AR1020/AR1021 SPI/I^{2}C Open Source Linux Driver Documentation

Document #: AD-110029-002
Title: **AR1020/AR1021 SPI/I^{2}C Open Source Linux Driver Documentation**
Subtitle:
Date: 27-April-2012
Description: This document describes the reference AR1020/AR1021 SPI-I^{2}C driver software developed for Linux.

Revision History

Version	Date	By	Description
001	01/03/2011	SG	• Initial Revision
002	04/27/2012	AR	• Updated for AR1011/AR1021 release.

Table of Contents

1.0 Introduction

The goal of the documentation presented is to provide instructions on how to quickly and easily setup AR1020/AR1021 SPI or AR1020/AR1021 I^2C controller communications and calibrate touch controller drivers for most embedded Linux configurations. After this setup and calibration, applications running on target machine will be aware of the position and touch state of the touch screen connected to the touch controller. We have provided driver source code, discussed some common open source touch libraries, and have provided an open source Android calibration application to aide in enabling the driver to function on any platform the provided source code is compiled for.

A kernel source tree installation script ("update_kernel_tree.sh"), a controller test utility ("inputverify"), and source code for our Android calibration utility are provided tools that were created to help simplify the task of setting up a touch controller driver for the target computer. These tools and the libraries discussed in this documentation are only recommendations and there are a variety of many good open source solutions available. However, at a minimum the Microchip kernel driver will need to be compiled within the Linux kernel source code tree.

After all of the documentation steps have been completed for the intended target platform, a kernel device path will appear under "/dev/input" that will send touch state information (in evdev format) to any application or touch library that reads this location. Normally, a library will be configured to read from this location and the application will process the touch state information in the form of mouse events interpreted from the GUI framework the application is using.

2.0 Installation of the driver

2.1 Preliminary steps

Please complete the following instructions before setting up the AR1020/AR1021 installation:

- Setup a compiler or cross-compiler environment on the host which will be used for compiling applications and the kernel.
- The kernel source for the target system will need to be available on the host.
- Find available pins on the target system for AR1020/AR1021's SPI or I^2C bus lines and the interrupt line to be connected to.
- The platform specific file path within the kernel tree responsible for setting up host communication with SPI and/or I^2C busses, and associating IRQ numbers with devices will need to be determined. For example, on the Samsung 6410 board platform, this file is located at "arch/arm/mach-s3c6410/mach-smdk6410.c".

2.2 I^2C interface specific steps

2.2.1 Connecting the controller to the embedded board

Often it is easier testing the kernel with the embedded target board using the AR1000 Development Kit board before integrating the chip with the board. To accomplish this, please connect the lines as follows:

Connect the AR1020/AR1021 IRQ line to an available GPIO line
Connect the AR1020/AR1021 +5V line to VDD power
Connect the AR1020/AR1021 GND line to the board ground
Connect the AR1020/AR1021 SDA line to the I^{2}C bus's data line
Connect the AR1020/AR1021 SCL line to the I^{2}C bus's clock line

When connecting the AR1020/AR1021 IRQ line to the embedded target, we ideally want to choose a GPIO line that is configured as an input by default. When looking on the board or CPU schematic, available lines marked as EINT (external interrupt) usually work well with the operating system as a touch interrupt.

2.2.2 Setting up the Linux kernel source

In order for the included installation script and utilities to function, the provided source code will need to be compiled. To setup the environment for compiling source code for the target system, please see the relevant online Linux documentation on how to best accomplish this.

2.2.2.1 Automatic configuration

Extract installation files and run a command such as "tar zxvf AR1020-LINUX-SPI-I2C-V102.tar.gz". After the files are extracted, run the command **"sh ./update_kernel_tree.sh"** at the root of the extracted files. The script will prompt for the root of the kernel source tree intended for the target system. After entering this path the appropriate files in the kernel source will automatically be updated such that new drivers will appear in the kernel configuration.

2.2.2.2 Manual configuration

Note: The steps in this section may usually be skipped since the "update_kernel_tree.sh" script automates this process. However, to verify the script kernel source update or to manually configure the setup of the kernel, these steps may be useful as a reference.

Copy the "ar1020-i2c.c" file to the "drivers/input/touchscreen" directory.

Modify the "Kconfig" in the "drivers/input/touchscreen" such that it contains the following at the bottom of the file before the "endif" line.

```
config TOUCHSCREEN_AR1020_I2C
        tristate "Microchip AR1020 I2C touchscreen"
        depends on I2C
        help
          Say Y here if you have a Microchip AR1020 I2C Controller and
          want to enable support for the built-in touchscreen.

          To compile this driver as a module, choose M here: the
          module will be called ar1020-i2c.
```

The "Makefile" in the "drivers/input/touchscreen" needs to contain the following at the bottom of the file.

```
obj-$(CONFIG_TOUCHSCREEN_AR1020_I2C)  += ar1020-i2c.o
```

Next, to include Microchip touch support within the kernel, run "make menuconfig" for a text kernel configuration interface or "make xconfig" for a graphical configuration interface.

Browse to "Device Drivers->Input device support->Touchscreens" in the kernel tree interface and choose "Microchip AR1020 I2C touchscreen" as a built-in option.

Note: If the "Microchip AR1020 I2C touchscreen" option does not appear in the kernel menu, verify that “Device Driver->I2C support” is an included kernel configuration component (the driver will not appear otherwise).

2.2.3 Setting up the board’s platform specific settings

The platform specific files will now need to be modified so we may associate the AR1020/AR1021 I^2C driver with the appropriate I^2C bus and GPIO interrupt. Begin by searching for the keywords "struct i2c_board_info" and then go to the source code line specific to the current I^2C bus that is to be used.

For one of this structure's elements, please add the following line:

```
{ I2C_BOARD_INFO("ar1020_i2c", 0x4d),.irq=116,},
```

This line will cause the "ar1020_i2c_probe" function within the AR1020/AR1021 kernel driver to be called on system startup. The ".irq" variable's value will need to be changed to the IRQ number of the GPIO line the controller IRQ is attached to. If unsure of the IRQ value to use, we may set this value later using a kernel parameter and the following line may be used to finish registration of the AR1020/AR1021 driver with the kernel:

```
{ I2C_BOARD_INFO("ar1020_i2c", 0x4d),},
```

Please see section **“2.5.3 Verifying/probing touch IRQ ID”** for more information on determining the appropriate IRQ ID to use.

2.3 SPI interface specific steps

2.3.1 Connecting the controller to the embedded board

Often it is easier testing the kernel with the embedded target board using the AR1000 Development Kit board before integrating the chip with the board. To accomplish this, please connect the lines as follows:

Connect the AR1020/AR1021 IRQ line to an available GPIO line
Connect the AR1020/AR1021 +5V line to VDD power
Connect the AR1020/AR1021 GND line to the board ground
Connect the AR1020/AR1021 SDI line to the SPI bus's output line
Connect the AR1020/AR1021 SCK line to the SPI bus's clock line

Connect the AR1020/AR1021 SDO line to the SPI bus's input line

When connecting the AR1020/AR1021 IRQ line to the embedded target, we ideally want to choose a GPIO line that is configured as an input by default. When looking on the board or CPU schematic, available lines marked as EINT (external interrupt) usually work well with the operating system as a touch interrupt.

2.3.2 Setting up the Linux kernel source

In order for the included installation script and utilities to function, the provided source code will need to be compiled. To setup the environment for compiling source code for the target system, please see the relevant online Linux documentation on how to best accomplish this.

2.3.2.1 Automatic configuration

Extract installation files and run a command such as "tar zxvf AR1020-LINUX-SPI-I2C-V102.tar.gz". After the files are extracted, run the command **"sh ./update_kernel_tree.sh"** at the root of the extracted files. The script will prompt for the root of the kernel source tree intended for the target system. After entering this path the appropriate files in the kernel source will automatically be updated such that new drivers will appear in the kernel configuration.

2.3.2.2 Manual configuration

Note: The steps in this section may usually be skipped since the "update_kernel_tree.sh" script automates this process. However, to verify the script kernel source update or to manually configure the setup of the kernel, these steps may be useful as a reference.

The "ar1020-spi.c" file should be copied to the "drivers/input/touchscreen" directory.

Modify the "Kconfig" in the "drivers/input/touchscreen" such that it contains the following at the bottom of the file before the "endif" line.

```
config TOUCHSCREEN_AR1020_SPI
       tristate "Microchip AR1020 SPI touchscreen"
       depends on SPI_MASTER
       help
         Say Y here if you have a Microchip AR1020 SPI Controller and
         want to enable support for the built-in touchscreen.

         To compile this driver as a module, choose M here: the
         module will be called ar1020-spi.
```

The "Makefile" in the "drivers/input/touchscreen" needs to contain the following at the bottom of the file.

```
obj-$(CONFIG_TOUCHSCREEN_AR1020_SPI)  += ar1020-spi.o
```

Next, to include Microchip touch support within the kernel, run "make menuconfig" for a text kernel configuration interface or "make xconfig" for a graphical configuration interface.

Browse to "Device Drivers->Input device support->Touchscreens" in the kernel tree interface and choose "Microchip AR1020 SPI touchscreen" as a built-in option.

Note: If the "Microchip AR1020 SPI touchscreen" option does not appear in the kernel menu, verify that "Device Driver->SPI support" is an included kernel configuration component (the driver will not appear otherwise).

2.3.3 Setting up the board's platform specific settings

The platform specific files will now need to be modified so we may associate the AR1020/AR1021 SPI driver with the appropriate SPI bus and GPIO interrupt. Begin by searching for the keywords "struct spi_board_info" and then go to the source code line specific to the current SPI bus that is to be used.

In the relevant section, change the value of the ".modalias" variable to "ar1020-spi". This change will cause the "ar1020_spi_probe" function within the AR1020/AR1021 kernel driver to be called on system startup. The ".mode" variable's value will need to be changed to SPI_MODE_1 to setup the host to output to the clock line in the expected matter (there are four modes for the kernel to setup the host clock line for SPI). Finally, the ".irq" variable's value will need to be changed to the IRQ number of the GPIO line the controller IRQ is attached to. If unsure of the IRQ value to use, we may leave this at the default value and later this IRQ value may be set using a kernel parameter instead. If the chip select line is used, the ".chip_select" variable will need to be set to a value of one.

Please see section **"2.5.3 Verifying/probing touch IRQ ID"** for more information on determining the appropriate IRQ ID to use.

2.4 Updating Target System

2.4.1 Setting up debugging

When working with an embedded target, ideally there should be means to monitor debug messages from the kernel and run commands on the target. Having such a debugging connection will give us a great deal of control in testing and configuring the target system's setup.

An example of a common kernel debugging setup may be a Windows host computer running terminal software such as "DNW" or "Hyperterminal" connected between the embedded target and the host using an NULL modem cable on RS-232 ports. Other possibilities include a remote connection over Ethernet between the host and the target. Another option may be to run the "dmesg" command on the target system as needed to see the most recent kernel messages.

2.4.1.1 Increasing kernel log level using kernel parameters

The kernel log messages within the driver have various log levels based on importance. When setting up the target system, it's often useful to enable all logging to better ensure the driver can be up and running sooner if something doesn't work on the first attempt. For example, a higher log level will enable the

AR1020/AR1021 driver to be able to output touch coordinates using kernel debug messages. This can be very valuable in verifying the AR1020/AR1021 driver configuration since seeing valid touch coordinates is an indication that the driver setup is mostly complete.

The Linux kernel supports a parameter called “loglevel”. To see all debug messages within the AR1020/AR1021; we want to set this “loglevel” parameter to a value of eight. A kernel parameter is set within the boot loader. For the U-Boot boot loader, the kernel parameters are stored in an environmental variable called “bootargs”. The following is an example of the command to run within the U-Boot command shell to query the kernel parameters:

Command:

```
echo $bootargs
```

Example Result:

```
noinitrd console=ttySAC0 init=/init usbi.mtd=1 root=ubi0:rootfs
rootfstype=ubifs
```

The result the command returns will be different for your target system. The above result will be used as part of the next command.

Command:

```
saveenv bootargs noinitrd console=ttySAC0 init=/init usbi.mtd=1
root=ubi0:rootfs rootfstype=ubifs loglevel=8
saveenv
```

Example Result:

```
Saving Environment to NAND…
Erasing Nand…Writing to Nand… done
```

Replace the text in italics with your current boot arguments determined from the “echo $bootargs” command above.

If the Grub boot loader is used (on x86 systems), this parameter can be set within the Grub menu, by editing the “menu.lst” file (grub V1), or editing the “grub.cfg” (grub V2). In general, this parameter needs to be added to the line starting with the keyword “kernel” (Grub version 1.XX) or “linux” (Grub version 2.XX). For further information, please see online communication documentation on Grub (“http://www.gnu.org/software/grub”).
After a reboot of the system, the new log level will be applied.

2.4.2 Overview of updating kernel on embedded target

In order to enable the AR1020/AR1021 driver on the target system, the kernel will need to updated. After the kernel source has been modified appropriately from section 2.2.2 or section 2.3.2 above, enter the command "make" at the root of the kernel source. After several minutes, the compiler will generate a kernel image "zImage" that will now additionally contain the AR1020/AR1021 driver component.

To use the AR1020/AR1021 driver that was configured as a built-in component, the entire kernel will need to be replaced with data in the "zImage" file (for the ARM architecture, this is file is generated in the "arch/arm/boot" subdirectory). The methods to update this kernel can vary from system to system, however the boot loader (such as U-Boot) usually plays the primary role in updating the kernel.

The general procedure for updating the kernel using U-Boot is loading the kernel image from "zImage" into RAM, erasing the blocks of NAND flash the kernel will be written to, and then writing kernel image from RAM onto the erased NAND blocks. The following is a U-Boot command shell example using the "dnw" tool to load the "zImage" file to RAM location 0xc0008000 into NAND flash blocks between 40000 and 300000:

```
dnw
nand erase 40000 300000
nand write c0008000 40000 300000
nand read c0008000 40000 300000
bootm c0008000
```

2.4.3 Activating driver as kernel module

Usually the AR1020/AR1021 driver component is selected as a built-in kernel component. However, in some scenarios it may be more convenient to use the driver component as a kernel module. For example, if the driver is a module, the driver may be separately updated without the need to update the system's kernel.

In order for a kernel module to work properly, the kernel module will need use the same kernel source version as the kernel module. Also, the kernel will need to be updated at least once such that the platform changes are applied (otherwise the bus driver will not associate with the AR1020/AR1021 driver component). The command to load the I^{2}C kernel driver component is "insmod ar1020-i2c.ko", and "insmod ar1020-spi.ko" for the SPI kernel driver component.

To unload the kernel module, run the command "rmmod ar1020-i2c" for the I^{2}C kernel driver component and "rmmod ar1020-spi" for the SPI kernel driver component.

Note: If an error appears such as "rmmod: chdir(2.6.28.6): No such file or directory" after running the "rmmod command", then create the missing directory under "/lib/modules" with a command such as "mkdir /lib/modules/2.6.28.6" to correct this issue.

2.4.4 Verifying touch packets from driver

2.4.4.1 Monitoring kernel debug messages

The quickest way to verify touch packets is often by setting the kernel log level to eight as discussed in section 2.4.1.1 following by a touch. If the touch is detected successfully, output similar to the following may be seen in the kernel debug log (format below is X position, Y position, and pen state):

```
AR1020 I2C: 2932 2446 1
AR1020 I2C: 2970 2426 1
AR1020 I2C: 2993 2403 1
AR1020 I2C: 2993 2403 0
```

If there is there is no coordinates reported as a result of a touch, please see the section **"2.5 Driver diagnostic tools"** to help better determine the source of the issue.

2.4.4.2 Building inputverify application

Alternatively, it may be more convenient to verify touch coordinates using the provided application "inputverify" rather than by looking at kernel debug messages. To generate this application, we need to compile the "inputverify.c" file from within the root of the driver installation folder. For example, to generate an "inputverify" application file using the cross-compiler "arm-linux-gcc" we may enter the command "arm-linux-gcc –o inputverify inputverify.c" on the host. If Android is the target system, the –static flag will need to be set in order to ensure "inputverify" runs properly using a command such as "arm-linux-gcc –o inputverify inputverify.c -static".

2.4.4.3 Finding a device path using inputverify

A controller's device path is the path in which data may be read from the touch controller using an application. To help automatically discover the controller, the "inputverify" tool that is included with the driver may be used to discover the path of the controller. To detect the device path, run the command "./inputverify –f –p EVDEV" from the installation directory.

```
Command:
./inputverify -f -p EVDEV

Example output:
Microchip Touchscreen Controller Utility V1.00

Command-line options selected:
Find mode enabled.
Protocol set to EVDEV

Found controller at device path: /dev/input/event2
```

If there is no controller detected, please see the section **"2.5.1 Verifying complete driver configuration"** to verify a complete kernel driver configuration.

2.4.4.4 Reading packet data using inputverify

If the controller is detected, to verify touches are being correctly being seen by the kernel, **r**un "inputverify" on the target machine with the device's evdev path as a parameter. For example "./inputverify -p EVDEV -d /dev/input/event2".

```
Command:
./inputverify -d /dev/input/event2 -p EVDEV

Example output:
Microchip Touchscreen Controller Utility V1.00

Command-line options selected:
Device path set to /dev/input/event2
Protocol set to EVDEV

Decoding EVDEV packets (Press Ctrl-C to exit)
State: 2188 2522 0
State: 2188 2522 1
State: 2176 2522 1
State: 2176 2517 1
```

If there is there is no coordinates reported as a result of a touch, please see the next section **"2.5 Driver diagnostic tools"** to help better determine the source of the issue.

2.5 Driver diagnostic tools

It is not uncommon for some of the platform settings or the configuration settings to be slightly off such that it causes the driver to not be fully enabled or communications issues on the bus and/or touch IRQ line. To aide in getting the platform configuration correct, some mechanisms have been added to the driver such as kernel logging (**"2.5.1 Verifying complete driver configuration"**), data bus polling (**"2.5.2 Verifying controller bus data"**) and IRQ probing (**"2.5.3 Verifying/probing touch IRQ ID").** These three tests need to be followed in this order since Section 2.5.2 depends on 2.5.1 in order to run and likewise diagnostic test 2.5.3 depends on 2.5.2.

2.5.1 Verifying complete driver configuration

If there is no response as a result of a touch, the first thing to check is if the AR1020/AR1021 kernel driver has successfully been included within the kernel.

If the AR1020/AR1021 SPI kernel driver has been included, "ar1020_spi_init: begin" will be seen within the kernel debug output. If the AR1020/AR1021 I^{2}C kernel driver has been included, "ar1020_i2c_init:

begin" will be seen within the kernel debug output. This message will appear regardless if the kernel platform settings have been modified or not.

To verify the platform settings within the kernel have been modified appropriately to associate the SPI or I^2C bus with the AR1020/AR1021 driver, we will check to see if the driver's probe function has been called. If the AR1020/AR1021 SPI kernel driver has been registered correctly, "ar1020_spi_probe: begin" will be seen within the kernel debug output. If the AR1020/AR1021 I^2C kernel driver has been registered correctly, "ar1020_i2c_probe: begin" will be seen within the kernel debug output. This message will only appear if the board's platform settings source file has been modified correctly. If this debug message has not been seen, please verify the correct platform source file has been changed. Adding a printk() debug message within the target board's initialization function in the platform specific source file can help in determining this.

2.5.2 Verifying controller bus data

To ensure the SPI or I^2C bus has been enabled and configured correctly, it may be useful to know if we are reading data correctly.

Within the AR1020/AR1021 driver, there is a mode such that the data is constantly clocked from the controller (ignoring the state of the IRQ). To enable this mode, add the kernel parameter "ar1020-i2c.testI2Cdata=1" if using the AR1020/AR1021 I^2C driver component or add the kernel parameter "ar1020-spi.testSPIdata=1" if using the AR1020/AR1021 SPI driver component. For an example of setting up a kernel parameter, please see section **"2.4.1.1 Increase kernel log level using kernel parameters"**.

Using this mode, it can quickly be determined by touching the touch screen if good touch data is being seen by the system. A good packet will consist of a "0x80" or "0x81" value followed by four data bytes.

The following is an example of the expected output in this mode in the SPI component (the output with I^2C component is nearly identical):

```
AR1020 SPI: ar1020_spi_init: begin
AR1020 SPI: ar1020_spi_probe: begin
AR1020 SPI: In testing mode to verify packet.  To inhibit this mode,
Unset the "testSPIdata" kernel parameter
0x04
0x80 0x4a 0x0f 0x26 0x14

0x81 0x4a 0x0f 0x26 0x14

0x81 0x3d 0x0f 0x22 0x14

0x80 0x2d 0x0f 0x1d 0x14
```

If there are no bytes being seen in this mode, it's recommended to verify that the correct SPI or I^2C bus has been configured in the board's platform source file. For the AR1020/AR1021 SPI driver, if seemingly random bytes are appearing, the clock may not be set correctly (there are four different kernel modes available for the kernel clock line on SPI) or an unsupported speed value may have been specified (see AR1000 datasheet for information on the AR1020/AR1021 supported SPI speed range).

2.5.3 Verifying/probing touch IRQ ID

The external interrupt number will be different than the internal interrupt id the operating system will use. To determine this interrupt number, kernel macros/functions may be used. For example, if the external interrupt number is known to be seven, the “IRQ_EINT(7)” macro may be used which should return the correct interrupt ID to use. If the GPIO number is known to be 48, the “gpio_to_irq(48)” function may be used which should return the correct interrupt ID to use. Occasionally, there are errors in board support package and these macros do not work properly, but this should be uncommon.

If there is some trouble in acquiring the appropriate ID using macros, a probing mechanism has been added to the driver to help make this determination of the touch IRQ ID easier. To enable this mode, add the kernel parameter “ar1020-i2c.probeForIRQ=1” if using the AR1020/AR1021 I^2C driver component or add the kernel parameter “ar1020-spi.probeForIRQ=1” if using the AR1020/AR1021 SPI driver component. For an example of setting up a kernel parameter, please see section **“2.4.1.1 Increase kernel log level using kernel parameters”.**

After this feature is enabled, kernel debug message will prompt for a continuous touch before probing different IRQ IDs until the probe completes. By default, the driver will scan IRQ IDs from 0 to 200. This range may be changed by using the kernel parameters “probeMin” and “probeMax”. For example, to probe the IRQ IDs from 100 to 130, the parameters “ar1020-spi.probeMin=100 ar1020-spi.probeMax=130” should be added if using SPI component and “ar1020-i2c.probeMin=100 ar1020-i2c.probeMax=130” if using I^2C component.

The following is an example of the expected output in this mode using the SPI component (the output with I^2C component is identical) using kernel parameters “ar1020-spi.probeForIRQ=1 ar1020-spi.probeMin=110 ar1020-spi.probeMax=120”:

```
AR1020 SPI: ar1020_spi_init: begin
AR1020 SPI: ar1020_spi_probe: begin
AR1020 SPI: Probing for interrupt id.
AR1020 SPI: Please touch screen before IRQ probe for successful
detection.
AR1020 SPI: Probing will commence in five seconds.

AR1020 SPI: Kernel exception messages may appear during the
AR1020 SPI: probing process.
AR1020 SPI: Testing IRQ 110
AR1020 SPI: Testing IRQ 111
AR1020 SPI: Testing IRQ 112
AR1020 SPI: Testing IRQ 113
AR1020 SPI: Testing IRQ 114
AR1020 SPI: Testing IRQ 115
AR1020 SPI: Testing IRQ 116

AR1020 SPI: Touch IRQ detected at ID: 116.
input: AR1020 Touchscreen as /class/input/input2
```

2.5.4 Setting touch IRQ using a kernel parameter

Setting a touch IRQ value can always be done directly from the kernel source. However, if an IRQ value is set using the kernel source, the target system's Linux kernel will need to be recompiled and updated after every change. To make setting the touch IRQ value easier, the AR1020/AR1021 driver supports directly setting the touch IRQ using a kernel parameter. For example, to set the IRQ to 116, the parameter "ar1020-spi.touchIRQ=116" should be added if using SPI component and "ar1020-i2c.touchIRQ=116" if using I^2C component. If an IRQ value is set using a kernel parameter, this will override any value set within the kernel source. For an example of setting up a kernel parameter, please see section **"2.4.1.1 Increase kernel log level using kernel parameters".**

The following is example of the kernel debug output that will appear if "ar1020-i2c.touchIRQ=116" is used with the AR1020/AR1021 driver component.

```
AR1020 I2C: ar1020_i2c_probe: begin
AR1020 I2C: Using IRQ 116 set via kernel parameter.
```

Command communication

2.5.5 Controller commands from an application

Controller command communication is usually hardcoded into the kernel driver. However, in order to enable applications to access the full AR1020/AR1021 command set, a driver feature using a Linux kernel component called "sysfs" has been added so commands may be easily sent to the controller without the need for an additional library.

After the AR1020/AR1021 driver has finished loading, a directory located at "/sys/kernel/ar1020" will appear in the file system.

The variables relating to controller commands within this directory are as follows:

commandMode – The current mode of the controller. To enable command communication, this value must first be set to a value of "1". The decoding of touch packets will be inhibited until it is set back to a value of "0".
sendBuffer – This is the location that gets written to write a command to the controller. The expected value is in hexadecimal.
receiveBuffer – This value contains the response of the last sent command to the controller.
commandDataPending – After a command has been sent to the controller, this value will be "1" until a response has been processed. After a response has been processed, this value will be "0". The purpose of this value is that an application may quickly poll this "commandDataPending" value until it has a value of "1" so the application may more quickly respond after the controller has finished processing the command. Usage of this value is optional as a delay of 100 milliseconds after the command is sent is sufficient for most commands.

The following is an example of the enable command using these kernel values from the command-line:

```
Command:

cd /sys/kernel/ar1020
echo “1”>commandMode
echo “0x55 0x01 0x12”>sendBuffer
cat receiveBuffer

Example output:

0x55 0x02 0x00 0x12
```

Please note that touch reporting is disabled until we exit command mode. To exit command mode, run the following command:

```
echo “0”>commandMode
```

To send a command using a custom application, it should be noted that these kernel locations should be opened as a text file rather than binary file.

Note: the following command format is also interpreted correctly by the driver if preferred:

```
echo “55 1 12”>sendBuffer
```

2.5.5.1 Leading zero on I^{2}C controller commands

When sending commands using the I^{2}C interface, a zero is required as a first command byte. If a zero is not sent as a first byte, the controller commands will sometimes, but not always succeed.

For example, the following is an example of the recommended way of sending the enable command using the I^{2}C interface:

```
echo “0x00 0x55 0x01 0x12”>sendBuffer
```

2.5.5.2 I^{2}C inter-byte delays on host-to-controller writes

Our AR1020/AR1021 controller does not have a clock stretching capability when bytes are being written to it from the host. This means that in order for the controller to understand the command we are sending, we need to add delays where appropriate so bits are not misread by the controller. To add these delays, we will modify the I^{2}C kernel bus source that is applicable to the target board. This I^{2}C bus driver source code for the target system should be located within the “drivers/i2c/busses” subdirectory. We will need to modify the code such that there will be a 170 micro-second inter-byte delay (delay in-between bytes) and a 30 micro-second stop bit delay (delay before stop bit). The specific source file and code modifications vary from different target systems, however the below is a tested example using a Samsung 2410 I^{2}C bus controller of how the changes may be implemented.

In the file “drivers/i2c/busses/i2c-s3c2410.c” from the kernel source, after the include definitions, add the following lines:

```
#define INTERBYTEDELAY 170
#define STOPDELAY 30
```

Search for the text “ndelay(i2c->tx_setup);” and replace this line with the following lines:

```
// ndelay(i2c->tx_setup);
udelay(INTERBYTEDELAY);
```

Next search for the text “void s3c24xx_i2c_stop” and add the following lines at the beginning of this function:

```
udelay(STOPDELAY);
```

Finally, we will recompile and update the kernel in our embedded device.

3.0 Calibration Methods

Up to this point, we have been setting things up such that the system reads the controller data and decodes this into controller’s raw coordinates. Raw coordinates are coordinates that the controller calculates and communicates as a result of a touch. These raw coordinates will be accurate according to its electrical connections from the touch screen to the controller. However, to make these raw coordinates match up with the touched area of the display, a touch calibration will often be necessary.

3.1 Console calibration

3.1.1 Using built-in kernel driver’s software calibration

The AR1020/AR1021 kernel driver has a built-in min-max software calibration that supports all of the orientations the touchscreen may be mounted on the display. This section details the steps to be followed to best accomplish using the driver calibration components.

Communicating with the kernel driver

Communication with the AR1020/AR1021 is done via a kernel mechanism called sysfs. Sysfs appears as a set of directories on the file system under “/sys” that can be used to communication with all kernel components that support this interface. The directory that the AR1020/AR1021 kernel component is registered under is “/sys/kernel/ar1020”.

The variables relating to calibration within this directory are as follows:

minX –The smallest possible value on x-axis.
minY –The smallest possible value on y-axis.
maxX –The largest possible value on x-axis.
maxY –The largest possible value on y-axis.
swapAxes –Set to one if horizontal touch movement on x-axis result in movement in the y-axis and vice versa.
invertX –Set to one to cause x coordinates to appear on the opposite side of x-axis.
invertY – Set to one to cause y coordinates to appear on the opposite side of y-axis.

Setting Raw Mode

In order to fully ensure the calibration constants are valid, it is highly recommended that the kernel driver be set to raw mode. In this mode, the coordinate returned by the kernel will exactly match the coordinate returned by the controller. In order to set this mode, the calibration values will need to be set as follows:

minX - 0
minY - 0
maxX - 4095
maxY - 4095
swapAxes - 0
invertX - 0
invertY - 0

A calibration variable may be set by writing a string value to the filename. As an example of how to write the above value, from the "/sys/kernel/ar1020" directory, run the following commands:

```
echo “0” > minX
echo “0” > minY
echo “4095” > maxX
echo “4095” > maxY
echo “0” > swapAxes
echo “0” > invert
echo “0” > invertY
```

Setting these variables will cause an immediate change in the reported coordinates for every change in value.

Correct swapped or inverted axes

1. Using the "inputverify" tool, touch across the touchscreen horizontally. If the y-values change much more than the x-value, the x and y are swapped. To correct this, run the command "echo "1"> "swapAxes".

2. Using the "inputverify" tool, touch across the touchscreen horizontally. If on the left side of the display the x value is greater than the right side, then the x axis is inverted. To correct this, run the command "echo "1"> "invertX".

3. Using the “inputverify” tool, touch across the touchscreen vertically. If on the top side of the display the x value is greater than the bottom side, then the y axis is inverted. To correct this, run the command “echo “1”> “invertY”.

Determine the minimum/maximum values of touch area over display

1. Using the “inputverify” tool, touch the left side of the viewable area of the display on the touchscreen. Please write down this x value down as “minX”.

2. Using the “inputverify” tool, touch the right side of the viewable area of the display on the touchscreen. Please write down this x value down as “maxX”.

3. Using the “inputverify” tool, touch the top side of the viewable area of the display on the touchscreen. Please write down this y value down as “minY”.

4. Using the “inputverify” tool, touch the bottom side of the viewable area of the display on the touchscreen. Please write down this y value down as “maxY”.

Using the values written down above, set the “minX”, “minY”, “maxX”, “maxY” variables within the sysfs tree. For example if minX, minY, maxX, maxY were 310, 305, 3700, 3720 respectively, the commands to run would be as follows:

```
echo “310” > minX
echo “305” > minY
echo “3700” > maxX
echo “3720” > maxY
```

Verify calibration

If a GUI drawing application or environment is easily available, run the application and verify the touch is evaluated correctly under the touched location based on its numerical value.

If a GUI drawing application or environment is not easily available, using the “inputverify” tool, touch in the upper-left of the display area under the touch screen and verify the coordinates returned are near (0, 0). Next, touch in the lower-right of the display area under the touch screen and verify the coordinates returned are near (4095, 4095). If this is not the case, tweaking of the minimum and maximum values may be necessary such that upper-left and lower-right corners are near (0, 0) and (4095, 4095) respectively.

3.1.2 Using controller’s hardware calibration feature

If no software calibration library is easily available on the embedded target and there is a need for a calibration to persist after a reboot, the AR1020/AR1021’s hardware calibration can be a very convenient feature as an AR1020/AR1021’s hardware calibration stores calibration data within EEPROM.

In general, we do the following (as described in the AR1000 datasheet) for a hardware calibration:

1. Set inset by setting appropriate RAM offset (the default controller inset is 12.5 percent). For a console hardware calibration, choose zero percent.
2. Send the calibrate command.
3. If using a GUI application, draw calibration target at current inset after every controller response from a touch.

4. If using a console calibration, touch and release at the corners of the display on the touchscreen in the following order: upper-left, upper-right, lower-right, and finally lower-left.

The following is a command line example of the above:

```
echo “1”>commandMode
echo “0x55 0x05 21 0x00 0x2e 0x01 0x00”>sendBuffer
echo “0x55 0x02 0x14 0x04”>sendBuffer
echo “0”>commandMode
```

If it becomes more convenient on the embedded system to have a software calibration after previously having done a hardware calibration, it is very important to disable the hardware calibration before the software calibration. The “TouchOptions” register of the AR1020/AR1021 controller will be used to un-set the “Calibrated Coordinates Enable” bit following by running the “Registers Write to EEPROM” controller command. If an 8-wire sensor is attached a value of two will be written to the “TouchOptions” registers, otherwise value of zero will need to be written. The following example commands may be used on a 4 or a 5 wire touch setup to disable calibration:

```
echo “1”>commandMode
echo “0x55 0x05 0x21 0x00 0x2d 0x01 0x00”>sendBuffer
echo “0x55 0x01 0x23”>sendBuffer
echo “0”>commandMode
```

3.2 Calibration under the Android OS

Once the AR1020/AR1021 kernel driver has been setup and verified, the device needs to be calibrated such that coordinates returned by the kernel correspond with the touched area of the touchscreen. On devices running the Android operating system, this can be done by installing and running the provided calibration application. This calibration application uses the AR1020/AR1021 kernel driver’s software calibration and applies calibration settings both after calibration and after every reboot.

3.2.1 Installing Android driver calibration tool package

In the root of the installation folder, there is a “Calibration.apk” file that is intended to be used for installing this calibration application onto the target device. There are a couple different ways to install this installation package.

One way to install this calibration package is to use a package installer that may already exist on the target device. Using this package installer, select this calibration package file (“Calibration.apk”) from either over the network or through a removable storage medium such as MicroSD.

Another way to install this package is to use an Android utility called “adb”. This utility is run from the development host and communicates with an “adbd” daemon on the target device. Communications with a target device may be verified using the command “adb devices”. If there is good communication, the target devices will appear as a result of this command. If the device is not listed, please see the “adb” documentation on Androids web site. After communication is established, run the command “adb install calibration.apk” from the root of the installation folder on the development host.

The following is an example set of commands to setup adb on an Android embedded target (we setup on port 80 to avoid firewall issues) for a network connection:

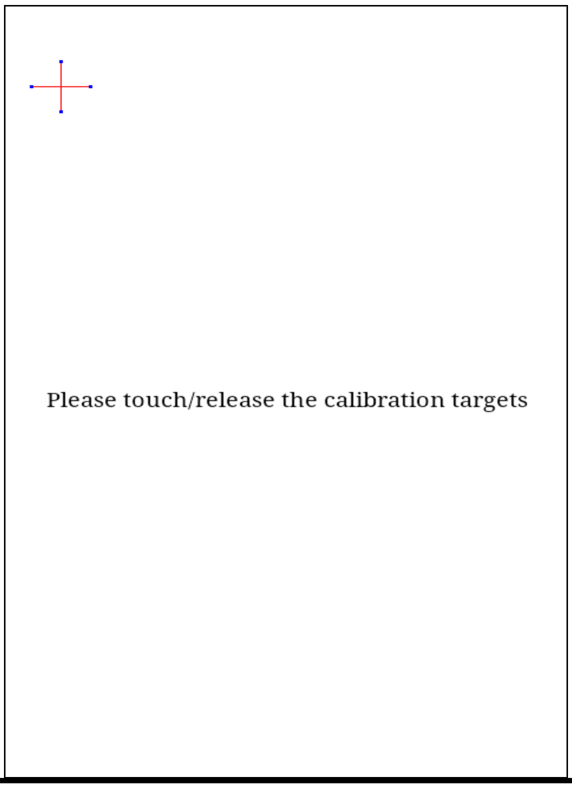
```
netcfg eth0 dhcp
setprop service.adb.tcp.port 80
stop adbd
start adbd
netcfg
```

After the last command, make note of the IP address returned. In our example, we will use IP address “192.168.0.101”.

The following is an example set of commands to setup adb and install the calibration package from the host system over a network connection:

```
adb connect 192.168.0.101:80
adb install Calibration.apk
```

Using the buttons on the target Android device, start the calibration application from the list of currently installed applications. A calibration application will appear with a target in the upper-left corner as shown below.



Touch the three calibration targets and the application will automatically exit. The touchscreen is now calibrated (calibration data is stored within Android configuration settings) and will be applied after every reboot of the device.

3.2.2 Compiling Android calibration from source

Normally, installing the provided calibration package will be sufficient in most cases. However, to make changes to the calibration software such as changing the inset or adding custom graphics, the source code will need to be recompiled.

The calibration application's source files, found within the "Calibration" subdirectory of the driver installation files, have been setup to be compiled within the Eclipse IDE tool. Before opening the calibration project, the Android SDK will need to be installed and Eclipse will need to be setup with the Android plug-in. Instructions on how to do this may be found at http://developer.android.com/sdk/index.html.

Once Eclipse and the Android SDK have been setup, with Eclipse "select File->Import..". An import dialog will appear and the option "Existing Projects into Workspace" under the "General" tree should be selected followed by clicking the "Next >" Button. Finally, for the "Select root directory" option, browse to the path of the "Calibration" directory located at the root of the installation source.

To compile the project, selection the "Calibration" package from package explorer and select "Run As->Android Application" and the "Calibration.apk" package will be built within the "Calibration\bin" subdirectory of the installation source.

3.2.3 Adjusting calibration inset

Within the "Calibration.java", located at "src\com.microchip.calibration" within project explorer, near the beginning of the "CalibrationView" class, find the following line (can simply search for "inset=10"):

```
final int inset=10;
```

This value represents the inset percent the calibration will display the targets at. For example, to have the calibration display the targets at 20 percent; change this line to:

```
final int inset=20;
```

3.3 Tslib

3.3.1 Cross-compile Tslib

By default, the "install-tslib.sh" script has been setup to use the default tools of the host computer. However, for most embedded system, the binary format on target embedded system does not match the binary format of the host system. For example, a host computer may be running a desktop Linux operating system such as Ubuntu using an x86 based binary format while the target computer may be running arm based binary files. In order to compile to a different binary format, a cross-compiler is needed.

If a cross-compiler is being used, please run the following steps before running this script:

1. Using the editor of choice, edit the file "install-tslib.sh" and uncomment (remove "#" symbols) from the following lines:

```
#       export CC=arm-linux-gcc
#       export CXX=arm-linux-g++
#       export AR=arm-linux-ar
#       export RANLIB=arm-linux-ranlib
```

2. After these lines are uncommented, follow the steps detailed in **"3.3.2 Setting up Tslib library"** to generate the binary files**.** After the binary files have been generated, it is a good idea to verify that the binaries were compiled to the target format (usually arm) instead of the host format (usually x86). To do this run the following commands on the host:

```
file /usr/local/tslib/bin/ts_test
file /usr/local/tslib/lib/ts/input.so
```

For each of these commands, if the target format is arm based, you should see output similar to the following:

```
/usr/local/tslib/bin/ts_test: ELF 32-bit LSB executable, ARM, version 1
(SYSV), dynamically linked (uses shared libs), for GNU/Linux 2.6.14,
not stripped
/usr/local/tslib/lib/ts/input.so: ELF 32-bit LSB shared object, ARM,
version 1 (SYSV), dynamically linked, not stripped
```

3.3.2 Setting up Tslib library

The Tslib library is an open source touch library mostly aimed at embedded systems. The Tslib libraries come with a calibration application ("ts_calibrate") and a test application ("ts_test") that utilize the graphics frame buffer.   The frame buffer is a device that abstracts the graphics hardware so that software may access graphics capabilities using a common interface across various hardware platforms. The frame buffer interface is very common on embedded devices.

A Tslib installation script, the Tslib version 1.0 source bundle, and patch files are provided to ensure the code functions correctly with Microchip touchscreen controllers. Before running this startup script, please ensure the "autotools" packages are already installed on host computer such as "autoconf" and "automake" that this script depends upon. If the "autotools" packages are not installed, please reference the relevant documentation of how to install these for the Linux distribution of the host computer. An example usage of this script is shown below.

```
Command:
```

```
sh ./install-tslib.sh

Example output:
Extracting files
tslib-1.0/
tslib-1.0/m4/
tslib-1.0/m4/external/
tslib-1.0/m4/external/PLACEHOLDER
tslib-1.0/m4/internal/
tslib-1.0/m4/internal/visibility.m4
tslib-1.0/plugins/
.
.
.
make[2]: Leaving directory `/home/steve/Desktop/AR1XXX-LINUX-I2C-
V102/tslib-1.0'
make[1]: Leaving directory `/home/steve/Desktop/AR1XXX-LINUX-I2C-
V102/tslib-1.0'
Installation complete.

Next environmental variables will need to be setup in order to
run 'ts_calibrate' and 'ts_test' tslib utilities.

For example if evdev path is at /dev/input/event5,the minimum
environmental variables will need to be set by using the following
commands.

export TSLIB_TSDEVICE=/dev/input/event5
export TSLIB_PLUGINDIR=/usr/local/lib/ts
```

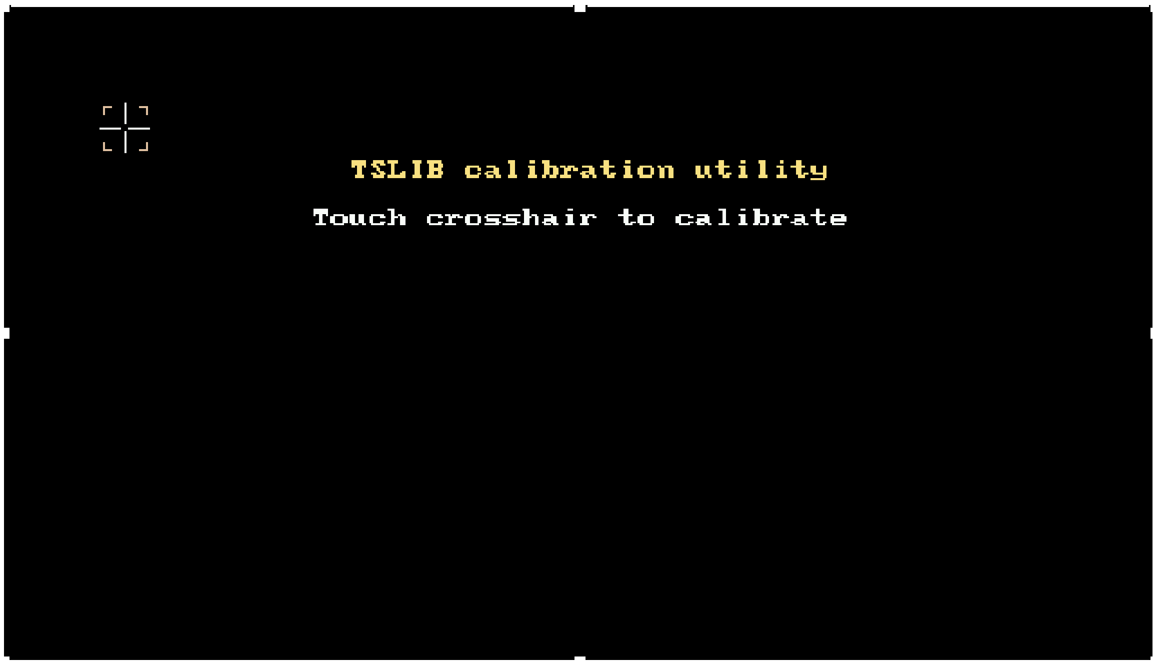
If everything is setup correctly, the "ts_calibrate" application can be used to calibrate the display and the "ts_test" application can be used to test the calibration. However, before we can use these utilities, we need to setup a couple environmental variables. At a minimum, we need to set the environmental variables "TSLIB_TSDEVICE" and "TSLIB_PLUGINDIR" for the utilities to work correctly with our Tslib configuration. On system startup, these variables should always be set before running the Tslib-based application that depends on these variables.

```
export TSLIB_TSDEVICE=/dev/input/event5
export TSLIB_PLUGINDIR=/usr/local/lib/ts
```

Other environmental values that can be set include the following:

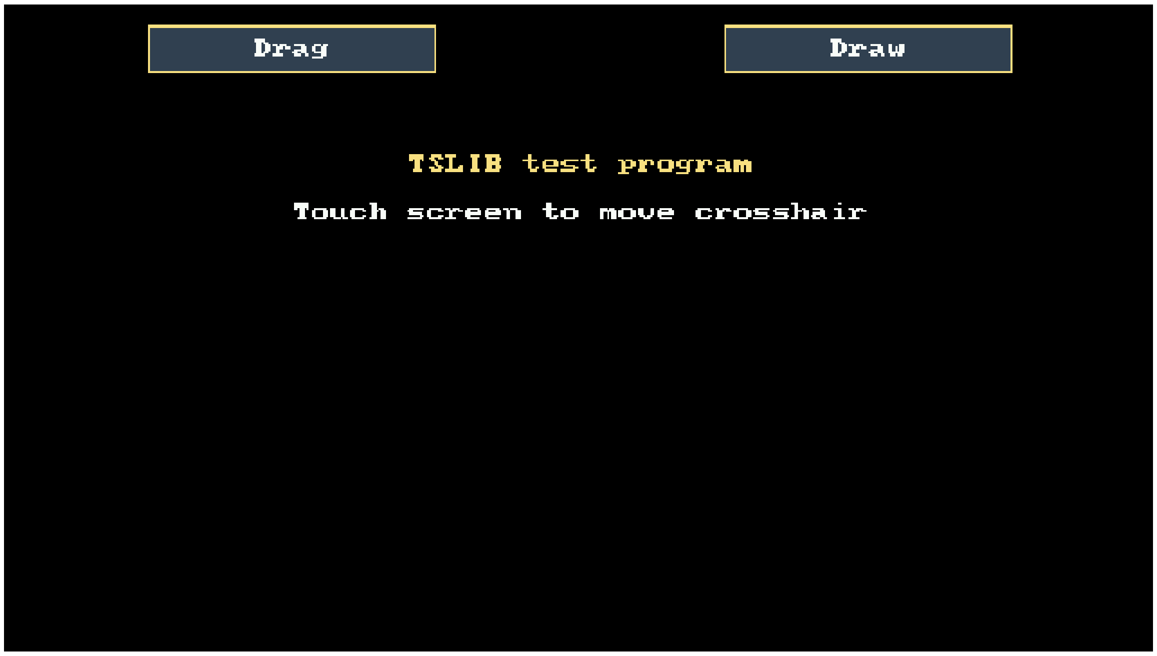
```
export TSLIB_FBDEVICE=/dev/fb0
export TSLIB_CONSOLEDEVICE=none
export TSLIB_CALIBFILE=/etc/pointercal
export TSLIB_CONFFILE=/etc/ts.conf
```

After setting the environmental values, we now can run "ts_calibrate". The images below shows what can be expected to see after running this calibration.


TSLIB calibration utility

Touch crosshair to calibrate

To test the calibration, run the "ts_test" command. The image below demonstrates the test screen.



TSLIB test program

Touch screen to move crosshair

3.4 Calibration under the Qtopia OS

If the Qtopia operating system is used, all of the Tslib steps above still apply since Qtopia uses Tslib. The only other step to do before running Qtopia is set the “QWS_MOUSE_PROTO” environmental value “TPanel:REPLACE_THIS_TEXT_WITH_EVDEV_PATH”. The “QWS_MOUSE_PROTO” variable is the one that Qtopia specifically uses. For example, if the AR1020/AR1021 device is located at “/dev/input/event2”, the command to set this variable may be as follows using the bash shell:

```
export QWS_MOUSE_PROTO=”TPanel:/dev/input/event2”
```