TERIDIAN SEMICONDUCTOR CORP.

Simplifying System Integration™

73S12xxF
Software User Guide

September 14, 2009
Rev. 1.50
UG_12xxF_016

Teridian Semiconductor Corp., 6440 Oak Canyon, Suite 100, Irvine, CA 92618
TEL (714) 508-8800, FAX (714) 508-8877, http://www.teridian.com

Table of Contents

Figures

Tables

1 Introduction

The Teridian Semiconductor Corporation 73S12xxF single-chip Smart Card Terminal Controllers consist of the 73S1209F, 73S1210F, 73S1215F and 73S1217F. These System-on-Chip devices provide the functions necessary to build a low-cost smart card terminal.

The 73S12xxF Evaluation Board allows development of an embedded application in conjunction with an In-Circuit Emulator (ICE). An application can be programmed in either ANSI C or 80515 assembly language using this evaluation board.

Teridian provides a development Toolkit that includes a set of libraries (Application Programming Interface or API). The API is written in ANSI C to control all the features present on the evaluation boards. These libraries include functions to manage the low-level 80515 core functions such as memory, clock, power modes, interrupts; and high-level functions such as the Liquid Crystal Display (LCD), keyboard, Real-Time Clock (RTC), smart card interfaces, Universal Serial Bus (USB)/Serial interfaces and I/Os. These APIs reduce development time dramatically, since they allow the developer to focus on developing the application without dealing with the low-level layer such as hardware control, timing, etc. This document describes the Toolkit's hierarchical layers and how to use them.

Certain function blocks (such as USB and RTC) are not available on all 73S12xxF devices. As a result, the related APIs can not be used with some ICs. Refer to the data sheets for further details.

This document applies to the following components:

- LAPI Version 4.00 (DFU), LAPI Version 3.30 (BL), LAPI Version 2.30 (non-BL)
- HAPI Version 4.00 (DFU), HAPI Version 3.30 (BL), HAPI Version 2.40 (non-BL)
- Serial Pseudo-CCID Application Version 3.1
- USB CCID Application Version 2.1 (DFU), USB CCID Application Version 1.5 (non-DFU)
- Devices: 1215A05, 1217A06 and 1210/1209A02

1.1 Acronyms

APDU	Application Protocol Data Unit
API	Application Programming Interface
ATR	Answer To Reset
BL	Boot Loader
CCID	Integrated Circuit Card Interface Device
COM	Communication Port
DFU	Device Firmware Upgrade
DTK	Development ToolKit
DTM	Device Test Manager
EMV	Euro, MasterCard®, Visa®
HAPI	High-level API
HCT	Hardware Compatibility Test
ICC	Integrated Circuit Card
ISO	International Standards Organization
ISP	In-System Programming
JICSAP	Japan IC Card System Application council
LAPI	Low-level API
LAPIE	Low-level API exerciser
LCD	Liquid Crystal Display
Non-BL	Non Boot Loader
PC	Personal Computer
PIN	Personal Indentification
RAM	Random Access Memory
ROM	Read Only Memory

RTC	Real Time Clock
TSC	Teridian Semiconductor Corporation
USB	Universal Serial Bus
WHQL	Windows Hardware Quality Lab

1.2 Use of this Document

The reader should be familiar with microprocessors, particularly the 80C51/80C52/80515 architecture, firmware, embedded software development and smart card application. Knowledge of the *USB 2.0 Specification*, ISO 7816 Parts 1/2/3/4 and EMV2000 standards may also be helpful.

This document presents the software features as designed in the 73S12xxF Evaluation Board. Users should also have available other 73S12xxF publications such as the data sheet for the particular 73S12xxF device being used, the *72S12xxF Evaluation Board User's Guide*, and the application notes for additional details and recent development information.

1.3 Statement of Compliance

The software and hardware for the 73S12xxF meets or exceeds USB 2.0 Full Speed, EMV2000 (both T=0 and T=1 protocols) and ISO 7816 protocol standards. The Windows® XP USB CCID Driver is designed to meet Microsoft Windows Logo compliance. Refer to the respective documentation for further information about these standards.

The embedded applications (and their associated libraries) have passed the USB.org USB Command Verifier version 1.3 Beta, Microsoft HCT version 12.01.1 for USB, Microsoft DTM for both XP and Vista, and EMV Level 1 compliant testing.

2 Design Guide

This section provides designers with basic guidance in developing smart card reader applications utilizing the TSC 73S12xxF devices. There are three types of applications that can be developed:

- A Host application (for example: an application residing on a PC, e.g. Windows 2000, Windows XP, Windows CE or in a host microprocessor).
- An Embedded application using both High-level APIs and Low-level APIs (in Flash).
- An Embedded application using the Low-level APIs only (in Flash).

There are two options to connect the 73S12xxF Evaluation Board or demo boards to a PC or host controller:

- UART/RS232 serial interface.
- USB V2.0 full speed/12 Mbps interface.

2.1 Development Environment

2.1.1 Hardware Requirements

The recommended hardware requirements include:

- Teridian 73S12xxF Evaluation Board.
- AC Adaptor (AC/DC output) or Variable Bench Power Supply.
- PC Pentium® with 512 MB RAM and 10 GB hard drive, 2 COM ports, and 2 USB port (if the USB interface is utilized) running Windows XP.

Optional Hardware includes:

- Signum Systems ADM-51 In-Circuit Emulator (for debugging the embedded application) with or without trace capability. Signum references this device as the ADM-51 Emulator. This device is configured to use one of the PC's USB Ports. Contact Signum Systems at www.signumsystems.com for the latest version of the ICE software.
- The Teridian Flash Programming Tool (for programming Flash when a Signum ADM-51 ICE is not available).

2.1.2 Software Requirements

The following are the recommended software requirements:

- For embedded application programming:
  - Keil™ Compiler. Version 7.0 or later is recommended.
- Keil µVision®2 or µVision3 IDE.
- If an ICE is used, Signum Systems software (comes with Signum Systems ADM-51 ICE hardware). The ICE can also be used to program Flash.
- A Teridian Flash Programmer (TFP) module for programming Flash.
- For Windows PC application programming:
  - Visual Basic®, Visual Studio® or Visual C/C++® for Windows 2000 or Windows XP.
- Optionally, Keil's extended linker (LX51 instead of BL51) for code size optimization purposes.

The following software tools/programs are included in the 73S12xxF Development Kit and should be installed on the development PC:

- USB View – a shareware PC tool which can be downloaded from www.usb.org.
- usbccid.sys/usbccid.inf – a Microsoft generic Windows XP CCID USB driver.

- CCIDTSC-*.sys/CCIDTSC-*.inf – Teridian Windows Drivers.
- ccidusb-*.hex – an embedded application used for USB CCID communications with Windows or Linux OS. This embedded program can be used with either Microsoft's generic USB driver or the Teridian driver.
- ccidrs232.hex – an embedded application used for RS232/Serial communications with a host running Windows OS.
- CCID-USB.exe – a PC application written in C# to be used in conjunction with the CCIDTeridian.sys USB driver with the evaluation board programmed with the CCIDUSB-*.hex firmware.
- Low-level API Library – an embedded flash module that provides low-level APIs (physical layer) to control the 73S12xxF.
- High-level API Library – an embedded flash module at the protocol level that provides APIs to control different features of the Smart Card. EMV Level I protocol layer is implemented within this module.
- Include/header files for both the Low-level API and the High-level API libraries.
- Sample code for Serial's Pseudo CCID protocol. For the USB interface, the USB CCID firmware source code is also included.
- Linux driver for USB CCID and Linux Application for USB DFU interface.

2.2 Software Build Environment

Install the Keil compiler and select all default options (recommended). When prompted for a target device, select TSC-73S12xxF. This option may not be available on older versions of the compiler. In this case, select TSC 73M6513. For development, an upgrade to a newer version of the Keil compiler is highly recommended. This option can be changed at any time by:

Under 'Project' – 'Select Device for target 'Target1' '– CPU tab' – scroll down to TDK Semiconductor (with older version of Keil) or Teridian Semiconductor (available with newer version of Keil) – select either 71M6513 or 73S12xx.

Under 'Project' – 'Options for target 'target1'' – Target tab – Xdata memory field, the RAM start address should be set to 0x0000, and RAM size should be set to 0x0800.

2.2.1 Software Architecture

The 73S12xxF software architecture is partitioned into three separate layers:

- The Low-level API (LAPI) device or physical layer, which contains a set of function calls to directly manage the peripherals and CPU management (such as clock, timing, power saving, etc.).
- The second layer is the High-level API (HAPI), which is essentially the protocol layer. It provides functions for communication with the smart card (ICC).
- The third layer is the application layer. This layer is left for the application software developer to design any suitable smart card reader applications.

Figure 1 shows the partitions for each software component and its approximate memory size. As illustrated, there are many different ways an application can be designed and implemented. Section 4 Design Reference describes the API functions within each component in more detail. The embedded application sample source code for most of the main features of the chip is provided in the release.

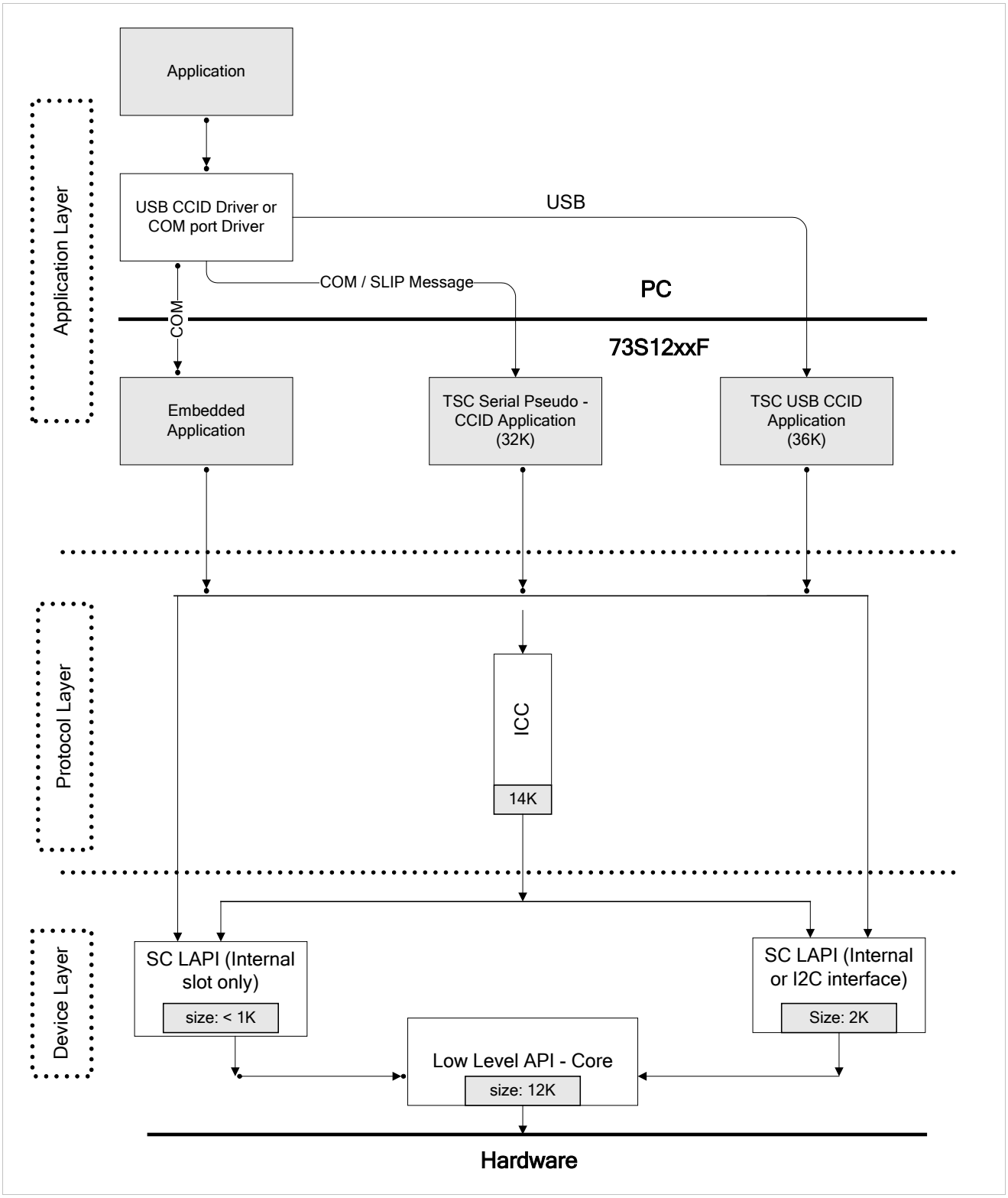

Figure 1: Software Architecture Diagram
(Sizes indicated are approximate)

2.2.2 API/Library and Header Files

The library files included in the software development kit are listed below (see Figure 1: Software Architecture Diagram for the library partitions). The following nomenclature applies to the file names:

- '?' is either 'S' (for Single-8010) or 'M' (for Multiple-8010)
- 'xx' is 'BL' (for Serial Boot Loader), 'DFU' for USB DFU Boot Loader or empty.
- There are other versions such as LAPI-MS1.LIB, LAPI-SLED.LIB, that are specific builds for specific hardware setups and are typically NOT included in the sample application project files. For all evaluation purposes, these files will NOT be used.

The High-level API library does not include a USB API. For the *USB 2.0 Specification* (specifically, chapter 9 of the specification), the LAPI layer handles all endpoint 0 (or control endpoint) communications with the host; thus a High-level API for USB is not necessary. An application can call the LAPI USB functions directly for all of its bulk and interrupt endpoint communications.

➢ LAPI-?xx.lib[1]	Low-level API library containing all necessary core controls.
➢ I2C_SC-?xx.lib	Low-level API library containing control of both internal and external smart card slots. This library should be included in an application that supports an I2C interface via an 8010 IC.
➢ ICC_API-?xx.lib	High-level Smart Card API library for Smart Card.

In addition, the following header (.h) and source (.c) files are included:

➢ API_12.h	Low-level header file. Include this file in all embedded applications.
➢ Api_struct_12.h	Low-level header file with all enumeration types defined.
➢ Reg12xx.h	Low-level header file specific to the 73S12xxF peripheral registers.
➢ LangIDs.h	Low-level header file. Include this file in all embedded applications for USB communication.
➢ Portable.h	Low-level header file. Include this file in all embedded applications.
➢ Reg_banks_12.h	Low-level header file. Include this file in all embedded applications.
➢ ICCMgt.h	High-level ICC header file. Include this file when the ICC_API.lib is used.
➢ Allocate.c	High-level API common file that indicates the specific number of commands that can be used for each High-level API library.
➢ Allocate.h	High-level API common header file used with allocate.c.
➢ Commands.h	High-level API common header file used with allocate.h.
➢ CCID Source	See the CCIDAP_73S12xx_V...pdf file under the CCID USB Firmware folder.
➢ P-CCID Source	See the 12xxANPseudo-CCID-SerialProtocol…pdf file in the TSC-12xxRS232\vx.xx\TSCP-CCID\Firmware\Doc folder.

[1] When the low-level LAPI-?xx.LIB library is included in an application, the compiler requires that the modules within the library be exclusively selected; otherwise, it will not be included. No error/warning will be displayed during the build but when the hex file is downloaded and run, the intended function will not operate as expected. To avoid this problem, right-click on the low-level LAPI-?xx.LIB library (after adding it to the project), select 'options for File LAPI-?xx.LIB' – and under 'Select Modules to Always Include', select the modules that the application uses, as listed. If program space permits, select all listed modules. This setup can be optimized later (once the application code is stable) by deselecting unused modules to reduce code space.

2.2.3 External Application

An external application can run on a PC or any host microprocessor. The 73S12xxF supports two options to interface with a PC external application: RS-232/Serial interface or USB V2.0 full speed interface.

RS232 / Serial Interface

When using the RS-232 or a generic asynchronous serial interface, TSC can provide the sample source code for a serial / RS-232 firmware application. It implements a SLIP protocol which enables the transfer of data between the PC (or a host microprocessor) and the 73S12xxF. The serial communication speed can range from 9600 bps to 115200 bps, with consideration of the selected CPU clock speed. Contact a Teridian Semiconductor sales representative for availability. For more information, refer to the *Pseudo-CCID Host GUI Users Guide*, the *Pseudo-CCID Host Application Guide*, and the *Pseudo-CCID Serial/RS232 Firmware Application Note*.

USB V2.0 Full Speed Interface

When using the USB V2.0 full speed interface, the communication speed is fixed at 12 Mbps. Software included with the TSC 73S12xxF Evaluation Boards provides sample source code for a USB firmware application to be run with either a Microsoft generic CCID USB driver (for XP) or the TSC customized driver (also included in the release). When the CCID-USB firmware is loaded and either the Microsoft or TSC driver is used on the host side, any PC/SC compliant application for Windows XP would be able to communicate and control the Reader. For more information, refer to the *USB-CCID Host GUI Users Guide* and the *CCID Application Note*.

2.2.4 Embedded Application

A user written embedded application can link directly to the TSC low-level API to better handle hardware control, timing or interrupts. In addition, an embedded application can be significantly simplified by using the TSC High-level API to gain access to the Smart Card protocol interfaces, especially when software certification is desired to meet USB 2.0, EMV 2000, ISO 7816 or Microsoft Windows Logo.

2.2.5 Build Environment with the Serial Boot Loader

Starting with the Teridian P-CCID Release version 2.00 or higher, the Boot Loader API is included in the Low-Level library (LAPI-?BL.lib). When linking the application with the libraries containing the boot loader, a few rules must be adhered to. Follow the instructions described below.

LAPI-?BL.lib version 3.xx supports the Boot Loader with Serial/RS-232 interface only.

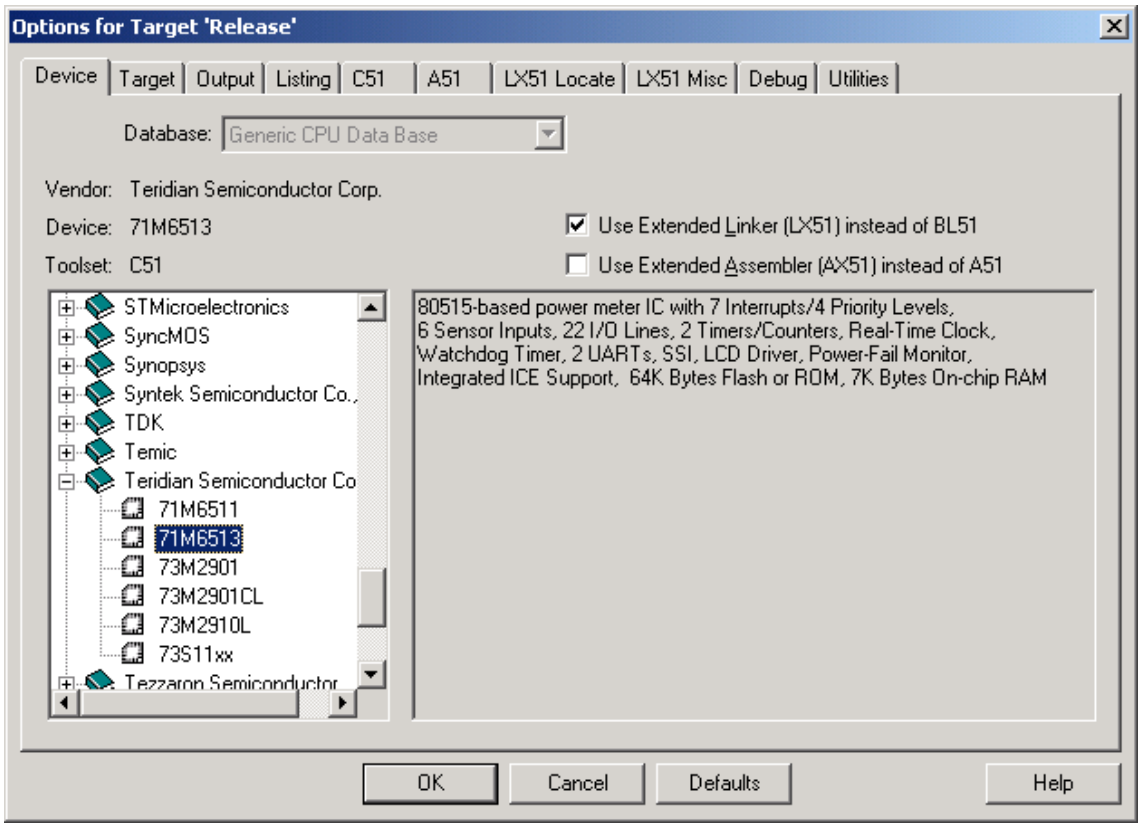

Figure 2: Device Options for Building with the Boot Loader

Figure 2 shows the Device configuration build options. The selected device should be one of the Teridian 80515-based products, either the 6513 or the 73S12xxF family.

The Extended Linker (LX51) option is required to pack both the Boot Loader and the Teridian Pseudo-CCID application within 32 K of Flash. As a result, applications that use these components must be built with this option enabled in the compiler in order for the project to build.

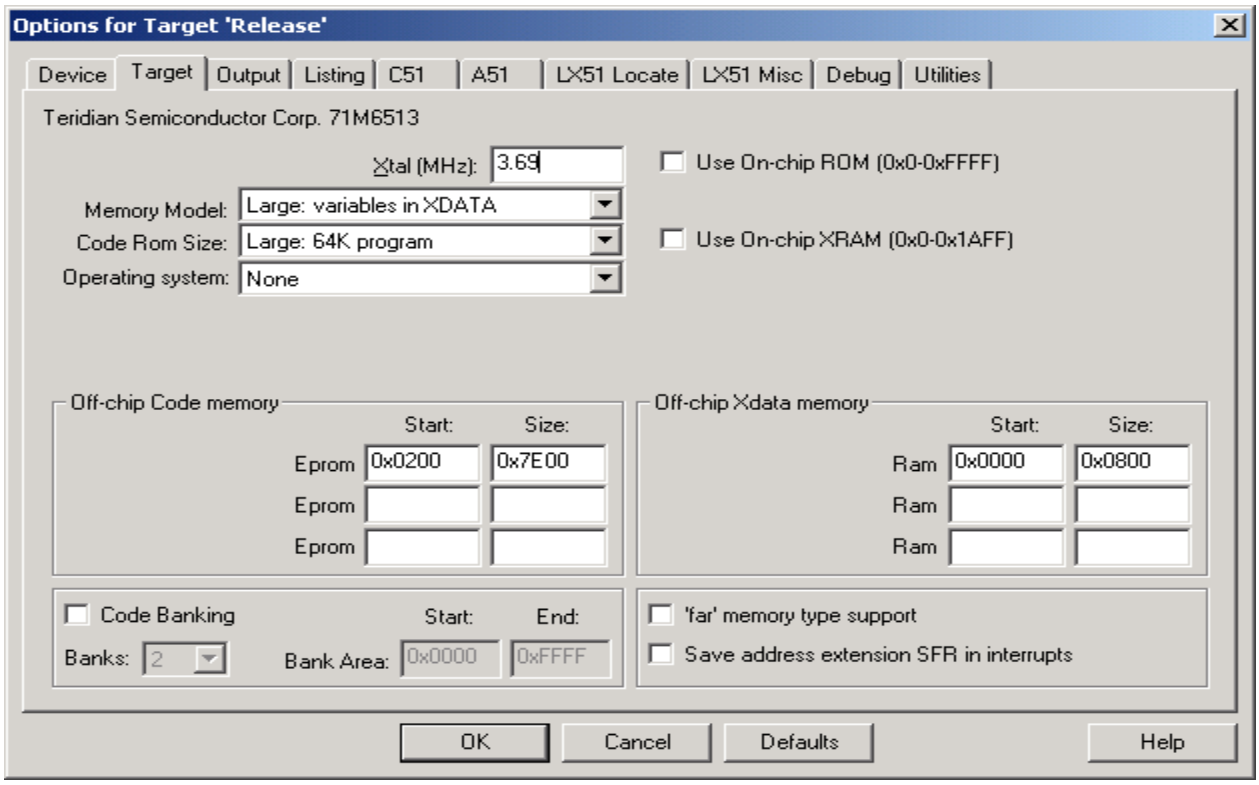

Figure 3: Target Options for Building with the Boot Loader

Off-chip code memory should be set to start at 0x0200 since the first page (512 bytes) of Flash is reserved for boot code (see Figure 3).

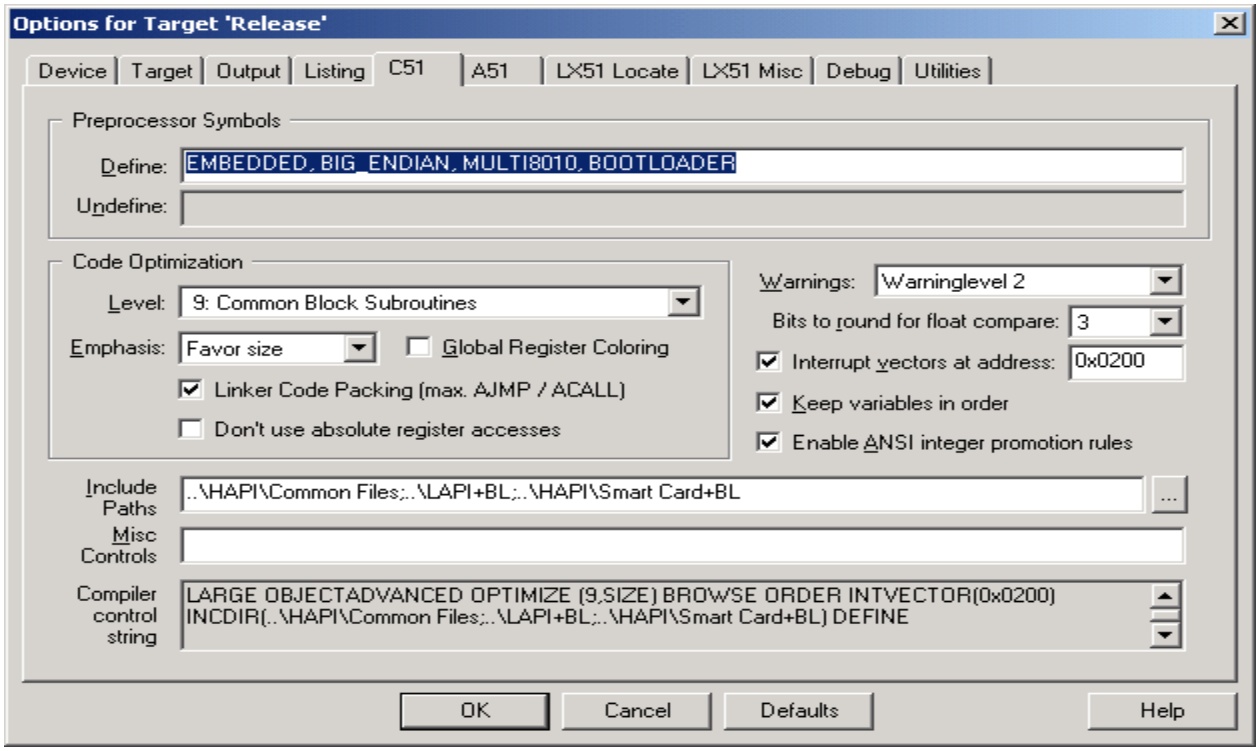

Figure 4: C51 Options for Building with the Boot Loader

The Preprocessor Symbol “MULTI8010” (refer to Figure 4) is used for a hardware configuration where multiple 8010 parts are used to drive up to four external smart card slots (any slot that is higher than ICC_1ST). Associated library or project files set up for a multiple 8010 configuration have the letter ‘M’ in their filename. For example, the Low-Level library that supports the MULTI8010 configuration would be named LAPI-MBL.lib.

The Preprocessor Symbol “SINGLE8010” is used instead of “MULTI8010” for a hardware configuration where a single 8010 and a custom mux are used to drive up to four external smart card slots. Associated library or project files set up for single 8010 configuration have the letter ‘S’ in their filename. This configuration is supported under the Serial/RS232 interface only.

The preprocessor symbol “BOOTLOADER” is used to incorporate the Serial/RS232 boot loader portion of the LAPI. This directive is used when linking to a library with ‘BL’ in its name. Interrupt vectors start at address 0x0200 due to the BOOTLOADER’s residence. All associated library or project files set up for the Boot Loader configuration have the letters ‘BL’ in their filename.

TSCP-CCID Release version 2.00 or higher (a separate release built specifically for Teridian’s Pseudo-CCID RS-232 Serial interface) has configured all its build files for the appropriate build environment and specific configuration. It is NOT recommended that these values be changed, in other words, the project files (*.uv2) should not be altered. Contact a Teridian Sales Representative for further inquiry.

2.2.6 Build Environment with the USB DFU Boot Loader

Starting with the Teridian CCID USB Release version 2.00 or higher, the DFU Boot Loader is included in the Low-Level library (LAPI-MDFU.lib). The DFU Boot Loader has a different architecture than the Serial/RS232 Boot Loader described in above section. It is a stand-alone firmware application that runs as a DFU class (0xFE) device whereas the Serial/RS-232 Boot Loader is part of the LAPI library. When linking the application with the libraries containing the DFU boot loader a few rules must be adhered to. Follow the instructions described below.

The DFU boot loader is supported under the USB interface only.

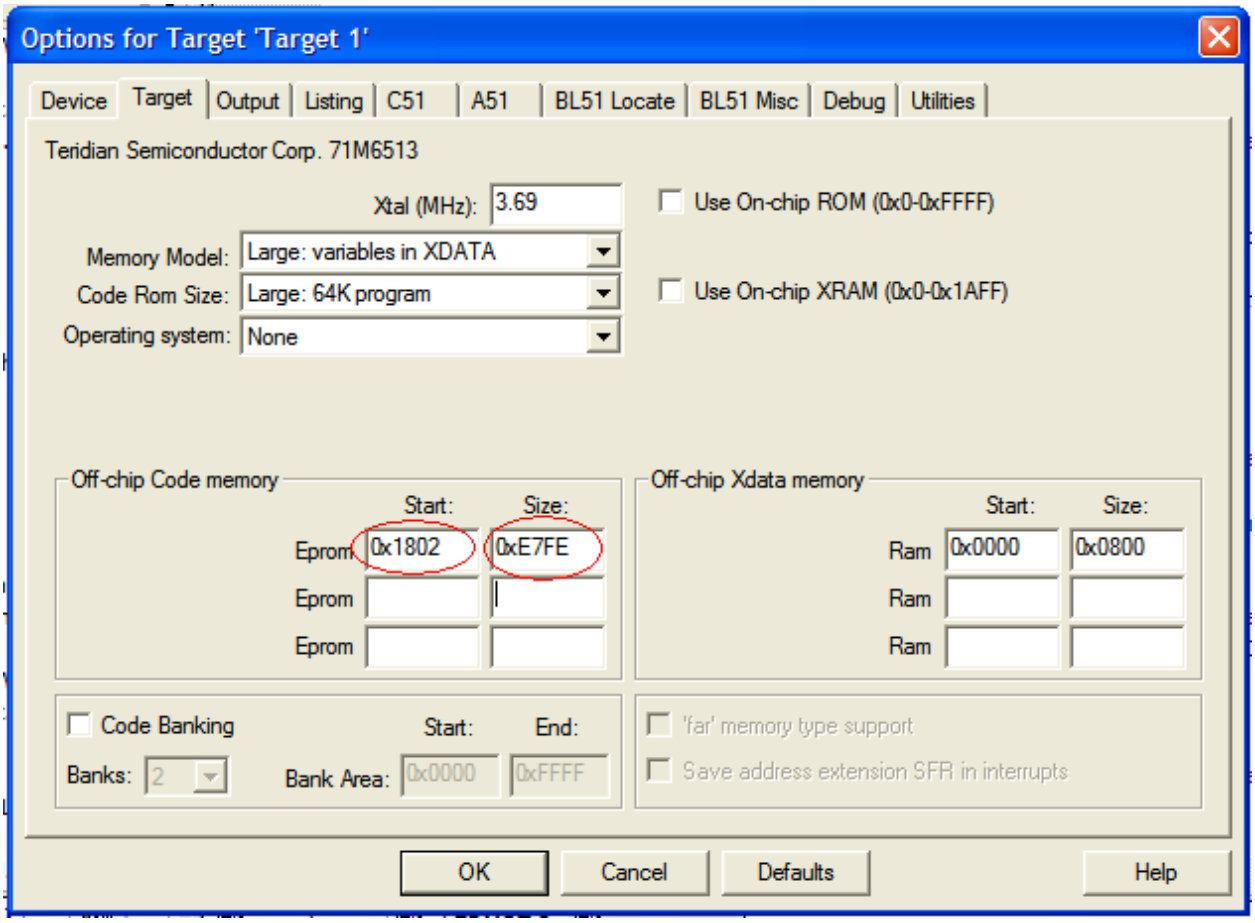

Figure 5: Target Options for Building with the DFU Boot Loader

Figure 5 shows the application's starting address. It must be set to start at 0x1802. The Flash address range from 0x0000 through 0x17FF is reserved inclusively for the DFU Boot Loader.

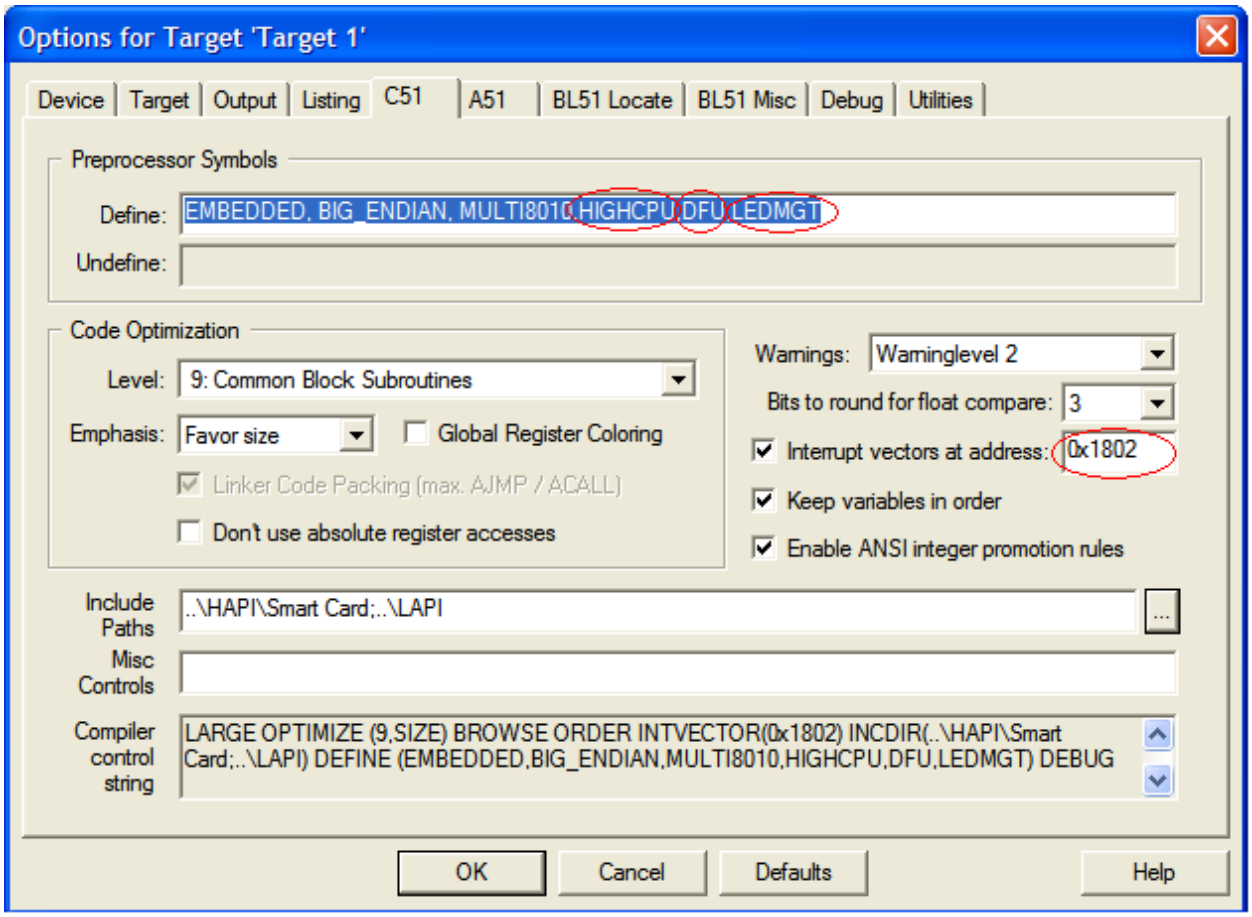

Figure 6: C51 Options for Building with the Boot Loader

In Figure 6, the interrupt vectors address must be set to where the application starts, as described above.

The Preprocessor Symbol “DFU” is used to link in the DFU option as implemented in LAPI-MDFU.LIB and ICC-API_MDFU.lib. Review the 12xxBootLoaderFirmwareANV…pdf for detailed information about this option. The associated libraries built to support this option has the letters “DFU” in their filenames.

The Preprocessor Symbol “LEDMGT” indicates status of Smart Card’s events as described in the CCIDAP_73S12xx_V…pdf. This option is only implemented and applicable at the firmware application level. Any version of the libraries would work whether this option is used or not.

The USB interface only supports “MULTI8010” hardware configuration; thus, “SINGLE8010” will not be used.

The Preprocessor Symbol “HIGHCPU” is to set the CPU clock to run at 24MHz. This option is set and used in the CCID application code. Without this option, the default CPU clock is 3.69MHz.

The source code included in the CCID release has all its build files set up for the appropriate build environment and specific configuration. It is NOT recommended that these values be changed, in other words, the project files (*.uv2) should not be altered. Contact a Teridian Sales Representative for further inquiry.

3 Testing Environment

Teridian performs conformance and certification testing to verify both the 73S12xxF IC (hardware and electrical) and libraries (protocol, timing and application firmware and drivers). These tests are dictated by specific standards. Each standard has a specific set of hardware and software configuration requirements. This section describes the protocol portion of the testing that has been performed.

3.1 EMV Level I Compliant Testing

This section describes the EMV Level I compliant protocol testing. EMV Level I Electrical testing is beyond the scope of this document.

Depending on the accredited test labs, EMV Level I compliant protocol testing can be done using either a Visa or a MasterCard test suite under the Payment System Environment (PSE). Each test environment has its own setup/configuration and selected tests for each setup.

For example, in the MasterCard test suite, there are a few configurable choices that must be considered. Each answer to these six questions would require a different test setup and expected behavior of the reader. The true/false answers shown after the questions below were selected for Teridian's EMV Level I compliant testing.

Terminal supports parameters negotiation technique: (true/false) - true
Terminal deactivates after I-Block with LEN = 0xFF: (true/false) - true
Terminal supports resynchronization: (true/false) - false
Terminal deactivates after BWT excess: (true/false) - true
Terminal deactivates after CWT excess: (true/false) - true
Terminal sends S(Abort Response): (true/false) – false

The EMV Level I Master Card test suite was performed by both an independent/accredited test lab and by Teridian's internal test lab. The testing was done using internal slot (slot #1) and all internal EMV Level I tests passed.

EMV Level I formal compliant test at an accredited lab (Cetecom) is in progress.

3.2 CCID Testing

The 73S12xxF part provides two control interfaces to the host: USB and Serial/RS232. Using either of these bus types, a command can be sent from the host to the chip to control the Smart Card. Because of this control interface, testing under the CCID Specification usually involves two parts: Smart Card and Bus type (USB or Serial/RS232).

3.2.1 USB Testing: Microsoft HCT/DTM, and USB Command Verifier

Teridian used three different tests to verify USB CCID compliance:

1. Microsoft Hardware Compatibility Test (HCT) for Windows Logo or their updated test suite: Device Test Manager for Vista and Windows XP, which tests CCID and PC/SC compliances.
2. USB 2.0 Chapter 9 Test (USB Command Verifier Test).
3. Linux Driver test using the available smart card test tool downloaded from the internet.

The HCT/DTM Smart Card test was completed in-house using the IFDTest.exe that came with HCT version 12.01.1. (Note: This test still requires the use of older PC/SC test cards that are no longer available for purchase) At this release, the two Vista drivers have been certified with Microsoft. The ad-hoc testing was performed using both the TSC USB driver and the Microsoft generic USB CCID driver. The DTM test was run and certified with Microsoft using only the TSC customized driver.

The USB Command Verifier version 1.3 Beta (latest version) was also run with this release using the TSC customized driver. The USB.ORG compliant testing was completed both in house and by an accredited, independent lab (NTS). Both tests passed.

The Linux driver was done in-house, but previously version 1.20 of the TSC CCIDUSB firmware was certified by the Linux CCIDUSB driver developer.

3.2.2 Serial Testing

PCCID is built in a separate release that includes a host application to control the 73S12xxF device programmed with the PCCID firmware. The testing was done using this proprietary interface ranging from EMV Level I protocol testing to slot-switching Smart Card Tests.

4 Design Reference

As depicted in Figure 1: Software Architecture Diagram, the 73S12xxF provides many design options for an application developer. Details of the software modules are described in this section.

4.1 Memory Map

The 73S12xxF contains a high performance, embedded 80515 core, referred to as the 'core'. It executes all ASM51 instructions and has the same instruction set as the 80C51.

The core supports separate Program and Data memory as shown in Figure 7. The 73S12xxF family of devices has two program sizes (32 KB and 64 KB). This program space is segmented into 512-byte pages. There are 2048 bytes of external data RAM (XData or XRAM) and 256 bytes of internal data RAM (IData or IRAM).

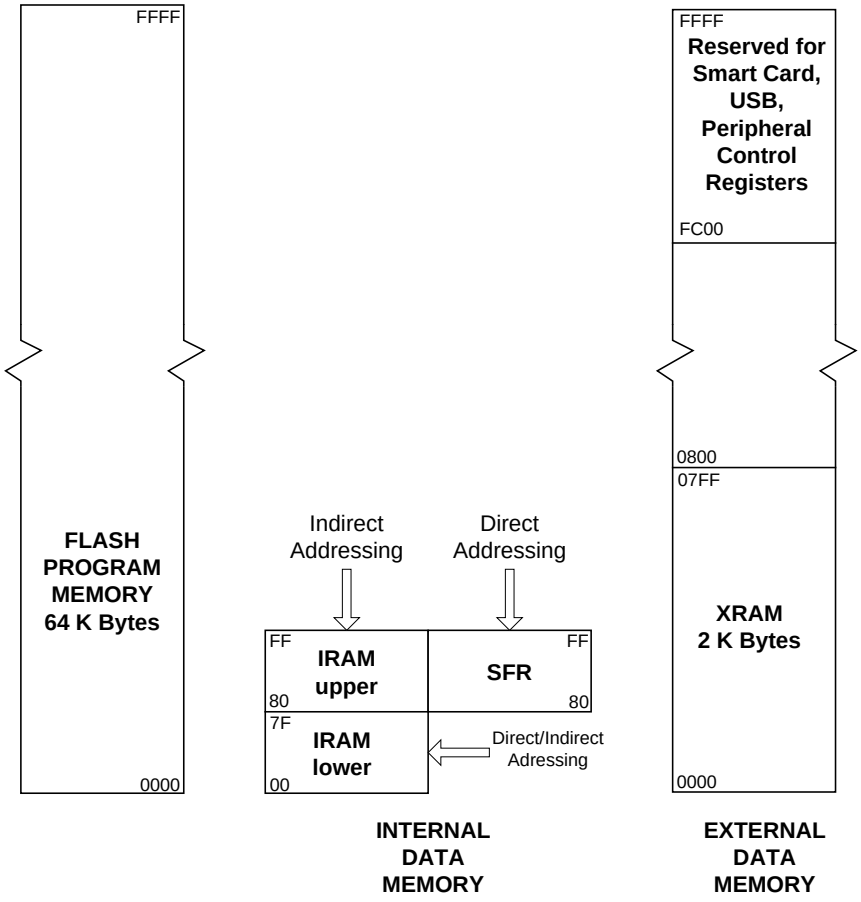

Figure 7: Memory Layout

4.1.1 Program Memory

The 73S12xxF is available with either 64 KB or 32 KB of program memory. The 73S1209F/73S1210F have 32 KB and the 73S1215F/73S1217F have 64 KB. Contact a Teridian Sales Representative for ordering information.

4.1.2 External Data Memory

The 2 KB of XRAM are available at address locations 0x0000 through 0x07FF. The upper 1K bytes of the external data memory space (from 0xFC00 to 0xFFFF) is reserved for additional special function registers for the 73S12xxF chip and is segmented as shown in Table 1.

Table 1: Upper 1 KB External Data Memory layout

Function Space	# Bytes	Starting Address	Ending Address
Peripheral Control	128	0xFF80	0xFFFF
Smart Card Control	384	0xFE00	0xFF7F
USB Control	512	0xFC00	0xFDFF

4.1.3 Internal Data Memory

The 256 bytes of IRAM are available for use as scratch-pad RAM.

The IRAM Special Function Registers are mapped as shown in Table 2. All registers appearing in the first column (000b) are bit-addressable:

Table 2: IRAM Special Function Register Map

HEX/BIN	000b	001b	010b	011b	100b	101b	110b	111b
0xF8 - 0xFF								
0xF0 - 0xF7	b							
0xE8 - 0xEF								
0xE0 - 0xE7	acc							
0xD8 - 0xDF	wdcon							
0xD0 - 0xD7	psw	kcol	krow	kscan	kstat	ksize	korderl	korderh
0xC8 - 0xCF	t2con							
0xC0 - 0xC7	ircon							
0xB8 - 0xBF	ien1	ip1	s0relh	s1relh				
0xB0 - 0xB7	p3		flshctl					pgadr
0xA8 - 0xAF	ien0	ip0	s0rell					
0xA0 - 0xA7	usr158	udir158						
0x98 - 0x9F	s0con	sobuf	ien2	s1con	s1buf	s1rell		
0x90 - 0x97	usr70	udir70	dps		erase			
0x88 - 0x8F	tcon	tmod	tl0	tl1	th0	th1	ckcon	mclkctl
0x80 - 0x87		sp	dp1	dph	dpl1	dph1	wdtrel	pcon

4.2 Low-level API

A low-level Application Programming Interface (API) is provided to control all hardware peripherals. An application can link to the low-level API library to utilize the functions described in subsequent sections.

Before using the low-level API, an embedded application should call the API_Init() routine in the beginning of the program as part of its initialization. As a general guideline, any feature to be included or used in an application must first call its associated initialization routine. For example, to link the USB module into an application, USB_Init() must first be called to initialize its associated registers for the USB functions.

Each feature available with the 73S12xxF device has a sample application to demonstrate the library function's usage.

The core provides four interrupt priority levels among six group sources. The 73S12xxF sources are grouped as shown in Table 3. All sources belonging to a group share the same interrupt priority level. The LAPI sets up the four priority levels for these groups as shown in Table 3.

Table 3: Interrupt Sources and Priority Level

Group 6	Group 5	Group 4	Group 3	Group 2	Group 1
I2C	USB	Smart Card	Ext Int 1	Timer 0	Ext Int0
VDD	RTC	Timer 1	Ext Int 3	Ext Int 2	Serial 1
Analog	Keypad				
	Serial 0				
Priorities as set in LAPI.LIB:					
Highest	Low	High	Lowest	Lowest	Low

The low-level APIs are listed below. Additional details are in the subsequent subsections.

- Keyboard Driver API – Available with all 73S12xxF Devices (page 21).
- LCD Driver API – Available with all 73S12xxF Devices (page 23).
- LED Driver API – Available with all 73S12xxF Devices (page 24).
- Real Time Clock API - Available with the 68-pin 73S12xxF (page 26).
- Smart Card Interface Driver API – Available with all 73S12xxF Devices (page 30).
- SERIAL (RS232) Driver API – Available with all 73S12xxF Devices (page 39).
- USB API – Available with 64K Flash version of the 73S12xxF (page 42).
- Clock Generator Circuit API – Available with all 73S12xxF Devices (page 51).
- Power Management API – Available with all 73S12xxF Devices (page 52).
- Analog Threshold Management Driver API – Available with all 73S12xxF Devices (page 53).
- Event Management API – Available with all 73S12xxF Devices (page 55).
- Timers API – Available with all 73S12xxF Devices (page 57).
- User IO API – Available with all 73S12xxF Devices (page 58).
- External Interrupts API – Available with all 73S12xxF Devices (page 60).
- Special Function Register API – Available with all 73S12xxF Devices (page 61).
- Flash/Memory API – Available with all 73S12xxF Devices (page 63).
- Boot Loader and Passcode Management – Available with the LAPI-*BL.lib Only (page 67).
- Security Mode Management - Available with the LAPI-*BL.lib Only (page 69).
- Other Miscellaneous API Calls – Available with all 73S12xxF Devices (page 71).

4.2.1 Keyboard Driver API – Available with all 73S12xxF Devices

The Keyboard Driver manages the keystroke acquisition using a scrambled algorithm. It is the High-level API's role to manage the scrambling algorithm. Up to 30 keys can be managed in a 6 row by 5 column configuration. The APIs below are written to use Hardware Scan enabling. An application can be written to bypass Hardware Scanning to perform its own manual key scan functionality. Refer to the *73S12xxF Data Sheet* for information on the keypad bypass mode.

The Keyboard Driver API includes:

- KEY_Init () (page 22)
- KEY_Wait () (page 22)

KEY_Init ()

Purpose — Configure the keyboard. This API will call the Set_Event() routine to enable an Interrupt Service Routine (ISR) to handle keyboard scanning.

Synopsis — **Void KEY_Init (** IN unsigned char rows_cols, IN unsigned char debounce_scan **)**;

Parameters —
RowsCols: Input parameter
Number of rows on keypad (max value is 6) times eight plus number of columns on keypad (max value is 5), i.e. rows*8 + cols.
Debounce_scan: Input parameter
Number of milliseconds to debounce key (granularity is 4ms) plus number of milliseconds between key scans (1 to 4 ms) 1 = 1 ms; 2 = 2 ms; 3 = 3 ms and 0=4 ms.

Return Codes — None.

KEY_Wait ()

Purpose — Wait for a keypad input during a maximum time specified by the TimeOut value. Other events (e.g. Smart Card insertion and removal) can be specified as exit conditions for this function.

Synopsis —
KEY_RC KEY_Wait (
IN Unsigned Int ScanOrder,
IN Unsigned char TimeOut,
IN Unsigned Int ExitOnICCInsert,
IN Unsigned Int ExitOnICCRemoval,
IN Unsigned Word ExitOnEvent,
OUT Unsigned Word *ExitOn,
OUT Unsigned char *KeyCode **)**;

Parameters —
ScanOrder: Input parameter
Column scan order, in five sets of three bits each, with the least significant 3-bits (0-2) indexing the first column scanned and bits (12-14) indexing the fifth column scanned.
TimeOut: Input parameter
Timeout value in seconds. If no key is entered within this period, the function is aborted.
ExitOnICCInsert: Input parameter
Specifies which, if any, SmartCard insertion events are exit conditions. Bit[n] corresponds to ICC[n], bit[1] being mapped to the least significant bit. This option should NOT be used simultaneously with the *ExitOnICCRemoval* option.
ExitOnICCRemoval: Input parameter
Specifies which, if any, SmartCard removal events are exit conditions. Bit[n] corresponds to ICC[n]. This option should NOT be used simultaneously with the *ExitOnICCInsert* option.
ExitOnEvent : Input parameter
On input, specifies which, if any, other events are exit conditions. Possible values are any combination of the following:

EVENT_EXT0	0x00000001	
EVENT_EXT1	0x00000002	
EVENT_EXT2	0x00000004	
EVENT_EXT3	0x00000008	
EVENT_TIMER0	0x00000010	
EVENT_TIMER1	0x00000020	
EVENT_ICC	0x00000040	
EVENT_RTC	0x00000080	//not available with 73S1205F
EVENT_KEY_DETECT	0x00000100	
EVENT_USB	0x00000200	//not available with 73S1205F

EVENT_VDDF	0x00000400
EVENT_I2C	0x00000800
EVENT_ANALOG	0x00001000
EVENT_USR0	0x00002000
EVENT_USR1	0x00004000
EVENT USR2	0x00008000
EVENT_USR3	0x00010000
EVENT_ES	0x00020000

ExitOn: Output parameter
If KEY_ERR_SMARTCARD_xxx return code, it specifies which SmartCard event occurred. Bit[n] corresponds to ICC[n]. If KEY_ERR_EVENT, it specifies which EVENT occurred.

KeyCode: Output parameter
Specifies the KeyCode that was pressed. The KeyCode is equal to ((row-1) * KeypadCols) + (col-1), where **column** ranges from 1 to KeypadCols and **row** ranges from 1 to KeypadRows.

Return Codes

KEY_OK	Successful operation: a valid key was pressed.
KEY_ERR_TIMEOUT	TimeOut error.
KEY_ERR_SMARTCARD_INSERTED	SmartCard insertion detected.
KEY_ERR_SMARTCARD_REMOVED	SmartCard removal detected.
KEY_ERR_EVENT	

ScanOrder can be any permutation of the values 0,1,2,3 and 4. If an unscrambled order is desired, set ScanOrder = 0x4688. The scrambling algorithm is handled by the caller. If the event was a Smart Card event exit, use ICC_Status() to discover which card caused the exit.

4.2.2 LCD Driver API – Available with all 73S12xxF Devices

The LCD interface supports a generic external LCD controller and uses USR I/O that is accessed by the LCD driver API. The LCD calls manage a generic 7-bit (4-bit data bus) interface to the external LCD controller. For the 73S1215F Evaluation Board, where the MDL-16265 LCD is used, USR pins 0-3 are used for the 4-bit data bus, USR pins 4, 5 and 6 are used for E, RW and RS, respectively).

The USR IO pins can be used for several different features such as LCD, I2C addressing for external slots and a Serial RS232 interface to a Windows XP host depending on the TSC evaluation board and the application firmware being used. Care must be taken by the application to make sure there is no conflicting usage. The LCD API includes:

LCD_Init ()

Purpose: Initialize the LCD interface. After initialization the Display will be ON and cleared. A block cursor will be at the home position.

Synopsis: **Void LCD_Init (** void **);**

Parameters: None.

Return Codes: None.

LCD_Command ()

Purpose Send command to LCD.

Synopsis **Void LCD_Command (** IN char LcdCmd **)**;

Parameters *LcdCmd*: Input parameter
8-bit command to control the LCD. Available commands are:

LCD_CLEAR	Clear display and return cursor to home.
LCD_HOME	Return display and cursor to home position.
LCD_MODE | LCD_INC	Increment cursor on read/write of display.
LCD_MODE | LCD_INC|LCD_SHIFT	Increment cursor and shift display on read/write of display.
LCD_CTRL | LCD_DON | LCD_CON	Display ON and cursor ON (visible).
LCD_CTRL | LCD_DON	Display ON and cursor OFF (invisible).
LCD_CTRL | LCD_BON	Display ON and blinking ON.
LCD_CURSOR	Shift cursor left.
LCD_CURSOR | LCD_RL	Shift cursor right.
LCD_CURSOR | LCD_SC	Shift display left.

Return Codes None.

LCD_Data_Write ()

Purpose Write data to LCD.

Synopsis **Void LCD_Data_Write (** IN char LcdData **)**;

Parameters *LcdData* : Input parameter
8-bit data written to the LCD at the current cursor position.

Return Codes None.

LCD_Data_Read ()

Purpose Read data from the LCD.

Synopsis **Void LCD_Data_Read (** OUT char *LcdData **)**;

Parameters *LcdData* : Output parameter
8-bit data read from the LCD at the current cursor position.

Return Codes None.

4.2.3 LED Driver API – Available with all 73S12xxF Devices

The 73S12xxF provides four LEDs that can be programmed with four levels of output current: 0 mA (LED_OFF), 2 mA (LED_DIM), 4 mA (LED_NORMAL) and 10 mA (LED_BRIGHT). On the 73S1205F, only LED0 and LED1 (lower two bits) are available.

The LED Driver API includes:

- LED_Config () (page 25)
- LED_Write () (page 25)
- LED_Read () (page 25)

LED_Config ()

Purpose Configure the LED interface (all pins).

Synopsis **Void LED_Config (** IN Bbool PU_Enable, IN enum LED_CURRENT LC **);**

Parameters *PU_Enable*: Input parameter
Boolean value specifies enable (TRUE) or disable (FALSE) pull-up.
LC: Input parameter
Enum type indicating the output current level for the LEDs (when turned on). The following values are available:

LED_OFF:	turn off LED (0 mA).
LED_DIM:	turn LED on at dim level (2 mA).
LED_NORMAL:	turn LED on at normal level (4 mA).
LED_BRIGHT:	turn LED on at brightest level (10 mA).

Return Codes None.

LED_Write ()

Purpose Turn LED on/off. If turned on, the brightness control is possible at one of three levels: Dim, Normal, Bright as specified in LED_Config().

Synopsis **Void LCD_ Write (** IN unsigned char LED_Pin, IN unsigned char Value **)**;

Parameters *LED_Pin* : Input parameter
Selected LED(s) to be turned on, where bit[n] = 1 indicates LED[n] to be written.
Value: Input parameter
Value to be written to selected LED. Bit[n] is the value to be written to LED[n] if LED[n] is selected to be written to.

Return Codes None.

LED_Read ()

Purpose Read current state of the LED data.

Synopsis **Void LCD_Read (** OUT unsigned char LEDValue **)**;

Parameters *LEDvalue*: Output parameter
Bit[0] indicates the current state of LED0.
Bit[1] indicates the current state of LED1.
Bit[2] indicates the current state of LED2 – **Not available in 73S1205F**.
Bit[3] indicates the current state of LED3 – **Not available in 73S1205F**.
Bit[4] indicates the state of the pull-up (1=enable).
Bit[5,6] indicate the output current level where:
00 = OFF
01 = DIM
10 = NORMAL
11 = BRIGHT

Return Codes None.

4.2.4 Real Time Clock API - Available with the 68-pin 73S12xxF

The 73S12xxF provides a 32-bit counter selectable in 0.5, 1 or 2 second increments to measure time. This time mark can also be used to generate RTC interrupts at 0.5, 1, 2, 4, or 8-second intervals. A 24-bit trimming function, along with a 24-bit accumulator, is provided to correct the clock drift induced by the quartz crystal. The device also supports a watchdog capability. This feature will give the processor 0.5 seconds to respond to an RTC interrupt. If the RTC interrupt is not serviced within 0.5 seconds, a full RESET to the 72S12xxF is performed. To use the watchdog timer function, the RTC interrupt must be enabled. Consequently, the watchdog will always be enabled when the RTC interrupt is enabled. It is not possible to turn off the watchdog while the RTC interrupt is enabled.

Care should be taken as it is possible for the device to be put into an infinite reset loop when an RTC interrupt is not serviced on time (within 0.5 second). When this problem occurs, reprogram the Flash with a known good application/program using the TFP.

The RTC block uses the 32768 Hz oscillator signal or divider logic (from the 12 MHz oscillator circuit) to produce 0.5 second time marks. The 32768 Hz oscillator can be disabled (see the PowerOFF API), but is intended to operate at all times in all power consumption modes. If a 32 kHz crystal is not provided, this oscillator must be disabled and the RTC will operate from an internal 96 MHz clock divided by 2930. In this case, the RTC trim value should be set as described in the following paragraph.

The 3-byte accumulator can hold 2^{24} = 16,777,216, which yields a 0.0596 PPM resolution. Using the 12 MHz oscillator gives 96 MHz/2930 which will generate 32,764 Hz. The core RTC uses a 32,768 Hz Oscillator crystal. This yields a 106 PPM error. Therefore, the trim value = (106 PPM) / (.0596 PPM) ~= 1778 (0x06F2).

When the accumulator reaches overflow, it will advance the counter one additional count if the trim value is positive, or prevent the counter from advancing one count if the trim value is negative.

The trim value can be set in the API_12.h file. RTClk_Init () must be called prior to using any of the RTC APIs. Once the RTC is initialized, the 32 kHz OSC clock will always be running. To turn it off, use the PowerOFF (DISABLE_RTC) API.

The Real Time Clock API is described in detail below and includes:

RTClk_Init ()

Purpose — Initialize the Real Time Clock values by setting the accumulator to 0 and setting the trim values as defined in api_12.h. The default base day is defined as 12:00:00, 01/01/2005 calculated using the Gregorian/Julian conversion defined in: http://webexhibits.org/calendars/calendar-christian.html.

When this function is called, the RTC is stopped and restarted. The RTC counter, trim and accumulator will be loaded at the next 32 kHz clock positive edge. The RTC interrupt is NOT set here. Use RTClk_Control() to set the interrupt. The RTC counter continues to count whether the RTC interrupt is enabled or not.

The Interrupt service routine for the RTC interrupt can be masked using the Set_Event (eRTC, pRTCVector) API. If customization of the RTC ISR is desirable, call Set_Event after this API is called. See Set_Event() for its usage description.

Synopsis **Void RTClk_Init (** void **);**

Parameters None.

Return Codes None.

RTCClk_Control ()

Purpose Enable or disable the RTC interrupt. If enabled, the interrupt interval must be specified.

Synopsis **RTCClk_Control (** IN Bbool RTCInt_Enb, IN enum RTC_INTERVAL intv **)**;

Parameters *RTCInt_Enb:* Input parameter
Enable (TRUE) or disable (FALSE) the RTC interrupt. If set to Enable when the *intv* parameter is set to NO_INT, the RTC interrupt will NOT be enabled.

Intv: Input parameter
The RTC interrupt interval as defined in API_STRUCT_12.h as follows:

HALF_SEC	Interrupt to occur every ½ second.
ONE_SEC	Interrupt to occur every 1 second (default).
TWO_SEC	Interrupt to occur every 2 seconds.
FOUR_SEC	Interrupt to occur every 4 seconds.
EIGHT_SEC	Interrupt to occur every 8 seconds.
NO_INT	No interrupt.

Return Codes None.

The watchdog timer will give the processor ½ second to respond to an RTC interrupt. If the RTC interrupt is not serviced within this timeframe, a full reset will be performed.

RTClk_Write ()

Purpose Initialize the Real Time Clock control, counter, accumulator and trim registers. When this function is called, the RTC is stopped and restarted.

Synopsis **Void RTClk_Write (** IN struct RTC_t *pRTC **)**;
Where RTC_t is defined as:

```
struct RTC_t
{
    Unsigned char  RTCCtl;
    Unsigned long  RTCCnt;
    Unsigned char   RTCAcc[3];
    Signed char    RTCTrim[3];
}
```

Parameters *RTCCtl*: Input parameter
BIT[7-6]: not used
BIT[5]: RTCLoad – when set, RTCCnt, RTCAcc and RTCTrim are loaded at the next 32kHz clock positive edge.
BIT[4-3]: Set tic interval as follows:
0x – 1 second
10 – ½ second
11 – 2 seconds
BIT[2-0]: Set interrupt interval as follows:
100 – ½ second
0xx – 1 second
101 – 2 seconds
110 – 4 seconds
111 – 8 seconds

RTCCnt: Input parameter
32-bit RTC counter value.

RTCAcc[3]: Input parameter
24-bit accumulator value. Normally these values are to be initialized only once during the manufacturing phase.

RTCTrim[3]: Input parameter
24-bit signed trimmer value. This is the offset value used to correct the quartz crystal drift. It is the number of ticks between each correction of the Real Time Clock. Use a negative numbers to decrease the tic-count by one and a positive number to increase the tick-count by 1.

Return Codes None.

The RTC can be enabled and disabled via the PowerON() and PowerOFF() or RTCClk_Control () functions. The new values will not be loaded until the RTCLoad bit (bit 5) of the RTCCtl register is set (HI).

RTClk_Read ()

Purpose Extract the current Real Time Clock control, counter, accumulator, and trimmer values.

Synopsis **Void RTClk_Read (** IN struct RTC_t *pRTC **)**;

```
struct RTC_t
{
    Unsigned char  RTCCtl;
    Unsigned long  RTCCnt;
    Unsigned char  RTCAcc[3];
    Signed char    RTCTrim[3];
}
```

Parameters *RTCCtl:* Output parameter
Current Real Time Clock Control register value (setting).

RTCCnt: Output parameter
Current Real Time Clock Counter value.

RTCAcc[3]: Output parameter
Current Real Time Clock accumulator value.

RTCTrim[3]: Output parameter
Current Real Time Clock Trimmer value.

Return Codes None.

RTCClk_GetTIME ()

Purpose Extract current calendar Time. Time conversion is done by the Gregorian/Julian conversion method as defined on website: http://webexhibits.org/calendars/calendar-christian.html.

Synopsis **Void RTCClk_GetTIME (** struct C_RTC_t xdata *pRTC_Time **)**

```
struct C_RTC_t
{
    Unsigned char          Sec;
    Unsigned char          Min;
    Unsigned char          Hour;
    Unsigned char          Date;
    enum MONTH             Month;
    Unsigned integer       Year;
    enum RTC_INTERVAL      TicInterval;    //Tic interval - 1, 1/2 or 2 sec
    enum RTC_INTERVAL      IntInterval;    //int interval. NO_INT=disable int.
};
```

Where MONTH is defined as: enum { JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC }; and RTC_INTERVAL is defined as:
enum {HALF_SEC, ONE_SEC, TWO_SEC, FOUR_SEC, EIGHT_SEC, NO_INT};

Parameters

Sec: Output parameter
Current second unit.
Min : Output parameter
Current minute unit.
Hour: Output parameter
Current hour unit.
Date: Output parameter
Current date unit.
Month: Output parameter
Current month unit as specified in the enum MONTH type.
Year: Output parameter
Current year unit, e.g. 2005.
TicInterval: Output parameter
Tic interval as HALF_SEC, ONE_SEC or TWO_SEC, defined in RTC_INTERVAL.
IntInterval: Output parameter
Interrupt interval as defined in RTC_INTERVAL.

Return Codes None.

RTCClk_SetTIME ()

Purpose Set time and start clocking immediately. Time conversion is done by the Gregorian/Julian conversion method as defined on website: http://webexhibits.org/calendars/calendar-christian.html.

Synopsis **Bbool RTCClk_SetTIME (** struct C_RTC_t xdata *RTC_Time **)**

```
struct C_RTC_t
{
    Unsigned char        Sec;
    Unsigned char        Min;
    Unsigned char        Hour;
    Unsigned char        Date;
    enum MONTH           Month;
    Unsigned integer     Year;
    enum RTC_INTERVAL    TicInterval;     //Tic interval - 1, 1/2 or 2 sec
    enum RTC_INTERVAL    IntInterval;     //int interval. NO_INT=disable int.
};
```

Where MONTH is defined as: enum { JAN=1, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC }; and RTC_INTERVAL is defined as: enum {HALF_SEC, ONE_SEC, TWO_SEC, FOUR_SEC, EIGHT_SEC, NO_INT };

Parameters

Sec: Input parameter
Current second unit.
Min: Input parameter
Current minute unit.
Hour: Input parameter
Current hour unit.
Date: Input parameter
Current Date unit.
Month: Input parameter
Current month unit as specified in the enum MONTH type.

Year: Input parameter
Current year unit, e.g. 2005.
TicInterval: Input parameter
Tic interval as HALF_SEC, ONE_SEC or TWO_SEC, defined in RTC_INTERVAL.
IntInterval: Input parameter
Interrupt interval as defined in RTC_INTERVAL.

Return Codes TRUE if success. FALSE if TicInterval or Interrupt Interval value is invalid.

4.2.5 Smart Card Interface Driver API – Available with all 73S12xxF Devices

The Smart Card Interface Driver API manages all the Smart Card interfaces. Each of the smart card slots can be individually activated, deactivated, etc. since most of the functions take the ICC identifier as an input. This API handles the physical layer, i.e., the inter-character and inter-block timeouts. Optionally, it can handle the LRC/CRC computation.

To switch between multiple activated cards, re-initialize the selected card (using the eIccId parameter). ICC_1ST refers to the internal Smart Card #1. ICC_2ND or higher refers to external slot #2 with I2C interface (8010 interface) and uses USRIO as its associated address.

When developing an application with an internal interface only (1 slot only), include libraries LAPI.LIB and Internal-SC.LIB. When developing an application with interface to an internal interface and/or external slot(s), include libraries LAPI.LIB and I2C-SC.LIB (replace Internal-SC.lib with I2C-SC.LIB).

For an external I2C interface, it is necessary to assign the I2C address, and I2C Card Event signal as specified by the board design (see ICC_InitUART() for more details).

The Smart Card Interface API includes:

Follow the general procedure described below to communicate with any asynchronous ICCs present:

1. Initialize the Smart Card UART parameters for each selected card.
2. Activate the card(s) that were initialized in Step 1.
3. Re-initialize the Smart Card UART parameters for each card, based on ATR analysis.
4. Negotiate the protocol and/or Fi/Di values for the selected cards. (Optional)
5. Update the Smart Card parameters based on the protocol negotiation if performed.
6. Re-initialize the selected card.
7. Transmit requests to a selected card via ICC_Tx call(s).
8. Receive responses from a selected card via ICC_Rx call(s).
9. Repeat Steps 6, 7 and 8 as needed for any activated card.
10. Deactivate active card(s).

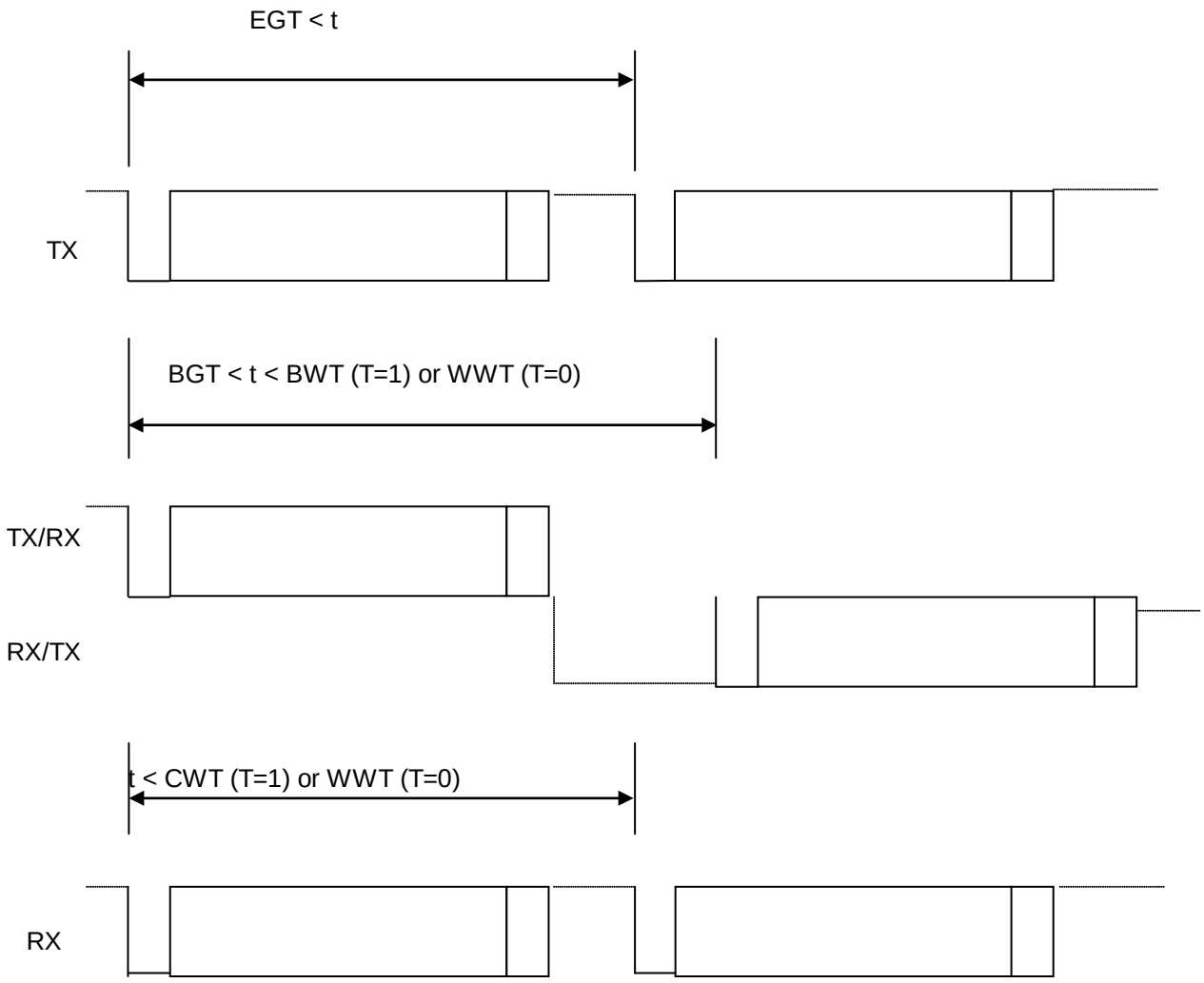

Figure 8: Smart Card Rx/Tx Timing

ICC_InitUART()

Purpose Initializes the Smart Card UART for the specified ICC.

Synopsis **Void ICC_InitUART (** IN struct ICC_Init_t *pICC_Init **)**;

```
Struct {
    enum ICC_ID IccId,
    Boolean IccDetect,
    enum ICC_HZ IccHz,
    unsigned char FiDi,
    Boolean IccProtocol,
    Boolean IccEDCEnabled,
    Boolean IccEDCTypeCRC,
    Boolean IccParityCheck,
    Boolean IccBreakGeneration,
    unsigned char IccBreakStartPosition,
    unsigned char IccBreakDuration,
    unsigned char IccRxRetry,
    Boolean IccBreakDetect,
    unsigned char IccTxRetry,
```

```
    int IccExtraGuardTime,
    int IccBlockGuardTime,
    int IccCharWaitingTime,
    long int IccBlockWaitingTime,
    long int IccWorkWaitingTime
    Boolean IccPUEnabled; Boolean IccPDEnabled;
    Boolean IccVDDFaultOff;.
    enum ICC_ADDR IccAddr;              // Useful for external I2C.
    enum ICC_RESET IccExtRst;           // Useful for external I2C.
    enum ICC_CARDEVENT IccCE;           // Useful for external I2C.
} ICC_Init_t;
```

Parameters

IccId: Input parameter
Specifies which SmartCard UART interface is to be initialized. Possible values are:

ICC_1ST 0 (Internal)
ICC_2ND 1 (External)
…
ICC_9TH 8 (External)

IccDetect: Input parameter
Specifies the detect polarity of the card present. Specify TRUE to detect a card present when the DET_CARDx pin is HIGH. Specify FALSE to detect a card with the DET_CARDx pin is LOW. This parameter is only valid for the internal ICC interface.

IccHz: Input parameter
Specifies the smart card frequency. Possible values are:

ICC_3600KHZ (0)
ICC_1800KHZ (1)
ICC_7200KHZ (2)

The LAPI supports these three rates to follow the same design as the older device family (73S11xx). The 73S12xxF hardware supports a wider range of smart card clock generations, which can be derived using a dividing factor (F) such that:
SC clock = 96 MHz / (F + 1) / 2 where F can be any value used to generate the desired clock. In the three cases above, the F values were defined as: 12, 24 and 6 respectively. See the data sheet for more information.

FiDi: Input parameter
Specifies the current Fi/Di values, equivalent in format to the PPS1 byte of a PPS request.

IccProtocolT1: Input parameter
Specifies the current protocol, T=1 (TRUE) or T=0 (FALSE). Used to set appropriate guard and wait times.

IccEDCEnabled: Input parameter
Specifies if the EDC byte(s) are to be updated on the fly.

IccEDCTypeCRC: Input parameter
Specifies if the EDC value is to be calculated with the CRC algorithm (TRUE) or with the LRC algorithm (FALSE).

IccParityCheck: Input parameter
Specifies if the parity bit is to be checked (TRUE) or not (FALSE).

IccBreakGeneration: Input parameter
Specifies whether the UART must generate a break on a parity error (TRUE) or not (FALSE).

IccBreakStartPosition: Input parameter
Specifies where the break signal should start. Possible values are 0x00 through 0x07 (in 0.125 ETU increments), which are associated with bit positions 10 through 10.875 ETUs. For example, 1 corresponds to 10.125 ETUs. The default value is 0x04 (10.5 ETUs).

IccBreakDuration: Input parameter
Specifies the break signal duration. Possible values are 0x00 (1 ETU), 0x01 (1.5 ETU) and 0x02 (2 ETU). The default value is 0x00 (1 ETU).

IccRxRetry: Input parameter
The number of retries to allow on reception of a bad byte. Retries = total tries – 1.

IccBreakDetect: Input parameter
Specifies whether to acknowledge (TRUE) or ignore (FALSE) a break coming from the Smart Card.

IccTxRetry: Input parameter
The number of retries to allow on transmission of a bad byte. Retries = total tries – 1.

IccExtraGuardTime: Input parameter
Specifies the minimum delay in ETUs between the leading edges of two consecutive characters sent to the Smart Card.

IccBlockGuardTime: Input parameter
Specifies the minimum delay in ETUs between the leading edges of two consecutive characters sent in opposite directions.

IccCharWaitingTime: Input parameter
Specifies the maximum delay in ETUs between the leading edges of two consecutive characters from the smart card (T=1 protocol).

IccBlockWaitingTime: Input parameter
Specifies the maximum delay in ETUs between the leading edges of two consecutive characters sent in opposite directions. Possible values are in the range 0x01 through 0x078000.

IccWorkWaitingTime: Input parameter
Specifies the maximum delay in ETUs between the leading edges of two consecutive characters from the Smart Card or of one sent to the Smart Card and the next one received from it (T=0 protocol).

IccPUEnabled: Input parameter
Enables (TRUE) or disables (FALSE) the Smart Card pull-up current source on the Card Detect pin.

IccPDEnabled: Input parameter
Enables (TRUE) or disables (FALSE) the Smart Card pull-down current source on the Card Detect pin.

IccVDDFaultOff: Input parameter
When enabled (1), allows the 73S12xxF to automatically perform a deactivation sequence when there is a VDD Fault. When disabled (0), allows the 73S12xxF to signal the companion circuit to set its VCC=0 when there is a VDD Fault.

IccAddr: Input parameter
The following settings are defined in API_STRUCT_12.h

ICC_INTERNAL – Use this value for slot #1 (Internal slot)
ICC_I2C_0 = 0x40, – Any of these values can be used for I2C, slot # > ICC_1ST
ICC_I2C_1 = 0x42,
ICC_I2C_2 = 0x44,
ICC_I2C_3 = 0x46,
ICC_I2C_4 = 0x48,
ICC_I2C_5 = 0x4A,
ICC_I2C_6 = 0x4C,
ICC_I2C_7 = 0x4E

IccExtRst: Input parameter
External Reset signal. The following settings are defined in API_STRUCT_12.h:

ICC_NRST = 0x00, – Use this value for slot #1 (Internal slot)
ICC_RST0 = 0x80,
ICC_RST1 = 0x90,
ICC_RST2 = 0xA0,

ICC_RST3 = 0xB0,
ICC_RST4 = 0xC0,
ICC_RST5 = 0xD0,
ICC_RST6 = 0xE0,
ICC_RST7 = 0xF0

IccCE: Input parameter
Card Event parameter. The following settings are available as defined in API_STRUCT_12.h. Set this variable according to the hardware design where either interrupt 2 or interrupt 3 is used for an external card event detect.

ICC_INT2_NONE = 0x00,
ICC_INT2_I2C = 0x01, //Use INT2 for card event detection
ICC_INT3_I2C = 0x02, //Use INT3 for card event detection

Return Codes None.

The T=0 and T=1 protocols affect which guard and wait times to use. For protocols other than these, use whichever protocol (either T=0 or T=1) best matches the timing requirements. If neither protocol matches, then the application will need to bypass the Smart Card UART.

ICC_Activate()

Purpose Activate the contacts of the selected Smart Card, as specified by eIccId. The VCC, RST, I/O and CLK signals are configured according to the ISO 7816-3 standard.

Synopsis **ICC_RC ICC_Activate (** IN struct ICC_Activate_t *pActivate, IN struct ICC_t *pATR **)**;

```
Struct {
    enum  ICC_VOLTAGE IccVCC;
    Unsigned char IccVCCOffDelay;   // # of etus delay before VCC should go off.
    Unsigned char  IccVCCTmr;       // # of etus to wait for VCC stable.
    unsigned char IccResetDelay,
    int IccInitialWaitingTime,
    Boolean IccATR_TimeoutEnabled,
    int IccATR_Timeout,
    int IccTS_Timeout
} ICC_Activate_t;
```

Where ICC_VOLTAGE is defined as:

VCC_0V = 0,
VCC_1V8 = 1,
VCC_3V = 2,
VCC_5V = 3

Parameters *IccVcc*: Input parameter
Voltage to apply when powering up this card.

IccVCCOffDelay: Input parameter
Number of ETUs to delay before shutting off the VCC. The default should be set to 3 ETUs. The maximum value is 15.

IccVCCTmr: Input parameter
Number of ETUs to wait for VCC to become stable. This time is calculated as: timer * 30.5 µs using a 32768 Hz clock. A value of 0 will result in no timeout (as opposed to zero time). The maximum value is 15.

IccResetDelay: Input parameter
Specifies the number of ETUs to keep RESET asserted after power has stabilized.

IccInitialWaitingTime: Input parameter
Specifies the maximum delay in ETUs between the leading edges of two consecutive characters of the ATR response.

IccATR_TimeoutEnabled: Input parameter
Specifies if the ATR timeout is enabled (TRUE) or disabled (FALSE).

IccATR_Timeout: Input parameter
Specifies the maximum delay in ETUs between the leading edge of the first character and last character of the ATR response, if enabled.

IccTS_Timeout: Input parameter
Specifies the maximum delay in ETUs between the de-assertion of the RST signal and the leading edge of the TS byte of the ATR.

pIccATR: Input parameter.
Pointer to the ICC_Rx_t structure.

Return Codes

ICC_OK
The SmartCard is present and active.

ICC_ACTIVATION_INCOMPLETE
The SmartCard is present, but its reset is still asserted.

ICC_ERR_OVERCURRENT or ICC_ERR_VCC_UNSTABLE
An attempt to activate the SmartCard caused an over current condition and it has been deactivated.

ICC_ERR_REMOVED
The SmartCard is not present.

ICC_ERR_TIMEOUT
One of the maximum delays was exceeded forcing a timeout. Take appropriate action.

ICC_ERR_BREAK
Data was always received with parity error, necessitating break signaling of the Smart Card (T=0 protocol).

ICC_ERR_PARITY
A byte was received with an invalid parity.

ICC_RX_PENDING
Reception has started, but is not yet completed. This code is returned on either a successful completion or a termination due to error.

ICC_RX_OVERRUN
An Rx overrun condition has occurred, resulting in the loss of at least one byte.

If the Smart Card was **powered-down** on entry, this function will perform a **cold-reset**.
If the Smart Card was **powered-up** on entry, this function will perform a **warm-reset**.
This function will return after reception of the TS byte or an error condition.

The **ICC_t** structure should be set up to expect the largest possible ATR response. Call ICC_RxLen() to determine the current total number of bytes received. Call ICC_RxDone() to complete ATR reception. (Refer to ICC_Tx() or ICC_Rx() for the **ICC_t** structure definition)

ICC_Status()

Purpose Retrieve the presence and active status of all the Smart Cards.

Synopsis **void ICC_Status (** OUT int *pnIccPresent, OUT int *pnIccActive **)**;

Parameters *pnIccPresent*: Output parameter
Specifies which SmartCards are present, Bit[n] corresponding to ICC[n], Bit[1] being mapped to the least significant bit.

PnIccActive: Output parameter
Specifies which SmartCards are active, Bit[n] corresponding to ICC[n].

Return Codes None.

ICC_Tx()

Purpose

Send data to the selected Smart Card. Before calling this function, the Smart Card UART must have been initialized and the selected Smart Card activated.

Synopsis

```
ICC_RC ICC_Tx ( INOUT struct ICC_t *pICC );
struct {
    IN char *IccData,
    INOUT unsigned int IccLen,
    IN Boolean IccLastByte,
    INOUT int IccEDC,
    OUT ICC_RC ICC_Status,
    OUT Boolean IccDone
} ICC_t;
```

Parameters

IccData: Input parameter
Contains the data to be transmitted.

IccLen: Input / output parameter
On input, specifies the number of bytes to send. On output, specifies the number of bytes successfully sent without errors, valid after *IccDone* is true.

IccLastByte: Input parameter
Specifies if the last transmitted byte is included in this buffer.

IccEDC: Input / output parameter
Contains the current LRC or CRC value (T=1).

ICC_Status: Output
Contains the current status of this transmission.

IccDone: Output
Set on completion of transmission, possibly with errors. Check *ICC_Status* for status.

Return Codes

ICC_ERR_PRESENT_INACTIVE
The SmartCard is present but inactive.

ICC_ERR_NO_CARD
The Smart Card is not present.

ICC_TX_PENDING
Transmission has started, but is not yet completed. On either a successful completion or a termination due to error, the ICC_Status will change to one of the following:

ICC_OK
Successful operation: All of the data was successfully transmitted to the SmartCard without parity error.

ICC_BREAK
Successful operation. All of the data was successfully sent to the SmartCard with at most a few retries. Initially, break signaling was detected, which necessitated at least one retransmission. (Only in T=0 protocol)

ICC_ERR_BREAK
All attempts at sending the next byte resulted in break signaling, indicating a perceived parity error at the SmartCard end. (Only in T=0 protocol) The SmartCard has been automatically deactivated. Check *ICCTxLen* to determine how many bytes were successfully transmitted.

IccLastByte should be set when it is time to switch to reception mode and start the BGT and BWT timers and possibly send the EDC.

The ICC_OK status **will not occur** until after the CRC/LRC has been sent to the SmartCard.

ICC_Rx()

Purpose

Receive data from the SmartCard interface. Before calling this function, the SmartCard UART must have been initialized and the selected SmartCard activated.

Synopsis

```
ICC_RC ICC_Rx ( INOUT struct ICC_t *pICC );
struct {
    OUT char *IccData,
    INOUT unsigned int IccLen,
    IN Boolean IccLastByte,
    INOUT Int IccEDC,
    OUT ICC_RC ICC_Status,
    OUT Boolean IccDone
} ICC_t;
```

Parameters

IccData: Output parameter
Contains the data currently received from the SmartCard.

IccLen: Input / output parameter
On input this is the number of requested bytes. On output this is the number of successfully received bytes, valid after bIccDone is true.

IccLastByte: Input parameter
Specifies if the last byte has been received after getting these *IccLen* bytes.

IccEDC: Input / output parameter
Contains the current LRC or CRC value (T=1).

Icc_Status: Output
Contains the current status of this reception.

IccDone: Output
Set on completion of reception, possibly with errors. Check *Icc_Status* for status.

Return Codes

ICC_ERR_PRESENT_INACTIVE
The SmartCard is present, but inactive.

ICC_ERR_NO_CARD
The SmartCard is not present.

ICC_RX_PENDING
Reception has started, but is not yet completed. On either a successful completion or a termination due to error, the ICC_Status will change to one of the following:

ICC_OK
Successful operation, *IccLen* bytes have been received from the SmartCard.

ICC_BREAK
Successful operation, *IccLen* bytes have been received from the SmartCard, but some bytes were initially received with parity error, necessitating some break signaling of the SmartCard, forcing it to retransmit at least once (T=0).

ICC_ERR_BREAK
Data was always received with parity error, necessitating break signaling of the SmartCard (T=0). The SmartCard has been automatically deactivated. Check ICC_RxLen() to determine how many bytes were successfully received.

ICC_ERR_TIMEOUT
A byte was not received before the maximum delay specified by nIccWorkWaitingTime (T=0) or nIccCharWaitingTime (T=1) expired.

ICC_ERR_PARITY
A byte was received with an invalid parity.

ICC_ERR_OVERRUN
An RX overrun condition has occurred, resulting in the loss of at least one byte.

IccLastByte should be set when it is time to switch to transmission mode and start the BGT timer. If it is not immediately known when it is time to switch, call ICC_RxDone() after all the bytes have been received.

To determine if all the expected bytes have been received, call ICC_RxLen(). Since the ICC_OK or ICC_BREAK status occurs after reception of *ICCLen* bytes, this value should include any CRC/LRC byte(s) received.

ICC_RxLen()

Purpose Return the number of bytes received thus far.

Synopsis **unsigned Int ICC_RxLen (** void **)**;

Parameters None.

Return Codes None.

ICC_RxDone()

Purpose Notify the SmartCard UART that all the expected bytes have been received. This forces the switch to activation of block guard and wait times.

Synopsis **Void ICC_RxDone (** void **)**;

Parameters None.

Return Codes None.

ICC_Deactivate()

Purpose Deactivate the contacts of the selected SmartCard interface as specified by IccId. The Vcc, RST, I/O and Clk signals are configured according to the ISO 7816-3 standard. This function uses the *IccVCCOffDelay*, as described in ICC_Activate() before turning VCC off.

Synopsis **void ICC_Deactivate (** void **)**;

Parameters None.

Return Codes None.

ICC_Mode()

Purpose Enable or Disable PC (via DIRECT mode) or Application (via Bypass mode) to directly control ICC I/O line.

Synopsis
```
ICC_RC ICC_Mode (
    IN enum ICC_ID Iccid,
    IN Boolean bEnable,
    IN Boolean bPC_DIRECT )
```

Parameters *IccId* : Input parameter
Specifies which SmartCard to allow direct access to I/O. Possible values are:

ICC_1ST	0, (Internal)
ICC_2ND	1, (External)
…	
ICC_9TH	8 (External).

bEnable: Input parameter
If TRUE, it enables the selected mode.

bPC_DIRECT: Input parameter
Enables or disables PC_DIRECT mode (if TRUE) or BYPASS mode (if FALSE).

Return Codes

ICC_OK	The SmartCard is present and active.
ICC_ERR_PRESENT_INACTIVE	The SmartCard is present but inactive.
ICC_ERR_NO_CARD	The SmartCard is not present.

This API is part of the support for Synchronous cards.

ICC_Clk_Restart()

Purpose Restarts an ICC's clock.

Synopsis **ICC_Clk_Restart (** IN int nIccDelayIO **)**;

Parameters *nIccDelayIO*: Input parameter
Delay in clock cycles after restart of the clock before allowing I/O.

Return Codes

ICC_OK	Successful operation. *IccLen* bytes have been received from the SmartCard.
ICC_ERR_PRESENT_INACTIVE	The SmartCard is present but inactive.
ICC_ERR_REMOVED	The SmartCard is removed.

The hardware TIMER1 is used by this routine, making it unavailable to the application. This approach helps support lower power consumption.

ICC_Clk_Stop()

Purpose Stops an ICC's clock.

Synopsis **ICC_RC ICC_Clk_Stop (** IN Boolean bIccClkStop, IN int nIccDelayStop **)**;

Parameters *bIccClkStop*: Input parameter
Specifies whether to stop the IccClk when HIGH (TRUE) or when LOW (FALSE).
nIccDelayStop: Input parameter
Delay in clock cycles before stopping clock.

Return Codes

ICC_OK	Successful operation.
ICC_ERR_PRESENT	The SmartCard is present but inactive.
ICC_ERR_REMOVED	The SmartCard is removed

The hardware TIMER1 is used by this routine, making it unavailable to the application. This approach helps support lower power consumption.

4.2.6 SERIAL (RS232) Driver API – Available with all 73S12xxF Devices

The Serial Driver API manages the RS232 interface (Serial Channel 0). It may be used to communicate through the UART with any host that supports an RS232 interface. The API includes:

After calling Serial_Init() and prior to receiving data from the RS232 interface, Serial_Rx() and Serial_Tx() must be called to pass on the receive/transmit buffer pointer, which is used to store Rx and Tx characters, respectively. For a sample of the Serial API usage, see the Pseudo-CCID application source code.

Serial_Init()

Purpose — Configure the communication speed, flow control, character parity and number of stop bits. The serial interrupt service routine is NOT maskable, the interrupt vector is set internally, so using Set_Event (eSerial, ...) will not have any effect. The baud rates listed below are available for specific CPU clock speeds only. Review the API_Struct_12.h for more information.

Synopsis

Bbool Serial_Init (
IN enum SERIAL_SPD speed,
IN Boolean parity_en,
IN Boolean parity,
IN Boolean two_stop_bits,
IN Boolean xon_xoff **)**

Parameters

Speed: Input parameter
This selects the communication speed. Possible values are:

_RATE_600,	0
_RATE_1200,	1
_RATE_2400,	2
_RATE_4800,	3
_RATE_9600,	4
_RATE_14400,	5
_RATE_19200,	6
_RATE_28800,	7
_RATE_38400,	8
_RATE_57600,	9
_RATE_115200,	10
_RATE_125000,	11
_RATE_250000,	12
_RATE_375000	13

Parity_enb: Input parameter
Specifies if the exchanged characters contain a parity bit (TRUE) or not (FALSE).

parity: Input parameter
Specifies if the characters parity bits must be odd (TRUE) or even (FALSE).

Two_stop_bits: Input parameter
Specifies if the exchanged characters contain two stop bits (TRUE) or one (FALSE).

xon_xoff: Input parameter
Specifies if Xon_Xoff control is on (TRUE) or off (FALSE).

Return Codes — None.

The lower speeds help support Plug & Play.

Serial_Tx()

Purpose — Setup the Tx buffer before sending data to the PC UART. An application should call this API immediately after calling Serial_Init().

Synopsis — **enum SERIAL_RC data *Serial_Tx (** U08x xdata *buffer, U16 len **)**

Parameters — *buffer*: Input parameter
Specifies a pointer to the data buffer containing data to send to the PC UART.

Len: Input parameter.
Specifies the current number of bytes to be sent.

Return Codes S_EMPTY Successful transmission.
S_PENDING, Successful transmission thus far but not yet finished.
Where return code **SERIAL_RC** is defined as**:** Enum SERIAL_RC.

Serial_CTx()

Purpose Put bytes into the transmit buffer and start sending. Prior to calling this function, Serial_Tx() must be call to setup the Tx buffer.

Synopsis **Unsigned Integer Serial_CTx (** U08x xdata *buffer, U16 len **)**

Parameters *buffer*: Input parameter
Specifies a pointer to the data buffer containing the data to send to the PC UART.
len: Input parameter.
Specifies the current number of bytes to be sent.

Return Value Unsigned integer specifying the number of bytes sent thus far.

After calling this API, an application can make sure all bytes were transmitted by checking that Serial_TxLen() returns a 0.

Serial_TxLen()

Purpose Number of bytes transmitted thus far.

Synopsis **Unsigned integer Serial_ TxLen (** void **)**

Parameters none.

Return Value Unsigned integer specifying the number of bytes left in the Tx buffer, i.e. the remaining bytes to be sent.

Serial_TxByte ()

Purpose Send a quick byte out.

Synopsis **void Serial_TxByte (** U08 cbyte **)**

Parameters ***c****byte*: Input parameter
Byte to put at the end of the Tx buffer to be sent out quickly.

Return Codes none.

This function performs similarly to Serial_CTx() (U08 &cbyte, 1) but it has much less overhead. Use this API when performance optimization is required yet only one byte can be sent at a time.

Serial_Rx()

Purpose Setup receive buffer and start receiving. Always call this function after Serial_Init() to make sure the receive buffer is available.

Synopsis **enum SERIAL_RC data *Serial_Rx (** U08x xdata *buffer, U16 len **)**

Parameters *buffer*: Input parameter
Specifies a pointer to the data buffer to store the data received from the PC UART.

len: Input parameter.
Specifies the maximum number of bytes to receive at any one time.

Return Codes On a successful completion or termination, the serial Status will return one of the following:

S_EMPTY	Reception has started but the receive buffer is still empty.
S_PENDING	Reception has started, but is not yet completed.
S_FULL	Reception has started and the buffer is now full.
S_PARITY_ERR	Parity error occurred on the received byte(s).
S_OVERRUN	Buffer overrun, which may result in a loss of at least 1 byte.

Serial_CRx()

Purpose Get additional bytes from the receiving buffer.

Synopsis **Unsigned Integer Serial_CRx (** U08x xdata *buffer, U16 len **)**

Parameters *buffer*: Input parameter
Specifies a pointer to the data buffer to store the data received from the PC UART.
len: Input parameter.
Specifies the maximum number of bytes to receive at any one time.

Return Value Upon completion, returns the number of bytes received thus far.

Serial_RxLen()

Purpose Number of bytes received thus far.

Synopsis **Unsigned Integer Serial_ RxLen (** void **)**

Parameters None.

Return Value Unsigned integer specifying the number of bytes received thus far.

Serial_RxByte ()

Purpose Get a quick byte out of the input buffer.

Synopsis **Unsigned Char Serial_RxByte (** void **)**

Parameters None.

Return Value Byte received from the serial interface.

This function performs similarly as Serial_CRx (U08 &cbyte, 1) but has much less overhead. Use this API when performance optimization is required yet only one byte can be read at a time.

4.2.7 USB API – Available with 64K Flash version of the 73S12xxF

This API manages the USB interface which is compatible with the USB Specifications 2.0 – Full Speed/12Mbps. The USB protocol Suspend, Resume and Reset operations are managed by this API. The API includes:

The USB interface contains four endpoints, which are defined as follows:

1. Endpoint 0 for the control transfer
2. Endpoint 1 IN for the Bulk transfer
3. Endpoint 1 OUT for the Bulk transfer
4. Endpoint 2 IN for the interrupt transfer

The Low-level API handles all Endpoint 0 (control endpoint) communications; thus PC driver enumeration takes place during the device reset time and is transparent to an application. An application can be written to monitor this reset signal to determine when it can start sending Endpoint 1 IN and/or Endpoint 2 IN packets to the host or to expect Endpoint 1 OUT packets from the host. (References to transmission direction (IN/OUT) are relative to the host.)

USB_Init()

Purpose — Configure the USB interface during the reset procedure. The descriptors (device, configuration, endpoint, interface and string) used in future enumeration requests are configured. When leaving this function, the USB interface is ready to answer any enumeration request.

Synopsis

```
Void USB_Init (
    IN struct USB_Init_t *pUSB,
    IN struct USB_LangID_t *pLangID,
    IN void (*pRESET) (),
    IN void (*pSUSPEND) (),
    IN void (*pRESUME) () );

struct USB_Init_t
{
    struct USB_Device_t    Device;
    struct USB_Config_t    Config;
    struct USB_CCID_t      CCID;
    struct USB_Strings_t   Strings;
};
```

The Device, Config, CCID, Strings and LangID structures are described at the end of this function description.

Parameters

pUSB: Input parameter
Specifies the Device, Configuration, Endpoints, Interfaces and String Descriptors defining the USB interface.

pLangID: Input parameter
Specifies the Language ID codes String Descriptor.

pRESET: Input parameter
Specifies a pointer to the function to call when RESET signaling has occurred on the USB bus.

pSUSPEND: Input parameter
Specifies a pointer to the function to call when SUSPEND signaling has occurred on the USB bus.

pRESUME: Input parameter
Specifies a pointer to the function to call when RESUME signaling has occurred on the USB bus.

Return Codes — None.

This function will activate the USB interface and ALL Endpoints will be ACTIVE. Endpoint 0 gets device descriptor requests and will return the device descriptor. Endpoint 0 gets configuration descriptor

requests and will return the configuration, endpoint, interface and string descriptors including the CCID class descriptor, depending on the maximum length requested.

There are two possible configurations for the USB: self-powered and bus-powered. Each configuration requires some power consumption management to effectively reduce the power according to the *USB 2.0 Specification*. As a result, the following must be implemented for each configuration:

- For Self-Power: Cable attach/detach must be detectable to turn D+ on/off respectively. The CCID USB sample code included as part of the release has implemented the use of USR7 with external interrupt 0 (all internal to IC) so that a cable attach/detach event will be detected and serviced via the INT0 interrupt service routine.
- For Bus-Power: D+ must be kept high at all times via the PowerON(ENABLE_USB) API. When the host puts the device in Suspend mode, the CCID USB sample code puts the device's CPU to sleep to conserve power. After Suspend mode, the host wakes up the device via a Reset or a Resume signal.

  This signal (D+ being pulled low for a period of time as described in the *USB 2.0 Specification*) will cause interrupt 0 to occur which will wake up the device's CPU. The interrupt is required because the USB clock is turned off during sleep mode, so the D+ (Reset/Resume) signal would not wake up the CPU. Upon waking up, the INT0 interrupt service routine will turn the USB clock back on to resume its function.

In order to configure the self-powered or bus-powered mode, modify the ATTRIBUTES variable, defined in API_12.h, to its respective value before building the application. See the CCID USB source code project for an example.

The USB initialization descriptors and relevant structures are described below. All the descriptors are modifiable, but some values should not change. These are flagged as "Always".

```
//
//  USB API.
//
/*** the total size of the configuration descriptor ***/
#define CONFIG_DESC_TOTAL_SIZE   93
/*** define the number of core endpoints (including EP0) ***/
#define NUMBER_OF_EPS                4
/*** define the total possible number of interfaces for all configurations ***/
#define NUMBER_OF_INTERFACES         1
/*******************************************************
Descriptor types
*******************************************************/
#define DEVICE_DESCRIPTOR            1
#define CONFIGURATION_DESCRIPTOR     2
#define STRING_DESCRIPTOR            3
#define INTERFACE_DESCRIPTOR                 4
#define EP_DESCRIPTOR                5
#define CCID_DESCRIPTOR              0x21
struct USB_Device_t
{
unsigned char   Length;              // Always 18.
unsigned char   DescriptorType;      // Always DEVICE_DESCRIPTOR = 1.
unsigned int    USB_spec_rev;        // 0x0002 (rev 2.0).
unsigned char   DeviceClass;         // Always  0.
unsigned char   DeviceSubclass;      // Always  0.
unsigned char   DeviceProtocol;      // Always  0.
unsigned char   MaxPacketSize0;      // Always  16.
unsigned int    idVendor;            //Always 0xc309
```

```
unsigned int     idProduct;              // Always 0x0500
unsigned int     Device;                 // 0x4600;
unsigned char  iManufacturer;            //  0, TBD;
unsigned char  iProduct;                        //  0, TBD;
unsigned char   iSerialNum;              //  0, TBD;
unsigned char   NumConfigs;              //  1, TBD;
};

struct USB_Interface_t
{
unsigned char   Length;                  // Always 9.
unsigned char   DescriptorType;          // Always INTERFACE_DESCRIPTOR = 4.
unsigned char   InterfaceNumber;         // 0.
unsigned char   AlternateSetting;        // 0.
unsigned char   NumEndPoints;            // Always NUMBER_OF_EPS - 1 = 3;
unsigned char   InterfaceClass;          // 0x0B.
unsigned char   InterfaceSubClass;       // 0.
unsigned char   InterfaceProtocol;       // 0.
unsigned char   iInterface;              // 0.
};

#define CLASS 0x0B                       //Always 0B for Smart Card Reader
#define SUBCLASS 0

struct USB_EP_t
{
unsigned char   Length;                  // Always 7.
unsigned char   DescriptorType;          // Always EP_DESCRIPTOR = 5.
unsigned char   EndpointAddress;         // (IN or OUT) plus (1 or 2).
unsigned char   Attributes;              // BULK or INTERRUPT.
unsigned int     MaxPacketSize;          // 16, 32 or 64.
unsigned char   Interval;                // 0 for BULK.  10 for INTERRUPT.
};

#define IN                0x80
#define OUT               0x00
#define BULK              0x02
#define INTERRUPT         0x03
#define INTERVAL          10             // Interrupt EndPoint interval in frames (about 10 msec).

struct USB_Config_t
{
unsigned char   Length;                  // Always  9.
unsigned char   DescriptorType;          // Always CONFIGURATION_DESCRIPTOR = 2.
unsigned char   TotalLengthL;            // CONFIG_DESC_TOTAL_SIZE = 93
                                         // (One Interface, three Endpoints).
unsigned char   TotalLengthH;            // Always 0.
unsigned char   NumInterfaces;           // NUMBER_OF_INTERFACES = 1.
unsigned char   ConfigurationValue;      // Always 1.
unsigned char    iConfiguration;         // Application specific.
unsigned char   Attributes;              // Application specific.
unsigned char   MaxPower;                // Application specific.
struct USB_Interface_t  I;
};

#define ATTRIBUTES    BIT7 | SELF_POWERED        // application specific, self-powered
```

```
#define SELF_POWERED    BIT6
#define REMOTE_WAKEUP    BIT5
#define MAXPOWER  50                                    // 100 mA, if bus-powered.
#define NUMBER_LANGIDS 1                                // Change as needed.

#ifdef DFU
#define NUMBER_STRINGS 4                                // Change as needed.
#else
#define NUMBER_STRINGS 3

struct USB_LangID_t
{
Uc      Length;                         // (NUMBER_LANGIDS * 2) + 2.
Uc      DescriptorType;                 // Always STRING_DESCRIPTOR = 3.
Ui      LangID [ NUMBER_LANGIDS ];          // Array of LangID codes.
};

#define MAX_STRING_LEN 80           // Change and duplicate as needed.

struct USB_String_t
{
Uc      Length;                         // Real STRING_LEN + 2.
Uc      DescriptorType;                 // Always STRING_DESCRIPTOR = 3.
Uc      String[ MAX_STRING_LEN ];   // UNICODE encoded string.
};
//Pointers to strings of a language are grouped together.

struct USB_Strings_t
{
U08     Number_Of_Strings;      // Number of String descriptors per language.
                                // Array of pointers to UNICODE encoded STRING descriptors.
struct USB_String_t code *Strings[ NUMBER_STRINGS * NUMBER_LANGIDS ];
};

// Define for any selection parameter below.
#define NONE   0x00000000L

// Defines for Voltage Support.
#define VOLTS5_0        0x01
#define VOLTS3_0        0x02
#define VOLTS1_8        0x04

// Defines for Protocols supported.
#define PROTOCOL_T_0        0x00000001L
#define PROTOCOL_T_1        0x00000002L

// Defines for Clock rates.
#define KHZ3600         3600L
#define KHZ4000         4000L
#define KHZ5050         5050L
#define KHZ6000         6000L
#define KHZ8000         8000L
#define KHZ9600         9600L
#define KHZ12000        12000L

// Defines for ICC bps.
```

```
#define BPS9600         9600L    // 0x00002580  (9600 +- 1%)
#define BPS14400        14400L
#define BPS19200        19200L
#define BPS28800         28800L
#define BPS38400        38400L
#define BPS57600        57600L
#define BPS57688        57688L
#define BPS115200       115200L // 0x0001C200 (115200 +- 1%)
#define BPS116129       116129L // 0x0001C200
#define BPS225000       225000L
#define BPS230400       230400L

// Defines for Mechanical.
#define ACCEPT          0x00000001L
#define EJECT           0x00000002L
#define CAPTURE         0x00000004L
#define LOCK            0x00000008L

// Defines for Features.
#define ATR_CONFIGURATION   0x00000002L
#define INSERT_ACTIVATION   0x00000004L
#define VOLTAGE_SELECTION   0x00000008L
#define CLOCK_FREQ_CHANGE       0x00000010L
#define BIT_RATE_CHANGE     0x00000020L
#define AUTO_NEGOTIATION    0x00000040L
#define AUTO_PPS            0x00000080L
#define CLOCK_STOP          0x00000100L
#define NAD_NOT_NULL        0x00000200L
#define AUTO_IFSD           0x00000400L
#define TPDU                0x00010000L
#define SHORT_APDU          0x00020000L
#define EXTENDED_APDU       0x00040000L

// Defines for LcdLayout
#define NO_LCD              0x0000
#define LCD_ROWS            0xFF00
#define LCD_COLS            0x00FF

// Defines for PinSupport.
#define NO_PIN              0x00
#define VERIFY              0x01
#define MODIFY              0x02

struct USB_CCID_t
{
unsigned char   Length;             // Always 54 (0x36).
unsigned char   DescriptorType;     // Always CCID_DESCRIPTOR = 0x21
unsigned int    CCID;               // 0x0100 CCID version number.
unsigned char   MaxSlotIndex;       // Application specific.
unsigned char   VoltageSupport;     // Application specific.
unsigned long   Protocols;          // Application specific.
unsigned long   DefaultClock;       // 3600kHz (0x00000E10).
unsigned long   MaximumClock;       // 7200kHz (0x00001C20)
unsigned char   NumClockSupported;  // 3. (1800, 3600 and 7200 kHz).
unsigned long   DataRate;           //   9600 bps (0x00002580).
unsigned long   MaxDataRate;        // 115200 bps (0x0001C200).
```

```
unsigned char   NumDataRates;       //  7.
unsigned long   MaxIFSD;            // Application specific.
unsigned long   SynchProtocols;     // 0x00000000.
unsigned long   Mechanical;         // Application specific.
unsigned long   Features;           // Application specific.
unsigned long   MaxCCIDMsgLen;      // Application specific.
unsigned char   ClassGetResponse;   // Application specific.
unsigned char   ClassEnvelope;      // Application specific.
unsigned int    LcdLayout;          // Application specific.
unsigned char   PinSupport;         // Application specific.
unsigned char   MaxCCIDBusySlots;   // Application specific.
struct USB_EP_t EP[ NUMBER_OF_EPS-1 ];
#ifdef DFU
struct USB_Interface_t DFU_I;   //One for DFU.
struct USB_DFUFunctional_t DFUFunctional;
#endif
};

struct USB_t
{
U08x *UsbData;
U16  UsbLen;
enum USB_RC UsbStatus;
};
```

USB_Status()

Purpose Gets the status of the USB interface and its endpoints.

Synopsis **Void USB_Status (**
OUT char *cUSB_CONTROL_Status,
OUT char *cUSB_INTERRUPT_Status,
OUT char *cUSB_BULK_IN_Status
OUT char *cUSB_BULK_OUT_Status **)**;

Parameters *cUSB_CONTROL_Status*: Output parameter
Current state of the Control EndPoint. Possible values are:

USB_ACTIVE	0
USB_SUSPENDED	1
USB_RESUMED	2
USB_STALLED	3
USB_RESET	4
USB_TX_PENDING	5
USB_RX_PENDING	6

cUSB_INTERRUPT_Status: Output parameter
Current state of the INTERRUPT EndPoint.

cUSB_BULK_IN_Status: Output parameter
Current state of the BULK_IN EndPoint.

cUSB_BULK_OUT_Status: Output parameter
Current state of the BULK_OUT EndPoint.

Return Codes None.

USB_ACTIVE indicates that either the Interrupt and Bulk_IN EndPoints are ready for another USB transmission (the previous one has finished) or that the Bulk_OUT EndPoint is ready for another USB reception**.**

USB_Stall()

Purpose Stalls portions of the USB interface: The Endpoints to be stalled are configurable.

Synopsis **Void USB_Stall (** IN char cUSBEndpointStall **)**;

Parameters *cUSBEndpointStall*: Input parameter
Specifies which endpoints are to be stalled by this function. The other endpoints remain in their previous state. This parameter can be the result of an OR operation between the following values if several endpoints are to be disabled:

ENDPOINT_0 0x01
ENDPOINT_1_IN 0x02
ENDPOINT_2_IN 0x04
ENDPOINT_1_OUT 0x08

Return Codes None.

Deactivated endpoints will have a STALLED status. If all endpoints are stalled, it will disconnect the USB interface from the Host.

USB_UnStall()

Purpose Unstalls portions of the USB interface: The Endpoints to be unstalled are configurable.

Synopsis **Void USB_UnStall (** IN char cUSBEndpointUnStall **)**;

Parameters *cUSBEndpointUnStall*: Input parameter
Specifies which endpoints are to be unstalled by this function. The other endpoints remain in their previous state. This parameter can be the result of an OR operation between the following values if several endpoints are to be disabled:

ENDPOINT_0 0x01
ENDPOINT_1_IN 0x02
ENDPOINT_2_IN 0x04
ENDPOINT_1_OUT 0x08

Return Codes None.

All unstalled endpoints will have status = IDLE.

USB_IN_1()

Purpose Send data to the Host through Endpoint 1 (BULK IN). When the buffer size is bigger than the Maximum Packet Size (specified by the Descriptor string initialized in the USB_Init() function), then the API will split the buffer into smaller blocks and transmit it in pieces.

Synopsis **Void USB_IN_1 (** IN struct USB_t *pUSB **)**;

```
struct
{
    unsigned char   *USBData,
    unsigned int    USBLen
    USB_RC          USBStatus
} USB_t;
```

Parameters *USBData*: Input parameter
Specifies the pointer to the data to be transmitted to the Host.

USBLen: Input/Output parameter
On input, specifies the number of bytes to send to the Host. On output, specifies the current number of bytes sent to the Host.

USBStatus: Output parameter
Contains the current status of this transmission, one of the following:
USB_TX_PENDING: Transmission has started, but is not yet complete. On either a successful completion or a termination due to error, the *USBStatus* will change to one of the following:
USB_ACTIVE: Successful data transmission.
USB_SUSPENDED: The HOST has suspended the USB bus, retry later.
USB_STALLED: This Endpoint has been STALLED.
USB_RESET: The HOST has reset the USB bus, retry later.

Return Codes None.

USB_IN_2()

Purpose Send data to the Host through Endpoint 2 (INTERRUPT IN).

Synopsis **Void USB_IN_2 (** IN struct USB_t *pUSB **)**;

Parameters *USBData*: Input parameter
Specifies the pointer to the data to be transmitted to the Host.
USBLen: Input/Output parameter
On input, specifies the number of bytes to send to the Host. On output, specifies the current number of bytes sent to the Host.
USBStatus: Output parameter
Contains the current status of this transmission, one of the following:
USB_TX_PENDING: Transmission has started, but is not yet completed. On either a successful completion or a termination due to error, the *USBStatus* will change to one of the following:
USB_ACTIVE: Successful data transmission.
USB_SUSPENDED: The HOST has suspended the USB bus, retry later.
USB_STALLED: This Endpoint has been STALLED.
USB_RESET: The HOST has reset the USB bus, retry later.

Return Codes None.

USB_OUT_1()

Purpose Receive a buffer from the Host through Endpoint 1 (BULK OUT). The data may be received within several packets if the buffer size is greater or equal to the Maximum Packet Size, specified by the Descriptor String initialized in USB_Init().

Synopsis **Void USB_OUT_2 (** IN struct USB_t *pUSB **)**;

Parameters *USBData*: Output parameter
Contains the bytes received from the Host.
USBLen: Input/Output parameter
On input, specifies the maximum number of bytes to receive from the Host. On output, specifies the current number of bytes received from the Host.
USBStatus: Output parameter
Contains the current status of this reception, one of the following:
USB_RX_PENDING: Reception has started, but is not yet completed. On either a successful completion or a termination due to error, the *USBStatus* will change to one of the following:
USB_ACTIVE: Successful data reception.
USB_SUSPENDED: The HOST has suspended the USB bus, retry later.
USB_STALLED: This Endpoint has been STALLED.
USB_RESET: The HOST has reset the USB bus, retry later.
USB_ERR_OVERFLOW: The HOST has sent too much data.

Return Codes None.

4.2.8 Clock Generator Circuit API – Available with all 73S12xxF Devices

The Clock Generator API configures the system clock speed.

CPU_Select()

Purpose Select the CPU speed. This API should be called after API_Init() is called if a change in CPU speed is desired. The default speed is 3.69 MHz. Care should be taken when selecting speeds other than 3.69 MHz, as the speed will affect the number of supported Serial baud rates and timers API. The timer and Serial baud rates will be adjusted internally by the LAPI. A wait state will be inserted to allow USB I/O access time, i.e. CKCON will be set to the appropriate value to adjust wait states.

Synopsis **void CPU_Select (** enum CPU_SPEED speed **)**

Parameters *speed*: Input parameter
Specifies which CPU speed to run. Possible values are
CPU_3Mhz69 0,
CPU_6Mhz 1,
CPU_12Mhz 2,
CPU_24Mhz 3

Return Codes None.

Some CPU speeds will limit the number of Serial baud rates supported. Table 4 lists the baud rates supported by specific CPU clock rates.

Table 4: Clock Speeds and Baud Rates Supported

Baud Rate (bps)	CPU Clock Rate			
	3.69 MHz	6 MHz	12 MHz	24 MHz
600	X			
1200	X	X		
2400	X	X	X	
4800	X	X	X	
9600	X		X	X
14400	X	X	X	X
19200	X			X
28800	X		X	X
38400	X			
57600	X			X
115200	X			
115385	X			
187500		X		
375000			X	
750000				X

//Can only support these rates when the CPU is running at 3.69 MHz
//CPU_3Mhz69
{BPS_600_3MHz69, BPS_1200_3MHz69, BPS_2400_3MHz69, BPS_4800_3MHz69,
BPS_9600_3MHz69, BPS_14400_3MHz69, BPS_19200_3MHz69, BPS_28800_3MHz69,

```
BPS_38400_3MHz69,BPS_57600_3MHz69, BPS_115200_3MHz69, 0, 0, 0  },

//Can only support these rates when the CPU is running at 6 MHz
//CPU_6MHz
{0,      BPS_1200_6MHz,     BPS_2400_6MHz,     BPS_4800_6MHz,
 0,      BPS_14400_6MHz,    0,                 0,
 0,      0,                 0,                 0 ,      0,      0},

//Can only support these rates when the CPU is running at 12 MHz
//CPU_12MHz
{0,      0,                 BPS_2400_12MHz,    0,                 BPS_9600_12MHz,
BPS_14400_12MHz,  0,        BPS_28800_12MHz,   0,
0,       0,                 BPS_125000_12MHz,  0,                 BPS_375000_12MHz },

//Can only support these rates when the CPU is running at 24 MHz
//CPU_24MHz
{0,      0,                 0,                 BPS_4800_24MHz,    BPS_9600_24MHz,
BPS_14400_24MHz,  BPS_19200_24MHz,  BPS_28800_24MHz,   0,
BPS_57600_24MHz,  0,                BPS_125000_24MHz,  BPS_250000_24MHz,
BPS_375000_24MHz}
```

4.2.9 Power Management API – Available with all 73S12xxF Devices

The Power Management API configures the active devices, thereby managing the power consumption of the 73S12xxF. The API includes:

- PowerON() (page 52)
- PowerOFF() (page 53)

PowerON()

Purpose Manage power consumption.

Synopsis **Void PowerON (** IN unsigned int PowerSelect **)**;

Parameters *PowerSelect*: Input parameter
Specifies which internal devices to enable. The following possible values (defined in API_12.h) or any combination (by OR'ing them) are allowed:

```
ENABLE_EICC       BIT14   // External Smart card.
ENABLE_VDDF       BIT13   // VDD fault detection
ENABLE_UART       BIT12    // Serial
ENABLE_PLL        BIT11
ENABLE_ANALOG     BIT10
ENABLE_USBXCVR    BIT9    //USB Transceiver, Unavailable with 73S1205F
ENABLE_USB        BIT8    //D+, Unavailable with 73S1205F
ENABLE_RTC        BIT7    // Unavailable with 73S1205F
ENABLE_KEYPAD     BIT6
ENABLE_ICC        BIT5    // Smart card.
ENABLE_USBCLK     BIT4    //USB Clock, Unavailable with 73S1205F
ENABLE_LS_OSC     BIT3    //Low speed OSC-32kHz, Unavailable with
                          73S1205F
```

Bits 2, 1 and 0 are reserved for MCount value. Devices which are already enabled will remain enabled.

Return Codes None.

PowerOFF()

Purpose Manage power consumption.

Synopsis **void PowerOFF (** IN unsigned int PowerSelect **)**;

Parameters *PowerSelect*: Input parameter
Specifies which internal devices to disable. The following possible values or any combination (by OR'ing them) are allowed:

DISABLE_EICC	BIT14	// External Smart card.
DISABLE_VDDF	BIT13	
DISABLE_UART	BIT12	
DISABLE_PLL	BIT11	
DISABLE_ANALOG	BIT10	
DISABLE_USBXCVR	BIT9	
DISABLE_USB	BIT8	
DISABLE_RTC	BIT7	
DISABLE_KEYPAD	BIT6	
DISABLE_ICC	BIT5	
DISABLE_ USBCLK	BIT4	
DISABLE_LS_OSC	BIT3	
// Bit 2 reserved.		
CPU_HALT	BIT1	
CPU_IDLE	BIT0	

Return Codes None.

Devices which are already disabled will remain disabled. Any enabled interrupt will cause an exit from this function.

CPU_HALT has precedence over CPU_IDLE. CPU_IDLE mode stops the clock going to the CPU, all other clocks keep running.

CPU_HALT mode can be stopped by the USB and RTC interrupts (if available), by a key press, by ICC insertion or removal or by Reset of the 73S12xxF. In order to use these external events to wake up the CPU, they must first be individually initialized. For example, ICC_InitUART() or KEY_Init () should be called prior to calling PowerOFF (DISABLE_ICC) or PowerOFF (DISABLE_KEYPAD). Internally, the API will configure INT0 to be active upon any of the events (key press, Smart Card event, etc..); thus an application may setup its own INT0 interrupt service routine via Set_Event (eEXT0, ...) to customize its specific needs upon waking up.

4.2.10 Analog Threshold Management Driver API – Available with all 73S12xxF Devices

This API controls the analog voltage comparison against the voltage on the ANA_IN pin. The API includes:

ANALOG_Detect_Enable()

Purpose Select the analog threshold level and enable the interrupt according to the polarity setting.

Synopsis **void ANALOG_Detect_Enable** (
IN Unsigned char threshold_select,
IN Unsigned char acomp_pol)

Parameters *threshold_select*: Input parameter
Specifies which input voltage channel must be compared against Vcompare. Allowable values are in the range [0, 7].

acomp_pol: Input parameter
Specifies the polarity for which an interrupt occurs; when voltage level is Above (acomp_pol = 1) or Below (acomp_pol = 0). Voltage levels are values in the range [0, 7].

7 corresponds to 2.50 volts //**Only available with the 73S1205F**
6 corresponds to 2.30 volts //**Only available with the 73S1205F**
5 corresponds to 2.00 volts //**Only available with the 73S1205F**
4 corresponds to 1.75 volts //**Only available with the 73S1205F**
3 corresponds to 1.50 volts
2 corresponds to 1.40 volts
1 corresponds to 1.24 volts
0 corresponds to 1.00 volts.

Return Codes None.

ANALOG_Detect_Disable()

Purpose Disable the Analog level detect interrupt.

Synopsis **Void ANALOG_Detect_Disable (** void **);**

Parameters None.

Return Codes None.

ANALOG _Compare()

Purpose Compare the selected input voltage against specified threshold.

Synopsis **enum ANALOG_RC ANALOG_Compare (**
IN unsigned char threshold_select,
IN unsigned char acomp_pol **);**

Parameters *threshold_select*: Input parameter
Specifies which input voltage to compare against ANA_IN. Values are in the range [0, 7] as specified below.

acomp_pol: Input parameter
Specifies which level to compare against (above or below). Values are in the range [0, 7].

7 corresponds to 2.50 volts
6 corresponds to 2.30 volts
5 corresponds to 2.00 volts
4 corresponds to 1.50 volts
3 corresponds to 1.40 volts
2 corresponds to 1.28 volts
1 corresponds to 1.24 volts
0 corresponds to 1.00 volts.

Return Codes ANALOG_OK
ANALOG_BELOW
ANALOG_ABOVE

4.2.11 Event Management API – Available with all 73S12xxF Devices

The Event Management API allows the application to handle all system events. An application should always call Events_Init() to initialize all event vectors at the beginning of the main program. The API includes:

Events_Init()

Purpose: Initialize the system default event vectors. Upon exiting this function, all vectors will point to a null_isr. For this reason, every feature (Smart card, USB, RTC, Keypad, LCD, etc.) must call its initialization routine so that its interrupt service routine will be set properly.

Synopsis: **void Events_Init (** void **)**;

Parameters: None.

Return Codes: None.

Events_Clear()

Purpose: Clear selected events.

Synopsis: **Void Events_Clear (** unsigned long Events **)**;

Parameters: *Events*: Input parameter
Specifies which events to clear. Multiple events are specified by OR'ing together individual events. Possible values are:

EVENT_EXT0	BIT0	
EVENT_EXT1	BIT1	
EVENT_EXT2	BIT2	
EVENT_EXT3	BIT3	
EVENT_TIMER0	BIT4	
EVENT_TIMER1	BIT5	
RFU	BIT6	
EVENT_RTC	BIT7	**//not available with the 73S1205F**
EVENT_KEY_DETECT	BIT8	
EVENT_USB	BIT9	**//not available with the 73S1205F**
EVENT_VDDF	BIT10	
EVENT_I2C	BIT11	
EVENT_ANALOG	BIT12	
EVENT_USR0	BIT13	
EVENT_USR1	BIT14	
EVENT_USR2	BIT15	
EVENT_USR3	BIT16	
RFU	BIT17	

Return Codes: None.

Get_Event ()

Purpose Get selected event vector.

Synopsis **(* void ()) Get_Event_Vector (** IN enum EVENT_ID eEventID **)**;

Parameters *eEventID*: Input parameter
Specifies for which event to return the current vector. Possible values are:

```
eEXT0,          // 0
eEXT1,          // 1
eEXT2,          // 2
eEXT3,          // 3
eTIMER0,        // 4
eTIMER1,        // 5
eICC,           // 6 – will return pointer to a Null_isr
eRTC,           // 7
eKEY_DETECT,    // 8
eUSB,           // 9
eVDDF,          // 10
eI2C,           // 11
eANALOG,        // 12
eUSR0,          // 13
eUSR1,          // 14
eUSR2,          // 15
eUSR3,          // 16
eSERIAL         // 17– will return pointer to a Null_isr
```

Return Value Selected Event vector.

Set_Event ()

Purpose Set selected event vector. Use this function to redirect an interrupt service routine to a customized function/routine. Care must be taken when using this function as other functions within the LAPI may no longer work.

Synopsis **Void Set_Event_Vector (**
IN enum EVENT_ID eEventID,
IN void (*pEventVector)(void) **);**

Parameters *eEventID*: Input parameter
Specifies for which event to add the handler. Possible values are:

```
eEXT0,          // 0
eEXT1,          // 1
eEXT2,          // 2
eEXT3,          // 3
eTIMER0,        // 4
eTIMER1,        // 5
RFU,            // 6
eRTC,           // 7
eKEY_DETECT,    // 8
eUSB,           // 9
eVDDF,          // 10
eI2C,           // 11
eANALOG,        // 12
eUSR0,          // 13
eUSR1,          // 14
eUSR2,          // 15
eUSR3,          // 16
```

RFU // 17

pEventVector: Input parameter
Pointer (vector) to the function to call when the event occurs.

Return Codes None.

The **eUSB** handler should check the **x.USBStatus** value and/or call the USB_Status() routine to determine which USB event occurred. All other events have unique causes.

4.2.12 Timers API – Available with all 73S12xxF Devices

The Timers API allows up to four 16-bit 10 ms timers to be run concurrently. Hardware timer T0 is dedicated for the Timers API. The API includes:

Timers_Init ()

Purpose Initialize all registers and functions associated with Timer 0 and Timer 1.

Synopsis **Void Timers_Init (** void **);**

Parameters None.

Return Codes None.

Wait()

Purpose Wait (10 x *nTimeWait*) milliseconds and then return.

Synopsis **Void Wait (** IN unsigned int nTimeWait **);**

Parameters *nTimeWait*: Input parameter
Specifies how many 10 msec units to wait before returning to the caller.

Return Codes None.

Wait_1ms()

Purpose Wait (*nTimeWait*) milliseconds and then return.

Synopsis **Void Wait (** IN unsigned int nTimeWait **);**

Parameters *nTimeWait*: Input parameter
Specifies how many 1 msec units to wait before returning to the caller.

Return Codes None.

Add_Timer()

Purpose Add a 10 ms software timer.

Synopsis **Unsigned Integer *Add_Timer** (IN Unsigned integer nDuration)

Parameters *nDuration*: Input parameter
Specifies duration of time in 10 ms units.

Return Value Pointer to added timer. If the value is zero (NULL), there are no timers available.

This function can be called inside an ISR. When the timer expires (*pTimer == 0), it will be automatically removed. If all timers are in use, a NULL pointer will be returned. The Process_Timers() routine must be called to keep the timer updated.

Add_Timer_Func()

Purpose Add a 10 ms software timer and the function to execute on timer expiration.

Synopsis **Unsigned Integer *Add_Timer_Func (**
IN unsigned int nDuration,
IN void (*pfExpire (void)) **);**

Parameters *nDuration*: Input parameter
Specifies duration of time in 10 ms units.
pfExpire: Input parameter
Specifies a pointer to the function to execute when the timer expires.

Return Value Pointer to the added timer. If value is zero, there are no timers available.

Remove_Timer()

Purpose Stop and remove the selected timer.

Synopsis **Void Remove_Timer (** IN unsigned integer *pTimerId **);**

Parameters *ptimerID*: Input parameter
Specifies which timer to stop and remove. This is the value returned by the Add_Timer() or Add_Timer_Func() functions.

Return Codes None.

Process_Timers()

Purpose Process and update the active timers. This function must be called from a foreground routine whenever there is active timer.

Synopsis **Void Process_Timers (** void **);**

Parameters None.

Return Codes None.

4.2.13 User IO API – Available with all 73S12xxF Devices

The USER IO Pins can be configured, individually, as an interrupt source to Timer 0, Timer 1, INT0 or INT1. For INT0 or INT1, the interrupt can be configured to occur on the rising edge/high level or falling edge/low level. Only INT0 can be used to wake up the CPU from sleep/halt mode. This API includes:

USR_INT_Config ()

Purpose Configure USER IO as an interrupt source for T0, T1, INT0 or INT1.

Synopsis **void USR_INT_Config (**
IN enum USRINTSRC usr_src,
IN enum USRINTSEL int_select,
IN Boolean Enable,
IN Boolean EdgeTrigger **)**

Parameters *usr_src*: Input parameter
Specifies which USER IO is to be used as the interrupt source. The available choices (defined in USRINTSRC in api_struct_12.h) are:

USR0SRC 0,
USR1SRC 1,
USR2SRC 2,
USR3SRC 3,
USR4SRC 4,
USR5SRC 5,
USR6SRC 6,
USR7SRC 7

Int_select: Input parameter
Selects the interrupt source (defined in USRINTSEL in api_struct_12.h) as:

NOT_USE1 0,
NOT_USE2 1,
SEL_TIMER0 2,
SEL_TIMER1 3,
SEL_INT0_HI_RISE 4,
SEL_INT1_HI_RISE 5,
SEL_INT0_LOW_FALL 6,
SEL_INT1_LOW_FALL 7;

Enable: Input parameter
If TRUE, enables the selected interrupt source; otherwise, disables it.

EdgeTrigger: Inupt parameter
For INT0 and INT1 only, sets the interrupt to be level (FALSE) or edge (TRUE).

Return Codes None.

USR_INT_Read()

Purpose Read the current interrupt settings of the specified USER IO pin.

Synopsis **enum USRINTSEL USR_INT_Read (** enum USRINTSRC usr_src **);**

Parameters *usr_src*: Input parameter
Corresponding USR pin to read the interrupt setting from.

Return Codes One of the following as defined in USRINTSEL:

NULL,
SEL_TIMER0 2,
SEL_TIMER1 3,
SEL_INT0_HI_RISE 4,
SEL_INT1_HI_RISE 5,
SEL_INT0_LOW_FALL 6,
SEL_INT1_LOW_FALL 7;

USER_IO_Config()

Purpose Configure the direction for the USER IO pins.

Synopsis **void USER_IO_ Config (** Unsigned char usrsrc, Unsigned char dir **)**

Parameters *usrsrc*: Input parameter
Corresponding USR pins to be configured as Input or Output. USRIO 0 through 7 will be configured according to the *Dir* parameter below.
Dir: Input parameter
Direction value (1=input, 0=output) for the selected pins.

Return Codes None.

USER_IO_Read()

Purpose Read the value of the USER IO pins.

Synopsis **void USER_IO_Read (** OUT char *user_dir, OUT char *user_data **);**

Parameters *User_dir:* Output parameter
Direction value (1=input, 0=output) for the selected pins.
User_data: Output parameter
Value of the corresponding USER IO pins. All outputs will reflect the last value written.

Return Codes None.

USER_IO_Write()

Purpose Write values to selected USER IO pins.

Synopsis **Void USER_IO_Write (** IN char cUserIO, IN char cUserIOselect **);**

Parameters *cUserIO*: Input parameter
Values to write to selected USER IO pins.
cUserIOselect: Input parameter
A '1' in the corresponding bit will enable writing to that USR pin provided it is configured as an output.

Return Codes None.

4.2.14 External Interrupts API – Available with all 73S12xxF Devices

This API allows direct access to the two external interrupt pins (EXT3 and EXT2). These two interrupts are only available as edge (falling/negative or rising/positive) sensitive. The API includes:

INT2_Config()

Purpose Configure External Interrupt 2.

Synopsis **void INT2_Config (** Unsigned char Enable, Unsigned char Polarity **);**

Parameters *Enable*: Input parameter
Enable (1) or disable (0) interrupt 2.
Polarity : Input parameter
Configure interrupt on rising edge (1) or falling edge (0).

Return Codes None.

INT2_Read()

Purpose Read value of External Interrupt 2.

Synopsis **void INT2_Read** (Unsigned char *polarity, Unsigned char *status)

Parameters *polarity*: Output parameter
Specifies the polarity of the external interrupt 2 pin, rising edge = 1, falling edge = 0.
status: Output parameter
External interrupt 2 edge flag.

Return Codes None.

INT3_Config()

Purpose Configure External Interrupt 2.

Synopsis **void INT3_Config (** Unsigned char Enable, Unsigned char Polarity **);**

Parameters *Enable*: Input parameter
Enable (1) or disable (0) interrupt 3.
Polarity: Input parameter
Configure interrupt on rising edge (1) or falling edge (0).

Return Codes None

INT3_Read()

Purpose Read value of External Interrupt 3.

Synopsis **void INT3_Read** (Unsigned char *polarity, Unsigned char *status **)**

Parameters *polarity*: Output parameter
Specifies the polarity of the external interrupt 3 pin, rising edge = 1, falling edge = 0.
status: Output parameter
External interrupt 3 edge flag.

Return Codes None.

4.2.15 Special Function Register API – Available with all 73S12xxF Devices

The API allows read/write access to all the 73S12xxF special function registers. The API includes:

- SFR_Read() (page 61)
- SFR_Write() (page 62)

SFR_Read()

Purpose Read from the specified Special Function Register.

Synopsis **SFR_RC SFR_Read (** IN char cSFRAddr, OUT char *pcSFRValue **);**

Parameters *cSFRAddr*: Input parameter
Specifies the address of the Special Function Register to be read.
pcSFRValue: Output parameter
Specifies the value read from the specified Special Function Register.

Return Codes SFR_OK Successful read from the SFR.
SFR_INVALID Invalid SFR referenced.

SFR_Write()

Purpose Write to the specified Special Function Register.

Synopsis **SFR_RC SFR_Write (** IN char cSFRAddr, IN char cSFR, IN char cSFROperation **);**

Parameters *cSFRAddr*: Input parameter
Specifies the address of the Special Function Register to be written.
cSFR: Input parameter
Specifies the value to use to either set, clear or assign bits of the Special Function Register specified.
cSFROperation: Input parameter
Specifies the operation to perform on the Special Function Register with the value supplied. Possible values are:

ASSIGN_SFR	0
AND_SFR	1
OR_SFR	2

Return Codes SFR_OK Successful write to the SFR.
SFR_INVALID Invalid or forbidden SFR referenced.

To set specific bits of the SFR, OR them with ‘1’s. To clear specific bits of the SFR AND them with ‘0’s.

4.2.16 Flash/Memory API – Available with all 73S12xxF Devices

Flash management assumes that the CPU is running at the default clock rate of 3.69 MHz. A Flash write is processed on a page basis. If a write to a Flash address overlays two pages, a two-page write operation will be performed.

The Flash write process involves 4 steps: Read, Erase, Verify, Write. Should any of these steps fail, the write operation will fail. The user must use caution when using these APIs as there will be no check in the LAPI for accidental writes. The API includes:

Flash_Init()

Purpose — Initialize the Flash management registers to values appropriate for the CPU running at the default speed.

Synopsis — **void Flash_Init (** void **);**

Parameters — None.

Return Codes — None.

memcpy_rx ()

Purpose — Flash management – use to write to a Flash page that the destination ROM address belongs to using the contents from the RAM source location. If the length of the source and the starting ROM location cause the write operation to span more than one 512-byte Flash page, the Read/Erase/Verify/Write will take place on all the pages involved. An erase operation will result in the Flash contents being set to 0xFF.

Synopsis —
Bbool memcpy_rx (
Unsigned char code *dst,
Unsigned char xdata *src,
Unsigned integer len **);**

Parameters —
dst: Input parameter
Specifies starting ROM address of Flash to be written (destination).
src: Input parameter
Use contents at this RAM address location as the source data.
len: Input parameter
Length (in bytes) of data to write to Flash.

Return Codes —
TRUE if the Write was successful.
FALSE if the Write was not completed.

memcpy_xx ()

Purpose — Memory management – use to copy the contents of external RAM (XRAM) location(s) to other XRAM location(s).

Synopsis —
memcpy_xx (
Unsigned char xdata *dst,
Unsigned char xdata *src,
Unsigned integer len **);**

Parameters —
dst: Input parameter
Destination: specifies starting address of XRAM to be written.
src: Input parameter
Use data at this XRAM address location as the source data.
len: Input parameter
Length (in bytes) of data to copy from source to destination.

Return Codes — None.

memcpy_xi ()

Purpose — Memory management – use to copy the contents of internal RAM (IRAM) location(s) to XRAM location(s).

Synopsis —
memcpy_xi (
Unsigned char xdata *dst,
Unsigned char idata *src,
Unsigned char len **);**

Parameters —
dst: Input parameter
Destination: specifies starting address of XRAM to be written.
src: Input parameter
Use data starting at this IRAM location as the source data.
len: Input parameter
Specifies the length (in bytes) to write to XRAM.

Return Codes — None.

memcpy_ix ()

Purpose — Memory management – use to copy the contents of XRAM locations to IRAM locations.

Synopsis —
memcpy_ix (
Unsigned char idata *dst,
Unsigned char xdata *src,
Unsigned char len **);**

Parameters —
dst: Input parameter
Destination: specifies starting address of IRAM to be written
src: Input parameter
Use data starting at this XRAM location as the source data.
len: Input parameter
Specifies the length (in bytes) to write to IRAM.

Return Codes — None.

memcmp_rx ()

Purpose — Memory management – use to compare the contents of an XRAM location and a ROM location.

Synopsis
```
Signed char memcmp_rx (
    Unsigned char code *dst,
    Unsigned char xdata *src,
    Unsigned integer len );
```

Parameters
dst: Input parameter
Specifies the starting address of the ROM data to be compared.
src: Input parameter
Specifies the starting address of the XRAM data to compare to.
len: Input parameter
Specifies the length (in bytes) of data to compare.

Return Codes — 0 if the compare is successful (data matched).
Non zero if the source and destination data do not match.

memcmp_xx ()

Purpose — Memory management – use to compare the contents of an XRAM location and another XRAM location.

Synopsis
```
Signed char memcmp_xx (
    Unsigned char xdata *dst,
    Unsigned char xdata *src,
    Unsigned integer len );
```

Parameters
dst: Input parameter
Specifies the starting address of the the first XRAM location to be compared.
src: Input parameter
Specifies the starting address of the the XRAM location to compare to.
len: Input parameter
Specifies the length (in bytes) of data to compare.

Return Codes — 0 if the compare is successful (data matched).
Non zero if the source and destination data do not match.

memset_x ()

Purpose — Memory management – use to fill the contents of XRAM with a specified value.

Synopsis
```
void memset_x (
    Unsigned char xdata *dst,
    Unsigned char s,
    Unsigned integer len );
```

Parameters
dst: Input parameter
Specifies the starting address of the XRAM locations to fill.
src: Input parameter
Specifies the value to fill the XRAM with.
len: Input parameter
Specifies the number of bytes to fill with the specified data.

Return Codes — None.

strlen_x ()

Purpose	Compute the string length of ASCII data in XDATA (XRAM).
Synopsis	**unsigned int strlen_x (** IN unsigned char xdata *psource **);**
Parameters	*psource*: Input parameter Specifies a pointer to the source data in XRAM.
Return Value	Length of the ASCII data string (in bytes) in the specified XRAM location.

strlen_r ()

Purpose	Compute the string length of ASCII data in ROM (Flash).
Synopsis	**unsigned int strlen_x (** IN unsigned char code *psource **);**
Parameters	*psource*: Input parameter Specifies a pointer to the source data in Flash.
Return Value	Length of the ASCII data (in bytes) in the specified Flash ROM location.

Log2 ()

Purpose	Compute the logarithm (base 2) of the input.
Synopsis	**unsigned int log2 (** IN unsigned int unumber **);**
Parameters	*unumber*: Input parameter The input value to determine the log2 of.
Return Value	log2 of the input.

4.2.17 Boot Loader and Passcode Management – Available with the LAPI-*BL.lib Only

The Boot Loader code occupies the first (lower) 512 bytes of the Flash program space (0x0000 - 0x01FF) including the passcode storage space. The Boot Loader assumes that the CPU is running at the default clock rate which is 3.69 MHz. As a result, the serial baud rate and all soft-timers (timer 0) are calculated based on the default CPU clock rate. The Boot Loader also only accepts Intel Hex files via the Serial RS-232 interface. The Boot Loader and Passcode Management API include:

- Boot() (page 68)
- CheckPassCode () (page 68)
- SetPassCode ( ) (page 69)

For security and authorization enforcement, the passcode is implemented, embedded and validated within the Boot Loader code. The Boot Loader will return a hardware error if Security Mode 0 or Security Mode 1 has already been enabled. Figure 9 depicts a successful Boot Loader scenario.

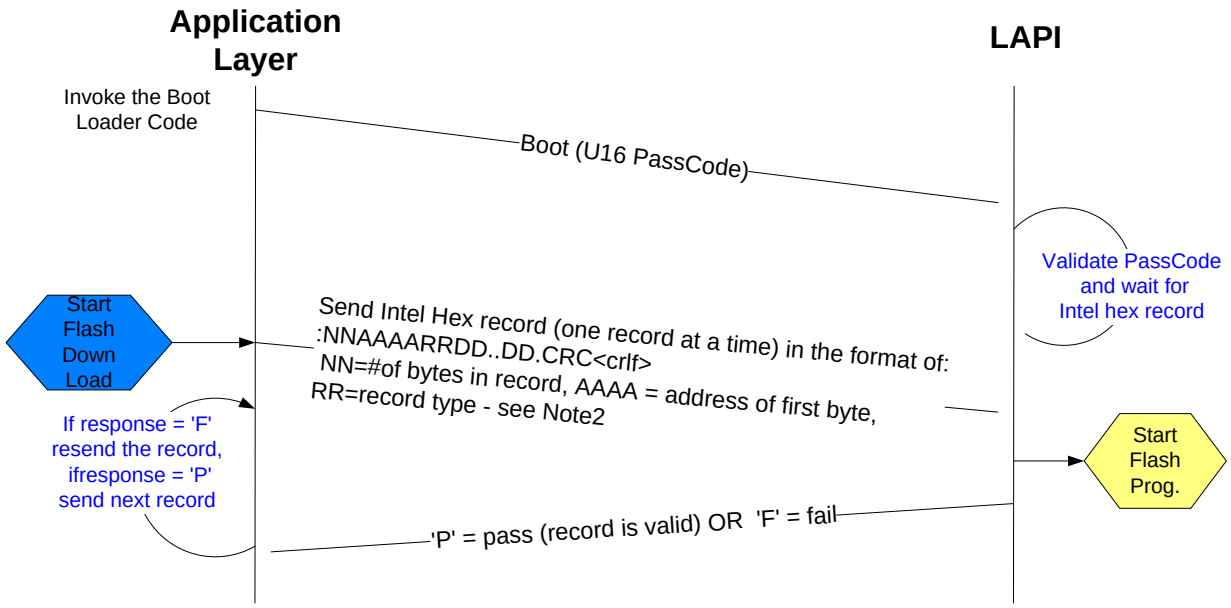

Figure 9: Boot Loader Scenario

The Passcode is a 2-byte integer data type where the second byte is the complement of the first byte. After validating the Passcode, the LAPI will configure the device's serial interface to be running at **115200 baud with 8 data bits, No stop bit and Xon/Xoff control enabled.** It then waits for a valid Intel hex record from the Serial RS-232 interface. The record's checksum is checked along with its Flash address location. The Boot Loader code area (the first page of Flash – 512 bytes at address 0x0000 – 0x01FF) will be protected. Any hex record with addresses on the first page will be ignored. Once a valid hex record is accepted by the device, all Flash space (except for the Boot Loader area) will be permanently erased. For each successful hex record received by the device, the device will respond with a 'P'. For each failed hex record received by the device (bad CRC), the device will respond with an 'F'. When the device receives the last successful hex record and if there is no error, the device will immediately jump to the start of the new application. For a sample of the usage of the Boot Loader API, review the Pseudo-CCID application code (built and shipped in a separate TSC 73S12xxF PCCID Serial release).

Figure 10 illustrates the Flash Download and Flash Programming process.

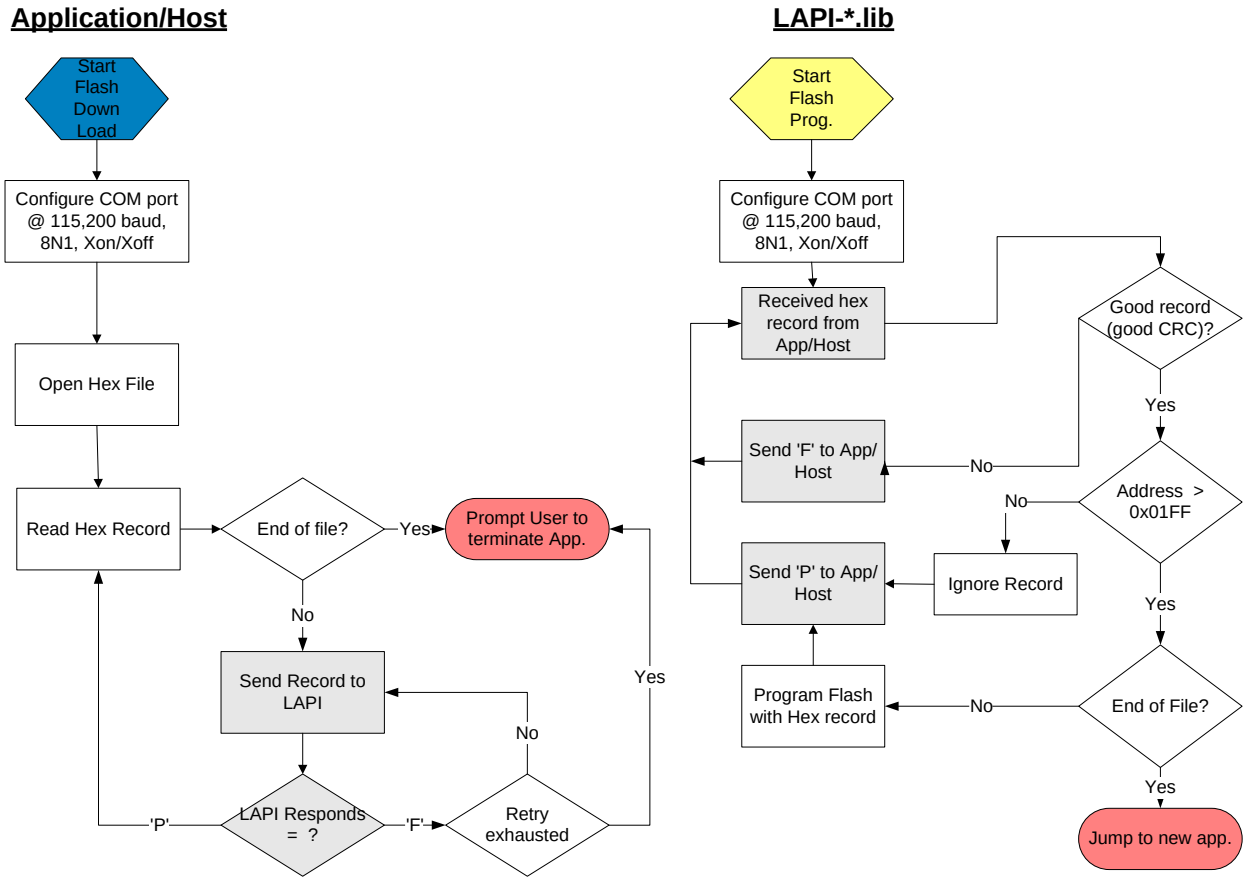

Figure 10: FLASH Download and Programming Process

Boot()

Purpose	Invoke the Boot Loader function to start reprogramming of Flash.
Synopsis	**void Boot (** U16 PassCode **);**
Parameters	*PassCode*: Input parameter Specifies the 2-byte passcode.
Return Codes	FALSE if the PassCode fails validation.

Once an application calls the Boot function, and the PassCode is validated, control will not be returned.

CheckPassCode ()

Purpose	Validate the PassCode for Security Mode management, PassCode management and the Boot function.
Synopsis	**Bbool CheckPassCode (** U16 PassCode **);**
Parameters	*PassCode*: Input parameter Specifies the 2-byte passcode.
Return Codes	TRUE if the PassCode is validated. FALSE if the PassCode doesn't match with the passcode stored in the Boot Loader code space.

SetPassCode ( )

Purpose	Change the PassCode to a new value. The first (and default) passcode is 0x5AA5 and is stored at location 0x01E0 and 0x01E1. When SetPassCode() is executed the first time, it writes 0x0000 to these two addresses. The new PassCode will be written at 0x01E2 and 0x01E3. Each time this function is called, the new PassCode will be written at the next two consecutive address locations and the location of the OldPassCode will be over-written with a value of 0x0000. Once the new PassCode reaches the last location (address location 0x01F1), this function will no longer be allowed to change the PassCode.
Synopsis	**SetPassCode (** U16 OldPassCode, U16 NewPassCode **);**
Parameters	*OldPassCode*: Input parameter Old PassCode as stored in the Boot Loader code space. *NewPassCode*: Input parameter NewPassCode to be stored in the new passcode location.
Return Codes	TRUE if both the OldPassCode is validated and the number of passcode changes has not exceeded 8. (The total number of passcode changes allowed is 8)

4.2.18 Security Mode Management - Available with the LAPI-*BL.lib Only

There are three possible security modes, defined as MODE0, MODE1 and MODE2. MODE0 and MODE1 are directly controlled by the hardware by a call to the LAPI. MODE2 is controlled by the application layer using the PassCode mechanism as designed by LAPI. A correct PassCode is required before the security mode can be set. The Security Mode Management API includes:

- SETSecurity () (page 70)
- SECStatus () (page 70)

The processes which occur when initiating each of the three modes are described below. Table 5 shows the actions allowed during each mode.

Mode 0

1. Flctl SFR (0xB2 bit 6) is already set in the current flash program.
2. Setup the Fuse Control Register.
3. Setup the Security Ctl Register.
4. Enable the Trim Pulse Ctl Register

After Mode 0 is executed, a full circuit reset must be done for mode 0 to be in effect.

Mode 1

1. SEC (JP15 on EVB) bit set to HI.
2. Setup the Fuse Control Register.
3. Setup the Security Ctl Register.
4. Enable the Trim Pulse Ctl Register

Mode 2

This mode is strictly firmware and is implemented at the application level by calling the SetPassCode ( ) API.

1. If the Passcode is valid and has been modified less than 8 times, the application should loop through to exhaust the number of times the passcode change is allowed.
2. Scramble the last passcode with an invalid value then write it to the last designated location of the passcode. The TSC Pseudo-CCID for Serial RS-232 release contains application source code that demonstrates this mode. Please contact a Teridian Sales Representative for a copy.

If the passcode has changed 8 times prior to entering Mode 2, it will be disabled since no more passcode changes are allowed. Since this mode is controlled by firmware, it can be reset/re-initialized by using the ICE to reprogram the Flash.

Table 5: Security Mode Actions Allowed

Action	Mode 0	Mode 1	Mode 2
PassCode Change	No	Yes[2]	No
Program Space Visible via the ICE	Flctl SFR bit 6: Set - No Not Set - Yes	No	Yes
Flash Programmable via ICE	Yes[1]	No	Yes
Flash Program via Boot Loader	Yes	Yes[2]	No

[1] To reprogram the Flash using the ICE, hit the Erase button via the ICE, hit Reset on the EVB, then reload the Flash. If the new program has the FlCtl SFR security mode bit (bit 6) set, the ICE will continue to be disabled (this is the case for all firmware with 'BL' in the filename).

[2] The Passcode and Boot Loader reside on the first page of Flash, which is protected by Mode 1. In order to completely protect the Flash, enable Mode 2 which will disable Passcode and Boot Loader changes.

SETSecurity ()

Purpose — Set the hardware security to Mode 0 or Mode 1. Care must be taken as once this API is finished, the action is not reversible.

Synopsis — **void SETSecurity** (U08 Mode)

Parameters — *Mode*: Input parameter
MODE0 – This mode only works when the Flash Control SFR (0xB2 bit 6) is set. LAPI-MBL.lib was built with this SFR already setup for this mode.
MODE1 – The security bit (JP15 on the 1215 EVB) must be set high in order to set this mode.

Return Codes — None.

SECStatus ()

Purpose — The current hardware security status of the part. This function only returns the hardware security status (MODE 0 or MODE 1). Since Mode 2 is a firmware/application level control, it will not be reported.

Synopsis — **U08 SECStatus** (void)

Parameters — None.

Return Codes — MODE0 = BIT0 (0x01) – returned if the part is already in MODE 0.
MODE1 = BIT1 (0x02) – returned if the part is already in MODE 1.
0x00 – returned if no security mode setup has been performed on the part.

4.2.19 Other Miscellaneous API Calls – Available with all 73S12xxF Devices

Several API calls are provided to help with initializing or re-initializing all the registers, interrupts and events. As a general guideline, API_Init() should always be called at the beginning of the application main routine. The miscellaneous APIs include:

API_Init()

Purpose	Enable TIMER0 and TIMER1 interrupts, clear all pending USB interrupts, set the CPU to run at 3.69 MHz. Always call this routine at the beginning of an embedded application.
Synopsis	**void API_Init (** void **);**
Parameters	None.
Return Codes	None.

Soft_Reset ()

Purpose	Initialize all 73S12xxF specific internal and external SFRs to their hardware power-on defaults. Initialize Smart Card SFRs to hard reset default values. Initialize USB SFRs to hard reset default values. Initialize most 80515 SFRs.
Synopsis	**void Soft_Reset (** void **);**
Parameters	None.
Return Codes	None.

4.3 High-Level API

The 73S12xxF comes with a high-level API library to control the Smart Card. This library is linked to the low-level API as described in Section 4.2 Low-level API for all smart card communication controls.

4.3.1 Smart Card Control

This API provides support for the asynchronous (T=0) and (T=1) Smart Card protocol management. Each Smart Card interface is individually addressed by specifying the correct **eIccId** value. Up to 9 smart card slots can be configured, with the first (ICC_1ST) being the internal slot and the next 8 being external slots. The API provides specific support for several test suite constraints (Microsoft WHQL (aka HCT), EMVCo (MCI and VCI)) by adding several options to most of the API functions.

The Smart Card API includes:

- ICC_Enable() or ICC_Enable_Ext () (page 73)
- ICC_WarmReset() (page 75)
- ICC_PTSNegotiate() (page 76)
- ICC_Send() (page 77)
- ICC_Send_Ext() (page 78)
- ICC_Configure_Ext () (page 79)
- ICC_Configure() (page 82)
- ICC_Disable() (page 83)
- ICC_CheckPresence() (page 84)

In order to use this API, the following minimum set of files is required:

- IccMgt.h: header file which contains the prototypes for the API services.
- Icc_api-*.lib: this is the ICC library itself.
- Allocate.c: allows configuring the number of ICC interfaces used (therefore, only the memory for the used interfaces is reserved).
- Api_12.h and API_Struct_12.h: header files for low-level API.
- Portable.h: contains definitions for C code portability.

Several applications are included in the release which deal with multiple smart card slots. In order for the multiple smart card slot feature to function properly, it must first be configured using the ICC_Configure() API. For a sample utilization of the ICC library, see the sample applications: CCID firmware or Pseudo-CCID firmware.

Configuring the ICC Interfaces

The ICC_USED_INTERFACE_NUMBER and INTERFACE_USED[9] definitions in the Allocate.c file must be modified as shown below before using the API services. In this example, Smart card slots 1 and 2 are to be utilized.

```
#define ICC_USED_INTERFACE_NUMBER      2  //two smartcard slots

unsigned char code INTERFACE_USED[9] =
{
ICC_1ST,
ICC_2ND,
ICC_NONE,
ICC_NONE,
ICC_NONE,
ICC_NONE,
```

```
ICC_NONE,
ICC_NONE,
ICC_NONE
};
```

ICC_Enable() or ICC_Enable_Ext ()

Purpose

Activate the Smart Card interface slot specified by *eIccId*, and return the ATR to the application. ICC_Enable_Ext() is the extended function. The extended version has one additional parameter as input. This last parameter is a pointer to a callback function to be called whenever the card sends a request for wait time extension (S(WTX) in T=1 or 0x60 (NULL) in T=0 per the *CCID Specification*).

Synopsis

```
AR_ICC_RC ICC_Enable (
    IN enum ICC_ID eIccId,
    IN BOOLEAN bIccATRAutoCheck,
    IN BOOLEAN bIccEMVCompliant,
    IN BOOLEAN bIccHighSpeed,
    IN BOOLEAN bIccClockStopEnable,
    IN enum ICC_POWER ucIccPowerSelect,
    OUT unsigned char *pucIccATR
    OUT unsigned char *pucIccATRLength );
or
AR_ICC_RC ICC_Enable_Ext (
    IN enum ICC_ID eIccId,
    IN BOOLEAN bIccATRAutoCheck,
    IN BOOLEAN bIccEMVCompliant,
    IN BOOLEAN bIccHighSpeed,
    IN BOOLEAN bIccClockStopEnable,
    IN enum ICC_POWER ucIccPowerSelect,
    OUT unsigned char *pucIccATR
    OUT unsigned char *pucIccATRLength,
    IN  (Send_WTE) ( void ) );
```

Parameters

eIccId: Input parameter.
The lower (least significant) 4-bits specify which Smart Card interface is to be activated. Possible values are:

ICC_1ST 0, (Internal)
ICC_2ND 1, (External)
…
ICC_9TH 8 (External).

The higher (most significant) 4-bits specify whether the card detect polarity is high (CARD_DET_H) or low (CARD_DET_L). See additional details at the end of the description for this function.

bIccATRAutoCheck: Input parameter
Specifies whether the Smart Card interface must be immediately de-activated in the case of an unsupported ATR (TRUE) or if the ATR must be transmitted back to the application, which will decide what needs to be done (e.g. performing a Warm Reset). [The default value should be TRUE, but for the various test suites, it may be necessary for the application to decide whether or not the interface is to be deactivated.]

bIccEMVCompliant: Input parameter
Specifies whether the Smart Card interface is to be managed according to the EMV Specification (TRUE) or to the ISO 7816-3 standard (FALSE). This is especially important on TimeOut errors and on IFSD negotiation. On TimeOut errors in the T=1 protocol, EMV specifies that the IFD must de-activate the Smart Card interface,

whereas the ISO 7816-3 standard allows an error recovery procedure. EMV specifies that the IFD must initiate an IFSD negotiation before the first command transmission in (T=1) protocol. When EMV mode is specified, *bIccIFSDRequestT1* is set to TRUE and *bIccDeactivatedOnTimeOutErrorT1* is set to TRUE.

bIccHighSpeed: Input parameter
Specifies if the reader has to switch to the high speed clock (7.384 MHz) if the smart card allows it (according to parameter *Fi* in the ATR). If this parameter is FALSE, the reader will always use the default clock, namely 3.692 MHz. If the value is TRUE, the reader will switch to 7.384 MHz after the ATR, if the smart card allows this speed.

bIccClockStopEnable: Input parameter
If this flag is set to TRUE and the smart card allows clock stop/start, the clock will be stopped when there is no transaction. That is to say that the clock is stopped after a command is done and restarted before the next command will be executed.

ucIccPowerSelect: Input parameter
Specifies the voltage that is to be applied to the card. Possible values are:

ICC_POWER_AUTOMATIC	00, (automatic voltage selection according to ISO 7816-3 standard)
ICC_POWER_SET_5	01,
ICC_POWER_SET_3	02, Other values are RFU.

pucIccATR: Output parameter
The ATR value returned by the Smart Card.

pucIccATRLength: Output parameter
Length of the ATR returned by the Smart Card. This value should not exceed 32, as the ATR cannot contain more than 32 bytes.

Send_WTE: Input parameter
Call this function whenever the card sends a Wait Time Extension request (in T=1, whenever the card sends an S(WTX) block; for T=0, whenever the card sends NULL (0x60) when it is waiting for data from the reader). This functionality is required by the CCID Specification. This callback function will NOT be called if the smart card is called for EMV mode because of the strict timing requirements of the EMV Specification.

Return Codes

AR_ICC_OK
Successful operation. The Smart Card was activated and the ATR is supported by the chip.

AR_ICC_ERR_BAD_PARAM
An invalid parameter (eIccId for example) was specified.

AR_ICC_ERR_CARD_MUTE
The Smart Card is mute.

AR_ICC_ERR_CARD_ABSENT
No Smart Card is inserted.

AR_ICC_ERR_CARD_DISCONNECTED
The Smart Card was deactivated.

AR_ICC_ERR_CARD_OVERLOAD
The Smart Card has generated an over current.

AR_ICC_ERR_CARD_UNSUPPORTED_ATR
The Smart Card ATR is not supported by the chip and is stored in *pucIccAtr*. The Smart Card interface has not been de-activated (*bIccATRAutoCheck* option is not selected).

AR_ICC_ERR_BAD_TCK
The ATR has a bad TCK byte.

AR_ICC_ERR_BAD_TS
The ATR has a bad TS byte.

AR_ICC_ERR_CARD_COMM_PB
The ATR has either a parity error, an Rx over-run or a VCC unstable error.

The *eIccId* parameter is split into two fields: Card Detect Polarity and Card Slot number, by using the most significant nibble and least significant nibble, respectively. Therefore, the most significant nibble of the *eIccId* parameter is used to determine if the Card Detect Polarity is to be configured High or Low. The constants CARD_DET_H and CARD_DET_L are defined in ICCMgt.h. Examples:

1. If Card Detect Polarity is to be set to High when a card is inserted, it can be done so by calling: ICC_Enable(eIccId | CARD_DET_H, bIccATRAutoCheck,etc.).
2. If Card Detect Polarity is to be set to Low when a card is inserted, it can be done so by calling: ICC_Enable(eIccId | CARD_DET_L, bIccATRAutoCheck,etc.).
3. Calling ICC_Enable (eIccId, bIccATRAutoCheck, etc.) without OR'ing *eIccId* with CARD_DET_L or CARD_DET_H (in other words, eIccID <= 0x09) will default to CARD_DET_H.

ICC_WarmReset()

Purpose — Perform a Warm Reset on the Smart Card and return the ATR to the application.

Synopsis

```
AR_ICC_RC ICC_WarmReset (
    IN enum ICC_ID eIccId,
    OUT unsigned char *pucIccATR,
    OUT unsigned int *punIccATRLength );
```

Parameters

eIccId: Input parameter
Specifies which Smart Card interface is to be activated. Possible values are:
ICC_1ST 0, (Internal)
ICC_2ND 1, (External)
…
ICC_9TH 8 (External).

pucIccATR: Output parameter
ATR value returned by the Smart Card.

punIccATRLength: Output parameter
Length of the ATR returned by the Smart Card. This value should not exceed 32, as the ATR cannot contain more than 32 bytes.

Return Codes

AR_ICC_OK
Successful operation. The received ATR is stored in pucIccATR.
AR_ICC_ERR_BAD_PARAM
An invalid parameter (eIccId for example) was specified.
AR_ICC_ERR_CARD_MUTE
The Smart Card is mute.
AR_ICC_ERR_CARD_ABSENT
No Smart Card is inserted.
AR_ICC_ERR_CARD_DISCONNECTED
The Smart Card was deactivated.
AR_ICC_ERR_CARD_OVERLOAD
The Smart Card has generated an over current.
AR_ICC_ERR_CARD_UNSUPPORTED_ATR
The Smart Card ATR is not supported by the chip and is stored in *pucIccAtr*. The Smart Card interface has not been de-activated (the *bIccATRAutoCheck* option not selected).
AR_ICC_ERR_BAD_TCK
The ATR has a bad TCK byte.
AR_ICC_ERR_BAD_TS
The ATR has a bad TS byte.
AR_ICC_ERR_CARD_COMM_PB
The ATR has either a parity error, an Rx over run or a VCC unstable error.

ICC_PTSNegotiate()

Purpose

Transmit the PTS negotiation request to the Smart Card and verify its answer. When the PTS negotiation succeeds, the 73S12xxF configures its internal protocol parameters with the negotiated values.

Synopsis

```
AR_ICC_RC ICC_PTSNegotiate (
    IN enum ICC_ID eIccId,
    IN BOOLEAN bIccChangeProtocol,
    IN BOOLEAN bIccChangeSpeed,
    IN BOOLEAN bIccSelectT1Protocol,
    IN unsigned char ucIccFiDi,
    OUT unsigned char *pucIccPTS,
    OUT unsigned int *punIccPTSLength );
```

Parameters

eIccId: Input parameter
Specifies which Smart Card interface is to be activated. Possible values are:
ICC_1ST 0, (Internal)
ICC_2ND 1, (External)
…
ICC_9TH 8 (External)

bIccChangeProtocol: Input parameter
Specifies whether the protocol is to be changed (TRUE) or not.

bIccChangeSpeed: Input parameter
Specifies whether the speed is to be changed (TRUE) or not.

bIccSelectT1Protocol: Input parameter
Specifies whether T=1 protocol is selected (TRUE) or T=0 (FALSE). This parameter is ignored if *bIccChangeProtocol* is set to FALSE.

ucIccFiDi: Input parameter
Specifies the new bit rate and clock adjustment factors to be used (in case *bIccChangeSpeed* is set to TRUE)

pucIccPTS: Output parameter
Contains the PTS answer received from the Smart Card.

punIccPTSLength: Output parameter
Contains the length of the PTS answer received from the Smart Card. This value should be in the range 03 to 06.

Return Codes

AR_ICC_OK
Successful operation. The PTS negotiation succeeded. The PTS answer from the Smart Card is stored in *pucIccPTS*.

AR_ICC_ERR_CARD_MUTE
The Smart Card does not respond to the PTS negotiation.

AR_ICC_ERR_CARD_ABSENT
No Smart Card is inserted.

AR_ICC_ERR_CARD_DISCONNECTED
The Smart Card was removed during the PTS negotiation operation.

AR_ICC_ERR_CARD_OVERLOAD
The Smart Card has generated an overload.

AR_ICC_ERR_BAD_PARAM
Incorrect bytes were sent by the card.

ICC_ERR_PTS_NEGOTIATION
FiDi to negotiate is unacceptable.

ICC_Send()

Purpose

Transmit a data command to the Smart Card interface and wais for the answer from the Smart Card. This function is responsible for the Smart Card command format analysis which computes the correct command case (1, 2 Short, 3 Short or 4 Short). The API assumes that the data record size is 255 bytes or less which is why it supports short cases only.

Synopsis

```
AR_ICC_RC ICC_Send (
    IN enum ICC_ID eIccId,
    IN unsigned int unIccCommandLength,
    IN unsigned char *pucIccCommand,
    OUT unsigned int *punIccResponseLength,
    OUT unsigned char *pucIccResponse,
    OUT unsigned char *pucSW1,
    OUT unsigned char *pucSW2,
    OUT BOOLEAN *pbitStatusJustAfterHeader );
```

Parameters

eIccId: Input parameter
Specifies which Smart Card interface is to be activated. Possible values are:
ICC_1ST 0, (Internal)
ICC_2ND 1, (External)
…
ICC_9TH 8 (External)

unIccCommandLength: Input parameter
Specifies the number of bytes of data pointed to by *pucIccCommand*.

pucIccCommand: Input parameter
Specifies the command to be transmitted to the Smart Card interface.

punIccResponseLength: Output parameter
Specifies the number of bytes of response data pointed to by *pucIccResponse*. (The length value does not include the status bytes SW1/SW2).

pucIccResponse: Output parameter
Contains the response of the Smart Card (SW1/SW2 not included).

pucSW1: Output parameter
Contains the received SW1value.

pucSW2: Output parameter
Contains the received SW2 value.

pbStatusJustAfterHeader: Output parameter
This boolean parameter specifies if the status bytes have been received just after the header (TRUE) or after the data (FALSE) [Useful for the EMV Tests suite]. This parameter must be taken into account only if the ICC_Configure() command sets the pbIccWarningStatusBytesManagementT0 bit.

Return Codes

AR_ICC_OK
Successful operation: the command was successfully transmitted to the Smart Card and a valid response was received.

AR_ICC_ERR_BAD_PARAM
An inconsistent command was specified. The API was unable to compute the command case value.

AR_ICC_ERR_CARD_MUTE
The Smart Card is mute.

AR_ICC_ERR_CARD_ABSENT
No Smart Card is inserted.

AR_ICC_ERR_CARD_DISCONNECTED
The Smart Card was removed during the activation operation.

AR_ICC_ERR_WRONG_LEN
Either the command case (T=0) is not correctly formatted, or the buffer size specified is too small; especially in a Case 2 where the card sometimes responds with a 61xx with xx > specified buffer size.

AR_ICC_ERR_CARD_COMM_PB
Too many errors occurred, so the interface has been closed.

AR_ICC_ERR_CARD_COMM_PB
There is a communication error between the reader and smart card such as a parity error, a bad response block from the card, a bad EDC, a transmission error or a bad procedure byte.

ICC_Send_Ext()

Purpose

Transmit a data command to the Smart Card interface and wait for the answer. This API is very similar to the ICC_Send() API except it requires a MaxBuffSize. The application is responsible for the Smart Card command format analysis to compute the correct command case (1, 2 Short and Extended, 3 Short and Extended or 4 Short and Extended). Calling ICC_Send_Ext() with a MaxBuffSize of 260 bytes (255 data bytes + 5 header bytes) is equivalent to calling ICC_Send. The ISO 7816-4 standard specifies that setting the first byte (of 3 bytes; or the 5th byte of the command header bytes) of the data length field equal to 0x00 indicates the extended cases; whereas JICSAP (version 1.1) specifies the value of this byte to be 0xFF. This API is written to accept the ISO format so any command with the fifth byte of the command header having a value of 0x00 will be treated as an extended case.

Synopsis

```
AR_ICC_RC ICC_Send_Ext (
    IN enum ICC_ID eIccId,
    IN unsigned int unIccCommandLength,
    IN unsigned char *pucIccCommand,
    IN unsigned int MaxBuffSize,
    OUT unsigned int *punIccResponseLength,
    OUT unsigned char *pucIccResponse,
    OUT unsigned char *pucSW1,
    OUT unsigned char *pucSW2,
    OUT BOOLEAN *pbStatusJustAfterHeader );
```

Parameters

eIccId: Input parameter
Specifies which Smart Card interface is to be activated. Possible values are:
ICC_1ST 0, (Internal)
ICC_2ND 1, (External)
…
ICC_9TH 8 (External)

unIccCommandLength: Input parameter
Specifies the number of bytes of data pointed to by *pucIccCommand*.

pucIccCommand: Input parameter
Specifies the command to be transmitted to the Smart Card interface.

MaxBuffSize: Input parameter
Specifies the maximum buffer size the reader should reserve to hold the data bytes sent from the card.

punIccResponseLength: Output parameter
Specifies the number of bytes of response data pointed to by *pucIccResponse*. (This length value does not include the status bytes SW1/SW2)

pucIccResponse: Output parameter
Contains the response of the Smart Card (SW1/SW2 not included).

pucSW1: Output parameter
Contains the received SW1value.

pucSW2: Output parameter
Contains the received SW2 value.

pbStatusJustAfterHeader: Output parameter
This boolean parameter specifies if the status bytes have been received just after the header (TRUE) or after the data (FALSE) [Useful for the EMV Tests suite]. This parameter must be taken into account only if the ICC_Configure() command set the *pbIccWarningStatusBytesManagementT0* bit.

Return Codes

AR_ICC_OK
Successful operation: the command was successfully transmitted to the Smart Card and a valid response was received.

AR_ICC_ERR_BAD_PARAM
An inconsistent command was specified. The API was unable to compute the command case value.

AR_ICC_ERR_CARD_MUTE
The Smart Card is mute.

AR_ICC_ERR_CARD_ABSENT
No Smart Card is inserted.

AR_ICC_ERR_CARD_DISCONNECTED
The Smart Card was removed during the activation operation.

AR_ICC_ERR_WRONG_LEN
Either the command case (T=0) is not correctly formatted, or the buffer size specified is too small; especially in a Case 2 where the card sometimes responds with a 61xx where xx > specified buffer size.

AR_ICC_ERR_CARD_COMM_PB
Too many errors with the Smart Card occurred, so the interface has been closed.

AR_ICC_ERR_CARD_COMM_PB
There is some communication error between the reader and smart card such as a parity error, a bad response block from the card, a bad EDC, a transmission error or a bad procedure byte.

ICC_Configure_Ext ()

Purpose

Get and Set the configurable protocol parameters of the specified Smart Card interface. This function was developed to support different conformance tests and different hardware configuration. This API should be called as the first HAPI call to make sure the hardware configuration is setup properly according to the hardware design.

This API is recently added to the CCID USB release version 1.50 (or PCCID release later than version 3.10). The API is an extension of ICC_Configure() to provide backward compatibility. Either ICC_Configure_Ext () or ICC_Configure() should be called, not both. ICC_Configure is exactly the same as ICC_Configure_Ext with default values for ICC_HWConfigure_t such that: IccHz = ICC_3600KHZ), DebouncePUEnable = SC_DEBOUNCEON | TRUE and DebouncePDEnable = SC_DEBOUNCEOFF | FALSE.

Synopsis

```
AR_ICC_RC ICC_Configure_Ext (
    IN enum ICC_ID eIccId,
    IN enum ICC_ADDR IccAddr,
    IN enum ICC_CARDEVENT IccCE,
    IN enum I2C_USRIO eIccUsrIO,          // only use this option in the case of single-8010
                                          // controlling multiple external slots.
    IN BOOLEAN bIccSetOperation,
    INOUT ICC_Configure_t *ptrConfigure,
```

```
    //This is newly added from version 1.50 release
    IN ICC_HWConfigure_t  ptrHWConfigure );
Struct ICC_Configure_t
{
    INOUT unsigned char ucIccIFSD;
    INOUT unsigned char ucIccNAD;
    //this variable when initialize to 0xFF the GetResponse command will carry the CLA
    //byte of the last C-APDU.  UcIccCLA = 0x00 for EMV, = 0xFF for non-EMV
    INOUT unsigned char ucIccCLA;
    INOUT unsigned char ucIccTSTimeOut;
    INOUT unsigned char ucIccRxErrorCounterT0;
    INOUT unsigned char ucIccTxErrorCounterT0;
    INOUT unsigned char ucIccTxErrorCounterT1;
    INOUT unsigned char ucIccConfigurationByte;
};
Struct ICC_HWConfigure_t
{
    IN enum ICC_HZ    Icc_Hz;   //Smart Card Clock Frequency desired by application.
    IN unsigned char  DebouncePUEnable;
    IN unsigned char  DebouncePDEnable;
}
```

Parameters

eIccId: Input parameter
Specifies which Smart Card interface is to be activated. Possible values are:

```
ICC_1ST   0, (Internal)
ICC_2ND   1, (External)
…
ICC_9TH   8 (External)
```

IccAddr: Input parameter
Specifies the address for the external I2C slot.

```
ICC_I2C0  0x40, (1st external slot)
ICC_I2C1  0x42,
ICC_I2C2  0x44,
…
ICC_I2C7  0x4E; (Last external slot)
```

IccCE: Input parameter
Specifies assignment of the INT2/INT3 pins for card events. Possible values are:

```
ICC_INT2_NONE      0x00,
ICC_INT2_I2C       0x01,
ICC_INT3_I2C       0x02;
```

bIccSetOperation: Input parameter
Specifies if the function is called to perform a Set operation (TRUE) or a Get operation (FALSE).

pucIccIFSD: Input/output parameter
Specifies the IFSD value to be used (or being used). [Default value is 32, as specified by ISO/IEC 7816-3]

pucIccNAD: Input/output parameter
Specifies the NAD value to be used (or being used). [Default value is 00]

pucIccCLA: Input/output parameter
Specifies the CLA value to be used (or being used) when performing a GetResponse in case 2 / 4 in the T=0 protocol. Setting *pucIccCLA* equal to 0xFF indicates that the GetResponse command shall echo the class byte of the APDU command. [Default value is 00]

punIccTSTimeOut: Input/output parameter
Specifies the maximum delay in clock cycles between the de-assertion of the RST signal and the leading edge of the TS character of the ATR. [Default value is 40 000]

pucIccRxErrorCounterT0: Input/output parameter
Specifies the maximum number of errors allowed when the Interface Device is in reception mode in T=0 protocol.

pucIccTxErrorCounterT0: Input/output parameter
Specifies the maximum number of errors when the Interface Device is in transmission mode in T=0 protocol.

pucIccTxErrorCounterT1: Input/output parameter
Specifies the maximum number of errors when T=1. The most significant nibble gives the maximum number of R(NA) block transmissions, while the least significant nibble gives the maximum number of I or S-block retransmissions.

ucIccConfigurationByte: Input/output parameter
This byte contains the following configuration bits:

[b0] *bIccIFSDRequestT1*: Input/output. Specifies if the next I-Block will be preceded by an S(IFS request) (TRUE) or not (FALSE).

[b1] *bIccDeactivatedOnTimeOutErrorT1*: Input/output. Specifies if the Smart Card interface is to be de-activated (TRUE) on a TimeOut error. Otherwise, an error recovery procedure is engaged.

[b2] *bIccNADErrorCheckingT1*: Input/output. Specifies whether the NAD errors are to be checked (TRUE) or not (FALSE).

[b3] *bIccABORTManagementT1*: Input/output. Specifies whether the ABORT Request is to be managed (TRUE) or if the contacts are to be de-activated on S(ABORT Request) reception (FALSE).

[b4] *bIccRESYNCHManagementT1*: Input/output. Specifies whether an S(RESYNCH Request) command is to be sent when too many errors occur (TRUE) or if a de-activation sequence is to be initiated (FALSE).

[b5] *bIccRetransmitLastBlockAfterRESYNCHT1*: Input/output. Specifies whether the last block in T=1 protocol is to be retransmitted after a resynchronization occurs or the whole command. [Useful for Microsoft IFDTESTs suite]

[b6] *bIccWarningStatusBytesManagementT0*: Input/output. Specifies if the IFD must inform the application level of whether the status bytes have been received just after the command header transmission or after the command data transmission (in T=0 protocol). [Useful for EMV Test suites]

[b7] *bIccDeactivateOnIFSDNegotiationError*: Input/output. Specifies if the Smart Card interface is to be de-activated on an IFSD negotiation error.

Icc_Hz: Input parameter
Specifies the desired Smart Card clock frequency. Available values are defined in the API_Struct_12.h file (LAPI). The default value for both the internal and external slots is 3.69 MHz.

Care must be taken to make sure the Smart Card Clock is slower than the CPU clock. Since the CPU has much more overhead to process, a faster Smart Card clock may out run the CPU resulting in unexpected or undesirable delays.

Since the Smart Card Clock for the external slots (slot ICC_2ND or higher) is driven by the 73S12xxF single source, all external slots will share the same SC clock configuration. For example, if an application sets the first external slot to one rate, subsequent calls to this function with a different rate will be ignored.

DebouncePUEnable: Input parameter
The higher order (most significant) nibble of this byte enables (SC_DEBOUNCEON) or disables (SC_DEBOUNCEOFF) card debounce. The low order (least significant) nibble of this byte enables (TRUE) or disables (FALSE) the Pull-Up.

DebouncePDEnable: Input parameter
The higher order (most significant) nibble of this byte enables (SC_DEBOUNCEON) or disables (SC_DEBOUNCEOFF) card debounce. The low order (least significant) nibble of this byte enables (TRUE) or disables (FALSE) the Pull-Down.

Return Codes AR_ICC_OK Successful operation.

ICC_Configure()

Purpose Get / Set the configurable protocol parameters of the specified Smart Card interface. This function was developed to support different conformance tests. This API should be called prior to calling ICC_Enable. It is the same as ICC_Configure_Ext, above, with default values for IccHz (ICC_3600KHZ), DebouncePUEnable = SC_DEBOUNCEON | TRUE, and DebouncePDEnable = SC_DEBOUNCEOFF | FALSE.

Synopsis

```
AR_ICC_RC ICC_Configure (
    IN enum ICC_ID eIccId,
    IN enum ICC_ADDR IccAddr,
    IN enum ICC_CARDEVENT IccCE,
    IN BOOLEAN bIccSetOperation,
    INOUT ICC_Configure_t *ptrConfigure );
Struct ICC_Configure_t
{
    INOUT unsigned char ucIccIFSD;
    INOUT unsigned char ucIccNAD;
    //this variable when initialize to 0xFF the GetResponse command will carry the
    CLA //byte of the last C-APDU. UcIccCLA = 0x00 for EMV, = 0xFF for non-EMV
    INOUT unsigned char ucIccCLA;
    INOUT unsigned char ucIccTSTimeOut;
    INOUT unsigned char ucIccRxErrorCounterT0;
    INOUT unsigned char ucIccTxErrorCounterT0;
    INOUT unsigned char ucIccTxErrorCounterT1;
    INOUT unsigned char ucIccConfigurationByte;
};
```

Parameters

eIccId: Input parameter
Specifies which Smart Card interface is to be activated. Possible values are:
ICC_1ST 0, (Internal)
ICC_2ND 1, (External)
…
ICC_9TH 8 (External)

IccAddr: Input parameter
Specifies the address for the external I2C slot. Possible values are:
ICC_I2C0 0x40, (1st external slot)
ICC_I2C1 0x42,
ICC_I2C2 0x44,
…
ICC_I2C7 0x4E; (Last external slot)

IccCE: Input parameter
Specifies the assignment of the INT2/INT3 pins for card events. Possible values are:
ICC_INT2_NONE 0x00,
ICC_INT2_I2C 0x01,
ICC_INT3_I2C 0x02;

bIccSetOperation: Input parameter
Specifies if the function is called to perform a Set operation (TRUE) or a Get operation (FALSE).

pucIccIFSD: Input/output parameter
Specifies the IFSD value to be used (or being used). [Default value is 32 as specified by ISO/IEC 7816-3]

pucIccNAD: Input/output parameter
Specifies the NAD value to be used (or being used). [Default value is 00]

pucIccCLA: Input/output parameter
Specifies the CLA value to be used (or being used) when performing a GetResponse in case 2 / 4 (in T=0 protocol). Setting *pucIccCLA* to 0xFF indicates that the GetResponse command will echo the class byte of the APDU command. [Default value is 00]

punIccTSTimeOut: Input/output parameter
Specifies the maximum delay in clock cycles between the de-assertion of the RST signal and the leading edge of the TS character of the ATR. [Default value is 40 000]

pucIccRxErrorCounterT0: Input/output parameter
Specifies the maximum number of errors allowed when the Interface Device is in reception mode (in T=0 protocol).

pucIccTxErrorCounterT0: Input/output parameter
Specifies the maximum number of errors allowed when the Interface Device is in transmission mode (in T=0 protocol).

pucIccTxErrorCounterT1: Input/output parameter
Specifies the maximum number of errors in the T=1 protocol. The most significant nibble gives the maximum number of R(NA) block transmissions, while the least significant nibble gives the maximum number of I or S-block retransmissions.

ucIccConfigurationByte: Input/output parameter
This byte contains the following configuration bits:

[b0] *bIccIFSDRequestT1*: Input/output. Specifies if the next I-Block will be preceded by an S(IFS request) (TRUE) or not (FALSE).

[b1] *bIccDeactivatedOnTimeOutErrorT1*: Input/output. Specifies if the Smart Card interface is to be de-activated (TRUE) on a TimeOut error. Otherwise, an error recovery procedure is engaged.

[b2] *bIccNADErrorCheckingT1*: Input/output. Specifies whether the NAD errors are to be checked (TRUE) or not (FALSE).

[b3] *bIccABORTManagementT1*: Input/output. Specifies whether the ABORT Request is to be managed (TRUE) or if the contacts are to be de-activated on S(ABORT Request) reception (FALSE).

[b4] *bIccRESYNCHManagementT1*: Input/output. Specifies whether an S(RESYNCH Request) command is to be sent when too many errors occur (TRUE) or if a de-activation sequence is to be initiated (FALSE).

[b5] *bIccRetransmitLastBlockAfterRESYNCHT1*: Input/output. Specifies whether the last block in T=1 protocol is to be retransmitted after a resynchronization occurs or the whole command. [Useful for Microsoft IFDTESTs suite]

[b6] *bIccWarningStatusBytesManagementT0*: Input/output. Specifies if the IFD must inform the application level of whether the status bytes have been received just after the command header transmission or after the command data transmission (in T=0 protocol). [Useful for EMV Test suites]

[b7] *bIccDeactivateOnIFSDNegotiationError*: Input/output. Specifies if the Smart Card interface is to be de-activated on an IFSD negotiation error.

Return Codes AR_ICC_OK Successful operation.

ICC_Disable()

Purpose Deactivate the Smart Card interface, slot number is specified by *eIccId*.

Synopsis **AR_ICC_RC ICC_Disable (** IN enum ICC_ID eIccId **);**

Parameters *eIccId*: Input parameter
Specifies which Smart Card interface is to be activated. Possible values are:
ICC_1ST 0, (Internal)
ICC_2ND 1, (External)
…
ICC_9TH 8 (External)

Return Codes		
	AR_ICC_OK	Successful operation.
	AR_ICC_ERR_BAD_PARAM	Invalid ICC Slot.

ICC_CheckPresence()

Purpose Return the status of the specified ICC interface.

Synopsis **AR_ICC_RC ICC_CheckPresence (** IN enum ICC_ID eIccId **);**

Parameters *eIccId*: Input parameter.
The lower (least significant) 4-bits specify which Smart Card interface to activate. Possible values are:
ICC_1ST 0, (Internal)
ICC_2ND 1, (External)
…
ICC_9TH 8 (External).
The higher (most significant) 4-bits specify whether the card detect polarity is high (CARD_DET_H) or low (CARD_DET_L). See additional details at the end of the description of this function.

Return Codes
AR_ICC_OK
Successful operation. The Smart card is present and activated.
AR_ICC_ERR_CARD_ABSENTNo
A Smart Card is present in this interface.
AR_ICC_ERR_CARD_DISCONNECTED
The Smart Card is present but not activated.

The *eIccId* parameter is split into two fields: Card Detect Polarity and Card Slot number, by using the most significant nibble and least significant nibble, respectively. Therefore, the most significant nibble of the *eIccId* parameter is used to determine if the Card Detect Polarity is to be configured High or Low. The constants CARD_DET_H and CARD_DET_L are defined in ICCMgt.h.

Examples:

1. If Card Detect Polarity is to be set to High when a card is inserted, it can be done so by calling: ICC_Enable(eIccId | CARD_DET_H, bIccATRAutoCheck,etc.).
2. If Card Detect Polarity is to be set to Low when a card is inserted, it can be done so by calling: ICC_Enable(eIccId | CARD_DET_L, bIccATRAutoCheck,etc.).
3. Calling ICC_Enable (eIccId, bIccATRAutoCheck, etc.) without OR'ing *eIccId* with CARD_DET_L or CARD_DET_H (in other words, eIccID <= 0x09) will default to CARD_DET_H.

4.4 Flash Programming

There are two ways to download a hex file to the 73S12xxF Flash:

1. Using a Teridian Flash Programmer Tool. This tool is packaged separately; contact a Teridian Sales Representative for more information.
2. Using a Signum Systems ADM51 ICE.

4.5 Test Tools and Certification/compliance Tests

A set of tools is provided with the 73S12xxF development kit to assist the application development. Teridian uses these tools to perform various certification and compliance tests such as WHQL (aka HCT), USB 2.0 certification, EMV Level 1 and ISO extended cases testing. These tools include the Smart ATR and CCID-USB Modules described below.

Smart ATR Test Tool

The Smart ATR tool runs on a PC under Windows 98/2000/XP. This tool is helpful when used in conjunction with the EMV Tool. It reads an ATR input by the user and translates each byte of the ATR per the *ISO 7816 Specification*.

CCID-USB Test Tool

This tool includes ccidtsc-*.sys,ccidtsc-*.inf and CCID-USB.exe. These modules use the USB communication interface to interface with a PC running Windows XP.

CCID-USB.exe is a Windows XP application used to test the PC/SC APIs as specified by the PCSC Workgroup and Microsoft. After the ccidusb.hex file is downloaded to Flash and ccidtsc-*.sys and ccidtsc-*.inf are loaded into a Windows XP Device Manager, any PC/SC application can be run on Windows XP to send commands to the 73S12xxFdevice. These tools are also used for HCT/DTM and USB command verifier testing. The following procedure describes the setup for this tool:

1. Program the Flash with ccidusb-*.hex.
2. Connect a USB cable between a PC running Windows XP and the 73S12xxF evaluation board. The Windows' 'Hardware found wizard' should pop up.
3. Follow the wizard procedure to install the .sys and .inf file on to Windows. A reboot is NOT necessary.
4. Insert a smart card into slot #1 of the evaluation board.
5. Run CCID-USB.exe to test a command going to the Smart Card.

   Another good test application is the Microsoft ifdtest.exe which is part of the HCT test suite. Run this program in manual mode (ifdtest –m) to observe the 73S12xxF communication to the smart card. The source code for both applications is included in the release.

The following embedded application source code is available, depending on the CD ROM included with your product:

- Ccidusb-*.hex: This application uses the USB communication interface and runs any PC/SC or CCID aware application to interface with the reader. Review the accompanying documentation and source code for usage and implementation details.
- tscPccid-*.hex: This application uses the Serial/RS232 interface and runs on any PC with a generic Serial COM driver. Review the accompanying documentation and source code for usage and implementation details.

4.5.1 EMV LEVEL I Certification Tests

The EMV compliant test suite follows the Payment System Environment specification. There are several test labs, listed on the www.emvco.com website, qualified to perform these tests.

Currently, there are two available Protocol test suites that can be used to qualify for EMV Level I compliance. Passing either of these suites will qualify as EMV Level I compliance. The Pseudo-CCID code links to the two TSC libraries LAPI and HAPI which comply with both tests. However, since each lab has its own test scripts and the test scripts differ according to the lab's setup, the application layer must be written and adapted specifically for each test lab's requirements. The following subsections describe the loopback tests that are to be written either on the host side or added to the TSC Pseudo-CCID firmware.

4.5.1.1 EMV Test Mode

An EMV test (or session) is defined as a Command/Response pair that runs from the Activation of the card to the Deactivation of the card. A Block Transfer may or may not occur during the session depending on the card's ATR response.

The host may set up the EMV PSE test environment via the USB CCID Card Control command (Escape command). The first parameter byte (B1) of the Escape command must specifically indicate whether a test mode is invoked and if so, it should be invoked using the MCI, VISA I or VISA II test environment. Review the Escape command section for details about this test mode.

Following a successful Escape command, the host should start the test by sending the PowerOn command (ScardConnect). Depending on the status of the ATR (good or bad), the host should send an empty Block Transfer command without the APDU command (command length = 0). For example: 6F 00 00 00 00… CRC.

The test loopback will be handled by the firmware and upon completion of each test, the firmware will respond to the host with the status, indicating whether the test session completed with a successful return code or not. An unsuccessful return code may or may not indicate that the test failed. The test result (test passed or test failed) is determined only by the card side.

Figure 11 depicts the minimal coding required on the host side to invoke the EMV PSE test environment.

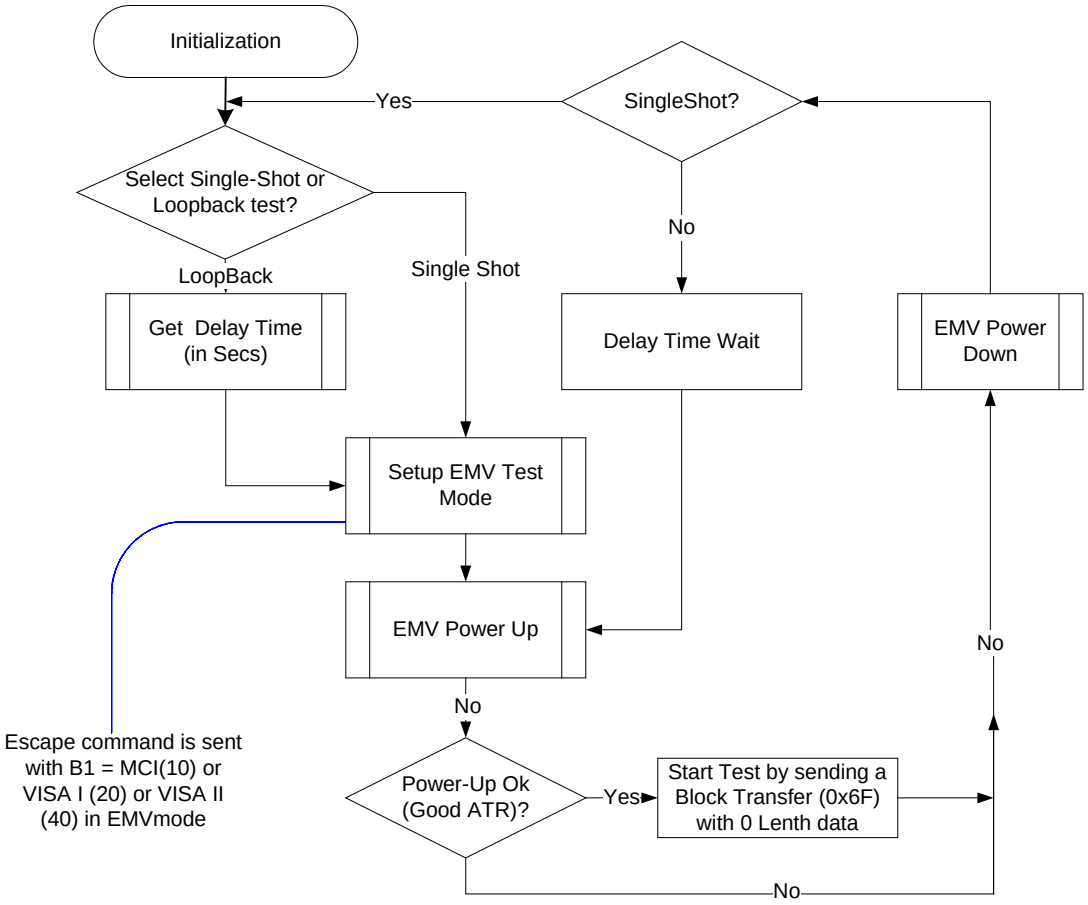

Figure 11: EMV PSE Test Flow Chart

4.5.1.2 MasterCard Loopback Test

Teridian used the CETECOM test lab in Germany and the FIME test lab in France (both listed on the www.emvco.com website) for MasterCard Loopback verification. These labs used the MCI test suite for their EMV Level I qualification test services.

Figure 12 and Figure 13 show the flow of the entire MCI test suite with the coding to be done on both the host side (invokes the test) and the device side (manages all aspect of the smart card's EMV test). These test flows are specific to both the FIME and the Cetecom's Level I Protocol test scripts. Source code is also included in the release.

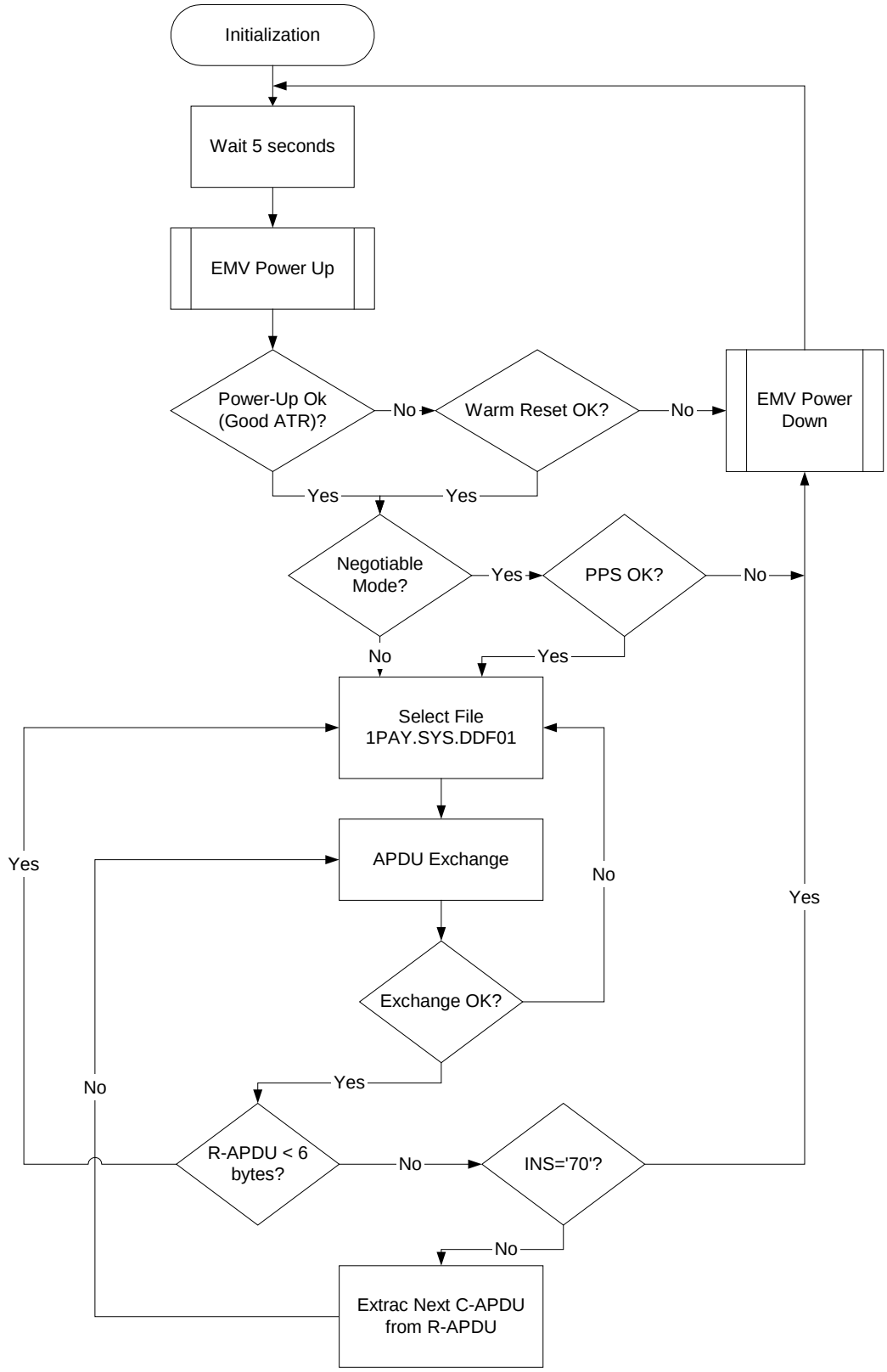

Figure 12: MCI Test Flow Chart with PTS/PPS

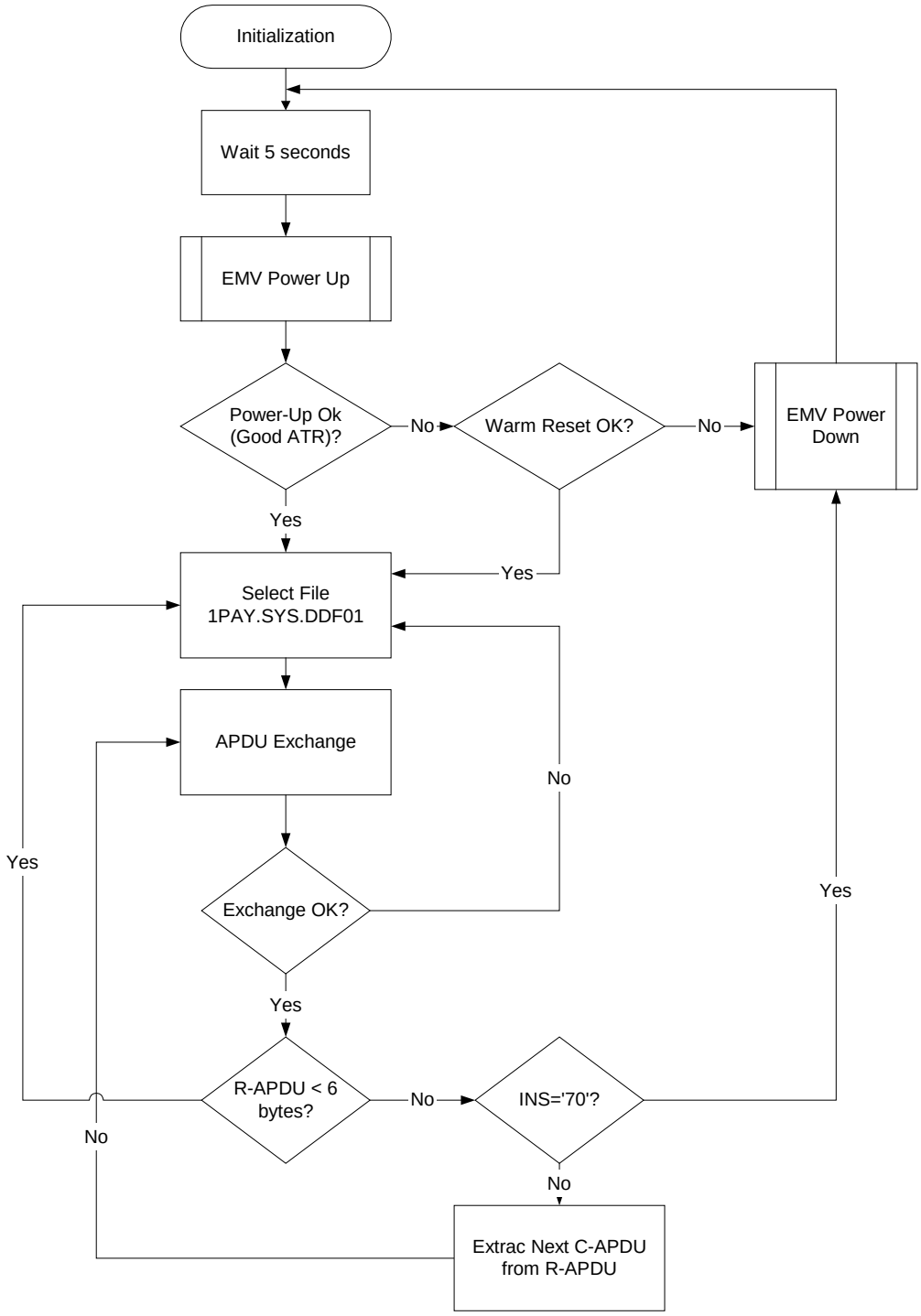

Figure 13: MCI Test Flow Chart without PTS/PPS

4.5.1.3 VISA-1 Loopback Test

Teridian used the RFI Global test lab in the UK (listed on the www.emvco.com website) for VISA-1 testing. This lab used the VISA test suite for their EMV Level I qualification test services. Figure 14 shows the VISA-1 test flow which is specific to RFI's test scripts. Source code for this test is also included in the release.

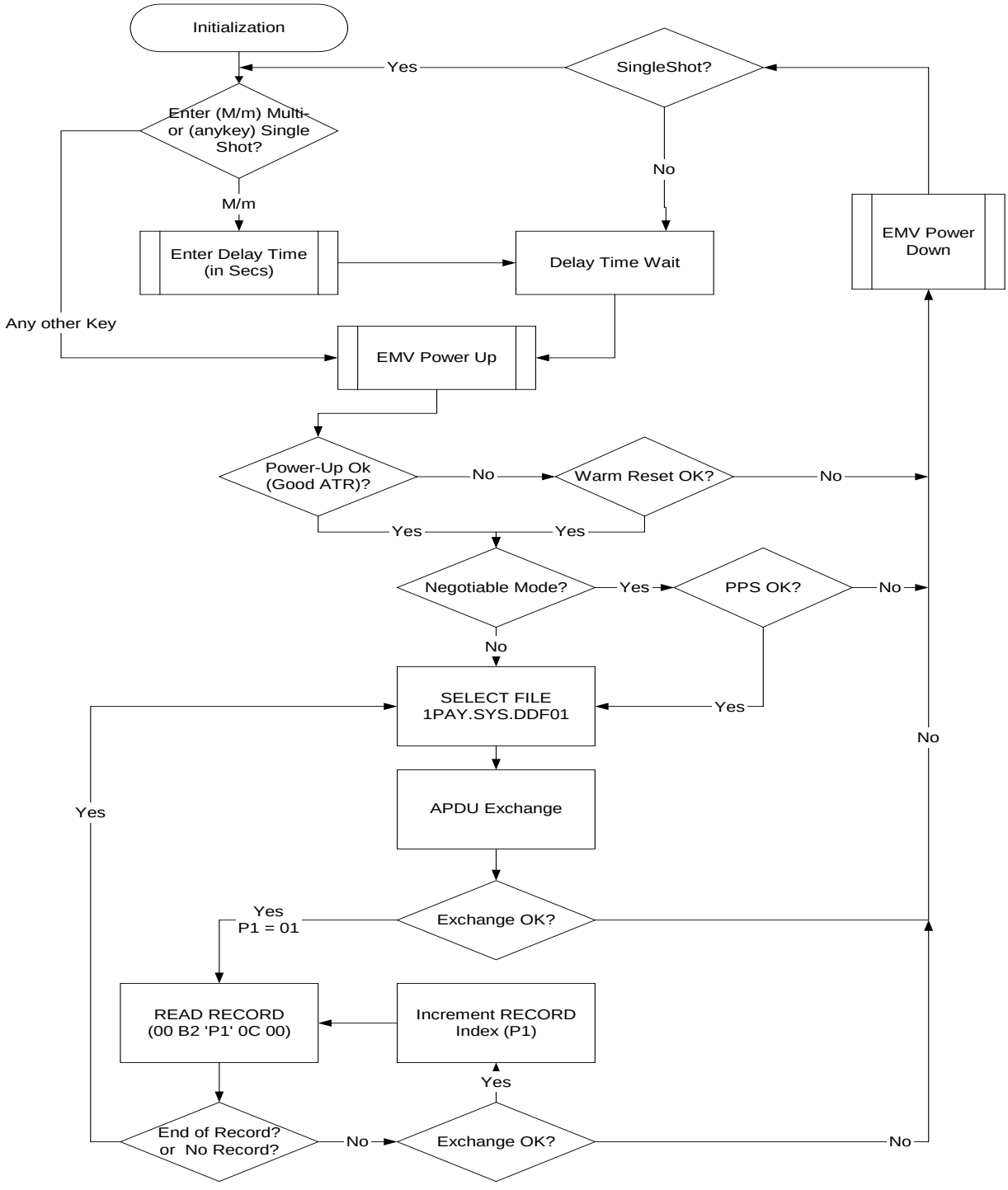

Figure 14: VISA-1 Loopback Test Flow Chart

4.5.2 VISA-2 Loopback Test

Teridian used the ICT-K test lab in Korea (listed on the www.emvco.com website) for VISA-2 loopback testing. This lab used the VISA test suite version 4.1 for their EMV Level I qualification test services (details shown in Figure 15). The USB CCID firmware includes the source code that implements this test.

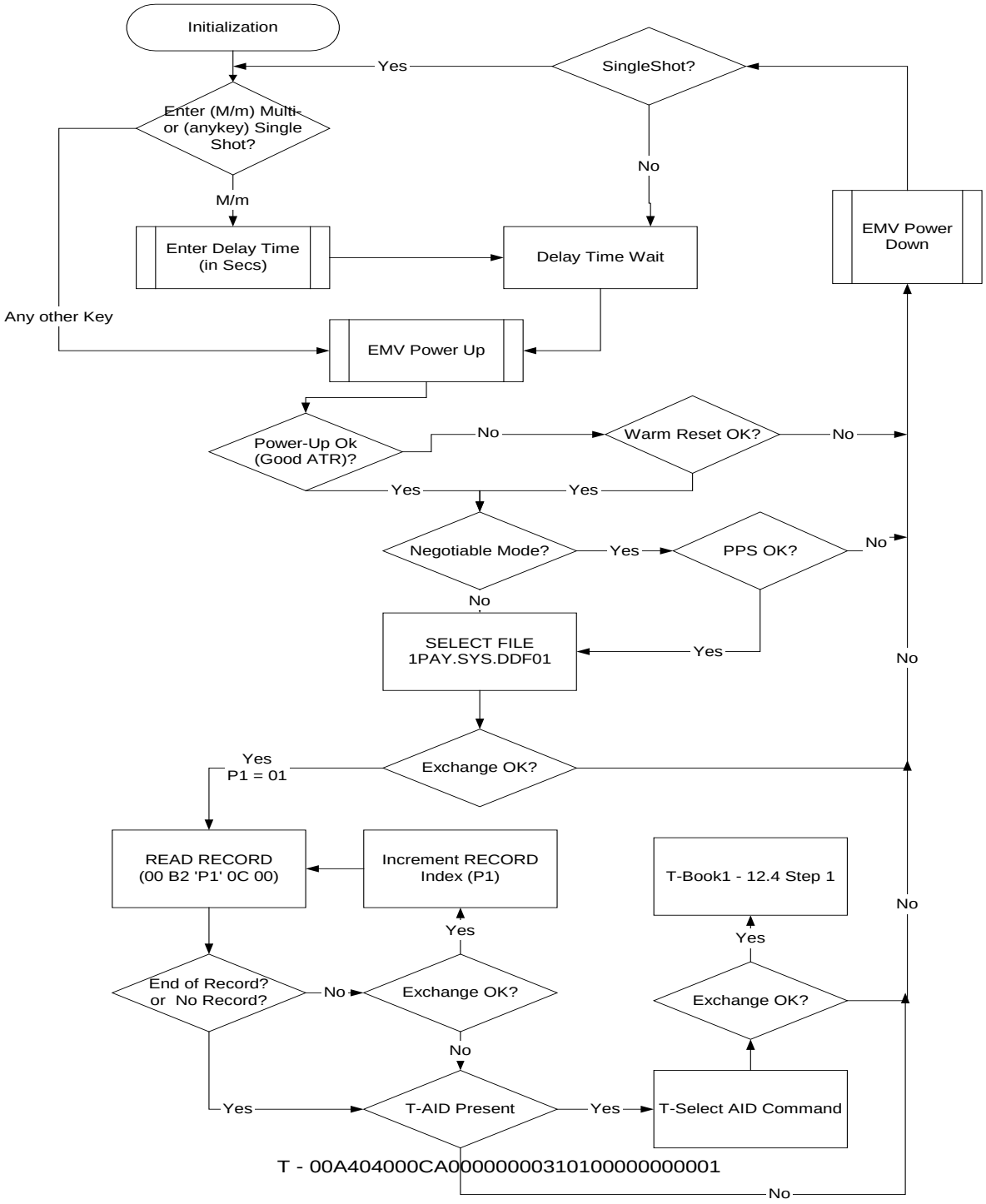

Figure 15: VISA-2 Loopback Test Flow Chart

5 Related Documentation

The following 73S12xxF documents are available from Teridian Semiconductor Corporation:

73S1209F Data Sheet
73S1210F Data Sheet
73S1215F Data Sheet
73S1217F Data Sheet
73S12xxF FPGA Evaluation Board User's Manual
Pseudo-CCID Host GUI Users Guide
Pseudo-CCID Host Application Guide
Pseudo-CCID Serial/RS232 Firmware Application Note
USB-CCID-Host GUI Users Guide
CCID Application Note

6 Contact Information

For more information about Teridian Semiconductor products or to check the availability of the 73S12xxF, contact us at:

6440 Oak Canyon Road
Suite 100
Irvine, CA 92618-5201

Telephone: (714) 508-8800
FAX: (714) 508-8878
Email: scr.support@teridian.com

For a complete list of worldwide sales offices, go to http://www.teridian.com.

Revision History

Date	Revision	Description
12/12/2005	0.01	Preliminary version
11/20/2006	0.80	Updated with the latest change since the last build. This version is still considered a Beta release.
3/1/2007	1.00	First production build. Includes modules that passed HCT/Microsoft WHQL, EMV Level I, USB.ORG's command verifier and goldtree testing.
3/19/2009	1.30	1. Updated to reflect the latest HAPI error return codes. 2. Added a new API (ICC_Configure_Ext) to the HAPI library to support: a. Setting Different Smart Card clock frequencies for both internal and external slots. Review this document carefully before implementing this feature. b. Enable/disable Pull-up with options to turn card debounce On/Off. c. Enable/disable Pull-down with options to turn card debounce On/Off.
4/27/2009	1.40	Added Linux driver for USB_CCID information and Linux Application for USB DFU Interface information. Added EMV Level 1 protocol layer information. Added Section 2.2.6, Build Environment with the USB Boot Loader.
9/14/2009	1.50	Modified the code sample on pages 44 to 48. Modified the code sample on pages 51 and 52.