Burr-Brown Products from Texas Instruments

Application Report
SBAA075 – March 2002

Windows CE Touch and Keypad Device Drivers for the TSC2200

Bob Green and Rick Downs *Data Acquisition Products*

ABSTRACT

This application report describes the development and usage of device drivers for the TSC2200 PDA Analog Interface Circuit for use in Windows™ CE platforms on the Intel™ StrongARM™ SA-1110 microprocessor and derivatives. The methods described are only some of the possible implementations and are intended only to serve as a guideline for hardware and software developers in developing their own solutions.

Contents

Introduction **2**
Hardware Considerations **2**
Software Architecture **3**
Functional Description **4**
System Setup 4
TSC2200 Touch Driver Functionality Overview 4
TSC2200 Keypad Driver Functionality Overview 7
SSP Overview 8
Source Code **9**
Individual Function Descriptions 10

Figures

Figure 1. Integration of Monolithic Device Drivers Within the Windows CE System **3**
Figure 2. Driver Operation Flowchart **6**

Tables

Table 1. Hardware Connections **3**
Table 2. Keypad Mapping **7**

Introduction

The Intel SA-1110 StrongARM processor and its derivatives are a popular platform for development of PDAs and other hand-held computing devices. The human interface for these devices is generally a touch screen.

A popular operating system for these devices is Windows CE. Drivers for both the touch screen and keypad are presented in this document, since the TSC2200 supports both interfaces. The touch screen driver alone will work well with the TSC2000 PDA Analog Interface circuit, since this device does not support a keypad.

The methods described herein are only some of the possible implementations and are only intended to serve as a guideline for hardware and software developers in developing their own solutions. The sample setup and drivers provided only demonstrate the basic functionality of using the TSC2200 with the SA-1110, but have not been extensively tested, and are not intended for use “as is” in production systems.

The document assumes the reader is familiar with hardware, touch screen, and keyboard device driver development in general, and specifically with the Windows CE Platform Builder 3.0 development environment, the Texas Instruments TSC2200 PDA Analog Interface Circuit, and the Intel StrongARM SA-1110 Microprocessor and Development Board.

Hardware Considerations

The development system developed for this project consisted of a Texas Instruments TSC2200EVM evaluation board and a modified Intel SA-1110 development board, which includes a touch screen. The SA-1110 development board is modified to allow connecting the TSC2200 in place of its touch screen controller. A full description of the modifications required to the Intel SA-1110 board is beyond the scope of this document. However, a brief summary of the necessary signal connections is described below.

In order to connect the TSC2200 to the Intel SA-1110 StrongArm processor, this project started with the same modifications used for the ADS7846 development system (please refer to Application Report SBAA070, “Using the ADS7846 Touch Screen Controller with the Intel SA-1110 StrongArm Processor” located at www.ti.com).

Following the ADS7846 solution, the TSC2200 is connected to the SA-1110’s serial port 4 through GPIO lines 10 to 13. This allows the use of the SA-1110’s built-in Motorola SPI™ communications interface.

The $\overline{\text{DAV}}$ signal from the TSC2200 is connected to the SA-1110’s GPIO line 23. This line is normally connected to the UCB1300 on the Intel SA-1110 development board.

The lack of unused GPIO lines on the SA-1110 development board means that something must be disabled in order to hook up the TSC2200’s $\overline{\text{KBIRQ}}$. The $\overline{\text{KBIRQ}}$ signal is connected to the SA-1110’s GPIO line 1 through the SW2 connection. **Please note that since the boot loader and other Intel development tools also use SW2, this signal should not be connected until after the Windows CE image has been loaded on the SA-1110. If the TSC2200 pulls this signal high during startup, the boot loader will not reset the device or download an image.**

Table 1 summarizes the connections required to duplicate this development system.

Table 1. Hardware Connections

TSC2200	SA-1110
$\overline{\text{DAV}}$	GPIO23
$\overline{\text{KBIRQ}}$	GPIO01
$\overline{\text{SS}}$	GPIO13
SCLK	GPIO12
MOSI	GPIO10
MISO	GPIO11

Software Architecture

Both the touch screen and keypad drivers have been developed as “layered” drivers using the Microsoft sample drivers as starting points. In addition, the touch panel driver relies upon the ADS7846 (another touch screen controller from Texas Instruments) driver design. This approach helped reduce overall project risk since the Model Device Driver (MDD) layer (supplied by Microsoft in the sample drivers) did not require any change. Only the DDSI functions in the PDD (Platform-Dependent Driver) layer were modified. Figure 1 shows the integration theory.

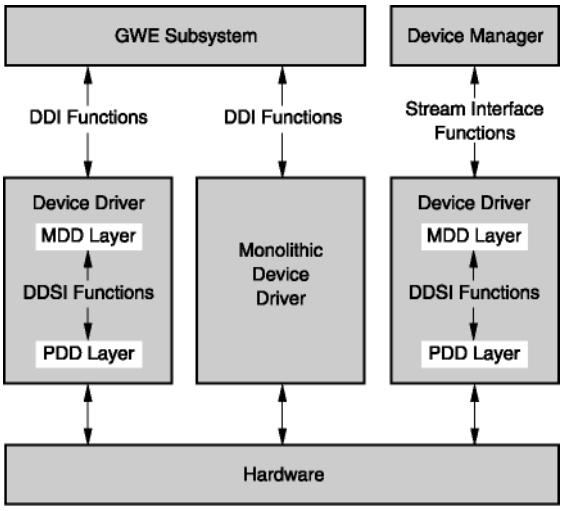

Figure 1. Integration of Monolithic Device Drivers Within the Windows CE System. [1]

Functional Description

System Setup

1. Setup the SA-1110 Development Board in stand-alone configuration (not using the SA-1111) according to the manufacturer's documentation.
2. Update the platform files (this assumes you have already installed the Platform Builder system and the Intel SA-1110 files).

   Update the SA11x0BD platform files by unzipping sa11X0bd_TSC2200_WCE300.zip to C:\, where C:\WINCE300 is the directory in which you have Platform Builder installed.

   Rebuild the SA11x0BD platform (see Microsoft documentation).
3. Connect the TSC2200EVM to the SA-1110 development board.
4. Ensure that the $\overline{\text{KBIRQ}}$ line is NOT connected.
5. Download image to target (see Microsoft documentation).
6. Once the calibration program has started, connect the $\overline{\text{KBIRQ}}$ line. Please note that since $\overline{\text{KBIRQ}}$ is connected after system startup, a spurious interrupt may have occurred when it was connected. Due to this, the first keystroke may not be recognized, but all subsequent ones should.
7. Calibrate the touch screen (will run automatically at boot). Once finished with calibration, press the 'F' (mapped to ENTER) on the TSC2200 keypad to accept the new calibration data.

TSC2200 Touch Driver Functionality Overview

The TSC2200 is setup to perform automatic X and Y conversions when the screen is pressed. Therefore, when the screen is pressed on the development board, the TSC2200 performs its X and Y conversions and asserts (active low) the $\overline{\text{DAV}}$ line. This line stays low until the X and Y values are read from the appropriate data registers at which point the $\overline{\text{DAV}}$ line goes back high. At this point, if the TSC2200 were to stay in continuous scan mode, the next conversion would start immediately after the first was finished. Then, after the X and Y data was read, the $\overline{\text{DAV}}$ line would stay high only until the next conversion was finished, and then it would be re-asserted. This caused some problems for the driver because the $\overline{\text{DAV}}$ line goes high as soon as the TSC2200 interprets the command to read the X and Y data, not waiting until the data has been clocked out. What occasionally occurred was that the part would be mostly through with a conversion when the driver requested the X and Y data. $\overline{\text{DAV}}$ would then go high, and the conversion would finish. However, $\overline{\text{DAV}}$ would go low again before the processor had time to clock the X and Y data in and re-enable the falling edge interrupt for $\overline{\text{DAV}}$. This would result in disabling the driver, since $\overline{\text{DAV}}$ would not go high until the data is read, and since the processor missed the falling edge, it would never know data was available.

Therefore, to avoid this issue, one of the first things done in the interrupt routine is to command the TSC2200 to stop conversions. After reading the data and all other processing, conversions are restarted on the TSC2200. This prevents the aforementioned problem.

Another issue that has to be dealt with is the fact that there is no single query to the TSC2200 that tells the host whether the pen is still down or not. $\overline{\text{PENIRQ}}$ and its sister register can partially tell this, but since $\overline{\text{PENIRQ}}$ (and, therefore, its mirror bit) pulses when each new conversion is started, it may indicate that the pen is lifted when in fact it is still down. This caused the driver difficulty in recognizing an actual pen up. To get around this issue, the interrupt is enabled only on a falling edge of $\overline{\text{DAV}}$. Then a timer interrupt is set to go off after a sufficient time to allow the TSC2200 to complete a conversion and assert $\overline{\text{DAV}}$ again, if the pen is still down. If a new $\overline{\text{DAV}}$ falling edge interrupt is triggered before the timer fires, the timer is reset and the routine continues. If the timer fires before another $\overline{\text{DAV}}$ interrupt is received, the processor checks to see if the pen is up. If so, it assumes the pen as actually been lifted and passes a pen up command to the system.

Figure 2 is a flowchart that summarizes the order of events.

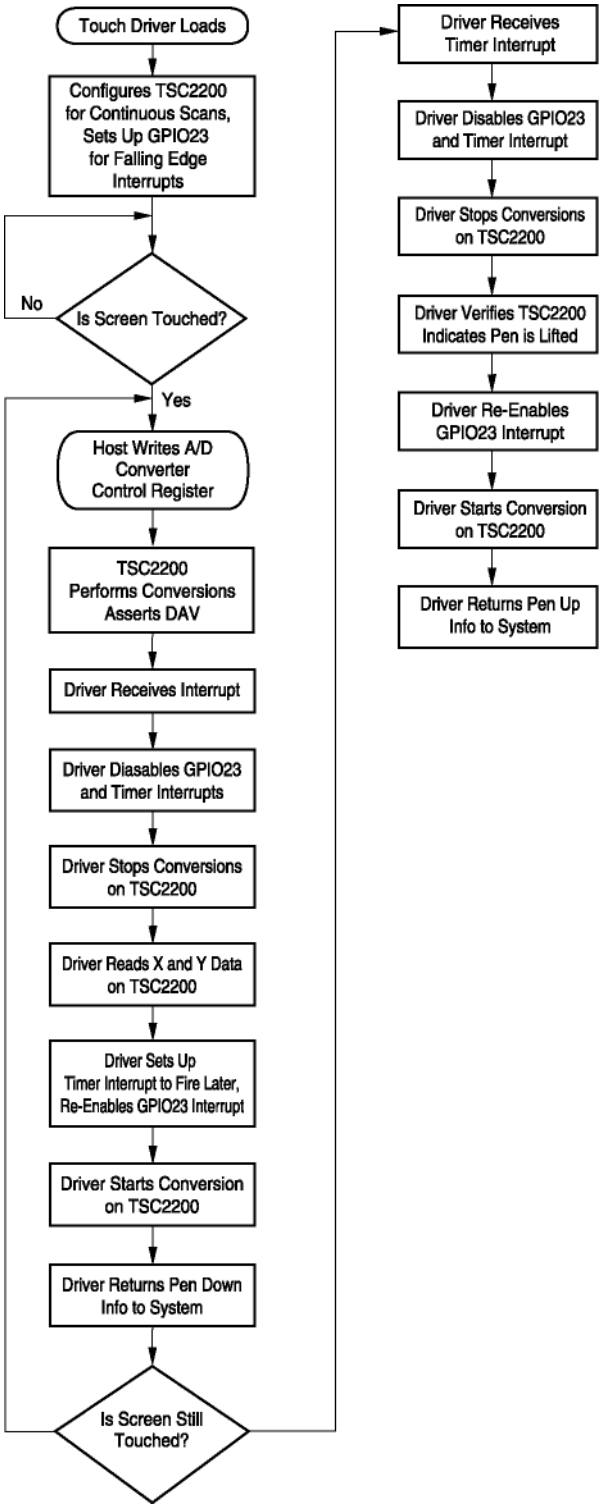

Figure 2. Driver Operation Flowchart

TSC2200 Keypad Driver Functionality Overview

The keypad driver actually works very similarly to the touch driver. $\overline{\text{KBIRQ}}$ is asserted (low) when a key is pressed. As with the touch driver's $\overline{\text{DAV}}$ line, $\overline{\text{KBIRQ}}$ also pulses high and then low again as the TSC2200 continues scanning the keypad while a button is still down. Again, this caused problems with recognizing the actual button up event. The keypad driver uses an approach similar to that used in the touch driver. When the $\overline{\text{KBIRQ}}$ interrupt fires, a timer is set long enough to allow the TSC2200 to complete at least one more keypad scan. When a timer interrupt is received before the next $\overline{\text{KBIRQ}}$ interrupt and the TSC2200 says that no keys are pressed, the driver assumes an actual button release occurred and acts accordingly. This timer is set in the KeybdIstLoop function in KB_DIST.CPP and it has been found that setting it to 1.5 times the TSC2200's debounce time works well.

The standard Windows AutoRepeat key functionality will not currently work with this driver. This is due to the fact that since the $\overline{\text{KBIRQ}}$ pulses high between each scan of the keypad. This pulse causes another $\overline{\text{KBIRQ}}$ falling edge interrupt to occur before the autorepeat minimum timeout can occur. Therefore, no autorepeating occurs in this driver.

There are two arrays (arrVkeys and arrScanCodes) at the top of TSC_KEYPD.CPP that are used to map the keypad keys to Windows Virtual Keys. Entry 0 in these arrays matches to Bit 0 in the KEYDATA register of the TSC2200. The current mapping of the keypad keys is shown in Table 2.

Table 2. Keypad Mapping

TSC2200 Key	Windows Virtual Key
0	'0'
1	'1'
2	'2'
3	'3'
4	'4'
5	'5'
6	'6'
7	'7'
8	'8'
9	'9'
A	VK_LEFT
B	VK_UP
C	VK_RIGHT
D	VK_DOWN
E	VK_ESCAPE
F	VK_RETURN

There are also two #defines (KEYREG_SETUP_VALUE and KEYMASKREG_SETUP_VALUE) in TSC_SSP.H that are used to setup the TSC2200 KEY and KPMASK registers.

SSP Overview

In order to share the single SPI connection with the TSC2200 between both the keypad and the touch drivers, a new module (tsc_ssp.cpp) was added to the DRVLIB library in the SA11x0bd platform. This module sets up, configures, and uses the SA-1110's SPI interface to communicate with the TSC2200 and uses a named Mutex to prevent collision between the two drivers. Additionally, since the Keyboard driver loads first, it is allowed to initially reset and configure the TSC2200, with the touch driver making any additional configuration changes it requires when it loads.

The SA-1110's SSP port is configured to use the Motorola SPI format with the SPH and SPO registers configured to match the requirements of the TSC2200 as follows:

- dss, DSS_16_BIT
- frf, FRF_MOTOROLA
- scr, SCR_600_KHZ
- rie, RIE_DISABLE
- tie, TIE_DISABLE
- lbm, LBM_DISABLE
- spo, SPO_IDLE_LOW
- sp, SPH_HALF_DELAY
- ecs, ECS_INTERNAL
- spr, SPR_USE_GPIO
- gafr, gp10, GAFR_ALTERNATE
- gafr, gp11, GAFR_ALTERNATE
- gafr, gp12, GAFR_ALTERNATE
- gafr, gp13, GAFR_ALTERNATE
- gpdr, gp10, GPDR_OUTPUT (SSP_TXD)
- gpdr, gp11, GPDR_INPUT (SSP_RXD)
- gpdr, gp12, GPDR_OUTPUT (SSP_SCLK)
- gpdr, gp13, GPDR_OUTPUT (SSP_SFRM)

Refer to the SA-1110 Developer's Manual, section 11.12.10.5, for more information on these registers.

One issue arises when using the SSP interface on the SA-1110 with the TSC2200. This relates to the TSC2200's command interface. Only the first word, after SFRM goes low, is interpreted as a command (refer to the TSC2200 data sheet). Therefore, if more than one command is to be sent, the driver must ensure that SFRM goes high in between the two commands. There is no direct way to control this while the SA-1110 is controlling the clocking in and clocking out of data. This is resolved in this driver by waiting a certain amount of time, based upon the SPI serial clock rate setup (see the SA-1110 Developer's Manual, section 11.12.9.4, for more info). This allows SFRM to go high before sending the next command.

SOURCE CODE

The following list represents those files either added or modified during this project with a short summary of what was done.

1. C:\WINCE300\PLATFORM\SA11x0bd\

   sa11x0bd.bat—changed the define to include a keyboard even without the SA-1111 companion board.

2. C:\WINCE300\PLATFORM\SA11x0bd\inc\

   TSC_SSP.H—new file contains prototypes and config defines for TSC2200 and SSP communications.

   Sa11X0BD.h—changed the TRACKPADRX interrupt macros to use GPIO01.

   sa11X0.h—removed a duplicate definition of armRegisterStruct.

3. C:\WINCE300\PLATFORM\SA11x0bd\drivers\TOUCHP\

   Tchpdd.cpp—major rewrite to work with TSC2200.

4. C:\WINCE300\PLATFORM\SA11x0bd\drivers\Kbdmouse\KBDMSCOMMON\

   KBDIST.CPP—rewrote isr routine.

   TSC_KEYPD.CPP—major changes for TSC2200.

   KBDMOUSE.CPP—change DLLMAIN for TSC2200.

   SOURCES—removed references to PS2 stuff, added TSC_KEYPD.

5. C:\WINCE300\PLATFORM\SA11x0bd\drivers\DRVLIB\

   TSC_SSP.cpp—new file to handle SSP and TSC communications.

   SOURCES—added TSC_SSP.

6. C:\WINCE300\PLATFORM\SA11x0bd\KERNEL\hal\

   Cfwsarm.c—changed interrupt config routines for keypad and touch.

7. C:\WINCE300\PLATFORM\SA11x0bd\KERNEL\hal\arm\

   int11x0.c—changed ISR function for keypad and touch.

Individual Function Descriptions

Descriptions of the individual functions used in these drivers are given below. This document assumes the reader understands Windows CE touch and keyboard drivers in general and will not cover every function needed for a working driver. Those functions that required modification in order to work with the TSC2200 are listed. Functions not listed below were not modified and worked as they were, either from the Microsoft, Intel, or ADS7846 code.

TSC_SSP Interface

(All the SSP functions are new for this driver.)

InitSSP—This function initialized the SSP interface, and should be the first SSP function that drivers should call.

```
void InitSSP(void)
{
 // setup the structures and pointers to the SSP registers
 SetupSSPRegisters();

 // configure the SSP registers
 setupSSP();

}
```

SetupSSPRegisters—Internal function called from within InitSSP to initialize the structures that point to the SSP registers.

```
void SetupSSPRegisters(void)
{
 //copied directly from the ADS7825 version driver

 // check to ensure we don't do this more than once per thread
 // but we don't use the start mutex, because each thread needs a copy.
 if (v_pSSPreg == NULL)
 {
       v_pSSPreg =  (volatile struct sspreg *) VirtualAllocCopy(sizeof(struct sspreg),
(char *)TEXT("TouchPanelEnable: SSP_BASE_VIRTUAL"), (PVOID)SSP_BASE_VIRTUAL);
       if (v_pSSPreg)
       {
              if (v_pPPCReg == NULL)
              {
                     v_pPPCReg = (volatile struct ppcreg *)
VirtualAllocCopy(sizeof(struct ppcreg),
                           (char *)TEXT("TouchPanelEnable: PPC_BASE_VIRTUAL"),
(PVOID)PPC_BASE_VIRTUAL);
              }
       }
 }

}
```

SetupSSP—Internal function called from within InitSSP to setup the SSP registers to work with the TSC2200.

```
void setupSSP()
{

 volatile struct icreg *v_pICReg  = (volatile struct icreg *)IC_BASE_VIRTUAL;
 volatile struct gpioreg *v_pGPIOReg = (volatile struct gpioreg *)GPIO_BASE_VIRTUAL;



 // Bg 19FEB02 - changing to using named mutex for protection
 if (CreateOurMutex())
 {
        if (GrabOurMutex())
        {
               // Disable SSP
               IOW_REG_FIELD (struct sscr0Bits, &v_pSSPreg->sscr0, sse, SSE_DISABLE);

               // Data size select = 16
               IOW_REG_FIELD (struct sscr0Bits, &v_pSSPreg->sscr0, dss, DSS_16_BIT);


               // Frame format = Motorola frame format
               IOW_REG_FIELD (struct sscr0Bits, &v_pSSPreg->sscr0, frf, FRF_MOTOROLA);

               // Serial clock rate = 614.4KHz
               IOW_REG_FIELD (struct sscr0Bits, &v_pSSPreg->sscr0, scr, SCR_600_KHZ);


               // Program SSP control register 1

               // Receive interrupt disabled
               IOW_REG_FIELD (struct sscr1Bits, &v_pSSPreg->sscr1, rie, RIE_DISABLE);
```

```
            // Transmit interrupt disabled
            IOW_REG_FIELD (struct sscr1Bits, &v_pSSPreg->sscr1, tie, TIE_DISABLE);



            // Loopback mode disabled
            IOW_REG_FIELD (struct sscr1Bits, &v_pSSPreg->sscr1, lbm, LBM_DISABLE);


            // Serial clock polarity = inactive low
            IOW_REG_FIELD (struct sscr1Bits, &v_pSSPreg->sscr1, spo, SPO_IDLE_LOW);


            // Serial clock phase (SCLK wait half SCLK period after SFRM asserted)
            IOW_REG_FIELD (struct sscr1Bits, &v_pSSPreg->sscr1, sp, SPH_HALF_DELAY);

            // Internal clock select
            IOW_REG_FIELD (struct sscr1Bits, &v_pSSPreg->sscr1, ecs, ECS_INTERNAL);



            // Program PPC Pin Assignment Register (PPAR)

            // SSP pin reassignment
            IOW_REG_FIELD (struct pparBits, &v_pPPCReg->ppar, spr, SPR_USE_GPIO);


            // Program GPIO alternate function register (GAFR)
            WRITE_BITFIELD (struct gpioregBits, &v_pGPIOReg->gafr, gp10,
GAFR_ALTERNATE);
            WRITE_BITFIELD (struct gpioregBits, &v_pGPIOReg->gafr, gp11,
GAFR_ALTERNATE);
            WRITE_BITFIELD (struct gpioregBits, &v_pGPIOReg->gafr, gp12,
GAFR_ALTERNATE);
            WRITE_BITFIELD (struct gpioregBits, &v_pGPIOReg->gafr, gp13,
GAFR_ALTERNATE);
```

```
                // Program GPIO pin direction register (GPDR)


                //SSP_TXD
                WRITE_BITFIELD (struct gpioregBits, &v_pGPIOReg->gpdr, gp10,
GPDR_OUTPUT);


                //SSP_RXD
                WRITE_BITFIELD (struct gpioregBits, &v_pGPIOReg->gpdr, gp11,
GPDR_INPUT);


                //SSP_SCLK
                WRITE_BITFIELD (struct gpioregBits, &v_pGPIOReg->gpdr, gp12,
GPDR_OUTPUT);


                //SSP_SFRM
                WRITE_BITFIELD (struct gpioregBits, &v_pGPIOReg->gpdr, gp13,
GPDR_OUTPUT);


                // Synchronous serial port enable (SET LAST!)
                IOW_REG_FIELD (struct sscr0Bits, &v_pSSPreg->sscr0, sse, SSE_ENABLE);


                // Bg 19FEB02
                ReleaseOurMutex();


        } else // GrabOurMutex
        {
                RETAILMSG(1, (TEXT("setupSSP - Failed to grab our mutex\r\n")));
        }


 } else // CreateOurMutex
 {
        RETAILMSG(1, (TEXT("setupSSP - Failed to grab our mutex\r\n")));
 }


}
```

DisableSSP—Called to shut off SSP operation when configuring the SSP interface or when powering down.

```
void DisableSSP(void)
{
 if (GrabOurMutex())
 {
        // disable SPP operation using the SSCR/SSE register
        IOW_REG_FIELD (struct sscr0Bits, &v_pSSPreg->sscr0, sse, 0);

        ReleaseOurMutex();

 } else // GrabOurMutex
 {
        RETAILMSG(1, (TEXT("DisableSSP - Failed to grab our mutex\r\n")));
 }

}
```

EnableSSP—Internal function called to restart SSP after configuration is done.

```
void EnableSSP(void)
{
 if (GrabOurMutex())
 {
        // enable SPP operation using the SSCR/SSE register
        IOW_REG_FIELD (struct sscr0Bits, &v_pSSPreg->sscr0, sse, 1);

        ReleaseOurMutex();

 } else // GrabOurMutex
 {
        RETAILMSG(1, (TEXT("EnableSSP - Failed to grab our mutex\r\n")));
 }
}
```

The following functions send commands to the TSC2200 to either write or read registers. Their names imply what registers they read or write. Only two need additional discussion: TSC2200WriteKeyReg and TSC2200WriteKeyMaskReg. In order to avoid the problem (in Rev B of the TSC2200) where the internal clock must be running for new values to be properly written to the KEY and KPMASK registers, a slow Port Scan is first started by writing to the ADC register, then the driver writes to the KEY or KPMASK register. Then to avoid a problem with $\overline{\text{DAV}}$ going low and confusing the touch driver, the port scan conversion is stopped. As these two registers (KEY and KPMask) are only written to during initialization of the keypad driver, this does not unduly affect performance.

```
unsigned __int16 TSC2200ReadADCReg(void)
void TSC2200WriteADCReg(unsigned __int16 newValue)
BOOL TSC2200ReadXY(unsigned __int16 *pXdata, unsigned __int16 *pYdata)
void TSC2200Reset(void)
unsigned __int16 TSC2200ReadKPDataReg(void)
unsigned __int16 TSC2200ReadKeyReg(void)
void TSC2200WriteKeyReg(unsigned __int16 newValue)
void TSC2200WriteKeyMaskReg(unsigned __int16 newValue)
unsigned __int16 TSC2200ReadKeyMaskReg(void)
unsigned __int16 TSC2200ReadCFGReg(void)
void TSC2200WriteCFGReg(unsigned __int16 newValue)
unsigned __int16 TSC2200ReadREFReg(void)
void TSC2200WriteREFReg(unsigned __int16 newValue)
unsigned __int16 TSC2200ReadDACReg(void)
void TSC2200WriteDACReg(unsigned __int16 newValue)
void TSC2200StopConversions(void)
void TSC2200StartConversions(void)
```

Touch Interface

These functions have been modified from the ADS7846 touch screen driver.

DdsiTouchPanelEnable—Added code to setup the SSP interface, GPIO23 falling edge interrupts, and the TSC2200 itself.

```
BOOL DdsiTouchPanelEnable()
{
 // setup all the structures needed to store pointers to various system pointers
 SetupTouchDriverStructs();

    // Set up mutex for access to shared registers
    if (!v_hTchAudMutex || !v_pOSTReg  || !v_pGPIOReg||
        !v_pICReg || !v_pDriverGlobals || !v_pMCPReg) {
        PddpTouchPanelDeallocateVm();
         DEBUGMSG(ZONE_ERROR,(TEXT("DdsiTouchPanelEnable(): Error
%u\r\n"),GetLastError()));
         return (FALSE);
    }

 // Setup pen down interrupts, but leave ints disabled until InterruptEnable().

 // Bg 20FEB02
 UCB1X00_TOUCH_PIN_DIRECTION(0);
 UCB1X00_TOUCH_ALTERNATE(0);

 // setup the GPIO lines for the IRQ
 UCB1X00_TOUCH_RISING_EDGE_CLR;          // Make FS falling edge triggered
 UCB1X00_TOUCH_FALLING_EDGE_SET;
 UCB1X00_TOUCH_INT_SET(1);               // clear interrupt just generated

 GPIO_GRP11_27_INT_MASK(1);
```

```
 v_pDriverGlobals->tch.touchIrq=0;
 v_pDriverGlobals->tch.timerIrq=0;
 v_pDriverGlobals->tch.penUpMissed=0;
 v_pDriverGlobals->tch.penUpFake=0;

 nextExpectedInterrupt=PEN_DOWN;



 // BG20FEB02 - setup the SSP system
 InitSSP();

 // Bg 20FEB02 - setup the main ADC REG
 TSC2200WriteADCReg(ADC_SETUP_VALUE);

    return(TRUE);     // we always succeed!!!!!!
}
```

SampleTouchScreenADS7846—As the name implies, this function was originally taken from the ADS7846 touch driver. However, it has been completely rewritten and simplified greatly for the TSC2200. It merely reads the X and Y data, checks to see that they are within proper bounds, and returns appropriate values.

```
TOUCH_PANEL_SAMPLE_FLAGS SampleTouchScreenADS7846(INT *x,INT *y)
{
 unsigned __int16 iReadX;
 unsigned __int16 iReadY;
 unsigned __int16 iADC;

 TOUCH_PANEL_SAMPLE_FLAGS TmpStateFlags = TouchSampleDownFlag;

 //Bg 12/28/01
 // read X and Y coord.
 TSC2200ReadXY(&iReadX, &iReadY);

 // check to ensure that the point is within 12 bit bounds

 if ( ((iReadX < 4095) && (iReadX > 0)) && ((iReadY < 4095) && (iReadY > 0)))
 {
       *x = (INT)(iReadX);
       *y = (INT)(iReadY);

       TmpStateFlags |= TouchSampleValidFlag;

 }
 else
 {
       TmpStateFlags |= TouchSampleIgnore;
 }

 return(TmpStateFlags);

}
```

DdsiTouchPanelGetPoint—This is the main workhorse of the touch driver. It gets called whenever the $\overline{DAV}$ interrupt fires, as well as when the timer interrupt fires. It disables and re-enables interrupts, stops and starts TSC2200 conversions, and calls SampleTouchScreenADS7846 to get X and Y data.

```
VOID DdsiTouchPanelGetPoint(TOUCH_PANEL_SAMPLE_FLAGS *pTipStateFlags,INT *pUncalX,INT
*pUncalY)
{
 static TOUCH_PANEL_SAMPLE_FLAGS PrevStateFlags = TouchSampleIgnore;
 static INT iPrevX=0;
 static INT iPrevY=0;
 unsigned InterruptType=SYSINTR_TOUCH;
 int iPenDown = 0;
 unsigned __int16 iData                  = 0;
 unsigned __int16 iTempData              = 0;
 unsigned __int16 iKeyData               = 0;

 // set default state flag
 *pTipStateFlags = TouchSampleIgnore;

 // read the ADC reg
 iTempData = TSC2200ReadADCReg();

 // record whether pen is down or not
 if (iTempData & 0x8000) // pen is down
       iPenDown = 1;
 else
       iPenDown = 0;

 // an actual falling edge on DAV or PENIRQ not the timer interrupt
 if(v_pDriverGlobals->tch.touchIrq)
 {
       v_pDriverGlobals->tch.touchIrq=0;
       v_pDriverGlobals->tch.timerIrq=0;
       InterruptType=SYSINTR_TOUCH_CHANGED;
```

```
        if (iPenDown)
        {
                // turn off conversions so that we don't miss a DAV down trigger while
we are in here.
                TSC2200StopConversions();

                // take sample
                *pTipStateFlags=SampleTouchScreenADS7846(pUncalX,pUncalY);

                if (PrevStateFlags & TouchSampleValidFlag)
                        *pTipStateFlags |= TouchSamplePreviousDownFlag;

                iPrevX = *pUncalX;
                iPrevY = *pUncalY;
                PrevStateFlags=*pTipStateFlags;
                nextExpectedInterrupt=PEN_UP_OR_TIMER;

                // reset interrupt for falling edge
                UCB1X00_TOUCH_FALLING_EDGE_SET;
                UCB1X00_TOUCH_RISING_EDGE_CLR;
                UCB1X00_TOUCH_INT_SET(1);        // clear interrupt just generated

                InterruptDone(InterruptType);

                // don't enable timer INT until done
                enableTouchTimerInterrupt(TOUCH_TIMER_INCREMENT);

                // turn conversions back on.
                TSC2200StartConversions();

        } else
        {       // Rising Edge Trigger on DAV
                // we should never get here – but just in case something else in the
system
                // sets the GPIO to trigger on rising edge, we'll handle it.
```

```
            // turn off timer, since we are done
            disableTouchTimerInterrupt();

            // turn off conversions
            TSC2200StopConversions();

            // Need to read data to ensure that DAV goes high for next time.
            SampleTouchScreenADS7846(pUncalX,pUncalY);

            // set the stateflags ourself to ensure the pen up is handled
            *pTipStateFlags = TouchSampleValidFlag;

            // record if the pen was down last time
            if (PrevStateFlags & TouchSampleValidFlag)
                  *pTipStateFlags |= TouchSamplePreviousDownFlag;

            // use the previous down X Y location
            *pUncalX = iPrevX;
            *pUncalY = iPrevY;
            PrevStateFlags = TouchSampleIgnore;
            nextExpectedInterrupt=PEN_DOWN;

            // reset interrupt for falling edge
            UCB1X00_TOUCH_FALLING_EDGE_SET;
            UCB1X00_TOUCH_RISING_EDGE_CLR;
            UCB1X00_TOUCH_INT_SET(1);        // clear interrupt just generated

            InterruptDone(InterruptType);

            // turn on conversions
            TSC2200StartConversions();

      }

}
```

```
else if(v_pDriverGlobals->tch.timerIrq)
{
       // Timer interrupt

       v_pDriverGlobals->tch.timerIrq=0;

       if (iPenDown)
       {
              // turn off conversions
              TSC2200StopConversions();

              *pTipStateFlags=SampleTouchScreenADS7846(pUncalX,pUncalY);

              // Save location for when we get a pen up
              iPrevX = *pUncalX;
              iPrevY = *pUncalY;
              PrevStateFlags=*pTipStateFlags;
              nextExpectedInterrupt=PEN_UP_OR_TIMER;

              InterruptDone(InterruptType);
              enableTouchTimerInterrupt(TOUCH_TIMER_INCREMENT);

              // turn on conversions
              TSC2200StartConversions();

       }
       else
       {

              // if we get the timer and the TSC2200 says the pen is up,
              // that means the pen really has been lifted.

              // turn off timer, since we are done
              disableTouchTimerInterrupt();
```

```
                // turn off conversions
                TSC2200StopConversions();

                // Need to read data to ensure that DAV goes high for next time.
                SampleTouchScreenADS7846(pUncalX,pUncalY);

                // act like this is a pen up
                *pTipStateFlags = TouchSampleValidFlag;

                if (PrevStateFlags & TouchSampleValidFlag)
                       *pTipStateFlags |= TouchSamplePreviousDownFlag;

                *pUncalX = iPrevX;
                *pUncalY = iPrevY;
                PrevStateFlags = TouchSampleIgnore;
                nextExpectedInterrupt=PEN_DOWN;

                // reset interrupt for falling edge
                UCB1X00_TOUCH_FALLING_EDGE_SET;
                UCB1X00_TOUCH_RISING_EDGE_CLR;
                UCB1X00_TOUCH_INT_SET(1);        // clear interrupt just generated
                InterruptDone(InterruptType);

                // turn on conversions
                TSC2200StartConversions();
        }
 }
 else
 {
        *pTipStateFlags = TouchSampleIgnore;
        PrevStateFlags = TouchSampleIgnore;
        InterruptDone(InterruptType);
 }

}
```

DdsiTouchPanelPowerHandler—Calls DisableSPP() upon power down and InitSSP upon power up.

```
void DdsiTouchPanelPowerHandler(BOOL bOff)
{
 unsigned __int16 iData           = 0;
 unsigned __int16 iTempData = 0;

 // Set flag so we know to avoid system calls
    bInPowerHandler = TRUE;

    if (bOff)
 {
        // Bg 1/10/02
        // Disable SSP before going to sleep
        DisableSSP();

         TouchPanelPowerOff();

    } else
 {

        // when waking up, re-initilize both SSP and the TSC2200 itself.
        // BG20FEB02 - setup the SSP system and reset the part
        InitSSP();

    }

 // reset the flag
    bInPowerHandler = FALSE;

}
```

Keypad Interface

These functions have been modified from the standard Microsoft sample PS2 keyboard driver.

KeybdPdd_InitializeDriverEx—Initializes SPP, resets, and configures the TSC2200.

```
BOOL WINAPI KeybdPdd_InitializeDriverEx(PFN_KEYBD_EVENT_CALLBACK_EX
 pfnKeybdEventCallbackEx)
{
 // Init SSP operations
 InitSSP();

 // Bg 20FEB02 - since the keyboard driver loads first, we'll reset and setup the part
here          TSC2200Reset();

 TSC2200WriteDACReg(0x0000);
 TSC2200WriteREFReg((REF_INT_INTERNAL | REF_DL_1_MS | REF_PDN_OFF | REF_RFV_250));
 TSC2200WriteCFGReg(CFG_SNS_96);

 // setup the KeyReg
 TSC2200WriteKeyReg(KEYREG_SETUP_VALUE);

 // and the KeyMaskReg
 TSC2200WriteKeyMaskReg(KEYMASKREG_SETUP_VALUE);

 // and finally the ADC Register
 TSC2200WriteDACReg(ADC_SETUP_VALUE);

 return TRUE;
}
```

KeybdIstLoop—Main thread for the keypad driver. Waits on KBIRQ and timer interrupts, and calls KeybdPdd_GetEventEx() to gather keypad data.

```
BOOL KeybdIstLoop(HANDLE   hevInterrupt)
{

 UINT32                    VKeyBuf[16];        //    hardcoded w/ PDD.
 UINT32                    ScanCodeBuf[16];    //    hardcoded w/ PDD.
 KEY_STATE_FLAGS     KeyStateFlagsBuf[16];     //  hardcoded w/ PDD.

 int    cKeyEvents;
 int    iKeyEventIdx;

 long AutoRepeatTimeout;

 SetThreadPriority(GetCurrentThread(), THREAD_PRIORITY_HIGHEST);

 AutoRepeatTimeout = INFINITE;

 while (1)
 {
       // Bg14FEB02 - wait for event either GPIO interrupt or timeout
       WaitForSingleObject(hevInterrupt, AutoRepeatTimeout);

       // since the interrupt handler eats the up transitions, this can only be a
keydown
       cKeyEvents = KeybdPdd_GetEventEx(VKeyBuf, ScanCodeBuf, KeyStateFlagsBuf);

       if (cKeyEvents > 0)
       {
              // set the timer, so we can tell if the key goes up.
              AutoRepeatTimeout = 25;
```

```
        } else
        {
                // if now KEY event, reset for next key down
                AutoRepeatTimeout = INFINITE;


        }


        // send the events to the system callback function
        for ( iKeyEventIdx = 0; iKeyEventIdx < cKeyEvents; iKeyEventIdx++
                              (*v_pfnKeybdEventCallbackEx)(VKeyBuf[iKeyEventIdx],
                              ScanCodeBuf[iKeyEventIdx], KeyStateFlagsBuf[iKeyEventIdx]);


        //  Ack the interrupt.
        InterruptDone(SYSINTR_KEYBOARD);


 }


 // should never get here
 ERRORMSG(1, (TEXT("Keyboard driver thread terminating.\r\n")));


 return 0;
}
```

KeybdPdd_GetEventEx—Reads the KPData register, interprets the keys that are down, and returns the appropriate virtual keys and scan codes from static array in module.

```
int WINAPI KeybdPdd_GetEventEx(UINT32 VKeyBuf[16],UINT32
ScanCodeBuf[16],KEY_STATE_FLAGS KeyStateFlagsBuf[16])
{
 INT                        cEvents = 0;

 // Static variable so we can know if the
 // pen was down the last time we were called
 static BOOL   bPrevKeyDown = FALSE;

 unsigned __int16 iKeyData;
 BOOL bKeyDown = FALSE;

 // variable needed to check the state of the GPIO line to ensure we have a pen down
 volatile struct gpioreg *v_pGPIOReg = (volatile struct gpioreg *)GPIO_BASE_VIRTUAL;

 // KPIRQ from TSC2200 is reversed logic, active low
 bKeyDown = (v_pGPIOReg->gplr.gp01 == 0);

 iKeyData = TSC2200ReadKPDataReg();

 if (!bKeyDown && !bPrevKeyDown)
 {
        // if the pen is not down and wasn't down previously, just return
        // because we shouldn't have gotten the interrupt the in first place
        return 0;
 }

 // need to fill up the key data arrays
 // go through the data from the KEYdata register
 // and for each bit that is on, fill in the return arrays
 // from the arrVKeys and arrScanCodes arrays
```

```
 for (int i=0;i<16;i++)
 {
       if (iKeyData & (0x0001<<i))
       {
              VKeyBuf[cEvents] = arrVkeys[i];
              ScanCodeBuf[cEvents] = arrScanCodes[i];

              if (bKeyDown)
              {
                     // set the appropriate state flags for the system
                     if (bPrevKeyDown)
                            KeyStateFlagsBuf[cEvents] = KeyStatePrevDownFlag;
                     else
                            KeyStateFlagsBuf[cEvents] = KeyStateDownFlag;
              } else
              {
                     // zero for the key up flag
                     KeyStateFlagsBuf[cEvents] = 0;
              }

              cEvents++;
       }
 }

 // setup bPrevKeyDown for next time;
 bPrevKeyDown = bKeyDown;

return cEvents;

}
```

Kernel Functions

The following kernel functions were modified to use GPIO01 for the keyboard. The touch portions of these files where not changed from the ADS7846 version.

OEMInterruptHandler—The main interrupt handler for the whole system. The case was modified for GPIO01 to return SYSINTR_KEYBOARD. Additionally, since the system uses the same GPIO pin as SW2, the driver must ensure that even if some other code (that hasn't been encountered yet) resets the IRQ for rising edge, it won't impact the keypad processing. Therefore, the rising edge is ignored if it is received.

```
int OEMInterruptHandler(unsigned int ra)
{
// other system code that we didn't touch
….
 else if(my_sa1100_intreg.gpio1)
 {
                     TRACKPADRX_INT_CLR(1);     // clear the interrupt

        // we want to eat the rising_edge interrupts
        if (v_pGPIOReg->gplr.gp01 == 1)
        {      // if the GPIO pin is high, this must have been rising edge
               return SYSINTR_NOP;
        }

        TRACKPADRX_INT_MASK(0);    // mask interrupts
        return SYSINTR_KEYBOARD;

// Bg - what used to be here
//             GPIO1_INT_CLR(1);
//             return SYSINTR_NOP;

 }
….
// other system code that we didn't touch
}
```

OEMInterruptEnable—Modified the SYSINTR_KEYBOARD not to deal with the keyboard on the daughter card, but to handle the interrupts as needed. Here, the driver must mask rising edge interrupts, set falling edge interrupts, clear any currently pending interrupts, and then set the mask so that GPIO01 interrupts are enabled.

```
BOOL OEMInterruptEnable (DWORD idInt, LPVOID pvData, DWORD cbData)
{
 // other system code that we didn't touch
… … …

case SYSINTR_KEYBOARD:

// Bg 14FEB02 changes to get the KEYPAD to work
        TRACKPADRX_INT_RISING_EDGE_CLR
        TRACKPADRX_INT_FALLING_EDGE;
        TRACKPADRX_INT_CLR (1);       // initial clear to irqs
        TRACKPADRX_INT_MASK(1);    // interrupts will be active

   //Bg 14FEB02 - removing to get the TSC2200 keypad to work correctly
      //DAUGHTER_CARD_INT_MASK (1);   // enable SA1101CB interrupts
      break;
… … …

}
```

OEMInterruptDisable—Modified the SYSINTR_KEYBOARD case to only deal with falling edge interrupts. Here, the driver clears the interrupt and then sets the mask register to disable GPIO01 interrupts.

```
void OEMInterruptDisable(DWORD idInt)
{
// other system code that we didn't touch
… … …
   case SYSINTR_KEYBOARD:

        //Bg 14FEB02 clearing the falling edge interrupts
        TRACKPADRX_INT_FALLING_EDGE_CLR;
        TRACKPADRX_INT_MASK(0);    // interrupts will not be active
        break;
… … …
// other system code that we didn't touch

}
```

OEMInterruptDone—Modified the SYSINTR_KEYBOARD case to only deal with falling edge interrupts.

```
void OEMInterruptDone(DWORD idInt)
{
// other system code that we didn't touch
… … …
   case SYSINTR_KEYBOARD:
        //Bg 14FEB02
        TRACKPADRX_INT_FALLING_EDGE;
     TRACKPADRX_INT_MASK (1);
        break;
… … …
// other system code that we didn't touch

}
```

References

1. *"Writing Device Drivers for Microsoft Windows CE 3.0", Microsoft*
2. *TSC2200 Datasheet, Texas Instruments (SBAS191A)*
3. *"Using the ADS7846 Touch Screen Controller with the Intel SA-1110 StrongArm Processor", Texas Instruments, (SBAA070)*
4. *StrongARM SA-1110 Microprocessor - Advanced Developer's Manual, Intel*
5. *StrongARM SA-1110 Microprocessor Development Board – Schematics, Intel*
6. *StrongARM SA-1110 Microprocessor Development Board - User's Guide, Intel*
7. *StrongARM SA-1110 Microprocessor Development Platform - Windows CE Software User's Guide, Intel*
8. *Related documentation for Windows CE Platform Builder 3.0.*

Many of these documents as well as other related documentation can be downloaded from their respective websites, or found on manufacturer's provided CDs.

http://www.ti.com

http://developer.intel.com/design/strong/

http://www.microsoft.com/windows/embedded/ce/tools/default.asp

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its hardware products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by government requirements, testing of all parameters of each product is not necessarily performed.

TI assumes no liability for applications assistance or customer product design. Customers are responsible for their products and applications using TI components. To minimize the risks associated with customer products and applications, customers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any TI patent right, copyright, mask work right, or other TI intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information published by TI regarding third–party products or services does not constitute a license from TI to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. Reproduction of this information with alteration is an unfair and deceptive business practice. TI is not responsible or liable for such altered documentation.

Resale of TI products or services with statements different from or beyond the parameters stated by TI for that product or service voids all express and any implied warranties for the associated TI product or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

Mailing Address:

Texas Instruments
Post Office Box 655303
Dallas, Texas 75265