Application Report

SNVA157A–April 2006–Revised April 2013

AN-1460 Emulating the PowerWise Interface Using GPIOs and Software

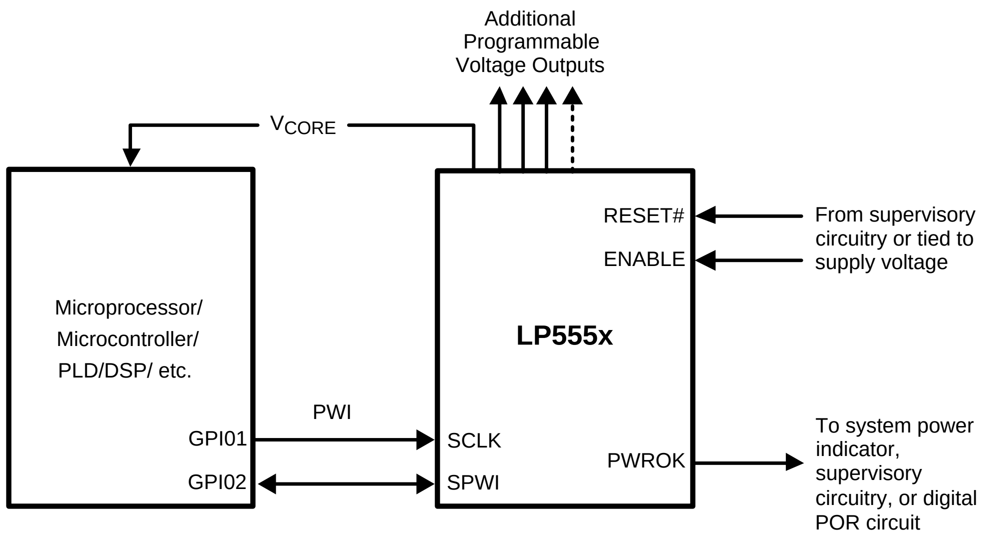
ABSTRACT

PowerWise Interface (PWI) compatible Energy Management Units (EMUs) from Texas Instruments are flexible, highly integrated, and digitally programmable system power managers that can be used in a variety of applications. Controlling or programming the EMU in the system is achieved using the PWI open-standard (www.pwistandard.org). The method for using the PWI1.0 connected EMU in systems with a processor containing a hardware PWI master like the Advanced Power Controller (APC) is obvious (http://www.ti.com/ww/en/analog/power_management/powerwise-avs.shtml). Using the PWI connected EMU in a system where the processor does not contain a hardware-based PWI master is straight forward using a GPIO port and a PWI protocol software driver.

Contents

1 Introduction

This application report describes a strategy for emulating a PWI master using general purpose input/output pins – GPIOs – and a PWI master software driver. This strategy can be deployed with a variety of processors, DSPs, and PLDs. Example C code for such a driver is included later in this document. The code presented has been verified using TI's COP8 microcontroller and the LP5550 PowerWise EMU, but should be easily portable to any processor architecture. This application applies specifically to PWI1.0, but a similar scheme could be used for PWI2.0 functionality.


Additional Programmable Voltage Outputs
VCORE
Microprocessor/ Microcontroller/ PLD/DSP/ etc.
GPI01
GPI02
PWI
LP555x
RESET#
ENABLE
SCLK
SPWI
PWROK
From supervisory circuitry or tied to supply voltage
To system power indicator, supervisory circuitry, or digital POR circuit

2 PWI Discussion

The PWI 1.0 specification provides for a single-master, single-slave point-to-point bus. The point to point nature of the specification greatly simplifies the emulation of the PWI master.

Digital communication over the PWI is managed with a 2-wire serial interface. The PWI data line, SPWI, is bidirectional. From the master viewpoint the SPWI line is driven by the master for all command and data write frames, and driven by the slave (the EMU) for data read frames. The PWI master always drives the bus clock, SCLK. The PWI physical layer is implemented as a push-pull architecture. There are very weak bus pull-down resistors, or bus holders, to maintain low signal levels on the bus during turn around cycles, and at power-on.

3 PWI Emulation

Emulating the PWI master requires that the software driver controls the allocated GPIOs to create a direct-drive, always output SCLK pin, and a bi-directional SPWI pin. The operating frequency range for the PWI bus is 0Hz to 15 MHz, with the clock being present only during a data transaction. The fact that the bus is specified down to DC means that any device, operating at any frequency, can handle the PWI master task. This also allows a great deal of flexibility in implementing the driver because the EMU is not timing dependent beyond the mandatory set-up and hold times and the minimum pulse width of the SCLK.

The emulation driver should include support for the following PWI commands:

- Core Voltage Adjust
- Reset
- Sleep
- Shutdown
- Wakeup
- Register Write
- Register Read
- Synchronize

It is important to pay attention to the I/O voltage levels on the PWI bus. The LP555x EMUs will signal at the level of their I/O voltage regulator (i.e., the regulator associated with PWI register R7). This regulator is programmable and should be set to the same voltage as used by the PWI master if it does not directly drive the I/O ring of the master. If the I/O voltage regulator is used to power the I/O ring of the master, then no special care needs to be taken.

4 Software For PWI Emulation

The first functions that need to be implemented are the initialization, frame write, and frame read functions. It will be helpful to refer to the PWI specification, available at www.pwistandard.org, as you read through the frame write and read functions.

The initialization code is device-dependent, so only pseudo-code is provided here.

```
pwi1_initialization(void)
{
GPIO1_value = LO;  //Make sure the SCLK line comes on driven low
GPIO1_configuration = push-pull output;
GPIO2_value = LO;  //Make sure that the SPWI line comes on driven low
GPIO2_configuration = push-pull output;  //All transactions begin with SPWI as an output
#define SCLK GPIO1
#define SPWI GPIO2
#define SPWI_DIR GPIO2 Configuration
pwi1_synchronize();  //A PWI synchronize command should always be issued at POR
}
The following two functions implement the data link layer of the PWI.
byte pwi1_write_frame(byte data)
{
byte error = 0;  //Return value
byte bit = 0;  //Used as data counter to serialize "data"
SPWI_DIR = OUT;  //Make sure we are driving the SPWI pin
 // START Bit
SPWI = HI;
 pwi1_clock_pulse();  //See the helper function later on in this code
// Send the two reserved bits, 00b
SPWI = LO;
pwi1_clock_pulse();  //Reserved 1
pwi1_clock_pulse();  //Reserved 2
// Now we will do the payload bits
 for(bit = 0; bit < 8; bit++){
SPWI = ((data & 0x80) ? HI : LO);   //Transfer data out the MSB of data
 pwi1_clock_pulse();  //Clock each bit out
data <<= 1;  //Slide the next MSB into place
   }
 // STOP Bit
 SPWI = LO;
 SCLK = HI;  //Take the SCLK pin high
 SPWI_DIR = IN;  // If this is a read command, the EMU will begin to drive SPWI on falling edge
 SCLK = LO;  //Complete the clock cycle
 return(error);  //Currently no error checking, any needed can be added
   }
 byte pwi1_read_frame(byte* data)
{
 byte error = 0;  //Return value
byte bit = 0;  //Used as data counter to handle serial-to-parallel conversion
 SPWI_DIR = IN;  //Ensure that SPWI is an input
 // Look for START Bit from EMU
 if(SPWI == LO){
  error = 1;  //No START Bit from slave, error code = 1
 }
 pwi1_clock_pulse();  //Acknowledge the START Bit
// Now ensure EMU drives 2 Reserved Bits, 00b
 if(SPWI == HI){
 error = 2;  //Reserve Bit 1 not correct, error code = 2
 }
 pwi1_clock_pulse();  //Clock in first Reserved Bit
 if(SPWI == HI){
  error = 2;  //Reserve Bit 2 not correct, error code = 2
  }
  pwi1_clock_pulse();  //Clock in second Reserved Bit
 for(bit = 0; bit < 8; bit++){
 (*data) = SPWI;  //Grab the bit
 pwi1_clock_pulse();  //Tell EMU we got it
 (*data) <<= 1;  //Make room for the next most significant bit
}
  // Look for STOP Bit from EMU
if(SPWI == HI){
error = 3;  //No STOP Bit from slave, error code = 3
     }
   pwi1_clock_pulse();
  return(error);  //Currently no error checking, any needed can be added
 }
```

This is a supporting function that toggles the SCLK GPIO pin. This could also be done with a macro, but is not here to aid clarity.

```
 void pwi1_clock_pulse(void)
{
     // SCLK assumed low upon entering
 SCLK = HI;
  /*** INSERT SOME DELAY HERE TO EVEN OUT THE DUTY CYCLE AND ENSURE THAT YOU
  MEET THE MINIMUM PULSE WIDTH REQUIREMENTS OF THE SPEC. ***/
  SCLK = LO;
 return();
 }
```

We now proceed to the register write and register read commands. A very bare-bones implementation could get by with nothing more than these two commands, but it is recommended that all PWI functionality be implemented.

```
 void pwi1_reg_write(byte register_number, byte write_data)
{
 byte error = 0;  //Error message; how to process?
  /* Note that this function does not ensure whether the register requested is valid,
   but it does lop off the high nibble since there are only 15 registers available. */
 register_number &= 0x0F;
 error = pwi1_write_frame(CMD_REG_WRITE | register_number);  //Tell the EMU what we want
  error = pwi1_write_frame(write_data);  //Push the data to the register
  return();
  }
 byte pwi1_reg_read(byte register_number)
 {
 byte error = 0;  //Error message, how to process?
byte register_contents = 0;  //The data that was read
 /* Note that this function does not ensure whether the register requested is valid,
   but it does lop off the high nibble since there are only 15 registers available. */
 register_number &= 0x0F;
  error = pwi1_write_frame(CMD_REG_READ | register_number);  //Tell the EMU what we want
  // Here we must handle the turn-around clock pulse
   pwi1_clock_pulse();
 // Now read what the SPC is driving back
 error = pwi1_read_frame(& register;_contents);
 // Send back what we got
return(register_contents);
   }
```

Below are the command functions for PWI. We start with the two most important, Core Voltage Adjust, and the Synchronize command. The synchronize command should be implemented and called at start-up to ensure that the EMU is synchronized with the master.

```
 byte pwi1_core_v_adjust(byte voltage)
 {
  byte error = 0;  //Error message
 if(voltage & 0x80){
   error = 1;  //The MSB of R0 is reserved per PWI spec
 }
else{
  pwi1_write_frame(CORE_V_ADJ | voltage);  //Command with MSB set indicates Core V Adjust
 }
return(error);
   }
void pwi1_synchronize(void)
{
 /* The synchronize command is a series of 11 1’s followed by a STOP bit
 byte bit = 0;  //Used for the counter below
SPWI_DIR = OUT;  //Make sure we are driving the SPWI pin
 // START Bit
 SPWI = HI;
 CLK_PULSE;
  // Send the two reserved bits, 11b, note that SPWI is already logic high
 CLK_PULSE;  //Reserved 1
CLK_PULSE;  //Reserved 2
// Now we will do the payload bits, note that SPWI is already logic high
 for(bit = 0; bit < 8; bit++){
 CLK_PULSE;  //Clock each bit out
 }
  // Stop Bit is a logic low
  SPWI = LO;
CLK_PULSE;
return(void);
    }
 void pwi1_reset(void) {
pwi1_write_frame(CMD_RESET);  //CMD_RESET = 0x10
return(void);
 }
 void pwi1_sleep(void) {
 pwi1_write_frame(CMD_SLEEP);  //CMD_SLEEP = 0x11
   return(void);
 }
 void pwi1_shutdown(void) {
 pwi1_write_frame(CMD_SHUTDOWN);  //CMD_SHUTDOWN = 0x12
return(void);
 }
void pwi1_wakeup(void) {
 pwi1_write_frame(CMD_WAKEUP);  //CMD_WAKEUP = 0x13
 return(void);
 }
```

Products		**Applications**	
Audio	www.ti.com/audio	Automotive and Transportation	www.ti.com/automotive
Amplifiers	amplifier.ti.com	Communications and Telecom	www.ti.com/communications
Data Converters	dataconverter.ti.com	Computers and Peripherals	www.ti.com/computers
DLP® Products	www.dlp.com	Consumer Electronics	www.ti.com/consumer-apps
DSP	dsp.ti.com	Energy and Lighting	www.ti.com/energy
Clocks and Timers	www.ti.com/clocks	Industrial	www.ti.com/industrial
Interface	interface.ti.com	Medical	www.ti.com/medical
Logic	logic.ti.com	Security	www.ti.com/security
Power Mgmt	power.ti.com	Space, Avionics and Defense	www.ti.com/space-avionics-defense
Microcontrollers	microcontroller.ti.com	Video and Imaging	www.ti.com/video
RFID	www.ti-rfid.com		
OMAP Applications Processors	www.ti.com/omap	**TI E2E Community**	e2e.ti.com
Wireless Connectivity	www.ti.com/wirelessconnectivity		