TrueTouch® Driver for Android (TTDA) 3.x User Guide

Doc. No. 001-90126 Rev. *D

CONFIDENTIAL - RELEASED ONLY UNDER NONDISCLOSURE AGREEMENT (NDA)

Cypress Semiconductor
198 Champion Court
San Jose, CA 95134-1709
Phone (USA): 800.858.1810
Phone (Intnl): +1.408.943.2600
http://www.cypress.com

Copyrights

Trademarks

PSoC Designer™, Programmable System-on-Chip™, and TTSP™ are trademarks and PSoC® and TrueTouch® are registered trademarks of Cypress Semiconductor Corp. All other trademarks or registered trademarks referenced herein are property of the respective corporations.

The Android Driver Source Code is free and open source and distributed according to the GNU General Public License version 2 (GPL v2). A copy of this license is included in the distribution files (COPYING.txt).

Source Code

Other Source Code (software and/or firmware) is owned by Cypress Semiconductor Corporation (Cypress) and is protected by and subject to worldwide patent protection (United States and foreign), United States copyright laws and international treaty provisions. Cypress hereby grants to licensee a personal, non-exclusive, non-transferable license to copy, use, modify, create derivative works of, and compile the Cypress Source Code and derivative works for the sole purpose of creating custom software and or firmware in support of licensee product to be used only in conjunction with a Cypress integrated circuit as specified in the applicable agreement. Any reproduction, modification, translation, compilation, or representation of this Source Code except as specified above is prohibited without the express written permission of Cypress.

Disclaimer

CYPRESS MAKES NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. Cypress reserves the right to make changes without further notice to the materials described herein. Cypress does not assume any liability arising out of the application or use of any product or circuit described herein. Cypress does not authorize its products for use as critical components in life-support systems where a malfunction or failure may reasonably be expected to result in significant injury to the user. The inclusion of Cypress' product in a life-support systems application implies that the manufacturer assumes all risk of such use and in doing so indemnifies Cypress against all charges.

Use may be limited by and subject to the applicable Cypress software license agreement.

Contents

Section A: TTDA Technical Description

This section documents the technical design and features in the TrueTouch® Driver for Android (TTDA) 3.x driver. Typically, the latest driver version that is released is recommended. However, the customer firmware may require a specific driver version (refer to the TrueTouch device release notes for compatibility requirements).

Section A contains the following chapters:

1. Overview

Cypress's TrueTouch® Driver for Android™ (TTDA) enables developers to integrate Cypress TrueTouch devices into mobile touchscreen applications. This driver targets Android implementations, but can be integrated into other Linux kernel-based products that support touchscreen input drivers also. The driver components are modular and can be integrated into products with either I^{2}C or SPI host to touchscreen device interfaces.

The code is developed under the open-source licensing terms of GPL v2.

1.1 TrueTouch Device Support

TTDA 3.x supports the following Packet Interface Protocol (PIP) TrueTouch devices:

- CYTMA545/CY8CTMA54X Base v1/v2
- CYTMA568 Base v1/v2
- CYTMA445A
- CYTMA448 Base v1/v2

TTDA 3.x provides support for the following functions:

- TrueTouch Packet Interface Protocol (PIP) host interface protocol
- Bootloader (field upgrade)
- Command and response via a standard file system interface (sysfs)
- Virtual keys
- In-system ("in-phone") manufacturing and test (TTDA 3.3 and above)

Table 1-1 Driver-Supported Features

Driver Version	Product Family	Firmware Release	Linux Kernel (Compile Versions)	Driver-Supported Features								
				Bus Model	EZ Wake	Hover	Stylus	Glove	Proximity	Device Tree	Simplified Core Driver	In-System Tests
TTDA 3.0	CY8CTMA525	TSG5_M. Base.V1	3.0 - 3.6	✓	-	-	-	-	-	-	-	-
TTDA 3.0.1	TMA568	TSG5_L. Base.V1	3.2 - 3.6	✓	-	-	✓	-	-	✓	-	-
TTDA 3.1	TMA545	TSG5 Base.V1,2	3.2 - 3.6	✓	✓	✓	✓	✓	✓	✓	-	-
TTDA 3.2	TMA448 TMA5XX	TSG5 Base.V1,2	3.2 - 3.14	-	✓	✓	✓	✓	✓	✓	✓	-
TTDA 3.3 TTDA 3.4	TMA448 TMA5XX	TSG5 Base.V1,2	3.0 - 3.15	-	✓	✓	✓	✓	✓	✓	✓	✓
TTDA 3.3 TTDA3.4	TMA445A	TSG6 Base.V1	3.0 - 3.15	-	-	-	-	✓	✓	✓	✓	✓

For TTDA 3.0.1 to TTDA 3.2 versions and for kernel versions earlier than 3.2, the driver adapter modules must be modified to remove references to the Device Tree support kernel API calls. For kernel versions earlier than than 3.0, TTDA may be limited to using only the multi-touch protocol A module; otherwise, attempts to build for multi-touch protocol B may return an unresolved symbol “input_mt_init_slots”.

For TTDA 3.0 to TTDA 3.1 versions and for kernel versions later than 3.6, the multi-touch protocol B kernel API call to “input_mt_init_slots()” needs to have a third parameter added. This parameter can either be set to 0 or the kernel-defined constant INPUT_MT_DIRECT can be used. Additionally, the use of the macro “__dev_exit_p” must be removed in the adapter modules.

Note: Latest versions of the driver have more bug fixes.

1.2 Android Touchscreen Device Driver

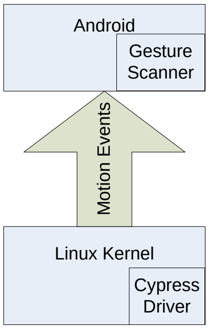
Android is a complete ecosystem designed specifically for mobile devices. This system includes a Java®-based user interface for which developers can create applications that can be installed on user products. Android uses a specialized version of the Linux operating system kernel. The Cypress Android driver is included in the product Android platform build as part of the boot image, which contains the runtime kernel. TTDA 3.x is compatible with Android 2.3 (Kernel 2.36), 4.0 (Kernel 3.0.8), and 4.1+ (Kernel 3.2+).

The Linux kernel supports several driver types. The input subsystem type has separate classifications for input devices. TTDA 3.x is classified as a “Touch Device” in the input subsystem.

A touchscreen device driver services touchscreen device touch reports, parses the touch signals, packs a Linux event with the touch signals as motion signals, and then submits the packed event to the Linux event-handling system. This event-handling system passes the event information to the Android platform layer, which passes the event to requesting applications as a Motion Event.

Android also includes a Gesture Scanner. The Motion Event information is sent to the Gesture Scanner, which can create a number of common gestures. Android applications that request gesture signals receive the gesture information from the Android Gesture Scanner.

Figure 1-1. Driver Overview



TTDA 3.x is based on the Linux bus driver and uses Linux module-based functions to provide the hardware bus and platform system interfaces. Cypress's custom user modules can subscribe for data from the core functions, allowing you to create innovative designs quickly.

Initially, the TTDA development used the Linux multi-touch protocol A signaling to support Android Gingerbread (GB 2.3) development platforms. Protocol B signaling was added to support the integration and testing on Android Ice Cream Sandwich (ICS 4.0+) development systems. Slot signaling associates each tracked (touch) ID with a slot number, making TTDA suitable for submission to the Linux repository for upstream releases into the global Linux community.

Cypress's reference platform is the Texas Instruments (TI) PandaBoard hardware.

1.3 Resources

1.3.1 Cypress Documentation

Document Number	Document Title	Description
001-89243	TrueTouch® CYTMA545 Product Family Release Notes - SRN89243	
001-87905	TrueTouch® CYTMA568 Product Family Release Notes - SRN87905	
001-89090	TrueTouch® CYTMA445 Product Family Release Notes - SRN89090	
001-88715	CYTMA545 TrueTouch® Multi-Touch All-Points Touchscreen Controller	The device datasheets contain information on features, system design, touchscreen performance, electrical specifications, and packaging.
001-82145	CYTMA568 TrueTouch® Multi-Touch All-Points Touchscreen Controller	
001-90820	CYTMA448 TrueTouch® Multi-Touch All-Points Touchscreen Controller	
001-87143	CYTMA445A TrueTouch® Multi-Touch All-Points Touchscreen Controller	
001-80396	CYTMA545 Technical Reference Manual (TRM)	Technical Reference Manual for CY8CTMA54X devices. Contains register map details and device operational theory.
001-87666	CYTMA568 Technical Reference Manual (TRM)	
001-89995	CYTMA448 Technical Reference Manual (TRM)	
001-88195	CYTMA445A Technical Reference Manual (TRM)	
001-85948	Touchscreen Manufacturing Testing with CYTMA448/545/568	Examples of how to use the built-in self-test interface on Cypress TrueTouch devices.
001-63571	CY3295-MTK	TrueTouch Manufacturing Test Kit User Guide
001-90764	Serial Wire Debug (SWD) Programming with TTDA User Guide	Includes prerequisites for the SWD firmware loader feature and provides guidance on its usage.

1.3.2 Public Reference Material

Reference	Description
http://en.wikipedia.org/wiki/Fastboot	Fastboot standard
http://developer.android.com/guide/developing/tools/adb.html	Android debugging host tool
http://source.android.com/tech/input/touch-devices.html	Android standards for touch devices
http://pandaboard.org	TI PandaBoard information
http://www.gnu.org/licenses/gpl-2.0.html	Open-source licensing used by the TTDA
LINUX Files in Build Tree	
kernel/Documentation/input/multi-touch-protocol.txt	Linux multi-touch protocol description
kernel/include/linux/input.h	Linux event signal definitions: Defines all multi-touch event signals used by the driver
kernel/Documentation/firmware_class/README	Text file containing the firmware class use design
Android	
	Android definition for multi-touch signaling

1.3.3 Definitions, Acronyms, and Abbreviations

Reference	Description
GPL	GNU General Public License
HID	Human Interface Device
I^2C	Inter-Integrated Circuit
NDA	Non-Disclosure Agreement
OTA	Over-the-Air (automatic firmware load at startup)
PIP	Packet Interface Protocol
SPI	Serial Peripheral Interface
TTDA	TrueTouch Driver for Android
TTHE	TrueTouch Host Emulator
TTSP	TrueTouch Standard Product
Upstream	Refers to the Linux repository found at http://www.kernel.org

1.4 Troubleshooting and FAQ

Q: *Touch device does not go into Sleep mode.*

A: Make sure the CONFIG_PM_RUNTIME function is enabled in TTDA. It must be enabled for the early suspend function in TTDA to work. The CONFIG_PM_RUNTIME function enables the runtime power management features of Linux so that TTDA can put the controlled touch device into the low-power state on early suspend/late resume events.

Q: *What are the benefits of the Bus Model in TTDA over monolithic design?*

A: Compared to traditional monolithic driver design, modular design is more flexible, customizable, scalable, and extensible. Modules can be added or removed at runtime. Each module subscribes to the required events from core modules. Debug modules can be inserted without a kernel rebuild, so debugging the driver during the product development and evaluation phase becomes easier. When you want to add new features to the driver, monolithic designs require driver redesign, while a modular design allows new features to be isolated to existing or new modules. It allows easier and faster development and implementation.

You can customize most of the features by modifying the Linux board configuration file. You do not need to modify the module files. Thus, the bus model lowers the barrier of entry and expedites the process for porting drivers.

Q: *What are the benefits of the Simplified Core Driver model TTDA over TTDA Bus Module model?*

A: The Simplified Core only needs a single Linux driver device. It also combines the most common worker modules into the Core module, thus reducing the complexity. The combined and integrated worker modules can be conditionally included in the kernel build and can include multi-touch (MTA or MTB), CapSense Button, and Proximity sensing. The full product support modules (device access, loader, and debug monitor) remain separate loadable kernel modules that can be built-in or built as modules. The loadable and integrated worker modules continue to subscribe to the Core for device interrupt event notification.

Q: *Where can I find register map information for TrueTouch parts?*

A: The register map information is documented in Technical Reference Manual (TRM). The document is distributed by a Cypress Field Applications Engineer (FAE) to customers under NDA. See Cypress Documentation for details.

Q: *Where can I find information about handshaking methods between the host and the TrueTouch products? How do I configure the handshaking method?*

A: The TTDA 3.x firmware is not configurable for handshaking method. It always uses the Synchronous Level Handshake: the firmware signals the event interrupt and holds it active until the host reads the last byte in the device read buffer. Normally, the driver will configure the interrupt event as low-edge trigger; however, if required by the host, the driver will configure the interrupt event for low-level trigger. This is done by setting the platform data field “level_irq_udelay” to a non-zero value.

Q: *I want to check and debug touch performance in an enclosed Android device with TTDA built in. Can I change the tuning configurable parameters without disassembling the unit?*

A: Yes. There are two ways of achieving this.

- Option 1: Connect the Android device to the TrueTouch Host Emulator (TTHE) via USB or Wi-Fi. You can see the touch performance, line drawing, and heat map data of the touch device in TTHE. It also allows you to make configuration changes in TTHE and apply directly to the Android device. See Mobile Tuner for details.
- Option 2: TTDA provides a Group 6 sysfs interface that you can use to read and write touch configurations. You can access the interface via Android Debug Bridge (ADB) connection. See Error Handling and Debug Interfaces for details.

Q: *What is the relationship between TTDA 2.x and TTDA 3.X?*

A: TTDA 3.X supports TSG5 products that use the Packet Interface Protocol (PIP) device interface, whereas TTDA 2.x supports TMA4xx/TMA1036 products that employ TrueTouch Standard Product (TTSP) register map interface. The TTDA 3.x design is based on the TTDA 2.2 architecture, but is re-designed specifically for the PIP interface.

Q: *In TTDA 3.0 and TDA 3.1, there is a #define CY_LDR_SWITCH_TO_APP_MODE_TIMEOUT in cyttsp5_loader.c file. What is its use and how do you set this value properly?*

A: The `CY_LDR_SWITCH_TO_APP_MODE_TIMEOUT` value is the timeout in the bootloader mode before the driver issues a launch application command to switch to the application mode. When the timer expires, the driver will get the HID descriptor for the bootloader and will execute the launch application command if the touch device is still in the bootloader mode. However, if the firmware has already launched the application before the `CY_LDR_SWITCH_TO_APP_MODE_TIMEOUT` timer expires, then the driver will not send the launch application command. The default value for this timeout is 300 ms.

Depending on your firmware implementation, this value may need to be modified. For example, if the firmware automatically launches the application in about 300 ms as well, chances are that the driver may get the HID descriptor indicating that the device is in the bootloader mode, which triggers a launch application command. At the same time, firmware may have switched to the application mode. Therefore, the launch application command will be executed in the application mode, which is not desirable. It is a good practice to set the two timeouts in the driver and firmware apart.

(**Note:** This definition is no longer used in TTDA 3.2 and later.)

2. Technical and System Description

2.1 Module Design

TTDA uses Linux modules for interfaces. These modules can be added or removed at runtime:

- TrueTouch Standard Product (TTSP) Bus Handler
  - Linux bus type; uses Linux-defined bus structure
  - Responsible for coordination of other modules
  - Creates a virtual bus that is represented to the Linux kernel
  - All other modules use the APIs provided by the bus module to register themselves
- Core Module
  - TTSP bus-defined core device structure (core device)
  - TTSP bus-defined core driver structure (core driver)
  - Responsible for interacting with the physical device (in terms of its host interface)
  - Needs helper functions provided by the core platform data to perform board-specific tasks, such as initialization, power-on and power-off, suspend and resume, and interrupt handling
  - Provides APIs to upper modules to access the physical device
- Adapter (I^{2}C/SPI) Modules
  - I^{2}C adapter device and driver
  - SPI adapter device and driver
  - TTSP bus-defined adapter structure (adapter device)
  - Linux host bus-defined driver structure
  - Responsible for interacting with the physical device (in terms of its bus interface)
  - Implement bus-specific read and write operations
- Multi-Touch Modules
  - TTSP bus-defined device structure (TTSP device)
  - TTSP bus-defined driver structure (TTSP driver)
  - Responsible for reporting touch events to Linux input subsystem
  - Two multi-touch modules for Linux’s multi-touch protocol A (MTA) and B (MTB)
    - MTA module provides multi-touch signaling to Android using Protocol A
    - MTB module provides multi-touch signaling to Android using Protocol B

Note: Do not run the MTA and MTB modules at the same time. Select and build with the one that the product’s Android version supports. See Multi-Touch Signal Types in Android for details.

- CapSense Button Module
  - Responsible for reporting button events to the Linux input subsystem
- Debug Module
  - Print in human- or machine-readable format using a standard file system (sysfs) switch
  - TTHE interface support
  - Provides kernel log messages on touch and button events for debugging information
- Device Access Module
  - Includes TTHE interface
  - Provides access to gather information (SysInfo) about the physical device
  - Provides access to execute TrueTouch Application commands
- Loader (Firmware Upgrade) Module
  - Responsible for the upgrade of the touch application
  - Manual ADB loader
    - Uses firmware class with manual concatenate file to loader firmware class sysfs
  - Automatic Over-the-Air (OTA) loader
    - Optional method to allow automatic OTA at bootup
    - Runs once if a new firmware image has a greater revision than that currently in the TrueTouch device
    - Uses firmware-class with an optionally built-in firmware file (as a *.h file) or firmware binary file added in to the loader firmware class sysfs (as a *.bin file)
- Device Access API Test Module (TTDA 3.1 and later only)
  - Provides sysfs interface to emulate the normal Device Access sysfs interface
  - Uses the Device Access API to send the sysfs inputs to the Device Access and return Device Access responses to the emulation interface back to the user.
- Proximity Module (TTDA 3.1 and later only)
  - Compatible with CY8CTMA545/CY8CTMA54X Base.V2 firmware
  - Provides proximity NEAR/FAR reporting
- Device Tree Module (TTDA 3.0.1 and later)
  - Provides dynamic mapping of platform data for each module requiring such information and requiring TTDA bus device registration at startup.
  - Provides an API for access to, and parsing of, a Device Tree for the TTDA. The API is called by each Adapter Module.
  - Although this module has init and exit module functions, there is no probe function.

  Note: The Device Tree module must be built-in.

2.2 System Architecture

Figure 2-1. TTDA 3.0 System Data Flow

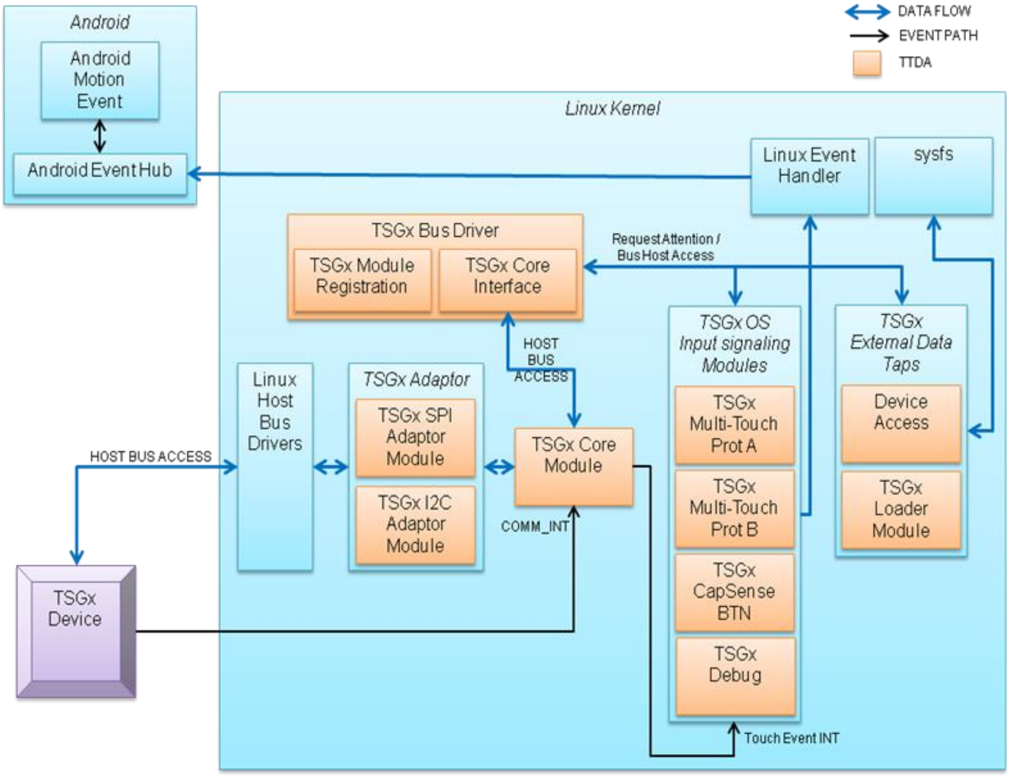


Figure 2-2. TTDA 3.0.1 System Data Flow

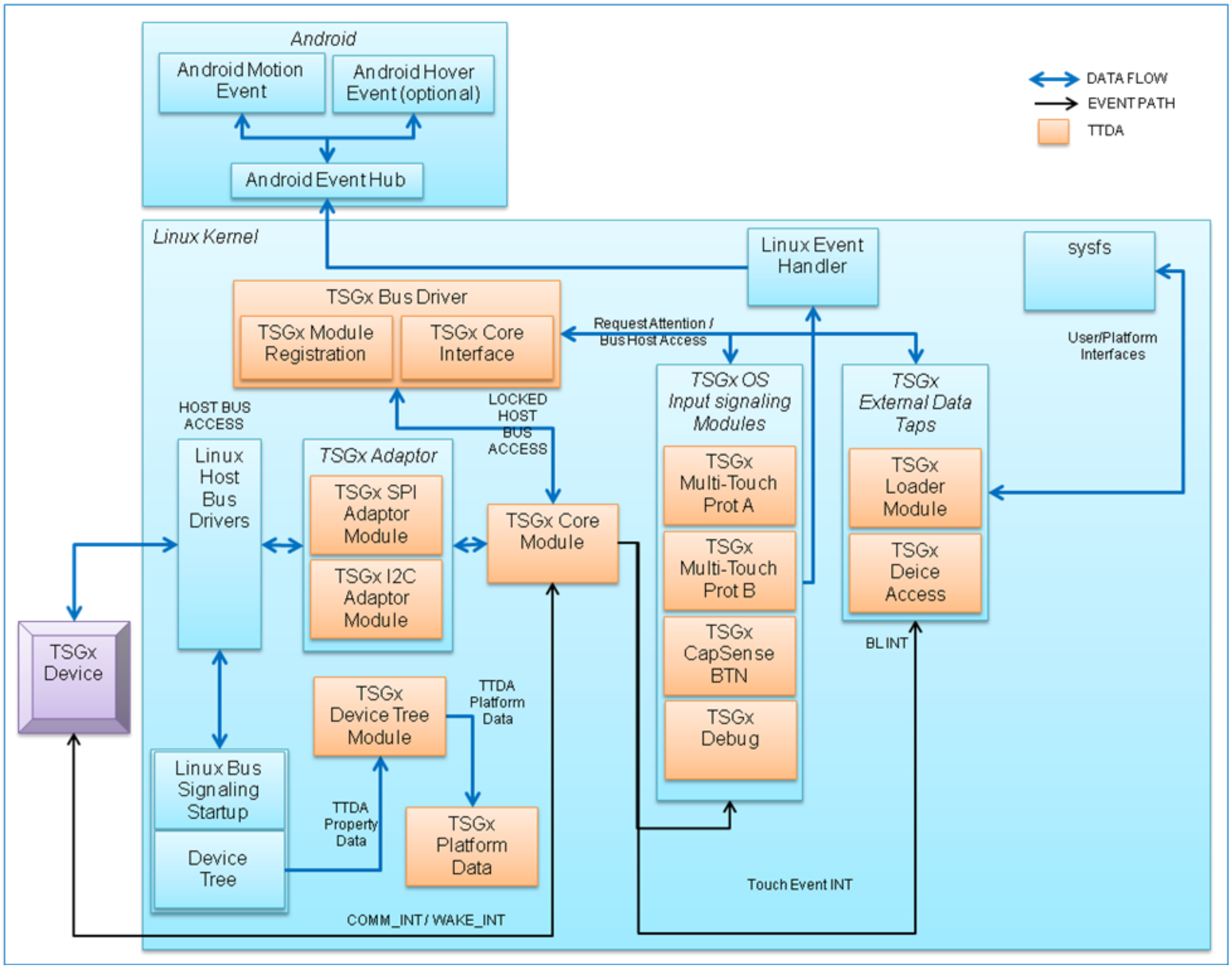


Figure 2-3. TTDA 3.1 System Data Flow

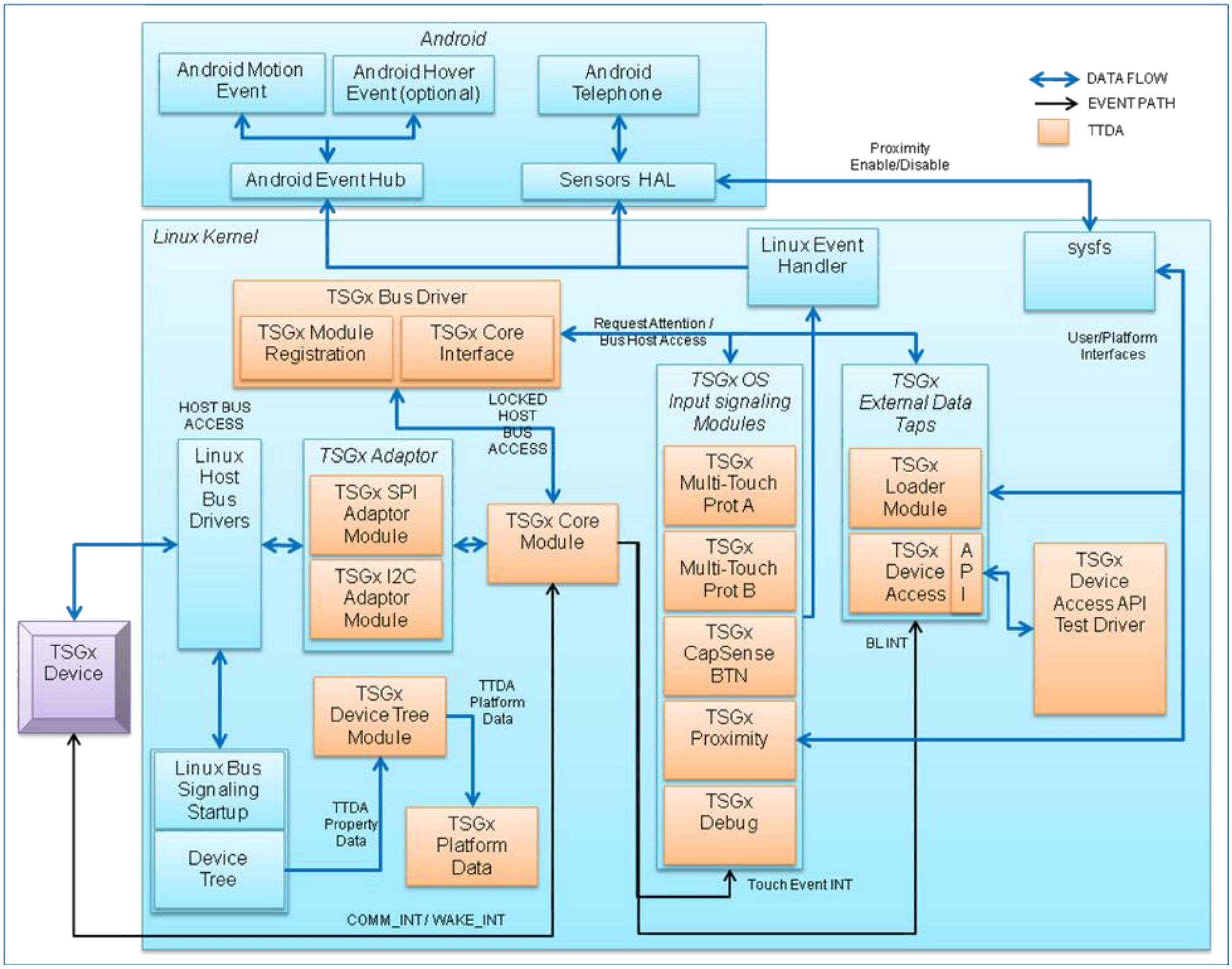
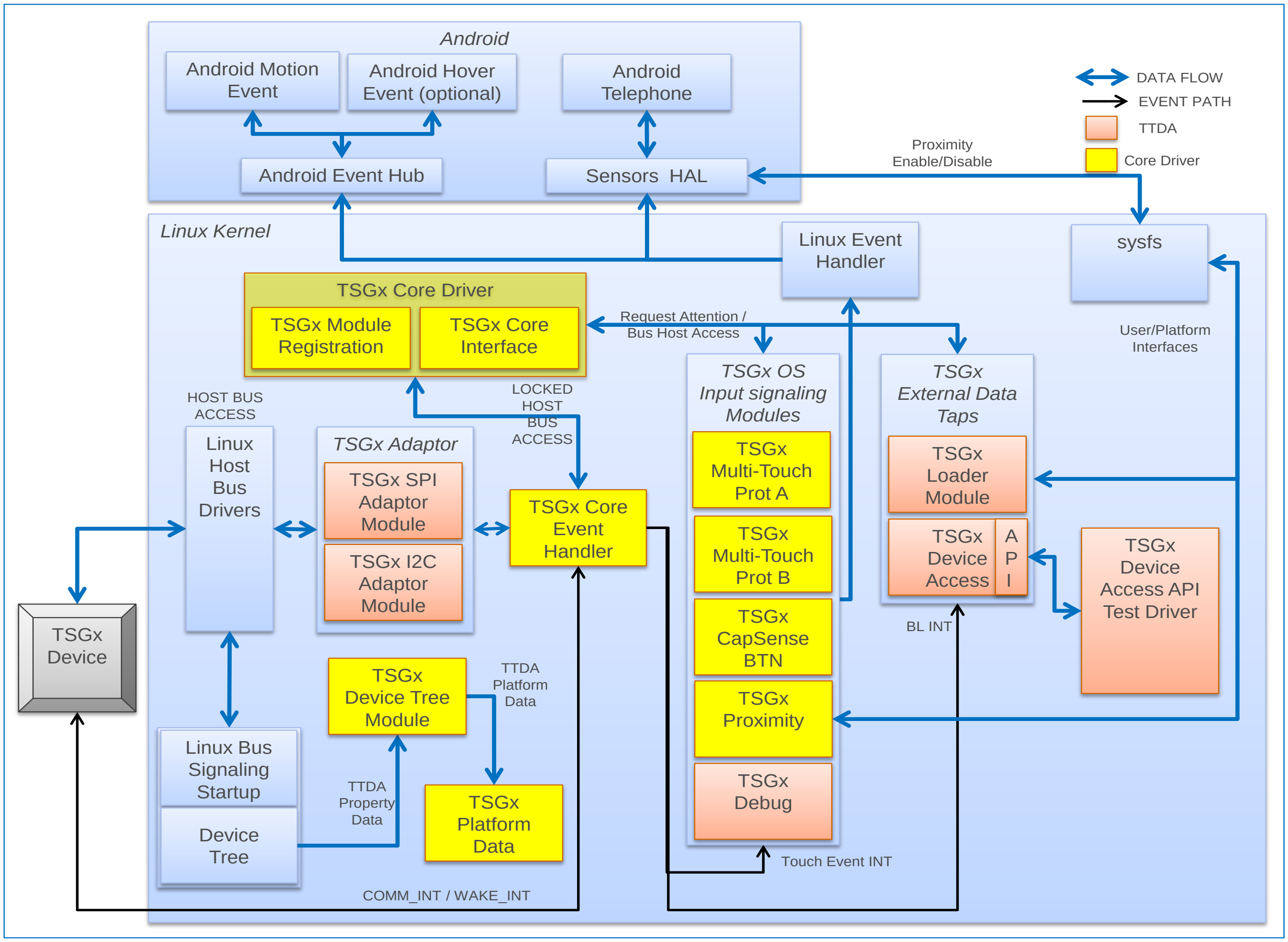


Figure 4 - TTDA 3.2, TTDA 3.3, TTDA 3.4 System Data Flow



2.3 Cypress Devices and Interfaces

TTDA 3.X is developed using Cypress's CYTMA545 development kit interfaced to TI PandaBoards. The CYTMA545 device register map and operational theory are available in the Cypress document 001-80396, *CY8CTMA54x/CYTMA545 Technical Reference Manual (TRM)*.

TTDA 3.X supports PIP, which exposes the following:

- Touch reports
  - Touch ID: range [0..9]
  - X: range [0..xmax] (xmax read from system information)
  - Y: range [0..ymax] (ymax read from system information)
  - Z: range [0..255]
  - Touch major: range [0..255]
  - Touch minor: range [0..255]
  - Orientation: range [-128..127]
  - Hover: optional
  - Event ID: check for touch lift-off; do not signal lift-off touches
- Button reports
- Operational interface for touch and CapSense button support
  - Touch report data will be converted to Linux multi-touch event signals.
  - Button report data will be converted to key codes.
  - Operation commands
  - A TTSP device user module provides a user interface to send operational commands and receive status via a standard file system (sysfs) interface.
  - The user can send (STORE) operational commands on the product command line interface using "echo" commands and receive the returned operational command status data (SHOW) using the "cat" command.
- Bootloader
  - In-field firmware upgrade using TTDA 3.x.
- Manufacturing test support
- Virtual key example code
  - Demonstrates how to set up a product baseline to assign touch area information to the Android platform so that it will automatically replace touch information in the defined key area with key codes.
  - Provides key code assignment to touch-geometry.

2.4 Multi-Touch Signal Types in Android

2.4.1 Multi-Touch Protocol A (MTA)

These signals are used in Android 2.0 and later and Android Éclair including revision 2.1. Protocol A signals include X and Y positions, and Z pressure for multiple touches. The Android Motion Event generator creates track information for each event report from the driver. Therefore, no hardware track ID information is included in the motion data. The X and Y position information is reported as absolute positions.

2.4.2 Multi-Touch Protocol B (MTB)

These signals are used in Android 4.0 Ice Cream Sandwich and later. Protocol B signals include the same X and Y positions and Z pressure as Protocol A signals; however, a hardware-generated track ID is also included in the interface. The X and Y position information is reported as absolute positions. Protocol B is required if your application is to be submitted to Linux.

3. TTDA Kit Contents

The TTDA kit includes the following:

- TTDA User Guide (this document)
- Release notes for the most current revision of the TTDA kit
- Source files
  - The “kernel” directory is the root of a tree structure of the files supplied by Cypress to build the driver, mirroring the destination in Linux
  - Table 3-2 and Table 3-2 list the files included in TTDA 3.x. The board configuration file provided is for the TI PandaBoard reference platform, which is used for TTDA development. TTDA users must customize the board configuration file for specific custom design. See the Driver Porting section for more details.
- COPYING.txt
  - This is the required open source “GNU GENERAL PUBLIC LICENSE”. If you are not familiar with this license, read it completely. Cypress can supply this file in accordance with the license.

Table 3-1. TTDA 3.1 and Below File List

File Name and Location	Description
..kernel/arch/arm/mach-omap2/board-omap4panda.c	Product-specific configuration with Cypress driver for the PandaBoard platform.
..kernel/arch/arm/boot/dts/omap4-panda.dts	Example Device Tree file that contains the hardware configuration of the Cypress driver for the PandaBoard platform. (TTDA 3.0.1 and later).
..kernel/drivers/input/touchscreen/cyttsp5_btn.c	Provides Linux signaling for CapSense buttons. The buttons typically map to the Linux HOME, MENU, BACK, and SEARCH keys.
..kernel/drivers/input/touchscreen/cyttsp5_bus.c	TSGX bus driver module. It provides module registration and core interface API.
..kernel/drivers/input/touchscreen/cyttsp5_core.c	Provides device startup, interrupt service support, device data extraction, output command management, and device suspend and resume threads.
..kernel/drivers/input/touchscreen/cyttsp5_debug.c	Provides a debug module for printing the raw touch and button information onto the kernel log .
..kernel/drivers/input/touchscreen/cyttsp5_device_access.c	Provides external user access to the device through the driver to provide an interface for writing commands to the device and to receive the corresponding responses.
..kernel/drivers/input/touchscreen/cyttsp5_device_access.h	Header file for above. Device access module name string.
..kernel/drivers/input/touchscreen/cyttsp5_i2c.c	Provides an interface primitive function for performing I^2Ctransfers. Transfers are read or write operations. This module also provides bus binding for the TSG5 core module.

File Name and Location	Description
..kernel/drivers/input/touchscreen/cyttsp5_devtree.c	Device Tree module that parses the Device Tree and creates the platform data for devices and registers them with the bus. (TTDA 3.0.1 and later).
..kernel/drivers/input/touchscreen/cyttsp5_devtree.h	Header file for the Device Tree module. (TTDA 3.0.1 and later).
..kernel/drivers/input/touchscreen/cyttsp5_i2c.h	Module identification string.
..kernel/drivers/input/touchscreen/cyttsp5_loader.c	Provides both manual and automatic firmware update capability.
..kernel/drivers/input/touchscreen/cyttsp5_mt_common.c	Common multi-touch functions for the TSG5 Multi-touch module.
..kernel/drivers/input/touchscreen/cyttsp5_mt_common.h	Common function pointer structures and context data structure.
..kernel/drivers/input/touchscreen/cyttsp5_mta.c	Multi-touch Protocol A module. Provides Linux Protocol A touch signaling.
..kernel/drivers/input/touchscreen/cyttsp5_mtb.c	Multi-touch Protocol B module. Provides Linux Protocol B touch signaling.
..kernel/drivers/input/touchscreen/cyttsp5_regs.h	TSG5 device register bit definitions.
..kernel/drivers/input/touchscreen/cyttsp5_spi.c	Provides an interface primitive function for performing SPI transfers. Transfers are read or write operations. This module also provides bus binding for the TSG5 core module.
..kernel/drivers/input/touchscreen/cyttsp5_spi.h	Module identification string.
..kernel/drivers/input/touchscreen/Kconfig	Local modified build file for PandaBoard/TTDA driver.
..kernel/drivers/input/touchscreen/Makefile	Local modified build file for PandaBoard/TTDA driver.
..kernel/include/linux/cyttsp5_btn.h	Button module name string and platform data definition.
..kernel/include/linux/cyttsp5_bus.h	Defines TSG5 device and driver structures and the core interface API.
..kernel/include/linux/cyttsp5_core.h	Defines the platform data structure of the Core module.
..kernel/include/linux/cyttsp5_mt.h	Multi-touch module platform data structure and name string.
..kernel/Makefile	Kernel root build. PandaBoard build modifiers to include Cypress driver.
<android>/system/usr/idc/cyttsp5_mt.idc	Sample file that tells Android how to interpret multi-touch signals sent by the Cypress driver. Needs to be pushed into Android /system/usr/idc.
<android>/system/usr/keylayout/cyttsp5_mt.kl	Sample virtual key layout file. Needs to be pushed into Android /sys/usr/keylayout if the virtual key is to be used.
<android>/system/usr/keychars/cyttsp5_mt.kcm	Sample virtual key configuration file. Needs to be pushed into Android /sys/usr/keychars if virtual key is to be used.
<kernel>/firmware/cyttsp5_fw.bin	TTSP firmware binary file for automatic driver flashing at startup based on firmware revision information.

Table 3-2. TTDA 3.2 and Above File List

File Name and Location	Description
..kernel/arch/arm/mach-omap2/board-omap4panda.c	Product specific configuration with Cypress driver PandaBoard platform.
..kernel/arch/arm/boot/dts/omap4-panda.dts	Example Device Tree file that contains hardware configuration for the Cypress driver PandaBoard platform. (TTDA 3.0.1 and later).
..kernel/arch/arm/boot/dts/apq8074-dragonboard.dtsi	Example Device Tree file that contains hardware configuration for the Cypress driver MSM8074 Dragon Board platform. (TTDA3.4).
..kernel/arch/arm/configs/msm8974_defconfig	Sample default configuration file for the MSM8074 Dragon Board platform. This file is modified to include the TTDA driver. (TTDA3.4).
..kernel/drivers/input/touchscreen/cyttsp5_btn.c	Provides CapSense button Linux signaling. The buttons typically map to the Linux HOME, MENU, BACK, and SEARCH keys.
..kernel/drivers/input/touchscreen/cyttsp5_core.c	Provides device startup, interrupt service support, device data extraction, output command management, and device suspend and resume threads.
..kernel/drivers/input/touchscreen/cyttsp5_debug.c	Provides a debug module for printing to the kernel log the raw touch and button information.
..kernel/drivers/input/touchscreen/cyttsp5_device_access.c	Provides external user access to the device through the driver to provide an interface for writing commands to the device and to receive the corresponding responses.
..kernel/drivers/input/touchscreen/cyttsp5_devtree.c	Device Tree module that parses the Device Tree and creates platform data for devices and registers them with the bus. (TTDA 3.0.1 and later).
..kernel/drivers/input/touchscreen/cyttsp5_i2c.c	Provides an interface primitive function for performing I^{2}C transfers. Transfers are read or write operations. This module also provides bus binding for the TSG5 core module.
..kernel/drivers/input/touchscreen/cyttsp5_loader.c	Provides both manual and automatic firmware update capability.
..kernel/drivers/input/touchscreen/cyttsp5_mt_common.c	Common multi-touch functions for the TSG5 Multi-touch module.
..kernel/drivers/input/touchscreen/cyttsp5_mta.c	Multi-touch Protocol A module. Provides Linux Protocol A touch signaling.
..kernel/drivers/input/touchscreen/cyttsp5_mtb.c	Multi-touch Protocol B module. Provides Linux Protocol B touch signaling.
..kernel/drivers/input/touchscreen/cyttsp5_platform.c	Holds platform data for the core and primitive functions which are customizable.
..kernel/drivers/input/touchscreen/cyttsp5_proximity.c	Provides the proximity feature capability.
..kernel/drivers/input/touchscreen/cyttsp5_regs.h	TSG5 device register bit definitions.
..kernel/drivers/input/touchscreen/cyttsp5_spi.c	Provides an interface primitive function for performing SPI transfers. Transfers are either a read or write operation. This module also provides bus binding for the TSG5 core module.
..kernle/drivers/input/touchscreen/cyttsp5_test_device_access_api.c	Example module that shows how to interface to the TTDA3.x device access through another driver.

File Name and Location	Description
..kernel/drivers/input/touchscreen/Kconfig	Local modified build file for PandaBoard/TTDA driver.
..kernel/drivers/input/touchscreen/Makefile	Local modified build file for PandaBoard/TTDA driver.
..kernel/drivers/input/touchscreen/Kconfig.dragonboard-apq8074	Local modified build file for MSM Dragon Board/TTDA driver (TTDA3.4).
..kernel/drivers/input/touchscreen/Makefile.dragonboard-apq8074	Local modified build file for MSM Dragon Board/TTDA driver (TTDA3.4).
..kernel/include/linux/cyttsp5_core.h	Defines the Core module platform data structure.
..kernel/include/linux/cyttsp5_device_access-api.h	Calling interfaces for the device access through the TTDA from another driver.
..kernel/include/linux/cyttsp5_platform.h	Holds platform data for the core, loader, and primitive functions which are customizable.

Not included in driver distribution – Contact your local Cypress FAE for more details on these files.	
..kernel/Makefile	Kernel root build. PandaBoard build modifiers to include Cypress driver.
<android>/system/usr/idc/cyttsp5_mt.idc	Sample file that tells Android how to interpret multi-touch signals sent by Cypress driver. Needs to be pushed into Android /system/usr/idc.
<android>/system/usr/keylayout/cyttsp5_mt.kl	Sample virtual key layout file. Needs to be pushed into Android /sys/usr/keylayout if virtual key is to be used.
<android>/system/usr/keychars/cyttsp5_mt.kcm	Sample virtual key configuration file. Needs to be pushed into Android /sys/usr/keychars if virtual key is to be used.
<kernel>/firmware/cyttsp5_fw.bin	TTSP firmware binary file for automatic driver flashing at startup based on firmware revision information.

4. Software Design

This chapter describes the TTDA software design, including the Linux bus type defines and implementation, build methods, introduction to the CapSense button keycode signaling module, proximity module, and debug module. It also documents virtual key signaling, user interfaces, implementation of TTDA data types and control structure, as well as application development environment. This TTDA bus design is for TTDA 3.1 and earlier.

For TTDA 3.2 and later, this method is simplified to use the binding of the host bus instead of the special TTDA bus. Also, the MTA/B, Button, and Proximity worker modules are combined with the core module to form a Core Driver module and are conditionally included in the build. The loader and device access modules remain loadable modules.

4.1 Linux Bus Type

4.1.1 Defines

- TTSP Bus Handler
  - Linux bus-defined structure
- Core Modules
  - TTSP bus-defined device structure
  - TTSP bus-defined driver structure
- Worker Modules
  - TTSP bus-defined device structure
  - TTSP bus-defined driver structure
- Adapter Modules
  - TTSP bus-defined device structure
  - Linux bus-defined driver structure

4.1.2 TTSP Bus Handler (TTDA3.1 and below)

- Must be present and running in Linux before any other modules
- Will be compiled as built-in
- Started automatically by the kernel at system startup
- Provides definition and registration for Core Module devices:
  - Core Modules
  - Worker Modules
  - Adapter Modules
- Provides binding between Adapter, Core, and Worker modules
- Core Module devices are registered using call to TTSP bus registration added to the end of the system board configuration system startup sequence
  - Platform data and device structures are instantiated in the board configuration file

- Core Module drivers are started by the system as either built-in or as LKM using the *insmod* directive and the driver *module_init()* function calls the TTSP bus driver registration
  - □ The Core Module device must be registered before the Core Module driver is registered
- Worker Module devices are either registered using call to TTSP bus registration in the board configuration or as part of the *module_init()* function call
- Worker Module drivers are started by the system as either built-in or as LKM using the *insmod* directive and the driver *module_init()* function calls the TTSP bus driver registration
  - □ The Worker Module device must be registered before the Worker Module driver is registered
- Adapter Module devices are started by the system as built-in as part of the appropriate product communications bus initiation sequence
- A complete set of corresponding Adapter, Core, and Worker Module devices and drivers must be registered with the TTSP bus handler
  - □ When all necessary modules are registered, the TTSP Bus Handler will call the probe functions for each of the modules.

4.1.3 Three Module Types Defined

1. Core Module

Provides service for:

- Device startup
- Device interrupt pin
  - □ The interrupt service will read the current mode from the device and call back the subscribers to run within the context of the ISR to ensure that handling is complete before the interrupt service complete signal is sent to the OS by the ISR. The subscriber call back can run inline or send a waking signal to another thread. Subscriptions will be set up and served using Linux List objects. Wait on event queues with conditional wake up is the preferred method of synchronizing threads
  - □ The interrupt service will base the current mode on device conditions. New modes are only added if they can be determined by reading the device
- Concurrent access protection
  - □ Interlocks are provided
  - □ Request access
    - o Core provides access to each module in the order of request
    - o Core blocks other requester(s) until access is relinquished by accessing module
- Mode-protected read/write access calls to allow subscribed modules access to adapter primitives. Core translates read/write requests into appropriate adapter read/write adapter calls
- Platform data
  - □ Driver default Easy Wakeup Gesture ID (TTDA 3.1 and later)
- Platform data configurable function calls
  - □ Initialization
    - o Platform-specific startup initialization code
  - □ Hard reset
    - o Includes reset and wait blocking capability
    - o Includes soft reset as default if hard reset function is not defined
  - □ Wakeup
    - o Provide hard wakeup interrupt pin strobing
    - o Alternative includes power cycling for power management instead of using device low power mode

- Suspend/Resume handling
    - Suspend thread
        - Core puts the device into Deep Sleep.
            - i. Sets the Deep-Sleep bit in the Host Mode register
        - Core puts the device into the Easy Wakeup low-power mode
            - i. *Sysfs* object for user selection of wake gesture
            - ii. Sets interrupt properties to allow waking host
            - iii. Writes the Easy Wakeup command to device
    - Resume thread
        - Core wakes the device from Deep Sleep
            - i. Wakes by pulsing the interrupt pin
            - ii. Wakes by performing a read of device
        - Device wakes host on Easy Wake gesture recognition
            - i. Ping command to verify device

2. Worker Module

Provides an interface to OS signaling and OS file system objects

- Subscribes to Core Module for interrupt event signaling. Provides interrupt attention callback function to access device data. Typically the callback is called inline without thread rescheduling so that it is ensured that data is read before the core releases the interrupt back to the kernel (IRQ_HANDLED).
    - Multi-touch event signaling
        - Sends multi-touch signal usage, including min/max signal values, to kernel to set capabilities at startup
            - i. Uses platform data to configure signals to be used based on product requirements
        - Runtime touch positions
            - i. Sends each touch value as part of its Linux signal
    - CapSense button keycode signaling
    - Proximity signaling (TTDA 3.1 and later)

Provides optional device access through user space

- User interfaces
    - Sysfs interface standard
        - Show
        - Store
    - Firmware Class standard
        - Loading object
        - Data object

3. Adapter Module

- Provides an interface to the device communication bus
  - □ I^2C
  - □ SPI
- Exposes read/write primitives for Core access to device
- Adapter devices launched by adding adapter device structure to system launch list for the bus type
- Adapter drivers launched as loadable kernel module (LKM) with TTSP Bus driver registration call in the module_init() function
- In TTDA 3.0.1 and later, the Adapter Modules are updated to provide optional dynamic parsing of the Device Tree to see if matching tree properties are provided for the TTDA

  The normal board configuration calls to initiate the host bus for the adapter and to register the TTDA modules using statically-defined platform data. If the device tree compile-time option is used, this data is replaced with a system call to provide binding of the Device Tree. The Device Tree allows removing the platform data from the board configuration and adding it into the Device Tree in the form of property fields. The Device tree also includes property fields used by the system to initiate the host bus. The adapter Modules add calls to dynamically allocate a platform data structure for the module, parse the Device Tree to access the platform data content for each TTDA module using system parsing calls, and copy the Device Tree data to the allocated structure and then provide calls to register the modules with the TTDA Bus driver. The calls include references to the dynamically allocated platform data.

4.2 Build Methods

4.2.1 Built-In

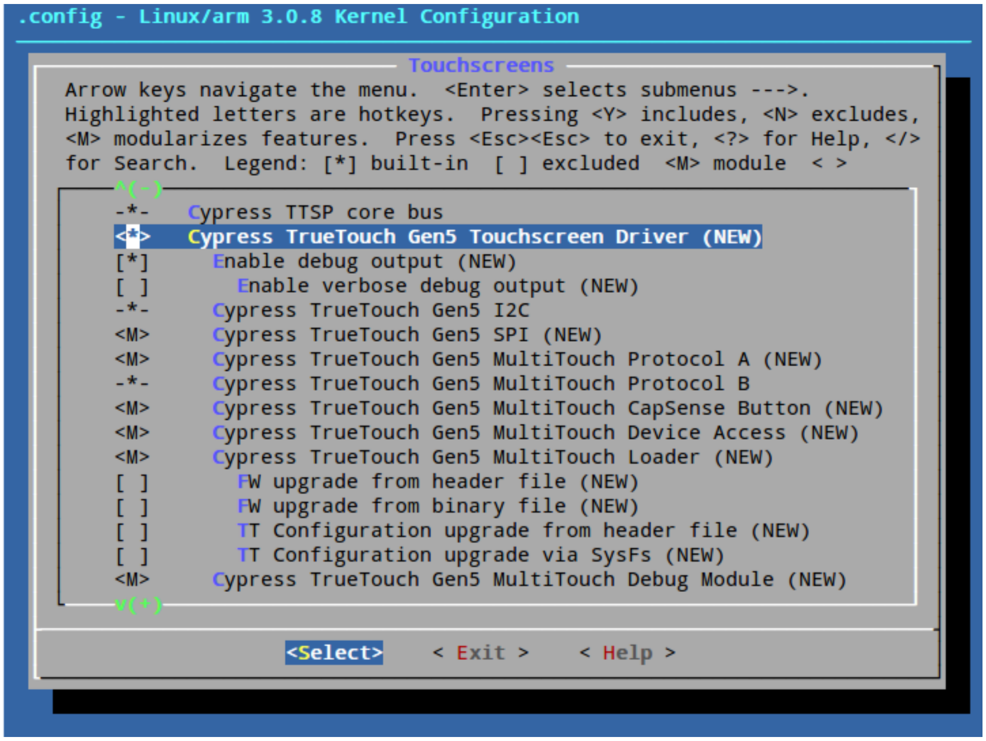
- Use the standard build with the “-y” option to create “.o” files
- Included in the built-in ”.o” intermediate linker/loader files

4.2.2 LKM

- Uses Linux loadable kernel modules for touch driver components
  - □ These modules are built with the “-m” option to create “.ko” files.
  - □ The “.ko” files can be loaded by hand/script using the insmod command line directive.
  - □ The “.ko” files can be loaded at system startup by adding insmod directives in the Android “init.rc” file.
- Three types of LKM touch drivers

1. Core: System Information and COMM_INT interrupt service handling. Interrupt attention and device mode handling.
   - □ The core reads the length bytes (TTDA 3.x) on each interrupt.
   - □ Subscribing modules use the status information provided by the core.
     - o Eliminates redundant read of status bytes by each subscriber.
2. Worker: Provides device information to system OS signal. This type of module can include multi-touch signaling, user sysfs driver control, and firmware class device loader modules.
   - □ Interprets the length bytes (TTDA 3.x) read by the core.
   - □ Reads any required additional touch record bytes.
3. Adapter: System communication bus interface module.

Figure 4-1 Module Selection (TTDA 3.0 Example)



```
.config - Linux/arm 3.0.8 Kernel Configuration
                          Touchscreens
  Arrow keys navigate the menu.  <Enter> selects submenus --->.
  Highlighted letters are hotkeys.  Pressing <Y> includes, <N> excludes,
  <M> modularizes features.  Press <Esc><Esc> to exit, <?> for Help, </>
  for Search.  Legend: [*] built-in  [ ] excluded  <M> module  < >
      ^(-)
      -*-   Cypress TTSP core bus
      <*>   Cypress TrueTouch Gen5 Touchscreen Driver (NEW)
      [*]     Enable debug output (NEW)
      [ ]       Enable verbose debug output (NEW)
      -*-     Cypress TrueTouch Gen5 I2C
      <M>     Cypress TrueTouch Gen5 SPI (NEW)
      <M>     Cypress TrueTouch Gen5 MultiTouch Protocol A (NEW)
      -*-     Cypress TrueTouch Gen5 MultiTouch Protocol B
      <M>     Cypress TrueTouch Gen5 MultiTouch CapSense Button (NEW)
      <M>     Cypress TrueTouch Gen5 MultiTouch Device Access (NEW)
      <M>     Cypress TrueTouch Gen5 MultiTouch Loader (NEW)
      [ ]       FW upgrade from header file (NEW)
      [ ]       FW upgrade from binary file (NEW)
      [ ]       TT Configuration upgrade from header file (NEW)
      [ ]       TT Configuration upgrade via SysFs (NEW)
      <M>     Cypress TrueTouch Gen5 MultiTouch Debug Module (NEW)
      v(+)

                    <Select>    < Exit >    < Help >
```

4.2.3 Device

Refer to *Part-Specific Support* in TrueTouch Device Support.

4.2.4 O/S

Refer to the Linux multi-touch protocol, Linux event signal definitions, and the Android definition for multi-touch signaling. Links to these documents are available in the Cypress Documentation section.

Protocol A (required for Alpha deliverables and Gingerbread support):

1. ABS_MT_TRACKING_ID
2. ABS_MT_POSITION_X
3. ABS_MT_POSITION_Y
4. ABS_MT_PRESSURE
    a. Z or 0 if hover.
5. ABS_MT_TOUCH_MAJOR
    a. Touch major or 1 if touch major is 0 and Z is greater than 0. If Z is 0, then touch major is 0.
6. ABS_MT_TOUCH_MINOR
    a. Touch minor or 1 if touch minor is 0 and Z is greater than 0. If Z is 0, then touch minor is 0.
7. ABS_MT_ORIENTATION
8. Input reports are signaled with either BTN_TOOL_FINGER if the touch type is Finger or Glove or with BTN_TOOL_PEN if the touch type is Stylus. These signals are used for both the button press and button release signaling for the touches.

9. The following Linux API calls are made to synchronize multi-touch signaling:
    a. Input_mt_sync: after each touch set of signals
    b. Input_sync: after all the touches. Results in Linux event handling

Protocol B (requires format for Linux Submission – ICS and later):

1. ABS_MT_TRACKING_ID – this value is assigned by the Linux event handler
2. ABS_MT_POSITION_X
3. ABS_MT_POSITION_Y
4. ABS_MT_PRESSURE
    a. Z or 0 if hover (optional).
5. ABS_MT_TOUCH_MAJOR
    a. Touch major or 1 if touch major is 0 and Z is greater than 0. If Z is 0, then touch major is 0.
6. ABS_MT_TOUCH_MINOR
    a. Touch minor or 1 if touch minor is 0 and Z is greater than 0. If Z is 0, then touch minor is 0.
7. ABS_MT_ORIENTATION
8. ABS_MT_SLOT
    a. ABS_MT_TOOL_TYPE
        i. MT_TOOL_FINGER: Finger or Glove
        ii. MT_TOOL_PEN: Stylus
9. The following Linux API call is made to synchronize multi-touch signaling:
    a. Input_sync: after all the touch signals. Results in Linux event handling.

The following tables provide device to O/S signal mapping. All device signals are translated and sent to the O/S as type “int”.

Note: TTDA obtains the report structure from the Report Descriptor register and does not use hardcoded values. Typical offset and size values are shown in Table 4-1. Protocol A and Table 4-2. Protocol B.

.

Table 4-1. Protocol A

Item	Device Signal	O/S Signal	Signal Evaluation
1	Touch ID	ABS_MT_TRACKING_ID	T = (int)tch_rec[1] & 0x1F
2	X	ABS_MT_POSITION_X	X = (int)tch_rec[3] << 8 + tch_rec[2]
3	Y	ABS_MT_POSITION_Y	Y = (int)tch_rec[5] << 8 + tch_rec[4]
4	Z	ABS_MT_PRESSURE	P = (int)tch_rec[6]
5	Event ID	N/A	E = (int)tch_rec[1] >> 5 & 0x03
6	Major	ABS_MT_TOUCH_MAJOR	M = (int)tch_rec[7]
7	Minor	ABS_MT_TOUCH_MINOR	m = (int)tch_rec[8]
8	Orientation	ABS_MT_ORIENTATION	O = (int)tch_rec[9]

Table 4-2. Protocol B

Item	Device Signal	O/S Signal	Signal Evaluation
1	Touch ID	ABS_MT_SLOT	T = (int)tch_rec[1] & 0x1F
2	X	ABS_MT_POSITION_X	X = (int)tch_rec[3] << 8 + tch_rec[2]
3	Y	ABS_MT_POSITION_Y	Y = (int)tch_rec[5] << 8 + tch_rec[4]
4	Z	ABS_MT_PRESSURE	P = (int)tch_rec[6]
5	Event ID	N/A	E = (int)tch_rec[1] >> 5 & 0x03
6	Major	ABS_MT_TOUCH_MAJOR	M = (int)tch_rec[7]
7	Minor	ABS_MT_TOUCH_MINOR	m = (int)tch_rec[8]
8	Orientation	ABS_MT_ORIENTATION	O = (int)tch_rec[9]

Touch Error Handling

- If the interrupt service detects an unexpected device initiated reset
  - A driver restart is queued
  - On restart, the driver notifies startup notification subscribers.
- If a TTSP module receives a startup notification from the core, then all current touch tracks are terminated and then the module continues normally. A debug print can be posted to the kernel log.
- If a TTSP module detects that a touch report has an invalid number of touches reported:
  - If the number of touches is greater than the maximum touches reported by System Information, then the module uses the maximum touch number to handle touches. In this case, the device may be tracking more touches than can be reported. Only the touches that can be reported are converted to touch signals to the OS. An error can be posted to the kernel log or reported as a debug print.
- A Linux system timer is used to implement a watchdog to periodically ping the device using the PIP defined operational NULL command. If the device fails to respond, then the driver queues a restart.

4.3 CapSense Button Keycode Signaling Module – BTN Module

The BTN module subscribes to the Core module for touch event interrupt notification. The Core calls the subscription callback for the subscribing BTN module.

- Provides the method to map keycodes to buttons in touch module platform data.
  - The Core module provides an array of button-to-key code conversion values. A copy of the array is attached to the system information read by the Core module. This array is part of the Core module platform data and is made available to the BTN module when the BTN module requests a pointer to the system information.
- Adds button to keycode signaling setup in touch module probe function.
- This is implemented as a separate TTSP module that subscribes for the interrupt events and uses the status information read by the core. If debug is enabled, the module also reads the I^{2}C bus for the button difference report data.
- The module keeps a record of which buttons are pressed and released and reports key press and release when it detects a change in button condition. Multiple buttons can change state on each interrupt event.

4.4 Proximity Sensing – Proximity Module (TTDA 3.1 and Later Only)

The Proximity module subscribes to the Core module for touch-event interrupt notification. The Core calls the subscription callback for the subscribing Proximity module. The Proximity module signals the operating system with the ABS_DISTANCE input sensor signal with value either 0 (NEAR) or 1 (FAR).

The Proximity module provides an enable/disable sysfs interface to allow the user space to control turning proximity detection on and off. Proximity detection must also be disabled during suspend. The device proximity detection is enabled by setting the Proximity Bit in the device Scan Type register and disabled by clearing the bit. The sysfs interface is `/sys/bus/ttsp5/devices/cyttsp5_proximity.main_ttsp_core/enable`.

SHOW: current enable/disable state of Proximity detection

STORE: set enable/disable state of Proximity detection

Additionally, for Android compatibility, a special “sensors.c” or “sensors.cpp” file is required and must be built into the Android image. The “sensors” file contains the Hardware Abstraction Layer (HAL) interface for the Android for all supported sensor devices including Proximity detectors. Proximity is disabled by default on startup.

The Proximity module should be a built-in module so that it is available to the OS during startup if the OS needs to enable the proximity immediately. See the Proximity section.

4.5 Dynamic Debugging – Debug Module

The Debug module subscribes to the Core module for interrupt notifications for touch events. . The Core calls the subscription callback for the subscribing Debug module. The Debug module formats the touch record information and prints the information to the kernel log. The format of the touch data is in either human readable or terse machine readable format. The machine readable format is suitable for the touch interface of the TTHE Mobile Tuner.

The Debug module should only be built as LKM and not built-in because it will degrade normal touch performance. The Debug module should be loaded by the user on demand and removed when the debug session ends.

- Subscribes for touch interrupts
- On-touch interrupt
  - Increments an interrupt count
  - Reads complete touch record and prints raw or formatted byte content onto the kernel log along with the interrupt count. Default is raw bytes suitable for TTHE Mobile Tuner monitoring for TTHE touch tracking and line drawing. This information is printed with the string preface tag “cyttsp5_OpModeData”.
  - Prints the sensor data onto the kernel log with the string preface tag “cyttsp5_sensor_monitor”. This is activated automatically when the start *Sensor Data Mode* Configuration and test command is sent by TTHE when using the device access Group 15.
  - Provides the sysfs interface `cyttsp5_interrupt_count` to show or store the interrupt count number
    i. SHOW: current interrupt count
    ii. STORE: reset interrupt count
  - Provides the sysfs interface `cyttsp5_formated_output` to select raw or formatted byte output
    i. SHOW: current raw or formatted selection
    ii. STORE: set raw or formatted selection

4.6 Virtual Key Signaling

Example code and files:

- Key geometry mapping
- Keycode assignment
- See Virtual Keys for details

4.7 User Interfaces

Typically exposed via user LKM based on TTSP Module type

4.7.1 Custom User Commands

4.7.1.1 Standard

- Driver version
    - SHOW: Driver name, date, revision. Example of checking driver revision:

    -
      ```
      # cd /sys/bus/ttsp5/devices/main_ttsp_core.cyttsp5_i2c_adapter
      # cat drv_ver
      ```
  - STORE: N/A
- Firmware version
    - SHOW: TrueTouch product ID, firmware major revision, firmware minor revision, and firmware revision control number. Example of checking firmware major and minor revision:

    -
      ```
      # cd /sys/bus/ttsp5/devices/main_ttsp_core.cyttsp5_i2c_adapter
      # cat ic_ver
      ```
  - STORE: N/A

4.7.1.2 Debug

- Tools: These allow manual testing of the suspend/resume functions and testing the soft/hard device reset functions.
  - SHOW: flags
  - STORE:
    - Set Driver Suspend
    - Set Driver Resume
    - Soft Reset Device
    - Hard Reset Device
- Hardware interrupt pin status: This allows reading the current state of the COMM_INT pin as a digital value.
  - SHOW: interrupt pin level in range [0,1]
  - STORE: N/A
- Hardware reset: This allows restarting the driver and resetting the hardware; the driver is resynchronized with the hardware.
  - SHOW: N/A
  - STORE: perform driver restart; hardware is reset during restart
- Hardware recover: This allows manual reset of the device or wakeup of the device.
  - SHOW: N/A
  - STORE:
    - 0: Hard Reset Device
    - 1: Wakeup Device
- Driver IRQ enable/disable: This allows manual disabling/re-enabling COMM_INT pin interrupts to the driver.
  - SHOW: Driver IRQ is enabled or disabled
  - STORE:
    - 0: disable driver interrupts
    - 1: enable driver interrupts

4.7.1.3 Extended Command Set

This exposes a set of commands that can be used to control the driver to get access to TrueTouch Application and Bootloader commands. The Output ID, Command, Status, and Response registers are accessible via the sysfs interface.

- Output Register ID
  - SHOW: display the selected Output Register ID
  - STORE: Register ID select:

    1: Select Bootloader Output Register ID

    2: Select Operational Output Register ID
- Command
  - SHOW: N/A
  - STORE: Output Register ID

    0000h – Driver command: driver receives and processes the output report (no driver commands are defined at this time)

    Non-zero – Device pass-through command: driver passes the entire command as it is received directly to the device
- Status
  - SHOW: Response status

    0: Response not ready

    1: Response ready
  - STORE: N/A
- Response
  - SHOW: Response bytes

    a. Driver command response bytes

    b. Device pass-through command response bytes
  - STORE: N/A

4.7.2 Firmware Class for Device Firmware Loading

- The Loader module starts up the firmware class.
- The firmware class creates two special sysfs objects: loading and data.

  When a ‘1’ is written to the loading object, the firmware class starts the loader (typical paths shown here),

  ```
  echo 1 > /sys/bus/ttsp5/devices/cyttsp5_loader.main_ttsp_core/manual_upgrade
  echo 1 > /sys/class/firmware/cyttsp5_loader.main_ttsp_core/loading
  ```

  The contents of a file can be copied to the firmware class by concatenating a file to the data sysfs.

  ```
  cat <file> > /sys/class/firmware/cyttsp5_loader.main_ttsp_core/data
  ```

  When the “cat” operation is complete, the loader can be stopped by writing ‘0’ to the loading object

  ```
  echo 0 > /sys/class/firmware/cyttsp5_loader.main_ttsp_core/loading
  ```
- When this is done, the firmware class will copy the file contents into a kernel space memory block and invoke a driver callback function with a (u8 *) pointer to the kernel space memory block and a size value indicating how many bytes were copied.
- The format of the loader file will be binary:
  - The content will be according to the existing driver conversion tool.
  - To support OTA, the loader startup sequence uses either an included platform data firmware image or a firmware class to detect if a firmware binary file is saved into the firmware class default directory.

4.8 Data Types

You can develop your own TTSP Device and TTSP Adapter Driver modules in TTDA 3.1 and earlier using the following device and driver structures.

4.8.1 TTSP Device Modules

4.8.1.1 Device Structure

```
struct cyttsp5_device {
        struct list_head node;
        char const *name;
        char const *core_id;
        struct device dev;
        struct cyttsp5_core *core;
};
struct cyttsp5_device_info {
        char const *name;
        char const *core_id;
        void *platform_data;
};
```

4.8.1.2 Driver Structure

```
struct cyttsp5_driver {
        struct device_driver driver;
        int (*probe)(struct cyttsp5_device *dev);
        int (*remove)(struct cyttsp5_device *fev);
};
```

4.8.2 Adapter Device Modules

4.8.2.1 Device Structure

```
struct cyttsp5_adapter {
        struct list_head node;
        char id[NAME_MAX];
        struct device *dev;
        int (*read_default)(struct cyttsp5_adapter *adap,
                void *buf, int size);
        int (*read_default_nosize)(struct cyttsp5_adapter *adap,
                void *buf, int max);
        int (*write_read_specific)(struct cyttsp5_adapter *adap, u8 write_len,
                u8 *write_buf, u8 *read_buf);
};
```

4.8.2.2 Driver Structure

Note: Adapter driver modules use Linux-defined host bus driver structures.

I^{2}C: use Linux struct i2c_driver

SPI: use Linux struct spi_driver

4.9 Control Structures

To interface with the TTDA 3.X Core module, the user modules use the following Core module control structures and Core module interface calls:

4.9.1 Core Module Structures

4.9.1.1 Core Device Structure

```
struct cyttsp5_core {
       struct list_head node;
       char const *name;
       char const *id;
       char const *adap_id;
       struct device dev;
       struct cyttsp5_adapter *adap;
};

struct cyttsp5_core_info {
       char const *name;
       char const *id;
       char const *adap_id;
       void *platform_data;
};
```

4.9.1.2 Core Module Structure

```
struct cyttsp5_core_driver {
       struct device_driver driver;
       int (*probe)(struct cyttsp5_core *core);
       int (*remove)(struct cyttsp5_core *core);
       int (*subscribe_attention)(struct cyttsp5_device *ttsp,
                     enum cyttsp5_atten_type type,
                     cyttsp5_atten_func func,
                     int flags);
       int (*unsubscribe_attention)(struct cyttsp5_device *ttsp,
                     enum cyttsp5_atten_type type,
                     cyttsp5_atten_func func,
                     int flags);
       int (*request_exclusive)(struct cyttsp5_device *ttsp, int timeout_ms);
       int (*release_exclusive)(struct cyttsp5_device *ttsp);
       int (*request_reset)(struct cyttsp5_device *ttsp);
       int (*request_restart)(struct cyttsp5_device *ttsp);
       struct cyttsp5_sysinfo *(*request_sysinfo)(struct cyttsp5_device *ttsp);
       struct cyttsp5_loader_platform_data
              *(*request_loader_pdata)(struct cyttsp5_device *ttsp);
       int (*request_stop_wd)(struct cyttsp5_device *ttsp);
       int (*request_get_hid_desc)(struct cyttsp5_device *ttsp, int protect);
       int (*request_get_mode)(struct cyttsp5_device *ttsp, int protect,
                     u8 *mode);
       int (*write_specific)(struct cyttsp5_device *ttsp, int mode,
              __le16 reg_le, const void *buf, int size);
       int (*read_specific)(struct cyttsp5_device *ttsp, int mode,
              __le16 reg_le, void *buf, int size);
       int (*read_default)(struct cyttsp5_device *ttsp, int mode,
              void *buf, int size);
       int (*write_read_specific)(struct cyttsp5_device *ttsp, u8 write_len,
              u8 *write_buf, u8 *read_buf);
#ifdef TTHE_TUNER_SUPPORT
       int (*request_tthe_print)(struct cyttsp5_device *ttsp, u8 *buf,
                     int buf_len, const u8 *data_name);
#endif
       struct cyttsp5_core_nonhid_cmd *cmd;
};

struct cyttsp5_core_nonhid_cmd {
       int (*start_bl) (struct cyttsp5_device *ttsp, int protect);
       int (*suspend_scanning) (struct cyttsp5_device *ttsp, int protect);
       int (*resume_scanning) (struct cyttsp5_device *ttsp, int protect);
       int (*get_param) (struct cyttsp5_device *ttsp, int protect,
                     u8 param_id, u32 *value);
       int (*set_param) (struct cyttsp5_device *ttsp, int protect,
                     u8 param_id, u32 value);
       int (*verify_config_block_crc) (struct cyttsp5_device *ttsp,
                    int protect, u8 ebid, u8 *status, u16 *calculated_crc,
                    u16 *stored_crc);
       int (*get_config_row_size) (struct cyttsp5_device *ttsp,
                    int protect, u16 *row_size);
       int (*calibrate_idacs) (struct cyttsp5_device *ttsp, int protect,
                    u8 mode);
       int (*initialize_baselines) (struct cyttsp5_device *ttsp, int protect,
                    u8 test_id);
       int (*exec_panel_scan) (struct cyttsp5_device *ttsp, int protect);
       int (*retrieve_panel_scan) (struct cyttsp5_device *ttsp, int protect,
                    u16 read_offset, u16 read_count, u8 data_id, u8 *config,
                    u16 *actual_read_len, u8 *read_buf);
       int (*write_conf_block) (struct cyttsp5_device *ttsp, int protect,
             u16 row_number, u16 write_length, u8 ebid, u8 *write_buf,
             u8 *security_key, u16 *actual_write_len);
       int (*user_cmd)(struct cyttsp5_device *ttsp, int protect,
                    u16 read_len, u8 *read_buf,
                    u16 write_len, u8 *write_buf,
                    u16 *actual_read_len);
       int (*get_bl_info) (struct cyttsp5_device *ttsp, int protect,
                    u8 *return_data);
       int (*initiate_bl) (struct cyttsp5_device *ttsp, int protect,
             u16 key_size, u8 *key_buf, u16 row_size, u8 *metadata_row_buf);
       int (*launch_app) (struct cyttsp5_device *ttsp, int protect);
       int (*prog_and_verify) (struct cyttsp5_device *ttsp, int protect,
                    u16 data_len, u8 *data_buf);
       int (*verify_app_integrity) (struct cyttsp5_device *ttsp, int protect,
                    u8 *result);
};
```

4.9.1.3 TTSP Bus Registration Interface

Core Module registration/unregistration

```
       extern int cyttsp5_register_core_device(
             struct cyttsp5_core_info const *core_info);
       extern int cyttsp5_register_core_driver(struct cyttsp5_core_driver *drv);
       extern void cyttsp5_unregister_core_driver(struct cyttsp5_core_driver *drv);
```

TTSP Device registration/unregistration

```
       extern int cyttsp5_register_device(struct cyttsp5_device_info const *dev_info);
       extern int cyttsp5_unregister_device(char const *name, char const *core_id);

       extern int cyttsp5_register_driver(struct cyttsp5_driver *drv);
       extern void cyttsp5_unregister_driver(struct cyttsp5_driver *drv);
```

4.9.1.4 Core Service Interface

```
       enum cyttsp5_atten_type {
             CY_ATTEN_IRQ,
             CY_ATTEN_STARTUP,
             CY_ATTEN_EXCLUSIVE,
             CY_ATTEN_NUM_ATTEN,
       };

       static inline int cyttsp5_write_specific(struct cyttsp5_device *ttsp, int
             mode,__le16 reg_le, const void *buf, int size)

        static inline int cyttsp5_read_specific(struct cyttsp5_device *ttsp, int
             mode,__le16 reg_le, void *buf, int size)

      static inline int cyttsp5_read_default(struct cyttsp5_device *ttsp, int mode,void
             *buf, int size)

       static inline int cyttsp5_write_read_specific(struct cyttsp5_device *ttsp,
       u8 write_len, u8 *write_buf, u8 *read_buf)

static inline int cyttsp5_adap_write_specific(struct cyttsp5_adapter *adap,
       __le16 reg_le, const void *buf, int size)

static inline int cyttsp5_adap_read_specific(struct cyttsp5_adapter *adap,
       __le16 reg_le, void *buf, int size)

static inline int cyttsp5_adap_read_default(struct cyttsp5_adapter *adap,
       void *buf, int size)

static inline int cyttsp5_adap_read_default_nosize(struct cyttsp5_adapter *adap,void
       *buf, int max)

static inline int cyttsp5_adap_write_read_specific(struct cyttsp5_adapter *adap,u8
       write_len, u8 *write_buf, u8 *read_buf)

static inline int cyttsp5_subscribe_attention(struct cyttsp5_devce *ttsp,
       enum cyttsp5_atten_type type, cyttsp5_atten_func func, int flags)

static inline int cyttsp5_unsubscribe_attention(struct cyttsp5_device *ttsp,
       enum cyttsp5_atten_type type, cyttsp5_atten_func func, int flags)

static inline int cyttsp5_request_exclusive(struct cyttsp5_device *ttsp,
       int timeout_ms)

static inline int cyttsp5_release_exclusive(struct cyttsp5_device *ttsp)

static inline int cyttsp5_request_reset(struct cyttsp5_device *ttsp)

static inline int cyttsp5_request_restart(struct cyttsp5_device *ttsp)

static inline struct cyttsp5_sysinfo *cyttsp5_request_sysinfo(
       struct cyttsp5_device *ttsp)

static inline struct cyttsp5_loader_platform_data
       *cyttsp5_request_loader_pdata(struct cyttsp5_device *ttsp)

static inline int cyttsp5_request_stop_wd(struct cyttsp5_device *ttsp)

static inline int cyttsp5_request_nonhid_user_cmd(struct cyttsp5_device *ttsp,
       int protect, u16 read_len, u8 *read_buf, u16 write_len, u8 *write_buf,
       u16 *actual_read_len)

static inline int cyttsp5_request_nonhid_verify_config_block_crc(
       struct cyttsp5_device *ttsp, int protect, u8 ebid, u8 *status,
       u16 *calculated_crc, u16 *stored_crc)

static inline int cyttsp5_request_nonhid_calibrate_idacs(
       struct cyttsp5_device *ttsp, int protect, u8 mode)

static inline int cyttsp5_request_nonhid_exec_panel_scan(
       struct cyttsp5_device *ttsp, int protect)

static inline int cyttsp5_request_nonhid_retrieve_panel_scan(
       struct cyttsp5_device *ttsp, int protect, u16 read_offset,
       u16 read_count, u8 data_id, u8 *config, u16 *actual_read_len,
       u8 *read_buf)

static inline int cyttsp5_request_nonhid_write_conf_block(
       struct cyttsp5_device *ttsp, int protect, u16 row_number,
       u16 write_length, u8 ebid, u8 *write_buf, u8 *security_key,
       u16 *actual_write_len)
```

```
static inline int cyttsp5_request_nonhid_start_bl(struct cyttsp5_device *ttsp,
             int protect)

static inline int cyttsp5_request_nonhid_suspend_scanning(
             struct cyttsp5_device *ttsp, int protect)

static inline int cyttsp5_request_nonhid_resume_scanning(
             struct cyttsp5_device *ttsp, int protect)

static inline int cyttsp5_request_nonhid_get_bl_info(
             struct cyttsp5_device *ttsp, int protect, u8 *return_data)

static inline int cyttsp5_request_nonhid_initiate_bl(
             struct cyttsp5_device *ttsp, int protect, u16 key_size,
             u8 *key_buf, u16 row_size, u8 *metadata_row_buf)

static inline int cyttsp5_request_nonhid_launch_app(
             struct cyttsp5_device *ttsp, int protect)

static inline int cyttsp5_request_nonhid_prog_and_verify(
             struct cyttsp5_device *ttsp, int protect,
             u16 data_len, u8 *data_buf)

static inline int cyttsp5_request_nonhid_verify_app_integrity(
             struct cyttsp5_device *ttsp, int protect, u8 *result)

static inline int cyttsp5_request_get_hid_desc(struct cyttsp5_device *ttsp, int
       protect)

static inline int cyttsp5_request_get_mode(struct cyttsp5_device *ttsp,
       int protect, u8 *mode)

static inline int cyttsp5_request_tthe_print(struct cyttsp5_device *ttsp, u8 *buf,
       int buf_len, const u8 *data_name)
```

4.10 Application Development Environment

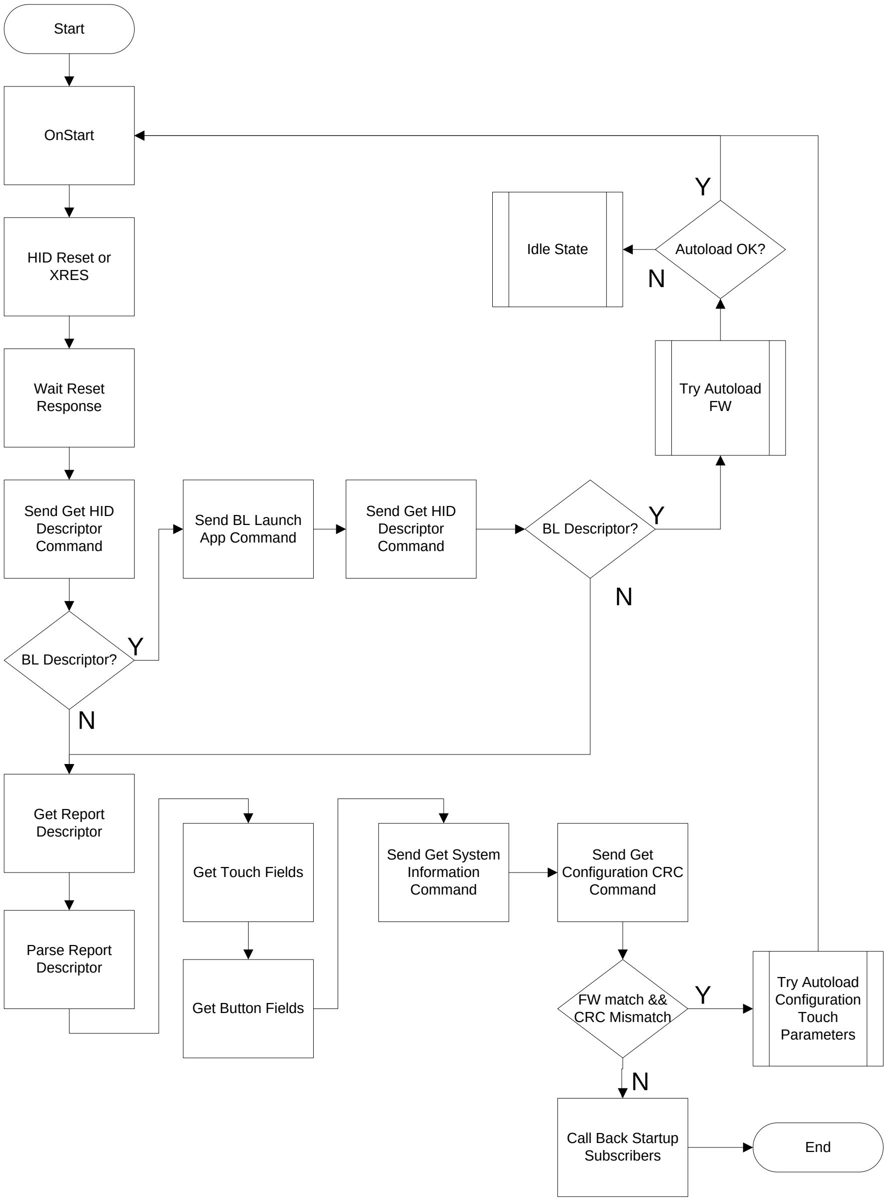
Applications use simple “echo” and “cat” file system interface commands. To write commands, either “echo” a single value or a quoted string and redirect to the file system interface (such as group data). To read the status, perform a “cat” and the status data is displayed at the command line with line separation between each status byte returned.

4.10.1 Characteristics

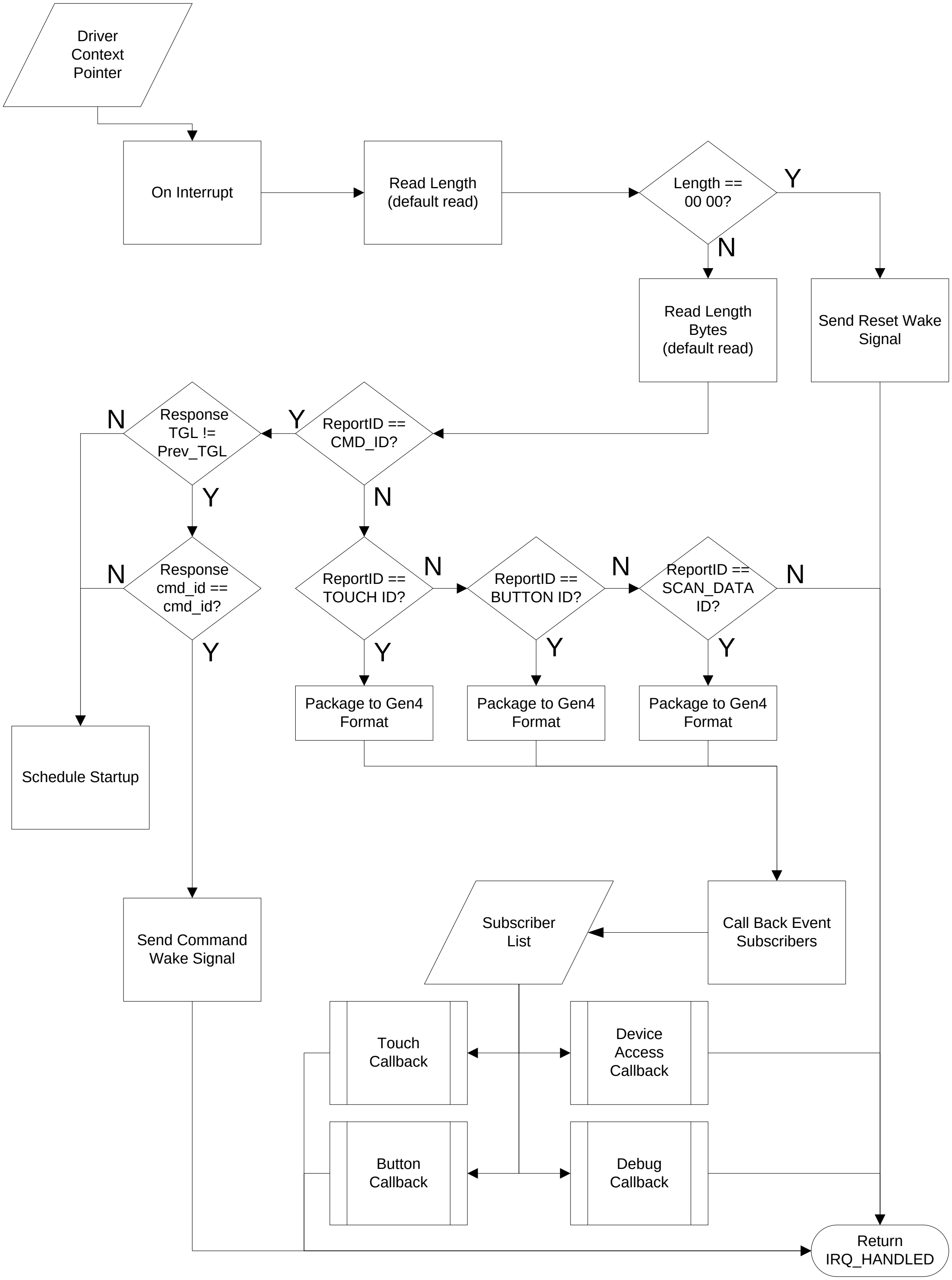
One purpose of the Bus and Module architecture is to allow alternate user modules to be created that can run in parallel with the standard set of modules. The alternate modules can subscribe for the same information. For example, a Debug module can be created that subscribes for interrupt attention and can dump the raw touch data to the command display without changing the normal touch flow.

This chapter illustrates the flow diagrams software design.

4.11 Start Thread


Start
OnStart
HID Reset or XRES
Wait Reset Response
Send Get HID Descriptor Command
BL Descriptor?
Y
N
Send BL Launch App Command
Send Get HID Descriptor Command
BL Descriptor?
Y
N
Try Autoload FW
Autoload OK?
Y
N
Idle State
Get Report Descriptor
Parse Report Descriptor
Get Touch Fields
Get Button Fields
Send Get System Information Command
Send Get Configuration CRC Command
FW match && CRC Mismatch
Y
N
Try Autoload Configuration Touch Parameters
Call Back Startup Subscribers
End

4.12 Interrupt Service Thread



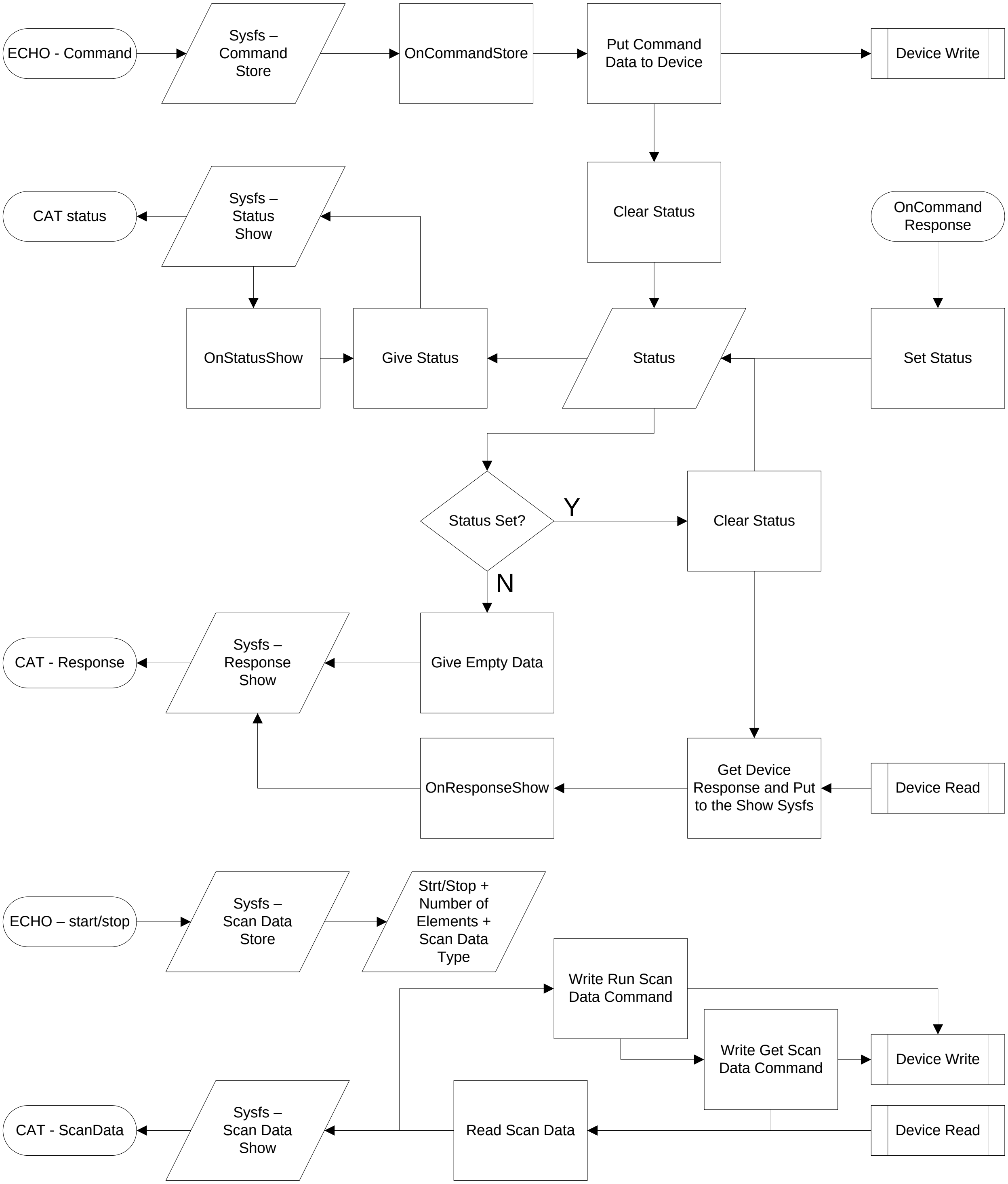
4.13 Loader Flow

```
============
Loader Probe Thread
        request_loader_pdata -> [platform FW, TT_CFG params, AUTO_CALIBRATE flag]
        start FW Upgrade WORK.
END Loader Probe Thread
============
```

```
============
FW Upgrade Thread
        request_sysinfo
        if (platform FW) -> [.h file]
                upgrade = check_version -> [major, minor, revctrl in this order]
        if (upgrade)
                load_firmware()
                goto EXIT

        if (TT_CFG crc modified)
                load_params()
                goto EXIT

#ifdef USE_FW_BIN_FILE
        create_firmware_class
        if (built-in FW) -> [.bin file]
                upgrade = check_version -> [major, minor, revctrl in this order]
        if (upgrade)
                load_firmware()
                goto BIN_EXIT

        if (TT_CFG crc modified)
                load_params()
                goto BIN_EXIT
BIN_EXIT:
        release_fw_class
        goto EXIT
#endif

EXIT:
        request_restart()
END Upgrade Thread
============
============
Load Firmware and Load Params Procedures

load_firmware:
        get_exclusive_access
        request_stop_wd
        mode = request_get_mode
        if (mode == BL)
                start_bl()
        get_bl_info
        get last row metadata
        initiate_bl(metatdata)
        for_each_row except last row
                prog_and_verify(row)
        launch_app()
        if (AUTO_CALIBRATE)
                start Calibrate WORK() after startup -> [STARTUP subscriber for Calibrate]
        release_exclusive_access
return

load_params:
        get_exclusive_access
        for_each_tt_cfg_row
                write_conf_block(row)
        verify_config_block_crc
        resume_scanning
        release_exclusive_access
return
============
```

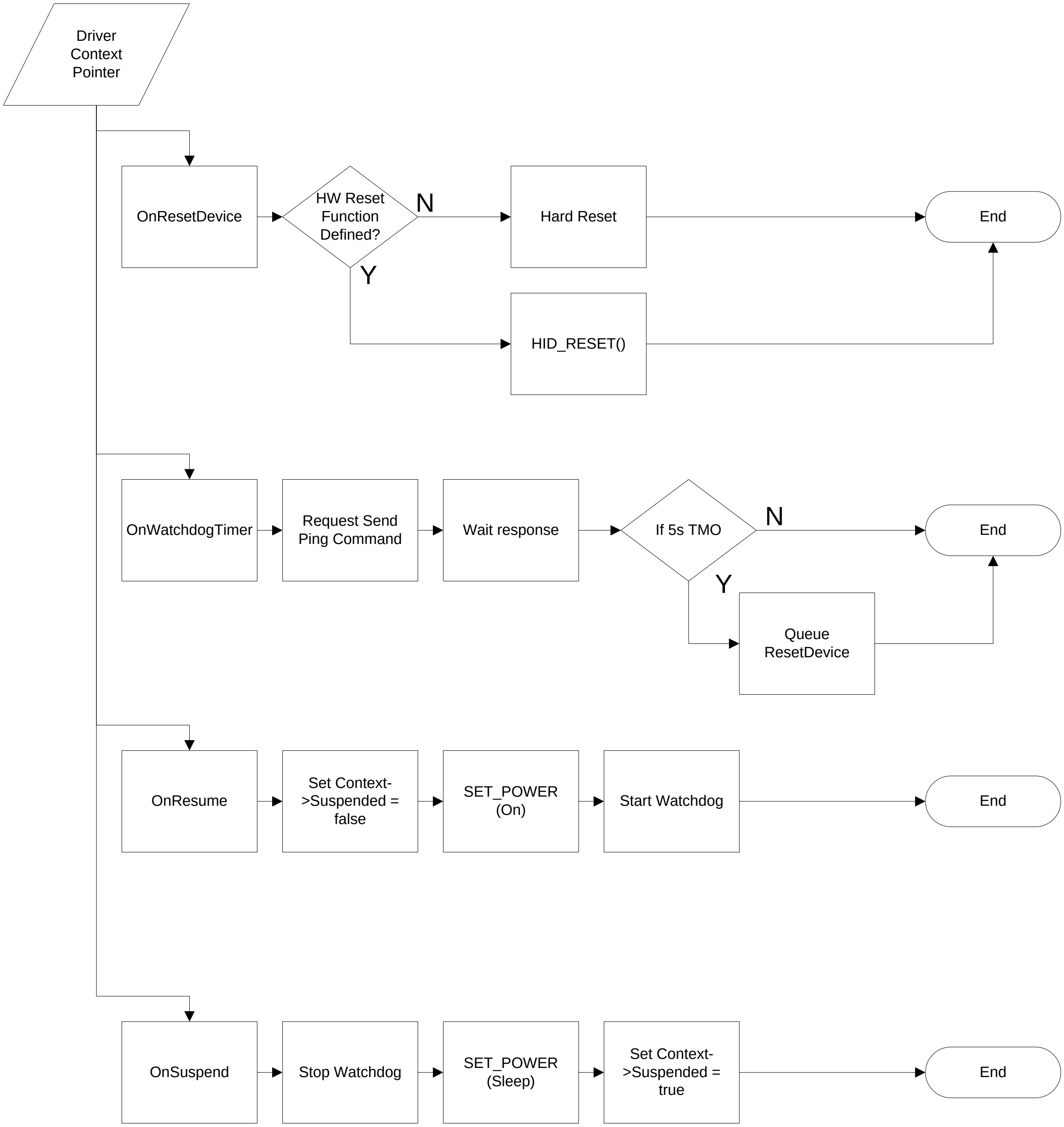
```
============
Calibrate Thread
        get_exclusive_access
        suspend_scanning
        calibrate_idacs
        resume_scanning
        release_exclusive_access
END Calibrate Thread
============

============
User Manual Upgrade Thread
[User input to manual_upgrade sysfs node] echo 1 > manual upgrade
        create_firmware_class
        WAIT_FOR_USER_TO_SUPPLY_FW
        load_firmware()
        request_restart
        release_fw_class
END User Manual Upgrade Thread
============
```

4.14 Device Access Flow

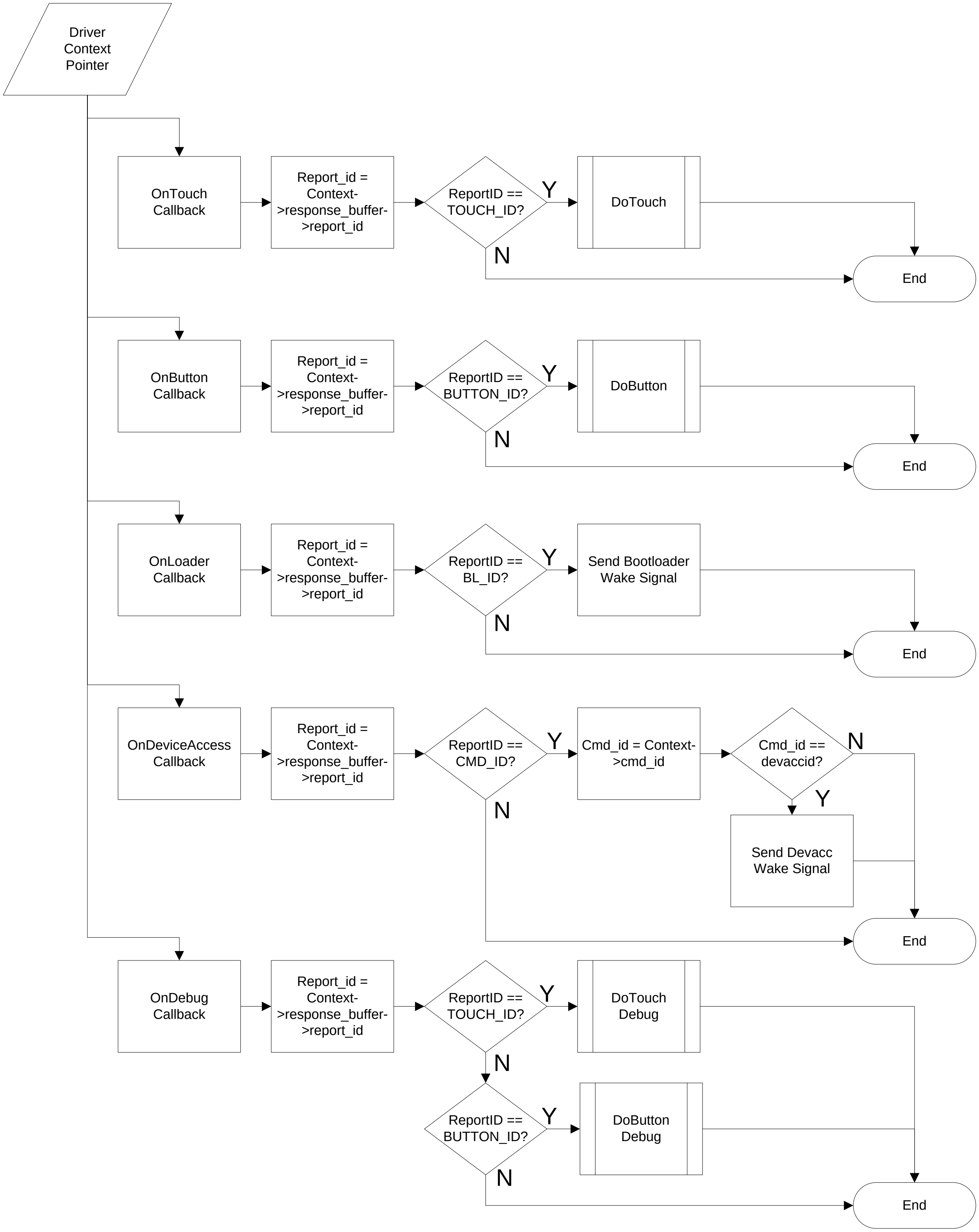


CONFIDENTIAL - RELEASED ONLY UNDER NONDISCLOSURE AGREEMENT (NDA)

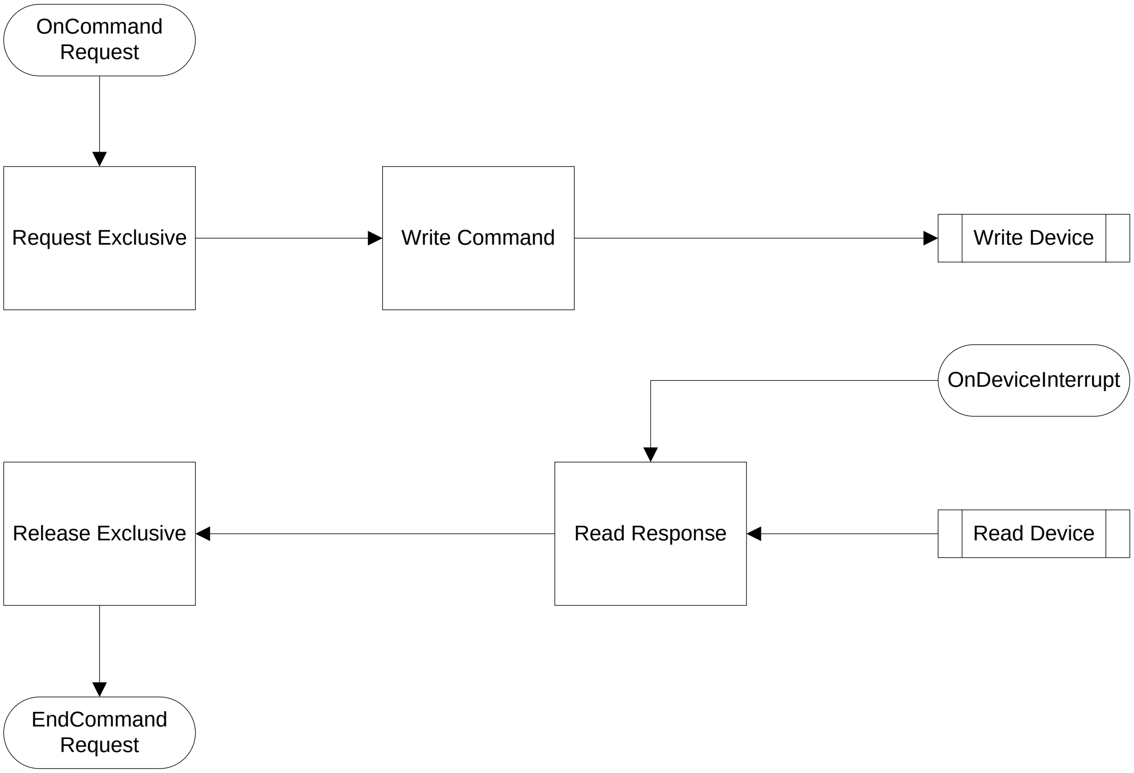
4.15 Thread Flow



4.16 Callback Flow



4.17 Request Command Output



4.18 HID Report Descriptor Parsing

```
At Startup:
        Set Report bit offset = 0
        Foreach touch parameter usage page
                Record the udage page identifier
                Record the current report bit offset
                Record the parameter bit size
                Record the maximum, minimum values
                Report bit offset += bit size

At Touch:
        Touch byte size = Touch,heaer.len - sizeof(Touch.header)) /
Touch.header.NumTouches
        Foreach touch
                For each param that is actually used
                        Find the byte+bit offset and mask for size
                        (note: bits can cross byte boundaries)
```

5. New Features in TTDA 3.0.1

TTDA 3.0.1 is designed and implemented to be compatible with TrueTouch TSG5_L devices. This release is based on TTDA 3.0 and additionally includes support for stylus and device trees.

5.1 Stylus

The multi-touch Protocol A module is updated to report any touch with the touch type of Stylus as BTN_TOOL_PEN. If the touch type is finger, then the BTN_TOOL_FINGER signal is sent. The multi-touch Protocol B module is updated to report any touch with the touch type of Stylus as MT_TOOL_PEN. If the touch type is Finger, then the MT_TOOL_FINGER signal is sent.

Note: For proper stylus performance, ensure that the board configuration has the following line added to the cyttsp5_abs[] table:
`“ABS_MT_TOOL_TYPE, 0, MT_TOOL_MAX, 0, 0,”`

For Device Tree booting method, ensure that the “cy,abs” property has the following string added to the field data: `“0x37 0 1 0 0”`

5.2 Device Tree

To port an existing project over to use the device tree method, refer to the Driver Porting (Device Tree) chapter.

6. New Features in TTDA 3.1

TTDA 3.1 is designed and implemented to be compatible with TrueTouch CYTMA545 devices using the Base.V2 firmware. This release includes four new features: proximity, hover, glove, and easy wakeup.

6.1 Proximity

A new proximity module is created that scans the current touch report for any touch record where the touch type is Proximity. If the touch type is Proximity and the touch ID is Face, then the module signals ABS_DISTANCE with value NEAR (0). If no record has the Proximity touch type and the last touch report had a record with the touch type as Proximity, it indicates that Face has moved away from the touch device. Then the proximity module signals the ABS_DISTANCE with the value FAR (1).

To enable proximity sensing, the `Cypress TrueTouch Gen5 Proximity` module must be selected as built-in in the Linux Kernel Configuration window before compiling the kernel. TTDA 3.1 must be compiled as built-in to allow the Android Sensor Framework recognize the proximity sensor.

In addition, Android requires a board/platform-specific Sensor HAL shared library (`sensors.<boardname>.so` in `/system/lib/hw/`) to make the Android Sensor Framework determine the number of sensor devices and their types, query them, and acquire data from them. The Sensor HAL library converts the Android Sensor Framework requests into sensor device kernel driver-specific requests.

The three steps to get the Android Sensor Framework functional are as follows:

1. Compile Sensor HAL library.
2. Copy Sensor HAL library into the file system.
3. Modify `init.<boardname>.rc` to change SysFs node permissions to make it accessible for Sensor HAL library

Note: The sysfs path for enabling proximity sensing is `/sys/bus/ttsp5/devices/cyttsp5_proximity.main_ttsp_core/enable`.

Write ‘1’ to this sysfs path to enable proximity sensing.

```
# echo 1 > /sys/bus/ttsp5/devices/cyttsp5_proximity.main_ttsp_core/enable
```

Write ‘0’ to this sysfs path to disable proximity sensing.

```
# echo 0 > /sys/bus/ttsp5/devices/cyttsp5_proximity.main_ttsp_core/enable
```

The following instructions describe how to enable proximity sensing for Android Open Source Platform (AOSP) and Linaro with TTDA 3.1 on PandaBoard. Depending on the platform, the Sensor HAL for the target project needs to be implemented accordingly.

- Step 1: Flash CYTMA545/CY8CTMA54X Base.V2 Firmware with support for proximity sensing into the touch IC

Flash the touch IC with CYTMA545/CY8CTMA54X Base.V2 firmware that supports proximity sensing. Proximity sensing is disabled by default. This is necessary to enable 10-finger APA-MC (all points accessible – mutual cap) scanning. Proximity sensing will be enabled and disabled at runtime by TTDA on requests from the Android Sensor HAL.

- Step 2: Compiling TTDA 3.1

Compile TTDA 3.1 as built-in with the `Cypress TrueTouch Gen5 Proximity` module enabled.

Note: TTDA 3.1 must be compiled as ***built-in*** to make the Android Sensor Framework recognize the proximity sensor.

- Step 3: Setting AOSP for proximity sensing

The Android Sensor HAL shard library should be compiled and copied into the file system. The example here applies to PandaBoard only.

Compiling the Sensor HAL shared library with AOSP:

(1) Obtain the Sensor HAL source code.

(2) Copy `device/ti/panda/sensors` into `<aosp>/device/ti/panda` where `<aosp>` is the root AOSP folder.

(3) Add the following two lines to *device/ti/panda/device.mk* file in AOSP.

```
PRODUCT_PACKAGES := \
       sensors.panda
```

(4) Add the following three lines to the *device/ti/panda/init.omap4pandaboard.rc* file in AOSP under `on fs` section after `mount` operations.

```
# change permissions for Cypress Proximity sensor

chmod 0660 /sys/bus/ttsp5/devices/cyttsp5_proximity.main_ttsp_core/enable

chown root system /sys/bus/ttsp5/devices/cyttsp5_proximity.main_ttsp_core/enable
```

(5) Compile the AOSP and load it to PandaBoard using `fastboot`.

6.2 Hover

The Multi-touch Protocol A and B modules are updated to report any touch when the Touch Type is Hover as MT_TOOL_FINGER and sends ABS_MT_PRESSURE signal of 0, regardless of the touch record Z value. The Z value will be reported as ABS_DISTANCE.

6.3 Glove

The multi-touch Protocol A and B modules are updated to report any touch when the touch type is Glove as MT_TOOL_FINGER and sends all other touch signals same as for normal finger.

6.4 Easy Wakeup

The Core module is updated to include the ability for the touch device to wake up the host on recognition of one or more specified gestures. An operational parameter is added to allow selection of either a predefined wakeup gesture (allows device to wake up the host) or use deep sleep (host must wake the device) when the product enters low-power state. A user-to-kernel system interface is added to allow the host to tell the driver what the selection is for the wake gesture or if deep sleep is specified.

- The initial easy wakeup gesture value can be set in `_cyttsp5_core_platform_data` struct in the Linux Board Config file. An example of initializing easy wakeup gesture as 0xF is shown in the following code as part of the Linux Board Config.

Note: Different firmware might have different values for the various easy wakeup gestures.

```
static struct cyttsp5_core_platform_data _cyttsp5_core_platform_data = {
    .irq_gpio = CYTTSP5_I2C_IRQ_GPIO,
    .rst_gpio = CYTTSP5_I2C_RST_GPIO,
    .xres = cyttsp5_xres,
    .init = cyttsp5_init,
    .power = cyttsp5_power,
#ifdef CYTTSP5_DETECT_HW
    .detect = cyttsp5_detect,
#endif
    .irq_stat = cyttsp5_irq_stat,
    .sett = {
          NULL,  /* Reserved */
          NULL,  /* Command Registers */
          NULL,  /* Touch Report */
```

CONFIDENTIAL - RELEASED ONLY UNDER NONDISCLOSURE AGREEMENT (NDA)

```
          NULL,  /* Cypress Data Record */
          NULL,  /* Test Record */
          NULL,  /* Panel Configuration Record */
          NULL,  /* &cyttsp5_sett_param_regs, */
          NULL,  /* &cyttsp5_sett_param_size, */
          NULL,  /* Reserved */
          NULL,  /* Reserved */
          NULL,  /* Operational Configuration Record */
          NULL,   /* &cyttsp5_sett_ddata, *//* Design Data Record */
          NULL,   /* &cyttsp5_sett_mdata, *//* Manufacturing Data Record */
          NULL,  /* Config and Test Registers */
          &cyttsp5_sett_btn_keys,    /* button-to-keycode table */
    },
    .loader_pdata = &_cyttsp5_loader_platform_data,
    .flags = CY_CORE_FLAG_WAKE_ON_GESTURE,
    .easy_wakeup_gesture = CY_CORE_EWG_TAP_TAP
          | CY_CORE_EWG_TWO_FINGER_SLIDE,
};
```

- The easy wakeup gesture flags and macro definitions are in the *cyttsp5_core.h* file.

```
enum cyttsp5_core_platform_flags {
    CY_CORE_FLAG_NONE = 0x00,
    CY_CORE_FLAG_WAKE_ON_GESTURE = 0x01, /* enables device to wake system on gesture */
    CY_CORE_FLAG_POWEROFF_ON_SLEEP = 0x02, /* TTDA3.2+ power off for sleep*/
    CY_CORE_FLAG_RESTORE_PARAMETERS = 0x04, /* TTDA3.3 restore RAM params on restart */
};

enum cyttsp5_core_platform_easy_wakeup_gesture {/* TTDA3.3 */
    CY_CORE_EWG_NONE = 0x00,
    CY_CORE_EWG_TAP_TAP = 0x01,
    CY_CORE_EWG_TWO_FINGER_SLIDE = 0x02,
    CY_CORE_EWG_RESERVED = 0x03,
    CY_CORE_EWG_WAKE_ON_INT_FROM_HOST = 0xFF,
};
```

- The easy wakeup gesture can also be changed using a sysfs interface during runtime. The sysfs path is

```
/sys/bus/ttsp5/devices/main_ttsp_core.cyttsp5_i2c_adapter/easy_wakeup_gesture
```

Note: Setting easy_wakeup_gesture to 0 or 255 disables easy wakeup functionality. Instead, the regular deep-sleep option will be applied.

7. New Features in TTDA 3.2.x

TTDA 3.2 adds a single core driver module to replace the registration of multiple devices with a single device. The Linux Open/Close module implementation for power management replaces the Early Suspend mechanism in previous releases of TTDA. TTDA 3.2.1 includes the new features of TTDA 3.2 in addition to the SWD firmware loader.

7.1 TTDA 3.2 Features

7.1.1 Single Core Driver Module

The Core Driver module combines all the functions of the original Multi-Touch (MTA/B), CapSense Button (BTN), Proximity, Core, and Bus modules. The Core Driver is built as a single module and each adapter (I2C and SPI) is built as a separate module. The feature modules will be buildable as loadable modules however; they will not be registered with separate Linux devices. They will share the same device as the Core Driver module. They will continue to subscribe for event service for interrupt callback notification.

7.1.2 Linux Open/Close module

This module replaces the early suspend and late resume power management with open/close callback functions. This provides ability to power on/off the device instead of putting device into low power on close. The design is to allow for LCD on/off activity to callback the open/close respectively. To fully enable this capability, the input device name must match the expected name in the product build and the target product *ueventd.rc* file must be modified to have the TTDA input device SysFs node enabled with system as owner. For example:

```
"/sys/bus/i2c/devices/4-0024/input/input* enabled 0660 system system"
```

The "pm_runtime" functions will be used to control low-power, based on running module activity. Early suspend will continue to be supported using the Linux-defined conditional compile constant: CONFIG_HAS_EARLYSUSPEND.

7.1.3 Path Name Convention

For TTDA 3.1 and earlier, the path name convention is based on the TTDA bus path for module mounting:

```
/sys/bus/ttsp5/devices/main_ttsp_core.cyttsp5_i2c_adapter/easy_wakeup_gesture
```

For TTDA 3.2 and later, the path name convention is based on the host-bus mounting. For example, for an I2C Host Bus:

```
/sys/bus/i2c/devices/4-0024/easy_wakeup_gesture
```

Where: in this example, the mounting is on the I^{2}C address 0x24 on the host I^{2}C bus number 4.

And for TTDA 3.2 and later, the SPI host bus:

```
/sys/bus/spi/devices/spi1.0/easy_wakeup_gesture
```

Where: in this example, the mounting is on the first (0th) slave select for the SPI bus number 1.

For TTDA 3.2 and later, paths to firmware class sysfs objects (manual file loading for I^{2}C) are:

```
/sys/ bus/i2c/devices/4-0024/manual_upgrade
/sys/class/firmware/4-0024/loading
/sys/class/firmware/4-0024/data
```

7.1.4 Updated Structures

7.1.4.1 Core Platform Data

The following structures show samples of the updated Button Keycode conversion table and the Core Platform Data structure

```
/* Button to keycode conversion */
static u16 cyttsp5_btn_keys[] = {
        /* use this table to map buttons to keycodes (see input.h) */
        KEY_HOMEPAGE,           /* 172 */ /* Previously was KEY_HOME (102) */
                                /* New Android versions use KEY_HOMEPAGE */
        KEY_MENU,               /* 139 */
        KEY_BACK,               /* 158 */
        KEY_SEARCH,             /* 217 */
        KEY_VOLUMEDOWN,         /* 114 */
        KEY_VOLUMEUP,           /* 115 */
        KEY_CAMERA,             /* 212 */
        KEY_POWER               /* 116 */
};

static struct cyttsp5_core_platform_data _cyttsp5_core_platform_data = {
        .irq_gpio = CYTTSP5_I2C_IRQ_GPIO,
        .rst_gpio = CYTTSP5_I2C_RST_GPIO,
        .hid_desc_register = CYTTSP5_HID_DESC_REGISTER,
        .xres = cyttsp5_xres,
        .init = cyttsp5_init,
        .power = cyttsp5_power,
        .irq_stat = cyttsp5_irq_stat,
        .sett = {
                NULL,  /* Reserved */
                NULL,  /* Command Registers */
                NULL,  /* Touch Report */
                NULL,  /* Cypress Data Record */
                NULL,  /* Test Record */
                NULL,  /* Panel Configuration Record */
                NULL,  /* &cyttsp5_sett_param_regs, */
                NULL,  /* &cyttsp5_sett_param_size, */
                NULL,  /* Reserved */
                NULL,  /* Reserved */
                NULL,  /* Operational Configuration Record */
                NULL,  /* &cyttsp5_sett_ddata, *//* Design Data Record */
                NULL,  /* &cyttsp5_sett_mdata, *//* Manufacturing Data Record */
                NULL,  /* Config and Test Registers */
                &cyttsp5_sett_btn_keys,    /* button-to-keycode table */
        },
        .flags = CY_CORE_FLAG_WAKE_ON_GESTURE
                      | CY_CORE_FLAG_RESTORE_PARAMETERS,
};
```

7.1.4.2 Core Driver Commands Structure

- The Core Driver exposes a set of module interface commands to subscriber modules:

```
struct cyttsp5_core_nonhid_cmd {
        int (*start_bl)(struct device *dev, int protect);
        int (*suspend_scanning)(struct device *dev, int protect);
        int (*resume_scanning)(struct device *dev, int protect);
        int (*get_param)(struct device *dev, int protect, u8 param_id,
                      u32 *value);
        int (*set_param)(struct device *dev, int protect, u8 param_id,
                      u32 value);
        int (*verify_config_block_crc)(struct device *dev, int protect,
                      u8 ebid, u8 *status, u16 *calculated_crc,
                     u16 *stored_crc);
        int (*get_config_row_size)(struct device *dev, int protect,
                     u16 *row_size);
        int (*get_data_structure)(struct device *dev, int protect,
                     u16 read_offset, u16 read_length, u8 data_id,
                     u8 *status, u8 *data_format, u16 *actual_read_len,
                     u8 *data);
        int (*run_selftest)(struct device *dev, int protect, u8 test_id,
               u8 write_idacs_to_flash, u8 *status, u8 *summary_result,
               u8 *results_available);
        int (*get_selftest_result)(struct device *dev, int protect,
               u16 read_offset, u16 read_length, u8 test_id, u8 *status,
               u16 *actual_read_len, u8 *data);
        int (*calibrate_idacs)(struct device *dev, int protect, u8 mode,
                     u8 *status);
        int (*initialize_baselines)(struct device *dev, int protect,
                     u8 test_id, u8 *status);
        int (*exec_panel_scan)(struct device *dev, int protect);
        int (*retrieve_panel_scan)(struct device *dev, int protect,
                     u16 read_offset, u16 read_count, u8 data_id,
                     u8 *response, u8 *config, u16 *actual_read_len,
                     u8 *read_buf);
        int (*write_conf_block)(struct device *dev, int protect,
                     u16 row_number, u16 write_length, u8 ebid,
                     u8 *write_buf, u8 *security_key, u16 *actual_write_len);
        int (*user_cmd)(struct device *dev, int protect, u16 read_len,
                     u8 *read_buf, u16 write_len, u8 *write_buf,
                     u16 *actual_read_len);
        int (*get_bl_info)(struct device *dev, int protect, u8 *return_data);
        int (*initiate_bl)(struct device *dev, int protect, u16 key_size,
                     u8 *key_buf, u16 row_size, u8 *metadata_row_buf);
        int (*launch_app)(struct device *dev, int protect);
        int (*prog_and_verify)(struct device *dev, int protect, u16 data_len,
                     u8 *data_buf);
        int (*verify_app_integrity)(struct device *dev, int protect,
                     u8 *result);
};
typedef int (*cyttsp5_atten_func) (struct device *);
struct cyttsp5_core_commands {
        int (*subscribe_attention)(struct device *dev,
                     enum cyttsp5_atten_type type, char id,
                     cyttsp5_atten_func func, int flags);
        int (*unsubscribe_attention)(struct device *dev,
                     enum cyttsp5_atten_type type, char id,
                     cyttsp5_atten_func func, int flags);
        int (*request_exclusive)(struct device *dev, int timeout_ms);
        int (*release_exclusive)(struct device *dev);
        int (*request_reset)(struct device *dev);
        int (*request_restart)(struct device *dev);
        struct cyttsp5_sysinfo * (*request_sysinfo)(struct device *dev);
        struct cyttsp5_loader_platform_data
               *(*request_loader_pdata)(struct device *dev);
        int (*request_stop_wd)(struct device *dev);
        int (*request_get_hid_desc)(struct device *dev, int protect);
        int (*request_get_mode)(struct device *dev, int protect, u8 *mode);
        int (*request_enable_scan_type)(struct device *dev, u8 scan_type);
        int (*request_disable_scan_type)(struct device *dev, u8 scan_type);
        struct cyttsp5_core_nonhid_cmd *nonhid_cmd;
#ifdef TTHE_TUNER_SUPPORT
        int (*request_tthe_print)(struct device *dev, u8 *buf, int buf_len,
                     const u8 *data_name);
#endif
};
```

7.2 TTDA 3.2.1 Features

7.2.1 SWD Firmware Loader

IMPORTANT: TTDA 3.2.1 is a limited distribution, which enables SWD programming in the driver, in addition to bootloading. This distribution of code is only compatible with specific TrueTouch part numbers. Please work directly with a Cypress Design Services (CDS) engineer or Field Application Engineer (FAE) to ensure that all prerequisites are met before using this limited distribution driver. For guidance on prerequisites and usage of the SWD firmware loader feature in TTDA 3.2.1, refer to Cypress specification 001-90764.

The Loader is updated to include the ability to acquire the device via the SWD Interface. The Loader provides SWD word serializer for write operations including writing the acquisition key and writing the load firmware data to the device. The Loader also provides SWD word deserializer to read response data from the device.

The SWD Loader expects a load file in HEX format, so the Loader is updated to read and parse firmware from either a HEX file compiled as an image in the driver or available as a file in the file system and accessible via Linux firmware class.

Implementing SWD Loader on a product requires proper GPIO pin wiring to support the SWD electrical interface. Additionally, the product Linux build environment must allow direct “bit-bang” of the SWD Clock and I/O pins by the SWD Loader. If these requirements cannot be met, then the SWD Loader is not available for that product. Standard TTDA Loader must be used instead, limiting the device firmware to what will fit on the device along with the bootloader firmware.

8. New Features in TTDA 3.3.x

The TTDA 3.3 is based on the TTDA 3.2 (see New Features in TTDA 3.2.). It includes the Manufacturing Test user application interfaces to allow simple scripting to perform the following tests: get Panel Scan data, get IDAC data, run Automatic Shorts test, run Opens test, Calibrate sensors, and Initialize Baselines. Similar to the TTDA 3.2, the TTDA 3.3 is released as TTDA 3.3 and TTDA 3.3.1. The TTDA 3.3.1 release includes a loader module that can be optionally built to use an SWD Loader instead of the standard BL Loader (see the caution note in section 7.2.1 before you plan to use the SWD Loader in your product design).

8.1 TTDA 3.3 Features

8.1.1 Manufacturing Tests

The Manufacturing Tests are provided as sysfs object interfaces that are collected under a common path such as /path/to/mfg_test/auto_shorts (for example, /sys/bus/i2c/devices/4-0024/mfg_test/auto_shorts as described in Path Name Convention). The complete set of added interfaces is:

- `/path/to/mfg_test/panel_scan`
- `/path/to/mfg_test/get_idac`
- `/path/to/mfg_test/auto_shorts`
- `/path/to/mfg_test/opens`
- `/path/to/mfg_test/calibrate`
- `/path/to/mfg_test/baseline`

To understand the data that is returned by the Manufacturing Tests, refer to the Application Note: AN85948 Touchscreen Manufacturing Testing with CYTMA448/545/568.

8.1.1.1 Panel Scan

The Panel Scan test will stop the normal touch scanning using the firmware Stop Scanning command. Then, the test will perform a single scan using the firmware Execute Panel Scan. The test will call the firmware Retrieve Panel Scan for as many times as necessary to extract a complete set of scan data. This data is returned as a single contiguous block of data with the length and format information. The user application can parse this data based on the touch device part type and panel configuration. Finally, the test will call the firmware Resume Scanning command to restore normal touch processing.

8.1.1.1.1 Panel Scan Host Interface

Panel Scan:

```
> echo "ID" > /path/to/mfg_test/panel_scan //where: ID is Data ID for Retrieve Panel Scan
from TRM
                                           // [0=Raw-MC, 1=Base-MC, 2=Diff-MC, etc.]
> cat /path/to/mfg_test/panel_scan
Status S                                  //where: S is [-1=bad command, 0=pass]
== Below data is sent by the driver if Status S is pass ==
S1                                     // where: S1 is retrieve status byte [0=pass, 1=fail]
ID                                        //where: ID is Data ID
EE                                        //where: EE is actual number of elements (7:0)
EE                                        // where: EE is actual number of elements (15:8)
FF                                        // where: FF is format of the data
                                          // (Sign type, matrix, data element size)
DD                                        // where: DD is the data in the same byte order as
read from the device
```

```
DD
:
```

8.1.1.2 Get IDAC Data

The Get IDAC Data test will stop the normal touch scanning using the firmware Stop Scanning command. Then, the test will perform successive calls to the firmware Retrieve Data Structure command. Each call is to get the next block of the data as required until all the data has been read from the device. This data is returned as a single contiguous block of data with the length and format information. The user application can parse this data based on the touch device part type and panel configuration. Finally, the test will call the firmware Resume Scanning command to restore normal touch processing.

8.1.1.2.1 Get IDAC Data Host Interface

Get IDAC:

```
> echo "ID" > /path/to/mfg_test/get_idac     //where: ID is the scan mode data type to 
retrieve
> cat /path/to/mfg_test/get_idac
Status S                                     //Where: S is [-1=bad command, 0=pass]
== Below data is sent by the driver if Status S is pass ==
S1                                    // where: S1 is retrieve status byte [0=pass, 1=fail]
ID                                   // where: ID is the type of scan mode data to retrieve
LL                                  // where: LL is length of returned data (7:0)
LL                                 // where: LL is length of returned data (15:8)
FF                                 // where: FF is format of the data (matrix)
DD                    // where: DD is the data in the same byte order as read from the device
DD
:
```

8.1.1.3 Run Automatic Shorts Test

The Run Automatic Shorts Test will stop the normal touch scanning using the firmware Stop Scanning command. Then the test will perform the firmware Automatic Shorts command. Then the test will call the firmware Retrieve Shorts Data command. If the test summary result is a pass then the summary results are returned to the user application. If the summary results are a fail, then the test will return the summary results along with the actual return data which provides information to identify which sensor(s) is/are shorted. The user application can parse this data based on the product panel configuration. Finally, the test will call the firmware Resume Scanning command to restore normal touch processing.

8.1.1.3.1 Run Automatic Shorts Host Interface

```
Automatic Shorts:
> cat /path/to/mfg_test/auto_shorts
Status S              // Where: S is [-1=bad command, 0=pass]
== Below data is sent by the driver if Status S is pass ==
S1                   // where: S1 is Run Automatic Shorts Self Test Command byte
                     // [0=pass, 1=fail, ff=bad command]
S2                  // where: S2 is test result byte [0=pass, 1=fail, ff=interpret results]
== Below data is sent by the driver if S2 is not pass ==
S3                     // where: S3 is Get Automatic Shorts Self Test Results
                       // Command status byte [0=pass, 1=fail]
ID                    // where: ID is Auto Shorts Self-Test ID (4)
LL                    // where: LL is length of returned data (7:0)
LL                   // where: LL is length of returned data (15:8)
DD                  // where: DD is the data in the same byte order as read from the device
DD
:
```

8.1.1.4 Run Opens Test

The Run Opens Test will stop the normal touch scanning using the firmware Stop Scanning command. Then the test will perform the firmware Opens command. Then the test will call the firmware Retrieve Opens Data command. If the test summary result is a pass then the summary results are returned to the user application. If the summary result is a fail, then the test will return the summary results along with the actual return data which provides information to identify which sensor(s) is/are open. The user application can parse this data based on the product panel configuration. Finally, the test will call the firmware Resume Scanning command to restore normal touch processing.

8.1.1.4.1 Run Opens Test Host Interface

Opens:

```
> cat /path/to/mfg_test/opens
Status S                     // Where: S is [-1=bad command, 0=pass]
== Below data is sent by the driver if Status S is pass ==
S1                          // where: S1 is Run Opens Self Test Command byte
                           // [0=pass, 1=fail, ff=bad command]
S2                        // where: S2 is test result byte [0=pass, 1=non-zero,
ff=interpret results]
== Below data is sent by the driver if S2 is not a pass  ==
S3                      // where: S3 is Get Opens Self- Test Results Command result byte
                       // [0=pass, 1=fail, ff=interpret results]
ID                    // where: ID is Opens Self-Test ID (3)
LL                   // where: LL is length of returned data (7:0)
LL                   // where: LL is length of returned data (15:8)
DD                  // where: DD is the data in the same byte order as read from the device
DD
:
```

8.1.1.5 Run Calibration

The Run Calibration Test will stop the normal touch scanning using the firmware Stop Scanning command. Then the test will perform the firmware Calibration command. If the Initialize Baseline flag is set by the user in the call to this test, then the test will call the firmware Initialize Baseline command. The test will return the pass/fail results for both the Calibrate and Initialize Baseline commands. Finally, the test will call the firmware Resume Scanning command to restore normal touch processing.

8.1.1.5.1 Run Calibration Host Interface

Calibrate:

```
> echo "ID,BB" > /path/to/mfg_test/calibrate // where: ID is the scan mode in the range
                                                                   //   [0=MC, 1=BTN, 2=SC]
                                   // where: BB is Perform baseline initialize flag
                                   //   [0=no-init, 1=Perform]
> cat /path/to/mfg_test/calibrate
Status S                          // Where: S is [-1=bad command, 0=pass]
== Below data is sent by the driver if Status S is pass ==
S1                               // where: S1 is calibration status byte [0=pass, 1=fail]
== Below is valid if Perform Baseline Initialization after Calibration is Perform ==
S2                              // where: S2 is initialize baseline status [0=pass, 1=fail]
```

8.1.1.6 Initialize Baselines Test

The Initialize Baselines Test will stop the normal touch scanning using the firmware Stop Scanning command. Then the test will perform the firmware Initialize Baseline command. The test will return the pass/fail results for Initialize Baseline command. Finally, the test will call the firmware Resume Scanning command to restore normal touch processing.

8.1.1.6.1 Initialize Baseline Host Interface

Init Baselines:

```
> echo "ID" > /path/to/mfg_test/baseline   // where: ID is the scan mode in the range:
                                         // [1=MC, 2=BTN, 3=MC+BTN,4=SC, 5=MC+SC,
                                        // 6=BTN+CS, 7=MC+BTN+SC]
> cat /path/to/mfg_test/baseline
Status S                    // Where: S is [-1=bad command, 0=pass]
== Below data is sent by the driver if Status S is pass ==
S1                         // where: S1 is initialize baseline status byte [0=pass, 1=fail]
```

8.2 TTDA 3.3.1 Features

8.2.1 SWD Firmware Loader

The SWD Loader build is similar to the TTDA 3.2.1 SWD Loader build and is available as TTDA 3.3.1 according to the same restrictions, as defined in TTDA 3.2.1 Features.

9. New Features in TTDA 3.4

The TTDA 3.4 is based on the TTDA 3.3 (see New Features in TTDA 3.3.x). The additional features are:

- Multiple firmware and TT configuration file load capability based on the device reported Panel ID.
- Linux Notifier method (based on platforms that sent notifications for LCD Blank and LCD Unblank.

9.1 TTDA 3.4 Features

9.1.1 Panel ID

The TTDA adds another structural layer over the previous loader structures used to match the required load files to Panel IDs. The mechanism will be backward compatible to allow single load file implementations using the previous one file method that does not use the Panel ID.

9.1.1.1 Bin Files

The new method will provide a standard naming for load files that include a reference to Panel ID. These have a fixed naming convention: for example, for Panel ID 0, the file name to use is cyttsp5_fw00.bin; for Panel ID 1, the file name to use is cyttsp5_fw01.bin; and so on.

9.1.1.2 Included Image Files

To handle the multiple include file images files, the following structures are updated to allow both the original single bin file and the new multiple bin file implementations.

```
/* Embedded Firmware image pointer structure */
struct cyttsp5_touch_firmware {
        const uint8_t *img;
        uint32_t size;
        const uint8_t *ver;
        uint8_t vsize;
        uint8_t panel_id;           /* Panel ID (if no Panel ID then this is set to 0xFF) */
};

/* Embedded TT Config image pointer structure */
struct cyttsp5_touch_config {
        struct touch_settings *param_regs;
        struct touch_settings *param_size;
        const uint8_t *fw_ver;
        uint8_t fw_vsize;
        uint8_t panel_id;           /* Panel ID (if no Panel ID then this is set to 0xFF) */
};

/* Embedded image structure containing pointers to:
 * Either Single image structures or
 * Multiple image structures lists
 */
struct cyttsp5_loader_platform_data {
        struct cyttsp5_touch_firmware *fw;       /* No Panel ID; Firmware image pointer */
        struct cyttsp5_touch_config *ttconfig;   /* No Panel ID; TT Config image pointer */
```

CONFIDENTIAL - RELEASED ONLY UNDER NONDISCLOSURE AGREEMENT (NDA)

```
        struct cyttsp5_touch_firmware **fws;    /* Panel ID; Firmware image list pointer */
        struct cyttsp5_touch_config **ttconfigs;/* Panel ID; TT Config image list pointer */
        u32 flags;
};
```

9.1.2 Linux Notifier Power Management

As an alternative to Early Suspend power management, the Linux Notifier system can be used with some target platforms. In those platforms, the LCD driver can send a notification to subscribing drivers when the LCD is put into blanking and when it is removed from blanking. The TTDA is modified to allow selecting the Notifier method if the Early Suspend method is disabled. To enable Notifier, make sure that the CONFIG_FB is enabled in the default configuration file.

Note: The platform must support CONFIG_FB. The Dragon board supports the Notifier method; however, the Panda board does not.

10. Driver Module Loading

Modules can be built to load automatically or manually. Modules that are compiled as “built-in” are loaded automatically at startup, while loadable kernel modules (LKM) must be loaded manually or with a script. Cypress’s build instructions always make the bus driver “built-in” to ensure that it loads first. The commands in this section are example commands for TI PandaBoard reference platform.

10.1 Automatically Loading Modules

To build automatically loading modules:

1. Under the kernel directory

```
make panda_defconfig
make
```

2. Reboot to load modules.

10.2 Manually Loading Modules

To build manually loading modules:

1. Under the kernel directory

```
make
```

2. Push the generated .ko files to the running Android system using adb command.

10.3 Module Dependencies

10.3.1 TTDA 3.1 and Earlier

There is no dependency for module loading. Modules can be inserted in any order. However, when the device is probed, there is module dependency. MTA/MTB/BTN/Loader/Debug/Device Access modules depend on Core module. Core module depends on I^{2}C/SPI modules. When all are probed, to remove modules, MTA/MTB/BTN/Loader/Debug/Device Access modules need to be removed before Core module can be removed. Similarly, Core module needs to be removed before removing I^{2}C/SPI module. **Note:** In TTDA 3.1, the Proximity module must be compiled as built-in. See the Proximity section for more details.

Table 10-1 lists the modules that must be launched before activating features.

Table 10-1. Activation Dependencies

	Launched Modules							
	Core	**I^{2}C**	**SPI**	**MTA/MTB**	**BTN**	**Loader**	**Debug**	**Proximity**
Device	✓	✓ or	✓					
Touches	✓	✓ or	✓	✓				
CapSense Buttons	✓	✓ or	✓		✓			
Touches and CapSense Buttons	✓	✓ or	✓	✓	✓			
Touch Report Debug Prints	✓	✓ or	✓				✓	
Proximity	✓	✓ or	✓					✓

Module loading/unloading guidelines:

- The bus driver should always be compiled as “built-in.”
- End users will never load/unload driver modules.
- The debug module is not loaded automatically (unless it is compiled as “built-in”).
- Mobile product developers want to be able to load the debug module on any product.
- If OTA firmware is distributed in the kernel build, the loader module is loaded automatically.
- The dynamic load and unload of modules is limited to the multi-touch (MTA/MTB), CapSense button (BTN), and Debug (dbg) user modules.

10.3.2 TTDA 3.2 and Later

The Core Driver module includes the Multi-touch (MTA or MTB), CapSense Button, and IR replacement Proximity functions. The Core module must be loaded first and then any or all of the loadable modules (Loader, Device Access, and Debug Monitor) can be loaded. The loadable modules can be loaded in any order after the Core Driver has been loaded.

11. Error Handling and Debug Interfaces

It is important to consider testability when designing your system software. Proper Linux error handling is required. Whenever a value is returned from any function, the value is tested for error. If an error is returned, the calling function prints an error message to the kernel log.

The kernel log print primitives are: dev_info, dev_err, dev_dbg, and dev_vdbg.

11.1 Dev_info()

Dev_info() is called whenever something is to be printed to the kernel log unconditionally. This includes registration information. The format of the call is:

dev_info(struct device *dev, char *fmt, __func__, arglist…)

where:

struct device is defined by Linux

char *fmt is a format string starting with "%s:" for __func__

arglist are all items specified in *fmt

11.2 Dev_err()

Dev_err() is called for any error condition to provide a record in the kernel log. The format of the call is the same as dev_info(). The kernel log can be printed dynamically for real time monitoring of program flow.

11.3 Dev_dbg()

Dev_dbg() is called to report information to the kernel log to assist in debugging. The format of the call is the same as dev_info(). This is primary debug information and will be limited to allow debug logging without affecting real time performance (such as simple X, Y reporting on each touch). This call is conditionally compiled by placing the following define at the beginning of the code file:

```
#define DEBUG = y
```

11.4 Dev_vdg()

Dev_vdg() is called to report additional information to the kernel log to assist in debugging. This call is the same as dev_dbg() but is conditionally compiled for verbose debug by placing the following define at the beginning of the code file:

```
#define VERBOSE_DEBUG
```

11.5 Watchdog

Driver watchdog is a timer that updates periodically to check for touch device failure and trigger a device reset for recovery if necessary. When an ESD event happens, touch device may not react or malfunction.

In TTDA 3.0, a Linux system timer is used to implement a watchdog to periodically ping the device using the PIP defined operational NULL command. If the device fails to respond, then the driver queues a restart.

To disable the Watchdog, set the constant CY_WATCHDOG_TIMEOUT to 0 in the *cyttsp5_regs.h* file:

```
#define CY_WATCHDOG_TIMEOUT 0
```

To enable watchdog, set the constant CY_WATCHDOG_TIMEOUT to non-zero in the *cyttsp5_regs.h* file. The CY_WATCHDOG_TIMEOUT constant defines the interval before a watchdog reset occurs, in milliseconds (For example, 1000 sets a 1-second interval):

```
#define CY_WATCHDOG_TIMEOUT 1000
```

Section B: Customization and Debugging

This section documents information about TTDA driver customization, including driver porting instructions, methods to enable virtual keys, driver modification for system without XRES signal. It also provides driver debugging information on various ADB commands to perform during product development and evaluation, firmware upgrade instruction, IDAC calibration and open/short tests scripts. Mobile tuner setup instruction and system debugging tools description are also captured in this section.

Section B contains the following chapters:

- Driver Porting (Linux Board Configuration)
- Driver Porting (Device Tree)
- Virtual Keys
- Firmware Updates
- IDAC Calibration and Opens/Shorts Tests
- Driver Without XRES Signal
- System Debugging
- Mobile Tuner

12. Driver Porting (Linux Board Configuration)

This chapter describes the driver porting instructions for projects that employ the Linux Board Configuration option. For platforms that use the Device Tree option, see Driver Porting (Device Tree).

12.1 Overview

TTDA provides easy ways of driver porting using the Linux board configuration option. Most customization in TTDA can be achieved by modifying the Linux board configuration file. The following settings are available in the Linux board configuration file:

- Setting GPIO for I2C/SPI and XRES line
- Init function setting up XRES and IRQ
- Wakeup/sleep functions
- IRQ status function
- Setting up virtual buttons
- Screen size, major axis, minor axis, orientation settings
- Registering Core, MT, and Button devices
- Initializing I^2C or SPI

12.2 TTDA Porting Example

Take the board configuration file */kernel/arch/arm/mach-omap2/board-omap4panda.c* that comes with the TTDA release as an example. The following code pieces show part of the drive porting steps for PandaBoard.

- Setting GPIO for I^2C/SPI and XRES lines

Note GPIO numbers must be different when both I^2C and SPI are on. I^2C/SPI addresses may vary from design to design. Following is an example of the PandaBoard configuration.

```
#ifdef CYTTSP5_USE_I2C
#define CYTTSP5_I2C_NAME “cyttsp5_i2c_adapter”
#define CYTTSP5_I2C_TCH_ADR 0x24
#define CYTTSP5_LDR_TCH_ADR 0x24
#define CYTTSP5_I2C_IRQ_GPIO 38 /* Customization item */
#define CYTTSP5_I2C_RST_GPIO 37 /* Customization item */
#endif
#ifdef CYTTSP5_USE_SPI
#define CYTTSP5_SPI_IRQ_GPIO 38 /* Customization item */
#define CYTTSP5_SPI_RST_GPIO 37 /* Customization item */
#endif
static int cyttsp5_xres(struct cyttsp5_core_platform_data *pdata,
          struct device *dev)
{
   int rst_gpio = pdata->rst_gpio;
   int rc = 0;
   gpio_set_value(rst_gpio, 1);
   msleep(20);
   gpio_set_value(rst_gpio, 0);
   msleep(40);
   gpio_set_value(rst_gpio, 1);
   msleep(20);
   dev_info(dev,
          “%s: RESET CYTTSP gpio=%d r=%d\n”, __func__,
          pdata->rst_gpio, rc);
   return rc;
}
```

- Init function setting up XRES and IRQ

  In `static int cyttsp5_init(struct cyttsp5_core_platform_data *pdata, int on, struct device *dev)` function,

  Initialize XRES pin: `int rst_gpio = pdata->rst_gpio;`

  Initialize IRQ pin: `int irq_gpio = pdata->irq_gpio;`

  Request XRES pin: `gpio_request(rst_gpio, NULL);`

  Set XRES pin as output GPIO: `gpio_direction_output(rst_gpio, 1);`

  Request IRQ pin: `gpio_request(irq_gpio, NULL);`

  Set IRQ pin as input GPIO: `gpio_direction_input(irq_gpio);`

- Wakeup/Sleep functions

  The actual function for wakeup/sleep is cyttsp5_power(), which calls cyttsp5_wakeup() or cytssp4_sleep() depending on the parameter “on”.

```
static int cyttsp5_power(struct cyttsp5_core_platform_data *pdata,
          int on, struct device *dev, atomic_t *ignore_irq)

static int cyttsp5_wakeup(struct cyttsp5_core_platform_data *pdata,
          struct device *dev, atomic_t *ignore_irq)

static int cyttsp5_sleep(struct cyttsp5_core_platform_data *pdata,
          struct device *dev, atomic_t *ignore_irq)
```

- IRQ status function

```
static int cyttsp5_irq_stat(struct cyttsp5_core_platform_data *pdata,
          struct device *dev)
{
   return gpio_get_value(pdata->irq_gpio);
}
```

- Setting up virtual keys

  Set cyttsp5_mt_platform_data properly to enable or disable virtual keys. See Virtual Keys for more details.

```
static struct cyttsp5_mt_platform_data _cyttsp5_mt_platform_data = {
   .frmwrk = &cyttsp5_framework,
   .flags = 0x38,
   .inp_dev_name = CYTTSP5_MT_NAME,
};
```

  Here, the .flags = 0x38 indicates that CY_FLAG_FLIP is enabled, CY_FLAG_INV_X is enabled, CY_FLAG_INV_Y is enabled, and virtual keys CY_FLAG_VKEYS is disabled. To enable virtual keys, the flags need to be set to 0x78 for this configuration. The CY_MT_FLAG_NO_TOUCH_ON_LO tells the driver to kill all touch tracks while Large Object is detected.

```
enum cyttsp5_flags {
    CY_FLAG_NONE = 0x00,
    CY_FLAG_HOVER = 0x04,
    CY_FLAG_FLIP = 0x08,
    CY_FLAG_INV_X = 0x10,
    CY_FLAG_INV_Y = 0x20,
    CY_FLAG_VKEYS = 0x40,
    CY_MT_FLAG_NO_TOUCH_ON_LO = 0x80, /* TTDA3.0.1 and above */
};
```

- Screen size, major axis, minor axis, orientation settings

```
#define CY_MAXX 880
#define CY_MAXY 1280
#define CY_MINX 0
#define CY_MINY 0
```

- Registering Core, MT and Button devices

In `static void __init omap4_panda_cyttsp5_init(void)` function,

```
/* Register core and devices (TTDA3.1 and below) */
cyttsp5_register_core_device(&cyttsp5_core_info);
cyttsp5_register_device(&cyttsp5_mt_info);
cyttsp5_register_device(&cyttsp5_btn_info);
```

- Initializing I2C or SPI

```
/* Initialize muxes for GPIO pins */

#ifdef CYTTSP5_USE_I2C
   omap_mux_init_gpio(CYTTSP5_I2C_RST_GPIO, OMAP_PIN_OUTPUT);
   omap_mux_init_gpio(CYTTSP5_I2C_IRQ_GPIO, OMAP_PIN_INPUT_PULLUP);
#endif
#ifdef CYTTSP5_USE_SPI
   omap_mux_init_gpio(CYTTSP5_SPI_RST_GPIO, OMAP_PIN_OUTPUT);
   omap_mux_init_gpio(CYTTSP5_SPI_IRQ_GPIO, OMAP_PIN_INPUT_PULLUP);
#endif
```

13. Driver Porting (Device Tree)

This chapter describes the driver porting instructions for projects that employ the Device Tree method. For platforms that use the Linux Board Configuration option, see the section Driver Porting (Linux Board Configuration).

13.1 TTDA Porting

Use the following procedure to set up the device tree build for the end device using TTDA 3.X:

- Step 1. Copy TTDA 3.x source files into your kernel directory.
- Step 2. Modify Device Tree files according to the end system.
  - Related files include the *omap4-panda.dts* file in the *…\kernel\arch\arm\boot\dts\omap4-panda.dts* directory and the *cyttsp5_platform.c* file in the *…\kernel\drivers\input\touchscreen\cyttsp5_platform.c* directory.
  - The *omap4-panda.dts* file is an example Device Tree file that contains hardware-specific configuration of the Cypress driver development platform PandaBoard. To set up a build for a different platform, you should create a similar device tree file that is specific to the hardware configuration of the platform used.
  - The *cyttsp5_platform.c* file contains platform-specific functions such as XRES, INIT, and Power. It also contains loader platform data that includes .h firmware, TT_CONFIG, and loader flags.

    You should modify such platform specific functions according to your design.

    If automatic firmware upgrade or TT_CONFIG update is required, make sure to `#include` the firmware header file *cyttsp5_img.h* and/or TT_CONFIG header file *cyttsp5_params.h* in the *cyttsp5_platform.c* file.
- Step 3. Enable Device Tree support in Linux menuconfig.
  - In Linux menuconfig, in addition to other necessary options, select the **Enable Device Tree support** item under “Cypress TrueTouch Gen5 Touchscreen Driver.”
- Step 4. Compile the kernel.
  - Compile the kernel and build an image for the end device.

14. Virtual Keys

14.1 Virtual Key Overview

In mobile touchscreen applications, the active area describes the part of the touch device that actually covers the display. The system automatically reports touches within the active area. You can use parts of the touchscreen that are outside the active area as virtual keys. When you map virtual keys, you configure Android to process touches beyond the active area and parse them as key events. Hardware buttons and virtual keys look the same to applications. This process is described as "performing the mapping from touch coordinates to key codes in software" on the website. The virtual key mapping for the TTDA is hard coded with the following parameters:

- Screen resolution: X=1280, Y=720
- Touchscreen resolution: X=720, Y=1440
- Screen orientation: Flags in the board file are: CY_FLAG_FLIP(0x08), CY_FLAG_INV_X(0x10), CY_FLAG_INV_Y(0x20). Set one or more of these flags to swap X, Y axis, invert X axis, and/or invert Y axis.
- Virtual Key enable: The flag in the board file is CY_MT_FLAG_VKEYS(0x40). Add this flag to the orientation flag settings above to enable Virtual Keys (For example, 0x78 has all orientation flags and the virtual key flag set).

14.2 Virtual Key Map

The virtual key map is presented as a board property and embedded in the board configuration source file. It is visible on the board sysfs and has the following syntax:

0x01 <Linux key code> <centerX> <centerY> <width> <height>

where:

0x01: The version code (must always be 0x01)
<Linux key code> : The Linux key code of the virtual key

<centerX>: The X pixel coordinate of the center of the virtual key

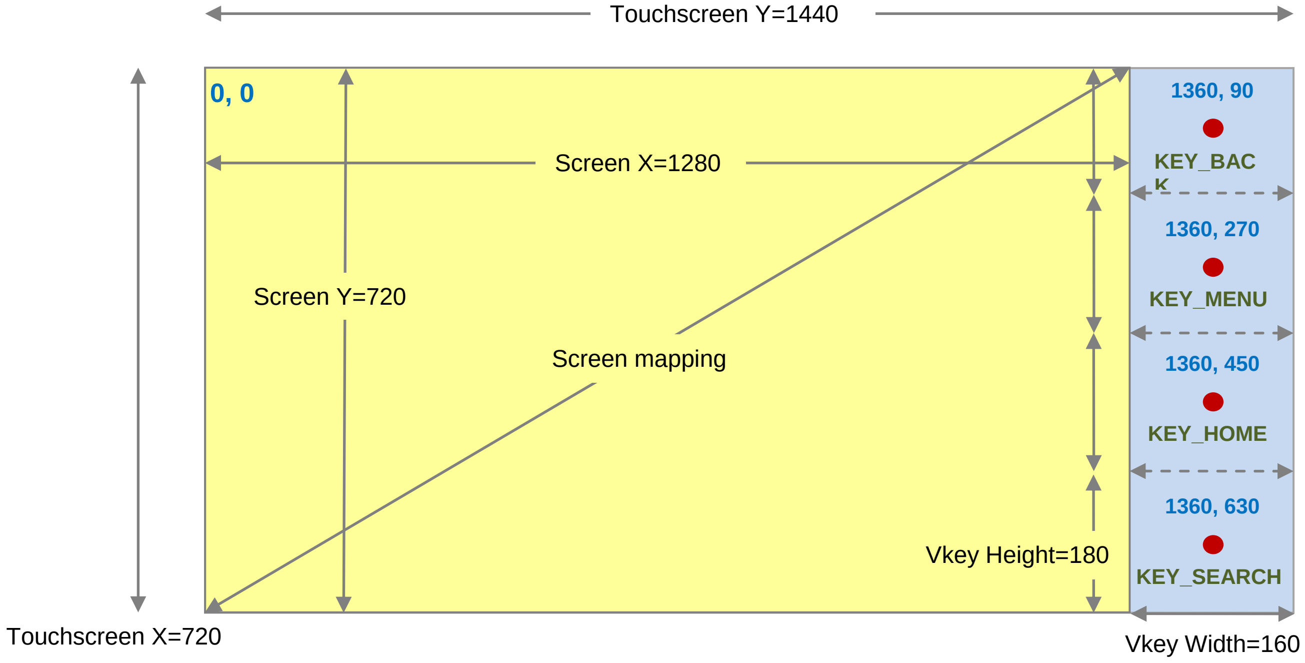
<centerY>: The Y pixel coordinate of the center of the virtual key

<width>: The width of the virtual key in pixels

<height>: The height of the virtual key in pixels

Note The virtual keys mapping file */sys/board_properties/virtualkeys.cyttsp5_mt* (TTDA 3.X) is a runtime-generated object. You can modify the name of this object by modifying the board configuration source file *kernel/arch/arm/mach-omap2/board-omap4panda.c.* The name of this file should be set the same as the Input device name *(cyttsp5_mt* is the input device name in the driver files (reference the macro #define CYTTSP5_MT_NAME "cyttsp5_mt"). The system will look for the key map and key layout files using this name *(cyttsp_mt.kcm, cyttsp5_mt.kl).* You can modify the content of these files according to the mapping of the product build.

Figure 14-1. Example Virtual Key Mapping



For the touchscreen application shown in Figure 14-1, the virtual key map is:

0x01:158:1360:90:160:180:0x01:139:1360:270:160:180:0x01:102:1360:450:160:180:0x01:217:1360:630:160:180

14.3 Implementing Virtual Keys

If you use Device Tree, code changes that modify flag values only within the Device Tree file require rebuilding only the Device Tree. All other changes to driver and/or board configuration code require that the driver and kernel are rebuilt.

1. Update the board configuration source file with the virtual key mapping.

   The board configuration source file for the PandaBoard is *kernel/arch/arm/mach-omap2/board-omap4panda.c.*

   Reduce the active area of the touchscreen reported to Android to leave room for the virtual keys. For example, change *_cyttsp5_mt_platform_data.flags* from 0x38 to 0x78 (set the CY_MT_FLAG_VKEYS virtual keys enable).

   This will reduce the maximum resolution of the touchscreen by an area equal to CY_VKEYS_X by CY_VKEYS_Y. These parameters are specified in the sample board configuration file (*kernel/arch/arm/mach_omap2/board-omap4panda.c*) and can be set by the defines CY_MAXX and C_MAXY respectively. For the device tree method, the values are coded into the vkeys_x and vkeys_y fields respectively in the device tree file (*kernel/arch/arm/boot/dts/omap4_panda.dts*).

   Note: If Virtual Keys are not enabled, then the driver will use the maximum X and Y values as defined in the device firmware System Information for X, Y resolution.

2. Update the touchscreen resolution in firmware using the TTHE.

   RES_X = 720

   RES_Y = 1440

3. Update the board configuration with the key layout and key character map files.

   Copy <*development_sandbox*>*/samples/cyttsp5_mt.kl* to board: */system/usr/keylayout/cyttsp5_mt.kl*

   Copy <*development_sandbox*>*/samples/cyttsp5_mt.kcm* to board: */system/usr/keychars/cyttsp5_mt.kcm*

   Use ADB to push these configuration files to the board:

   ```
   cd <development_sandbox>/samples
   ```

```
adb remount
adb push cyttsp5_mt.kl  /system/usr/keylayout/cyttsp5_mt.kl
adb push cyttsp5_mt.kcm  /system/usr/keychars/cyttsp5_mt.kcm
```

15. Firmware Updates

The TTDA provides an interface to load a new firmware image into a TTSP device. You need the firmware <*project*>*.bin* file or the <*project*>*.h* file generated by the TTHE tool for either method.

These methods describe how to update the firmware image. If you flashed the TTSP device and have not made any changes, you do not have to update the firmware.

15.1 Remote Host

To update the firmware manually from a remote host, you will need to have the ADB utility installed and serial debugging enabled in your device. Execute the following commands using a command window or script.

echo Start data transfer

```
adb shell “echo 1 > /sys/bus/ttsp5/devices/cyttsp5_loader.main_ttsp_core/manual_upgrade”
```

```
adb shell “echo 1 > /sys/class/firmware/cyttsp5_loader.main_ttsp_core/loading”
sleep 1
```

push firmware (for example, the Firmware.bin file) to the device (typically in /data directory)

```
adb push FW.bin /data
```

echo Transfer data

```
adb shell “cat /data/Firmware.bin > /sys/class/firmware/cyttsp5_loader.main_ttsp_core/data”
sleep 1
```

echo Initiate flashing

```
adb shell “echo 0 > /sys/class/firmware/cyttsp5_loader.main_ttsp_core/loading”
```

15.2 Automatic OTA Updates

TTDA can update the firmware automatically based on revision information. The firmware to be used during the upgrade is built into the kernel that will be used during an OTA upgrade. When the Android device receives the new kernel, TTDA upgrades the firmware automatically. After the firmware is upgraded, TTDA will not upgrade the firmware again until the Android device receives another new kernel.

15.2.1 Kernel Upgrade

Both the <*project*>*.h* and <*project*>*.bin* files are generated by the Cypress TrueTouch Host Emulator (TTHE) tool. If you do not know the two-byte major/minor version, inspect the <*project*>*.bin* file in a hexadecimal viewer. The major/minor version is the second and third byte of the file.

When new firmware is available for upgrade, build the firmware .h file into the kernel so firmware upgrade will be performed at the next device reboot.

In loading the new firmware fails, the TTDA will try again for a specific number of times. If the firmware is not successfully loaded before this number of retries, then the TTDA will stop trying and quietly wait for a restart or a manual upgrade attempt. To reset the TTDA to retry the automatic updates, repower the product in order to reload the driver. The maximum number of retries can be set by a constant in the *cyttsp5_loader.c* file (it is set at default of 3):

```
#define CYTTSP5_LOADER_FW_UPGRADE_RETRY_COUNT 3
```

15.2.2 ADB Push

For firmware upgrade using .bin file, a .bin file needs to be built into the kernel once. For later upgrades, user can either rebuild the kernel with new .bin file, or ADB push the new .bin file to */system/etc/firmware* (typical) directory. The firmware upgrade will be executed in the next device reboot.

15.2.3 Board Configuration File Change

Make the following changes to the Linux Board Configuration file:

Add the following code to the board configuration file before the cyttsp5 platform data is declared

Note: The loader flags must be set in the *cyttsp5_platform.c* file.

```
#ifdef CONFIG_TOUCHSCREEN_CYPRESS_CYTTSP5_PLATFORM_FW_UPGRADE
#include "cyttsp5_fw.h"
static struct cyttsp5_touch_firmware cyttsp5_firmware = {
        .img = cyttsp5_img,
        .size = ARRAY_SIZE(cyttsp5_img),
        .ver = cyttsp5_ver,
        .vsize = ARRAY_SIZE(cyttsp5_ver),
};
#else
static struct cyttsp5_touch_firmware cyttsp5_firmware = {
        .img = NULL,
        .size = 0,
        .ver = NULL,
        .vsize = 0,
};
#endif
```

Update the platform data structure for the core to include the fw field:

```
static struct cyttsp5_core_platform_data _cyttsp5_core_platform_data = {
        .irq_gpio = CYTTSP5_I2C_IRQ_GPIO,
        .rst_gpio = CYTTSP5_I2C_RST_GPIO,
        .hid_desc_register = CYTTSP5_HID_DESC_REGISTER,
        .xres = cyttsp5_xres,
        .init = cyttsp5_init,
        .power = cyttsp5_power,
       .detect = cyttsp5_detect, /* TTDA3.3 */
        .irq_stat = cyttsp5_irq_stat,
        .sett = {
            NULL, /* Reserved */
            NULL, /* Command Registers */
            NULL, /* Touch Report */
            NULL, /* Cypress Data Record */
            NULL, /* Test Record */
            NULL, /* Panel Configuration Record */
            NULL, /* &cyttsp5_sett_param_regs, */
            NULL, /* &cyttsp5_sett_param_size, */
            NULL, /* Reserved */
            NULL, /* Reserved */
            NULL, /* Operational Configuration Record */
            NULL, /* &cyttsp5_sett_ddata, *//* Design Data Record */
            NULL, /* &cyttsp5_sett_mdata, *//* Manufacturing Data Record */
            NULL, /* Config and Test Registers */
            &cyttsp5_sett_btn_keys, /* button-to-keycode table */
       },
       .loader_pdata = &_cyttsp5_loader_platform_data, /* TTDA3.1 */
       .flags = 0x0, /* Core flags */
       .easy_wakeup_gesture = 0x00,  /* TTDA3.3 */
};
```

Update loader platform data and set the flags properly. This example indicates that automatic calibration must be done following a firmware load.

```
static struct cyttsp5_loader_platform_data _cyttsp5_loader_platform_data = {
   .fw = &cyttsp5_firmware,
   .ttconfig = &cyttsp5_ttconfig,
   /* Sample shows Loader flags set for calibrate after FW load */
   .flags = CY_LOADER_FLAG_CALIBRATE_AFTER_FW_UPGRADE,
};
```

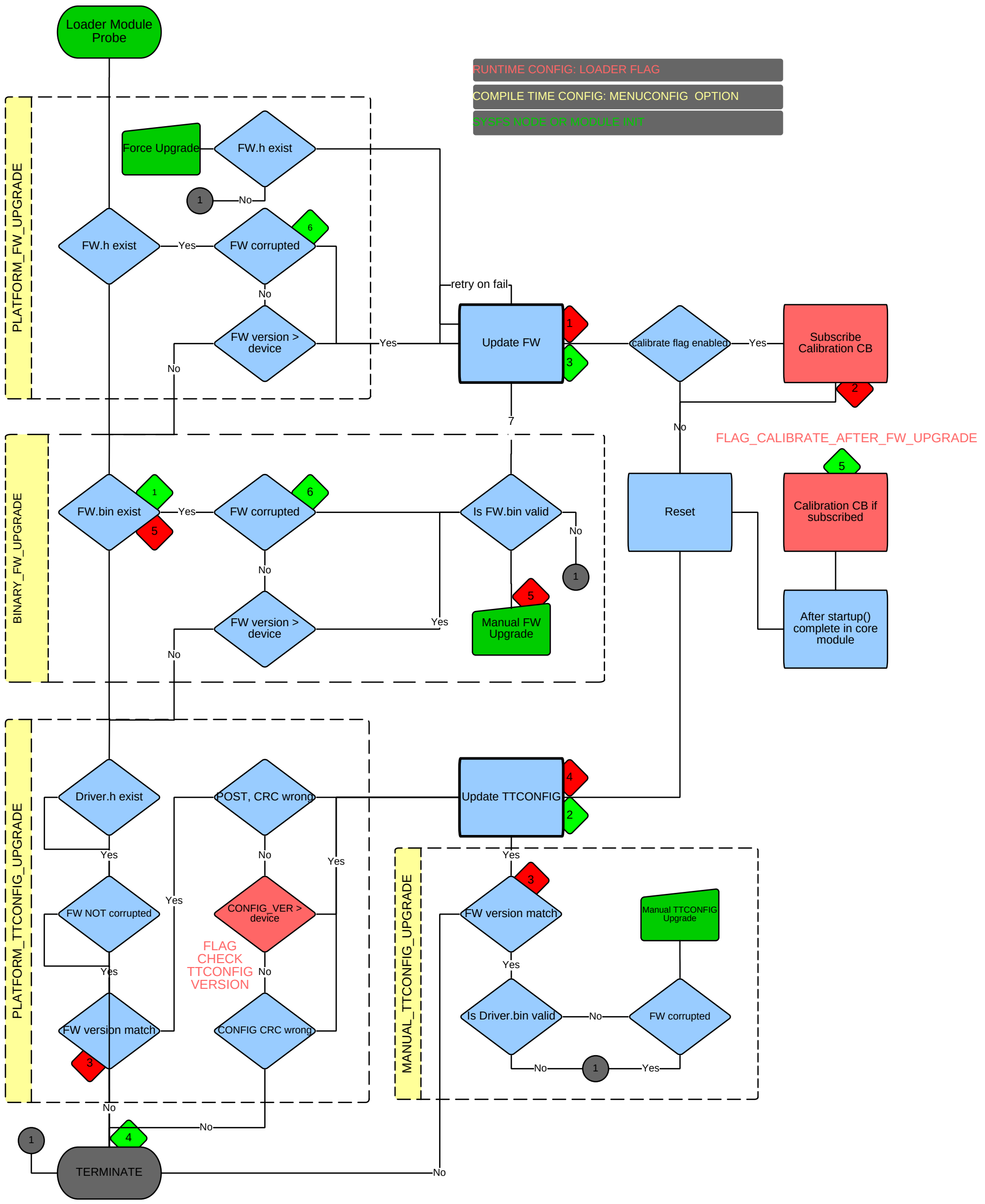
The following flags are used in the loader.

```
enum cyttsp5_loader_platform_flags {
    CY_LOADER_FLAG_NONE = 0x00,
    CY_LOADER_FLAG_CALIBRATE_AFTER_FW_UPGRADE = 0x01,
    /* Use CONFIG_VER field in TT_CFG to decide TT_CFG update */
    CY_LOADER_FLAG_CHECK_TTCONFIG_VERSION, /* TTDA3.3 */
    CY_LOADER_FLAG_CALIBRATE_AFTER_TTCONFIG_UPGRADE, /* TTDA3.4 */
};
```

CAUTION: The CY_LOADER_FLAG_CALIBRATE_AFTER_FW_UPGRADE must be used carefully. It allows a product to be configured to automatically perform a calibration after the firmware is upgraded. The product environment cannot be controlled when the updates are performed and it is not recommended to perform calibrations while fingers or other objects are contacting the touch sensor. It is recommended to use this selection in a factory environment only. Also, note that this automatic calibration requires that the firmware is upgraded. TT configuration only updates are not covered by this setting.

RECOMMENDATION: To test calibration in a factory environment, use an ADB connection to either use the general Device Access interface to script a calibration command (see IDAC Calibration and Opens/Shorts Tests) or, for TTDA3.3, use the calibration command as described in Run Calibration Host Interface. The TTHE Mobile Tuner tool can be used to connect to the product and perform the calibrations.

Figure 15-1 - Rules for Automated Firmware and TT Configuration Updates



🟥 1	"%s: Firmware update failed with error code %d\n"
🟥 2	"%s: No calibration will be performed\n"
🟥 3	"%s: FW versions mismatch\n"
🟥 4	"%s: CRC Failed, ebid=%d, status=%d, scrc=%X ccrc=%X\n"
🟥 5	"%s: Fail request firmware class file load\n"

🟩 1	"%s: Found firmware\n"
🟩 2	"%s: TT_CFG updated: rows:%d bytes:%d\n"
🟩 3	"%s: start load app\n" "%s: Send BL Loader Enter\n" "%s: Send BL Loader Blocks\n" "%s: Send BL Loader Terminate\n" "%s: APP Checksum Verified\n"
🟩 4	"%s: No FW upgrade method selected!\n"
🟩 5	"%s: Calibration Done\n"
🟩 6	"%s: No firmware infomation found, device FW may be corrupted\n"

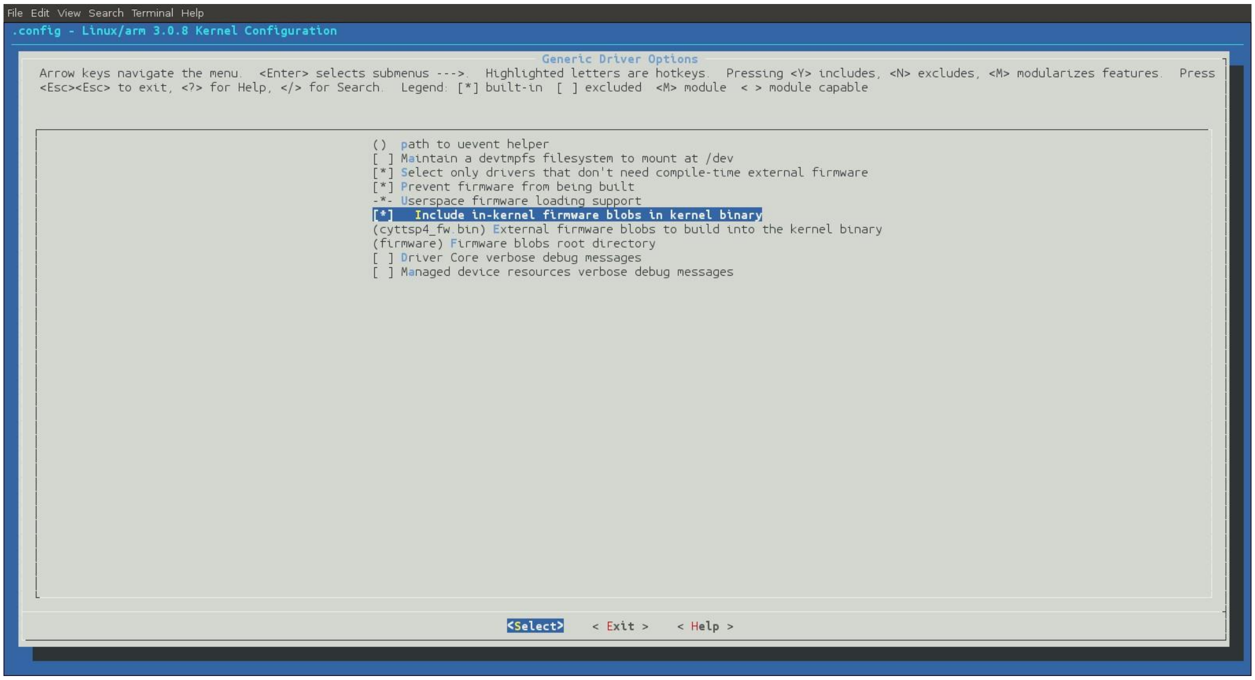
15.2.4 Create a New Kernel with *<project>.bin* File

The following steps describe how to create a new kernel:

1. Enable CONFIG_FIRMWARE_IN_KERNEL kernel configuration.

   Call “make menuconfig” and browse to **Device Drivers > General Driver Options**. Press <Y> to include **[*] Cypress TrueTouch Gen5 Multitouch Loader Press** <Y> to include **[*] Include in-kernel firmware blobs in kernel binary,** as shown in Figure 15-1**.**

Figure 15-1. General Driver Options in menuconfig



```
.config - Linux/arm 3.0.8 Kernel Configuration

                                   Generic Driver Options
 Arrow keys navigate the menu.  <Enter> selects submenus --->.  Highlighted letters are hotkeys.  Pressing <Y> includes, <N> excludes, <M> modularizes features.  Press
 <Esc><Esc> to exit, <?> for Help, </> for Search.  Legend: [*] built-in  [ ] excluded  <M> module  < > module capable

                          ()  path to uevent helper
                          [ ] Maintain a devtmpfs filesystem to mount at /dev
                          [*] Select only drivers that don't need compile-time external firmware
                          [*] Prevent firmware from being built
                          -*- Userspace firmware loading support
                          [*]   Include in-kernel firmware blobs in kernel binary
                          (cyttsp4_fw.bin) External firmware blobs to build into the kernel binary
                          (firmware) Firmware blobs root directory
                          [ ] Driver Core verbose debug messages
                          [ ] Managed device resources verbose debug messages

                             <Select>    < Exit >    < Help >
```

2. Enter the firmware name *cyttsp5_fw.bin* in the CONFIG_EXTRA_FIRMWARE kernel configuration option.

   From the same General Driver Options window, enter **(cyttsp5_fw.bin) External firmware blobs to build into the kernel binary,** as shown in Figure 15-1.

   Note: The file name is hard-coded to “*cyttsp5_fw.bin*.”

3. Confirm that the CONFIG_EXTRA_FIRMWARE_DIR option is set to firmware.

   From the same General Driver Options window, **(firmware) Firmware blobs root directory** should be the default value, as shown in Figure 15-1.

4. Copy the firmware file into the kernel source directory.

   Copy the firmware *.bin* file into to the *<kernel>/firmware/* subdirectory, and give it the name *cyttsp5_fw.bin .*

   The file and subdirectory names must match the ones in the previous steps. You can use a different name for the kernel subdirectory; however, the firmware file name must be *cyttsp5_fw.bin*.

5. Compile the new kernel.

   Use the “make” command in the kernel source directory to compile the new kernel. Your new firmware will be included in the resulting kernel.

   If the kernel compiled correctly, the following lines should be added to kernel compile log:

```
CHK     include/linux/version.h
CHK     include/generated/utsrelease.h
```

```
make[1]: `include/generated/mach-types.h' is up to date.
CALL    scripts/checksyscalls.sh
CHK     include/generated/compile.h
AS      firmware/cyttsp5_fw.bin.gen.o
LD      firmware/built-in.o
LD      vmlinux.o
MODPOST vmlinux.o
GEN     .version
CHK     include/generated/compile.h
UPD     include/generated/compile.h
CC      init/version.o
LD      init/built-in.o
LD      .tmp_vmlinux1
KSYM    .tmp_kallsyms1.S
AS      .tmp_kallsyms1.o
LD      .tmp_vmlinux2
KSYM    .tmp_kallsyms2.S
AS      .tmp_kallsyms2.o
LD      vmlinux
```

If the firmware file was not included when the kernel compiled, it will generate a compile error similar to this:

```
CALL    scripts/checksyscalls.sh
CHK     include/generated/compile.h
make[1]: *** No rule to make target `firmware/cyttsp5_fw.bin', needed by
`firmware/cyttsp5_fw.bin.gen.o'. Stop.
make: *** [firmware] Error 2
```

15.2.5 Testing the OTA Feature Using the New Kernel

1. Boot with the new kernel.

   Put the new kernel into the SD card boot partition using fastboot, adb, or linaro tools.

2. Load the *cyttsp5_loader.ko* module.

   Loading the *cyttsp5_loader.ko* module will trigger a firmware upgrade if the firmware revision in the SD card kernel is newer than the TTSP device's current firmware. If the *cyttsp5_loader.ko* module is compiled as "built-in", an upgrade will be triggered automatically during boot up.

   To observe loader messages, *cyttsp5_loader.ko* should be compiled with debug messages enabled.

   Using the host system command line, give following commands:

   For TMA5xx devices using TTDA 3.X,

```
insmod cyttsp5_i2c.ko

insmod cyttsp5_core.ko

insmod cyttsp5_loader.ko
```

   If the firmware revision in the TTSP device and the firmware revision in the SD card kernel are not the same, and the version of the firmware revision in the SD card kernel is newer than the revision in the TTSP device then loader will try to update the firmware in the TTSP device. If the load is successful, the following information should be printed into the kernel log:

```
cyttsp5_load_app: Send BL Loader Enter

cyttsp5_load_app: Send BL Loader Blocks

cyttsp5_load_app: Send BL Loader Terminate
```

   If all of these prints appear, then there should be a successful load of the new firmware, otherwise it is possible that a failure occurred.

15.2.6 Summary Menuconfig System Generated Defines

Automatic Defines Created by Menuconfig Selections	Description	TTDA 3.1	TTDA 3.2	TTDA 3.3	TTDA 3.4
CONFIG_TOUCHSCREEN_CYPRESS_ CYTTSP5_DEVICETREE_SUPPORT	Use Device Tree instead of standard Board Configuration for Platform Data	✓	✓	✓	✓
CONFIG_TOUCHSCREEN_CYPRESS_ CYTTSP5_PLATFORM_FW_UPGRADE	Enable Automatic FW upgrades using Platform Data included FW image array	✓	✓	✓	✓
CONFIG_TOUCHSCREEN_CYPRESS_ CYTTSP5_PLATFORM_TTCONFIG_UPGRADE	Enable Automatic TT Configuration upgrade using Platform Data included TT Configuration array	✓	✓	✓	✓
CONFIG_TOUCHSCREEN_CYPRESS_ CYTTSP5_BINARY_FW_UPGRADE	Enable Automatic FW upgrade using binary file (This is also required if TTHE_TUNER_SUPPORT is defined)	✓	✓	✓	✓
CONFIG_TOUCHSCREEN_CYPRESS_ CYTTSP5_MANUAL_TTCONFIG_UPGRADE	Enable manual upgrade of TT Configuration using Binary file	✓	✓	✓	✓
CONFIG_TOUCHSCREEN_CYPRESS_ CYTTSP5	Includes TTDA driver in the kernel build	✓	✓	✓	✓
CONFIG_TOUCHSCREEN_CYPRESS_ CYTTSP5_I2C	Includes the TTDA I2C Adapter module in the kernel build	✓	✓	✓	✓
CONFIG_TOUCHSCREEN_CYPRESS_ CYTTSP5_SPI	Includes the TTDA SPI Adapter module in the kernel build	✓	✓	✓	✓
CONFIG_TOUCHSCREEN_CYPRESS_ CYTTSP5_MT_A	Includes the Multi-touch Protocol A module in the kernel build	✓	✓	✓	✓
CONFIG_TOUCHSCREEN_CYPRESS_ CYTTSP5_MT_B	Includes the Multi-touch Protocol B module in the kernel build	✓	✓	✓	✓
CONFIG_TOUCHSCREEN_CYPRESS_ CYTTSP5_BUTTON	Includes the CapSense Button module in the kernel build	✓	✓	✓	✓
CONFIG_TOUCHSCREEN_CYPRESS_ CYTTSP5_PROXIMITY	Includes the Proximity module in the kernel build	✓	✓	✓	✓
CONFIG_TOUCHSCREEN_CYPRESS_ CYTTSP5_DEVICE_ACCESS	Includes the Device Access module in the kernel build	✓	✓	✓	✓
CONFIG_TOUCHSCREEN_CYPRESS_ CYTTSP5_LOADER	Includes the Loader module in the kernel build	✓	✓	✓	✓
CONFIG_TOUCHSCREEN_CYPRESS_ CYTTSP5_DEBUG	Defines the system “DEBUG” to expand all dev_dbg() macros	✓	✓	✓	✓
CONFIG_TOUCHSCREEN_CYPRESS_ CYTTSP5_VDEBUG	Defines the system “VERBOSE_DEBUG” to expand all dev_vdbg() macros	✓	✓	✓	✓
CONFIG_TOUCHSCREEN_CYPRESS_ CYTTSP5_DEBUG_MODULE	Includes the Debug module in the kernel build	✓	-	-	-

Automatic Defines Created by Menuconfig Selections	Description	TTDA 3.1	TTDA 3.2	TTDA 3.3	TTDA 3.4
`CONFIG_TOUCHSCREEN_CYPRESS_ CYTTSP5_DEBUG_MDL`	Includes the Debug module in the kernel build	-	✓	✓	✓
`CONFIG_TOUCHSCREEN_CYPRESS_ CYTTSP5_DEVICE_ACCESS_API`	Includes the Device Access API for other driver interface to the TT Device via the TTDA Device Access	-	-	✓	✓
`CONFIG_TOUCHSCREEN_CYPRESS_ CYTTSP5_TEST_DEVICE_ACCESS_API`	Includes a test module for the Device Access API in the kernel build (used for TTDA validation and coding example).	-	-	✓	✓
`CONFIG_FB`	Build enabled for FB Notifications for Power Management	-	-	-	✓
`UPGRADE_FW_AND_CONFIG_IN_PROBE`	Perform FW and config upgrade during probe instead of scheduling a worker for it. This is disabled by default.	-	-	-	✓

16. IDAC Calibration and Opens/Shorts Tests

TTDA allows you to perform firmware self-test for detecting opens and shorts and execute IDAC calibration via ADB commands. This section describes the necessary steps required to run IDAC calibration, opens, and shorts test. ADB utility is required to be installed and serial debugging be enabled.

16.1 Testing With TTDA

Note: For TTDA 3.1 and earlier: Typical `/path/to` in this chapter is `/sys/bus/ttsp5/devices/cyttsp5_device_access.main_ttsp_core`.

Note: For TTDA 3.2 and later: Typical `/path/to` in this chapter, for I^{2}C host bus products, is `/sys/bus/i2c/devices/4-0024`.

16.1.1 Calibrate IDAC for Mutual Capacitance Sensors

To calibrate IDAC, execute the following commands in order using a command window or script:

1. Suspend scanning is required for IDAC calibration test.

   ```
   adb shell
   ```

   ```
   echo “04 00 05 00 2F 00 03” > /path/to/command
   ```

2. Check if the response status is ready.

   ```
   cat /path/to/status
   ```

3. Get device passthrough command response bytes.

   ```
   cat /path/to/response
   ```

4. Send mutual-cap Sensors IDAC calibration command to command sysfs node.

   ```
   echo “04 00 06 00 2F 00 28 00” > /path/to/command
   ```

5. Check if the response status is ready.

   ```
   cat /path/to/status
   ```

6. Get device passthrough command response bytes.

   ```
   cat /path/to/response
   ```

7. Resume scanning after IDAC calibration test is done.

   ```
   echo “04 00 05 00 2F 00 04” > /path/to/command
   ```

8. Check if the response status is ready.

   ```
   cat /path/to/status
   ```

9. Get device passthrough command response bytes.

   ```
   cat /path/to/response
   ```

16.1.2 Calibrate IDAC for Mutual Capacitance Buttons

To calibrate IDAC, execute the following commands in sequence using a command window or script:

1. Suspend scanning is required for IDAC calibration test.

   ```
   adb shell
   ```

   ```
   echo "04 00 05 00 2F 00 03" > /path/to/command
   ```

2. Check if the response status is ready.

   ```
   cat /path/to/status
   ```

3. Get device passthrough command response bytes.

   ```
   cat /path/to/response
   ```

4. Send mutual-cap Button IDAC calibration command to command sysfs node.

   ```
   echo "04 00 06 00 2F 00 28 01" > /path/to/command
   ```

5. Check if the response status is ready.

   ```
   cat /path/to/status
   ```

6. Get device passthrough command response bytes.

   ```
   cat /path/to/response
   ```

7. Resume scanning after IDAC calibration test is done.

   ```
   echo "04 00 05 00 2F 00 04" > /path/to/command
   ```

8. Check if the response status is ready.

   ```
   cat /path/to/status
   ```

9. Get device passthrough command response bytes.

   ```
   cat /path/to/response
   ```

16.1.3 Calibrate IDAC for Self Capacitance

To calibrate IDAC, execute the following commands in sequence using a command window or script:

1. Suspend scanning is required for IDAC calibration test.

   ```
   adb shell
   ```

   ```
   echo "04 00 05 00 2F 00 03" > /path/to/command
   ```

2. Check if the response status is ready.

   ```
   cat /path/to/status
   ```

3. Get device passthrough command response bytes.

   ```
   cat /path/to/response
   ```

4. Send self-cap Sensors IDAC calibration command to command sysfs node.

   ```
   echo "04 00 06 00 2F 00 28 02" > /path/to/command
   ```

5. Check if the response status is ready.

   ```
   cat /path/to/status
   ```

6. Get device passthrough command response bytes.

   ```
   cat /path/to/response
   ```

7. Resume scanning after IDAC calibration test is done.

   ```
   echo "04 00 05 00 2F 00 04" > /path/to/command
   ```

8. Check if the response status is ready.

```
cat /path/to/status
```

9. Get device passthrough command response bytes.

   ```
   cat /path/to/response
   ```

16.1.4 Shorts Test

For the shorts test, execute the following commands in sequence using a command window or script:

1. Suspend scanning is required for shorts test.

   ```
   adb shell
   echo “04 00 05 00 2F 00 03” > /path/to/command
   ```

2. Check if the response status is ready.

   ```
   cat /path/to/status
   ```

3. Get device passthrough command response bytes.

   ```
   cat /path/to/response
   ```

4. Send Shorts Test command to command sysfs node.

   ```
   echo “04 00 07 00 2F 00 26 04 00” > /path/to/command
   ```

5. Check if the response status is ready.

   ```
   cat /path/to/status
   ```

6. Get device pass-through command response bytes.

   ```
   cat /path/to/response
   ```

7. Get Shorts Test results.

   ```
   echo “04 00 0A 00 2F 00 27 00 00 FF FF 04” > /path/to/command
   ```

8. Check if the response status is ready.

   ```
   cat /path/to/status
   ```

9. Get device passthrough command response bytes.

   ```
   cat /path/to/response
   ```

10. Resume scanning after Shorts test is done.

    ```
    echo “04 00 05 00 2F 00 04” > /path/to/command
    ```

11. Check if the response status is ready.

    ```
    cat /path/to/status
    ```

12. Get device passthrough command response bytes.

    ```
    cat /path/to/response
    ```

16.1.5 Opens Test

For the opens test, execute the following commands in sequence using a command window or script:

1. Suspend scanning is required for opens test.

   ```
   adb shell
   echo “04 00 05 00 2F 00 03” > /path/to/command
   ```

2. Check if the response status is ready.

   ```
   cat /path/to/status
   ```

3. Get device passthrough command response bytes.

   ```
   cat /path/to/response
   ```

4. Send Opens Test command to command sysfs node.

   ```
   echo “04 00 07 00 2F 00 26 03 01” > /path/to/command
   ```

5. Check if the response status is ready.

   ```
   cat /path/to/status
   ```

6. Get device passthrough command response bytes.

   ```
   cat /path/to/response
   ```

7. Get Opens Test results.

   ```
   echo “04 00 0A 00 2F 00 27 00 00 E7 00 03” > /path/to/command
   ```

8. Check if the response status is ready.

   ```
   cat /path/to/status
   ```

9. Get device passthrough command response bytes.

   ```
   cat /path/to/response
   ```

10. Resume scanning after Opens test is done.

    ```
    echo “04 00 05 00 2F 00 04” > /path/to/command
    ```

11. Check if the response status is ready.

    ```
    cat /path/to/status
    ```

12. Get device passthrough command response bytes.

    ```
    cat /path/to/response
    ```

17. Driver Without XRES Signal

Some hardware platforms for Linux/Android may not connect the TrueTouch XRES to a GPIO. Other cases may have creative solutions such as power cycling the TrueTouch part. This is acceptable, but if not done correctly, the reset operation in the driver will not work. Even though the driver supports using soft reset, Cypress strongly recommends using hardware reset or power cycle because this is more reliable than the soft reset command because if the part is not addressable, then the soft reset will fail.

17.1 Solution to Designs Without XRES Signal

The driver will try the hardware reset and if that fails, it will perform a soft reset command. See Technical Notes for function names.

The hardware reset driver functions look for the board configuration file definition of the atomic hardware reset function. If you use a sample board configuration file with an XRES reset defined, it generally returns zero (success) after pulsing the default GPIO assignment, even if the reset does not actually happen. This may need to be customized. If you are not using any hardware reset, it should not be defined so that the function returns a null. If implementing a hardware reset, it should return zero only if some action is performed. Otherwise, the failure return (-ENOSYS) will tell the driver to try the soft reset.

The soft reset command will take place as a backup only if either the reset function is not defined or returns -ENOSYS for a failure.

Debugging suggestion: If you ask for a copy of the kernel log file, it will show which reset failed. If the hardware reset is not causing an error message, the driver assumes it happened. If the XRES is not actually in use, this is an indication that there may be an incorrect function in the board configuration file.

17.2 Technical Notes

The board configuration file is a product configuration file that adapts the product to use various drivers and software based on the specific hardware. The ones sent out with our drivers are suggested scenarios. This is an area of development that needs the involvement of the customer’s product integration engineer who can determine what platform data selections should be made for the driver.

The program flow for the reset is:

```
cyttsp5_reset_and_wait()          // reset and wait for bootloader mode

cyttsp5_hw_reset()                // try XRES then firmware reset command if fails

cyttsp5_hw_hard_reset()           //checks existence of reset function then calls it and if that fails

cyttsp5_hw_soft_reset()
```

Board configuration file XRES pulse example

```
static int cyttsp5_xres(struct cyttsp5_core_platform_data *pdata,
             struct device *dev)
{
       int rst_gpio = pdata->rst_gpio;
       int rc = 0;
       gpio_set_value(rst_gpio, 1);
       msleep(20);
       gpio_set_value(rst_gpio, 0);
       msleep(40);
       gpio_set_value(rst_gpio, 1);
       msleep(20);
       dev_info(dev,
              “%s: RESET CYTTSP gpio=%d r=%d\n”, __func__,
              pdata->rst_gpio, rc);
       return rc;
}
```

The power cycling source may look similar if the GPIO is connected to a transistor switch instead of XRES.

If you choose to code a power cycle into the cyttsp5_xres, be aware that the interrupt from the TrueTouch part is active LOW, so a power cycle will appear as an interrupt to the driver. Cypress recommends that you nest the power cycle inside a disable/enable of the interrupts.

18. System Debugging

This section discusses tools and methods for successfully debugging your system.

18.1 Cypress’s TrueTouch Host Emulator Tool

The first tool you should use when debugging your system is Cypress’s TrueTouch Host Emulator (TTHE) v2.1 or later. Because the TTDA does not modify data or touch events, it is a good idea to verify the touchscreen configuration and operation. If the touch performance is poor, you should connect the touchscreen directly to TTHE to verify behavior.

18.2 Logic Analyzer

A logic analyzer is required to debug the communication between your Android system and the TTSP device. Both the I^{2}C and SPI electrical interfaces use a limited number of signals. You must capture, (after the grounding connection) reset, interrupt, and the serial interface signals with decoding capability. This will allow you to trigger on the initial host reset pulse and capture the initial communication up to and including the first touch events.

Typically, you will see “heartbeat” pulses on the interrupt line indicating that the part is in bootloader mode. This may go on for a long time (seconds) while Linux and Android start up. The TTDA driver will then send a reset pulse, and one or two more interrupts will occur as the system re-enters bootloader mode. The TTDA will then communicate to switch from bootloader mode to operating mode. After this, interrupts are triggered by touch events and will stay asserted until the TTDA clears the event because it must always be in synchronous, hand-shaking mode.

If you need to report bugs or request support for this type of communication problem, a logic analyzer trace is useful.

Note When you measure the clock rate of a TTSP device that is set to operate at 1 MHz, you will observe an initial legacy negotiating phase of 400-kHz operation during initialization. This is expected.

18.3 Enable Linux Debug Macros

Cypress uses the standard Linux define device debug macros such as dev_dbg(). To enable these debug features, define DEBUG, and optionally VERBOSE_DEBUG, at the appropriate level. An example of this is in *cyttsp5_bus.h*. When the debug features are enabled, the TTDA will log error messages as they occur and significant normal milestones such as bootloader exit.

18.4 Helpful Linux Host Windows

The following three Linux host windows are helpful for system debug. You must have Android USB debugging enabled and be connected to a host Linux system with ADB installed. These separate command windows can run at the same time. You may need to locate your ADB command path.

18.4.1 logcat

```
$ sudo ./adb shell logcat
```

This will run the continuously updated Android log file.

18.4.2 getevent

```
$ sudo ./adb shell getevent
```

This will list all of the input event devices that are currently running. Look for the device associated with the TTDA such as */dev/input/event4*.

Use ^C to exit getevent and then restart getevent with details enabled for the device associated with the TTDA

```
$ sudo ./adb shell getevent  -lt /dev/input/event4
```

This will run continuously updated event logging for the TTDA. These are the event signals sent by the Linux event handler to the Android platform event hub.

18.4.3 kmsg

```
$ sudo ./adb shell cat /proc/kmsg
```

This will run the continuously-updated kernel log. This log is where debug prints will appear from the debug enabled TTDA for each touch.

18.4.3.1 Debug Monitor

It is possible to get formatted prints into the kernel log dynamically. By loading the TTDA Debug Monitor module, for each touch event, a formatted print of the touch record information is printed to the log. This must be used cautiously, because it will degrade the touch update performance in order to handle the printing.

```
$ sudo ./adb shell push cyttsp5_debug.ko /data
$ sudo ./adb shell insmod /data/cyttsp5_debug.ko
```

When the debug monitoring is no longer desired, the Debug Monitor can be disabled:

```
$ sudo ./adb shell rmmod cyttsp5_debug.ko
```

To see what modules are currently inserted and running, use the listing command:

```
$ sudo ./adb shell lsmod
```

18.5 Android Display Settings

The Android system can be configured to help you debug touch issues using the normal user GUI. Here are some suggestions:

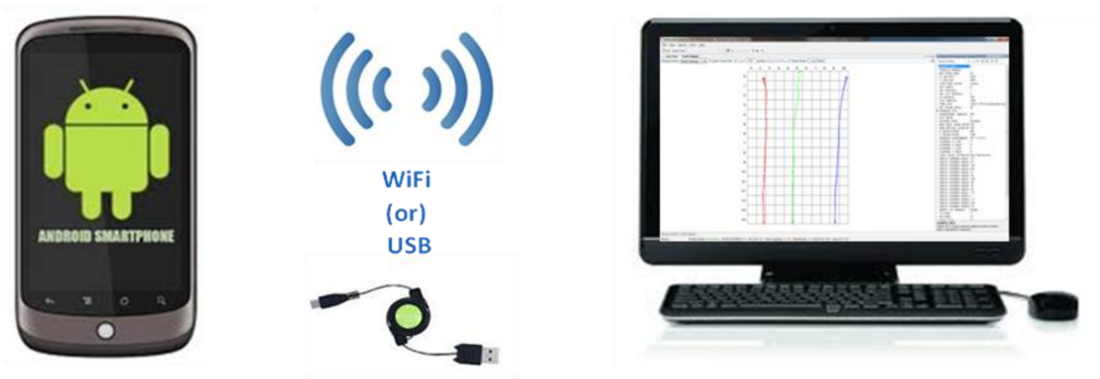
- In **Settings > Display**, set **Sleep** to **30 minutes**. This will keep the display on for as long as possible.
- In **Dev Tools**, use the **Pointer Location** tool to show touch ID, coordinates, cross hairs, and trail ending colors.
- In **Developer Options**, turn the following options on:
  - **Set USB Debugging**: This enables ADB remote debugging, which is used throughout this guide.
  - **Pointer Location**: This provides touch ID, coordinates, cross hairs, and trail ending colors everywhere in the GUI. It is important to know that if you have this turned on and go to the **Pointer Location** tool in **Dev Tools** you will see two traces for every touch.
  - **Show Touches**: This displays a circle or ellipse for each touch event proportional to the size of touch.

19. Mobile Tuner

This section discusses describes how to use ADB with TTHE 3.1 or later. ADB allows tuning and testing with a closed device environment using USB or Wi-Fi. TTHE is able to keep full functionality while using ADB.

19.1 Overview

Figure 19-1. Mobile Tuner Overview



The Mobile Tuner feature allows touchscreen solutions to be tested in an end system using TTHE. For example, testing touchscreens integrated on mobile phones.

19.2 Prerequisites

- Android SDK. Download the SDK at developer.android.com.
- Start SDK Manager after Android SDK installation is done. Use the SDK Manager to install the SDK tools and the API for your build of Android and the Google USB driver.
- TrueTouch Host Emulator v3.1 or later. See the TrueTouch device product release notes for TTHE version requirements. When the target Android device is connected to a Windows PC for the first time, chances are that USB driver cannot be installed properly. Thus ADB connection does not work as expected. To resolve the issue, see section Install USB Driver for Target Android Device for instructions on how to install USB driver for the target Android device on Windows PC.
- Ensure that the image flashed to the phone/tablet has the CONFIG_DEBUG_FS symbol defined. DEBUGFS must be selected in Linux menuconfig to run TTHE mobile tuner. Device Access module and Loader module must also be enabled in TTDA driver to use Mobile Tuner. In the *cyttsp5_bus.h* file for TTDA 3.0, TTDA 3.0.1, TTDA 3.1, or the *cyttsp5_core.h* file for TTDA 3.2 and TTDA 3.3, make sure that the constant TTHE_TUNER_SUPPORT is defined and enabled. Note that TTHE_TUNER_SUPPORT only needs to be enabled once. It is not necessary to enable it in the *cyttsp5_core.c*, *cyttsp5_device_access.c*, or *cyttsp5_loader.c* files. Also the “FW upgrade from binary file” option must be selected in menuconfig in order to enable the Mobile Tuner to perform firmware updates on the device.

  Note: The Mobile Tuner uses the manual_upgrade system object to load firmware.

  □ Load `cyttsp5_loader.ko` files to the Android device using the following commands:

  ```
  adb push cyttsp5_loader.ko /data
  ```

□ Use the following commands to check whether DEBUGFS is already mounted:

```
adb shell
# mount
```

If DEBUGFS is not already mounted, then perform the following commands to mount the debug file system on top of the `/sys/kernel/debug` directory.

```
# mount -t debugfs none /sys/kernel/debug
```

□ Insert `cyttsp5_loader.ko` is also needed.

```
adb shell
# cd /data
# insmod cyttsp5_loader.ko
```

19.3 Install USB Driver for Target Android Device

Note 1 For the same make and model of Android devices, this process of installing USB drivers only need to be done once on each Windows PC running the Mobile Tuner.

Note 2 Make sure the Android SDK is properly installed before proceeding with the instructions in this section.

Note 3 The images in this section are based a Windows 7 machine. Other Windows PCs should have similar options.

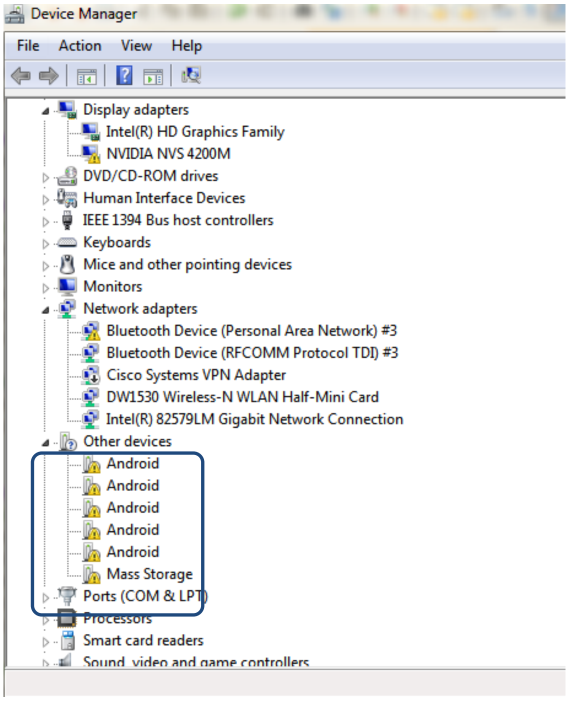
When a target Android device is connected to a Windows PC for the first time, it may display a warning message, “Unable to install USB driver for the device”. Automatic driver update online may also fail. As a result, the ADB cannot work appropriately, which blocks further device testing and debugging.

To install the USB driver for the target Android device properly, follow these instructions:

1. Open the Device Manager window; you will see a warning sign (yellow triangle) on the lower right corner of the target Android device icon. It indicates that the driver for the device is not properly installed.

   Note: If there is no warning sign on the Android device icon, then the USB driver is already installed successfully. So there is no need to follow the steps in this section. Proceed to section Settings for Mobile Tuning via USB or Settings for Mobile Tuning Via WiFi.

Figure 19-2. Device Manager Window

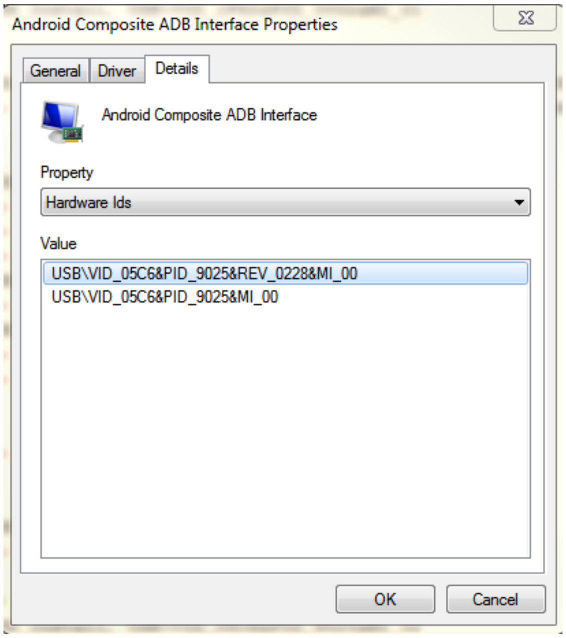

Device Manager
File
Action
View
Help
Display adapters
Intel(R) HD Graphics Family
NVIDIA NVS 4200M
DVD/CD-ROM drives
Human Interface Devices
IEEE 1394 Bus host controllers
Keyboards
Mice and other pointing devices
Monitors
Network adapters
Bluetooth Device (Personal Area Network) #3
Bluetooth Device (RFCOMM Protocol TDI) #3
Cisco Systems VPN Adapter
DW1530 Wireless-N WLAN Half-Mini Card
Intel(R) 82579LM Gigabit Network Connection
Other devices
Android
Android
Android
Android
Android
Mass Storage
Ports (COM & LPT)
Processors
Smart card readers
Sound, video and game controllers

2. Right-click on the warning sign and select **Properties**.

3. Click the **Details** tab, select **Hardware Ids**. Record the Hardware IDs of the device.

   Note: Figure 19-3 is only an example. Other Android devices have different Hardware IDs.

Figure 19-3. Example Hardware IDs Window



Android Composite ADB Interface

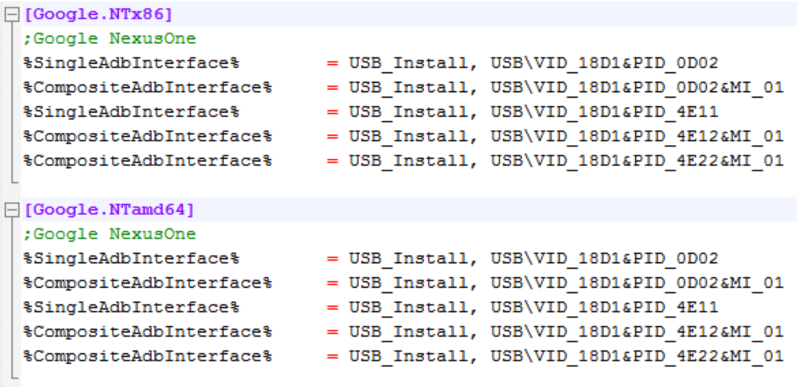
Property: Hardware Ids

Value:

USB\VID_05C6&PID_9025&REV_0228&MI_00
USB\VID_05C6&PID_9025&MI_00

4. Go to the Android installation folder and open **android_winusb.inf** file. Typical path is `C:\Program Files\Android\android-sdk\extras\google\usb_driver`. You can see a list of devices in `[Google.NTx86]` and `[Google.Ntamd64]` similar to Figure 19-4.

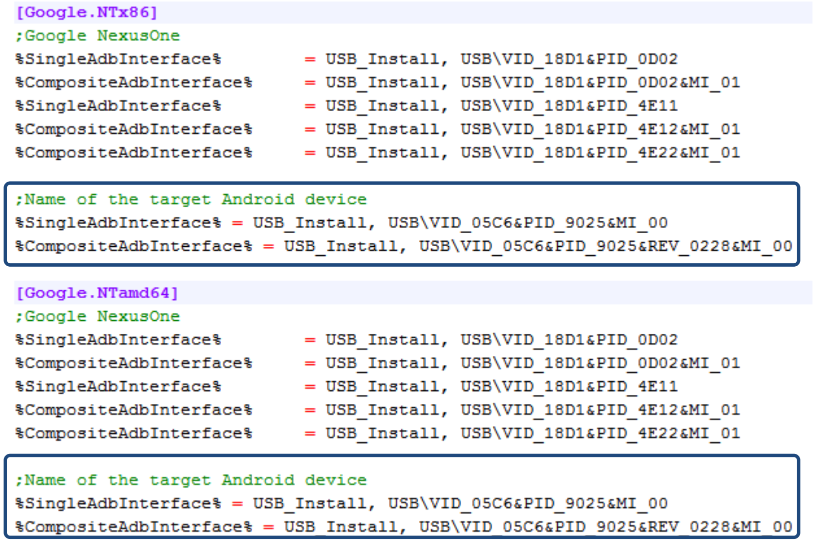
Figure 19-4. Example [Google.NTx86] and [Google.Ntamd64] in android_winusb.inf



```
[Google.NTx86]
;Google NexusOne
%SingleAdbInterface%        = USB_Install, USB\VID_18D1&PID_0D02
%CompositeAdbInterface%     = USB_Install, USB\VID_18D1&PID_0D02&MI_01
%SingleAdbInterface%        = USB_Install, USB\VID_18D1&PID_4E11
%CompositeAdbInterface%     = USB_Install, USB\VID_18D1&PID_4E12&MI_01
%CompositeAdbInterface%     = USB_Install, USB\VID_18D1&PID_4E22&MI_01

[Google.NTamd64]
;Google NexusOne
%SingleAdbInterface%        = USB_Install, USB\VID_18D1&PID_0D02
%CompositeAdbInterface%     = USB_Install, USB\VID_18D1&PID_0D02&MI_01
%SingleAdbInterface%        = USB_Install, USB\VID_18D1&PID_4E11
%CompositeAdbInterface%     = USB_Install, USB\VID_18D1&PID_4E12&MI_01
%CompositeAdbInterface%     = USB_Install, USB\VID_18D1&PID_4E22&MI_01
```

5. In both `[Google.NTx86]` and `[Google.Ntamd64]`, add the similar “`%SingleAdbInterface%`” and “`%CompositeAdbInterface%`” lines for the target Android device according to the Hardware Ids obtained in Step 3. Example is shown in Figure 19-5.

Figure 19-5. Example Addition of Android Device Information in android_winusb.inf



```
[Google.NTx86]
;Google NexusOne
%SingleAdbInterface%        = USB_Install, USB\VID_18D1&PID_0D02
%CompositeAdbInterface%     = USB_Install, USB\VID_18D1&PID_0D02&MI_01
%SingleAdbInterface%        = USB_Install, USB\VID_18D1&PID_4E11
%CompositeAdbInterface%     = USB_Install, USB\VID_18D1&PID_4E12&MI_01
%CompositeAdbInterface%     = USB_Install, USB\VID_18D1&PID_4E22&MI_01

;Name of the target Android device
%SingleAdbInterface% = USB_Install, USB\VID_05C6&PID_9025&MI_00
%CompositeAdbInterface% = USB_Install, USB\VID_05C6&PID_9025&REV_0228&MI_00

[Google.NTamd64]
;Google NexusOne
%SingleAdbInterface%        = USB_Install, USB\VID_18D1&PID_0D02
%CompositeAdbInterface%     = USB_Install, USB\VID_18D1&PID_0D02&MI_01
%SingleAdbInterface%        = USB_Install, USB\VID_18D1&PID_4E11
%CompositeAdbInterface%     = USB_Install, USB\VID_18D1&PID_4E12&MI_01
%CompositeAdbInterface%     = USB_Install, USB\VID_18D1&PID_4E22&MI_01

;Name of the target Android device
%SingleAdbInterface% = USB_Install, USB\VID_05C6&PID_9025&MI_00
%CompositeAdbInterface% = USB_Install, USB\VID_05C6&PID_9025&REV_0228&MI_00
```

6. If there are multiple devices, as shown in Figure 19-2, repeat Step 2 to 5 for each of them. Save the **android_winusb.inf** file.
7. Go back to Device Manager. Right-click on the Android device icon (shown in Figure 19-2), select **Update Driver Software…** > **Browse my computer for driver software**. In the "Search for driver software in this location" box, click on **Browse** and point to the directory where **android_winusb.inf** file is located. Typical path is `C:\Program Files\Android\android-sdk\extras\google\usb_driver`.
8. Driver update should start and be successful.

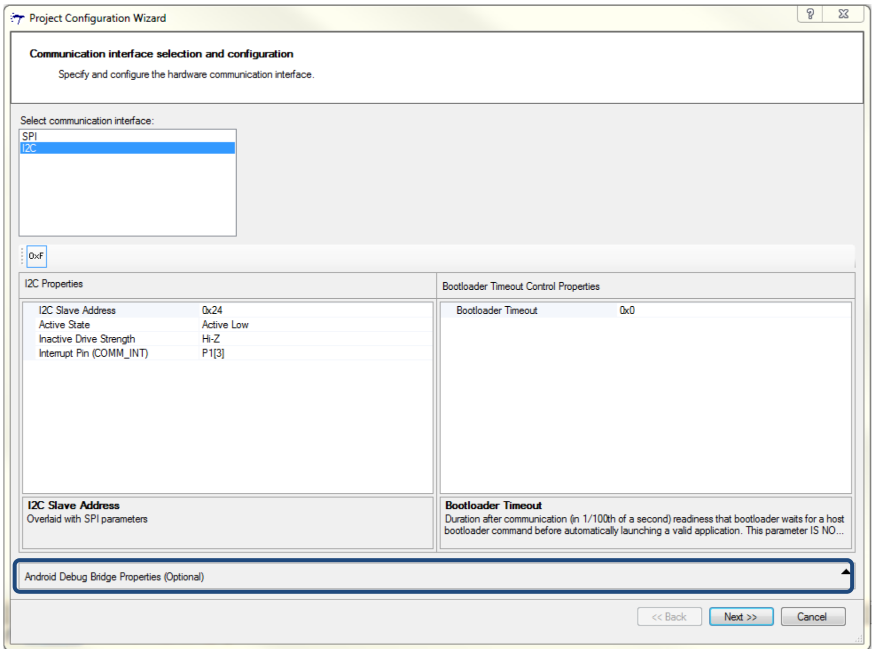
19.4 Settings for Mobile Tuning via USB

19.4.1 TTHE 3.1 and later

Follow these steps to set up the mobile tuner:

1. In TTHE menu, select **View** > **Project Configuration Wizard**.

Figure 19-6. Project Configuration Wizard in TTHE 3.1 and later



Project Configuration Wizard

Communication interface selection and configuration

Specify and configure the hardware communication interface.

Select communication interface:
- SPI
- I2C

I2C Properties

I2C Slave Address	0x24
Active State	Active Low
Inactive Drive Strength	Hi-Z
Interrupt Pin (COMM_INT)	P1[3]

I2C Slave Address
Overlaid with SPI parameters

Bootloader Timeout Control Properties

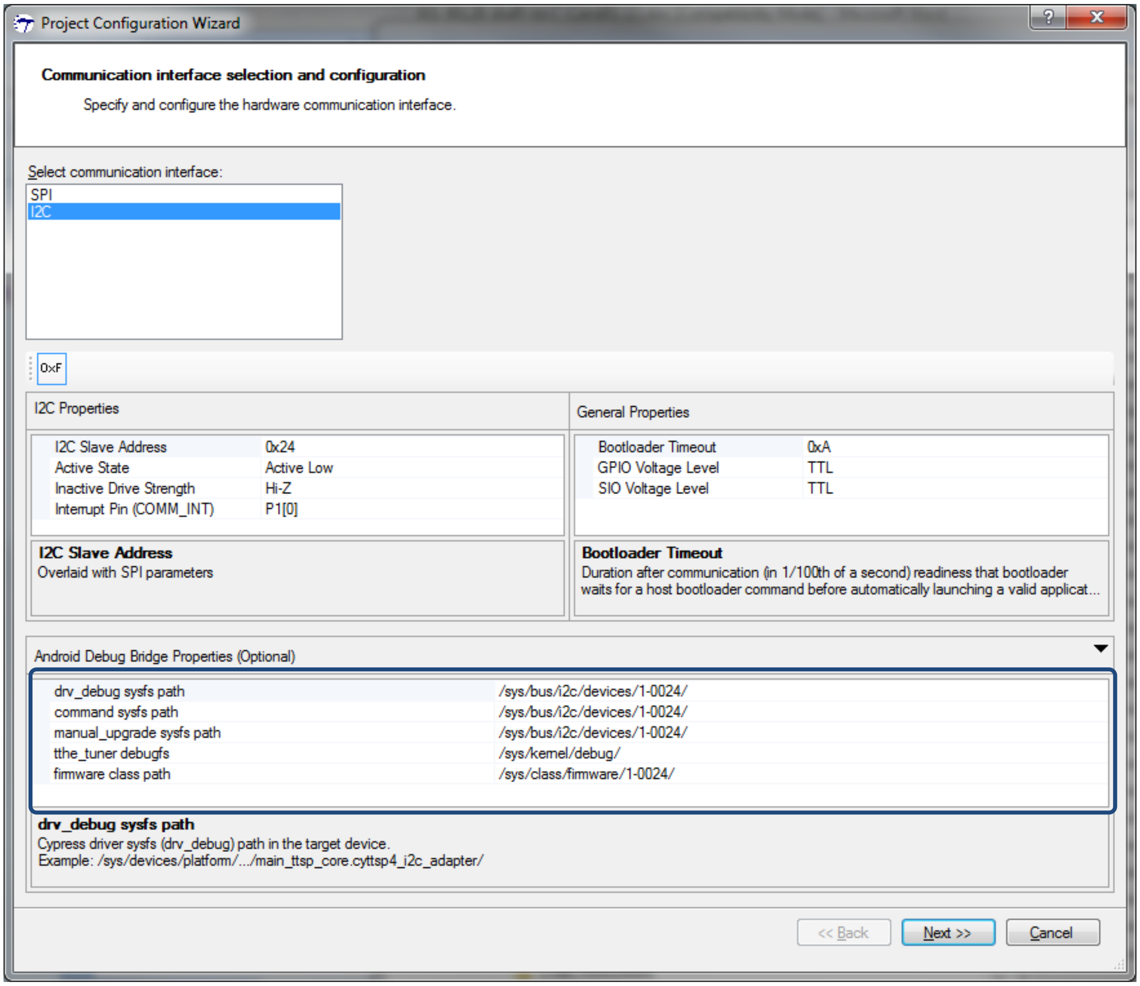
Bootloader Timeout	0x0

Bootloader Timeout
Duration after communication (in 1/100th of a second) readiness that bootloader waits for a host bootloader command before automatically launching a valid application. This parameter IS NO...

Android Debug Bridge Properties (Optional)

2. Click **Android Debug Bridge Properties (Optional)** to expand the ADB settings menu, as shown in Figure 19-7.

Figure 19-7. Example Android Debug Bridge Settings in TTHE 3.2 and later



Communication interface selection and configuration

Specify and configure the hardware communication interface.

Select communication interface:

SPI
I2C

I2C Properties

I2C Slave Address	0x24
Active State	Active Low
Inactive Drive Strength	Hi-Z
Interrupt Pin (COMM_INT)	P1[0]

I2C Slave Address
Overlaid with SPI parameters

General Properties

Bootloader Timeout	0xA
GPIO Voltage Level	TTL
SIO Voltage Level	TTL

Bootloader Timeout
Duration after communication (in 1/100th of a second) readiness that bootloader waits for a host bootloader command before automatically launching a valid applicat...

Android Debug Bridge Properties (Optional)

drv_debug sysfs path	/sys/bus/i2c/devices/1-0024/
command sysfs path	/sys/bus/i2c/devices/1-0024/
manual_upgrade sysfs path	/sys/bus/i2c/devices/1-0024/
tthe_tuner debugfs	/sys/kernel/debug/
firmware class path	/sys/class/firmware/1-0024/

drv_debug sysfs path
Cypress driver sysfs (drv_debug) path in the target device.
Example: /sys/devices/platform/.../main_ttsp_core.cyttsp4_i2c_adapter/

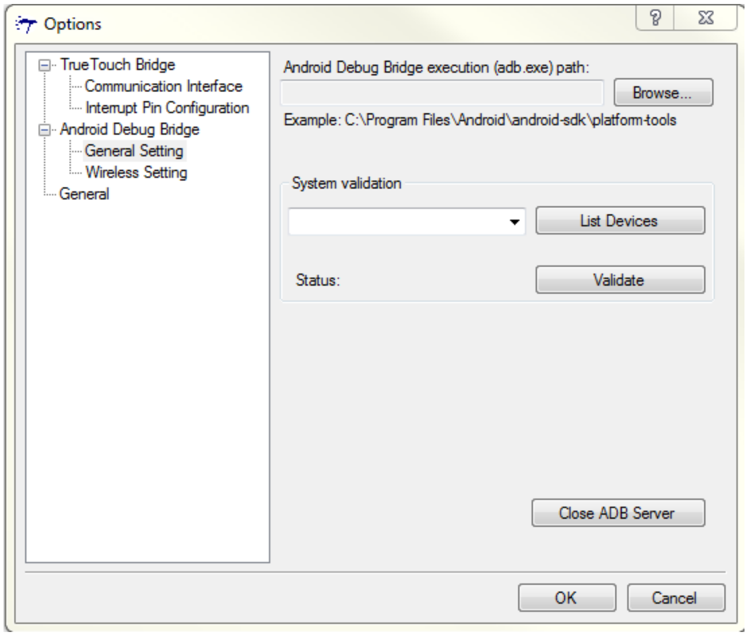
3. Type in the ADB paths for each of the sysfs and debugfs required according to Table 19-1. These are sample paths; the actual paths may vary depending on where the driver objects are located in the product file directory structure. The "tthe_tuner" path in Table 19-1is a fixed path location. The "firmware class" path in the table is fixed for the "/sys/class/firmware/" portion of the path. Refer to Section 19.4.2 for information about finding the path.

Table 19-1. Sample Android Debug Bridge Path Settings for TSG5 Devices

drv_debug sysfs path	/sys/bus/i2c/devices/1-0024/
command sysfs path	/sys/bus/i2c/devices/1-0024/
manual_upgrade sysfs path	/sys/bus/i2c/devices/1-0024/
tthe_tuner debugfs	/sys/kernel/debug/
firmware class path	/sys/class/firmware/1-0024

4. Open TTHE menu **Tools** > **Options** > **Android Debug Bridge** > **General Setting** tab. Enter the *adb.exe* path in the "Android Debug Bridge execution (adb.exe) path". Typical path is *C:\Program Files\Android\android-sdk\platform-tools.*

Figure 19-8. Android Debug Bridge Execution General Setting in TTHE 3.1



5. Click **List Devices** and then **Validate**. If ADB is set up correctly, “Status:” shows “Validate success”. Then you are ready to use mobile tuner features.

19.4.2 Finding the Driver Directory Paths

For TTDA3.1 and below, the path for the “drv_debug”, “command”, and “manual_upgrade” sysfs objects are found at the path “/sys/bus/ttsp5/devices/cyttsp5_loader.main_ttsp_core/”.

For TTDA3.2 and above, the example paths shown in Section 19.4.1 are for devices that are integrated onto a host I2C bus and at the default Cypress device address 0x24 (“1-0024” where the 1 is for the bus number and the 0024 is for the device address on that bus). For SPI bus, the directory name typically shows the SPI bus and the device slave select number (for example, “spi1.1”).

It is not always known where the device is located in the host buses so knowing some of the expected sysfs object names associated with the driver helps to find the path. One way to determine this is to perform a recursive directory listing to a text file and search for the driver sysfs objects. The following objects should always exist if the driver comes up normally:

- drv_debug
- drv_irq
- drv_ver

When the path for these objects is found, that path is the one for use in the “drv_debug”, “command”, and “manual_upgrade”; you should also see those objects in the list for that directory.

If you do not see the “manual_upgrade” in the list, then either the Loader module (cyttsp5_loader.ko) was not built or was built as a module but not inserted at run time. If this object is missing, then the TTHE will fail to connect through the driver.

If you see the objects: “formated_output” and “int_count”, then the driver debug module is running (cyttsp5_debug.ko). This module should only be loaded and running during specific touch debug time. It reduces touch performance significantly and is recommended not be compiled as built-in but rather as a module and only inserted when needed.

Note that is not possible to find the firmware class path until the driver activates the firmware class library. This only occurs when a ‘1’ is written to the “manual_upgrade” sysfs object such as:

```
> adb  push  cyttsp5_loader.ko  /data
> adb  shell
# insmod  /data/cyttsp5_loader.ko
# echo 1 > /sys/bus/i2c/devices/1-0024/manual_upgrade
```

Then you should be able to see the following path (using the sample path) and the “data” and “loading” objects:

```
# ls /sys/class/firmware/1-0024
data
```

```
loading
power
subsystem
uevent
```

Note that the “data” and “loading” objects are used for downloading firmware to the device as described in Section 4.7.2 above.

19.5 Settings for Mobile Tuning Via WiFi

1. First, an ADB Wireless app needs to be installed and run in the Android device for the Mobile Tuner to be working with WiFi. The ADB Wireless app can be obtained and installed from Google Play. Allow root access to the wireless application.

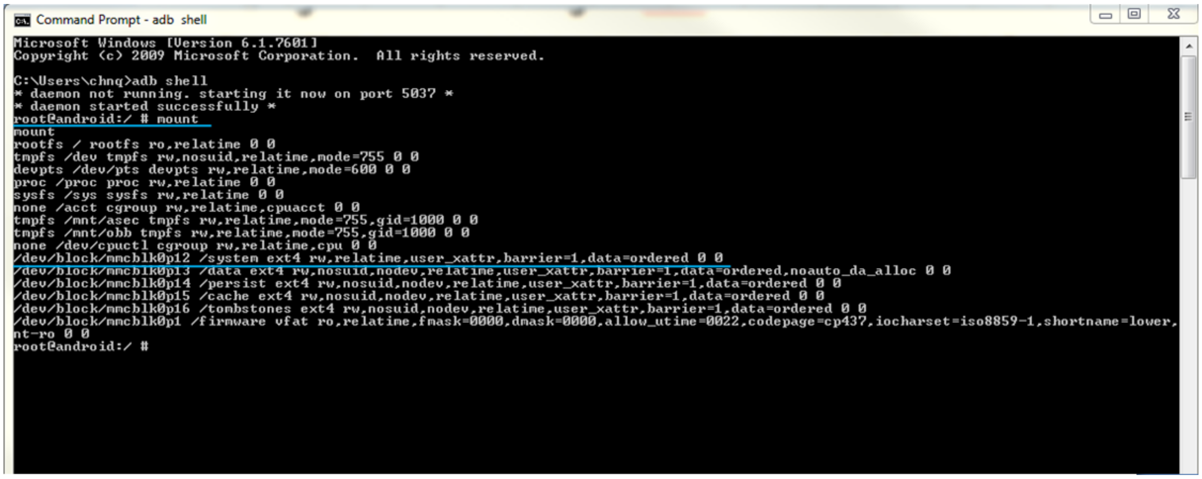
   Alternatively, ADB Wireless app apk and *Superuser.apk* can be acquired and installed in the Android device manually to achieve the above requirement. The following steps show the example procedure to install *adbWireless.apk* and allow root access. Depending on the end Android device, some paths in the example need to be changed. Consult a Cypress FAE to get the ADB Wireless app apk and *Superuser.apk* if you do not have them already.

   (1) Find system settings using command:

   ```
   adb shell
   # mount
   ```

   Record the line that contains the flash device information under the */system* directory. Example from a Cypress development kit is shown in Figure 19-9. Customer platform would result in a different path.

   Figure 19-9. Example Output of ADB `mount` Command



```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\chnq>adb shell
* daemon not running. starting it now on port 5037 *
* daemon started successfully *
root@android:/ # mount
mount
rootfs / rootfs ro,relatime 0 0
tmpfs /dev tmpfs rw,nosuid,relatime,mode=755 0 0
devpts /dev/pts devpts rw,relatime,mode=600 0 0
proc /proc proc rw,relatime 0 0
sysfs /sys sysfs rw,relatime 0 0
none /acct cgroup rw,relatime,cpuacct 0 0
tmpfs /mnt/asec tmpfs rw,relatime,mode=755,gid=1000 0 0
tmpfs /mnt/obb tmpfs rw,relatime,mode=755,gid=1000 0 0
none /dev/cpuctl cgroup rw,relatime,cpu 0 0
/dev/block/mmcblk0p12 /system ext4 rw,relatime,user_xattr,barrier=1,data=ordered 0 0
/dev/block/mmcblk0p13 /data ext4 rw,nosuid,nodev,relatime,user_xattr,barrier=1,data=ordered,noauto_da_alloc 0 0
/dev/block/mmcblk0p14 /persist ext4 rw,nosuid,nodev,relatime,user_xattr,barrier=1,data=ordered 0 0
/dev/block/mmcblk0p15 /cache ext4 rw,nosuid,nodev,relatime,user_xattr,barrier=1,data=ordered 0 0
/dev/block/mmcblk0p16 /tombstones ext4 rw,nosuid,nodev,relatime,user_xattr,barrier=1,data=ordered 0 0
/dev/block/mmcblk0p1 /firmware vfat ro,relatime,fmask=0000,dmask=0000,allow_utime=0022,codepage=cp437,iocharset=iso8859-1,shortname=lower,
nt-ro 0 0
root@android:/ #
```

   (2) Using system information, construct the following command to remount the system:

   ```
   # mount -o remount,rw -t ext4 /dev/block/mmcblk0p12 /system
   ```

   Note: The above line is example only. According to Step (1) example, the */dev/block/mmcblk0p12* is the flash device of fs type “ext4” and the */system* directory is mounted as “rw”. Depending on the information recorded in Step (1), modify the mount command accordingly. Make sure that */system* is remounted as “rw” (read and write). Then proceed with the commands below.

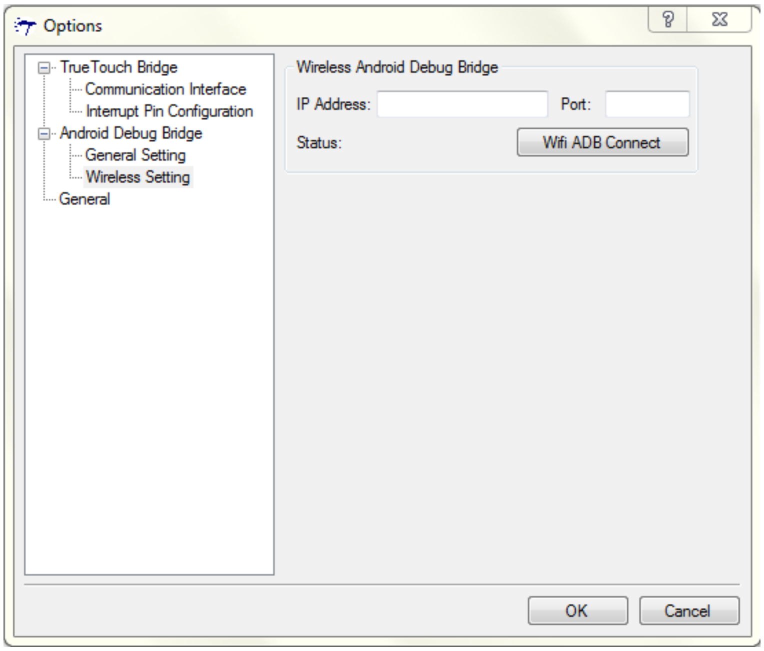
   ```
   adb push /bin/su system
   adb push /app/Superuser.apk system
   adb push adbWireless.apk system
   adb shell
   # cd system/xbin
   # mv su su_old
   ```

```
# mv ../su .

# chmod 04755 su

# cd ../

# mv Superuser.apk app

# mv adbWireless.apk app
```

(3) Restart the system.

2. In TTHE menu, select **Tools** > **Options**. In **Android Debug Bridge** > **Wireless Setting** tab, enter the IP address and port for the Wireless Android Debug Bridge.
3. Click **Wifi ADB Connect**.
4. Check “Status”, **connected** indicates successful connection to the target device. Now, you can tune your mobile.

Figure 19-10. Tools Options Wireless Settings Example



Revision History

Document Revision History

Document Title: TrueTouch® Driver for Android (TTDA) 3.x User Guide

Document Number: 001-90126

Revision	Issue Date	Origin of Change	Description of Change
**	11/21/2013	SWU	New User Guide.
*A	02/11/2014	SWU	Added TTDA 3.2 Features Fixed several grammatical edits Added missing hover feature to New Features in TTDA 3.1
*B	06/18/2014	KEV	Added TTDA3.3 Features Updated TTDA3.2 Features section to include the new Path Name Convention (Section 7.1.3) and Updated Structures (Section 7.1.4). Added Manufacturing (In-system) Tests - Section 8.1.1 Added fixes for User Guide defects: 1. CDT112303 - Added text to Section Virtual Keys to describe CY_MT_FLAG_VKEYS. Updated note in 14.2 Virtual Key Map to clarify the naming of the “virtualkeys” object and the key map and key layout file names. Added information to 14.3 item 1) to explain that CY_MAXX and CY_MAXY need to be set manually when Virtual Keys are used, otherwise the driver uses the X, Y resolution information from the System Information. 2. CDT124223 - Added explanation in section 1.4 that PIP always effectively uses Synchronous Level handshake method. Added comment to FAQ Answer to question about handshake that the level_irq_udelay is used as Boolean to determine whether to set the host interrupt service to low edge trigger (default) or low level trigger (any non-zero value. 3. CDT172586 - Added Table 1-1 Driver Supported Features. Table include Driver PIP versions and is matrixed by supported features. Includes applicable kernel versions, FW versions and devices. 4. CDT172589 - Cleaned up the bullet 4 in Section 19.2 to include all needed driver settings to enable Assisted Tuning in the driver. 5. CDT174215 - Section 19.2 shows prerequisites for Mobile Tuner. Bullet 4 explains what modules and definitions are required to enable the Mobile Tuner including the CONFIG_DEBUG_FS and TTHE_TUNER_SUPPORT definitions. 6. CDT174272 - Section 19.2 shows prerequisites for Mobile Tuner. Added note to Bullet 4 that explains that Mobile Tuner uses the Manual_upgrade system object to load firmware. 7. CDT175694 - Added comment to section 14.3 that for Device Tree, changes

Revision	Issue Date	Origin of Change	Description of Change
			may be made to the Device Tree which only requires that the Device Tree is rebuilt. Otherwise changes to driver code require driver and kernel rebuild. 8. CDT179464 - Updated Section 11.5 to include enable and disable Watchdog instructions. 9. CDT181245 - CDT Items 1-8 above. 10. CDT181475 - Added instructions at the end of section 15.2.1 for updating the constant: CYTTSP5_LOADER_FW_UPGRADE_RETRY_COUNT in the loader source to control the number of times to attempt to automatically load the firmware at startup. 11. CDT184253 - Added Caution and Recommendation paragraphs at end of section 15.2.3
*C	10/08/2014	KEV	Added NDA requirement to cover page. Added NDA requirement to footers. Section A - Added New Features in TTDA3.4 to the list. Table 1-1 Added Row for TMA445A at bottom of table in order to exclude the features that are not supported in TSG6_M.BASE.V1 (EZ-wake, Hover, Stylus). Section 1.3.1 Added TMA445A Datasheet and TRM to the documentation table. Section 1.3.1 Added CY3295-MTK User guide to documentation table. Table 3-2 Added Dragon board files to the table of TTDA3.2 and above file list with notes that the files are for TTDA3.4. Table 3-2 Added missing files: cyttsp5_test_device_access_api.c and cyttsp5_device_access-api.h to the list. Section 9.0 - Added new section for TTDA3.4 new features including: support for Panel ID reporting by the bootloader for use by the Driver to select proper configuration to load and LCD blanking and unblanking notification for power management functions (sleep and wake). Added CM/CP Test Program description. Figure 15-1 Added figure for automatic firmware and TT Configuration update rules. Section 15.2.6 Added TTDA3.4 column to the definitions table. Added CONFIG_FB and UPGRADE_FW_AND_CONFIG_IN_PROBE for TTDA3.4. Figure 19-7 Updated Wizard picture to show the typical paths for TTDA3.2 and above devices for ADB connection. Table 19-1 Updated to show typical paths for TTDA3.2 and above. Added firmware class path.
*D	10/13/2014	KEV	Section 9.1.3: Removed CM/CP test program description. This test program is not part of the TTDA distribution. Section 19.4.1 Item 3: Added comment that that the paths shown are sample paths and the actual paths may vary depending on where the driver objects are located in the product file directory structure. Section 19.4.2: Added the section to describe steps for locating the directory path the driver sysfs objects.