interrupt masks are stored within the register stack. Address 50h is the global interrupt vector - which is always sent out of the SDO pin for every SPI transaction as the first byte – this register monitors INT1 – INT4, INT_CANBUS, and INT6-INT7 where each interrupt register gets a bit in the global vector. The other registers contain the eight interrupt registers as well as enable masks for those interrupts in case certain interrupts want to be suppressed by the application.

That is the basic overview of the register stack – and the general categories in which the registers fall under.

4 EEPROM

One of the most important features that is included in the TCAN28xx and TCAN24xx families of devices is the inclusion of an EEPROM which has a dedicated section for designers to use. The EEPROM has two major purposes: contains device trimming configuration that is not accessible to end users and an end user accessible portion that can be used to save select register bits so the device does not have to be reconfigured every time. This section details what information can be saved in the EEPROM and how the EEPROM functions.

The EEPROM can be used to save the following bits for the TCAN28xx line of devices– this is similar to table found in section 8.5.2 of the device's data sheet.

Table 4-1. TCAN28XX - Register Bits That can be Saved to EEPROM

Register Name	Register Address	Bits Saved	Configuration
SPI_CONFIG	9h	0-3	Byte Count, SDI Polarity, SPI Mode
SBC_CONFIG	Ch	0-1,4,6	VEXCC Current Limit, VCC1 Sink Capability, VCC2 Configuration
VREG_CONFIG1	Dh	0-7	VEXCC Configuration, VEXCC Output Selection, VCC1 Sink current control, VCC1 Configuration
SBC_CONFIG1	Eh	0,3-5,7	SW pin Polarity, UVCC1 selection, VCC1 Sleep Mode Action, CAN transceiver Slope Control
WAKE_PIN_CONFIG1	11h	0-4	Min and Max wake pin pulse width configuration
WAKE_PIN_CONFIG2	12h	0-1,5,6	WAKE1 level, WAKE1 Sensing Mode, cyclic wake configuration
WD_CONFIG_1	13h	0-7	Watchdog Mode Select, Watchdog prescaler select, watchdog in sleep mode configuration, watchdog during standby mode configuration, long window time for initial watchdog.
WD_CONFIG_2	14h	0,5-7	Watchdog timer setting, watchdog during standby mode disable
WD_RST_PULSE	16h	4-7	Watchdog Error threshold before device restart.
DEVICE_CONFIG1	1Ah	0,4,7	LIMP during FSM control, LIMP disable control, Cyclic Sensing Wakeup during FSM control.
DEVICE_CONFIG2	1Bh	0	nINT toggle at interrupt generation enable.
SWE_TIMER	1Ch	3-7	SWE Enable Control, SWE Timer Select
nRST_CNTL	29h	5	nRST Pulse Width length when device restarts due to watchdog failure
WAKE_PIN_CONFIG4	2Bh	0-1,3,4-5,7	WAKE3 Threshold, WAKE3 static or cyclic control, WAKE2 Threshold, WAKE2 static or cyclic control.
WD_QA_CONFIG	2Dh	0-7	Watchdog answer generation configuration, watchdog Q&A polynomial configuration, Watchdog Q&A polynomial seed value
HSS_CNTL3	4Fh	0,4	Restart Timer Selection, Cyclic wake in sleep mode enable

While similar action – the bits able to be saved for the TCAN24xx are slightly different.

Table 4-2. TCAN24XX - Register Bits That can be Saved to EEPROM

Register Name	Register Address	Bits Saved	Configuration
SBC_CONFIG	Ch	0-1,4,7	OVCC1 threshold select, VCC1 Sink Capability, VCC2 Configuration
VREG_CONFIG1	Dh	3,5,6-7	FPWM during OVSUP control, VCC1 Sink current control, VCC1 Configuration
SBC_CONFIG1	Eh	0,3-5	SW pin Polarity, UVCC1 selection, VCC1 Sleep Mode Action
WAKE_PIN_CONFIG1	11h	0-3	Min and Max wake pin pulse width configuration
WAKE_PIN_CONFIG2	12h	0-1,5-7	WAKE1 level, WAKE1 Sensing Mode, cyclic wake configuration, wake pulse configuration

Table 4-2. TCAN24XX - Register Bits That can be Saved to EEPROM (continued)

Register Name	Register Address	Bits Saved	Configuration
WD_CONFIG_1	13h	0-7	Watchdog Mode Select, Watchdog prescaler select, watchdog in sleep mode configuration, watchdog during standby mode configuration, long window time for initial watchdog.
WD_CONFIG_2	14h	0,5-7	Watchdog timer setting, watchdog during standby mode disable
WD_RST_PULSE	16h	4-7	Watchdog Error threshold before device restart.
DEVICE_CONFIG2	1Bh	2	UVLO on VSUP that disables BUCK
SWE_TIMER	1Ch	3-7	SWE Enable Control, SWE Timer Select
nRST_CNTL	29h	4,5	nRST Pulse Width length when device restarts due to watchdog failure, GFO polarity select
WAKE_PIN_CONFIG3	2Ah	4-7	WAKE1-WAKE4 enable bits
WAKE_PIN_CONFIG4	2Bh	0-1,3,4-5,7	WAKE3 Threshold, WAKE3 static or cyclic control, WAKE2 Threshold, WAKE2 static or cyclic control.
HSS_CNTL3	4Fh	0,4	Restart Timer Selection, Cyclic wake in sleep mode enable
SMPS_CONFIG1	65h	0-7	Spread spectrum modulation frequency spread, buck regulator switching frequency, PFM/PWM mode configuration in normal mode, PFM/PWM in standby/sleep mode, VCC1 current limit select
WAKE_ID_PIN_CONFIG1	79h	1-3,5-7	ID1 and ID2 enable bits and bias resistor selection
WAKE_ID_PIN_CONFIG2	7Ah	1-3,5-7	ID3 and ID4 enable bits and bias resistor selection
WAKE_PIN_CONFIG5	7Bh	4-5,7	WAKE4 level and cyclic or static control bit.

For the EEPROM configuration to be saved SPI with CRC check must be configured at the very minimum. To save the register bits as stated beforehand the processor must write 0b1 to register address 4E at bit position 7 as well as writing the default code Ah to register address 4Eh in the least significant nibble of register (bit field 3-0) followed by the CRC byte. Register 4Eh's least significant nibble can read back 0x0 and once the configuration bits have been saved register 4Eh's bit position 7 can read back 0b0. In applications where the host processor doesn't support CRC or CRC is not needed for anything else but the EEPROM save there is a work-around. First, the device is configured as this normally can be. Then set the CRC polynomial at bit position 0 of register at address Bh where 0b0 is AutoSar and 0b1 is SAE J11850. Next, the SPI CRC must be enabled by writing 0x1 to bit position 0 in register Ah. Then, the EEPROM configuration can be saved as previously described. Finally disable the SPI CRC if processor does not support this or is unwanted for rest of the application. As a final note, **the EEPROM can only be reprogrammed up to 500 times.**

To understand how the EEPROM functions under different events – there are five main ones to be aware of.

1. UVSUP event; no action is taken w.r.t. EEPROM because register configuration is not lost.
2. Power on Reset (POR) event; EEPROM is read and registers are restored in Init Mode when VSUP > UVSUP_33R
3. Soft Reset; EEPROM is read, registers are restored, and device transitions to Standby mode
4. Hard Reset; EEPROM is read, registers are restored, and device transitions to Init mode

nRST input; EEPROM is read, registers are restored, and device transitions to Restart mode

5 Suggested Data Structures and Program Flow

With an overview of *how* to program the device – this is time to step outside of the part and look into how to simplify the configuration of the device from a firmware perspective. In this section a general overview of what type of firmware is needed to adequately work with this device, suggested type definitions to simplify the overall programming, as well as some general advice for features that can require a bit more complexity than simple getter and setter functions.

The vast majority of configuration functions on this device are a simple act of reading and/or writing registers with some exceptions w.r.t. selective wake and the watchdog timer. That means the bulk of the firmware is largely just going to be getter and setter functions and in the simplest of implementations have a generic type definition for register structure as well as one function for reading and one function for writing (which is going to be built upon any additional SPI driver code needed for the controller). In short there needs to be at least 4 different groups of files when working with this device for best results: a register definition map (.h file only) that defines base register addresses as a constant, a data structure definition file (.h file only) which defines the data types and structures that are used by the firmware, a SPI driver file (.h and .c) which can vary based on controller but is used to interact with the SPI bus of the SBC, and finally a main library file (.h and .c) that contain all getter and setter functions + any special use functions that go beyond reading and writing functions. This library can be further integrated into the final firmware for the system.

From the four suggested file groups listed previously, there are suggestions on how each one can be structured. In the first suggested file – the register map – needs to essentially follow the following guidelines.

```
/*header comment*/
#ifndef TCAN28_REGS_CONSTS
#define FIRST_REG_NAME baseAddress
...
#define LAST_REG_NAME baseAddress

/*any other constants that can be useful to define in application such as
*specific configuration bit patterns used in application*/

/*the following Bit definitions are optional but can be used to generate bit patterns without
havingto go through the bitwise operators*/

#define BIT0 0x01
#define BIT1 0x02
#define BIT2 0x04
#define BIT3 0x08
#define BIT4 0x10
#define BIT5 0x20
#define BIT6 0x40
#define BIT7 0x80
#endif
```

The next file is going to be a little more involved and that is the data structure definition file. At the simplest form only 1 register data structure needs to be defined – and that is a generic register.

```
/*Data Structure header Comment - simple Data Structure*/
typedef struct
{
    union
    {
        uint8_t word;
        struct
        {
            uint8_t Bit0: 1;
            uint8_t Bit1: 1;
            ...
            uint8_t Bit6: 1;
            uint8_t Bit7: 1;
        };
    };
}GenReg;
```

To cover what this code snippet does a look line by line can be taken. The first line states typedef struct – which what that is saying is that a custom struct is being defined – this can be used by creating a variable of “GenReg” type (as listed in last line of example). Taking a look at the second portion of the structure definition comprises of a union of an unsigned 8-bit integer and a structure of more unsigned 8-bit integers; the structure contains data members that resemble specific register fields – in the generic example each of the 8 bits in the byte are represented, but is not directly required. The union has the structure's members and the uint8_t byte to reside in the same memory location to allow for fine control of specific register fields or control over the entire byte. The data members are arranged by LSB – in the example that means the LSB of the byte is represented by the first data member, if the length was greater than 1, call this *N,* then the first data member represents the least significant *N* bits of the byte. This is important that within the union the main byte and the data members within the structure have the same number of bits, in this case 8, to avoid any unforeseen behavior.

Using these types of data structures is also straightforward when passed by reference – the following example gives a basic use case of the previously defined data type.

```
/**Example Usage of Generic Register Data Structure*
*Assume that previously defined typedef structure is defined elsewhere in file*
*For example, both read/write are in one function - these can be split into a read and write
function*/

int genRegInteract(GenReg* gr, uint8_t bitField, uint8_t bitData, uint8_t readWriteBit)
{
    int result;
    if(readWriteBit == 0) //read mode
    {
        switch(bitField)
        {
            case 0:
                result = gr->Bit0;
            case 1:
                result = gr->Bit1;

            ...

            case 6:
                result = gr->Bit6;
            default:
                result = gr->Bit7;
        }
    }
    else // write mode
    {
        switch(bitField)
        {
            case 0:
                gr->Bit0 = bitData;
                result = gr->Bit0;
            case 1:
                gr->Bit1 = bitData;
                result = gr->Bit1;

            ...

            case 6:
                gr->Bit6 = bitData;
                result = gr->Bit6;
            default:
                gr->Bit7 = bitData;
                result = gr->Bit7;
        }
    }
    return result;
}

int main()
{
    //create instance of typedef struct
    GenReg regGen;

    //set all 8 bits to 0b1 using the main uint8_t variable
    //Optional but gives known default value
    regGen.word = 0xFF;

   //read back byte
   uint8_t genRegByte = regGen.word //Byte in genReg = 0xFF;
   //perform read action on bit in position 4 (4th Most Significant Bit)
   int readOneResult = genRegInteract(&regGen, 4, 0, 0);
```

```
    //perform write action on bit in position 7 (Most Significant Bit)
    int writeEchoOne = genRegInteract(&regGen,7,0,1);
    //access main byte
    uint8_t genRegByte = regGen.word// Byte is set to 0x7F now

    //return finish code
    return 0;
}
```

This data structure definition allows quick construction and use of register writes and register reads that can be specified down to a specific bit field. The previous example kept every bit individual – but register specific type definitions can be made from this template with slight modifications and not only can code readability be increased but this can add an extra layer of protection from sending data to the incorrect location – however if this method is pursued there can be a lot of data-structures defined in firmware – so for memory limited applications a single register definition can be the best idea.

The next major file group is the SPI driver code to actually interact with the SBC – this code is extremely dependent on the MCU used in the system as there can be slightly different for every application. There are a few commonalities that can be shared across the SPI driver code – namely they need to be able to perform a SPI read and a SPI write – so at the most simple there can only need to be 2 functions – read and write. Most MCUs have documentation on how to access the 4-Wire SPI modules and that needs to be core of any SPI driver for these SBCs – all the other functionality needed can be handled by the other file groups.

Finally – the core backbone of any firmware stack for the SBC – the main library file. This is going to be comprised of the function definitions and linking to other needed file groups (registers, data structures, and SPI driver) within the .h file. The .C file can contain all the implementation details needed – this is the core file that needs to be used in the system firmware stack to integrate communication and control of the SBC. The good news is that the vast majority of the main library file is comprised of simple getter and setter functions that can read and write to the abstracted register data types and depending on specific use case this can be all the main library functions need to do – however there are a few caveats where simple getters/setters can not be sufficient – this is with respect to partial networking/selective wake and watchdog timer control.

For partial networking the actual programming flow is still essentially just getters/setters, but there are much more than 1 selective wake register so even though each data-structure representation of a register can be used to create multiple data types to be used during partial networking configuration this is usually designed for to create a separate struct just for partial networking configuration if used. The example starter code that is available for both the TCAN28xx and TCAN24xx families uses the following partial network struct for selective wake configuration.

```
//States TCAN28xx, but applicable to TCAN24xx as well
typedef struct TCAN28xx_PN_Config
{
    //ID: the ID to look for selective wake.
    //Right justified whether 11 or 29 bits.
    //If Using Extended ID, make sure to set IDE to 1
    uint32_t ID: 29;

    //IDE: Extended ID used. Set to 1 if using 29 bit IDs
    //Valid values are 0 or 1.
    uint8_t IDE: 1;

    //IDMask: The Value to be used for masking the ID.
    //All 0's means all bits must match exactly.
    //A 1 means that bit is ignored in the ID field for ID Verification
    uint32_t IDMask: 29;

    //DLC: A DLC field value to match
    // Valid Values are 0,1
    uint8_t DLC: 4;

    //Data Mask Enable. Set to 1 if using a data field mask. Turning this
    //on requires the DLC to match, and the data fields are masks.
    //A 1 means match this bit in the data fields. A single match across
    //all the data fields will be a valid WUF

    //Valid values are 0, 1
    uint8_t DataMaskEN: 1;
    //CAN data to match
    uint8_t DATA[8];

    //Selective Wake FD passive. Set to 1 to ignore frames with
    // FD enabled. A 0 (default) results in error counter increment

    //Valid values are 0, 1
    uint8_t SWFDPassive: 1;

    //CAN Data Rate
    uint8_t CAN_DR: 3;

    //CAN FD Data Rate vs. The selected CAN Data Rate
    uint8_t FD_DR: 2;

    //Frame counter threshold. Sets at what value the error counter must
    //reach to set a frame overflow flag (+1). Default is 31

}TCAN28xx_PN_Config;
```

The main difference between the previous example compared to the generic register setup is that instead of focusing on the register structure of all the partial networking registers – this focuses purely on the configurable parameters and the data contained within. This sets up the following configuration parameters:

1. ID – a CAN device can have either an 11bit ID or use an extended ID that is 29 bits; since this can have up to 29 bits the 32-bit unsigned integer type is used to store this. This is important to note that multiple registers can contain the ID string for both classic and extended ID's so is usually worthwhile to save the ID as a stand-alone variable instead of keeping partial IDs in separate variables if the variables were just directly read from respective registers.
2. IDE – a single bit value that indicates if classic ID or extended ID is used.
3. IDMask – an 11- or 29-bit string which is used to determine which bits are "don't cares" for incoming ID requests
4. DLC – this is the data length code that is configured – with possible values of 0 – 15 where values 0 – 8 represents a transaction that is between 0 and 8 bytes respectively for classic CAN or CAN FD and where values 9-15 represent 8 bytes for classic CAN. This is not required for all instances of partial networking – only ones where the data mask enable bit is set.
5. DataMaskEnable – if set to 0b1 DLC matching and Data matching are enabled for selective wake (remote frames are ignored) and if set to 0b0 DLC and Data matching are disabled and assumed to be valid.
6. DATA – an 8 position array (since up to 8 bytes can be transmitted in a single transaction depending on DLC configuration); if the DataMaskEnable bit is set the data for the incoming Wake Up Frame (WUF) then the subsequent data bytes can be checked for at least 1 matching logic high bit – for example, if there is 1 byte to check and the data stored at register DATA0 is 0x0F the incoming data needs at least 1 matching high bit – so valid bit strings can include any byte where the first nibble, four bits, is not 0. There are 8 registers to store data matching bytes if the DataMaskEnable bit is set during operation.
7. SWFDPassive – this is a control bit that when set can ignore frames with FD enabled and if the bit is cleared the error counter can increment.
8. CAN_DR – the data rate of the CAN transceiver stored in a 3-bit integer to match the 3-bit long bit field within SBC to specify CAN data rate.
9. FD_DR – the scaler multiplier of the CAN_DR for CAN FD data rate – stored as a 2-bit number to correspond with SBC configuration of the CAN FD data rate.
10. FrameErrorThreshold – The amount of errors that need to be received for a frame overflow flag to occur; stored as an 8-bit value and defaults to 31 where the frame overflow occurs on the 32**nd** error.

This structure can be used to read and write selective wake configurations on the TCAN28xx and TCAN24xx – but in general can apply to many similar selective wake structures as this partial networking is highly standardized.

For reading the following example is what TI generally gives:

```
/*Reads Partial Networking Configuration*/

/*TCAN_READ and TCAN_WRITE functions are calls to the SPI driver for */
/*SPI reads and writes*/
```

```
TCAN28xx_STATUS_ENUM TCAN28xx_PN_Config_Read(TCAN28xx_PN_Config *pnConfig)
{
    uint8_t data[17] = {0}; //Data stream of the registers
    uint8_t sw_config[2] = {0}; //Registers for SW configuration
    uint8_t index = 0;
    uint32_t id = 0;
    uint8_t address = REG_TCAN28xx_SW_ID1;

    for(index = 0; index < 17; index++)
    {
        TCAN_READ(address, &data[index]);
        address++; //Increment address we are writing to.
    }

    //Read the sw_config registers
    TCAN_READ(REG_TCAN28xx_SW_ID1,&sw_config[0]);
    TCAN_READ(REG_TCAN28xx_SW_ID3, &sw_config[1]);

    //If IDE is enabled
    if(data[2] & REG_BITS_TCAN28xx_SW_ID3_IDE)
    {
        pnConfig->IDE=1;

        //SW_ID1[7:0] which is ID[17:10]
        id = (uint32_t)data[0]<<10;

        //SW_ID2[7:0] which is ID[9:2]
        id |= (uint32_t)data[1]<<2;

        //SW_ID[4:0] which is ID[28:24]
        id |= (uint32_t)(data[2]& 0x1F)<<24;

        //SW_ID3[7:6] which is ID[1:0]
        id |= (uint32_t)(data[2] & 0xC0) >> 6;

        //SW_ID4[7:2] which is ID[23:18]
        id |= ((unit32_t)(data[3] & 0xFC) >> 2) << 18;

        pnConfig->ID = id;

        //SW_ID_MASK1[1:0] which is IDM[17:16]
        id = (uint32_t)(data[4] & 0x03) << 16;

        //SW_ID_MASK2[7:0] which is IDM[15:8]
        id |= (uint32_t)data[5] << 8;

        //SW_ID_MASK3[7:0] which is IDM[7:0]
        id |= (uint32_t)data[6];

        //SW_ID_MASK4[7:0] which is IDM[28:21]
        id |= (uint32_t)data[7]<<21;

        //SW_ID_MASK_DLC[7:5] which is IDM[20:18]
        id |= ((uint32_t)(data[8] & 0xE0)>>5)<<18;

        pnConfig->IDMask = id;

        pnConfig->DLC = ((data[8] >>1) & 0x0F);
    } else {
        //Else we just build normal ID
        pnConfig->IDE = 0;

        //SW_ID3[5:0] which is ID[10:6]
        id = (uint32_t)(data[2] & 0x3F) << 6;

        //SW_ID4[7:2] which is ID[5:0]
        id |= (uint32_t)(data[3] & 0xFC) >> 2;

        pnConfig->ID = id;

        //SW_ID_MASK4[7:0] which is IDM[10:3]
        id = (uint32_t)data[7]<<3;

        //SW_ID_MASK_DLC[7:5] which is IDM[2:0]
        id |= (unit32_t)(data[8] & 0xE0)>>5;

        pnConfig->IDMask = id;
        pnConfig->DLC = ((data[8] >> 1) & 0x0F);
    }
    //Check if data mask is enabled
    if(data[7] & REG_BITS_TCAN28xx_SW_ID_MASK_DLC_DATA_MASK_EN)
    {
        pnConfig->DataMaskEN = 1;
    }
    //Move the data fields over to the struct
    for(index = 0; index < 8; index++)
    {
        pnConfig->DATA[index] = data[index + 9];
    }

    //Now the selective wake configuration registers
    //If SWFD passive bit is set

    if(sw_config[0] & 0x80)
    {
        pnConfig->SWFDPassive = 1;
    } else {
        pnConfig->SWFDPassive = 0;
    }
    pnConfig->CAN_DR = ((sw_config[0]>>4) & 0x07);
    pnConfig->FD_DR = ((sw_config[0]>>2) & 0x03);
    pnConfig->FrameErrorThreshold = sw_config[1];
    return TCAN28xx_SUCCESS;

}
```

The code snippet can seem rather long – but the overall goal of the read function is very simple. The first step is to create the device ID by reading from the ID registers as well as checking if the application is using standard ID or the extended ID. After the ID has been constructed and saved in the partial networking struct the DLC value is set next. After that the data mask enable bit is checked followed by the moving all data registers over to the struct – if the data mask is not enabled the function still loads all the data register values – which if not configured can just be 0. After that the final steps are to collect the SWFDPassive bit, CAN_DR, FD_DR, and the frame error threshold. Finally, after the structure has been fully filled out the function returns an ENUM that indicates success with this part – the ENUM is not necessary to function (a simple int can work fine) - but this can add a layer of separation for SBC specific successes/failures and add a guardrail by creating this enumerated data type.

The next example w.r.t. partial networking is writing to the partial networking configuration registers using the partial networking data structure. The first step is to assign all appropriate values to the struct before passing this into the write function.

```
/*Writes Partial networking Configuration*/
TCAN28xx_STATUS_ENUM TCAN28xx_PN_Config_Write(TCAN28xx_PN_Config *pnConfig)
{
    //Selective Wake Registers used for ID/XID
    uint8_t sw_id[4] = {0};

    //Selective Wake registers used for ID/XID Mask
    uint8_t sw_idm[5] = {0};

    //Data field if used
    uint8_t data[8] = {0};

    //Selective Wake Configuration register
    uint8_t sw_config = 0;

    uint8_t index = 0;

    //If XID is enabled (IDE = 1)
    if(pnConfig->IDE)
    {
        //SW_ID1[7:0] which is ID[17:10]
        sw_id[0] = (uint8_t)(pnConfig->ID >> 10);

        //SW_ID2[7:0] which is ID[9:2]
        sw_id[2] = (uint8_t)(pnConfig->ID >> 24) & 0x1F);

        //SW_ID3[4:0] which is ID[28:24]
        sw_id[2] |= (uint8_t)((pnConfig->ID >> 24) & 0x1F);
        //SW_ID3[7:6] which is ID[1:0]
        sw_id[2] |= (uint8_t)(pnConfig->ID << 6);

        //SW_ID3[5] which sets the IDE flag
        sw_id[2] |= 0x20;

        //SW_ID4[7:2] which is ID[23:18]
        sw_id[3] = (uint8_t)(((pnConfig->ID >> 18)<<2) & 0xFC);

        //SW_ID_MASK1[1:0] which is IDM[17:16]
        sw_idm[0] = (uint8_t)((pnConfig->IDMask >> 16) & 0x03);

        //SW_ID_MASK2[7:0] which is IDM[15:8]
        sw_idm[1] = (uint8_t)(pnConfig->IDMask >> 8);

        //SW_ID_MASK3[7:0] which is IDM[7:0]
        sw_idm[2] = (uint8_t)(pnConfig->IDMask);

        //SW_ID_MASK4[7:0] which is IDM[28:21]
        sw_idm[3] = (uint8_t)(pnConfig->IDMask >> 21);

        //SW_ID_MASK_DLC[7:5] which is IDM[20:18]
        sw_idm[4] = (uint8_t)(((pnConfig->IDMask >> 18) << 5) & 0xE0);
    } else {
        //Else we build normal ID

        //SW_ID3[5:0] which is ID[10:6]
        sw_id[2] = (uint8_t)((pnConfig->ID >> 6) & 0x1F);

        //SW_ID4[7:2] which is ID[5:0]
        sw_id[3] = (uint8_t)((pnConfig->ID << 2) & 0xFC);

        //SW_ID_MASK4[7:0] which is IDM[10:3]
        sw_idm[3] = (uint8_t)(pnConfig->ID >> 3);

        //SW_ID_MASK_DLC[4:1] which is the DLC
        sw_idm[4] = (uint8_t)((pnConfig->ID << 5) & 0xE0);
    }

    //SW_ID_MASK_DLC[4:1] which is the DLC
    sw_idm[4] |= (uint8_t)(pnConfig->DLC) << 1;

    //Check if data mask is enabled
    if(pnConfig->DataMaskEn)
    {
        //SW_ID_MASK_DLC[0] which is the DATA_MASK_EN
        sw_idm[4] |= 1;
    }
    //If Data mask is enabled
    if(pnConfig->DataMaskEN)
    {
        for(index = 0; index < 8; index++)
        {
            data[index] = pnConfig->DATA[index];
        }
    }
    //All our temp. buffers have been created. Now to write to the device
    uint8_t address = reg-tcan28xx_SW_ID1;
    for(index = 0; index < 17; index++)
    {
        uint8_t buffer;

        //if we are under 4, then we are on SW_ID
        if(index < 4)
        {
            buffer = sw_id[index];
        } else if (index < 9)
        {
            //If we are under index 9, then we are on ID_MASK
            buffer = sw_idm[index-4];
        } else {
            //Then we are on the data mask
            buffer = data[index-9];
        }
        TCAN_WRITE(address,&buffer);
        #ifdef TCAN28xx_CONFIGURE_VERIFY_WRITES
            uint8_t readBuffer;
            TCAN_READ(address, &readBuffer);
            if(readBuffer != buffer)
            {
                return TCAN28xx_FAIL;
            }
        #endif
        address++; //Increment the address we are writing to
    }
    //Now to do the selective wake configuration registers
    sw_config = pnConfig->SWFDPassive << 7;
    sw_config |= pnConfig->CAN_DR << 4;
    sw_config |= pnConfig->FD_DR << 2;
    TCAN_WRITE(REG_TCAN28xx_SW_CONFIG_1, &sw_config)

    #ifdef TCAN28xx_CONFIGURE_VERIFY_WRITES
        uint8_t readBuffer;
        TCAN_READ(REG_TCAN28xx_SW_CONFIG_1,&sw_config);
        if(readBuffer != sw_config)
        {
            return TCAN28xx_FAIL;
        }
    #endif

    sw_config = pnConfig->FrameErrorThreshold;
    TCAN_WRITE(REG_TCAN28xx_SW_CONFIG_3,&sw_config);
    #ifdef TCAN28xx_CONFIGURE_VERIFY_WRITES
        uint8_t readBuffer;
        TCAN_READ(REG_TCAN28xx_SW_CONFIG_1,&sw_config);
        if(readBuffer != sw_config)
        {
            return TCAN28xx_FAIL;
        }
    #endif

    //This write must be thge last one to enable the selective wake
    //Otherwise another configuation write will clear the selective wake
    //Configure Bit
    sw_config = REG_BITS_TCAN28xx_SW_CONFIG_4_SWCFG;
    TCAN_WRITE(REG_TCAN28xx_SW_CONFIG_4,&sw_config);
    #ifdef TCAN28xx_CONFIGURE_VERIFY_WRITES
        uint8_t readBuffer;
        TCAN_READ(REG_TCAN28xx_SW_CONFIG_1,&sw_config);
        if(readBuffer != sw_config)
        {
            return TCAN28xx_FAIL;
        }
    #endif
    return TCAN28xx_SUCCESS;
}
```

The write procedure can look a little more intimidating than the read procedure at first glance – but ultimately the procedures are essentially extremely similar processes just in reverse as the read function reads registers and loads data into the struct while the write function takes data in the struct and organizes this and then writes the data to the correct registers in the correct configuration – please see 6.2 for more details on how to configure partial networking.

The other part of the firmware that can be a bit more complex than getters and setters is configuration of the watchdog timer when in question and answer mode. QA watchdog has a similar purpose to other watchdog configurations – this requires the controller to hit watchdog triggers to confirm that communication between controller and SBC remains intact – but with the extra layer of protection that requires the controller to answer a question posed by the SBC to confirm a proper watchdog trigger has been used. This can be wanted in a system that has a high robustness requirement as QA watchdog not only is checking if the communication is there but also making sure a level of accuracy to test more than a simple signaling.

In a Q/A watchdog the controller must read a question from the SBC and write the computed answers back to the SBC. The correct answer is a 4-byte response and each byte must by correct, written in the correct order, and meet the required timing constraints. There are 2 watchdog windows known as WD response window #1 and WD response window #2 where each window is 50% of the total watchdog window time which is selected from the WD_TIMER and WD_PRE register bits. In response window #1 the controller reads the question and responds with response bytes 3-1 where the final response byte, byte 0, happens during WD response window #2. If this is done without error the error counter is decremented if above 0 and a new question is generated and the cycle repeats. If anything is incorrect a new question cannot be generated, the error counter is incremented, and if the error threshold is surpassed the watchdog failure action is performed.

To start the QA watchdog, process the first step is to read the question from the SBC. This can be done simply by creating a void function that uses a pointer to the memory location of a uint8_t value, reading the register and then clearing bits 4-7 and only retaining bits 0-3.

```
/*Read Question*/
void TCAN2XXX_SBC_WDT_QA_READ_QUESTION(uint8_t *question)
{
    //Set up unsigned int for read data
    uint8_t read;

    //Pseudo-Code for SPI read function for SBC
    SPI_READ(REG_TCAN2XXX_WD_QA_QUESTION, &read);

    //Save masked read data to question pointer
    *question = read & 0x0F;
}
```

When this function is called the question pointer can contain the question posed by the SBC. After the question has been read the next part is answering the question – but there is one caveat – there are feedback settings for the answer generation that can change the answer. By default, this value is 0b00 but this can be changed to 0b01, 0b10, or 0b11 in the configuration settings – these values control the configuration multiplexers.

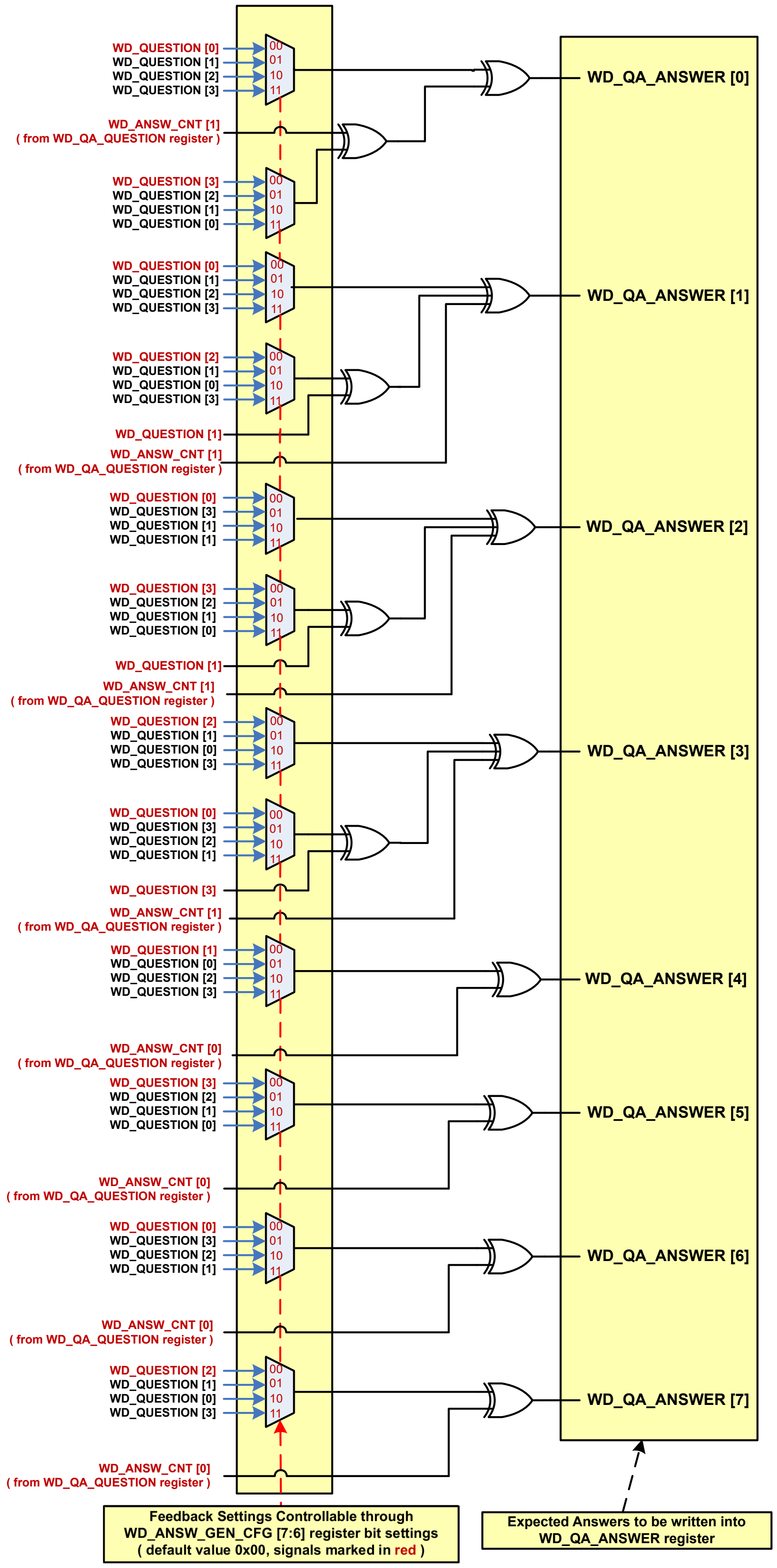


Figure 5-1. WD Expected Answer Generation

For simplicity the following explanation can assume feedback to be default and set to 0b00. There are three general forms used to calculate the answer bits. Bit 0 uses two 2-input XOR gates with 3 inputs; bits 4, 5, 6, and 7 use a single 2 input XOR gate with 2 inputs; bits 1,2, and 3 use 4 input signals and 2 XOR gates. This is advisable to calculate the answer 1 bit at a time starting at the LSB and building the byte and shifting bits to create the full response.

For bit zero there are two inputs to the final XOR gate and therefore can be directly XOR'd to find the answer bit – "var1 ^ var2". The first input of the final XOR gate is just a bit from the question – since configuration 0b00 is used this is the LSB of the question. The second input of the final XOR gate is the output of another XOR gate which has the following inputs bit 1 of the response number, where response numbers start at 3 and decrement down to 0, and the second input is the MSB of the question. To compute the 0 bit in this, answer the bit 1 of response number can equal *a,* the question MSB can equal *b,* the LSB of the question can equal *c,* and the final design be set to *f.*

d = a ^ b

f = d ^ c

As this is shown – the actual first bit calculation is straight forward, bits 1-3 are a bit more involved and require a few more steps to tabulate. For bits 1-2, when using default feedback configuration, the LSB of the question can be a stand-alone input on final XOR gate – this value can still be referred to as *c*. The second XOR gate in bits 1-2 has two inputs – a bit from the question (2**nd** MSB for bit 1, MSB for bit 2), refer to as *a,* is XOR'd with the 2**nd** LSB of the question nibble, referred to as *b*, and this is fed into the final XOR gate. The third input is the MSB of the response count represented by *e*. The final answer of these bits is found by the following steps.

h = a^b

h += c //this is an addition sign not an OR sign

h += e //this is an addition sign not an OR sign

f = h%2 // the modulo '%' operator gives the integer remainder after division

Bit 3 is very similar to bits 1 and 2 in form – but not in values. The value referred to as *c* for bit 3 is not the LSB of the question, but instead the 2**nd** MSB. The values referred to as *a* and *b* is the LSB of the question nibble and MSB of the question nibble respectively. The value known as *c* is no longer the LSB of the question and is instead the 2**nd** MSB of the question nibble. The value *e* is still the MSB of the response count.

h = a^b

h += c //this is an addition sign not an OR sign

h += e //this is an addition sign not an OR sign

f = h%2 // the modulo '%' operator gives the integer remainder after division

Bits 4-7 all have the same form – this is just a single 2 input XOR gate where inputs *a* and *b* are XOR'd to get the necessary bits. The value *a* is going to be one of the bits of the question, for bits 4-7 with a default feedback setting the order is 2**nd** LSB, MSB, LSB, and 2**nd** MSB respectively for bits 4-7. The value *b* is always the same; this is the LSB of the response number.

f = a^b

The starter code that is available for both the TCAN24xx and TCAN28xx line of devices combines this functionality into 1 function that makes use of a couple private functions. The steps are as follows:

1. Initialize a bit-array that has all four feedback configurations programmed. This can be used to determine the correct question bits to be computed depending on feedback setting. There are 12 different instances of the question bits being used so this array needs to be of length 12.

```
/*Generation of Feedback setting bit Array*/
/*Each Position of the bit Array has variable name associated with this*/
/*Saved data is the bit position for specific XOR gate*/
/* for example, XOR gate 0 is going to use wd0*/
```

```
typdef struct
{
    uint8_t wd0 : 2;
    uint8_t wd1 : 2;
    uint8_t wd2 : 2;
    uint8_t wd3 : 2;
    uint8_t wd4 : 2;
    uint8_t wd5 : 2;
    uint8_t wd6 : 2;
    uint8_t wd7 : 2;
    uint8_t wd8 : 2;
    uint8_t wd9 : 2;
    uint8_t wd10 : 2;
    uint8_t wd11 : 2;
} wdt_bits;

void TCAN28xx_WDT_QA_Generate_Question_Bit_Array(uint8_t config, wdt_bits *bitArray)
{
    switch(config)
    {
        default: return;
        case 0x00:
        {
            bitArray->wd0 = 0;
            bitArray->wd1 = 3;
            bitArray->wd2 = 0;
            bitArray->wd3 = 2;
            bitArray->wd4 = 0;
            bitArray->wd5 = 3;
            bitArray->wd6 = 2;
            bitArray->wd7 = 0;
            bitArray->wd8 = 1;
            bitArray->wd9 = 3;
            bitArray->wd10 = 0;
            bitArray->wd11 = 2;
            break;
        }
        case 0x01:
        {
            bitArray->wd0 = 1;
            bitArray->wd1 = 2;
            bitArray->wd2 = 1;
            bitArray->wd3 = 1;
            bitArray->wd4 = 3;
            bitArray->wd5 = 2;
            bitArray->wd6 = 1;
            bitArray->wd7 = 3;
            bitArray->wd8 = 0;
            bitArray->wd9 = 2;
            bitArray->wd10 = 3;
            bitArray->wd11 = 1;
            break;
        }
        case 0x02:
        {
            bitArray->wd0 = 2;
            bitArray->wd1 = 1;
            bitArray->wd2 = 2;
            bitArray->wd3 = 0;
            bitArray->wd4 = 1;
            bitArray->wd5 = 1;
            bitArray->wd6 = 0;
            bitArray->wd7 = 2;
            bitArray->wd8 = 2;
            bitArray->wd9 = 1;
            bitArray->wd10 = 2;
            bitArray->wd11 = 0;
            break;
        }
        case 0x03:
        {
            bitArray->wd0 = 3;
            bitArray->wd1 = 0;
            bitArray->wd2 = 3;
            bitArray->wd3 = 3;
            bitArray->wd4 = 1;
            bitArray->wd5 = 0;
            bitArray->wd6 = 3;
            bitArray->wd7 = 1;
            bitArray->wd8 = 3;
            bitArray->wd9 = 0;
            bitArray->wd10 = 1;
            bitArray->wd11 = 3;
            break;
        }
    }
}
```

2. After the bitArray has been defined the next step is to setup a loop to send 4 different responses
3. Within the loop an answer variable is defined which is linked to an answer array passed by reference into the main function.
4. The answer is calculated LSB to MSB based on previously defined computations and saved to the answer array.

```
/*Calculates WDT Questions Answer*/

void TCAN28xx_WDT_QA_Calculate_Answer(uint8_t *question, uint8_t answer[], uint8_t answerConfig)
{
    //We'll need a way to map the configuration mux to the array of bits
    wdt_bits bitArray = {0};
    TCAN28xx_WDT_QA_Generate_Question_Bit_Array(answerConfig,&bitArray);

    //Now we can generate teh answer
    uint8_t i;

    //Gets us the current WD_ANS_CNT value (starts at 3, and goes to 0)
    for(i = 0; i < 4; i++)
    {
        uint8_t wd_cnt = 3-i;
        uint8_t bitAnswer = 0;
        uint8_t temp;

        //Bit 0
        temp =
TCAN28xx_WDT_Get_Bit(wd_cnt,1);
        temp ^= TCAN28xx_WDT_Get_Bit(*question, bitArray.wd1);
        temp ^= TCAN28xx_WDT_Get_Bit(*question, bitArray.wd0);
        bitAnswer = temp;

        //Bit 1
        temp = TCAN28xx_WDT_Get_Bit(*question,1);
        temp ^= TCAN28xx_WDT_Get_Bit(*question, bitArray.wd3);
        temp += TCAN28xx_WDT_QA_Get_Bit(wd_cnt,1);
        temp += TCAN28xx_WDT_Get_Bit(*question, bitArray.wd2);
        bitAnswer |= (TCAN28xx_WDT_QA_Return_XOR(temp)<<1);

        //Bit 2
        temp = TCAN28xx_WDT_Get_Bit(*question,1);
        temp ^= TCAN28xx_WDT_Get_Bit(*question, bitArray.wd5);
        temp += TCAN28xx_WDT_Get_Bit(wd_cnt,1);
        temp += TCAN28xx_WDT_Get_Bit(*question,bitArray.wd4);
        bitAnswer |= (TCAN28xx_WDT_QA_Return_XOR(temp) << 2);

        //Bit 3
        temp = TCAN28xx_WDT_Get_Bit(*question,3);
        temp ^= TCAN28xx_WDT_Get_Bit(*question,bitArray.wd7);
        temp += TCAN28xx_WDT_Get_Bit(wd_cnt,1);
        temp += TCAN28xx_WDT_Get_Bit(*question,bitArray.wd6);
        bitAnswer |= (TCAN28xx_WDT_QA_Return_XOR(temp) << 3);

        //Bit 4
        temp = TCAN28xx_WDT_Get_Bit(wd_cnt,0);
        temp ^= TCAN28xx_WDT_Get_Bit(*question,bitArray.wd8);
        bitAnswer |= (temp << 4);

        //Bit 5
        temp = TCAN28xx_WDT_Get_Bit(wd_cnt,0);
        temp ^= TCAN28xx_WDT_Get_Bit(*question,bitArray.wd9);
        bitAnswer |= (temp << 5);

        //Bit 6
        temp = TCAN28xx_WDT_Get_Bit(wd_cnt,0);
        temp ^= TCAN28xx_WDT_Get_Bit(*question,bitArray.wd10);
        bitAnswer |= (temp << 6);

        //Bit 7
        temp = TCAN28xx_WDT_Get_Bit(wd_cnt,0);
        temp ^= TCAN28xx_WDT_Get_Bit(*question,bitArray.wd11);
        bitAnswer |= (temp << 7);

        answer[i] = bitAnswer;
    }
}
```

Most of the device is relatively straightforward where the firmware is not complicated with the only action needed is to read and or write registers. The part of the software that is not as simple just requires a bit more work to understand, but ultimately this is not much more complex than the getters and setters and TI does have pre-written firmware that can simplify this process. Next, a few example configurations can be shown to highlight the order in which the configurations need to be programmed, if any, to give a look into what to program.

6 Example Register Configurations

To help highlight how to use this information three different configuration schemes are looked at. In general, the devices can turn on and transition from Init Mode -> Restart Mode -> Standby Mode. For initial software development this is highly suggested that the SW pin be held high, the default active state, as this can prevent action to be taken due to missed watchdog events. The reason this is suggested is that there is a long window watchdog cycle that begins upon the entry into standby mode and if this is missed the device can restart; if the device restarts too much (5**th** restart by default) the device can transition to sleep and/or fail-safe mode depending on configuration which generally occurs within a few seconds from entering standby mode with missed watchdog triggers. Holding the SW pin active makes initial software prototyping easier and the watchdog timer handler needs to be done last. Once in standby mode register configuration can begin and for most configuration settings the order in which this is done in does not really matter, the exceptions being partial networking and watchdog configuration which has more restrictions on how to configure. The three configuration schemes that are examined include: basic SBC/CAN configuration, partial networking, and watchdog timer setup.

6.1 SBC and CAN Transceiver Mode Configuration

In every single application and use case of these SBCs the SBC mode can need to be configured and in the vast majority, if not all, the CAN transceiver can also need to be configured. For both the TCAN24xx and TCAN28xx line of devices this is done in the same way and primarily only concerns a couple of registers which are SBC_CONFIG (address Ch) and CAN_CNTRL_1 (address 10h). This is not an exhaustive list of SBC configuration registers, but ones that are used in many different applications.

The first register, SBC_CONFIG contains six different configuration options – VCC1_OV_SEL (bit 7), OVCC1_ACTION (bit 6, TCAN24xx only), VEXCC_ILIM_DIS (bit 6, TCAN28xx only), PWM_SEL (bit 5), VCC1_SNK_DIS (bit 4), SBC_MODE_SEL (bits 3-2), VCC2_CFG (bits 1-0). For simplicity assume that bits 7-4 and 1-0 are kept as default conditions – a relatively common use case. Bits 3-2 determine what mode the SBC is in with the following options available: Sleep Mode (0b00), Standby Mode (0b01), and Normal Mode (0b10) – most of the devices active operation can take place in normal mode and the controller must put the device into Normal mode by writing to this register. The controller is the one that can move the SBC into normal mode – the device doesn't automatically transition so this step must always be done after configuration and during device wake-up.

The second register, CAN_CNTRL_1, concerns the high-level operation of the CAN transceiver and there are 5 different configuration options with one reserved bit: SW_EN (bit 7), TXD_DTO_DIS (bit 6), FD_EN (bit 5), reserved (bit 4), CAN1_FSM_DIS (bit 3), and finally CAN1_TRX_SEL (bit 2-0). For simple configuration only the CAN1_TRX_SEL bitfield is important – there are six different operating modes as well as two reserved bits. The options are: off (0b000), SBC mode Control WUP disabled (0b010), wake capable (0b100 – default condition), listen (0b101), SBC mode control (0b110), on (0b111), with bit patterns 0b001 and 0b011 reserved. The CAN transceiver state is limited by the overall SBC mode with the choice to control the transceiver independently or SBC or have the SBC mode force the CAN transceiver mode (SBC mode control).

Table 6-1. CAN Transceiver State Description

CAN Bus State	Function
On	CAN Transceiver is on and can transmit and receive messages
Listen	CAN receiver is on, but not able to transmit data
Wake Capable	Low Power Receiver is active and can watch for wake up (either local or CAN bus)
Off	Transceiver is off and cannot be woken through wake-up procedures (Local or CAN bus)
SBC Mode Control	SBC state controls CAN state (most common)
SBC Mode Control no WUP	SBC state controls CAN, CAN bus doesn't wake up from CAN bus wake which starts with WUP

Table 6-2. CAN Transceiver State Vs. SBC Mode

SBC Mode	On	Listen	Wake Capable	Off	SBC Mode Control
Normal	Supported	Supported	Supported	Supported	On
Standby	No	Supported	Supported	Supported	Wake Capable
Sleep	No	No	Supported	Supported	Wake Capable
Restart	No	No	Supported	Supported	Wake Capable
Fail-Safe	No	No	Supported	Supported	Wake Capable

To illustrate an example of what to write to the configuration register a quick look at three cases can be shown: Tie SBC mode to CAN mode, turn CAN on independently, turn CAN to listen mode during standby.

CASE 1: Tie SBC mode to CAN mode

Step 1: Power on Device with SW pin held high (assuming SW_POL is default condition)

Step 2: Once device is in standby mode write 0x06 to address 10h; This sets CAN transceiver to SBC Mode Control

Step 3: Write 0x8A (TCAN24xx) | 0x0A (TCAN28xx) to address Ch; this puts the SBC into normal mode, which also forces the CAN transceiver to be in the “On” state.

CASE 2: Control CAN independently of SBC Mode

Step 1: Power on Device with SW pin held high (assuming SW_POL is default condition)

Step 2: Once device is in standby mode write 0x8A (TCAN24xx) | 0x0A (TCAN28xx) to address Ch; this puts the SBC into normal mode – since CAN is not tied to SBC state CAN transceiver stays wake capable.

Step 3: Write 0x07 to address 10h; this can turn on the CAN transceiver – if SBC mode was not in normal this can be an illegal action and cannot be completed.

CASE 3: Set CAN to listen mode during standby.

Step 1: Power on Device with SW pin held high (assuming SW_POL is default condition)

Step 2: Write 0x05 to address 10h; this can set the CAN transceiver to listen mode. Listen is available in standby so no further actions necessary.

6.2 Partial Networking

As one of the more complicated operations on the TCAN24xx/TCAN28xx line of families this example can go through the procedure or setting up partial networking as well as optional options such as DLC and data validation in addition to standard ID verification. The terms partial networking and selective wake are used interchangeably throughout most documentation – these terms refer to the same thing. These steps must be followed **exactly as stated or else partial networking/selective wake configuration can fail**.

The main registers of concern are SBC_CONFIG (address Ch), CAN_CNTRL_1 (address 10h), SW_IDx (addresses 30h through 33h), SW_ID_MASKx (addresses 34h through 37h), SW_ID_MASK_DLC (address 38h), DATAx (addresses 39h through 40h), and SW_CONFIG_x (addresses 44h through 47h).

The high-level procedure is as follows:

1. Write all control registers for frame detection – this includes ID, ID_MASK, SW_ID_MASK_DLC, DATA, and SW_CONFIG registers; do not write 0b1 to bit 7 of SW_CONFIG_4 until step 3.
2. Read back all registers that were written to verify correct data exists within them
3. Set SWCFG (bit 7 of SW_CONFIG_4) to 0b1 when partial networking setup is complete.
4. Enable Selective wake in CAN_CNTRL_1 register (write a 0b1 into bit field 7 (MSB))
5. Set Device into standby mode regardless of current state (even if device is currently in standby mode).

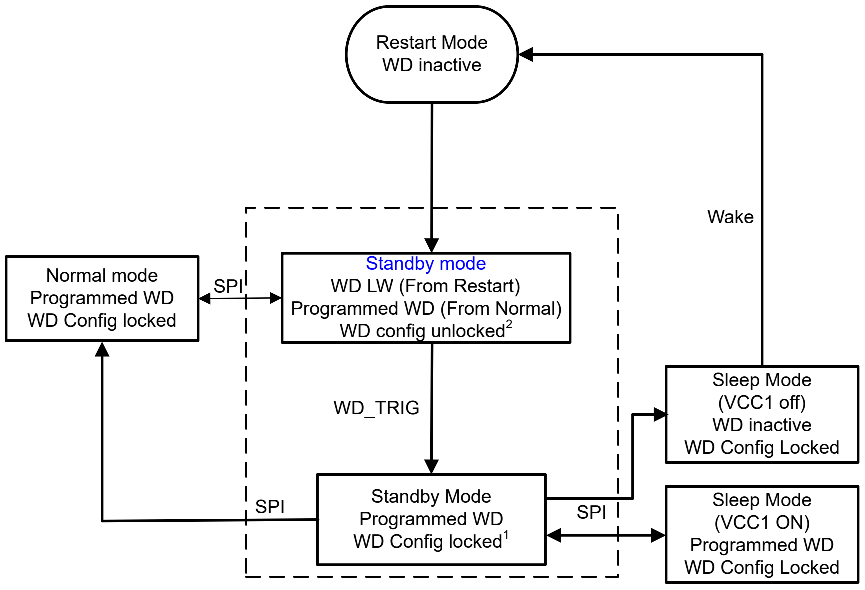
For a straight example the most common setup is examined – which is enabling selective wake to only do ID verification – so DLC and data verification is not needed in this example. The setup assumes 11-bit IDs, that must match the following ID – 0b1100xxxxxxx (x = do not care) – for a selective wake to occur. The CAN Data rate is 125kbps, CAN FD frames are ignored for error counter, frame error counter threshold is set to 31 (frame overflow flag can be set on 32**nd** frame error). Using the guidelines previously mentioned the programming flow is as follows

1. Since we are working with classic 11-bit IDs SW_ID1 and SW_ID2 registers can be skipped over.
2. In SW_ID3 there are a few questions to answer
    a. Bits 7-6 are extended ID bits – so these can be kept at 0b00
    b. Bit 5 is the flag selecting Extended or Classic ID; since standard is used here this needs to be kept at default level of 0b0
    c. Bits 4-0 represent ID bits 10:6 – the 5 most significant bits of the ID; since we only care about the 4 most significant bits in this example either 0b11000 or 0b11001 needs to be written here
    d. So, to configure this register write 0x18 or 0x19 to address 32h
3. Next is SW_ID4 and there is only one main question here:
    a. Bits 7-2 represent ID bits 5:0 – the 6 least significant bits of the ID; since we have listed these as *do not care* the default value of 0b000000 can be kept and no further write is needed.
4. SW_ID_MASK1 through SW_ID_MASK3 only deal with extended ID bits and can be kept default for this example.
5. SW_ID_MASK4 does need to be used as this register controls which bits are *do not care*. The entire byte represents ID mask values for bits 10:3. A bit value of one in bit position *n* represents that the SBC needs to consider this a *do not care* while a bit value of 0 indicates that the SBC needs to care.
    a. Since only the 4 most significant bits are cared about the byte needs to be 0b00001111 or 0x0F
    b. To configure the register, write a value of 0x0F to address 37h.
6. The SW_ID_MASK_DLC register has multiple functions, this contains mask bits (positions 7:5), DLC code (positions 4:1), and the data mask enable. In this example DLC and data verification are not used and can be ignored.
    a. Since bits 7:5 contain mask data for bits 2:0 of a classic 11-bit ID and these values are *do not care* in this example and need to be written to with a bit pattern of 0b111 while the rest of the register is kept default (0b00000) – which results in data 0xE0.
    b. Write 0xE0 to register address 38h.
7. Since data matching is not enabled in this example all data registers (39h through 40h) can be kept with default configuration and skipped over
8. Next is SW_CONFIG_1 which has 3 configuration options: SW_FD_PASSIVE (bit 7) which modifies the error counters behavior upon the reception of a CAN FD frame – by default (0b0) and are counted as errors but setting to 0b1 can ignore CAN FD frames. Next is the CAN bus data rate which is in bit field 6:4 with 6 available options. Next there is the CAN FD data rate in bit field 3:2. Finally the last 2 bits are reserved.
    a. In the example CAN FD frames are ignored for error counter so that needs to be changed to 1.
    b. The CAN data rate is 125kbps which is the value 0b010
    c. The FD_DR bit field can be kept default in this example – so this can stay 0b00.
    d. To finalize this configuration, write 0xA0 to address 44h
9. SW_CONFIG_2 is a read only register this contains the frame error counter value – for configuration this register can be ignored.
10. SW_CONFIG_3 contains the frame error counter threshold; the default value is 31 which means the 32**nd** error can cause the frame overflow flag to be set. The example keeps the default value but this is able to be set from 0 to 255.
11. Read back all modified registers in the previous steps to make sure that the proper configuration has been input. This step isn't required for functionality – but strongly encouraged.
12. SW_CONFIG_4 is the last register to configure as only bit 7 can be modified – bit 7 is the SWCFG bit which by default is 0b0 – when set to 0b1 the SBC has selective wake parameters configured. That means to finalize the configuration 0x80 needs to be written to address 47h. If any register between address 30h to 44h and address 46h are written to after this bit has been set this can automatically clear, this is also the case when a frame overflow error occurs.
13. Now with the partial networking configured, this needs to be enabled via the CAN_CNTRL_1 register. Assuming default condition (0x04) the correct bit pattern to write to address 10h is 0x84.

14. After partial networking has been enabled the device must be set to standby mode – this step must be done even if the device is already in standby mode. Assuming default conditions that means the data 0x86 (TCAN24xx devices) | 0x06 (TCAN28xx devices) needs to be written to address Ch.
15. Partial networking has been configured.

6.3 Watchdog Timer

The watchdog timer is one of the more complex systems within the SBC as when SW pin is not held in an active state the watchdog registers lock and there is only a brief period of time to configure the four main watchdog registers.



1 As long as SW pin is active in Standby mode, WD is unlocked for programming in Standby mode
2 Allows one write to registers 8'h13, 8'h14, 8'h16 and 8'h2D before WD trigger.

Figure 6-1. WD Locking Mechanism

The watchdog registers are unlocked upon entry into standby mode and remain unlocked until the watchdog trigger for the initial long window watchdog has occurred or the SBC transitions to normal mode. The four registers that are subject to this locking are WD_CONFIG_1 (address 13h), WD_CONFIG_2 (address 14h), WD_RST_PULSE (address 15h), and WD_QA_CONFIG (address 2Dh). There is enough time to write to all four registers before the transmission of the watchdog trigger to address 15h during the long window period upon entering standby.

For the following example the watchdog can be configured as such:

1. Watchdog is set to QA
2. Watchdog prescaler is set to factor 1
3. Watchdog is disabled in sleep mode
4. Watchdog is disabled in standby (this doesn't override the long window watchdog upon entry to standby)
5. The long window duration needs to be set to 1000ms
6. For QA watchdog is suggested to have a window size of at least 64ms.
7. QA watchdog can have default feedback, polynomial configuration, and polynomial seed value.
8. Watchdog error count needs to be set to 0 so that if 1 missed watchdog occur an action can be taken (in this case transition to restart mode)

With the configuration goal described the following configuration flow needs to be implemented.

1. Write configuration to WD_CONFIG_1 (address 13h)
   a. Bit Field 7:6 gives watchdog options (disabled, timeout, window, or QA) – QA is represented by 0b11.

b. Bit Field 5:4 gives 4 prescaler options (factor 1 through factor 4) – factor 1 is represented by 0b00
c. Bit Field 3 when set enables watchdog in sleep mode; this can be kept default as the example has watchdog disabled in sleep mode.
d. Bit field 2 sets the standby watchdog type, after the initial long window, which is either timeout or matches the type chosen in bit field 7:6. An important note however is that standby watchdog is disabled in another location – so this can be kept default.
e. Bit field 1:0 represents the length of the long window upon entry into standby mode with the following options: 150ms, 300ms, 600ms, 1000ms. 1000ms is represented by 0b11
f. Putting all the bit fields together means that the byte 0xC3 need to be written to address 13h.

2. Write configuration to WD_CONFIG_2 (address 14h)
   a. Bit field 7:5 represents the window (for window and QA) or timeout delay. The window uses this value and the prescaler to determine window size, please see data sheet for full table. Since the minimum recommended for QA is 64ms and there is a factor 1 prescaler that means the smallest configuration option is 128ms which is represented by 0b010 in this field. If 64ms must be hit then the prescaler needs to be set to factor 2 and this field needs to be set to 0b001.
   b. Bit field 4-1 is a read only register that gives the watchdog error counter value.
   c. Bit field 0 when set can disable watchdog in standby (but not the long window up entry into standby) so this needs to be set to 0b1 in this example.
   d. Putting all the bit fields together means that the byte 0x41 needs to be sent to register address 14h
3. Write configuration to WD_RST_PULSE (address 16h)
   a. Bit field 7:4 represents the watchdog error threshold that sets the limit before SBC responds to watchdog error. This can be kept at the default 0b0000 so that every watchdog failure result in SBC action – typically means restarting device.
   b. Bit field 3:0 represents the restart counter; this is read or write 1 to clear and shows how many times the device has reset.
   c. This register can be kept at the default 0x00 for this example and no further configuration is needed.
4. Write configuration to WD_QA_CONFIG (address 2Dh)
   a. Bit field 7:6 represents the feedback settings of the WD QA expected answer block. This is kept to default 0b00 in this example.
   b. Bit field 5:4 represents the QA polynomial configuration and can remain as the default value of 0b00 in this example.
   c. Bit field 3:0 represents the QA polynomial seed and can remain as the default value of 0b1010in this example.
   d. This address can stay default value (0x0A) and no further action is needed.
5. The watchdog configuration is now complete.

7 Summary

After reading this guide, a few concepts can be better understood in the following:

1. How to communicate with this device through 4-Wire SPI
2. An understanding of how the registers are defined and interacted with.
3. How to use the EEPROM located in these devices for saving custom configurations
4. A better understanding on how to model the registers within the firmware
5. Understanding how to complete more complex programming tasks such as partial networking and watchdog timer

While this guide does not offer an exhaustive list of every single potential configuration option, this needs to be used as a guide to get started quickly and needs to be used in addition to the data sheet for firmware development for both of these families of devices.

8 References

- Texas Instruments, *TCAN241x-Q1 Automotive CAN FD System Basis Chip (SBC) with Integrated Buck Regulator and Watchdog*, data sheet.
- Texas Instruments, *TCAN245x-Q1 Automotive Signal Improvement Capable CAN FD System Basis Chip (SBC) with Integrated Buck Regulator and Watchdog,*, data sheet.
- Texas Instruments, *TCAN284x-Q1 Automotive CAN FD and LIN System Basis Chip (SBC) with Wake Inputs and High-side Switches*, data sheet.
- Texas Instruments, *TCAN285x-Q1 Automotive CAN FD SIC and LIN System Basis Chip (SBC) with Wake Inputs and High-side Switches*, data sheet.