

AR1000 Windows CE Documentation

Document #:	AD-100013-001
Title:	AR1000 Windows CE Documentation
Subtitle:	
Date:	12-April-2010
Description:	This document describes the reference AR1000 driver software developed for Microsoft Windows CE. Two software bundles are available, I2C/SPI and the other for UART. Both are covered below.

Revision History

Version	Date	By	Description
001	04/12/10	SG	• Initial Draft

Table of Contents

1.0	INTRODUCTION	4
2.0	INSTALLATION OF AR1000 COMMUNICATIONS DRIVER.....	4
3.0	AR1020 (I2C, SPI) COMMUNICATIONS DRIVER.....	4
3.1	INCLUDED FILES DESCRIPTION	4
3.2	SOURCE CODE CONFIGURATION.....	5
3.3	HARDWARE CONNECTION	5
3.3.1	Power	5
3.3.2	SPI Interface	5
3.3.3	I2C Interface	6
3.4	INITIALIZATION	6
3.4.1	SPI	6
3.4.2	I2C	6
3.5	DATA READY INTERRUPT.....	7
3.5.1	Initializing GPIO of touch interrupt pin.....	7
3.5.2	Monitoring touch interrupt pin in driver.....	7
3.6	COMMUNICATION.....	8
3.6.1	SPI Byte Reception/Transmission	8
3.6.2	I2C Byte Reception/Transmission	9
3.6.3	Bit Rate.....	9
3.6.4	Inter-byte delay.....	9
4.0	AR1010 (UART) COMMUNICATIONS DRIVER.....	9
4.1	INCLUDED FILES DESCRIPTION	9
4.2	SOURCE CODE CONFIGURATION.....	10
4.3	HARDWARE CONNECTION	10
4.4	INITIALIZATION	10
4.5	COMMUNICATION.....	11
4.5.1	Baud Rate	11
5.0	CALIBRATION	11
6.0	APPENDIX	12
6.1	ADDITIONAL INFORMATION ON WINDOWS CE NATIVE TOUCH DRIVERS	12
6.1.1	Structure of a Native Touch Driver	12
6.1.2	Windows CE Calibration Algorithm Overview	13
6.1.3	Right-Click Emulation.....	13
6.1.4	Building an image with the Microchip Touch Component.....	14

1.0 Introduction

The Microchip AR10x0 is a highly developed and cost effective analog resistive touch screen controller.

This document is a guide to using the provided AR10x0 communications driver example code for SPI, I2C and UART interfacing with a Windows CE target system. This example code should also work well for Platform Builder 4.2 and 5.0, however this documentation is mostly geared toward Platform Builder 6.0.

2.0 Installation of AR1000 Communications driver

In order to have the AR1020 or AR1010 driver included on a target CE device, the driver files will first need to be extracted to the appropriate source code directory at the Windows CE development machine as follows:

1. Browse to the "<WINCEROOT>\PLATFORM\<PlatformDirName>\SRC\DRIVERS" directory, where <WINCEROOT> is the root of Platform Builder (often "[C:\WINCE600](#)") and <PlatformDirName> is the directory name of the target platform's board support package source tree.
2. If a "TOUCH" directory already exists, remove the current contents of this directory and replace it with the contents of the source package. If this directory doesn't exist, modify the "dirs" file in the "DRIVERS" directory such that the line " touch \\" is at the bottom.
3. Verify that the AR1000 Communications drivers is correctly seen by Platform Builder 6.0. To accomplish this, browse to location of Microchip touch driver files and the Microchip touch files can be found in the "Solution Explorer" tab under "<ProjectName>\<WINCEROOT>\PLATFORM\<PlatformName>\SRC\drivers\touch".

3.0 AR1020 (I2C, SPI) Communications Driver

3.1 Included Files Description

Platform.cpp - This source file implements the platform specific code to enable communication with the AR1000.

Platform.h - This file is used to define the communication API to the AR1000.

touch.cpp - This file implements the interface that Windows CE expects to exist in order for a native touch driver to function properly. Nothing should need to be modified in this file for UART, SPI and I2C interfaces

touch.h - This file is used to define the communications interface to use

sources - Platform builder project file

makefile - Platform builder project file

3.2 Source Code Configuration

1. Modify the “touch.h” file as follows:

For SPI communication:

```
#define COMMINTERFACE_CURRENT (COMMINTERFACE_SPI)
```

For I2C communication:

```
#define COMMINTERFACE_CURRENT (COMMINTERFACE_I2C)
```

2. The “Platform.cpp” file will need to be modified such that the lines marked as “Platform specific” will need to match the communication function interface of the target's board support package. Please note that an function interface to the I2C or SPI communications lines and GPIO interrupt lines needs to be available for the target CE device in order to be able to make the necessary modifications. Usually this function interface is available with the target device's board support package.
3. Verify with the target devices users manual for source files and parameter files that may need to be modified to enable all the communications lines that will be used with the AR1000 device. Usually making the appropriate modifications on the "platform.reg" file or a "#define" definition enables the desired functionality if it is not already on by default.

3.3 Hardware Connection

A free GPIO line will need to be found that can be setup as an input for the purposes of an interrupt from the touch controller. As a word of caution, on many CE board support packages only GPIO pins found on GPIO bank 0 can be used as inputs. Most GPIO pins on board support packages are set as output only. The interrupt line of the AR1000 should be connected to an input GPIO pin on the host device.

3.3.1 Power

Connect the AR1020 VDD to a VDD power source with the support range on the target device.
Connect the AR1020 VSS to a ground on the target device.

3.3.2 SPI Interface

Connect the AR1020 SCK to a SPI clock pin of the target device.
Connect the AR1020 SDI to a SPI data out pin of the target device.
Connect the AR1020 SDO to a SPI data in pin of the target device.
Connect the AR1020 SIQ to a GPIO input pin data of the target device.
Connect the AR1020 SS to an SPI channel pin of the target device. Alternatively, this AR1020 SS pin may be connected to the the host VDD if the AR1020 is the only device on an SPI port.

3.3.3 I2C Interface

Connect the AR1020 SCL to a I2C clock pin of the target device.
Connect the AR1020 SDA to a I2C data pin of the target device.
Connect the AR1020 SDO (Irq) to a GPIO input pin of the target device.

Please see the Microchip AR1000 documentation for more specifics on connection information.

3.4 Initialization

3.4.1 SPI

As an example of SPI initialization, we have initialized the SPI channel as follows:

```
config = MCSPI_PHA_EVEN_EDGES |  
        MCSPI_POL_ACTIVEHIGH |  
        MCSPI_CHCONF_CLKD(11) |  
        MCSPI_CSPOLARITY_ACTIVELOW |  
        MCSPI_CHCONF_WL(8) |  
        MCSPI_CHCONF_TRM_TXRX |  
        MCSPI_CHCONF_DMAW_DISABLE |  
        MCSPI_CHCONF_DMAR_DISABLE |  
        MCSPI_CHCONF_DPE0;
```

This configuration has the following meaning:

SPI operates in slave mode with an idle low SCK and data transmitted on the SCK falling edge.

3.4.2 I2C

As an example of I2C initialization, we have initialized the I2C channel as follows:

```
//initialization  
I2CSetSlaveAddress(AR1000TouchDevice.hI2C, 0x4d);  
I2CSetSubAddressMode(AR1000TouchDevice.hI2C, 0);  
// set at 100 khz  
I2CSetBaudIndex(AR1000TouchDevice.hI2C, SLOWSPEED_MODE);
```

This configuration has the following meaning:

Communicate with device over channel using a slave address of 0x4d with a data rate of 100 khz. The slave address value will be used when sending and receiving data for the first clocked byte. If writing data, 0x9b will be the first byte (the 0x4d 7 bit address plus a one bit). If reading data, 0x9a will be the first byte (the 0x4d 7 bit address plus a zero bit).

The sub address mode is sometimes used to allow different interfaces to a device by adding an offset value to the slave address. The Microchip touch controller does not require any such address mode so we specify a zero offset.

3.5 Data Ready Interrupt

3.5.1 Initializing GPIO of touch interrupt pin

The GPIO line the Microchip AR1000 IRQ pin (for SPI or I2C) is connected to needs to be configured as an input. During initialization of the board support, the processor is setup accordingly depending on how pins are connected to the motherboard. The appropriate pin usually cannot be hooked by Window CE as an interrupt by the touch driver unless the target hardware is setup such that that CPU recognizes the hardware line as a GPIO line. The specifics for setting this up varied between various CPUs and board support packages. For both I2C and SPI, we want to setup the GPIO such that we recognize the line as asserted when we see a transition from low to high.

```
// open and initialize GPIO line so we may use it as an interrupt
TouchDevice.hGPIO = GPIOOpen();
if (TouchDevice.hGPIO == NULL)
{
    DEBUGMSG(1, (TEXT("ERROR: Init: Failed open GPIO device driver\r\n")));
    return FALSE;
}

// Setup for input mode, falling edge detect, debounce enable
GPIOSetMode(TouchDevice.hGPIO, TouchDevice.GPIOIDForTouchInterrupt,
GPIO_DIR_INPUT|GPIO_INT_LOW_HIGH| GPIO_DEBOUNCE_ENABLE);
// Get the IRQ ID from the GPIO ID
*receivedIRQIDToHook = GPIOGetIrq(TouchDevice.hGPIO, TouchDevice.GPIOIDForTouchInterrupt);

// Set debounce time on GPIO
debounce.gpioId = TouchDevice.GPIOIDForTouchInterrupt;
debounce.debounceTime = 10;
GPIOIoControl(TouchDevice.hGPIO, IOCTL_GPIO_SET_DEBOUNCE_TIME, (UCHAR*)&debounce,
sizeof(debounce), NULL, 0, NULL, NULL);
```

3.5.2 Monitoring touch interrupt pin in driver

After we have established communication on the GPIO line, we need to hook it such that the touch driver is notified every time the AR1000 interrupt is asserted. The following is the CE function call that is used in the DdsiTouchPanelEnable() function to accomplish this:

```
if (!KernellIoControl(
    IOCTL_HAL_REQUEST_SYSINTR,
    &(receivedIRQIDToHook),
    sizeof(receivedIRQIDToHook),
    &gIntrTouchOAL,
    sizeof(gIntrTouch),
    NULL
))
{
    DEBUGMSG(
        1,
        (TEXT("ERROR: TOUCH: Failed to request the touch sysintr.\r\n"))
    );

    gIntrTouch = SYSINTR_UNDEFINED;
    goto cleanup;
}
```

The KernellIoControl() function call above causes the OS to call DdsiTouchPanelGetPoint() when touch packet ready. The ID WinCE and the Microchip AR1000 driver uses to identify the interrupt ID is stored in the “gIntrTouch” variable. The hardware ID as seen by the CPU needs to be set in the “receivedIRQIDToHook” variable prior to the KernellIoControl() function call. The “gIntrTouch” variable is what’s known as a SYSINTR variable which WinCE uses as a logical identifier for the hardware interrupt ID.

The actual monitoring of this interrupt happens in the WinCE source file “tchmain.c” which will examine the value of “gIntrTouch” we setup from this call and enables monitoring of this interrupt via it’s call to the WinCE function InterruptInitialize(). One of the parameters for InterruptInitialize() is an event which will be set every time an interrupt occurs.

3.6 Communication

3.6.1 SPI Byte Reception/Transmission

For sending and receiving bytes over SPI, the function interface under Windows CE is board support package specific. The following functions in “Platform.cpp” will need to be modified to match the API of the target CE platform’s board support package API:

```
BOOL SendCommandSPI(UINT8* sendBytes, UINT8 numBytesToSend);
BOOL ReceiveResponseSPI(UINT8* responseBytes, UINT8* numBytesToReceive, DWORD
timeoutInMilliseconds);
```

Note: The SendCommandSPI() function only needs to be implemented if command initialization during the CE boot sequence is desired.

3.6.2 I2C Byte Reception/Transmission

For sending and receiving bytes over I2C, the function interface under Windows CE is board support package specific. The following functions in "Platform.cpp" will need to be modified to match the API of the target CE platform's board support package API:

```
BOOL SendCommandI2C(UINT8* sendBytes,UINT8 numBytesToSend);  
BOOL ReceiveResponseI2C(UINT8* responseBytes, UINT8* numBytesToReceive, DWORD  
timeoutInMilliseconds);
```

Note: The SendCommandI2C() function only needs to be implemented if command initialization during the CE boot sequence is desired.

3.6.3 Bit Rate

The AR1020 can support an SPI bit rate of up to ~900KHz and an I2C bit rate of up to ~400KHz. For verifying basic communication, we recommended starting with an SPI bit rate of ~39KHz or a I2C bit rate of 100 KHz.

3.6.4 Inter-byte delay

The AR1020 requires at least a 50us inter-byte delay time for both SPI and I2C. This means you must wait at least 50us between each byte of a data packet that you are clocking. In the example code, a short delay between bytes for SPI is achieved simply by sending and receiving one byte at a time.

For I2C, it is usually not necessary to insert a delay between bytes when receiving data since the AR1020 has a clock stretching mechanism built-in. However, when writing bytes, it may be necessary to insert a short delay (300us recommended) between every I2C byte written. When writing bytes using I2C, the first byte written after the start byte should be a zero byte.

If the minimum inter-byte delay is not present, the data may become out of sync and appear to the driver as random data byte since the controller may miss a host clock while processing a byte.

4.0 AR1010 (UART) Communications Driver

4.1 Included Files Description

MCUART.CPP – This source file implements the UART specific code to enable communication with the AR1000.

MCUART.h - This file is used to define the UART communication API to the AR1000.

touch.cpp - This file implements the interface that Windows CE expects to exist in order for a native touch driver to function properly.

sources – Platform builder project file

makefile - Platform builder project file

4.2 Source Code Configuration

1. The following line in the "MCUART.cpp" file will need to be modified to match the target UART port of the embedded system:

```
#define UART_PORT L"COM2:"
```

2. Verify with the target devices users manual for source files and parameter files that may need to be modified to enable all the communications lines that will be used with the AR1000 device. Usually making the appropriate modifications on the "platform.reg" file or a "#define" definition enables the desired functionality if it is not already on by default. Also, sometimes the configuration of the target device's debug port may effect whether or not communication is enabled with the driver.

4.3 Hardware Connection

Connect the AR1020 TX to a RX pin of the target device.

Connect the AR1020 RX to a TX data out pin of the target device.

Connect the AR1020 VDD to a VDD power source with the support range on the target device.

Connect the AR1020 VSS to a ground on the target device.

4.4 Initialization

During UART initialization, UART is setup such that a Windows CE UART event occur after any data is received from the AR1010. Also, a thread is created such that UART data is monitored for packets (WaitForUARTData()). Once a packet is retrieved, the thread calls a GWES callback function directly to move the cursor. Please see the Init() function of the "MCUART.cpp" source file for a specific example of UART initialization.

For some board support packages, a source file or a platform.reg entry may need to be configured such that the CPU outputs to the targets device's receive buffer after one byte of data is received. Usually the board support package is already set up to trigger and output to the receive buffer after one byte. For example, if the target board support package has a trigger to output to the receive buffer after 32 bytes, mouse movement will only occur after the buffer has been filled and there will be an appearance of frequently missed pen-ups (since the pen-up packet may still be in a CPU cache during processing).

No additional modifications on the Microchip touch driver are necessary to support proper UART initialization.

4.5 Communication

When processing commands on the AR1010, the packet monitoring thread `WaitForUARTData()` will need to be suspended temporarily so that we may send and process commands without interference from the thread. The functions `ReceiveResponseUART()` and `SendCommandUART()` may be used for command communication.

No additional modifications on the Microchip touch driver are necessary to support proper UART data communication.

4.5.1 Baud Rate

The UART interface needs to be setup with a baud rate of 9600 bps.

5.0 Calibration

To calibrate the AR1000 using the Windows CE operating system, select "Start->Control Panel->Stylus" and then select "Recalibration". A Windows CE calibration sequence will now begin. Internally, this control panel calls the function "`TouchCalibrate()`". This function may be called from any windows application. For example, the following lines of code will trigger a Windows CE calibration:

```
#include "windows.h"

int WINAPI WinMain(HINSTANCE hInstance,
    HINSTANCE hPrevInstance,
    LPWSTR lpCmdLine,
    int nCmdShow)
{
    TouchCalibrate();
}
```

After calibration, the calibration data will be stored in the registry and written to persistent storage if hive-base storage is setup on the target Windows CE device.

To maximize the accuracy of the windows CE algorithm, it is recommended that calibration data be cleared from EEPROM (the calibration data is cleared by default and is usually not necessary to clear). This will ensure the AR1000 is in a raw mode.

6.0 APPENDIX

6.1 Additional Information on Windows CE Native Touch drivers

6.1.1 Structure of a Native Touch Driver

A Windows CE touch driver is what's considered to be a monolithic driver. This type of driver combines a platform dependent driver (PDD) and a model device driver (MDD). The Microchip provided example code is considered to be at the PDD layer while the WinCE common touch component is considered to be at the MDD layer.

Windows CE required the every native touch driver that the following specific functions are defined:
The most common way to implement a touch driver is to implement the following PDD layer functions:

```
DdsiTouchPanelGetDeviceCaps()  
DdsiTouchPanelSetMode()  
DdsiTouchPanelEnable()  
DdsiTouchPanelDisable()  
DdsiTouchPanelAttach()  
DdsiTouchPanelDetach()  
DdsiTouchPanelGetPoint()  
DdsiTouchPanelPowerHandler()
```

The above function are indirectly called as a result of a GWES call into the touch DLL driver using the following exported interface implemented in the MDD layer WinCE touch source file "tchmain.c":

```
TouchPanelGetDeviceCaps();  
TouchPanelSetMode();  
TouchPanelEnable();  
TouchPanelDisable();  
TouchPanelReadCalibrationPoint();  
TouchPanelReadCalibrationAbort();  
TouchPanelSetCalibration();  
TouchPanelCalibrateAPoint();  
TouchPanelPowerHandler();
```

Also, the touch PDD code is expected to have the following CE global variables be defined:

```
DWORD gIntrTouchChanged = SYSINTR_NOP; // Not used here.  
DWORD gIntrTouch = SYSINTR_NOP; // Interrupt to be asserted from touch  
extern "C" DWORD gdwTouchIstTimeout; // MDD thread wait timeout.  
extern "C" const int MIN_CAL_COUNT = 20;
```

- Nothing needs to be done with the "gIntrTouchChanged" variable.
- The "gIntrTouch" variable is used when we hook an interrupt. We default this to "SYSINTR_NOP" until the interrupt is hooked.
- The "gdwTouchIstTimeout" variable defines how long we wait until the next touch packet. We will set this to "INFINITE" after every touch packet processed.

- The "MIN_CAL_COUNT" defines how many samples (in the AR1000's case packets) until GWE has verified we have a good calibration point and can go on to the next calibration point.

6.1.2 Windows CE Calibration Algorithm Overview

Internally within GWES this function works as follows:

1. Call TouchPanelEnable() to start the screen sampling. Call TouchPanelEnable() to start the screen sampling.
2. Call TouchPanelGetDeviceCaps() to request the number of sampling points. Call TouchPanelGetDeviceCaps() to request the number of sampling points.
3. For every calibration point, perform the following steps: For every calibration point, perform the following steps:
 - a. Call TouchPanelGetDeviceCaps() to get a calibration coordinate. Call TouchPanelGetDeviceCaps() to get a calibration coordinate.
 - b. Draw a crosshair at the returned coordinate. Draw a crosshair at the returned coordinate.
 - c. Call TouchPanelReadCalibrationPoint() to get calibration data. Call TouchPanelReadCalibrationPoint() to get calibration data.
4. Call TouchPanelSetCalibration() to calculate the calibration coefficients. Call TouchPanelSetCalibration() to calculate the calibration coefficients.

6.1.3 Right-Click Emulation

Windows CE has a shell extension called "AYGSHELL" that can be used to simulate right-clicks if the same area of the touch screen is touched for about a second. This component can be found in the Platform Builder 6.0 catalog under "Core OS\CEBASE\Shell and User Interface\Shell". With this component included in the image, if the touch screen is touched and held in the same region of the screen, black dots will appear

and a right click will occur in the same area that is being touch.



6.1.4 Building an image with the Microchip Touch Component

To build all files and generate a Windows CE image in Platform Builder 6.0, this is done by selecting “Build->Build Solution” from the menu. To save time in testing touch source modifications, the following procedure may be followed within a Platform Builder 6.0 project after the Visual Studio solution has already been built at least once:

- 1.** Verify in the menu under “Build->Targeted Build Settings” that “Make Run-Time Image After Building” is checked.
- 2.** Select the “Solution Explorer” tab and browse to the Microchip touch files under “<ProjectName>\<WINCEROOT>\PLATFORM\<PlatformName>\SRC\drivers\touch”.
- 3.** Right-click on the “touch” project and select “Rebuild”.

The Microchip touch files will be rebuilt and a new “nk.bin” Windows CE image file will be created under “<WINCEROOT>\OSDesigns\<Project Name>\<Project Name>\<Project Name>_<Target CPU Platform>_<Configuration Target Type>.” To use this image file, this file will need to be copied to a storage medium that has been configured for use with the target device.