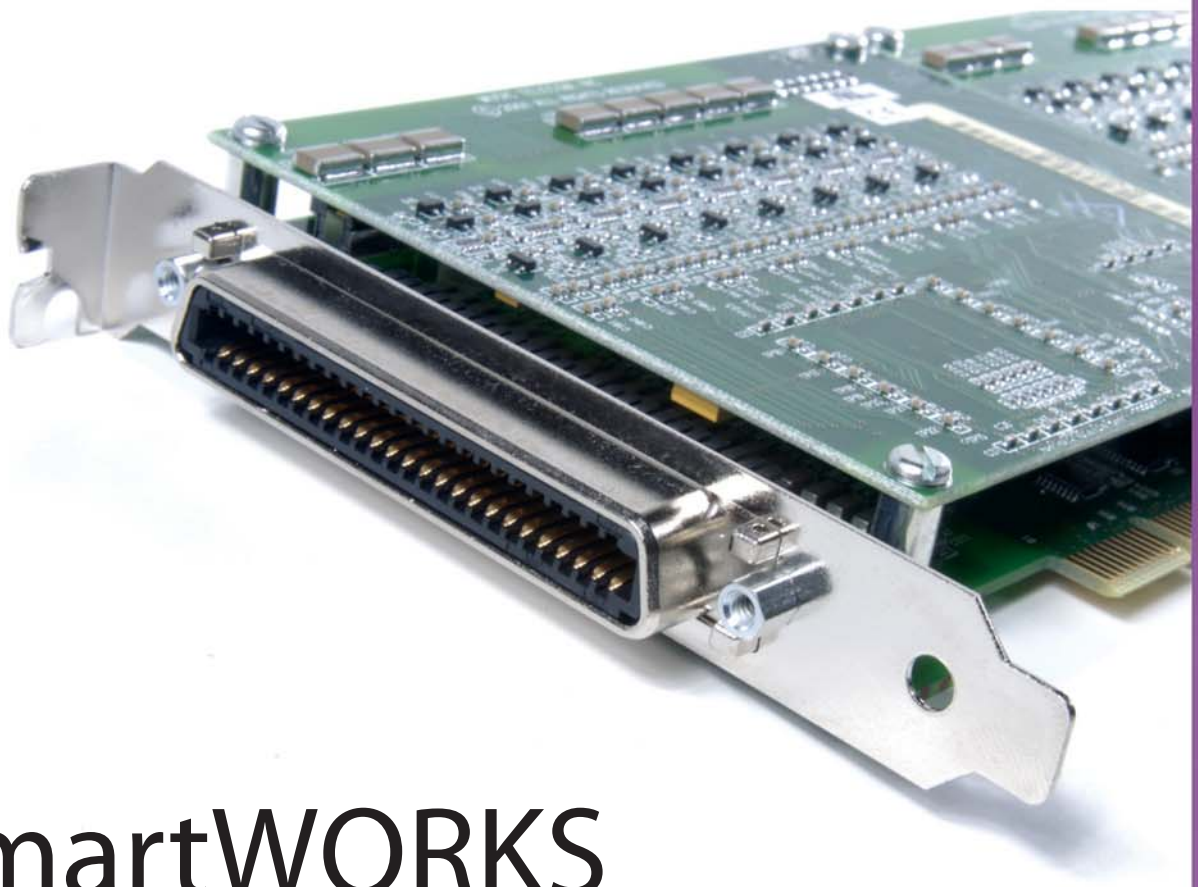
AudioCodes

BLADES BUSINESS LINE



SmartWORKS Developer's Guide

v. 5.2.0

AudioCodes USA
www.audiocodes.com/blades
27 World's Fair Drive, NJ · 08873
T: 732-469-0880 · F: 732-469-2298

404-0001-003 · Build 090828.01 REVB

Table Of Contents

Table of Contents

Table of Contents

Chapter 1

Welcome

Legal Notice

About This Documentation

This document applies to the following AudioCodes products:

NOTE: Lead free boards are referenced by weight.

Product Name	Part Number	Weight	Status
SmartWORKS VR3200	910-0303-001		Retired
SmartWORKS VR3209	910-0303-002		Maintenance
SmartWORKS VR6400	910-0301-001		Retired
SmartWORKS VR6409	910-0321-001		Maintenance
SmartWORKS AT409	910-0328-001		Retired
SmartWORKS AT809	910-0318-001		Retired
SmartWORKS AT1600	910-0309-001		Retired
SmartWORKS AT1609	910-0309-002		Retired
SmartWORKS DP3200	910-0308-001		Retired
SmartWORKS DP3209	910-0308-002	255 g	Released
SmartWORKS DP6400	910-0304-001		Retired
SmartWORKS DP6409	910-0324-001	280 g	Released
SmartWORKS DP3209-eh	910-0703-001	245 g	Released (special order only)

Product Name	Part Number	Weight	Status
SmartWORKS DP6409-eh	910-0703-002	270 g	Released (special order only)
SmartWORKS NGX800	910-0314-001	204g	Released
SmartWORKS NGX1600	910-0314-002	286 g	Released
SmartWORKS NGX2400	910-0314-003	366 g	Released
SmartWORKS MX80 (Expansion)	910-0315-001	68 g	Released
SmartWORKS MX80A (Expansion)	910-1315-001	60 g	Released
SmartWORKS NGX800-eh	910-0700-001	208 g	Released
SmartWORKS NGX1600-eh	910-0700-002	269 g	Released
SmartWORKS NGX2400-eh	910-0700-003	208 g	Released
SmartWORKS PT409	910-0307-002		Retired
SmartWORKS PT800	910-0305-001		Retired
SmartWORKS PT809	910-0320-001		Retired
SmartWORKS PT1600	910-0306-001		Retired
SmartWORKS PT1609	910-0319-001		Retired
SmartWORKS LD 101	910-0805-001	130 g	Retired
SmartWORKS LD 409	910-0801-001	165 g	Released
SmartWORKS LD 409H	910-0807-001		Released
SmartWORKS LD 809	910-0802-001	280 g	Released
SmartWORKS LD 809X	910-0808-001	385 g	Released
SmartWORKS LD 1609	910-0803-001	490 g	Released
SmartWORKS LD 2409	910-0804-001	605 g	Released
SmartWORKS LD 809-eh	910-0701-001	355 g	Released (special order only)
SmartWORKS LD 1609-eh	910-0701-002	460 g	Released (special order only)
SmartWORKS LD 2409-eh	910-0701-003	575 g	Released (special order only)
SmartWORKS DT3200	910-0312-001		Retired
SmartWORKS DT3209	910-0325-001		Maintenance
SmartWORKS DT6400	910-0313-001		Retired
SmartWORKS DT6409	910-0323-001		Maintenance
SmartWORKS DT6409TE	910-0323-002	275 g	Released

Product Name	Part Number	Weight	Status
SmartWORKS DT3209TE	910-0325-002		Released
SmartWORKS DT3209TE-eh	910-0704-001	240 g	Released (special order only)
SmartWORKS DT6409TE-eh	910-0704-002	265 g	Released (special order only)
SmartWORKS PCM 3209	910-0330-001		Released (special order only)
SmartWORKS PCM 6409	910-0329-001	270 g	Released (special order only)
SmartWORKS PCM 3209-eh	910-0702-001	265 g	Released (special order only)
SmartWORKS PCM 6409-eh	910-0702-002	240 g	Released (special order only)
SmartWORKS IPX	901-0331-001		Retired
SmartWORKS IPX-C	910-0331-007	250 g	Released

Value not available at time of document publication.

NOTE: Retired boards can no longer be purchased, but are still supported by the SmartWORKS software.

This documentation is intended for the developer of CTI application software. This manual assumes the reader is fairly proficient in standard C++ programming, computer telephony and voice processing.

Release Update History

SmartWORKS 3.11

IPX PBX Integrations: NEAX 2400 and Nortel Call Control support, H,.323, NGX PBX Integrations: Mitel ICP 3300 and Harris 2020, New Features: Japanese Caller ID (LD only), TCP re-ordering (IPX), Link Status and Network statistics (IPX)

SmartWORKS 3.10

IPX PBX Integrations: NEAX 2400 Dchannel support, NGX PBX Integrations: Ascom Ascotel (Vox & D-Channel) support, Siemens AC WIN D-Channel. New Features: pciExpress boards, Caller ID per NTT Telephone Interface Service Edition 5, MAC address has been added as station information (IPX only).

SmartWORKS 3.9

IPX PBX Integrations: Siemens HiPath 4000, Alcatel OMNI PCX Enterprise 6.0, SIP. NGX PBX Integrations: Rockwell Spectrum, Tadiran Coral, Aastra. New Features: Enhanced IPX/IPX-C performance, required license with IPX-C, Energy Tagging, one codeset 5 information element when tapping ISDN, multi-point BRI support with the NGX.

SmartWORKS 3.8

New board, IPX - C. Added DPNSS support on the SmartWORKS DP. Improvements to MSI install. PBX Integrations - NGX: New Panasonic Multi-point phone support, Alcatel OmniPCX beta DChannel support , Siemens Realitis iSDT(2W) Beta Vox support, Siemens Realitis DTI(4W) Beta Vox support. New IPX Integrations: Beta SIP support.

SmartWORKS 3.7

New PBX integrations with the SmartWORKS IPX: Avaya (Call Control), Ericsson (Call Control), Nortel (DChannel). New APIs used with the IPX: ***MTIpDChannelEventFilteringControl()*** and ***MTIpDChannelEventFilteringStatus()***. The functionality of the SmartWORKS IPX has been improved, refer to the release notes. Documentation changes: New book - *SmartWORKS Function Reference Library*.

SmartWORKS 3.6

Beta 2 support for the IPX board. New API, ***MTBoardGetCustomSwitchSetting()***. The following support has been added on the SmartWORKS NGX: Beta support of Intertel (Vox and Dchannel), Philips Sophos is supported with call control events similar to NGX BRI event reporting, Nortel Meridian; 3820 & 3310 series phones are now supported (Vox and Dchannel), Panasonic TDA 50; multi-point support has been added with KXT-7600 series phones, NEC NEAX2400; F revision line card support has been added with NEC I series phones, Astraa (EADS) Matra; MC420E series phones are now supported (Vox and Dchannel), eOn eQueue.

SmartWORKS 3.5

Added support for SmartWORKS PCM and LD809X boards. Beta support for new PBXs: Panasonic (Vox and Dchannel), EADS 4-wire support, and a new line card is supported with the Ericsson MD110.

SmartWORKS 3.4

D-channel support has been added for the following PBXs - LG Starex, Toshiba Strata Dk , Toshiba Strata CTX

SMARTWORKS 3.3

Enhanced CPM, improved buffer handling with the MT_EVENT structure, Voice/ Answering Machine detection, G.726 MSB first.

SMARTWORKS 3.2

Secondary input control for Activity detection and DTMF tone detection , Plug and Play, board clock synchronization, Flash firmware using APIs, and MSI support. Beta release of LD409H and the LD2409.

SMARTWORKS 3.0

Windows NT is no longer supported. Windows 2003 Server 32 Bit support has been added.

SMARTWORKS 2.10.0

SmartTERM DT6409TE, and DT 3209TE plus SmartTAP LD809 introduced to product line. Features added: multi-processor support, board identification, a signal profiling utility - SmartWORKS Profiler. See Release Notes for more information.

SMARTWORKS 2.9.0

Beta Release of LD 409. Support added for media format G.723.1. Support for terminate ISDN (SmartTerm DT). See Release Notes for more information.

SMARTWORKS 2.7.0

Minor changes. See release notes.

SMARTWORKS 2.6.0

Added D-Channel support for SmartTAP NGX

SMARTWORKS 2.4.2

2nd Beta release of SmartTAP NGX

SMARTWORKS 2.4.0

Beta release of SmartTAP NGX. G.729A support added (VR6409, PT1609).

SMARTWORKS 2.3.7

Alpha and Early Adopter release of SmartTAP NGX

SMARTWORKS 2.3.6

Early Adopter release of NFAS functionality

SMARTWORKS 2.3.5

Beta release of Windows 2000 WDM driver

Early Adopter release of H.100 functionality

SMARTWORKS 2.3.4

Beta release of SmartTERM AT and DT

SMARTWORKS 2.3.2

Release of SmartTAP PT series

SMARTWORKS 2.3.0

Beta release of SmartTAP PT series

SMARTWORKS 2.2.0

Release of SmartTAP DP series

SMARTWORKS 2.0.0

Release of SmartDSP VR series

Document Version Control

The following has been added to this document since the last release:

TABLE 1: VERSION CONTROL

Page	Description
REV A	
95, 96	Added note that ***MTSetGain()*** should not be used when stereo recording.
REV B	
95, 96	Clarified use of ***MTSetGain()*** and ***SetAGC()*** after summation. If mixing disabled, then only primary input is affected.

Contacting AudioCodes USA

Your feedback is important to maintain and improve the quality of our products. Use the information below to request technical assistance, make general inquiries, or to provide comments.

TECHNICAL SUPPORT

For programming, installation, or configuration assistance, use the following contact methods:

- Call technical support at 732.469.0880 or call toll free in the USA at 800.648.3647.
- For technical support log onto our online help system. Be sure to include a detailed description of the problem along with PC configuration, AudioCodes hardware, driver versions, firmware versions, a sample program that demonstrates the issue, and any other pertinent information.

To request an online help account please contact technical support at blade-support@audiocodes.com.

Sales and General Information

For sales and general information, use the following contact methods:

- Call us at 732.469.0880 or toll free from the USA at 800.648.3647.
- Fax us at 732.469.2298.
- E-mail us at bladesinfo@audiocodes.com.
- Visit our web site at www.audiocodes.com/blades.

MAILING ADDRESS—USA

Ship packages or send certified mail to us at the following address:

AudioCodes USA, Inc.

27 World's Fair Drive

Somerset, NJ 08873

Chapter 2

SmartWORKS Overview

SDK Overview

The SmartWORKS SDK is comprised of many files. These files, along with pertinent product documentation, are installed when you use the SmartWORKS universal installer CD that shipped with your product. Installation instructions are provided in the SmartWORKS User's Guide.

System Requirements

Intel Pentium IV platform or equivalent 2 GHz running

Microsoft® Windows® 2000, (Service Pack 3 is required)
Microsoft® Windows® XP, (Service Pack 1 is required)
Microsoft® Windows® Server 2003 32-bit
Microsoft® Windows® Server 2008 32-bit
RedHat Enterprise Server 4.0 AS
RedHat Enterprise Server 4.0 ES
RedHat Enterprise Server 4.0 WS
Suse Enterprise Server 10 - x86
Suse Enterprise Desktop 10 -x86

For detailed system and hardware requirements on a per product basis, please refer to the SmartWORKS User's Guide.

Developer's Notes

Microsoft IDE debug mode

The following applies to the SmartWORKS SDK version 2.0.0 and older:

The Microsoft IDE does not detach applications from associated dynamic linked libraries when *Stop Debugging* (Shift+F5) is requested. The termination of the SmartWORKS DLL leaves the interface between the DLL and driver in unclosed states. If the encode or decode interface is active at such a time, Windows crashes because the hardware interrupt is not serviced. This only happens in Microsoft IDE debug mode.

Preventing Errors During Shutdown

When using SmartWorks version 3.2.3 or greater inside of a Windows service, the following steps must be followed to prevent a system hang on shut down.

1. Upon initialization, the SERVICE_ACCEPT_SHUTDOWN flag must be specified in order to tell the operating system to send the SERVICE_CONTROL_SHUTDOWN notification. This is done during initialization of the service by setting the SERVICE_ACCEPT_SHUTDOWN bit of the SERVICE_STATUS.dwControlsAccepted field. This would be done in the CServiceModule::Init function of a Visual Studio 6.0 AppWizard based application:

```
m_status.dwControlsAccepted = SERVICE_ACCEPT_STOP | SERVICE_ACCEPT_SHUTDOWN;
```

2. The SERVICE_CONTROL_SHUTDOWN notification should be handled by setting a SERVICE_STOP_PENDING status. This is to allow time for the SmartWorks DLL to be properly shutdown. The default time of 20 seconds allocated to a service to shutdown by the OS should be enough time for an ***MTSysShutdown()*** call to complete. However, this time interval can be increased using the HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\WaitToKillService-

Timeout registry setting, or using the SERVICE_STATUS.dwWaitHint field when setting the SERVICE_STOP_PENDING status.

3. Perform any and all cleaning up of the SmartWORKS DLL (***MTSysShutdown()***), immediately after the CServiceModule::Run() function call in the ServiceMain routine.

Ensuring the SDK is Operational

Checking the Driver Status

When the SmartWORKS driver is invoked, it searches for SmartWORKS cards. When the card is successfully initialized, the CR17 LED is turned off and LEDs CR1 to CR16 are illuminated. (The CR number varies with each product. Refer to the User Guide for the exact LED number). With a SmartWORKS DP card, the two trunk status LEDs illuminate immediately after a successful board initialization.

When a SmartWORKS board fails initialization, the CR17 LED blinks, but the SmartWORKS driver is loaded nonetheless. An Error message about the failure is listed in the Windows Event Viewer under the Source name *NtiDrv*. These messages will help AudioCodes technical support to locate the problem. **NOTE:** When using Linux, all information is written to a 'messages' file located in the /var/log directory.

As each board is unique, the use of LEDs varies per SmartWORKS product. See the User Guide guide for LED locations on all SmartWORKS products.

Running the SmartView Demo Application

Running the SmartView demo program can help determine if the SmartWORKS Driver/DLL has been properly loaded. To do this view the total number of channels displayed in the SmartView application. If no channels appear, it is likely that the SmartWORKS card is either not present, not supported, not initialized, or the driver is not loaded.

SmartControl Status

Successful initialization of a SmartWORKS board can also be determined through SmartControl. When a board fails to initialize the *Board* tab will be blank.

Troubleshooting during Installation

Trouble shooting information is provided on a per product basis in the SmartWORKS User's Guide.

SDK Contents

When the SmartWORKS SDK is loaded onto a system, various applications and files are copied onto the PC. This section explains what is installed when SmartWORKS is installed.

Applications

The following applications are installed when the SmartWORKS software is installed:

SmartView.exe

SmartView is a demonstration program that comes as part of AudioCodes' SmartWORKS SDK. SmartView is capable of exercising most functions in the SmartWORKS SDK and serves as a quick test tool for SmartWORKS API functions and board status. SmartView can also aid during the installation and configuration of any system in the field by providing a quick and simple test of basic functionality.

SmartControl.exe

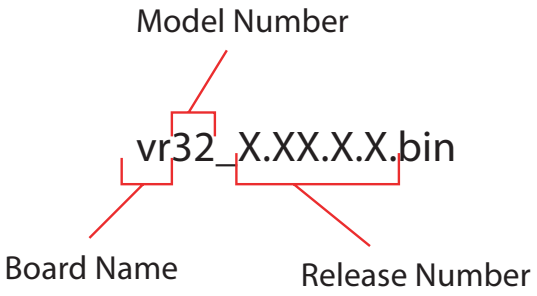
SmartControl is a Windows Control Panel utility designed to retrieve and configure the SmartWORKS operating environment. SmartControl works on system and board level configurations. Channel-related configuration is offered through API function calls. Tones, CPM tones, etc. are available through the SmartControl utility.

New configurations will not take effect until the driver is reloaded. It is recommended that you always reload the driver to ensure that any changes become effective.

SmartWF.exe

All SmartWORKS products (except the SmartWORKS NGX) use on-board flash memory to store the DSP and CPU firmware. AudioCodes will release new versions of this firmware from time to time and the user may want to update their boards. The SmartWF utility is used to retrieve or update the flash image (.bin) of SmartWORKS boards.

When using the SmartWF utility, you will see a complete list of all firmware options. When using this tool you must select the correct firmware for your hardware. The following naming convention is used to identify which firmware is required for each board type:



For information on upgrading firmware, see the *SmartWORKS Utilities Guide*.

FOLDERS AND FILES

The following files, organized by folder, are copied onto the PC when SmartWORKS is installed.

Documentation

Release Notes

General notes about the release installed.

SmartWORKS User's Guide

The universal user's guide for the SmartWORKS family of products.

SmartWORKS Developer's Guide

The developer's guide for the SmartWORKS SDK. This manual provides an overview of the SmartWORKS SDK, theories of operation for the SmartWORKS product family, and implementation instructions for SmartWORKS features.

SmartWORKS Function Reference Library

This document lists all function prototypes including all data structures, plus each event code is documented.

SmartWORKS Utilities Guide

A user guide explaining each of the SmartWORKS utilities: SmartWF, SmartView, SmartProfiler and the SmartControl panel.

NGX Integration Guide

Provides useful information about integrating the SmartWORKS NGX board with proprietary PBXs in environments where D-channel is supported.

IPX Integration Guide

Explains VoIP recording and provides useful information about integrating the IPX board with proprietary IP PBXs.

Product Quick Installs

Each product in the SmartWORKS family has its own Quick Install guide in PDF format that walks you through the basic installation of the product.

Firmware

All board firmware files are stored here. Firmware files are also required for PBX integration (SmartWORKS NGX board only). Refer to NGX installation instructions for more information.

Inc

The following files are present in the Inc folder:

Programming Interface

These files comprise the primary interface to the SmartWORKS API.

NtiApi.h contains the prototypes for entry points in NtiDrv.dll, which is the interface to the device driver.

NtiWFAPI.h declares the APIs used to flash firmware

NtiData.h and NtiEvent.h contain all data structures and event definitions. Both the NtiData.h and the NtiEvent.h also have files specific to Call Control and D-Channel support. For example: NtiDataCC.h and NtiDataDCC.h.

NtiErr.h contains all the possible return codes.

Legends.h contains the declaration for the API interface. User applications should include either windows.h or afxmt.h from the Microsoft Compiler before including NtiAPI.h.

Miscellaneous Files

NtiAPINull.h lists APIs that are not supported by the current SmartWORKS API.

Obsolete.h lists APIs that have been obsoleted and are no longer supported.

ToBeDeleted... shows the APIs that are scheduled for deletion. Developers should update their applications before these APIs are no longer supported.

SysLimits.h - defines SmartWORKs system limits

NV_WRAP.h - declaration for backward compatibility for NVDSP definitions

MS_Types.h - maps data types from Windows to Linux

NTI_NV.h - declares data structures that are not declared in the NtiData.h file, required for backward's compatability. Header file NTI_NV.h contains declarations that are also present in the NV.h header for the AudioCodes NVDSP SDK. When planning to use both the NVDSP and SmartWORKS SDKs, include NV.h first, define NV_CO_RESIDE macro, and then include NtiAPI.h header.

MS_Wave.h - defines the Microsoft WAVE format, used only in Linux platform

Lib

Includes offset addresses for each function name that the DLL exports.

Samples

Sample code is provided to assist Developers integrate applications with SmartWORKS boards.

DRIVERS

NtiWdmDrv.sys- the low-level interface between the application DLL and the SmartWORKS hardware. The device driver provides the basic functionality for managing all high-speed data I/O transfers with on-board resources.

The SmartWORKS driver must be installed before the user application can start. When running Windows 2000, 2003, 2008 or XP, the drivers start automatically with system startup.

NtiDrv.dll - A Dynamic Link Library (DLL) is provided as the primary interface between the customer's application and the AudioCodes driver. All APIs are declared as dll export and use the __stdcall calling convention.

ENSURING USER APPLICATION COMPATIBILITY WITH THE SDK

The SmartWORKS API provides the capability of checking the version of the SDK that the user application is compiled with and comparing it to the version of the AudioCodes SDK that is currently running. These values are returned via a SmartWORKS API. This can be accomplished at run time to protect the user application from running on an older version of the SmartWORKS SDK.

In SDK header files, there is version information defined in file "SysLimits.h" as VER_MAJOR, VER_MINOR, and VER_INTERNAL. The user application should be written to hold these values.

Then, at run time, the application can call for the SDK values on the SmartWORKS boards. (Use API ***MTSysGetVersion()***). If the returned values do not match the values stored in the application the end user can be alerted of incompatible software.

Plug and Play (PNP) Capabilities

When the application is running, a newly initialized board can be recognized. When this occurs, a system event is generated (EVT_SYS_BOARD_ADDED).

To use this feature, the user application must be collecting system events. Enable this with the APIs ***MTSysWaitforEvent()*** or ***MTSysSetEventCallback()***. Once the event is generated it is recommended that the user application invoke the ***MTGetSystemInfo()*** API to obtain board statistics. If multiple applications are running, all applications should invoke this API to obtain board statistics.

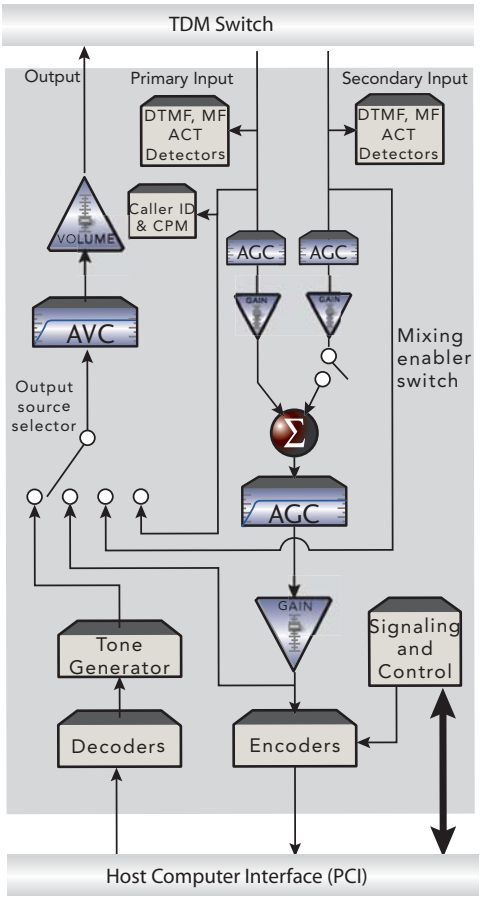
Architecture Overview

All SmartWORKS series products are built around a core set of powerful DSP algorithms. They also provide Industry standard codecs, echo cancellation, automatic gain control, DTMF, MF, CID and activity detectors.

THE SMARTWORKS CHANNEL MODEL

The figure below is a representative diagram of the SmartWORKS DSP channel architecture. This is a logical representation of the voice resource channel found on all SmartWORKS products. The diagram shows major functional blocks and how they are interconnected. **NOTE:** The IPX does not support the channel model.

SmartWORKS Channel Model



The following is a summary of the elements of the SmartWORKS Logical Channel Model. Please refer to the SmartWORKS User's Guide for a detailed explanation of these elements per each SmartWORKS board.

TDM BUS

Each logical channel uses a TDM switch as a means of communicating with other devices in the system. Each logical channel has three distinct connections to the TDM switch: two inputs and one output. SmartWORKS products use the primary input as the main source of voice data. The secondary input is used for call recording applications where access to both sides of the conversation is required (e.g. T1/E1 trunks).

DTMF, MF, Activity Detection

Each logical channel has two sets of DTMF, MF and activity detectors, one for each of the two voice data inputs. The primary input has an additional Caller ID detector and Call Progress Monitor (CPM). The CPM has pre-programmed profiles for typical call progress tones used in North America.

Input Mixer

Each logical channel has a two-way mixer that can be used to combine the Primary and Secondary voice data inputs. These inputs are very useful when recording digital trunks (T1 or E1) in which case the voice logger has to add (mix) both sides of the conversation. Activity detection and DTMF/MF tone detection is configurable on a per input basis.

AGC

Automatic Gain Control (AGC) optimizes voice data to facilitate a wide dynamic range typically encountered when a voice logger is connected close to a PBX or analog phone.

Gain

A fixed gain stage is provided to adjust the overall amplitude of the received voice data.

Volume

A fixed gain stage adjusts the overall amplitude of the transmit voice data.

AVC

Automatic Volume Control (AVC) adjusts playback levels of transmitted voice data, which makes the listening volume more comfortable. Recorded data volume levels are not affected.

Live Monitor

A special signal path is provided which allows a user to monitor a recording in real time. This feature allows incoming voice data to be routed to the TDM switch via the output side of the logical channel. Here AVC and gain can be applied without affecting the recorded signal.

Encoders/Decoders

The SmartWORKS product line offers a wide range of voice encoders and decoders. The selection of digitalization methods is user programmable on a per channel basis.

Host Interface

The host interface is used to move voice data to and from the logical channel and communicate control and event data. The host interface is a 33 MHz, PCI2.2 compliant bus.

Tone Generator

Each logical channel has a programmable tone generator that can be used to play tones to the TDM switch. User applications can generate standard DTMF tones of programmable amplitude and duration.

Board and Channel Numbering

Physical Board Numbering

When the SmartWORKS driver loads, it scans all PCI slots of the system to locate AudioCodes boards. As the boards are located, the driver assigns Physical Board Numbers to each. These board numbers are assigned sequentially from zero and are linked to the address (or slot number) of the physical PCI slot the card is located in.

An AudioCodes SmartWORKS board with the lowest address becomes Physical Board 0. The board located in the PCI slot with the next higher address will become Physical Board 1. This process is repeated until all PCI slots are scanned.

This PCI slot may or may not be the absolute lowest numbered PCI slot in the system; it is simply the lowest in relation to the other PCI slots with boards inserted into them. The PCI address or slot number is generally noted on the system motherboard or passive back plane.

Adding a New Board to the System

When a new SmartWORKS board is added to an existing system, board numbering MAY be impacted. This depends on the actual physical location of the new board and the PC's BIOS. Users are encouraged to monitor board and channel numbering each time a system is restarted.

Controlling Board Numbering

An API is available where users may control board numbering. Invoke ***MTSetAdapterConfig()*** to control board presentation in the *PresentationPreference* field.

Board and Channel Numbering

The SmartWORKS API supports up to 32 physical boards and/or up to 512 full duplex channels within a system. The API functions refer to a specific board and or channel within the system using one of two numbering schemes: physical board numbers, and Global Channel Index (logical channel numbers). All board numbers are assigned sequentially starting from zero. Channel numbers are assigned sequentially starting from either 0 or 1 (depending on how the user has configured this setting in the Smart Control panel).

Certain API functions will allow the developer to reference all boards simultaneously by using the nBoard = -1.

Global Channel Index

During initialization, as the Physical Boards are numbered, the SmartWORKS software builds a list of the logical channels available in the system. This list is the primary interface the API will use to refer to the channel resources in the system.

The Global Channel Index (GCI) specifies whether the channel list is numbered sequentially from 0 or 1 (depending on how the user has configured this setting in the Smart Control panel). Channel numbers are presented in ascending order of the Physical Board numbers. The maximum number of channels supported by SmartWORKS is 512.

Certain API functions will allow the developer to reference all channels simultaneously by using the nChannel = -1 (if GCI index = 0) or nChannel = 0 (if the GCI index = 1).

For Example:

Function ***MTSetEventCallback()*** takes channel number -1 or 0, and registers the callback function for all available channels.

The API has several commands that can be used to determine the relationship between the GCI and the physical channels on each board. ***The MTGetGCI()*** and ***MTGetGCIMap()*** command will match a GCI indexed channel to its physical board channel location.

For Example:

If GCI index = 0 and MTGetGCIMap(08, pBOARD, pBOARDTYPE, pGCI) returns with *pBOARD=1, and *pGCI=0, this indicates GCI channel 08 resides on board 1 as its first channel. However, MTGetGCI(1,0,pGCI) should return with *pGCI=08.

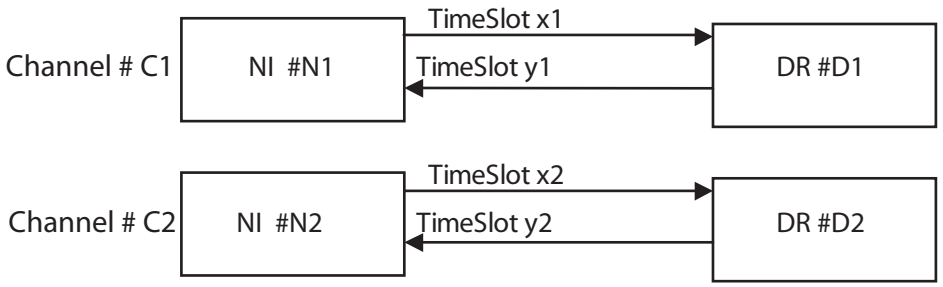
Definition of NI, DR, and Channel

An *NI* can be either a digital network interface from T1/E1, or an analog network interface from an analog interface card. A *DR* is the DSP resource on a SmartWORKS card; SmartWORKS VR cards have either 32 or 64 DRs depending on the VR model. A *Channel* is a DR paired with an NI, or a DR only, i.e. a SmartWORKS VR6400 has 64 channels. **NOTE:** The SmartWORKS IPX does not open with channels.

Channel Mapping

In SmartWORKS, a channel is a DR, preferably mapped with an NI if one exists. Hence, a channel is indexed through its DR indexing. A channel mapping can also be termed as adding NI to a channel, i.e. DR.

The relationship between one NI and one DR through the global timeslot is referred to as Channel Mapping. In order to connect an NI with a DR, two global timeslots are required: one for NI to transmit to and for DR to receive from, and one for DR to transmit to and for NI to receive from.



Though SmartWORKS supports dynamic channel mapping, the flexibility of mapping between NI and DR should include the following rules:

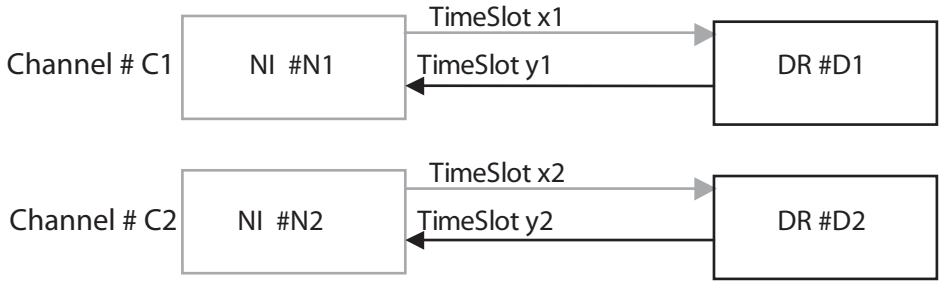
1)For NI from E1 with R2 signaling, the DR has to be on the same SmartWORKS card;

2)For NI from T1 with robbed-bit signaling, the DR has to be on the same SmartWORKS card;

3)For NI from ISDN network, the DR can be on any SmartWORKS card controlled by the same DLL/Driver;

4)For NI from an analog network (e.g. analog interface card without any DSP resources), the DR can only be assigned from a SmartWORKS VR card by the user of the SmartWORKS API.

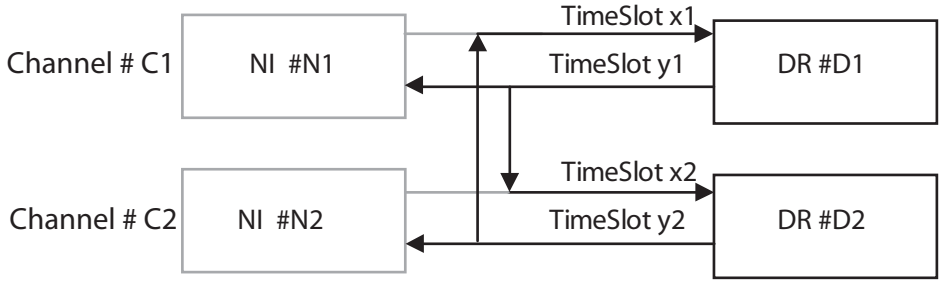
When a DR is not mapped with an NI, one local bus time slot is still allocated for this DR. This is shown below.



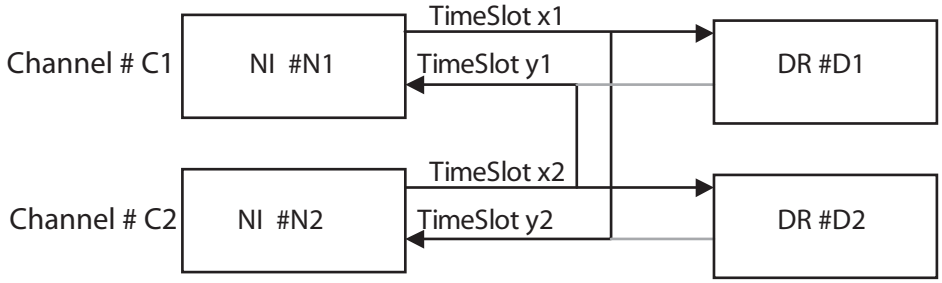
Channel Connection

By the same token, the connection between two channels is illustrated in this section.

First, the connection between two DR-only channels is shown.



Secondly, the connection between two NI-DR-paired channels. Note that the NI paired DR will not have a transmit time slot assigned.



CT Bus TimeSlot Allocation

Although the SmartWORKS DLL/Driver may not have the knowledge of the system view of the CT Bus time slot assignment, an effort is added to check that SmartWORKS does not connect two transmits to one time slot.

SmartWORKS offers functions where the application user can directly assign the CT Bus time slot to DR or NI. SmartWORKS VR also offers functions to connect channels without the application user being concerned with CT Bus time slot.

SmartWORKS allocates its assigned CT Bus time slots during the following situations:

1)Connecting channels across different SmartWORKS boards.

2)Connecting channels on the same SmartWORKS board if one of the channels has a CT Bus time slot assigned and one of the channels does not.

Default Mapping

SmartWORKS can be a resource or network card.

The SmartWORKS VR supports no NI interface and a channel contains DR only. With SmartWORKS DP card, each channel is made of pairing NI to DR to the minimum of these two. The number of NIs can be found through ***MTGetAdapterInfo()*** function. Each NI can be indexed through a combination of board identification and NI number. DRs are indexed through channel indexing.

Chapter 3

API Overview

API/DLL Structure

This section provides a high level view of the SmartWORKS API.

SmartWORKS Function Types

Immediate and Background Functions

SmartWORKS API functions are broken into two categories: *Immediate* and *Background*.

Immediate Functions

An *Immediate* API function is one that does not return until it is completed. This is also referred to as a synchronous function.

Background Functions

A *Background* API function is one that is queued and will be completed when the requested resource is available. Background functions are also referred to as asynchronous as they respond with a return code before they are completed.

NOTE - For a complete listing of SmartWORKS return codes, see "Return Codes" on page 49

There are several functions, such as playing a file or dialing a number, that do not complete immediately. The API does not wait for this type of function to finish, but returns immediately after the function is queued. The function will be executed when the channel resource is available, hence, running in the background and allowing the application to perform other tasks.

Only one background function per channel can be active at a time. Background functions can be queued any time. Background functions are stopped or flushed with the channel stop APIs: ***MTStopCurrentFunction()*** or ***MTStopChannel().***

Resource Queues

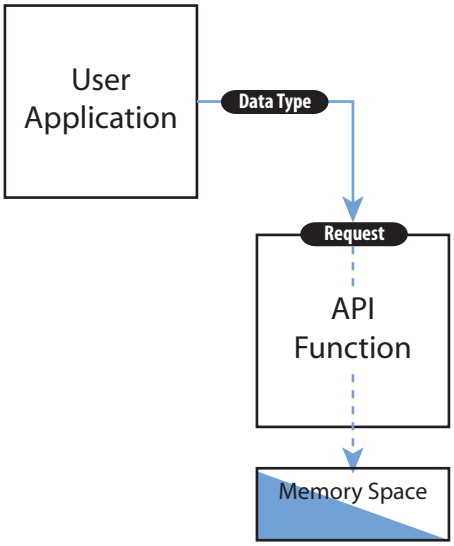
The SmartWORKS API supports three resource queues:

- Encode
- Decode
- DTMF collection queue

By definition, API functions that need access to the encode, decode, or DTMF queue are *background* functions.

Any time a user application requests a value from the API, the following sequence occurs:

1. The application passes a data type pointer to the function.
2. The API fills the referenced memory space with the value.



All API functions return a *long integer* describing the function's execution status. It is strongly suggested that the return value be checked to ensure successful operation. The requested value can be guaranteed as valid only when the function is executed successfully.

Parameters

The following section highlights the behavior of the API.

Changing Settings

There are functions that change system and channel parameters on a per application basis. These changes remain in effect until they are modified again by the next call to that function. These functions usually require a pointer to a structure with several parameters. Zero is a valid parameter value, and represents the default.

Preserve Data Buffer

For some functions like ***MTRecBuffer()*** or ***MTGetDigits()***, the application needs to pass the buffer pointer to the API and wait for the API to fill in the data buffer. Because of this, the user application must preserve the data buffer until the termination event is received. However, the user application *does not* need to preserve the MT_IO_CONTROL structure until the termination event is received. Once the API returns, the DLL maintains a copy of the MT_IO_CONTROL structure for the API being called.

Function Completion Notification

Function completion notification events are broken into two categories: *Overlapped events* and *asynchronous callbacks*.

Overlapped Events

Any API that ends with the suffix "Ex" uses the Windows overlapped mechanism as its function completion notification mechanism.

Take the play buffer function as an example. ***MTPlayBuffer()*** uses no notification mechanism. The user application retrieves events from the event queue until it encounters a playback specific event, which indicates completion of the ***MTPlayBuffer()*** function.

API function ***MTPlayBufferEx()*** uses the overlapped mechanism. In this case the user application provides an NT system event. DLL uses the signal when the **MTPlayBufferEx()** function has completed. The user application should wait for the signaling of the event and clear the event before it can be used again.

Asynchronous Callbacks

Any API that ends with the suffix Async uses asynchronous callback as its completion notification method.

API function ***MTPlayBufferAsync()*** requires a completion routine from the user application. The SmartWORKS function completion routine mechanism is implemented in the same fashion as a Windows system function completion. In addition to the completion routine address, the application passes a parameter of type LPARAM to the SmartWORKS DLL/Driver, which then passes the parameter back, without change, to the user application when the function has completed.

NULL Pointer Checking

NULL pointers are not allowed in the SmartWORKS API. An error code of invalid parameter (MT_RET_INVALID_PARAM) will be returned without executing the requested API function.

For Example:

Entering a value of 0 for a pointer will be taken as a NULL pointer. No default value will be assumed and the actual value you enter will be checked. Though previous AudioCodes products did allow NULL Pointers, the SmartWORKS SDK does not.

NOTE: For backwards compatability with older AudioCodes products, the LENGTH parameter for Set APIs allows the SDK to set up to the specified size. The rest of parameters are not changed. MT_RET_OK is returned.

Return Codes

All API functions respond with a return code indicating the status of the function's completion. All return codes are of type MT_RESULT.

For an *immediate* API function, the return code MT_RET_OK indicates that the API has been completed successfully.

For a *background* API function, the return code MT_RET_IO_PENDING indicates that the API task has been successfully queued for execution when a resource becomes available.

NOTE: All return codes are passed to the user application as a hex value. To obtain a text desciption use the API ***MTGetReturnCodeDescription()***.

EVENT CONTROL

Each channel has a FIFO buffer or event queue, which is used to temporarily store asynchronous messages from the board. These messages, or events, are used to indicate channel status information such as line signaling, voice activity, or errors. The event queue for each channel can store up to 64 messages. Regardless of which method is used, the user application is responsible for monitoring the channel status often enough to ensure that it does not overflow. If the event queue is full new events are lost and are reported in the Windows Event Viewer. **NOTE:** When using Linux, all information is written to a 'messages' file located in the /var/log directory.

There are two methods the application can use to retrieve a channel's events: *event queuing*, and *call backs*. The next chapter explains each method and provides a sample.

UNICODE Support

SmartWORKS provides UNICODE support under Microsoft Windows® in the following manner:

If UNICODE is defined in the user application, API functions that require a string parameter as input or output will use the PCWSTR definition for the string instead of the normal PCSTR definition. Following is a code segment from the ntiapi.h header file which shows how this functionality has been implemented.

```
#ifdef UNICODE
#define MTCallString        MTCallStringW
#else
#define MTCallString        MTCallStringA
#endif /* !UNICODE */
```

The definitions for the functions above are as follows:

```
MT_RESULT MTCallStringA(const CHANNEL          nChannel,
                        PCSTR                  pCall_string,
                        const PMT_IO_CONTROL   pIoCtl  );

MT_RESULT MTCallStringW(const CHANNEL          nChannel,
                        PCWSTR                 pCall_string,
                        const PMT_IO_CONTROL   pIoCtl);
```

The following table shows how SmartWORKS maps the APIs depending on whether UNICODE has been defined for the user application. If UNICODE is defined, SmartWORKS will call the API with a 'W' suffix. If UNICODE is not defined, SmartWORKS will call the API with an 'A' suffix.

TABLE 2: UNICODE/NON-UNICODE FUNCTION TABLE

Function	UNICODE not defined	UNICODE defined
MTCallString()	MTCallStringA()	MTCallStringW()
MTCallStringAsync ()	MTCallStringAsyncA()	MTCallStringAsyncW()
MTCallStringEx()	MTCallStringExA()	MTCallStringExW()
MTDialString()	MTDialStringA()	MTDialStringW()

TABLE 2: UNICODE/NON-UNICODE FUNCTION TABLE

Function	UNICODE not defined	UNICODE defined
MTDialStringAsync()	MTDialStringAsyncA()	MTDialStringAsyncW()
MTDialStringEx()	MTDialStringExA()	MTDialStringExW()
MTGetLastError()	MTGetLastErrorA()	MTGetLastErrorW()
MTRecFile()	MTRecFileA()	MTRecFileW()
MTRecFileEx()	MTRecFileExA()	MTRecFileExW()
MTRecFileAsync()	MTRecFileAsyncA()	MTRecFileAsyncW()
MTPlayFile()	MTPlayFileA()	MTPlayFileW()
MTPlayFileEx()	MTPlayFileExA()	MTPlayFileExW()
MTPlayFileAsync()	MTPlayFileAsyncA()	MTPlayFileAsyncW()
MTPlayIndex()	MTPlayIndexA()	MTPlayIndexW()
MTPlayIndexEx()	MTPlayIndexExA()	MTPlayIndexExW()
MTPlayIndexAsync()	MTPlayIndexAsyncA	MTPlayIndexAsyncW

Media Formats

SmartWORKS supports the following media formats:

SmartWORKS Compatible CODECs
µ-law 8-bit PCM 64 k bps
A-law 8-bit PCM 64 k bps
µ-law PCM+ (recording only)
A-law PCM+ (recording only)
Linear signed 8-bit PCM 64 k bps
Linear unsigned 8-bit PCM 64 k bps *
Linear signed 16-bit PCM 128 k bps *
Linear unsigned16-bit PCM 128 k bps
Linear signed, 6 Khz, 16-bit PCM, 96 k bps
GSM 6.10 13 k bps
Microsoft GSM 13 k bps *
Dialogic (Oki) ADPCM 24 k bps
Dialogic (Oki) ADPCM 32 k bps
G.723.1 5.3 kbps
G.723.1 6.3 kbps**
G.729A 8 kbps
G.726 ADPCM
G.726 ADPCM 16, 24, 32, 40 kbps MSB
µ-law 8-bit PCM 64 k bps, with energy tagging#
A-law 8-bit PCM 64 k bps, with energy tagging#
G.723.1 5.3 kbps, with energy tagging#

*Supports WAV headers
**Not supported on the NGX
+ Stereo Recording - Users must disable mixing on channel inputs
# Digital tapping only

Wave File Support

The SmartWORKS API supports Microsoft wave headers. Setting the appropriate flag in the MT_IO_CONTROL.StartControl data structure controls the inclusion of a header or data chunk descriptor in the media file.

`WAVE_RIFFFMTDATA_CHUNK`

add wave header of RIFF chunk, Fmt chunk, and Data chunk

`WAVE_DATACHUNK_ONLY`

add data chunk descriptor only

For example, when format MT_MSGSM (Microsoft GSM) is selected without either of the above flags, no wave chunk descriptor will be added in front of the first 65-byte GSM packet.

If flag WAVE_RIFFFMTDATA_CHUNK is set, a GSM wave header will be added before the first GSM voice packet.

If flag WAVE_DATACHUNK_ONLY is set, then an 8-byte long Data chunk descriptor will be added before the first GSM packet. Flag WAVE_RIFFFMTDATA_CHUNK implies flag WAVE_DATACHUNK_ONLY.

SmartWORKS currently supports WAVE 8-bit and16-bit LINEAR plus WAVE Microsoft GSM.

In functions for streaming and device IO, wave headers are not supported. MT_RET_INVALID_PARAM will be returned if either `WAVE_RIFFFMTDATA_CHUNK` or `WAVE_DATACHUNK_ONLY` are specified when invoking streaming or device IO functions. Use ***MakeWaveGSMHeader()*** and ***MakeWavePCMHeader()*** to insert headers when desired.

Wave File Playback

The ***MTPlayFile()*** function can playback files with a .wav header that were recorded with the SmartWORKS SDK. AudioCodes does not guarantee that files with .wav headers generated by other recorders will be processed correctly by the SmartWORKS SDK. Files packaged by another system may contain data in the header that our SDK does not support. For example, the SmartWORKS SDK cannot play files that contain information in the FACT section of the header. Also, files that contain data chunk elements throughout the file do not playback correctly. Only the first header (or data chunk) is processed. From here the SmartWORKS SDK plays the file until end of file is reached.

Media Format Naming

The supported voice media formats are defined with an `MT_` prefix and can be found in NtiData.h.

Voice Format	Bits/sec.	Description	Frame Size in Bytes
MT_PCM_uLaw MT_PCM	64000	µ-law 8 bit PCM	160
MT_PCM_ALaw	64000	A-law 8 bit PCM	160
MT_PCM_Raw_8bit MT_PCM_Linear_8bit	64000	Linear 8 bit PCM, signed	160
MT_PCM_Raw_u8bit MT_PCM_Linear_u8bit	64000	Linear 8 bit PCM, unsigned	160
MT_PCM_Raw_16bit MT_PCM_Linear_16bit MT_Linear	128000	Linear 16 bit PCM, signed	320
MT_PCM_Raw_u16bit MT_PCM_Linear_u16bit	128000	Linear 16 bit PCM, unsigned	320
MT_PCM_Raw6k_16bit MT_PCM_Linear6k_16bit	96000	Signed Linear 6 KHz 16-bit PCM	240
MT_PCM_µLaw_Stereo	128000	µ-law 8 KHz 8-bit stereo PCM (record only)	320
MT_PCM_ALaw_Stereo		A-law 8KHz 8-bit stereo PCM (record only)	320
MT_OKI_ADPCM_SR8	32000	Dialogic ADPCM 8 KHz sample rate	80
MT_OKI_ADPCM_SR6	24000	Dialogic ADPCM 6 KHz sample rate	60

Voice Format	Bits/sec.	Description	Frame Size in Bytes
MT_G729_8K*	8000	G.729	20
MT_G729A_8K	8000	G.729A	20
MT_GSM610	13000	GSM 6.10	33
MT_MSGSM	13000	Microsoft GSM	65 (40 ms frames)
MT_G723_DOT1_5300_FIX	5300	G.723.1 fixed rate 5.3 K	20 (30 ms frames)
MT_G723_DOT1_6300_FIX	6400	G.723.1 fixed rate 6.3 K	24 (30 ms frames)
MT_G726_16K	16000	G.726 16K bps	40 (20 ms frames)
MT_G726_16K_µlaw*	16000	G.726 16K bps for µ-law	40 (20 ms frames)
MT_G726_16K_Alaw*	16000	G.726 16K bps for A-law	40 (20 ms frames)
MT_G726_24K	24000	G.726 24K bps	60 (20 ms frames)
MT_G726_24K_µlaw*	24000	G.726 24K bps for µ-law	60 (20 ms frames)
MT_G726_24K_Alaw*	24000	G.726 24K bps for A-law	60 (20 ms frames)
MT_G726_32K	32000	G.726 32K bps	80 (20 ms frames)
MT_G726_32K_µlaw*	32000	G.726 32K bps for µ-law	80 (20 ms frames)
MT_G726_32K_Alaw*	32000	G.726 32K bps for A-law	80 (20 ms frames)
MT_G726_40K	40000	G.726 40K bps	100 (20 ms frames)
MT_G726_40K_µlaw*	40000	G.726 40K bps for µ-law	100 (20 ms frames)
MT_G726_40K_Alaw*	40000	G.726 40K bps for A-law	100 (20 ms frames)
MT_G726_16K_MSb_1st	16000	G.726 16K bps MS-bit first	40 (20 ms frames)
MT_G726_16K_µlaw_MSb_1st*	16000	G.726 16K bps for µ-law MS-bit first	40 (20 ms frames)
MT_G726_16K_Alaw_MSb_1st*	16000	G.726 16K bps for A-law MS-bit first	40 (20 ms frames)
MT_G726_24K_MSb_1st	24000	G.726 24K bps MS-bit first	60 (20 ms frames)
MT_G726_24K_µlaw_MSb_1st*	24000	G.726 24K bps for µ-law MS-bit first	60 (20 ms frames)
MT_G726_24K_Alaw_MSb_1st*	24000	G.726 24K bps for A-law MS-bit first	60 (20 ms frames)
MT_G726_32K_MSb_1st	32000	G.726 32K bps MS-bit first	80 (20 ms frames)
MT_G726_32K_µlaw_MSb_1st*	32000	G.726 32K bps for µ-law MS-bit first	80 (20 ms frames)
MT_G726_32K_Alaw_MSb_1st*	32000	G.726 32K bps for A-law MS-bit first	80 (20 ms frames)
MT_G726_40K_MSb_1st	40000	G.726 40K bps MS-bit first	100 (20 ms frames)
MT_G726_40K_µlaw_MSb_1st*	40000	G.726 40K bps for µ-law MS-bit first	100 (20 ms frames)
MT_G726_40K_Alaw_MSb_1st*	40000	G.726 40K bps for A-law MS-bit first	100 (20 ms frames)
MT_PCM_uLaw_POWER#	64000	µ-law 8 bit PCM, with energy tagging	160 (20 ms frames)
MT_PCM_ALaw_POWER#	64000	A-law 8 bit PCM, with energy tagging	160 (20 ms frames)
MT_G723_DOT1_5300_FIX_POWER#	5300	G.723.1 fixed rate 5.3 K	20 (30 ms frames)

*Decode Only
#Encode only

MF Detection

Each channel on SmartWORKS products has two sets of MF detectors. One set is connected to the primary input, the other is connected to the secondary input. There are two types of MF digits: R1 and R2. These digits are part of different signaling systems and in all practicality user applications will work with either R1 or R2 digits, but not both at the same time, hence there is only one MF digit queue. The API allows selection between R1 and R2 digit detection.

R1 MF Digits

In this document we will refer to R1 as defined by CCITT (now ITU) in CCITT Recommendation Q.151, vol.VI, Geneva 1980. MF R1 digits are designed as combinations of 2 out of 7 frequencies, beginning with 700Hz and separated by 200Hz.

The table below shows the 15 MF R1 digits and the frequencies corresponding to each digit:

R1 Digit	Lower Frequency (Hz)	Higher Frequency (Hz)	API Code
1	700	900	0x01
2	700	1100	0x02
3	900	1100	0x03
4	700	1300	0x04
5	900	1300	0x05
6	1100	1300	0x06
7	700	1500	0x07
8	900	1500	0x08
9	1100	1500	0x09
0	1300	1500	0x0A
Code 11	700	1700	0x0B
Code 12	900	1700	0x0C
KP1	1100	1700	0x0D
KP2	1300	1700	0x0E
ST	1500	1700	0x0F

R2 MF Digits

R2 digits referred to here were defined by CCITT (now ITU) in CCITT Recommendation Q.441, vol.VI, Geneva 1980. There are 15 digits called forward digits and 15 digits called backward digits. Forward digits are transmitted from Central Office (CO) to Customer Premises Equipment (CPE). Reverse digits are transmitted from CPE to CO. The forward digits are designed as combinations of two out of set of six frequencies: 1380, 1500, 1620,1740, 1860, and 1980 Hz. The backward digits are designed as a combinations of two out of six frequencies different then the forward set: 540, 660, 780, 900, 1020 and 1140 Hz

The table below shows frequency composition of each forward digit (values in italics are optional):

R2 Digit (Forward)	Lower Frequency (Hz)	Higher Frequency (Hz)	API Code
1	1380	1500	0x01
2	1380	1620	0x02
3	1500	1620	0x03
4	1380	1740	0x04
5	1500	1740	0x05
6	1620	1740	0x06
7	1380	1860	0x07
8	1500	1860	0x08
9	1620	1860	0x09
10	1740	1860	0x0A
11	*1380*	*1980*	*0x0B*
12	*1500*	*1980*	*0x0C*
13	*1620*	*1980*	*0x0D*
14	*1740*	*1980*	*0x0E*
15	*1860*	*1980*	*0x0F*

The table below shows fequency composition of each backward digit (values in italics are optional).

R2 Digit (Backward)	Lower Frequency (Hz)	Higher Frequency (Hz)	API Code
1	1020	1140	0x11
2	900	1140	0x12
3	900	1020	0x13
4	780	1140	0x14
5	780	1020	0x15
6	780	900	0x16
7	660	1140	0x17
8	660	1020	0x18
9	660	900	0x19
10	660	780	0x1A
11	*540*	*1140*	*0x1B*
12	*540*	*1020*	*0x1C*
13	*540*	*900*	*0x1D*
14	*540*	*780*	*0x1E*
15	*540*	*660*	*0x1F*

Board Type Naming

AudioCodes SmartWORKS boards are defined in NtiData.h as follows:

TABLE 3: BOARD TYPE NAMING

Board Name	Actual Product Name	Description
DT3209_T1_CARD	SmartWORKS DT3209	24 channel T1 terminate card G729A-compatible
DT3209_E1_CARD	SmartWORKS DT3209	30 channel E1 terminate card G729A-compatible
DT6409_T1_CARD	SmartWORKS DT6409	48 channel T1 terminate card G729A-compatible
DT6409_E1_CARD	SmartWORKS DT6409	60 channel E1 terminate card G729A-compatible
DT3209TE_T1_CARD	SmartWORKS DT3209	24 channel T1 terminate card G729A-compatible
DT3209TE_E1_CARD	SmartWORKS DT3209	30 channel E1 terminate card G729A-compatible
DT6409TE_T1_CARD	SmartWORKS DT6409	48 channel T1 terminate card G729A-compatible
DT6409TE_E1_CARD	SmartWORKS DT6409	60 channel E1 terminate card G729A-compatible
DP3209_T1_CARD	SmartWORKS DP3209	24 channel T1 high impedance call logging card, G729A-compatible
DP3209_E1_CARD	SmartWORKS DP3209	30 channel E1 high impedance call logging card, G729A-compatible
DP6409_T1_CARD	SmartWORKS DP6409	48 channel T1 high impedance call logging card, G729A-compatible
DP6409_E1_CARD	SmartWORKS DP6409	60 channel E1 high impedance call logging card, G729A-compatible
NGX_CARD	SmartWORKS NGX	8 channel modular digital station tap base card
	MX80	8 channel expansion for SmartWORKS NGX
ANALOG_LD409	SmartWORKS LD 409	4 channel high/low impedance analog call logging card
ANALOG_LD409H	SmartWORKS LD 409h	4 channel high/low impedance analog call logging card with H.100 bus
ANALOG_LD809	SmartWORKS LD 809	8 channel high/low impedance analog call logging card with H.100 bus
ANALOG_LD809X	SmartWORKS LD 809X	8 channel high/low impedance analog call logging card with H.100 bus
ANALOG_LD1609	SmartWORKS LD 1609	16 channel high/low impedance analog call logging card with H.100 bus
ANALOG_LD2409	SmartWORKS LD	24 channel high/low impedance analog call logging card with H.100 bus
PCM6409_32T_CARD	SmartWORKS PCM 6409	dual port terminate card with PCM option

TABLE 3: BOARD TYPE NAMING

Board Name	Actual Product Name	Description
PCM3209_32T_CARD	SmartWORKS PCM 3209	single port terminate card with PCM option

Windows Event Viewer

Following is a list of events that may appear in the Windows Event Viewer. Each listing has a description of the event next to it. This is not an exhaustive list of Windows events, only a list of SmartWORKS-specific events.

NOTE: When using Linux, all information is written to a 'messages' file located in the /var/log directory.

In the first table is a list of events stemming from the NtiDrv.sys, the SmartWORKS Driver. Driver issues can be Hardware, Firmware, or Firmware/Driver incompatibility issues.

TABLE 4: WINDOWS EVENT MESSAGES FROM NTIDRV.SYS

Event Message	Description
Error locating SmartWORKS hardware	At least one SmartWORKS board needs to be present (hardware problem: no recognizable board)
Hardware initialization timed out for brd #	Board # initialization timed out (Firmware problem: firmware unable to finish initialization)
Unsupported hardware detected for brd #	Board # format ID error (hardware problem: setup error)
Incompatible (V#.#.# and up required) FW on brd #	Old firmware on board # (Firmware-driver incompatibility problem: need to update firmware to specified version)
Board PANIC code at 0x10 for brd #	Board # firmware runs into problem (code at 0x10), contact AudioCodes (firmware problem)
SmartWORKS driver loaded nevertheless	Driver loaded in spite of the error, LED CR17 flashing (probable follow up message to errors 2-5)
SmartWORKS driver loaded	Driver loaded successfully (if no errors-normal)
SmartWORKS driver unloaded	Driver unloaded (if no errors-normal)

The next list of event messages stem from the NtiDrv.dll, the SmartWORKS API. API issues can be API, Application, Firmware, or Hardware issues.

TABLE 5: WINDOWS EVENT MESSAGES FROM NTIDRV.DLL

Event Message	Description
MVIP not specified	MVIP not selected in SmartControl
Clear MUX Timed out	Start MUX timed out [API function time-out MTStartMUX()]
Clear MUX on channel # Timed out	Start MUX on channel # timed out [API function time-out MTStartMUXOnChannel(#)]
Init MUX Timed out	Stop MUX timed out [API function time-out MTStopMUX()]
Init MUX on channel # Timed out	Stop MUX on channel # timed out [API function MTStopMUXOnChannel(#) has timed out]
Event Callback for Ch # dropped due to queue full	Event callback on channel # dropped (System problem: Event Callback queue full-contact AudioCodes)
Completion Callback for Ch # dropped due to queue full	Completion callback on channel # dropped (System problem: Completion Callback queue full-contact AudioCodes)
Ch # timed out StreamBufIn	Streaming in error on channel # (API error: contact AudioCodes)
Ch # timed out StreamBufOut	Streaming out error on channel # (API error: contact AudioCodes)
Brd # event 0x1234 dropped due to queue full	Board # event 0x1234 dropped (Application error: must clear event queue)
Ch # event 0x5678 dropped due to queue full	Channel # event 0x5678 dropped (Application error: must clear event queue)
DLL attached error status 0xabcd	SmartWORKS API load failed with code 0xabcd
DLL detached successfully	SmartWORKS API unloaded successfully (Normal)
DLL detached error status 0xabcd	SmartWORKS API unload failed with code 0xabcd
NTI DLL attached with maximum log count set to 100	SmartWORKS API loaded successfully (Normal)

With the occurrence of any other error, call AudioCodes.

Chapter 4

Writing An Application

Getting Started

This section provides basic information about writing an application with the SmartWORKS API.

Important Note for Linux Developers

When running the Linux it is important that the user's application run as root user. The following information shows the command required for each application to run as root user.

1) Log in as root;

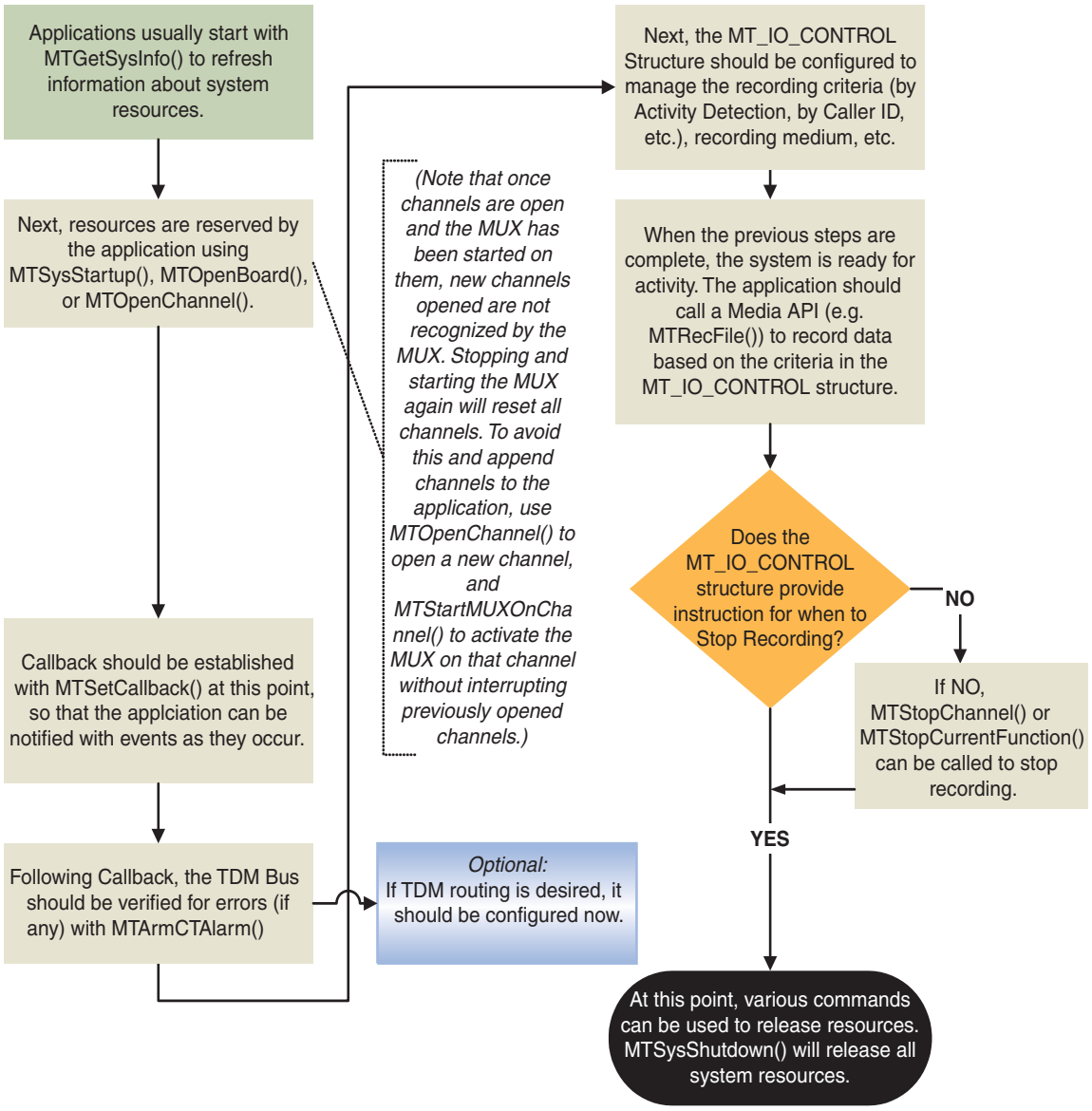
2) Make sure that the ownership of the application is root (or super user). Or, use command 'chown root application' to change it to root;

3) use command 'chmod +s application' to add run time superuser privilege to this application;

4) Log out.

SmartWORKS Flowchart

Each application written for SmartWORKS follows the same basic processing scheme. The following flowchart depicts this flow control.



Event Control

Three type of events are generated by AudioCodes boards: board events, channel events and system events.

NOTE: A detailed description of each event is available in the SmartWORKS Function Reference Library.

Each channel has a FIFO buffer or event queue, which is used to temporarily store asynchronous messages. These messages, or events, are used to indicate channel, board or system status information such as line signaling, voice activity, or errors. The event queue for each channel can store up to 64 messages. Regardless of which method is used, the user application is responsible for monitoring the channel status often enough to ensure that it does not overflow. If the event queue is full new events are lost and are reported in the Windows Event Viewer. **NOTE:** When using Linux, all information is written to a 'messages' file located in the /var/log directory.

With event queuing, the application can pull the messages off of the queue by calling ***MTWaitForChannelEvent()*** in a polling routine. All event queue messages must be pulled out of the FIFO in order to keep it from overflowing. Many of the messages can be ignored if they are not important to the application.

The second and preferred method of reading the event queue is to use a call back function. A call back function will be invoked by any event or by a specific event. When a callback function is invoked, events are passed as parameters. There is no need to call ***MTWaitForChannelEvent()***.

Following is a list of Event Information and Control API functions:

TABLE 6: EVENT INFORMATION AND CONTROL

MTClearBoardEventCallback()
MTClearEventCallback()
MTClearPriorityEventCallback()
MTEventControl
MTFlushEvents()
MTGetChannelEvent()
MTGetEventFilters()
MTPutChannelEvent()
MTSetBoardEventCallback()
MTSetEventCallback()
MTSetEventFilters() (prev. MTSetEvents)
MTSetPriorityEventCallback()
MTSysSetEventCallback()
MTSysClearEventCallback()
MTSysWaitForEvent()
MTWaitForAdapterEvent()
MTWaitForChannelEvent() (prev. MTWaitFor-Event)

FULL EVENT QUEUE

When events are dropped due to a full event queue, the message *ch n event 0xAB dropped due to full queue* will be posted through the Windows Event viewer, where *n* is the channel number and *0xAB* is the event code. **NOTE:** When using Linux, all information is written to a 'messages' file located in the /var/log directory.

POLLING

The function ***MTWaitForChannelEvent()*** is used to get the first queued event. The application passes a pointer to an empty MT_EVENT structure. If an event is available, the DLL fills the event structure with information such as time stamp, event code, or extended sub-reason codes. Should there be no event queued; the DLL waits for either the expiration of the specified time-out period or the arrival of a new event. Some common events that are unimportant to a specific channel can be disabled so that the events are never queued for that channel. Function ***MTSetEventFilters()*** uses a bit field to enable or disable a number of common events.

Events with Extra Information

Some events, such as call control, EVT_CC_ events, pass over extra information to the user application. The field *ptrBuffer* is set by the SmartWORKS DLL to identify the buffer holding the extra information. The SmartWORKS DLL allocates a buffer to hold this data. After the data is retrieved, the user application must invoke ***MTReturnEventBuffer()*** to return this buffer.

As of release 2.8, events that are generated with extra information are handled differently by the DLL. To enable this feature use ***MTSetSystemConfig()*** and set the field *CompileWithSDK* to a value that is greater than 0x02086600. Now the user application is required to allocate a buffer for this extra information. Use the *ptrXtraBuffer* field to identify the pointer for the extra information, and the *XtraBufferLength* field to set the length of the buffer. If these fields are not set by the user application, the extra data is not provided to the user application.

When data is returned to the user application, the SmartWORKS DLL sets the *XtraDataLength* field with the actual data length returned to the user application. If the actual length of data is larger than the allocated buffer, the data is truncated and a flag in Bit 2 of the *EventFlag* field is set to 0x00000004.

The following is an example of a basic Polling method:

```
#include <windows.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <NTiAPI.H>


int main(int argc, char *argv[])
{
        int totalChannels;
        MT_RESULT r;
        int chan =1;//this assumes GCI index = 1
        MT_EVENT event;
        int Code, Reason;
        char callerIDBuf[128];
        MTSYS_INFO sysInfo;
        r=MTSysStartup();
```

```
if(r!=MT_RET_OK)
{
printf("MT system failed to start");
return 1;
}
ULONG size = sizeof(sysInfo);
r=MTGetSystemInfo(&sysInfo,&size);
if(r==MT_RET_OK)
totalChannels=sysInfo.NumChans;
else
{
printf("MTGetSystemInfo failed...\n");
return 1;
}

while (1)
{
// Note: MTWaitForChannelEvent may reset your size to 0
// if there is no event, therefore set the size everytime
// before calling MTWaitForChannelEvent
size = sizeof(event);
r=MTWaitForChannelEvent(chan,10,&event,&size);
if(r==MT_RET_OK)
{
Code = event.EventCode;
Reason = event.SubReason;

switch (Code)
{
case EVT_CALLID_STOP:
r=MTGetCallerID(chan, 128, callerIDBuf );
printf("Ch. %d Caller ID length %d data '%s'",
chan, strlen(callerIDBuf), callerIDBuf );
break;

case EVT_CALLID_DROPPED:
printf("Ch. %d Caller ID dropped ", chan);
break;

case EVT_MAX_SILENCE:
break;

case EVT_MAXTIME:
break;

case EVT_MAXBYTES:
break;

case EVT_MAXDIGITS:
break;

case EVT_DIGIT:
```

```
            break;

            case EVT_EOF:
            break;

            case EVT_RINGS:
            break;
            }
            }

            chan++;
            if (chan >totalChannels)
            chan =1;
}
return 0;
}
```

CALL BACK FUNCTION

The second and preferred method of reading the event queue is to use a *call back* function. A call back function must be registered with the SmartWORKS API by the application and will be invoked when the specified event occurs. When a call back function is invoked, events are passed as parameters back to the application. A call back function does not require the application to poll for events.

Events with Extra Information

Some events, such as call control, EVT_CC_ events, pass over extra information to the user application. The field *ptrBuffer* is set by the SmartWORKS DLL to identify the buffer holding the extra information. The *DataLength* field is used to identify the size of the buffer. When event information is retrieved, the user application must invoke ***MTReturnEventBuffer()*** to release the buffer.

As of release 2.8, events that are generated with extra information are handled differently by the DLL. To enable this feature use ***MTSetSystemConfig()*** and set the field *CompileWithSDK* to a value that is greater than 0x02086600. In this scenario, the field *ptrXtraBuffer* is set by the SmartWORKS DLL to identify the buffer holding the extra information and the *XtraBufferLength* indicates the length of data in the buffer.

The following is an example of a basic Call Back method:

```
// include files
#include<NTiAPI.h>
#include<NTiEvent.h>

// callback
CINTERFACE void APICallback(CHANNEL iCh,int EventCount,PMT_EVENT pMt,LPARAM
lParam)
{
   int Code,Reason;

   for(int i=0 ; i < EventCount; i++)
   {
                                    Code= pMt[i].EventCode;
                                    Reason= pMt[i].SubReason;

                                    switch(Code)
                                    {
                                    case EVT_DIGIT:
                                    break;
                                    case EVT_CC_DISC_CONF:
                                    break;
                                    ...
                                    ...
                                    }
   }
}


      // How to set callback for channels.
      // register callback for channels
      // gci is gci index which you set using control pannel and we assume that
it is set to 1.

      MTSetEventCallback(0,(MTCALLBACK)APICallback,0);
```

System Wide Definitions

Return Codes

All API functions return a code of type `MT_RET_...` for the status of the API function's execution. It is vital that the user application checks the status of each call to see if it was successful.

NOTE: All return codes are passed to the user application as a hex value. To obtain a text desciption use the API ***MTGetReturnCodeDescription()***.

All SmartWORKS return codes are listed below:

SmartWORKS Return Codes[1]

Return Code	Hex value	Meaning
MT_EXCEPTION	0xE000220FL	An exception occurred during the API call
MT_RET_API_THREAD_PROTECTED	0xE0002019L	API is referred by other thread
MT_RET_BAD_CURSOR	0xE0002031L	Invalid index offset
MT_RET_BAD_JOIN	0xE0002029L	Invalid crosspoint coordinates
MT_RET_BOARD_INUSE	0xE0002018L	Board device in use by another application
MT_RET_BOARD_NOT_OPENED	0xE0002016L	Board device not opened for access
MT_RET_BUSY	0xE0002010L	Channel is busy
MT_RET_CANNOT_CREATE_EVTQEVENT	0xE0002205L	Unable to create event for the event queue
MT_RET_CANNOT_CREATE_MUTEX	0xE000220EL	Mutex creation failed
MT_RET_CHANNEL_INUSE	0xE0002017L	Channel device not opened for access
MT_RET_CHANNEL_NOT_OPENED	0xE0002015L	Channel device not opened for access
MT_RET_COMPAT_NOTSUPPORTED	0xE000220CL	Compatibility mode is not supported in this implementation
MT_RET_DATA_TRUNCATED	0x60002036L	Buffer allocated by application does not have enough space for data structure associated with specified API. This return code is only used if FlagDataTruncated is enabled in the MTSYS_CONFIG or MTSYS_CONFIGURATION data structure is enabled. Otherwise, the data is truncated the MT_RET_OK is returned to users.
MT_RET_DEVICE_NOTFOUND	0xE0002200L	Voice interface device was not found
MT_RET_EVTQMUTEX_ABANDONED	0xE0002208L	The event queue synchronization mutex was abandoned by another thread
MT_RET_EXTENDED_ERROR	0xE0002024L	Extended error
MT_RET_FORMAT_NOT_SUPPORTED	0xE0002001L	Media format is not supported
MT_RET_IDLE	0xE0002011L	Channel is idle
MT_RET_BOARD_INCOMPATIBLE_BINFILE	0xE0002043L	file type does not match board type
MT_RET_INTERNAL_ERROR	0xE0002201L	Internal DLL error
MT_RET_INVALID_BINFILE	0xE0002042L	image file version cannot be retrieved

SMARTWORKS RETURN CODES[1] (CONTINUED)

Return Code	Hex value	Meaning
MT_RET_INVALID_BOARD	0xE0002008L	Board number is out of range
MT_RET_INVALID_BUSTYPE	0xE0002030L	Bus type is invalid
MT_RET_INVALID_CHANNEL	0xE0002009L	Channel number is out of range
MT_RET_INVALID_DEVICE_IO	0xE0002013L	Invalid Device IO setting
MT_RET_INVALID_EVENT_QUEUE	0xE0002207L	Invalid event queue
MT_RET_INVALID_FILEINDEX	0xE000220AL	The MT_IO_CONTROL FileIndex field is out of range for the file
MT_RET_INVALID_FILENAME	0xE0002210L	Invalid filename
MT_RET_INVALID_FUNC	0xE0002006L	Function code is not valid
MT_RET_INVALID_HW_INT	0xE0002007L	Bad hardware IRQ at initialization
MT_RET_INVALID_PARAM	0xE0002012L	Invalid argument passed to function
MT_RET_IO_PENDING	0x60002033L	Background I/O operation is in progress or queued.
MT_RET_ISDNPT_BAD_TRUNK_NUMBER	0xE0002216L	ISDN channel not found
MT_RET_ISDNPT_CALL_NOT_FOUND	0xE0002215L	ISDN channel not found
MT_RET_ISDNPT_CHANNEL_NOT_FOUND	0xE0002214L	ISDN channel not found
MT_RET_LIB_LOAD_FAILED	0XE0002040L	driver load failed
MT_RET_MALFUNCTION	0xE0002005L	Hardware failed during initialization
MT_RET_MISSING_TERMINATOR	0xE0002212L	A terminating entry was not found in an array argument.
MT_RET_MUX_OFF	0xE0002028L	Multiplexer system was not started
MT_RET_MUX_ON	0xE0002027L	Multiplexer system was already started
MT_RET_MUX_THREAD_PROTECTED	0xE000201AL	MUX is referred by other thread
MT_RET_NO_ASSOC_EVTQ	0xE0002209L	No event queue has been associated with channel
MT_RET_NO_DIGIT	0xE0002002L	No digit available
MT_RET_NO_MEM	0xE0002206L	An operation could not be completed due to insufficient memory or an error occurred while attempting to allocate memory
MT_RET_NO_MUX	0xE0002026L	Multiplexer is not available
MT_RET_NOT_AVAILABLE	0xE0002014L	Feature is not available
MT_RET_NOT_APPLICABLE	0xE000201D	This feature is available, but is currently not enabled.
MT_RET_NOT_ENOUGH_BUFSPACE	0x60002035L	StreamOut does not have enough buffering space for API request
MT_RET_NOT_ENOUGH_DATA	0x60002034L	StreamIn does not have enough data for API request
MT_RET_NOT_STARTED	0xE0002004L	Voice system not started

SMARTWORKS RETURN CODES[1] (CONTINUED)

Return Code	Hex value	Meaning
MT_RET_OK	0x00000000L	No error
MT_RET_QFULL	0xE0002032L	Invalid index offset
MT_RET_SERVICE_ALREADY_STARTED	0xE0002204L	Requested service already started
MT_RET_SERVICE_NOT_STARTED	0xE0002203L	Voice service not started
MT_RET_SERVICE_STOPPED	0x60002213L	Voice service stopped or terminated
MT_RET_STARTED	0xE0002003L	Voice system already started
MT_RET_STREAMIN_THREAD_PROTECTED	0xE000201BL	STREAMIN is referred by other thread
MT_RET_STREAMOUT_THREAD_PROTECTED	0xE000201CL	STREAMOUT is referred by other thread
MT_RET_UNKNOWN_DEVICE_ERROR	0xE0002202L	Unknown device error
MT_RET_UNKNOWN_IOTRANS_HANDLE	0xE000220DL	The handle in the IO_TRANS structure was not obtained through the AudioCodes API file open functions
MT_RET_VERSION_MISMATCH	0xE0002211L	Version mismatch
MT_RET_ZERO_FILELENGTH	0xE000220BL	The MT_IO_CONTROL FileLength field is zero
MT_TIMEOUT	0xC0040013L	Wait for event timed out
MT_XRET_BADFUNCTION	0xE0002120L	Bad function
MT_XRET_BADHANDLE	0xE0002114L	Bad handle
MT_XRET_BADPARAM	0xE0002121L	Invalid argument passed to function
MT_XRET_QEMPTY	0xE0002119L	Event queue is empty
MT_XRET_QFULL	0xE0002118L	Event queue is full (has reached maximum limit)

1. All return codes are defined in the NtiErr.h file.

Matrix folder.

Using Data Structures

Structures are used to pass parameters to or from the application. Only a pointer to the structure is passed, not the whole structure. The application fills an input structure and passes the pointer to the API, which reads or copies the information. The application also passes a pointer to the output structure which the API fills with the requested information.

All input structures are buffered in the API. This means that the application can only destroy or reuse the structure after it has been populated by the function. Voice buffers and DTMF buffers used by a background function must stay in existence until the application receives a terminating event for that function.

As noted earlier, all entries in an input structure that are reserved, not needed, or not used must be set to zero to ensure future compatibility.

ZERO OUT PARAMETERS

Not all functions use all parameters. In cases where not all parameters are used, the value of unused parameters should always be set to zero.

There are several ways to "zero out" unused parameters:

One way is to make a default structure with the most commonly used parameters set to a default value and the rest to zero. You would then copy this structure to a local structure where specific values are filled in.

Another way is to always zero out the structure before filling in the values.

This section defines two structures that are used by many of the SmartWORKS APIs: MT_IO_CONTROL and MT_EVENT. For exact structure definitions refer to the NtiData.h header files and API definitions in the next chapter of this book.

NOTE - The structures in the following sections do not include reserved parameters.

Chapter 5

Theory of Operation

Overview

This chapter provides theoretical discussions when using the SmartWORKS API.

System Functions

The following section provides an overview of the SmartWORKS functions relative to system configuration and information.

System Configuration

Users can alter the configuration of the system via the ***MTSysSetConfig()*** function (***MTSetSystemConfig()*** is maintained for backwards compatibility). This API sets a system configuration into the Windows registry, however the NTI driver and DLL rely on system and adapter level parameters at load time to configure their behavior. When the configuration is changed, both the driver and the DLL must be reloaded for the new configuration to take effect. To do this the host machine must be rebooted.

System Information

MTGetSystemInfo() fills the provided MTSYS_INFO structure with information about the system status, number of boards, number of channels, and the MUX operation information of the system.

MTGetVersion() returns the version of the SmartWORKS driver and on board firmware. The versions are for the first board. Use **MTGetAdapterInfo()** to get the versions for each board. The version information is defined as MT_VERSION. The SmartWORKS software version number is composed of two 32-bit version long integers.

MTGetDLLVersion() is a function that is used to retrieve the version information associated with the NtiDrv.dll file.

Sync Host/Board Time

The SmartWORKS API allows a user to resync the host time with a SmartWORKS board. By default, the timestamp is synced when a board/channel is opened for the very first time. Use this API to re-synchronize the time between the host PC and SmartWORKs board without shutting down the NTI application. If the host PC's time is modified this API is called to resync the time so that event timestamps are adjusted.

This API resynchronizes the host time with all SmartWORKs board(s) in the platform provided that the application has access through either the board or a channel. In the case of an NGX card, whose board access is excluded from this resynchronization capability, the application requires access of one NGX channel per NGX board.

If access to a SmartWORKs board is not present, SDK assumes that time resynchronization is not required on this board.

NOTE: If multiple applications are running, such as one application per channel, this API needs to be called from each and every application.

Board Functions and Configuration

The following section provides an overview of the SmartWORKS functions used to manage SmartWORKS boads.

Board Control

Two functions ***MTOpenBoard()*** is used to open individual SmartWORKS boards. When ***MTOpenBoard()*** is used, channels remain closed. Users are recommended to use ***MTSysStartUp()*** to open all boards and channels in a single system.

To close an individual board, users can invoke ***MTCloseBoard()***. Should the user application need to release all resources (boards, channels, TDM timeslots, etc.), it is suggested to do so with **MTSysShutdown()**. Calling of this function is not necessary unless only a specified board is to be closed.

After a board is opened, any application can check the owner status of this board. ***MTGetOpenBoardStatus()*** returns the ownership of the specified board. A board needs to be opened before any access or control can be performed on it by the application.

A board can only be opened by one application at one time. When a board is opened and held by one application, attempting to open the same board will fail. However, APIs such as **MTGetAdapterInfo()**, **MTGetBoardOEMInfo()**, **MTGetBoardAssemblyInfo()** still return board information regardless of the ownership.

Board Information Functions

MTGetAdapterInfo() fills the MTADAPTER_INFO structure with information about the specified board. All SmartWORKS boards are indexed beginning with 0. The total number of boards in the system can be retrieved by using the **MTGetSystemInfo()** function. **MTGetAdapterStatusDesciption()** retrieves the text description that can be used to explain the value returned by the MTADAPTER_INFO.*Status* field.

When called, **MTGetAdapterDescription()** will return an ASCII description of the board in a system. A call to **MTGetAdapterInfo()** is required to use this API to obtain the first parameter, *BoardType*.

When using a SmartWORKS DT or DP board, users can obtain network interface protocol/variant settings by using the ***MTGetAdapterXInfo()*** function. The user application must have the ownership of the specified board.

Board Identification

The SmartWORKS API supports up to 16 physical boards and/or up to 512 full duplex channels within a system. The API functions refer to a specific board and or channel within the system using one of two numbering schemes: physical board numbers, and Global Channel Index (logical channel numbers). All board numbers are assigned sequentially starting from zero.

Certain API functions will allow the developer to reference all boards simultaneously by using the nBoard = -1.

The following set of functions can be used to help identify boards in a system or to set private OEM identifiers to SmartWORKS boards.

Locating Boards in a Chasis

The SmartWORKS API includes a function, ***MTBlinkBoard()*** that causes a board's CR17 LED on the specified board to blink its board index plus one 10 times so that the user can match a board's physical location with its board number. Most SmartWORKS boards use CR17, however some variations are present within the SmartWORKS family. Refer to the *SmartWORKS Users Guide* or the board's Quick Install for the exact CR number and location.

After ***MTBlinkBoard()*** is used, the user can also invoke ***MTStopBlinkBoard()*** at any time to stop the process.

Using a Board's Thumbwheel (NGX only)

The NGX board is designed with a thumb wheel that can be set prior to the board's installation. Once this thumbwheel is used, the function ***MTBoardGetCustomSwitchSetting()*** returns the value set on the board's thumbwheel.

Obtain Board's Serial Number

The ***MTGetBoardAssemblyInfo()*** API retrieves board serial information that resides on the board's EEPROM such as assembly number, hardware revision, serial number, assembly code, and the date code. Board serial information can be read but not altered. The following information is returned to the user application:

Name	Description
AssemblyNumber	NULL terminated string, maximum size 16 bytes
HWRevision	NULL terminated string of hardware revision number, maximum size 4 bytes
SerialNumber	NULL terminated string of board serial number, maximum size 8 bytes
AssemblyCode	NULL terminated string of assembly code, maximum size 4 bytes
DateCode	NULL terminated string of manufacturing date, maximum size 8 bytes

OEM Identification

The SmartWORKS API provides a method that allows users to authenticate products for distribution control purposes. The function ***MTSetAdapterEEPROMConfig()***, provides a method for creating and setting a unique 128 byte identifier to the board. Once this value is set, ***MTGetAdapterEEPROMConfig()*** can be used to retrieve the board's setting.

NOTE: When using MTSetAdapterEEPROMConfig(), if the user passes information that is longer than 128 bytes, the data will be truncated. When using MTGetAdapterEEPROMConfig(), the application returns the value, with a pLength of 128 bytes.

BOARD CONFIGURATION

Users can select to configure the board settings via their application using a SmartWORKS API or via the Control Panel. The NTI driver and DLL rely on system and adapter level parameters at load time to configure their behavior. When the configuration is changed, both the driver and the DLL must be reloaded for the new configuration to take effect. When using the Control Panel to change board configuration, users must restart the board's drivers via the Device Manager (Windos OS) or the command line program when running Linux. When changing board configuration via the SmartWORKS API, functions can be used to restart the board driver and load configuration.

BOARD CONFIGURATION FUNCTIONS

Most board configuration is set via a single API, ***MTBoardSetConfig()***. ***MTSetAdapterConfig()*** is obsolete, but maintained in the SDK for backwards compatibility. **NOTE:** When using the IPX board, users are required to use ***MTBoardSetConfig()*** will includes a data structure to configure Ethernet port parameters.

The following parameters are controlled via the ***MTBoardSetConfig()*** function. The fields that can be modified via the Control Panel are marked with an asterisk (*).

Name	Description
SystemIndex	Adapter index on PCI bus presented by OS, 0 - (MAX_BRD_DEVICES-1)
AdapterType	This field is read only: VR6400, VR6409, and etc. UNKNOWN_CARD for no board in index 'SystemIndex'
	General Adapter Parameters
MasterMode*	MODE_SLAVE(default), MODE_MASTER, MODE_MASTER_A, or MODE_MASTER_B. **NOTE:** This field can also be set using the ***MTSetCTMasterClock()*** function.
MasterClock	Master clock source reference value, default, CLOCK_SOURCE_OSC. **NOTE:** This field can also be set using the ***MTSetCTMasterClock()*** function.
Sec8kNetrefClock	MVIP Sec8K or H100 Netref clock output, default, SPRM_SOURCE_DISABLE
PresentationPreference	Board ordering preference for the associated adapter: Range: 0..MAX_BRD_DEVICES (default) Not implemented for Linux yet
	General CT Bus Parameters
CTBusType*	MUX_MVIP MUX_H100 MUX_NONE(default)
TDMEncoding*	0 for u-law, 1 for A-law
CTBusSegmentIndex	Bus segment index the associated adapter is on
	Interface Parameters
SMSize	Shared memory size in bytes, min MIN_SMSIZE, default 8K

Name	Description
	NGX Parameters Index 0 for base card, 1 for the 1st daughter card, 2 for the 2nd daughter card
CTBusTermination*	0 for disable, others for enable
PBXType[MAX_DC]*	PBX supported. Refer to the NtiData.h file for a complete list of valid values. **NOTE:** If a PBX type is used with an older version of the SDK, this field may not be valid. Users can set this value via the Control Panel.
DChOption[MAX_DC]*	0 for D-channel disabled
Termination[MAX_DC]*	0 for Hi-Z, 1 for 120 ohm
	DT/DP Trunk Parameters Index 0 for 1st trunk and etc.
E1Option*	T1_OPTION for T1, E1_OPTION for E1
Framing[MAX_TRUNKS]*	FRM_E1_G704(default), FRM_T1_SF, etc.
LineCoding[MAX_TRUNKS]*	LC_AMI(default), LC_E1_HDB3, or LC_T1_B8ZS
T1LineBuildOut[MAX_TRUNKS]*	LBO_T1_15DB(default), etc.
E1LineBuildOut[MAX_TRUNKS]*	LBO_E1_120OHM(default) or LBO_E1_75OHM
ZeroCodeSupression[MAX_TRUNKS]*	ZCS_NONE(default), ZCS_GTE, etc. Refer to the NtiData.h file for a complete list of valid values defined for the MT_FRAMER_STATE.ZCS field.
ProtocolSignaling[MAX_TRUNKS]*	SIGNALING_NONE SIGNALING_ISDN SIGNALING_NFAS SIGNALING_RBS (T1 only on SmartWORKS DT)
ProtocolVariant[MAX_TRUNKS]*	E1 & T1: (example PROTOCOL_ISDN_ETS300 national 2 At&t 5ESS DMS 100 NTT Austel 1 Q.SiG RBS (T1 only): (example PROTOCOL_RBS_LOOP_FXO) E&M Wink E&M Immediate Loop_FXO
ISDNInterfaceSide[MAX_TRUNKS]*	SUPPORT_TE(default) or SUPPORT_NT
RBSMaxDigits[MAX_TRUNKS]*	Default 10
RBSInterDigitTime[MAX_TRUNKS]*	Min 10. Default 3000, in unit of ms
RBSDialingDelay[MAX_TRUNKS]*	Min 300, Default 1000, in unit of ms
NFASIndex[MAX_TRUNKS]*	0 to (MAX_NFAS - 1)
	LD Board

Name	Description
OffhookImpedance*	0 = FCC_600, FCC 600 ohms resistive 1 = ETSI_270, ETSI 270 ohm+750 ohm || 0.15 micro-F 2 = AUSTRALIA_220, Australia 220 ohm+680 ohm || 0.12 micro-F 3 = CHINA_200, China 200 ohm+680 ohm || 0.1 micro-F
	IPX Board
MT_IPCONFIG*	IpNetParams[MAX_IP_NET_PORTS] An array of settings for the three ports on the IPX. The MT_IPCONFIG structure is defined below.

Setting the Board's Clock Source

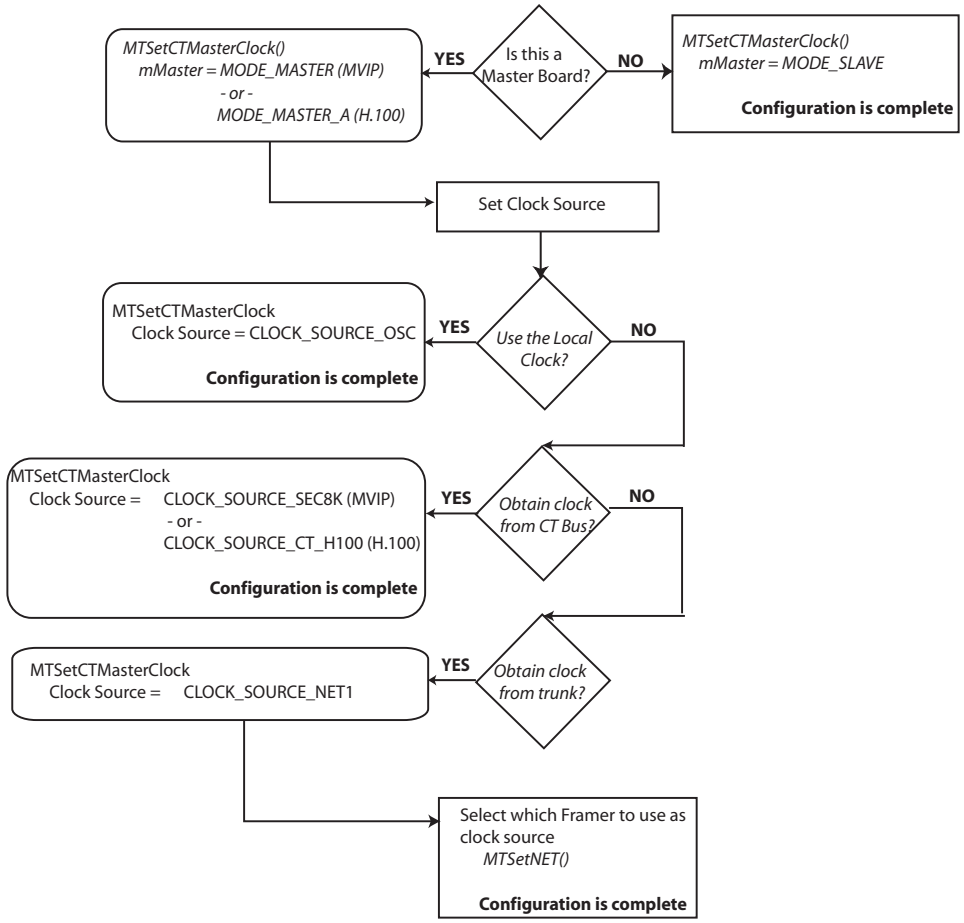
General Rules:

- All Bus Types - to create a system with multiple Master boards that are not connected with H.100 and MVIP cables, Bus Segmentation must be enabled.
- MVIP - to allow more than one Master board per system Bus Segmentation must be enabled using the Control Panel or ***MTSetAdapterConfig()***. If Bus Segmentation is not enabled, then only one Master board is allowed.
- H.100 - allows up to two Master boards per system without enabling Bus Segmentation. To create a system with more than two Master boards, Bus Segmentation must be enabled using the Control Panel or ***MTSetAdapterConfig()***.
- H.100 - if the system is created with two Master boards and Bus Segmentation is disabled, then one board must be set as MASTER_A and the other MASTER_B using ***MTSetCTMasterClock()***.

NOTE: When boards are connected with a Bus cable, AudioCodes does not recommend enabling Bus segmentation.

A flow diagram, on the following page, outlines the process for setting a board's clock source.

NOTE: This flow diagram assumes one Master Board per system and system boards are connected with a H.100 or MVIP cable.



Board Firmware Functions

The SmartWORKS API provides functions that enable the user to upgrade the board's firmware. Refer to the SmartWORKS Function Reference Library for information about the ***MTWFInit()*** and ***MTWFUpgradeFirmware()*** functions.

Managing Board Events

All SmartWORKS boards have system, board and channel events. The SmartWORKS DLL supports both the polling and callback method of event retrieval. Both methods are explained in the previous chapter in the Event Control section. The MT_EVENT data structure is passed to the user with every SmartWORKS events. This data structure contains event specific information. The MT_EVENT data structure is defined in the *SmartWORKS Function Reference Library* in Chapter Three, the Event Code Library.

To enable the polling method for board events use the ***MTWaitforAdapterEvent()*** function. (***MTWaitforBoardEvent()*** is obsoleted, but maintained in the SDK for backwards compatibility). The user application is responsible for monitoring the queue often enough to ensure that it does not overflow. If the event queue is full,

new events are lost and are reported in the Windows Event Viewer. All events are maintained in FIFO order. **NOTE:** When using Linux, all information is written to a 'messages' file located in the /var/log directory.

The second and preferred method is the callback method. A call back function must be registered with the SmartWORKS API by the application and will be invoked when the specified event occurs. When a call back function is invoked, events are passed as parameters back to the application. A call back function does not require the application to poll for events. To enable the callback method for board events, users must invoke the ***MTSetBoardEventCallback()*** function. To clear the registered callback function users can invoke ***MTClearBoardEventCallback().*** This API works as a synchronous function. Therefore, once callback is initiated, it does not release it's lock on the user application until a response has been received. Invoking ***MTClearBoardEventCallback()*** from a callback function would enter this control path into a deadlock. To prevent this the return MT_RET_API_THREAD_PROTECTED message is generated.

Putting an Event on the Board Queue

A user application can also put an event onto the board event queue so that other applications can receive them. To do this, use the ***MTPutBoardEvent()*** function. The user application must fill out the MT_EVENT data structure.

Channel Control and Information Functions

The following section provides an explanation of the SmartWORKS functions used to control, count and access information and statistics on all SmartWORKS channels.

Channel Control Functions

This section provides an overview of the functions used to control channels and obtain information and statistics for all SmartWORKS channels.

Opening and Closing Channels

MTOpenChannel() is used to open individual SmartWORKS channels. This is the preferred method when multiple applications will access a single board or multiple boards in a single system. Prior to invoking ***MTOpenChannel()***, users must first invoke ***MTOpenBoard()***.

Users are recommended to use ***MTSysStartUp()*** to open all boards and channels in a single system. When completed, ***MTSysStartUp()*** opens all boards and channels in a system, plus board communications between the DLL and driver will be opened, and the MUX will be enabled. This application is recommened when one application owns all boards in a single system.

To close an individual channel, users can invoke ***MTCloseChannel()***. Should the user application need to release all resources (boards, channels, TDM timeslots, etc.), it is suggested to do so with **MTSysShutdown()**.

A channel can only be opened by one application at one time. When a channel is opened and held by one application, attempting to open the same channel will fail. However, APIs such as **MTGetChannelInfo()**, **MTGetStatistics()**, and **MTGetChannelOpenStatus()** still return information regardless of the ownership.

Managing Background Functions

Many SmartWORKS functions are background functions which return to the user application when terminating conditions are met. A few functions exist so that users can control the termination of functions via the SmartWORKS API.

MTStopChannel() returns the channel to an idle state. All functions are terminated, and any functions queued for this channel are released. For each active function that is stopped, the event EVT_STOP is reported to the user application. For instance, if the channel is both playing and recording at the same time, then two EVT_STOP events are reported for this channel.

MTStopCurrentFunction() is used to stop the current function which is active on the channel. If more than one function is active, then all are released and a corresponding EVT_STOP is reported for each background function. Any function in the channel's queue begins once this API is completed.

Channel Configuration

When using the SmartWORKS API, various aspects of channel configuration are controlled via specific APIs. For instance, to control Automatic Gain Control use the ***MTChInputSetAGC()*** function. When configuring network settings, there are APIs which correspond to the specific type of network. All network interface APIs are defined in a specific section of this chapter.

Setting Channel to Default

One function, ***MTSetChannelToDefault()*** is used to return the channel to default settings.

Channel Numbering (GCI) Functions

The SmartWORKS API supports up to 16 physical boards and/or up to 512 full duplex channels within a system. The API functions refer to a specific board and or channel within the system using one of two numbering schemes: physical board numbers, and Global Channel Index (logical channel numbers). All board numbers are assigned sequentially starting from zero. Channel numbers are assigned sequentially starting from either 0 or 1 (depending on how the user has configured this setting in the Smart Control panel).

During initialization, as the Physical Boards are numbered, the SmartWORKS software builds a list of the logical channels available in the system. This list is the primary interface the API will use to refer to the channel resources in the system.

NOTE: The SmartWORKS API does not generate a channel list when using the SmartWORK IPX boards.

The Global Channel Index (GCI) specifies whether the channel list is numbered sequentially from 0 or 1 (depending on how the user has configured this setting in the Smart Control panel). Channel numbers are presented in ascending order of the Physical Board numbers. The maximum number of channels supported by SmartWORKS is 512.

Certain API functions will allow the developer to reference all channels simultaneously by using the nChannel = -1 (if GCI index = 0) or nChannel = 0 (if the GCI index = 1).

For Example:

Function ***MTSetEventCallback()*** takes channel number -1 or 0, and registers the callback function for all available channels.

The SmartWORKS API has several commands that can be used to determine the relationship between the GCI and the physical channels on each board. ***The MTGetGCI()*** and ***MTGetGCIMap()*** command will match a GCI indexed channel to its physical board channel location.

For Example:

If GCI index = 0 and MTGetGCIMap(08, pBOARD, pBOARDTYPE, pGCI) returns with *pBOARD=1, and *pGCI=0, this indicates GCI channel 08 resides on board 1 as its first channel. However, MTGetGCI(1,0,pGCI) should return with *pGCI=08.

Channel Information & Statistics

The SmartWORKS API maintains many functions which can be used to obtain information and statistics for all open channels on a system.

MTGetOpenStatus() returns information about the status of the channel - whether it is currently opened or closed. A channel can only be opened by one application at one time. When a channel is opened and held by one application, attempting to open the same channel will fail.

After a channel is opened, any application can check the owner status of this channel. ***MTGetChannelOwner()*** returns the process ID of the access owner of the specified channel. If the channel is currently not opened by any thread or application, the returned value will be 0.

Runtime Information

Channel runtime status is available with the ***MTGetChannelStatus()*** function which can be used to monitor a single channel. This function returns a current and detailed description of a channel's present state via the MTCHAN_STATUS data structure. Information such as line conditions, channel status, bytes decoded and encoded, as well as the total DTMF and MF detections are provided to the user application. The MTCHAN_STATUS structure is defined in the *SmartWORKS Function Reference Library* where ***MTGetChannelStatus()*** is discussed.

The ***MTGetChannelInfo()*** function returns channel properties applied to the channel at board startup. Information such as the bus type and stream speed, along with the timeslots mapped to this channel are provided within the MTCHAN_INFO data structure. The MTCHAN_INFO structure is defined in the *SmartWORKS Function Reference Library* where ***MTGetChannelInfo()*** is discussed.

Runtime Errors

Runtime errors such as encode overflow, decode underflow, and etc. are recorded for each SmartWORKS channel. The ***MTGetStatistics()*** function allows the user application to retrieve these counters. Once completed, the application must invoke ***MTResetStatistics()*** in order to reset the channel counters.

The user application does not need to have the access rights of the specified channel in order to obtain this information. To obtain runtime errors for each channel on the entire board, users may invoke ***MTGetBoardStatistics().*** This API is used in conjunction with ***MTResetBoardStatistics().***

Channel Event Reporting

All SmartWORKS boards have system, board and channel events. The SmartWORKS DLL supports both the polling and callback method of event retrieval. Both methods are explained in the previous chapter in the Event Control section. The MT_EVENT data structure is passed to the user with every SmartWORKS events. This data structure contains event specific information. The MT_EVENT data structure is defined in the *SmartWORKS Function Reference Library* in Chapter Three, the Event Code Library.

To enable the polling method for channel events use the ***MTWaitforChannelEvent()*** function. Once enabled, users invoke the ***MTGetChannelEvent()*** (previously ***MTGetEvent()***) to retrieve channel events. The user application is responsible for monitoring the queue often enough to ensure that it does not overflow. If the event queue is full, new events are lost and are reported in the Windows Event Viewer. All events are maintained in FIFO order. **NOTE:** When using Linux, all information is written to a 'messages' file located in the /var/log directory.

The second and preferred method is the callback method. A call back function must be registered with the SmartWORKS API by the application and will be invoked when the specified event occurs. When a call back function is invoked, events are passed as parameters back to the application. A call back function does not require the application to poll for events. To enable the callback method for board events, users must invoke the ***MTSetEventCallback()*** function. To clear the registered callback function users can invoke ***MTClearEventCallback().*** This API works as a synchronous function. Therefore, once callback is initiated, it does not release it's lock on the user application until a response has been received. Invoking ***MTClearEventCallback()*** from a callback function would enter this control path into a deadlock. To prevent this the return MT_RET_API_THREAD_PROTECTED message is generated.

Priority Events

Users can also set the channel give preferential treatment to an event. **MTSetPriorityEventCallback()** provides a function point that will be called when the specified event occurs on the specified channel. When invoking this API, users must specify the channel and event code that will be handled with priority status. When the specified channel receives the event specified through **MTSetPriorityEventCallback()** function, the priority event callback function will be invoked. Other events will invoke the general event callback function. To remove a priority event callback use the ***MTClearPriorityEventCallback()*** function.

Controlling Event Queues

The SmartWORKS API allows users to manage event queues on all channels. By using the ***MTEventControl()*** function, users can enable, disable or flush the event queue for all channels on the board. To flush events from a specified channel's queue invoke ***MTFlushEvents()***.

Controlling Event Reporting

The SmartWORKS API allows users to control how a channel handles the reporting of events pertaining to line conditions. Use the function ***MTSetEventFilter()*** to enable/disable event reporting of specified line conditions. This function controls event reporting on a per channel basis.

Putting an Event on the Channel Queue

A user application can also put an event onto the channel event queue so that other applications can receive them. To do this, use the ***MTPutChannelEvent()*** function (formally ***MTPutEvent()***). The user application must fill out the MT_EVENT data structure.

Call Connection Functions

Some boards of the SmartWORKS family support a Call Control API for the purpose of managing active calls. The Call Control API is used primarily on the SmartWORKS DT board to provide terminate support. The SmartWORKS DP card also relies on this API to receive call indications for passive monitoring.

SmartWORKS Call Control API

The SmartWORKS Call Control API is an abstraction layer that provides application developers the interface to issue outgoing calls, receive and accept incoming calls, and tear down calls without requiring an extensive knowledge of the signaling protocol. The SmartWORKS Call Control API provides a powerful interface for all signaling protocols.

Call Processing

This section describes how the SmartWORKS Call Control API allows an application developer to manage setup and tear down of incoming and outgoing calls.

The following ISDN variants are supported by the SmartWORKS DT:

ISDN PRI **T1** variants:

- NI-2 (North America)
- AT&T 5ESS (North America)
- Nortel DMS100 (North America)
- Euro-ISDN (Europe & Rest of the World)
- NTT Japan

ISDN PRI **E1** variants:

- Austel 1 (Australia)
- ETS 300
- QSIG (North America)

T1 RBS variants:

- E&M immediate
- E&M Wink
- Loop Start FXS
- Loop Start FXO

Dialog between user applications and the SmartWORKS API is handled by request/confirmation/indication/response exchanges as follows:

The user application issues a request and receives confirmation to its request as an event. The SmartWORKS API issues indications as events and the application responds. The diagram below shows this dialog:

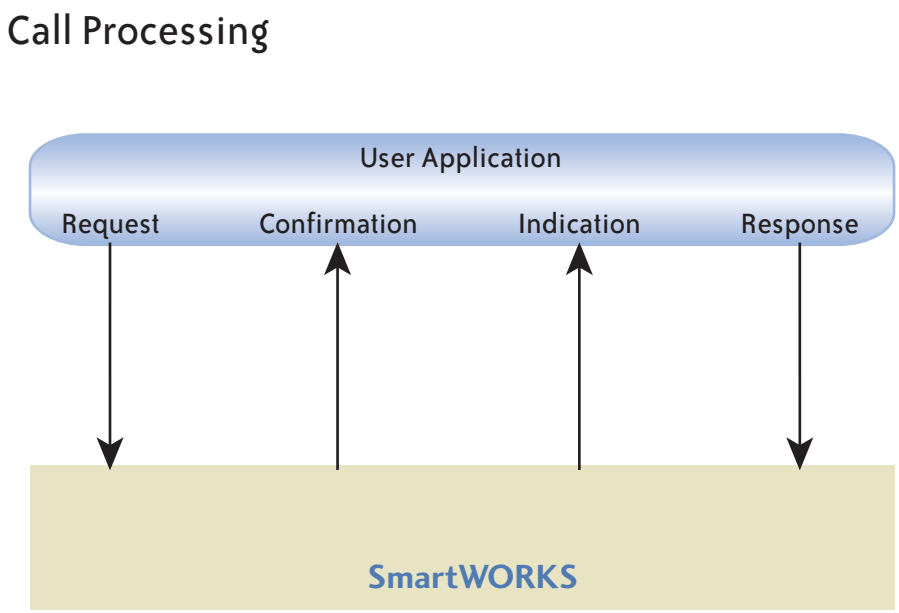


FIGURE 1: CALL PROCESSING

INCOMING CALLS

The diagram below illustrates a typical incoming call accepted by the user's application:

1 · Through the polling or callback mechanism, the application receives the event EVT_CC_CONNECT_IND

2 · The application processes the information carried by this event (*Sending Complete, Called Party Number, Calling Party Number*, etc.)

a When using ISDN: once this event is received, the application accepts the call by calling the API ***MTCC_ConnectResp()***.

b When using RBS: when the SendingComplete field is set, this means that all the called party information is provided. The application then accepts the call by calling the API function ***MTCC_ConnectResp()***. **NOTE:** If the SendingComplete field is 0, this means that the called party number information is not complete yet. The application must wait for the event EVT_CC_INFO_IND to get this information and then accepts the call by calling the API function ***MTCC_ConnectResp()***.

3 · When the call is connected point to point, the application receives the event EVT_CC_CONNECT_CONF.

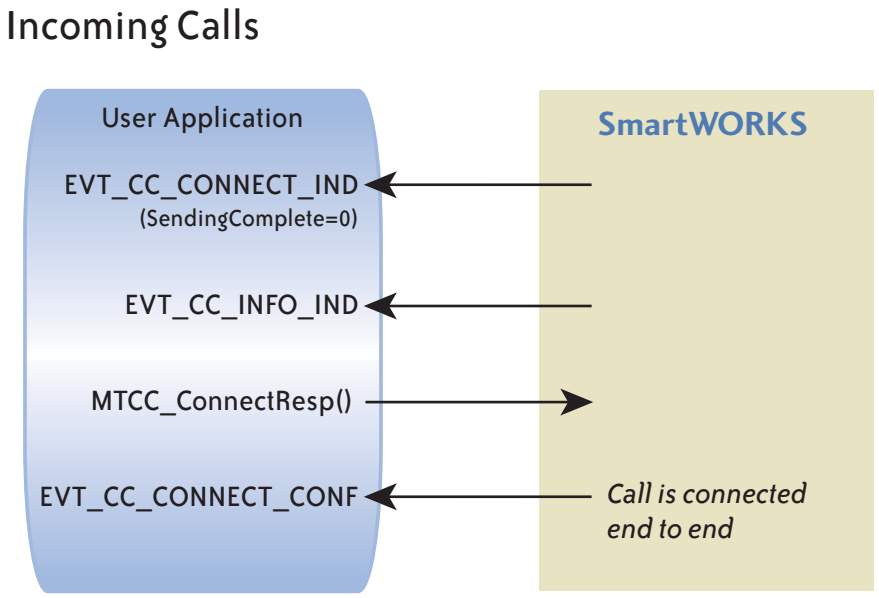


FIGURE 2: INCOMING CALLS

OUTGOING CALL

The diagram below illustrates the Call Control exchange for a typical outgoing call originated by the SmartWORKS application:

1 · The application formats a MT_CC_CONNECT_REQ structure and calls the API function ***MTCC_ConnectReq()***.

2 · Through the polling or callback mechanism, the application receives the event EVT_CC_PROGRESS_IND to indicate that the call is in progress (depending on which signaling protocol is in use.)

3 · The application receives the event EVT_CC_ALERT_IND to indicate that the remote side is being alerted (depending on which signaling protocol is in use.)

4 · When the remote side accepts the call, the application receives the event EVT_CC_CONNECT_CONF to indicate that the call is connected end to end.

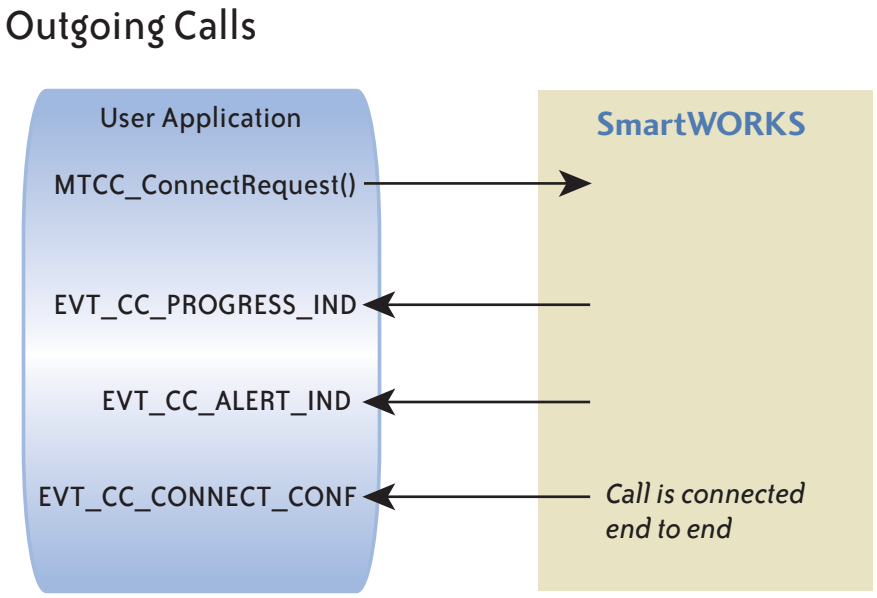


FIGURE 3: OUTGOING CALLS

CALL SCREENING AND NUMBER PRESENTATION

The SmartWORKS API allows users to control outbound calling presentation or screening. These options are controlled by the *PresScrnInd* field in the MT_CC_CONNECT_REQ structure. This field relies as two bits to control presentation and screening (user must apply OR logic, one can use presentation OR screening). The least significant bit controls screening while the most significant bit controls presentation. All fields are defined in the NTIDataCC.h file. The table below provides an explanation of each field:

TABLE 4: OUTBOUND SCREENING AND PRESENTATION OPTIONS

Bit	Name	Description
	Screening Options	
0x00	MT_CC_USR_NOTSCREENED	Calling number is not screened
0x01	MT_CC_USR_VERIFIED_PASSED	Calling number is verified by the local switch. If number is not valid, the call is passed through but flagged as such for the far end
0x02	MT_CC_USR_VERIFIED_FAILED	Calling number is verified by the local switch. If number is not valid, the call is not passed through
0x03	MT_CC_NET_PROVIDED	A calling number known by the local switch is displayed to the far end
	Presentation Options	
0x00	MT_CC_PRES_ALLOWED	Calling number is presented to far side
0x10	MT_CC_PRES_RESTRICTED	The local switch states that the calling number is restricted
0x20	MT_CC_NB_NOT_AVAILABLE	The local switch states that the calling number is not available

APPLICATION INITIATED CALL CLEARING

The diagram below shows the Call Control exchange for a typical call tear down initiated by a user application:

1 · The application formats an MT_CC_DISC_REQ structure and calls the API function ***MTCC_DiscRequest()***.

2 · The polling or callback mechanism notifies the application that the call tear down is complete and that the resources are available via the EVT_CC_DISC_CONF event.

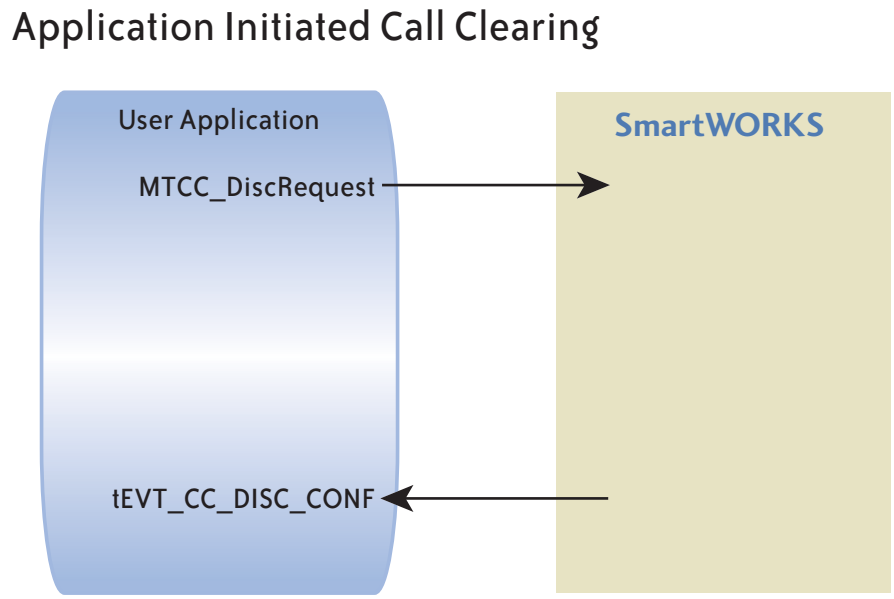


FIGURE 5: APPLICATION INITIATED CALL CLEARING

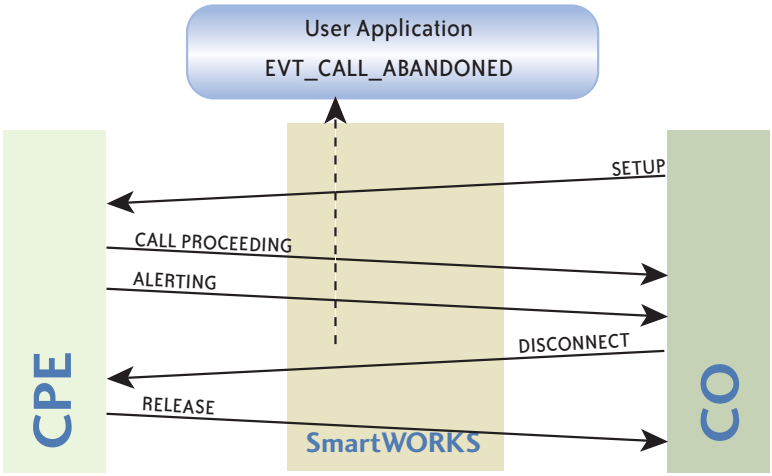
NETWORK INITIATED CALL CLEARING

The user application is notified of a network initiated call clearing request via EVT_CC_DISC_IND. Upon receipt of this event, the application considers the call released and the resources available. No application response is necessary for this event.

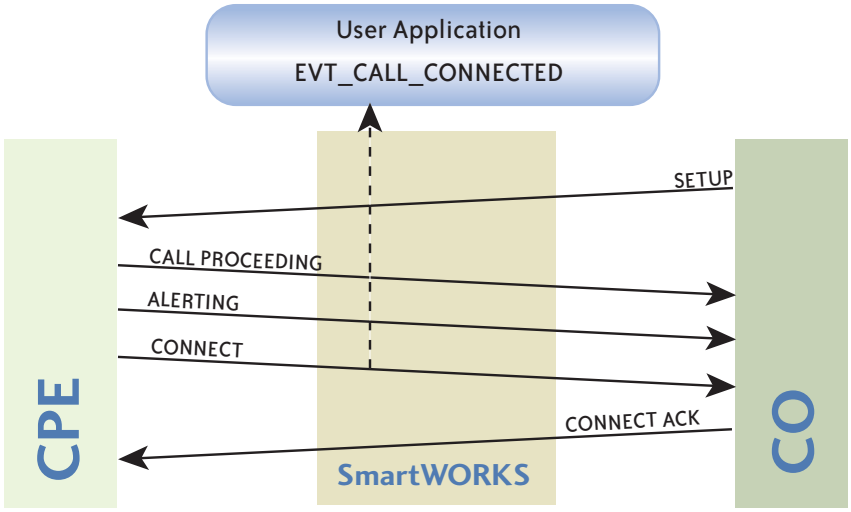
EVENTS GENERATED WHEN PASSIVE MONITORING

The following section provides examples of which events are generated by the passive call control system:

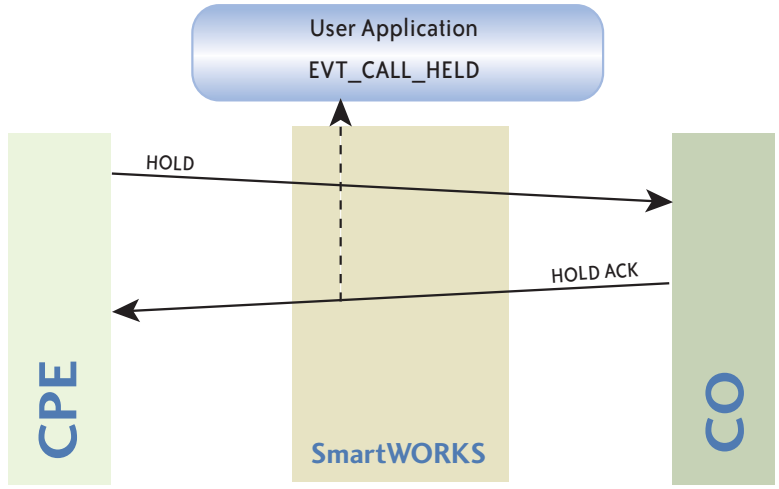
Call Abandoned



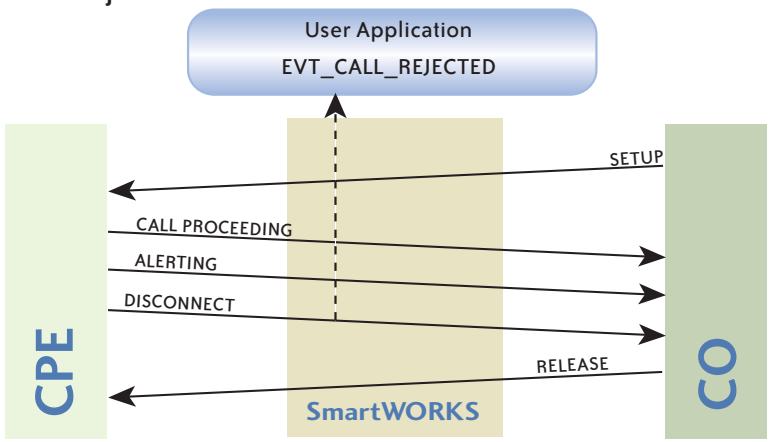
Call Connected



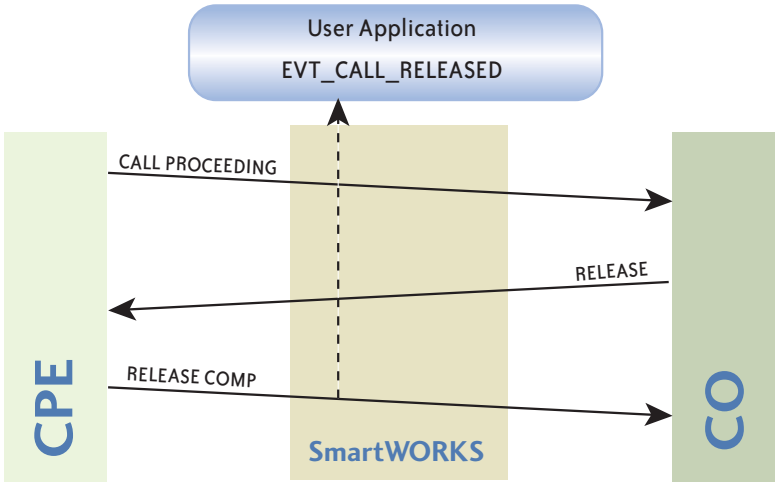
Call Held



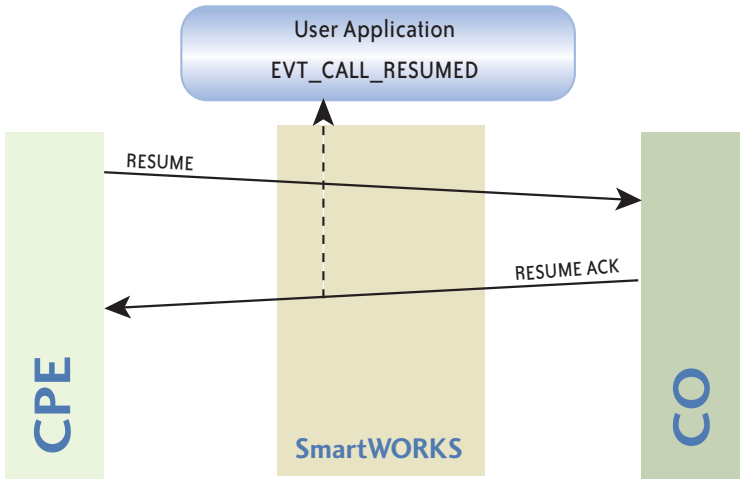
Call Rejected



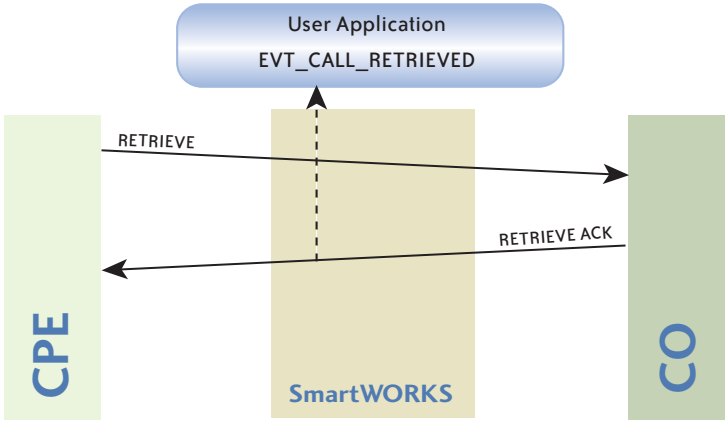
Call Released



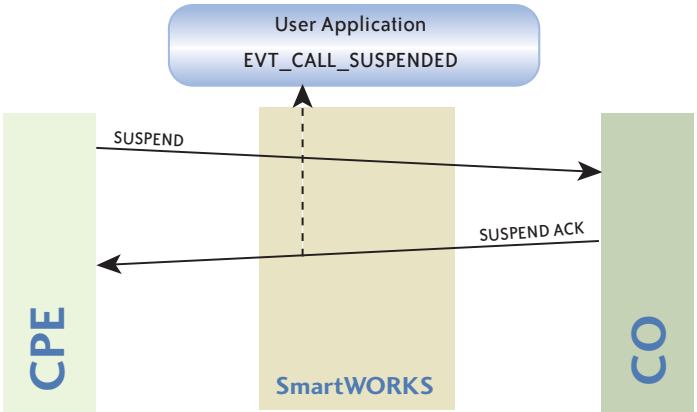
Call Resumed



Call Retrieved



Call Suspended



ISDN STANDARDS

TERMINATE CALL CONTROL. - SmartWORKS DT

ISDN PRI **T1** variants:

- NI-2 (North America)
- AT&T 5ESS (North America)
- Nortel DMS100 (North America)
- Euro-ISDN (Europe & Rest of the World)
- NTT Japan

ISDN PRI **E1** variants:

- Austel 1 (Australia)
- ETS 300
- QSIG (North America)

T1 RBS variants:

- E&M immediate
- E&M Wink
- Loop Start FXS
- Loop Start FXO

PASSIVE CALL CONTROL - SmartWORKS DP

- All ISDN protocols are supported as well as DPNSS, DASS2, and MFR2 (China, Brazil)

NOTE: The SmartWORKS NGX only supports the passive tapping of ISDN BRI.

SUPPLEMENTARY SERVICES FOR ISDN TERMINATE SUPPORT

Information elements are collected to provide the following supplementary services:

Call Hold /Call Retrieve	Call transfer
Call Forward on Busy	Charging
Call Forward unconditional	Recall
Call Forward on No Reply	Three party Conference
Suspend/Resume	Malicious Call ID

The supplementary services are provided through the following API structures:

- CC_FACILITY_REQ
- CC_FACILITY_CONF

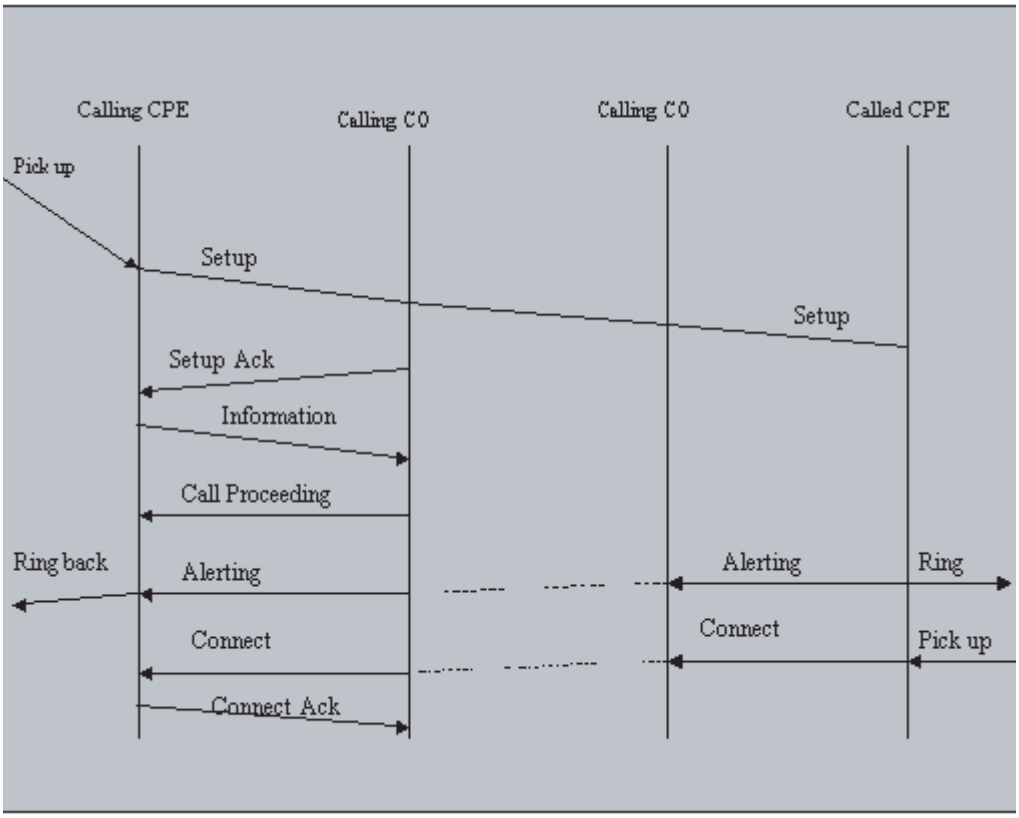
- CC_FACILITY_IND
- CC_CONNECT_REQ

NOTE:
The support of Supplementary services depends on ISDN variants

- Call Hold / Call Retrieve (valid for US variants only)
- Recall (valid for ETSI variant only)

Basic Call Setup

The following illustrates call setup as implemented on the DT card.



SmartWORKS RBS Signaling Protocols

The SmartWORKS DT supports Robbed Bit Signaling (RBS) protocols. Several variants of each protocol are also supported.

Robbed Bit Signaling

The Robbed Signaling protocol stack allows SmartWORKS DT cards to connect to T1 trunks using the following RBS variants:

- E&M Immediate Start
- E&M Wink Start

- Loop Start FXS
- Loop Start FXO

The E&M Immediate Start and Wink Start variants are symmetrical, meaning that the customer premises uses the same variant as the central office. The Loop Start FXS and FXO variants are not symmetrical, meaning that one side uses the FXS variant and the other side must use the FXO variant.

CONFIGURATION PARAMETERS

The variants mentioned above use Dual Tone Multi Frequency (DTMF) digits for dialing. Optimization of the call connection delay is provided by the following configuration parameters:

- Inter-digit Time
- DID Maximum number of digits

These two parameters are provided in the trunk configuration, and affect all channels on a trunk.

RBS IMPLEMENTATION CHARACTERISTICS

Incoming Calls

When the inbound side detects a line seizure (E&M Wink Start or E&M Immediate) or a ring indication (Loop Start), the RBS stack indicates a call presence to the application via the EVT_CC_CONNECT_IND event and starts a 10 second timer that listens for digits. If no digits are detected within 10 seconds, the RBS stack notifies the application that this call doesn't have a called party number via the EVT_CC_INFO_IND event. If the called party number is valid, the application accepts the request and establishes the call connection. If the called party number is invalid, the request is refused and a connection is not established.

RBS is an in-band signaling method that extracts the least significant bit of certain DS0 frames and uses them as signaling bits to monitor the ON-HOOK/OFF-HOOK status of the line. In the Wink Start variant, when a phone receiver is picked-up or goes *off-hook* by the calling party, the public switched telephone network (PSTN) verifies that it's ready to process a request (i.e. the called party digits) by going off-hook for approximately 200 ms, and then going back on-hook. This process is known as sending a *wink*.

When the called party digits are received, the called party goes off-hook and the call connection is established. Once the calling party hangs-up, the RBS stack notifies the application that the call connection is released at which point line resources are made available via the EVT_CC_DISC_IND event.

Outgoing Calls

In the Loop Start FXS variant, the protocol doesn't provide any information indicating that the called party has accepted the call. When the called party digits have been dialed, the RBS stack starts a 10 second timer. After 10 seconds expires, the RBS stack notifies the application that the call is connected via the EVT_CC_CONNECT_CONF event.

RBS protocol to Call Control API Mapping

This section shows how the SmartWORKS RBS implementation maps the Call Control API to each RBS protocol variant.

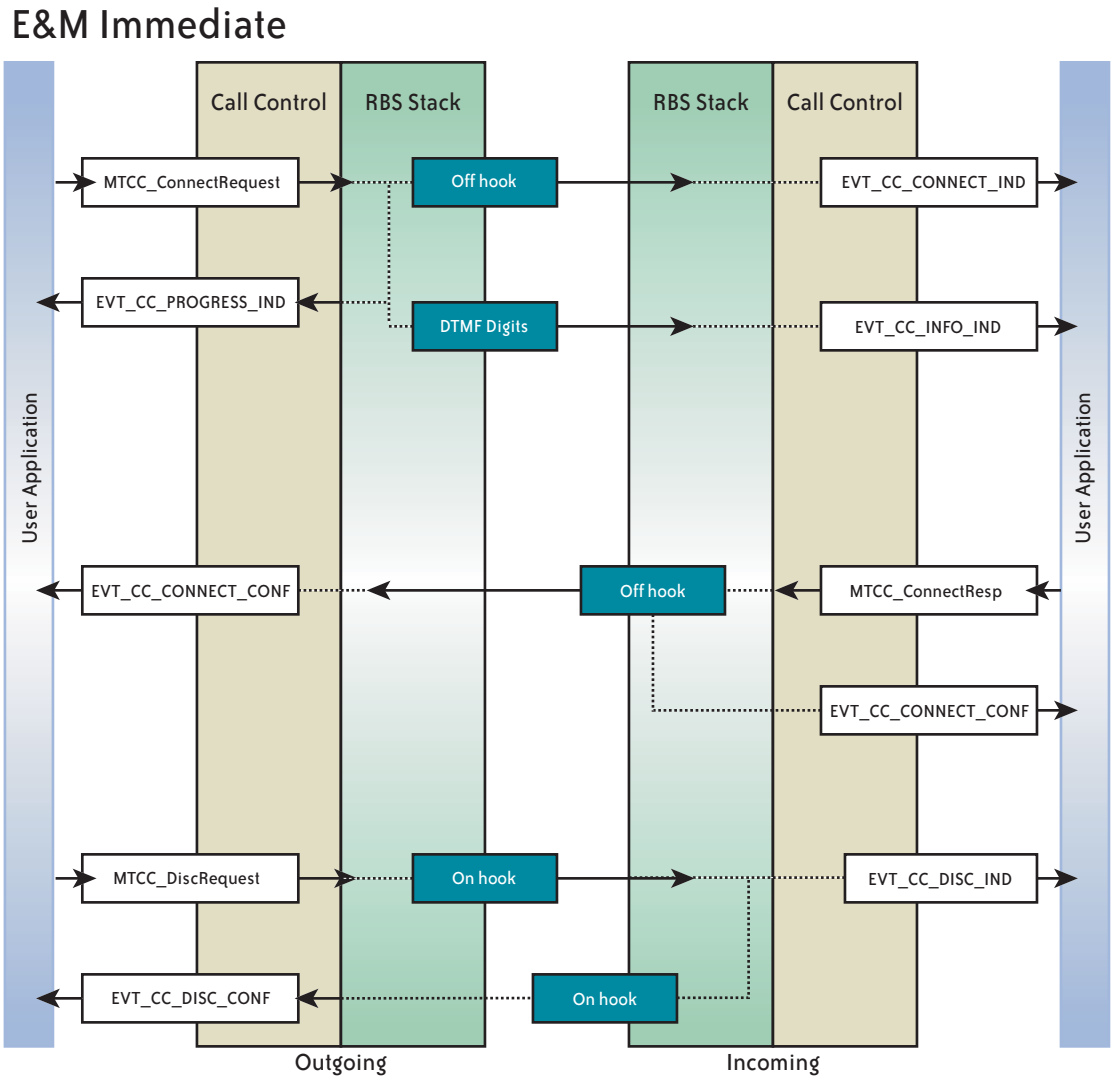


FIGURE 6: E&M IMMEDIATE

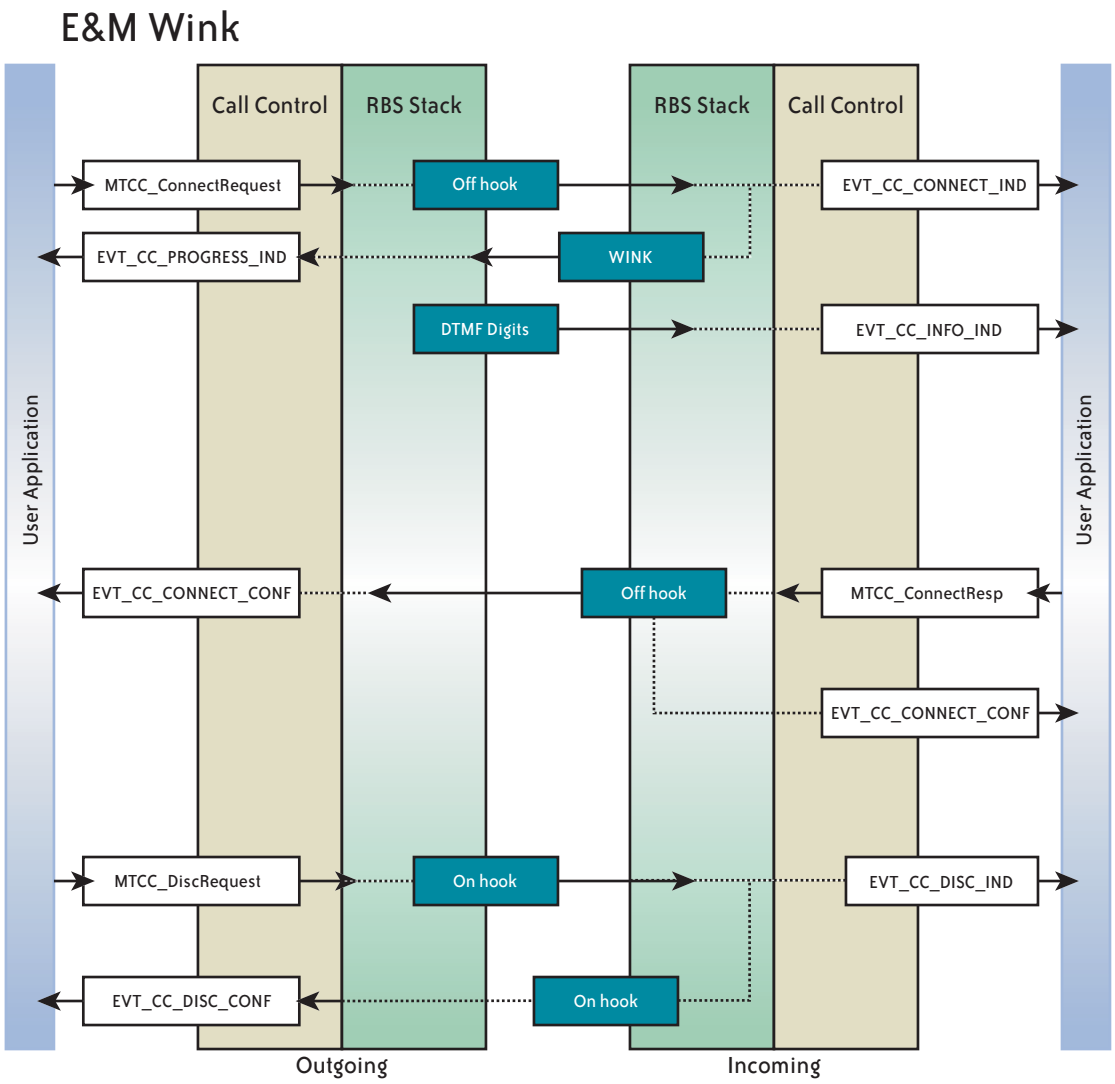


FIGURE 7: E&M WINK

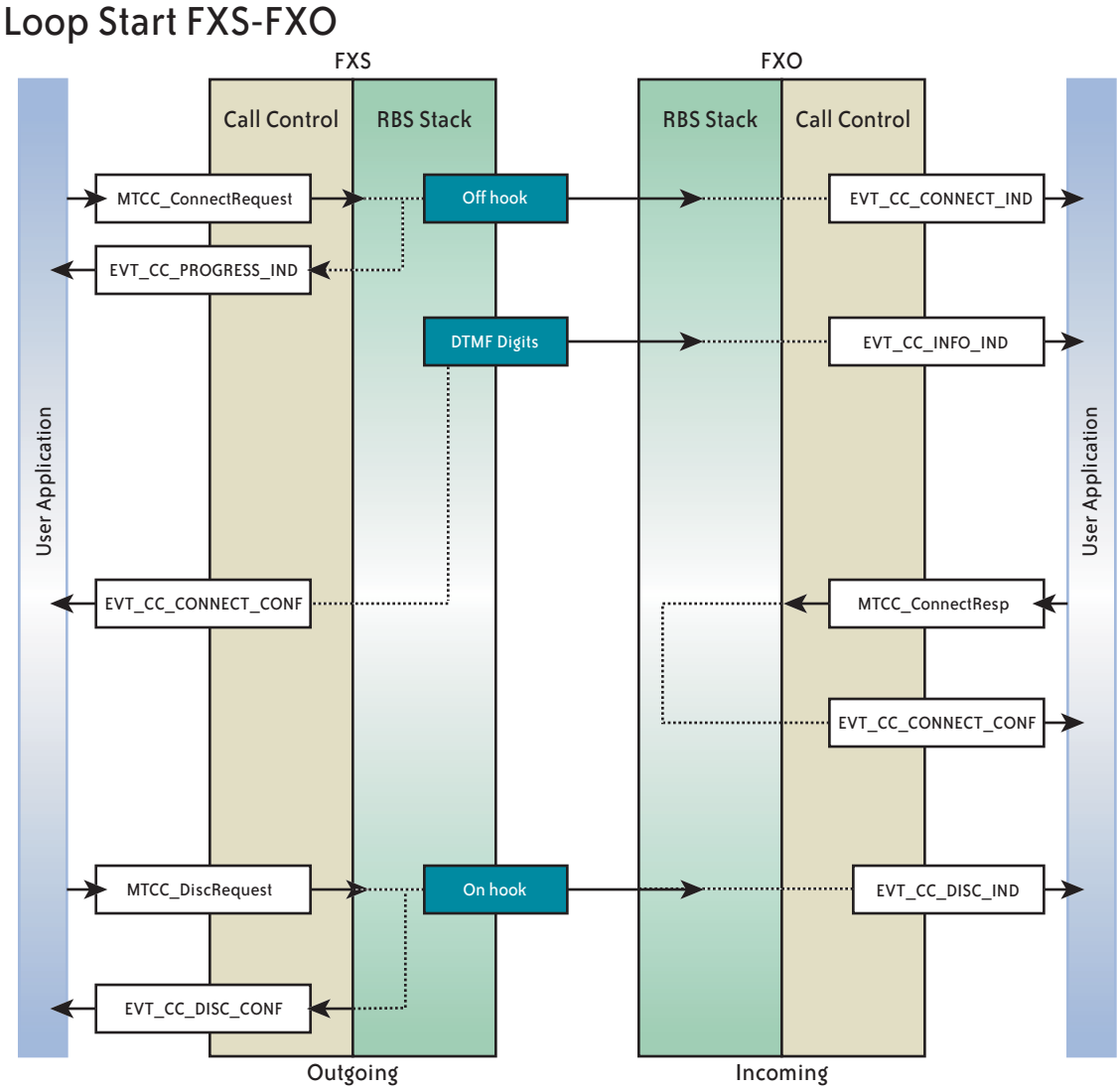


FIGURE 8: LOOP START FXS-FXO

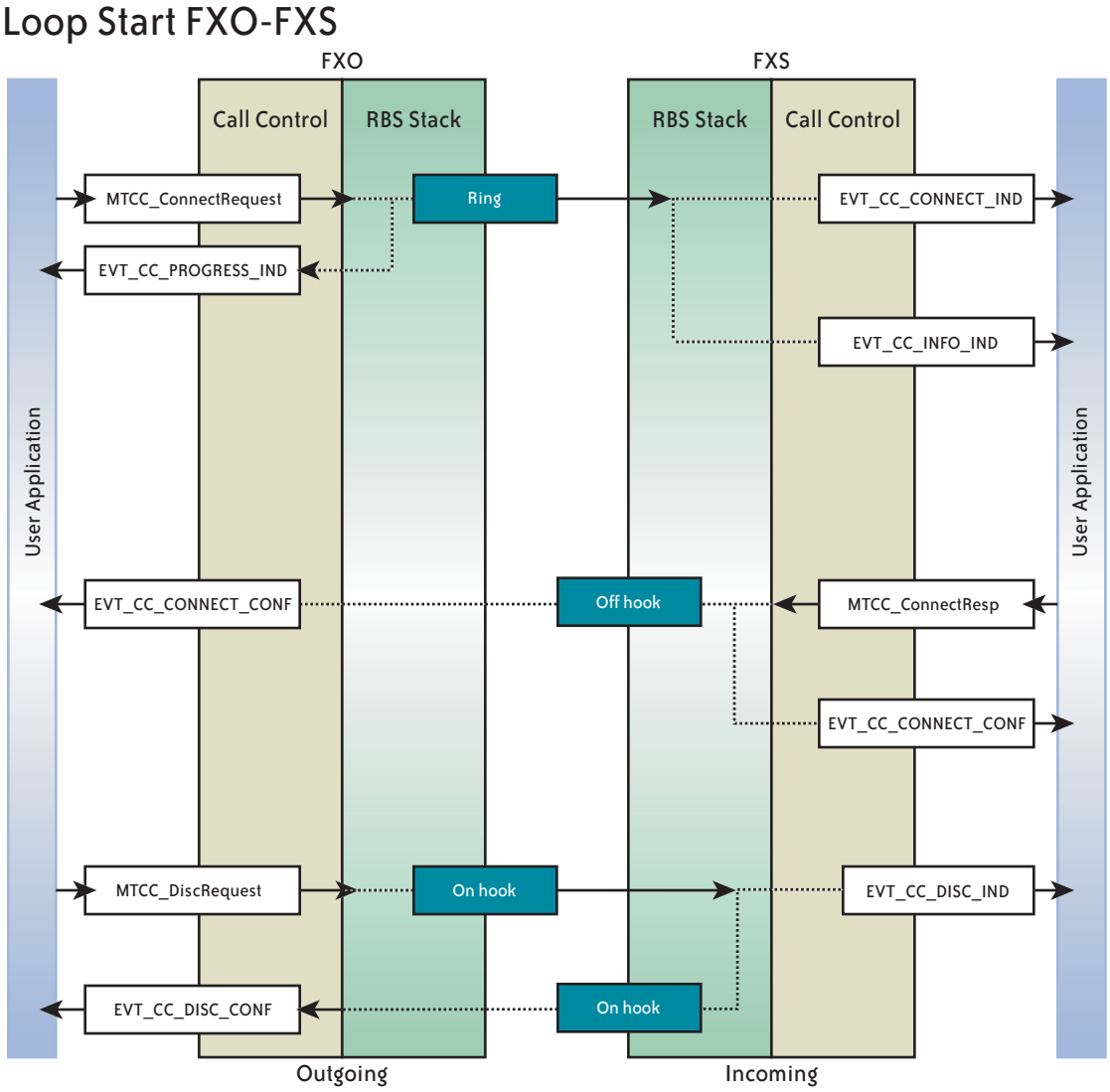


FIGURE 9: LOOP START FXO-FXS

NFAS SUPPORT

NFAS (Non Facility Associated Signaling) is a protocol which uses only one or two D channels while controlling calls on a larger number of trunks. In other words – the 64 kbps D channel (signaling channel) has enough capacity to control up to 14 T1 trunks. The immediate benefit of NFAS is that the number of B channels increases per system. Trunks which no longer use a single channel for D channel data now have a total of 24 B channels per trunk.

NFAS only makes sense in a T1 environment. E1 trunks have a dedicated timeslot for D channel signaling and therefore no saving can be accomplished by aggregating call control to one D channel.

NFAS is used to control T3 trunks (28 T1's). Implementing NFAS for PBX is beneficial only when the PBX supports more then one T1 trunk.

Passive Tapping NFAS

A full featured NFAS system has at least two T1 trunks coming from the Central Office (CO). These trunks are labeled from 0 to N. Trunk number 0 has timeslot 24 reserved for PRIMARY D channel. Trunk labeled 1 has timeslot 24 reserved for D CHANNEL BACKUP. Other trunks (labeled 2 to N) carry 24 B channels.

If the PRIMARY D channel fails the call control proceeds using D CHANNEL BACKUP. From time to time the CO may put PRIMARY D channel into maintenance mode and D CHANNEL BACKUP is used for call control. After maintenance is complete the call control is switched back to the PRIMARY D channel.

A simplified NFAS installation would use a small number of T1 trunks (usually two). The first trunk has 23 B channels and a PRIMARY D channel. If the D CHANNEL BACKUP is not used in this configuration all the remaining trunks will have 24 B channels.

When configuring the SmartWORKS DP for passive tapping, the Interface ID (or trunk ID) used by the local PBX, must match the Trunk Index ID on the SmartWORKS DP board. This enables the DP board to map D-channel events to the correct Channel ID. If this configuration is done incorrectly, the user application will receive the wrong Channel, Trunk and Timeslot information for all events.

NFAS group

The NFAS group is defined as a set of one or two trunks carrying D channel with trunks carrying associated B channels.

NFAS support under SmartWORKS

NFAS support is a standard feature when using the DPxx09 series of cards. Using SmartControl, users can define one or more NFAS groups. The recommended configuration consists of a single DP6409 supporting PRIMARY and BACKUP D channels. The trunk with PRIMARY D channel should be connected to the first framer, while the trunk with BACKUP D channel to the second framer on the same DP6409. When monitoring a NFAS group with both PRIMARY and BACKUP D channels, both trunks *must* be monitored by the same DP board. These trunks have to be monitored by the same protocol stack which knows about the state of each trunk.

Additional trunks (voice trunks) within the same NFAS groups may be supported by other DP6409's or DP3209's. A total of 15 trunks can be monitored by a single NFAS group; with trunk 0 monitoring the Primary D-Channel, trunk 1 monitoring the Back-Up D-Channel and trunks 2-14 used for voice only.

The next NFAS group should start with another DP6409 board.

NOTES:

- Using a back up D channel is optional, not all NFAS systems are configured with a backup D channel
- When both Primary and Backup D channel are used on a single NFAS system, both trunks *must* be monitored by the same DP board
- All DP boards monitoring a single NFAS group must be installed on the same host.

SmartWORKS Configuration

Two parameters are used to configure the T1 trunks when monitoring NFAS. When configuring a trunk - users must supply the Group ID of the monitored NFAS group - Index ID, and the type of NFAS trunk which is monitored - PRIMARY / BACKUP D channel or Voice only. All parameters are set with the ***MTSetAdapterConfig()*** function in the MT_ADAPTER_CONFIG data structure. **NOTE:** These parameters can also be set with the SmartWORKS Control Panel.

ProtocolSignaling[MAX_TRUNKS] - this array of values (one value per each trunk) identifies the signaling of the trunk. When monitoring an NFAS trunk the user relies on bit values to control whether the trunk is a PRIMARY or BACKUP D channel or voice

```
#define SIGNALING_NFAS          0x80    // NFAS for both T1
#define SIGNALING_NFAS_DCH      0x40    // Extension for NFAS definition
                                        // - bit 0x40 is for D-channel enable
#define SIGNALING_NFAS_BK_DCH   0x20    // Extension for NFAS definition
                                        // - bit 0x20 is for backup D-channel enable
#define SIGNALING_NFAS_TRUNK_BITS 0x0F  // - the lower nibble is for
unique trunk number within the NFAS indexed TrunkNFASId
```

Users must also specify the Group ID of the NFAS group. This is used in the bit field above and with another parameter in the MTADAPTER_CONFIG data structure:

NFASIndex[MAX_TRUNKS];

Configuration Example

In the following example, eight (8) trunks are used by the local network. Two separate NFAS groups are used by this network, therefore the PBX provides the following Trunk IDs:

NFAS group 0 consists of Trunk 0 that is used for Primary D-channel, while Trunk 1 is used for Back-Up. Trunks 2 & 3 carry voice data on all 24 timeslots.

The second NFAS Group - Group 1, is comprised of Trunk 0 which is used for Primary D-Channel, Trunk 1 which is used for Back-Up D-Channel while Trunks 2 & 3 carry voice data only.

To monitor this network - a total of four(4) DP6409s are used. Boards 0-1 monitor NFAS group 0 while Boards 2-3 monitor NFAS group 1.

The following configuration is required:

Board 0:

Signaling is set to NFAS.

NFAS Group / Index is set to 0

DP Trunk ID = 0, Trunk Index = 0, is set to NFAS type = Primary D-Channel

DP Trunk ID = 1, Trunk Index = 1, is set to NFAS type = Back Up D-Channel

Board 1:

Signaling is set to NFAS.

NFAS Group / Index is set to 0

DP Trunk ID = 0, Trunk Index = 2, is set to NFAS type = None

DP Trunk ID=1, Trunk Index = 3, is set to NFAS type = None

Board 2:

Signaling is set to NFAS.

NFAS Group / Index is set to 1

DP Trunk ID = 0, Trunk Index = 0, is set to NFAS type = Primary D-Channel

DP Trunk ID = 1, Trunk Index = 1, is set to NFAS type = Back Up D-Channel

Board 3:

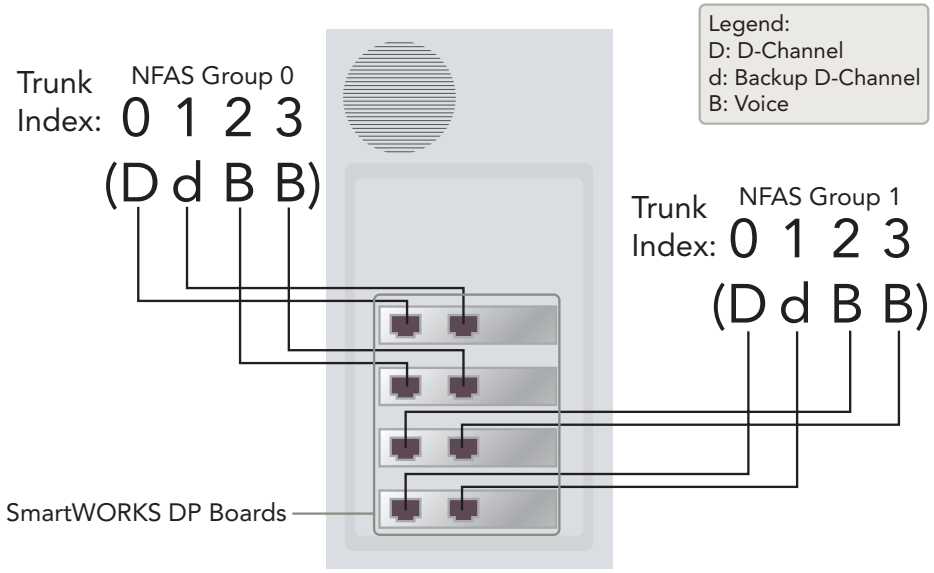
Signaling is set to NFAS.

NFAS Group / Index is set to 1

DP Trunk ID = 0, Trunk Index = 2, is set to NFAS type = None

DP Trunk ID = 1, Trunk Index = 3, is set to NFAS type = None

The following diagram illustrates how to configure multiple DP boards when monitoring a single system running two NFAS groups:



Channel Mapping

Once configuration is complete, the SmartWORKS DP board maps all D-channel events (call control events) to the correct Channel ID, Trunk ID and Timeslot ID. This information is passed to the user application when a call control event is reported. Before looking at event reporting, it is important to understand how SmartWORKS maps D-channel events to correct channel ID.

The following table is based off a network running two NFAS groups with 4 trunks per each group. The DP boards have been configured according to the example provided in the previous section of this document:

NOTE: The SmartWORKS DLL does not use a Channel ID for Timeslots with Primary or Back-UP Dchannel information. Voice channels are counted.

Table 10: NFAS Channel Mapping

DP Board ID	DP Trunk ID	NFAS Group	NFAS Trunk Index	Timeslot	Channel ID (GCI = 1)
0	0	0	0		
				1	1
				2	2
				3....	3....
				23	23
				24	(skipped)
0	1	0	1		
				1	24
				2	25
				3.....	26.....
				23	26
				24	skipped
1	0	0	2		
				1	47
				2	48
				3....	49....
				23	69
				24	70
1	1	0	3		
				1	71
				2	72
				3....	73....
				23	93
				24	94
2	0	1	0		
				1	1
				2	2
				3....	3....
				23	23
				24	(skipped)
2	1	1	1		
				1	24
				2	25
				3.....	26.....
				23	26
				24	skipped
3	0	1	2		
				1	47
				2	48
				3....	49....
3	1	1	3		
				1	71
				2	72
				3....	73....
				23	93
				24	94

Event Reporting

All call control events are reported as Channel events. When a call control event is reported, the Channel ID is passed to the user application in the *Channel* field of the MT_EVENT data structure.

All call control events are passed over with the MT_CC_CALL_INFO data structure. The *ChannelID* field contains a data structure that passes over the Trunk ID (NFAS Trunk Index), as well as the Timeslot number.

The following shows how a call control event is displayed in SmartView:

```
2006:10:17:15:12:39:934 Ch. 69  EVT_CC_CALL_ALERTING(6e)
Reason 0x0000, xInfo 0x17000000, Buffer: 0x011B6718,
DataLen: 328 Func 0x0--->CallRef=<24919>, Source=<2>
Trunk=<3>, Timeslot=<23> From <> To <5617024167>
```

Monitoring Select Trunks

At times, a network may be running a network with 10 trunks in a single NFAS group, however, the logger is only required to monitor 3 of the 10 trunks.

For example, the network is designed with a single NFAS group comprised of ten(10) trunks; numbered 0-9 respectively. Trunk 0 is used for Primary D-channel while Trunk 1 is used for Back-Up. Trunks 2-9 are only transmitting voice data.

The logger is only required to tap phones connected to trunks 4, 6 and 9. In this scenario five(5) trunks must be connected to the DP boards. Trunks 0 & 1 must be connected in order to monitor the signaling packets and trunks 4, 6 and 9 for the voice data.

Though only five(5) trunks are connected to DP boards, the NFAS Trunk Index of these trunks must still match the trunk Index or Interface ID used by the PBX to manage these NFAS trunks. The following table illustrates this concept:

Table 11: Monitoring Select Trunks

SmartWORK Board ID	DP Trunk ID	NFAS Group ID	NFAS Trunk Index
0	0	0	0
0	1	0	1
1	0	0	4
1	1	0	6
2	0	0	9

Passive ISDN Functions

These APIs are used on the SmartWORKS DP, and the SmartWORKS NGX cards to control passive ISDN capabilities.

NOTE: The SmartWORKS DP card supports ISDN PRI while the SmartWORKS NGX supports ISDN BRI.

The following API's are supported on the passive SmartWORKS DP cards for ISDN support.

The interface to the user application is a two-way interaction:

1 · Call Control indications from the board to the user application

2 · Application requests to the board for information about call Sessions

CALL CONTROL INDICATIONS

When configured for ISDN PRI, BRI or NFAS the following call control events are generated:

- EVT_CALL_ABANDONED
- EVT_CC_CALL_IN_PROGRESS
- EVT_CALL_CONNECTED
- EVT_CALL_HELD
- EVT_CALL_REJECTED
- EVT_CALL_RELEASED
- EVT_CALL_RESUMED
- EVT_CALL_RETRIEVED
- EVT_CALL_SUSPENDED

When configured for ISDN DASS2 or DPNSS the following call control events are generated:

- EVT_CALL_ABANDONED
- EVT_CALL_CONNECTED
- EVT_CALL_REJECTED
- EVT_CALL_RELEASED

All these indications are events in the SmartWORKS MT_EVENT structure. When a channel receives one of these events, the MT_EVENT fields are set as follows (the subreason field is used for group call on Integral PBXs - see below) :

1 · The *TimeStamp* field of the MT_EVENT structure indicates a time stamp of when the event occurred

2 · The *EventCode* field indicates the CC event code value as defined in the file NtiEvent.h

3 · The *channel* field indicates the channel on which the event happened

4 · The *PtrBuffer* field carries a pointer to an MT_CC_CALL_INFO structure containing all the available information about the call. This structure is defined in the file NtiDataCC.h

5 · The *DataLength* field contains the size of the MT_CC_CALL_INFO structure

Application Notes -Group Call Information

For the Integral 33/55 two-wire PBX, information is now processed for a group call situation.

Call Reference: ALL group calls have the same Call Reference as 0xE4 and the Call Reference of the ALL real calls MUST be less than 0x80. Use these values to distinguish real call events and group call events.

SubReason: The SubReason in ALL real calls events is 0x00.

The SubReason in ALL group calls events is the group number plus 0x100.

Use the subreason field to obtain group number.

Application requests

The User application may issue requests to retrieve statistics or information about a channel or a call session. This can be done through the following API functions:

- MTCC_GetStatusByChan()
- MTCC_GetStatusByRef()

Note: Events prefixed with EVT_CC_ typically require user applications to call MTReturnEventBuffer() passing in MT_EVENT.ptrBuffer to free the allocated memory. See the Functional Reference Library where MTReturnEventBuffer() is explained for more information.

MT_CC_CALL_INFO Structure

The MT_CC_CALL_INFO structure is relied on by most APIs used for passive ISDN.

The MT_CC_CALL_INFO structure is defined in the NtiDataCC.h file and contains the fields below:.

Table 12: MT_CC_CALL_INFO

Type	**Name**	**Function**
ULONG	CallRef	A unique value assigned by the user's application for each call.
ULONG	CallSource	Only 2 values are possible MT_CC_INCOMING_CALL or MT_CC_OUTGOING_CALL
ULONG	CallState	State of the call. Call states are defined in the NtiDataCC.h file. MT_CC_CAUSE values are used for ISDN while MT_CC_DASS values are used for DASS2 and DPNSS.
ULONG	CallTrunk	The trunk where the call occurred.
ULONG	CallDuration	The total length of the call
ULONG	Layer1Coding	Layer1 coding on B channel, defined in NtiDataCC.h file.

TABLE 12: MT_CC_CALL_INFO

Type	**Name**	**Function**
ULONG	Cause	The cause of the last release. Cause types are defined in the NtiDataCC.h file.
MT_CC_CHANNEL_ID	ChannelId	This structure is defined on the next pages.
MT_CC_PARTY_NUMBER	CallerNumber	This structure is defined on the next pages. If no number has been obtained, the number of digits field = 0.
MT_CC_PARTY_SUBADDR	CallerSubAddr	This structure is defined on the next pages. If no number has been obtained, the number of digits field = 0.
MT_CC_PARTY_NUMBER	CalledNumber	This structure is defined on the next pages. If no number has been obtained, the number of digits field = 0.
MT_CC_PARTY_SUBADDR	CalledSubAddr	This structure is defined on the next pages. If no number has been obtained, the number of digits field = 0.
MT_CC_PARTY_NUMBER	ConnectedNumber	This structure is defined on the next pages. If no number has been obtained, the number of digits field = 0.
MT_CC_PARTY_SUBADDR	ConnectedSubAddr	This structure is defined on the next pages. If no number has been obtained, the number of digits field = 0.
MT_CC_PARTY_NUMBER	RedirectingNumber	This structure is defined on the next pages. If no number has been obtained, the number of digits field = 0.
MT_CC_CALL_IDENTITY	CallIdentity	This structure is defined on the next pages.

CHANNEL IDENTIFICATION STRUCTURE

Table 13: MT_CC_CHANNEL_ID

Type	**Name**	**Function**
int	Pref_Excl	Preferred or Exclusive
int	InterfaceId	Trunk Number
int	TimeSlot	Time slot within the trunk

The *Pref_Excl* field is only for terminate products (SmartWORKS).

The *InterfaceId* field provides the trunk number where the B-channel resides.

The *TimeSlot* field provides the TDM time slot that is mapped to the SmartWORKS DP resource channel.

PARTY NUMBER STRUCTURE

This structure is used to indicate the *calling party number* (also called *caller number*), the *called party number* and the *connected number.*

Table 14: MT_CC_PARTY_NUMBER*

Type	**Name**	**Function**
int	TypeOfNumber*	Numbering Type :UNKNOWN, NATIONAL, SUBSCRIBER or ABBREVIATED
int	NumberingPlan*	UNKNOWN, NATIONAL, PRIVATE_PLAN, DATA_PLAN, TELEX_PLAN
int	NumberOfDigits*	Gives the size of the digits field. This field varies from 0 to MAX_PARTY_DIGITS, which is 32
UCHAR	Digits[MT_CC_MAX_PARTY_DIGITS]*	The called number digits

* When the board is configured for DASS2 and DPNSS: When this data structure is used to report *ConnectedNumber* or *RedirectingNumber* all fields are set to '0'. When this data structure is used to report *CallerNumber*, and *CalledNumber* only the *NumberofDigits* and *Digits* fields are populated.

NOTE: The NumberingPlan and the NumberOfDigits fields are defined in the NtiDataCC.h file.

SUB_ADDRESS STRUCTURE

This structure is used to indicate the *calling party* sub-address, the *called party* sub-address and the *connected* sub-address.

NOTE: This data structure is not used when the board is configured for DASS2 or DPNSS (all fields are set to '0')

Table 15: MT_CC_PARTY_SUBADDR

Type	**Name**	**Function**
int	NumberOfDigits	The number of digits in called number. This field varies from 0 to MAX_PARTY_DIGITS, which is 32
int	SubAddrType	See below.
int	OddEvenInd	See below.
UCHAR	Digits[MT_CC_MAX_SUBADDR_DIGITS]	The called number digits

SUBADDRTYPE

This field indicates the coding format used for the sub-address. The valid values for this field are:

MT_CC_SUBADDR_NSAPNSAP /x213 format

MT_CC_SUBADDR_USERUSER Specified format

ODDEVENIND

The Odd/Even indicator field declares whether the number of digits is odd or even. It is used when a byte is carrying two digits. The valid values for this field are:

MT_CC_SUBADDR_EVENnumber of digits is odd

MT_CC_SUBADDR_ODDnumber of digits is even

CALL IDENTITY STRUCTURE

Table 16: MT_CC_CALL_IDENTITY*

Type	**Name**	**Function**
int	Length*	The length of the call identity string
UCHAR	Identity[MT_CC_MAX_IDENTITY_SIZE*	The call identity string

* These fields are not used when the board is configured for DPNSS or DASS2.

Channel Functions

The SmartWORKS SDK supports a total of 512 channels. All channels within AudioCodes boards are numbered using GCI indexing sequentially from 0 or1 to the number of available channels. The GCI starting value (0 or 1) is controlled by the user via the SmartControl panel applet.

CALLERID CONTROL

SmartWORKS boards enable caller ID detection by default. How CallerID control is managed depends on the type of network:

Analog CallerID Control

All SmartWORKS boards can detect in-band CallerID with support for Bell 202 and V.23 standards. This feature is typically used only on analog networks where SmartWORKS LD boards are deployed.

There are two events that support the caller ID function: EVT_CALLID_DROPPED and EVT_CALLID_STOP. The EVT_CALLID_STOP event notifies the application that the SmartWORKS board has finished retrieving Caller ID data and the received data is successfully queued. The SubReason field of event EVT_CALLED_STOP contains the length of the caller ID packet, XtraInfo field contains the first 4 bytes of the caller ID data packet. Each channel is allowed to queue up to 15 Caller IDs. EVT_CALLID_DROPPED indicates that good Caller ID information is dropped due to a full queue.

Each channel can queue up to 15 caller ID packets. Use the API function, ***MTGetCallerID()*** to retrieve Caller ID information.

TABLE 17: CALLERID CONTROL

MTDisableCallerID()
MTEnableCallerID()
MTFlushCallerID()
MTGetCallerID()
MTGetCallerIDStatus()

ISDN CallerID Control

ISDN networks pass CallerID information in Dchannel messages and can be decoded by SmartWORKS DT, SmartWORKS DP, and SmartWORKS NGX (when tapping ISDN BRI networks only). All CallerID information is passed to the user application when call state events are reported.

The SmartWORKS DT reports call state with confirmation and indication events. Each call control event passes a unique data structure to the user application. When the EVT_CC_CONNECTED_IND event is reported, the MT_CC_CONNECT_IND data structure populates two fields: *CalledPartyAddr* and *CallingPartyAddr*.

The SmartWORKS DP and NGX boards work passively. Call state is reported to the user application via call control events. Each event passes over the MT_CALL_INFO data structure. Two fields are used that indicate CallerID: *CallerNumber* and *CalledNumber*.

To obtain CallerID information use ***MTSetEventCallback()*** or ***MTSetBoardEventCallback()***. ISDN events are reported as channel events on SmartWORKS DP, SmartWORKS NGX, plus SmartWORKS DT cards configured for RBS. Otherwise, all ISDN events reported on the SmartWORKS DT card are reported as board events.

Proprietary Dchannel

When using the SmartWORKS NGX and IPX to tap proprietary PBX environments, the callerID is passed on the line when the PBX relies on the phone to display the DNIS number. When Dchannel decoding is supported, both boards are able to provide callerID on systems where the phones have LCD displays, and the callerID is displayed on the phone. When the phone's LCD is updated, the EVT_MESSAGE_CHANGE is reported and the callerID can be parsed from the buffer. In order to know which EVT_MESSAGE_CHANGE event contains the callerID information, the user must observe the behavior of the phone and PBX in order to learn the event reporting sequence. Refer to the *NGX Integration Guide* and the *IPX Integration Guide* for more information on specific PBX models.

On some networks, the IPX board also provides call control events. When these events are reported the MT_CALL_INFO data structure contains the *CalledNumber* and *CallingNumber* if available. (Refer to the *IPX Integration Guide* for details).

DTMF/MF and Tone Control

SmartWORKS boards are equipped to send and receive DTMF digits, handle incoming digits, and dial numbers.

Dialing a string is accomplished by passing a string with valid DTMF digits or control characters and invoking one of the dial functions. Valid digits are 0 - 9 and ABCD#*. Control characters are as follows:

& - Generates a hook flash (SmartWORKS LD only)

, (comma) - Inserts a pause

T - Switches back to DTMF dialing mode

W - Signals that the SmartWORKS board must wait for a dial tone before dialing the remainder of the string. (Call Progess Monitoring must be enabled: ***MTCPMControl()***.

H - Take the line OFF_HOOK if it is ON_HOOK (SmartWORKS LD only)

The function ***MTDialString()*** is used to dial a string of digits. **MTCallString()** is a background function that automatically calls a number. This function takes the channel off hook, waits for dial tone, dials the number, and then determines if the call is answered.

When a SmartWORKS board detects a DTMF digit, the EVT_DIGIT event is issued with the digit information in the SubReason field. The digit is then stored in a separate queue (per channel) called the DTMF queue. A digit stays in the queue until the application discards or retrieves it. If the application never issues a DTMF retrieval call [***MTReadDigit()***] the SmartWORKS API will continue to store all detected digits until the DTMF queue for that channel is full. The DTMF queue is 65 digits deep for each channel. Should a received digit be dropped due to a full queue the EVT_DIGIT_DROPPED event is issued.

The same applies to AudioCodes boards that support MF tones. Events EVT_MFTONE and EVT_MFTONE_DROPPED report MF tone detection and queuing. SmartWORKS boards have the capability to differentiate whether the DTMF or MF tone is detected on the primary input or the attached secondary input. This information is also queued along with the tone. Functions ***MTReadDTMFTone()*** and ***MTReadMFTone()*** retrieve the queued tone along with the channel information.

MTClearDTMFDigits() clears the DTMF queue while ***MTClearMFTones()*** clears the MF queue. ***MTReadDigit()***, ***MTReadDTMFTone()***, and ***MTReadMFTone()*** return the next digit/tone in the queue, if there is one.

MTGetDigits() is a background function that receives a string of digits. The application must pass the address of a buffer where the received digits can be stored. The required MT_IO_CONTROL structure specifies the termination conditions. Possible termination conditions are a maximum number of digits, a certain termination digit, exceeding a time limit, exceeding a time limit between digits, etc. This function can terminate immediately if a specified termination digit or number of digits are already in the queue.

Though DTMF detection, event reporting, and queuing are on by default, it's possible to reconfigure this setting through function ***MTControlToneQ() or MTChInputToneDetectControl()*** which provides an on/off for tone detection and queuing.

Automatic Gain Control

Each channel is equipped with an automatic gain control (AGC) function, which is disabled by default. Use ***MTSetAGC()*** or ***MTChInputSetAGC()*** to equalize incoming signals, then ***MTSetGain()*** can be used to adjust to the final desired level. ***MTSetGain()*** is an optional step. The APIs - ***MTSetAGC()*** and ***MTSetGain()*** are applied after summation. Should mixing be disabled, only the primary input is affected by these APIs. To control gain on both inputs, use the ***MTChInputSetAGC()*** or ***MTChInputSetGain()*** functions.

Automatic Gain Control

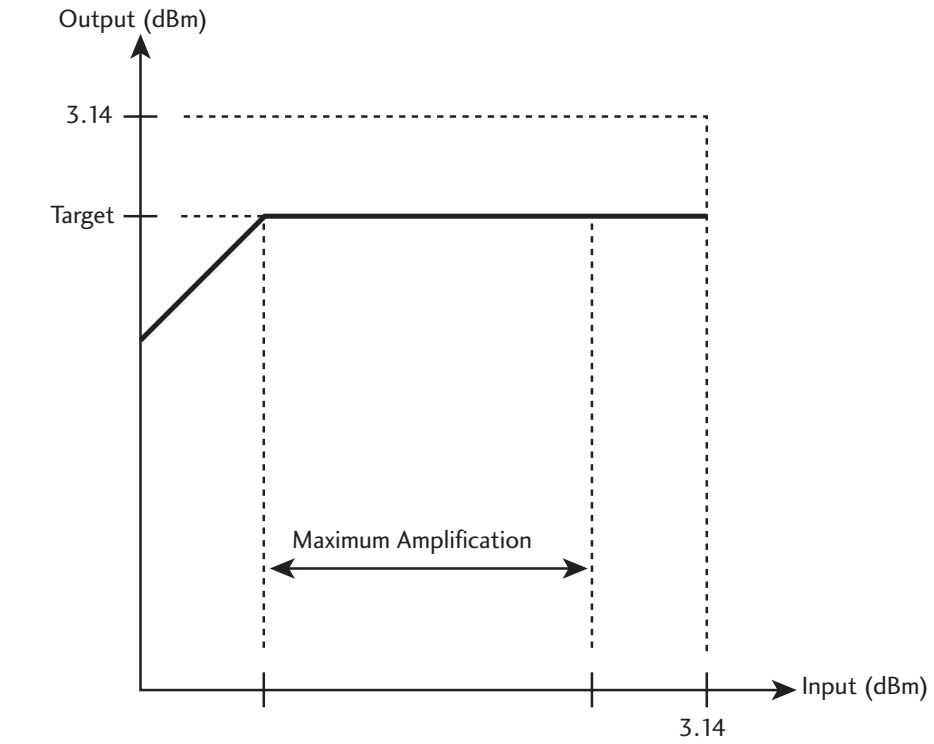
The operation of AGC is controlled by four parameters:

- Target amplitude
- Maximum Amplification (MA)
- Attack slew rate

- Decay slew rate

The static transfer characteristics of AGC is shown below.

Figure 5·1: AGC Static Transfer Characteristics



In steady state condition signal with power above the target will be attenuated to reach target power. Signals with energy lower then target will be amplified to reach the target level. The amount of amplification is limited by a parameter MA.

MA is programmable from the API in 6 dB steps.

The rate at which the gain changes is controlled by two parameters: attack slew rate and decay slew rate.

Attack controls the speed of the gain decrease, decay controls the speed of the gain increase. The attack and decay are programmable from the API in units of 0.00212 dB/millisecond. Changing the Attack and Decay settings are not recommended.

The default settings for AGC are:

Target amplitude (TMA)

-6 dBm

Maximum Amplification (MA)

30 dB

Attack

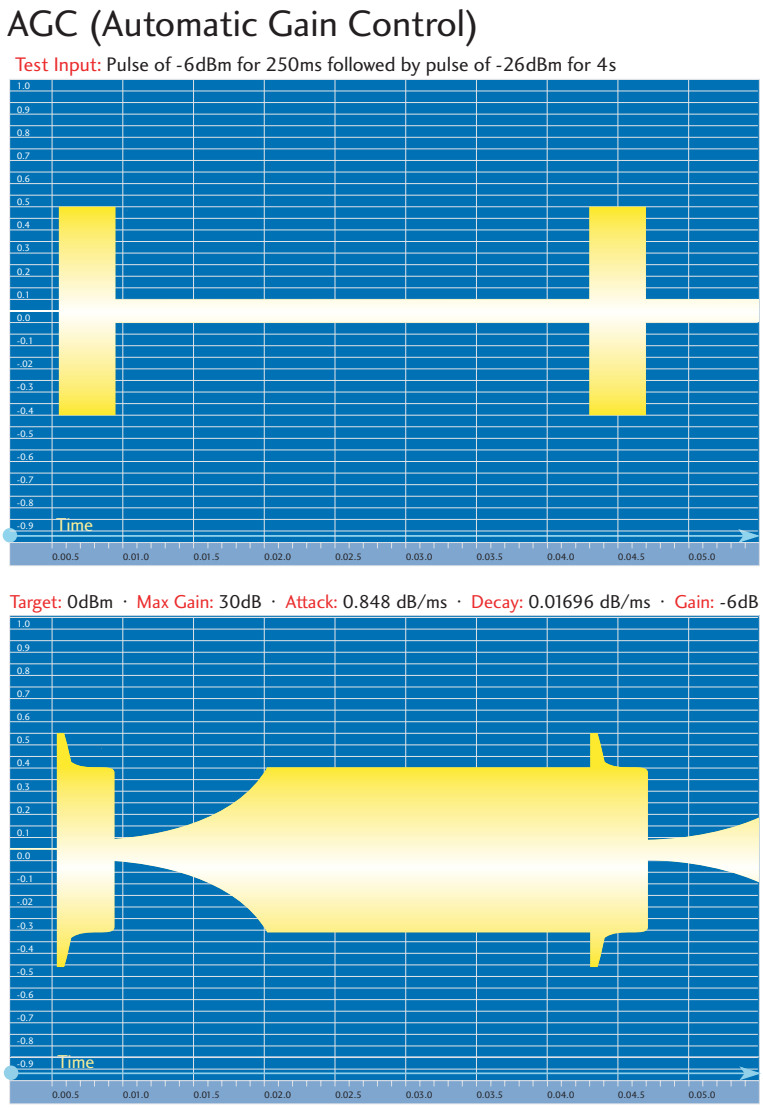
400

Decay

4

The following figure shows the effect of AGC on a test signal.

Figure 5-2: AGC Results



Recommended Gain Settings

The following table shows recommended AGC settings for both phone (digital and analog) and radio networks.

Table 18: Recommended Gain Settings

Type of Network	Target Amplitude	Attack	Decay	Maximum Amplitude
phone	-6	400	4	30

TABLE 18: RECOMMENDED GAIN SETTINGS

Type of Network	Target Amplitude	Attack	Decay	Maximum Amplitude
radio	-6	400	1*	30

*A small decay is required on a radio network to minimize squelch.

MTSetGain() can be used to adjust the equalized audio level. If **MTEnableMixing()** is enabled, this modifies the level of audio after the incoming signal on both the primary and secondary inputs have been combined.

The APIs - ***MTSetAGC()*** and ***MTSetGain()*** are applied after summation. Should mixing be disabled, only the primary input is affected by these APIs. To control gain on both inputs, use the ***MTChInputSetAGC()*** or ***MTChInputSetGain()*** functions.

NOTE: MTSetGain() and MTSetAGC() are not recommended when stereo recording. Should these functions be used, only the primary side will be affected. Use MTChInputGain() or MTChInputSetAGC() when stereo recording.

MORE INFORMATION ABOUT AGC

Target Maximum Amplification:

Signals lower than the target level will be amplified and signals higher will be attenuated.

Maximum Amplification:

The value set will amplify the signal by the MA.

1. If Signal Power + Maximum Amplification equals the Target then the Output Power will be Target.

2. If Signal Power + Maximum Amplification is greater than the Target, then the Output Power will be Target.

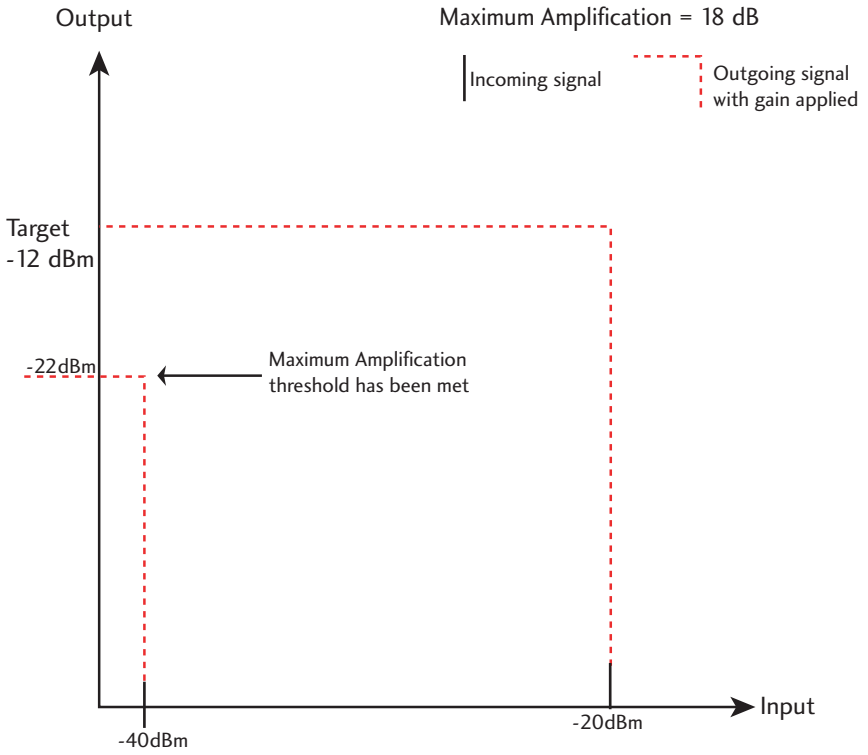
3. If Signal Power + Maximum Amplification is less than the Target, then the Output Power will equal the Signal Power + Max Amplification which is less than Target.

REDUCING BACKGROUND NOISE

At times background noise appears on the line when both parties are silent. Background noise may be described as a loud distortion or hissing that is present in the recording when both parties are silent. Modifying AGC parameters can reduce background noise. Background noise is typically at a low energy level. Reducing the *Maximum Amplification* parameter limits the allowable gain applied to low level signals. As a result low energy signals, such as background noise, will not be amplified to the level defined by the *Target Amplitude* parameter. The following diagram illustrates this point:

Figure 5·1: Reducing Background Noise

Reducing Background Noise



To effectivly apply gain to voice levels, but reduce background levels the following settings are recommended:

Target amplitude (TMA)

-6

Maximum Amplification (MA)

18 dB

Background noise is present on the line at a low signal level. By reducing the amount of allowable maximum gain, low level signals will not be amplified to the level defined by the Target Amplitude parameter.

Attack

400

Decay

4

Use **MTSetGain()** to adjust the equalized audio level to the dB required by your application.

NOTE: This function is not recommended when stereo recording. Should this function be used, only the primary side will be affected. Use MTChInputGain() when stereo recording.

Activity Detection Control

Originally Activity Detection was designed to work in the following way:

- activity detection, when enabled with **MTEnableACTD()**, only worked on the primary input of the channel. **MTEnableMixing()** then **MTEnableMixingDetect()** was used to enable activity detection on the secondary channel. The API **SetACTDParams()** configured activity detection parameters, but the same values were used for both inputs.

The APIs used to control activity detection were recently enhanced to allow for configuration on a per input basis.

Users can still control activity parameters with the API **MTSetACTDParams()**. However, this method is only recommended when activity detection configuration is the same on both inputs.

To control activity detection on the primary input, do the following:

- **MTChInputACTDControl()** - enable / disable activity detection

- **MTChInputSetACTDParams()** or **MTSetACTDParams()** - configures activity detection parameters

To control activity detection on both inputs, do the following:

- **MTEnableMixing()** - combine the primary and secondary inputs

- **MTChInputACTDControl()** - enables / disables activity detection (per input). **MTEnableMixingDetect()** will be obsoleted in Jan. 2005.

- **MTChInputSetACTDParams()** - configures activity detection parameters on both the primary and secondary inputs (NOTE: **MTSetACTD()** can still be used, but the configuration values will be the same for both the primary and secondary inputs).

Global Channel Index Functions

Board and Channel Numbering

The SmartWORKS API supports up to 16 physical boards and/or up to 256 full duplex channels within a system. The API functions refer to a specific board and or channel within the system using one of two numbering schemes: physical board numbers, and Global Channel Index (logic channel numbers). All board numbers are assigned sequentially starting from zero. Channel numbers are assigned sequentially starting from either 0 or 1 (depending on how the user has configured this setting in the Smart Control panel). **NOTE:** The IPX board does not impact channel numbering as this board does not open with channels.

Certain API functions will allow the developer to reference all boards simultaneously by using the nBoard = -1.

Global Channel Index

During initialization, as the Physical Boards are numbered, the driver will build a list of the logical channels available in the system called the Global Channel Index (GCI). This list is the primary interface the API will use to refer to the channel resources in the system.

The Global Channel Index (GCI) is numbered sequentially from 0 or 1 (depending on how the user has configured this setting in the Smart Control panel) and is in ascending order of the Physical Board numbers. The maximum GCI is currently limited to 512.

Certain API functions will allow the developer to reference all channels simultaneously by using the nChannel = -1 (if GCI index = 0) or nChannel = 0 (if the GCI index = 1).

For Example:

Function ***MTSetEventCallback()*** takes channel number -1 or 0, and registers the callback function for all available channels

GCI Functions

The API has a several commands that can be used to determine the relationship between the GCI and the physical channels on each board. The ***MTGetGCI()*** and ***MTGetGCIMap()*** command will match a GCI indexed channel to its physical board channel location.

For Example:

If MTGetGCIMap(65, pBOARD, pBOARDTYPE, pGCI) returns with *pBOARD=0, and

*pGCI=0, this indicates GCI channel 65 resides on board 0 as its first channel. However, MTGetGCI(0,0,pGCI) should return with *pGCI=65.

Media (IO Control) Functions

This group of APIs allows the user application to specify its own device read, write, and seek functionality for reading and writing recorded voice data. This gives the user application the ability to integrate the SmartWORKS API with a custom device that supports the Microsoft Windows device interface such as: open(), close(), read(), write() and seek().

NOTE: The MT_IO_CONTROL data structure is explained, in detail, in the SmartWORKS Function Reference Library.

MTSetDeviceIO() sets the user device I/O entries within SmartWORKS. The user device I/O is enabled through ***MTPlayDevice(), MTPlayDeviceIndex()*** and ***MTRecDevice()*** with file handle set within MT_IO_CONTRL.FileHandle field. It is the user application's responsibility to open associated file before enabling the SmartWORKS media function to access the file through device IO API functions.

Following is a list of Media Control API functions:

TABLE 6: MEDIA CONTROL

MTGetStreamingConfig()
MTPlayBuffer()
MTPlayBufferAsync()
MTPlayBufferEx()
MTPlayDevice()
MTPlayDeviceAsync()
MTPlayDeviceEx()
MTPlayFile()
MTPlayFileAsync()
MTPlayFileEx()
MTPlayIndex()
MTRecBuffer()
MTRecBufferAsync()
MTRecBufferEx()
MTRecDevice()
MTRecDeviceAsync()
MTRecDeviceEx()
MTRecFile()
MTRecFileAsync()
MTRecFileEx()
MTSetDeviceIO()
MTSetStreamingConfig()
MTStartStreaming()
MTStopStreaming()
MTStreamBufIn ()
MTStreamBufOut ()
MTStreamBufPause ()
MTStreamBufResume ()

Play/Record Functions

Record and playback are background functions that are queued for execution in the order of their arrival. Functions are separated to an encode or decode queue.

For record functionality, the SmartWORKS API starts encoding when a record request is queued and stops the encoding when all queued requests are filled. Should the recording media format differ from one request to the next, the encoding will be reset and restarted on the execution of the request with the new media format. The APIs for record are ***MTRecBuffer()*** and ***MTRecFile()***.

The same applies to playback functionality. APIs for playback are ***MTPlayBuffer()*** and ***MTPlayFile()***.

All queued record and playback requests can be deleted through function ***MTStopCurrentFunction()*** or ***MTStopChannel()***. When an application terminates a playback function manually with the ***MTStopChannel()*** function, then the data remaining in the SmartWORKS board hardware buffer is flushed.

Media Control

The SmartWORKS Media Control interface makes it possible to encode and decode audio voice data to and from the host system. The Media Control interface has two basic modes of operation: file mode or data-streaming mode. When using file mode, the Media Control interface manages all file I/O overhead and will record or play voice data directly to disk. In the data-streaming mode, voice data is passed to buffers, which the application must manage.

Indexed File Playback

A special playback mode is the indexed playback. With the ***MTPlayIndex()*** function, the application passes a file handle to the API, as well as a table of file offsets and byte counts. This table describes blocks of data from the file that should be played back in the order described. This is used for playing messages, prompts, and spoken numbers.

A simple indexed number file can consist of recordings of the numbers zero through nine, preceded by a table of indexes and block sizes where those recordings can be found. When the application wants to pronounce a phone number, it builds an index table of the desired numbers to playback. The built table is passed to the driver, which will play the blocks from the file in the specified order. The ***MTPlayIndex()*** function is provided to eliminate the overhead of opening and closing separate files for each digit to pronounce, as well as to facilitate smooth transitions between the blocks.

Stereo Recording

The SmartWORKS stereo media format allows for recording of synchronized signal samples from both the primary and secondary inputs of a channel.

Usage

Stereo format has two options:

MT_PCM_µLaw_Stereo
MT_PCM_ALaw_Stereo

Stereo format belongs to a class of encoders supported by SmartWORKS and is one of the components of the MT_IO_CONTROL structure used by media recording functions (record, stream).

Description

The PCM data from both the primary and secondary inputs are converted into linear format. Data from each channel is processed by AGC (if enabled) and by a gain stage (independent for primary and secondary input). The processed data is encoded into either A-law or mu-law byte. The resulting two bytes are combined into one 16-bit word with sample from primary input located in the low byte and sample from secondary input in the high byte. The 16-bit sample is then passed to the API.

When a file encoded in stereo format is played using GoldWave or CoolEdit player, the primary input will appear on the left channel the secondary will be on the right.

Please refer to the DSP logical channel model for more information on primary and secondary channels as well as AGC and gain blocks.

Energy Tagging

Most recording applications require mixing and compressing the audio for reduced storage requirements, but also need to retain information to determine which portions of the audio come from which input. Energy tags, included in the audio stream, provide the ability to separate the audio in all cases except where both speakers are active simultaneously.

Energy tagging is available when recording with SmartWORKS DP, NGX and PCM boards. The energy tagging feature provides both the primary and secondary input power every frame. The power is reported with two bytes put at the beginning of each compressed audio packet. The primary input signal (first byte) comes from the PBX (or CO) and the secondary input (second byte) comes from the agent side (or CPE).

Energy tagging is available for G.723.1 (5.3 kbps), mu-law and A-law. The power will be averaged over the duration of the speech sample (or frame). Specifically for G.723.1 the power will be calculated over the frame length of 30 ms and for mu-law and A-law the sample interval is 20 ms. All calculations occur after highpass filtering (when enabled). The following rules apply:

- Three codecs have been added that support energy tagging:
  MT_PCM_uLAW_POWER
  MT_PCM_ALAW_POWER
  MT_G723DOT1_5300_FIX_POWER
- These CODECs are available for encode only
- These CODECs are available for streaming, record to file and record to buffer

Energy tagging creates an 8-bit value representing the power on a channel. Power is measured at the output of the high-pass filter, so DC power and low frequency noise is not included if the filter is enabled. The value is biased so that the value 0xEF approximately represents a milli Watt. Therefore, if "x" is the value of the energy tag when treated as an unsigned 8-bit integer (so 0 <= x <= 255), then the approximate power on the channel is 0.376 * x- 89.9 [dBm].

- Power range: from –89.9 dBm to +3 dBm.
- Power measurement resolution: 1 dB.
- Byte location: the first two bytes in a frame.

Record Functions

Record and playback are background functions that are queued for execution on the order of their arrival to either encode queue or decode queue separately.

For record functionality, the SmartWORKS API starts the encoding when a record request is queued and stops the encoding when all queued requests are filled. Should the recording media format differs from one request to the next, the encoding will be reset and restarted on the execution of the request with the new media format. The APIs for record are ***MTRecBuffer()*** and ***MTRecFile()***.

The same applies to playback functionality. APIs for playback are ***MTPlayBuffer()*** and ***MTPlayFile()***.

All queued record and playback requests can be deleted through function ***MTStopCurrentFunction()*** or ***MTStopChannel()***. When an application terminates a playback function manually with the ***MTStopChannel()*** function, then the data remaining in the SmartWORKS board hardware buffer is flushed.

Media Control

The SmartWORKS Media Control interface makes it possible to encode and decode audio voice data to and from the host system. The Media Control interface has two basic modes of operation: file mode or data-streaming mode. When using file mode, the Media Control interface manages all file I/O overhead and will record or play voice data directly to disk. In the data-streaming mode, voice data is passed to buffers, which the application must manage.

Data Streaming

Instead of specifying the size of a recording, the application program could use a streaming function to enable voice data buffering. Then use stream in and stream out API functions to copy and queue voice block one by one until the application disables the streaming operation.

SmartWORKS API functions for streaming control are ***MTSetStreamingConfig()***, ***MTStartStreaming()*** and ***MTStopStreaming()***. The record and playback equivalent APIs are ***MTStreamBufIn()*** and ***MTStreamBufOut()***. Additional decode control APIs are ***MTStreamBufPause()*** and ***MTStreamBufResume()***.

Start streaming enables API internal buffering. The default internal buffer size is 64K, which would buffer about 8 seconds of 8-bit PCM encoded voice. This internal buffer size can be changed through ***MTSetStreamingConfig()***. When the internal buffer overflows, the application will be notified through event EVT_STREAMIN_DROPPED. No new voice data will be queued until the application issues an action to retrieve the queued data. Event EVT_STREAMOUT_EMPTY notifies the user application that the decode playback streaming is short of voice data and silence is played.

A watermark can be set when streaming is started. If set, event EVT_STREAMIN_WATERMARK or EVT_STREAMOUT_WATERMARK will be issued as an indication of streaming progress.

When streaming is used with termination (specified through MT_IO_CONTROL parameter), data buffering will stop when termination occurs. **NOTE:** MaxBytes and MaxTime cannot be used for termination limits when streaming. A termination event is issued. Both streamed data and the streaming buffer are kept until the user application either retrieves all streamed data or issues a stop streaming to flush the streamed data. Streaming buffer will be released in both cases. The return code for ***MTStreamBufIn()*** will be MT_RET_SERVICE_STOPPED or

MT_RET_SERVICE_NOT_STARTED to indicate that all data in streaming buffer has been read out and streaming task is now relinquished. For example, say a streaming task is started with silence termination set to 500 milli-second. At the detection of 500 milli-seconds of no activity, termination event of EVT_TERMSILENCE will be issued and the collection of data will stop. However, all the previously collected data are still kept in the streaming buffer. Users should invoke ***MTStreamBufIn()*** until one of the following return codes is reported: MT_RET_SERVICE_STOPPED or MT_RET_SERVICE_NOT_STARTED.

For example:

Assume that the size of data left in the streaming buffer is 40K at the time of silence detection, if the user application issues ***MTStreamBufIn()*** with a 32K buffer, then 32K of the streamed data will be moved into user buffer, the return code for ***MTStreamBufIn()*** will be MT_RET_OK while the channel status remains streaming. On the next call of ***MTStreamBufIn()*** call, only 8K of streamed data will be moved into user buffer, and the return code will be MT_RET_SERVICE_STOPPED or MT_RET_SERVICE_NOT_STARTED. At the completion of the second call of ***MTStreamBufIn()***, the streaming task along with the associated streaming buffer is released from SDK. The event EVT_STREAMIN_STOP is reported. At this time the channel status returns to idle.

NOTE: The SmartWORKS DLL limits buffer size: maximum streaming buffer size is set to 1MBytes, minimum cap to 1KBytes.

Activity Detection

The Activity detector measures input signal energy in a 20 ms sample. The energy measurement is then converted to average power and the result is compared against two programmable thresholds:

- The silence threshold (MT_ACTPARAMS.threshold_low)
- The activity threshold (MT_ACTPARAMS.threshold_high)

where MT_ACTPARAMS.threshold_high **must be greater than or equal to MT_ACTPARAMS.threshold_low.**

The result of this comparison is processed by the ACT state machine. This machine has three states:

- StReset
- StSilence
- StActivity

When the detector is disabled, it is held in stReset. Enabling the detector causes it to change to the stSilence state, the Silence Timer to be started from zero, and the Activity Timer to remain reset. Disabling the detector puts it in the stReset state.

Whenever the detector is in the stSilence state and the silence timer reaches the Maximum Silence Duration (MT_ACTPARAMS.max_silence), the silence timer is restarted from zero. If Activity Detection events are enabled, a Maximum Silence Period Detected (EVT_MAX_SILENCE) event is issued with SubReason field containing the value of the silence timer before it was restarted.

Whenever the detector is in the stActivity state and the activity timer reaches the Maximum Activity Duration (MT_ACTPARAMS.max_activity), the activity timer is restarted from zero. If Activity Detection events are enabled, a Maximum Activity Period Detected (EVT_MAX_ACTIVITY) event is issued with a SubReason field containing the value of the activity timer before it was restarted.

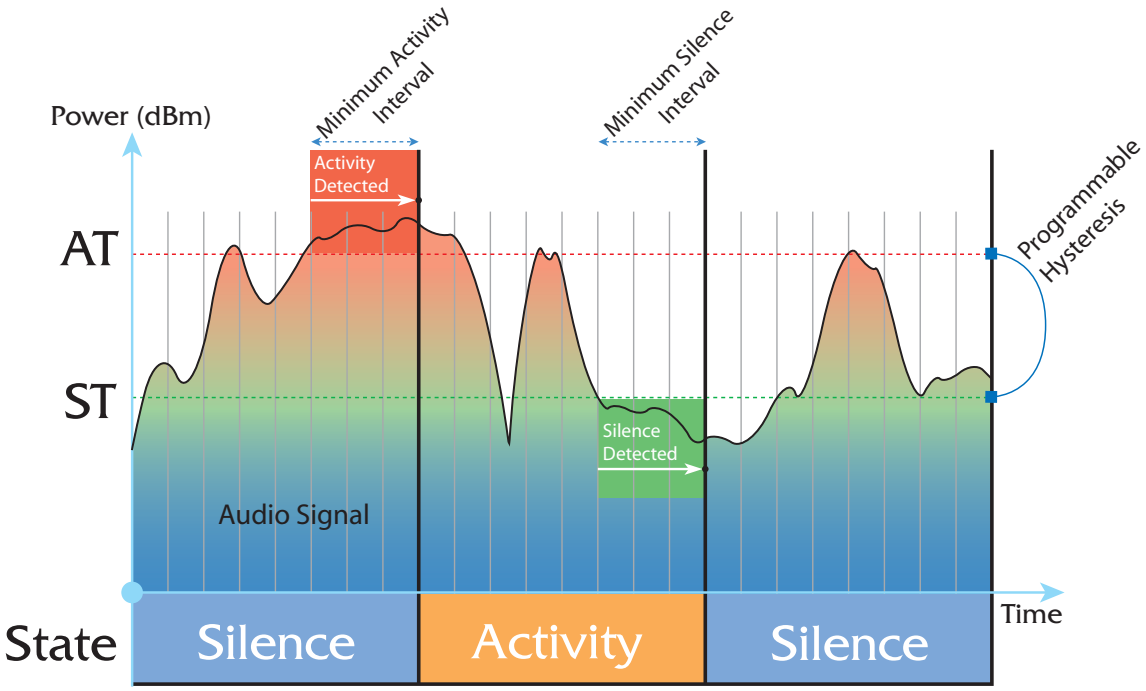
Whenever the detector is in the stSilence state, and the measured input signal energy remains above MT_ACTPARAMS.threshold_high for the Minimum Activity Duration (MT_ACTPARAMS.min_activity), the detector changes to the stActivity state, the activity timer is restarted from timActivityMinimum, and the silence timer remains reset. If Activity Detection events are enabled, a Minimum Activity Period Detected (EVT_MON_ACTIVITY) event is issued with SubReason field containing the value of the silence timer when the energy level crossed above the threshold.

Whenever the detector is in the stActivity state and the measured input signal energy remains below MT_ACTPARAMS.threshold_low for the Minimum Silence Duration (MT_ACTPARAMS.min_silence), the detector changes to the stSilence state, the silence timer is restarted from timSilenceMinimum, and the activity timer remains reset. If Activity Detection events are enabled, a Minimum Silence Period Detected (EVT_MON_SILENCE) event is issued with a parameter containing the value of the activity timer when the energy level crossed below the threshold.

Activity Detection

Legend:

AT · Activity Threshold (dBm)
ST · Silence Threshold (dBm)
Hysterisis · Difference between AT and ST (dB)



Legend

Minimum Activity Interval . . . . . MTACTPARAMS.min_activity

Minimum Silence Interval . . . . . MTACTPARAMS.min_silence

Programmable Hysteresis . . . . . MTACTPARAMS.threshold_high & MTACTPARAMS.threshold_low

API Control

Originally Activity Detection was designed to work in the following way:

- activity detection, when enabled with **MTEnableACTD()**, only worked on the primary input of the channel. **MTEnableMixing()** then **MTEnableMixingDetect()** was used to enable activity detection on the secondary channel. The API **SetACTDParams()** configured activity detection parameters, but the same values were used for both inputs.

The APIs used to control activity detection were recently enhanced to allow for configuration on a per input basis.

Users can still control activity parameters with the API **MTSetACTDParams()**. However, this method is only recommended when activity detection configuration is the same on both inputs.

To control activity detection on the primary input, do the following:

- **MTChInputACTDControl()** - enable / disable activity detection

- **MTChInputSetACTDParams()** or **MTSetACTDParams()** - configures activity detection parameters

To control activity detection on both inputs, do the following:

- **MTEnableMixing()** - combine the primary and secondary inputs

- **MTChInputACTDControl()** - enables / disables activity detection (per input). **MTEnableMixingDetect()** will be obsoleted in Jan. 2005.

- **MTChInputSetACTDParams()** - configures activity detection parameters on both the primary and secondary inputs (NOTE: **MTSetACTD()** can still be used, but the configuration values will be the same for both the primary and secondary inputs).

Loop Voltage / Loop Current / Ring Detect Functions

An explanation of Loop Voltage / Current on SmartWORKS boards

The following provides an overview of how loop voltage / current is managed when using the SmartWORKS API.

SmartWORKS LD Cards

LD can detect both loop voltage change and loop current change.

Loop Voltage:

LD provides user the configuration capability to set the thresholds of voltage high and voltage low. This allows custom definition of the three loop voltage states of ABOVE, BELOW, and MIDDLE (e.g. ONHOOK, REVERSE, and OFFHOOK states are the common terms for ABOVE, BELOW, and MIDDLE states). With LD's capability of detecting the three states of loop voltage change, LD can detect the presence of a wink, a state PT channel cannot detect. LD can also provide current voltage reading, a capability PT channel does not have.

De-Bouncing:

The specification of de-bouncing time for ring and loop is implemented through data structure MT_PSTN and APIs of ***MTSetPSTNParams()*** and ***MTGetPSTNParams()***. Data field ring_deglitch specifies the ring de-bouncing time, and field loop_deglitch specifies the loop current de-bouncing time for AT channel and the loop voltage de-bouncing time for PT channel.

Since LDA detects both loop current and loop voltage, the SDK needs supports the case where the de-bouncing time for loop current and loop voltage differs. A set of APIs for get and set loop voltage parameters are used: ***MTSetLVParams()***, ***MTGetLVParams()***.

Line Status:

Line status for LD card will be represented through status bit LINE_ONHOOK, LINE_POLARITY(0 for normal; 1 for reversed), LINE_NO_LOOP, and LINE_NO_LVOLTAGE_MIDDLE.

Event Filtering:

For LD channels, the possible yielding events are EVT_LVOLTAGE_ABOVE, EVT_LVOLTAGE_BELOW, or EVT_LVOLTAGE_MIDDLE (i.e. EVT_LVOLTAGE_OFFHOOK). Event filtering of SE_LCURRENT_CHANGE yields events of EVT_LOOP_ON, EVT_LOOP_DROP, and EVT_LOOP_REVERSE. Event filtering of SE_LREV is obsolete and not supported for the loop current or loop voltage polarity change states are already included in SE_LVOLTAGE_CHANGE and SE_LCURRENT_CHANGE filtering.

Line Polarity:

Line polarity monitoring, through API ***MTSetMoni()*** of MONI_REVERSAL bit is implemented on the LD by adding a new event EVT_MON_REVERSAL during detecting of state EVT_LOOP_REVERSE or EVT_LVOLTAGE_BELOW.

For events that indicates a loop current or voltage state change, the information of the current state, the previous state and the duration of the previous state are presented through field MT_EVENT.SubReason and MT_EVENT.DataLength:

CurrentState:

the least significant byte of field MT_EVENT.SubReason;

PreviousState:

the second least significant byte of field MT_EVENT.SubReason;

DurationOfPreviousState:

the value in MT_EVENT.DataLength times 125 micro-seconds

Event filtering of SE_WKRECV will not be implemented in SDK. User application can determine the presence of a WINK based on the above timing information from events.

API ***MTWink()*** does not apply to LD channel for it can generate a hook flash, not a wink.

Loop Termination:

The loop termination supported for media tasks are TERM_LOOP_DROP and TERM_LVOLTAGE_ABOVEORBELOW (i.e. TERM_LVOLTAGE_NOTOFFHOOK). The starting of a media task based on START_LVOLTAGE_MIDDLE (i.e. START_LVOLTAGE_OFFHOOK) implies the termination on LVOLTAGE_ABOVEORBELOW (i.e. LVOLTAGE_NOTOFFHOOK). The starting of a media task based on START_LOOPON implies the termination of LOOP_DROP. Termination TERM_LOOP_DROP and TERM_LVOLTAGE_ABOVEORBELOW is allowed one at one time on a LD channel. Media starting control of START_LVOLTAGE_MIDDLE and START_LOOPON is allowed one at one time on a LDA channel.

Also, in loop start settings where the tip is grounded and the ring is –48V, the normal loop polarity detection on LDA is ABOVE, BELOW and MIDDLE. However, in ground start settings where the ring is grounded and the tip is at 48V, the detection will be reported reversed as stated above. An API (***MTSetReverseLoopPolarity()***) is needed to report the same in ground start environment.

Firmware Functions

User applications can use the SmartWORKS API to update a board's firmware. The following rules must be adhered to:

- These APIs cannot be used with the SmartWORKS NGX
- In the event of error during the Update Flash operation, the user must restart the driver
- These APIs are declared in a separate header file: NtiWFAPI.h
- The application performing the firmware upload process must have exclusive access to the board. Multiple applications cannot be running when this function is initiated
- It is recommended that the application invoking ***MTWFUpdateFirmware()*** is a stand alone application. This API checks for any open threads on the board before flashing the firmware. If any are found an error message is returned
- It is recommended that all applications with access to this board invoke ***MTCloseBoard()*** before ***MTWFUpdateFirmware()*** is invoked

The diagram on the following page illustrates the steps required when updating a board's firmware:

Flashing a board using the SmartWORKS API
Note: In this scenario MTWFUpdateFirmware() is running as a background function