

Simple Processor Communication in Embedded Systems

A White Paper

Michael Parker

Software Engineer, Nucleus RTOS Group

Intended Audience: Embedded software engineers and managers looking for an overview of simple processor communication in embedded systems.

Summary of Contents: Explore the use of a simple processor communication method in an embedded system. The embedded system evaluated deploys multiple microprocessors where one or more microprocessors are viewed as slave devices. Specific implementation involves a master/slave multi-processor embedded system. The master device consists of an application residing in an embedded operating system environment. The slave devices consist of a microprocessor with a single processing thread. The goal is to provide a robust, yet simple method of communications between the two microprocessor devices. The communication method should be extendable to include a single master communicating with multiple slave devices. The discussion of this paper will include some problems that may be encountered and potential solutions.

Introduction

The market place is bursting at the seams with miniature microprocessors. The technology boom in all areas of our fast-paced society has resulted in a multitude of embedded system environments utilizing a myriad of microprocessors. Many of these microprocessors contain a small amount of on-chip flash and RAM, and thus are not suitable for most stand-alone embedded systems. However, when remotely combined with an embedded system application utilizing a more

substantial microprocessor, sufficient resources and a multi-threaded operating system, we have the makings of a versatile embedded control system. These control systems are able to tackle such tasks as home automation control systems, navigation control systems (more commonly known as autopilots), automated teller machines, robotics and many others. The application for such control systems is limitless. The nature of control systems dictates a variety of remote tasks - sensor monitoring, dampers control, servo management, and pulse width modulation (motor speed control), just to mention a few. To perform these remote tasks, the need presented itself to utilize small inexpensive microprocessor devices, and along with these devices, a method to communicate and control them as well as retrieve information. Due to the simplicity of the devices, the communications method had to be simple both in hardware and software requirements. The interface method of choice in the industry for this environment is the Serial Peripheral Interface (SPI). This interface is simple and very cost effective in both hardware and software resource requirements.

Description

The SPI employs four signals: data clock (SCLK); master data output, slave data input (MOSI); master data input, slave data output (MISO); and slave select (SS). These signals can be used with a single master/single slave configuration as seen in figure 1.

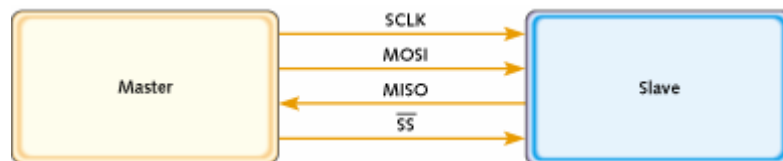


Figure 1

They can also be used in a single master/multiple slave configuration as illustrated in figure 2.

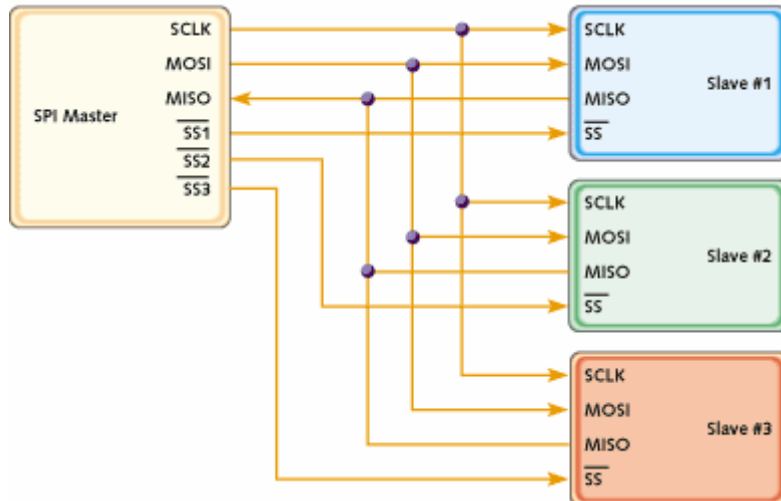


Figure 2

In a single master slave configuration, the slave select signal can be omitted for additional simplification. In this mode the one slave is always enabled. SCLK is generated by the master and is an input to all slave devices. SCLK can be generated by hardware using a costlier, more sophisticated implementation. Most commonly, a clock pulse is generated via software control and thus has a variable baud rate. MOSI carries data from the master to the slave. MISO carries data from the slave back to the master. This is performed in a full duplex mode, which means the data flows in both directions simultaneously. The signals are easily implemented using GPIO signals already available on most microprocessors.

In a multiple slave environment, a specific slave device can be selected by asserting its SS signal. Once a slave is selected, then the master can send data over the MOSI signal and receive data over the MISO signal. As serial data is clocked by the master along the MOSI data signal line, the same SCLK signal also clocks the data from the slave's data register to the master over the MISO data line. The entire communications operation is controlled by the master as the name implies. The data register is basically a shift register, and as a bit is shifted into the register from the master, one is shifted out from the slave. The internal SPI contains a bit counter and, when it is full (usually 8 data bits), the slave's microprocessor is signaled to read the register.

Because the master microprocessor is controlling the bit transmission, it also knows when the returning data is available to be read. The data unit sent from the slave to the master can be assembled in RAM or memory mapped register by testing the state of the MISO data line, and inserting the state in the form of a bit in the RAM location or register. There is no requirement of a special purpose register on the master as usually found on the slave.

Implementation

Implementing an SPI is best handled by hardware support for the control lines in the form of a discrete register, FPGA or GPIO, a data register and a bit counter. The data register is part of the slave microprocessor system and is used as a repository for the data transmitted from the master. The master transfers data to the slave by setting the state of the MOSI signal for a bit and then

changing the state of the SCLK signal. The opposing SCLK transition is normally used to transfer a data bit from the slave to the master and completes the clock pulse. This is repeated until the entire unit size number of bits is transferred. A counter maintains the number of bits transmitted to the slave device and will signal the device, usually via an interrupt, when the last bit is strobed into the data register. The data register supports both read and write capabilities, allowing the slave to read the data placed in the register and then response data can be placed in the register. This data will be sent to the master on the next sequence of clocks. When deciding to use an SPI for device communications, it should be noted that it is not a good interface for transferring large amounts of data. It is also not an appropriate interface to use when flow control is required.

Problem

SPIs are efficient for simple, inter-processor communication in a master/slave environment. The simplicity is a definite advantage. But data integrity problems can occur and are difficult to detect or correct in such a simple interface. The most likely cause of a data integrity problem is electrostatic discharge. A discharge entering the device can cause an extra strobe or multiple strobes on the SCLK signal of the interface. When this occurs, an extra bit or multiple bits are shifted into the data register corrupting its contents. If the data register is not full and an erroneous bit is strobed in, the bit counter is incremented based on the number of erroneous clock strobes. Thus, the next data unit to be sent to the slave from the master will appear to be complete before the entire data unit is transmitted. Other conditions, such as noise generated by other signals, can also create false strobes. Electronic noise can be created by other devices or surges due to motor startup and other sudden high amperage requirements, possibly resulting in power fluctuations on the controller.

Solution

Ideally, it is best to alleviate the effects of the electrostatic discharge or electronic noise through a clean hardware design and implementation as part of the development process. However, normally these issues are not discovered until the control system is completed and in a test environment - or worse, in the hands of the customer. Attempting to retrofit the hardware to eliminate a data corruption issue may include shielding the communications cable, grounding, adding capacitance and the use of ferrites. These can be considered band-aids only and may be costly. Additionally, if a problem is exposed in the field, hardware modification is usually not an option.

A software solution implementing a simple protocol can improve robustness and, in my experience, totally prevent the misinterpretation of corrupted data. It is important to point out that a software solution is a recovery, not a correction method. Data corruption still occurs, but the intent of the software solution is to detect that corruption and retry until there is a clean data transfer. Most slave devices utilize a small microprocessor such as an Atmel Tiny-26 with only 2K of flash for the system code. Thus, any solution must use a minimum of resources.

Attempting to implement a sophisticated CRC algorithm is out of the question. Implementation of a simpler communications protocol for an SPI is the industry recommended method. Proper implementation requires knowledge of the available resources and a maximum latency for the slave to read and update the data register. This is required so the master can properly delay between each

data unit (normally a byte). The most prevalent protocol used in an SPI implementation is a command/response protocol, which can be used to verify the data integrity of the actual data transmission. This command/response protocol can be implemented using a small amount of code space and limited RAM storage on the slave device (in my experience with the Atmel part less than 100 bytes).

The command consists of a single command byte that is sent from the master to the slave. At this time, the byte returned to the master is considered garbage and ignored. On the slave device the command is read from the register and then written back to the register, or in many applications the register retains the last byte written. Either way, the command byte is in the SPI data register. The command byte is then followed by an ACK from the master. The ACK byte is used to flush the previous command byte back to the master for verification. Once the command is verified, subsequent bytes are sent for the command and each one is verified by an ACK sent by the master to return the byte sent. If, at any time, the byte returned back to the master did not match the one sent by the master, then a SYNC command is sent to reset the command sequence.

The assumption with this mechanism is that the corruption takes place as part of the bit transmission and not in the register itself nor internal to the slave device after it is read. Each command is known by both the master and the slave, so there is no need for a length command. Once a complete transmission of the command sequence is successful, an execute command is sent and verified via the ACK mechanism. This command response mechanism can be used to retrieve status from the slave device as well. One of the defined commands can be a status command and after verification using the ACK method, the slave writes the status byte into the data register and a subsequent ACK is sent to return the status byte back to the master. You can see the importance of the master's delay between bytes. The slave being a single thread, it is usually an easy process to determine a reasonable delay, and if implemented using interrupt processing, it should be almost instantaneous, that is to say in the low microseconds. The command structure can be tailored for your specific needs and can be expanded to accommodate status responses longer than a byte if desired.

Additional Uses

Another issue that can arise with a control system using the master slave configuration and a command status protocol over an SPI is lack of software synchronization on multiple microprocessors when the devices are updated independently. Many microprocessors, such as the Atmel series, provide a mechanism to program the flash either via the SPI or other control lines. This opens the door for using the master as the one that determines the version of software to run on the slave(s), and allows the master to program the slave if the version is not compatible.

Resources

Internal document figures 1 and 2 as well as signal definitions were gathered from the following web site: <http://www.embedded.com/story/OEG20020124S0116>

Contact:

Mentor Graphics

Embedded Systems Division

Phone: 251.208.3400

Fax: 251.343.7074

Toll free: 800.468.6853

Email: embedded_info@mentor.com

Web: <http://www.mentor.com/embedded>

739 North University Blvd.

Mobile, Alabama 36608