Title: Using SDI-12 with ST Microelectronics MCU's

Author(s): Saari, Alexandra
Hinzey, Shawn Adrian
Frigo, Janette Rose
Proicou, Michael Chris
Borges, Louis

Intended for: Report

Issued: 2015-09-03

Using SDI-12 with ST Microelectronics MCU’s

Alexandra Saari
Shawn Hinzey
Jan Frigo
Mike Proicou
Louis Borges

Abstract

ST Microelectronics microcontrollers and processors are readily available, capable and economical processors. Unfortunately they lack a broad user base like similar offerings from Texas Instruments, Atmel, or Microchip. All of these devices could be useful in economical devices for remote sensing applications used with environmental sensing. With the increased need for environmental studies, and limited budgets, flexibility in hardware is very important. To that end, and in an effort to increase open support of ST devices, I am sharing my teams experience in interfacing a common environmental sensor communication protocol (SDI-12) with ST devices.

Using SDI-12 with ST Microelectronics MCU's

Introduction

My name is Alexandra Saari, and I am a physics student at University of California, Santa Barbara. My background centers primarily in the realms of hardware fabrication, software/hardware interfaces, electronic analysis/design, and experimentation. Over the last two years, I have been working with a team at Los Alamos National Laboratory on a remote sensing system for environmental monitoring. One of the features we wanted to implement was the ability to communicate with SDI-12 based sensors, a common type in environmental monitoring. This feature had several complications to overcome, partially due to the age of the SDI-12 protocol, and partially due to our choice of microprocessor, an ST Micro STM32F407. While other manufacturers have grown large, open, public support networks, ST Microelectronics devices have not enjoyed the same kind of support, being utilized more in commercial products than in small scale or hobbyist level development. As such, when developing for the ST devices, finding online help is difficult to come by, and often specialized beyond usefulness. In an effort to increase the open support for these useful devices, I would like to share and explain a method we found that successfully allowed our ST microprocessors to communicate with a SDI-12 device.

SDI-12 is a serial communication protocol developed in the late 1980's for use with "smart" sensors used in environmental monitoring. SDI-12 sensors use a 3-wire interface that consists of a 12 volt power line, a ground, and a data line. It uses a relatively low data speed of 1200 baud and inverted 5 volt logic levels to communicate ASCII character commands and responses[1]. SDI-12 is a difficult protocol to implement on modern hardware, due to the inverted logic and the 5 volt signaling (many newer microprocessor use 3.3 volt logic). There are several "off the shelf" components that adapt SDI-12 sensors to more standardized serial types, like RS232 or USB. However, these devices do not lend themselves to integration into an embedded system. There are modern microcontrollers and microprocessors that still use 5 volt logic, such as some Atmel products and the Arduino development boards. This could, and indeed does, simplify SDI-12 communication somewhat, but there are other tradeoffs that must be considered.

Our project was based upon the STM32 chip because of its flexibility, performance and extremely low power usage. The same architecture that made this processor so useful for us, also made it especially difficult to work with in terms of SDI-12 compatibility. It is a 3.3 volt chip, so some sort of level translating hardware is required at a minimum. Additionally, on the software development side, while many other companies provide "software serial" libraries that allow the developer to manipulate serial communications outside of hardware based UART's (Universal Asynchronous Receiver/Transmitter), ST does not provide these. These libraries typically allow the developer to create non-standard serial interfaces that uses unusual data rates, inverted logic, etc. This means that in order to produce an inverted logic signal, the developer must write their own "bit-bang" style driver that produces inverted logic, or find a method of inverting the signal through hardware.

Bit banging relies on the microprocessor to maintain timing and produce logic highs and lows over a GPIO (General Purpose Input/Output). The processor literally sends every 1 or 0 in some binary data to a communication pin by setting that pin to a high (3.3 volt) or low (0 volt) at some speed and timing that corresponds to some data rate (baud). To maintain successful

[1] (SDI-12 Support Group (Technical Committee), 2013)

communication in this manner, the processor must be tightly constrained to prevent it from responding to any resource calls from the system that may cause it to delay signal timing or miss an incoming bit. Either of these situations would result in corrupted or lost data. Our chip is being used in an embedded system application running an RTOS (Real Time Operating System) so the possibility of the processor being interrupted during an attempted bit bang interface is high. The better solution for our purposes is to use one of the available UART's included onboard the STM32F407. These take the transmitting and receiving requirements of serial communication off of the processor and ensure data integrity.

A UART usually uses two pins to transmit and receive data. Each pin is specifically tasked to be a transmitting pin that will only send data signals out, or a receiving pin that can only respond to incoming data. SDI-12, as previously mentioned, uses a single data wire, in which the signals can travel to or from the sensor (bi-directional). STM32F407 UART's can operate in a half-duplex mode[2] that allows one pin to be used for both transmitting and receiving. This does not solve the problem of inverting the signal, which will still require a hardware solution.

To summarize the needs up to this point, in order to implement SDI-12 communication with our STM32F407, we can use a UART in half-duplex mode going to a hardware solution that both inverts the signal and translates the 3.3 volt logic to 5 volt. Alternatively, we can use the UART in standard mode and use a hardware solution that inverts the signal, shifts the output to the required 5 volt levels, and converts the single, bidirectional, data line to and from the two UART TX/RX lines. Of course, we will also need sources for the 5 volt and 12 volt requirements for SDI-12.

[2] (STMicroelectronics, 2014, pp. 981-982)

Hardware

At first glance, the first option would appear to be the easiest. In our efforts, we found this not to be the case. For bidirectional level shifting, there was an existing solution that had been widely used for many other purposes. This circuit consists of a power MOSFET, two 10K resistors, and the appropriate power supplies. Please see the following diagram.

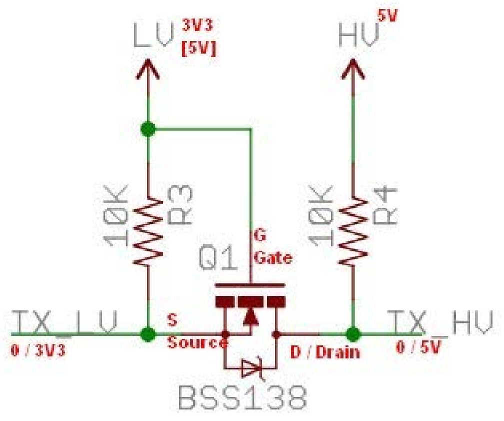

Figure 1 MOSFET based level shifter

This circuit has been tested and works well delivering the appropriate level voltages to both the microprocessor and sensor sides of the circuit. Creating a bidirectional inverter proved to be more difficult. No such component could be found through traditional commercial providers, and no references to such a circuit could be found in electronics forums. We attempted to design one, ourselves, but were unable to find a solution that worked using only one wire. As an aside, we also tried an integrated circuit from Maxim that claimed to be for 3.3-5 volt level shifting, the MAX3371. This chip proved to be unsatisfactory as it could not provide a consistent 5 volt logic level with a SDI-12 sensor's expected input impedance, and had trouble with logic low thresholds. We contacted Maxim for further advice and inquired about a simulation model for their component (for Multisim or other circuit simulation software), but they were not helpful.

We moved on to the alternative option, using the UART in standard mode and doing all the translation in hardware. This proved to be fairly easy, as it is the proposed method given by the SDI-12 specification[3]. Additionally, the engineering services company, Daycounter provides a circuit on their website that effectively inverts the signals, and level shifts the voltages in both directions[4]. The circuit is displayed below.

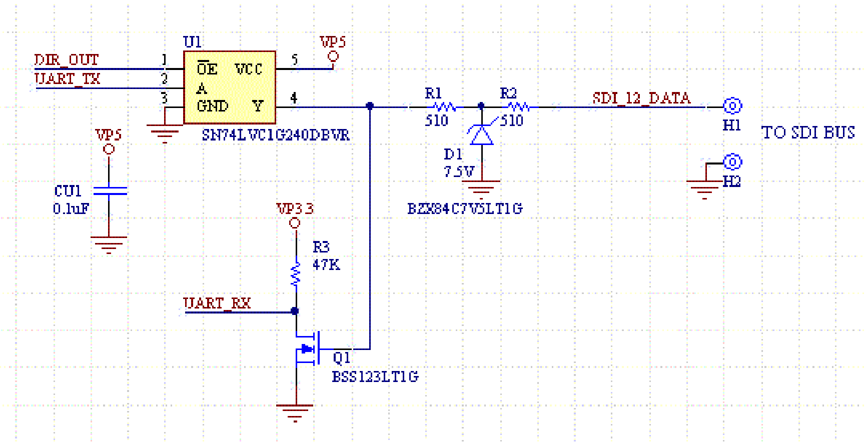

Figure 2 Daycounter SDI-12 circuit

As the STM32F407 is tolerant to 5 volt signals on the receive pin, we came up with another, very simple circuit using a multichannel inverter and a single MOSFET.

[3] (SDI-12 Support Group (Technical Committee), 2013, pp. A-1)
[4] (Daycounter Inc., 2015)

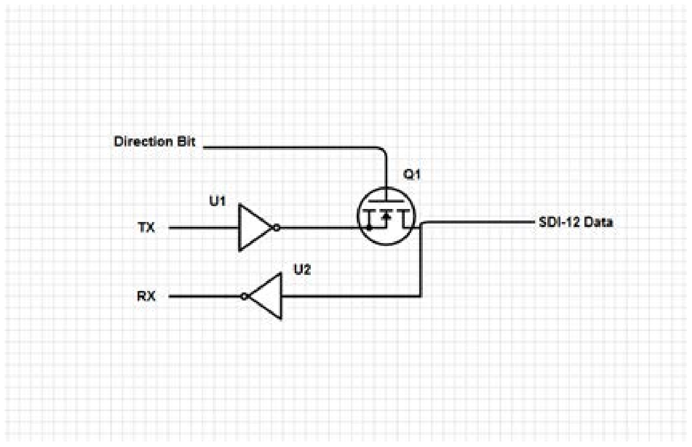

Figure 3 Alternative SDI-12 circuit

Both the Daycounter circuit and ours were tested by connecting them to a Cera Diver and DCX SDI-12 converter from Schlumberger Water Services. Both performed well and delivered clean, inverted digital signals back and forth. Each circuit requires three pins from the processor; a UART TX, RX and a GPIO for direction control. The GPIO acts like a railroad switch, directing outgoing and incoming transmissions to their respective destinations.

In the Daycounter circuit the GPIO enables the 3-state buffer (designated by component U1, Fig. 2) allowing the UART TX signal to be inverted and passed along to the SDI-12 data line. As soon as the TX line is finished, the GPIO signals the buffer to shutdown, putting the output into a high impedance state and allowing the incoming signals from the SDI-12 sensor to go to the MOSFET gate (component Q1, Fig. 2). This pulls the UART RX pin low by grounding the pin for each positive pulse from the signal, thus inverting the incoming signal back to the microprocessor.

In our circuit, the MOSFET (Component Q1, Fig. 3) is controlled by the GPIO. When the UART is in a transmit state, it sends the signal from the UART TX pin through one channel of an inverter (component U1, Fig. 3) with 5 volts Vcc. This inverts the signal and converts it to 5 volt levels. The GPIO triggers the gate of the MOSFET to allow the inverted transmission to pass through to the SDI-12 data line. Once the transmission is complete, the GPIO deactivates the MOSFET's gate, allowing incoming signals from the sensor into a second channel on the inverter (Component U2, Fig. 3) where it is inverted and sent to the UART RX pin. For our circuit, we used a hex Schmitt trigger inverter, Texas Instruments P/N SN74AHC14N. For the MOSFET we chose a 5LN01SP-AC from ON Semiconductor.

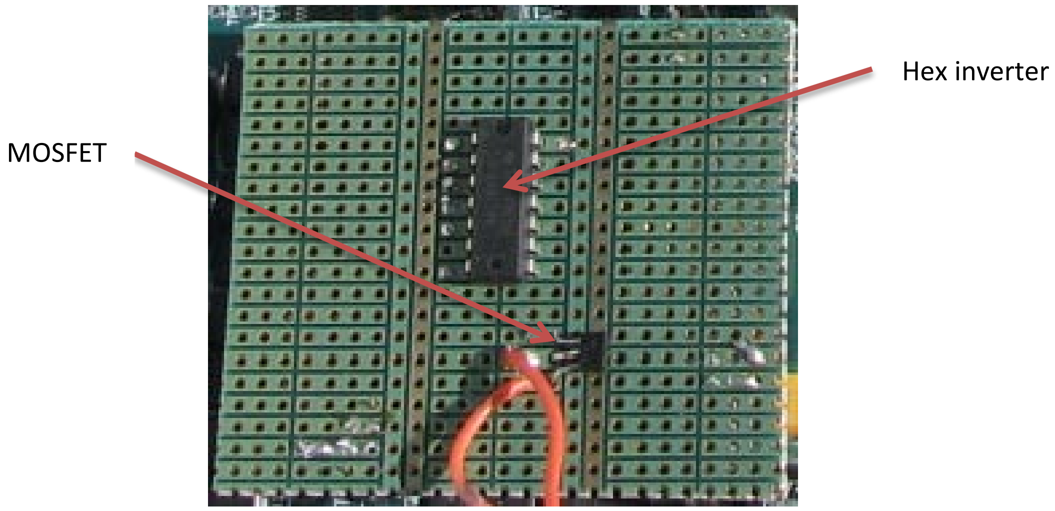

Figure 4 The assembled test circuit

These components, for testing purposes, were chosen primarily for their ease of use rather than space or power efficiency.

With the hardware solution working, we were able to move on to the software side of the problem. All we need in this case is to set up a standard two wire UART using ST's provided HAL (Hardware Abstraction Layer) drivers. Developers can choose to use the older Standard Peripheral Library drivers if they choose, but I will not cover their use here. The following is written with the understanding that the reader has a basic understanding of the C programming language and the use of functions.

Software

The first thing we had to do was choose a UART, for our purposes, UART 6 was available. Setting up the UART is fairly straightforward for SDI-12. The data is transmitted at 1200 baud, using a 7 bit word length with even parity. With a start and stop bit, that creates a 10 bit data string. Setting up the ST UART's with a standard 8 bit word, with even parity settings produced correct results, despite not technically being the correct configuration. So our UART initialization function looked like this:

```
UART_HandleTypeDef huart6;

void MX_USART6_UART_Init(void)
{
  huart6.Instance = USART6;
  huart6.Init.BaudRate = 1200;
  huart6.Init.WordLength = UART_WORDLENGTH_8B;
  huart6.Init.StopBits = UART_STOPBITS_1;
  huart6.Init.Parity = UART_PARITY_EVEN;
  huart6.Init.Mode = UART_MODE_TX_RX;
  huart6.Init.HwFlowCtl = UART_HWCONTROL_NONE;
  huart6.Init.OverSampling = UART_OVERSAMPLING_16;
  HAL_UART_Init(&huart6);
}
```

UART 6 on the STM32F407ZG (Our 407 variant) had its TX pin on PC6 and its RX on PC7[5]. Those pins translate to GPIO bank "C", pins 6 and 7. So our UART GPIO initialization function was as so:

```
void HAL_UART_MspInit(UART_HandleTypeDef* huart)
{

  GPIO_InitTypeDef GPIO_InitStruct;
  if(huart->Instance==USART6)
  {
    /* Peripheral clock enable */
    __USART6_CLK_ENABLE();

    /**USART6 GPIO Configuration
    PC6     ------> USART6_TX
```

[5] (ST Microelectronics, 2013, p. 52)

```
    */
    GPIO_InitStruct.Pin = GPIO_PIN_6|GPIO_PIN_7;
    GPIO_InitStruct.Mode = GPIO_MODE_AF_PP;
    GPIO_InitStruct.Pull = GPIO_PULLUP;
    GPIO_InitStruct.Speed = GPIO_SPEED_LOW;
    GPIO_InitStruct.Alternate = GPIO_AF8_USART6;
    HAL_GPIO_Init(GPIOC, &GPIO_InitStruct);
  }
}
```

Finally, we needed to initialize a GPIO pin to use as our direction controller. This can be any standard GPIO pin capable of producing the voltage and current required for triggering the traffic controlling device in the developers chosen hardware solution. For our purposes, we just used UART 6’s RTS pin (Request to send), which was otherwise unused. This was pin PG8 on our processor (GPIO “G”, pin 8)[6]. This means that at some point before running any SDI-12 code, we needed to initialize that GPIO as a pin that would allow us to turn it on (the pin delivers 3.3 volts to the circuit) or off (the pin delivers 0 volts). This corresponds to an “output” pin configured as required for the developer’s circuit. For our purposes, we configured the pin as a push/pull type with no pull up or down resistors enabled. Our GPIO configuration function had these lines buried amongst all the other GPIO pins we were using:

```
void MX_GPIO_Init(void)
{

    GPIO_InitTypeDef GPIO_InitStruct;

    /* GPIO Ports Clock Enable */
    __GPIOG_CLK_ENABLE();

    /*Configure GPIO PG8: USART6 RTS pin*/
    GPIO_InitStruct.Pin = GPIO_PIN_8;
    GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
    GPIO_InitStruct.Pull = GPIO_NOPULL;
    GPIO_InitStruct.Speed = GPIO_SPEED_HIGH;
    HAL_GPIO_Init(GPIOG, &GPIO_InitStruct);

    /*Stuff for other GPIO functions*//
}
```

[6] (ST Microelectronics, 2013, p. 52)

All of the above code just sets up the processor to use UART 6 and one GPIO pin. The main thing to understand here is that these functions must be run prior to attempting to use the UART or GPIO. This does not need to be written manually and can be auto-generated using a tool called Cube MX, provided freely by ST. The next part, however, will deal with initiating SDI-12 communication, so cannot be auto-generated.

SDI-12 communication is initiated by the processor. The processor asks an SDI-12 sensor something by sending it an SDI-12 command, then the sensor responds with an SDI-12 response. SDI-12 commands are generally formatted like so: <address><command>!
SDI-12 addresses can be upper or lowercase letters from a-z, or the numbers 0-9. So, for example, to ask a sensor with address "a" to start a measurement, you would send the command "aM!". If everything is working correctly, the sensor would send back a response like "a0015" followed by a carriage return and a line feed[7]. The "a" in the response represents the address of the sensor responding, and the "5" represents the number of measurements the sensor will have available within one second. For more information about SDI-12 commands and responses, please refer to the current specification available at sdi-12.org.

To initiate SDI-12 communication, first we prepare our hardware to transmit by setting the direction pin to high, then send a break using the command, "HAL_LIN_SendBreak(&huart6)", then waiting for 12 to 20 milliseconds before transmitting the SDI-12 command characters using "HAL_UART_Transmit(&huart6, data, len, timeout)", where data is the string of characters you wish to send, len is the length of that string, and timeout is the amount of time in milliseconds to wait before giving up. Then set the direction pin to low and set the receive pin to listen for the incoming data from the sensor using

[7] (SDI-12 Support Group (Technical Committee), 2013, pp. 11-12)

"HAL_UART_Receive(&huart6, data_buffer, BUFFER_SIZE, timeout)", where data_buffer is the array that will store the incoming data, BUFFER_SIZE is the maximum length the string of data characters might be, and timeout is the maximum time the UART should wait to receive the data. Putting all this together to make a simple SDI-12 command function looks like this:

```
int SDI12_Command(uint8_t *data, int timeout)
{
   len = strlen((char *)data);
   HAL_GPIO_WritePin(GPIOG, GPIO_PIN_8, GPIO_PIN_SET); //set outgoing
   //Send break
   if(HAL_LIN_SendBreak(&huart6) != HAL_OK) return HAL_ERROR;
   //set marking
   HAL_Delay(20);
   //send message
   if(HAL_UART_Transmit(&huart6, data, len, timeout) != HAL_OK) return HAL_ERROR;
   HAL_GPIO_TogglePin(GPIOG, GPIO_PIN_8); //set incoming
   //Begin Receive, wait for message back
   __HAL_UART_FLUSH_DRREGISTER(&huart6); //flush receive buffer
   if(HAL_UART_Receive(&huart6, data_buffer, BUFFER_SIZE, timeout) != HAL_TIMEOUT) return
HAL_ERROR;
   //end command function
   return HAL_OK;
}
```

This function will send a command to a SDI-12 sensor and store the sensor's response in the array "data_buffer". All that remains is to process "data_buffer" for whatever pertinent information is relevant to the application. That code will, of course, be entirely dependent on the final application's and developer's needs.

Conclusion

The above relates our experiences with interfacing SDI-12 and ST hardware. I've attempted to share some of the issues we had while trying to accomplish this, and share the basic requirements for a successful interface, in the hopes of saving future developers some time and energy. The system we developed works well, and the Daycounter circuitry was tested and also works well. Using the information provided here should give everything needed to set up the basics of a functioning SDI-12 interface. With the looming environmental issues facing the world, I hope this little bit of information will prove useful to the people engaged in environmental research, and save them a little time.

Works Cited

Daycounter Inc. (2015). *SDI-12 Bus Interface*. Retrieved August 19, 2015, from Daycounter Inc Engineering Services: http://www.daycounter.com/

SDI-12 Support Group (Technical Committee). (2013, January 26). *SDI-12 A Serial-Digital Interface Standard.* Retrieved August 19, 2015, from SDI-12 Support Group: http://www.sdi-12.org/

ST Microelectronics. (2013, June). *STM32F405XX/STM32F407XX Datasheet.* Retrieved from www.st.com

STMicroelectronics. (2014, May). RM0090 Reference Manual. *STM32F405xx/07xx, STM32F415xx/17xx, STM32F42xxx and STM32F43xxx advanced ARM®-based 32-bit MCUs*.