TASK-6000™ software reference guide

www.radisys.com

World Headquarters
5445 NE Dawson Creek Drive • Hillsboro, OR
97124 USA
Phone: 503-615-1100 • Fax: 503-615-1121
Toll-Free: 800-950-0044

International Headquarters
Gebouw Flevopoort • Televisieweg 1A
NL-1322 AC • Almere, The Netherlands
Phone: 31 36 5365595 • Fax: 31 36 5365620

007-00992-0002
December 2000

December 2000
Copyright ©2000 by RadiSys Corporation.
All rights reserved.

EPC, iRMX, INtime, Inside Advantage, and RadiSys are registered trademarks of RadiSys Corporation. Spirit, DAI, DAQ, ASM, Brahma, and SAIB are trademarks of RadiSys Corporation.

† All other trademarks, registered trademarks, service marks, and trade names are the property of their respective owners.

Before you begin

This guide explains how to install and use TASK-6000™ software.

TASK-6000 software provides programming methodology, tools, and runtime libraries that you use to quickly develop optimized, multi-algorithm, multi-channel telecom applications for the Texas Instruments' TMS320C620x (c6x) DSP.

About this guide

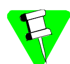
Contents

Chapter/appendix		Description
1	Introducing TASK-6000 software	Overviews TASK-6000 software solutions.
2	Understanding TASK-6000 software architecture	Explains how TASK-6000's kernel works with host, IOP, and DSP OSs to provide telecom functionality. It also lists and describes TASK-6000 components.
3	Installing and configuring TASK-6000 software	Explains how to install and uninstall TASK-6000 software.
4	Developing Host and IOP applications	Provides general guidelines for host and IOP application design.
A	Host functions	Lists and describes calls used to communicate with the host.
B	IOP functions	Lists and describes calls used to communicate with the i960† I/O processor.
C	HDLC driver library	Describes the HDLC driver library and its calls.
D	T1/E1 library	Describes the T1/E1 library and its calls.
E	T8100 library	Describes the T8100 library and its calls.
F	Service descriptions	Lists and describes TASK-6000 services.

Notational conventions

This guide uses the following conventions:

- `Screen text and syntax strings appear in this font.`
- All numbers are decimal unless otherwise stated.
- Bit 0 is the low-order bit. If a bit is set to 1, the associated description is true unless otherwise stated.


Notes indicate important information about the product.

Tips indicate alternate techniques or procedures that you can use to save time or better understand the product.

The globe indicates a World Wide Web address.

Cautions indicate situations that may result in damage to data or the hardware.

This includes situations that may cause damage to hardware via electro-static discharge (ESD).

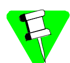
Warnings indicate situations that may result in physical harm to you or the hardware.

Where to get more information

About TASK-6000

You can find out more about TASK from these sources:

- **Release notes (relnotes.txt)**: Lists features and issues that arose too late to include in other documentation.
- **World Wide Web**: RadiSys maintains an active site on the World Wide Web. The site contains current information about the company and locations of sales offices, new and existing products, contacts for sales, service, and technical support information. You can also send e-mail to RadiSys using the web site.

  
  When sending e-mail for technical support, please include information about both the hardware and software, plus a detailed description of the problem, including how to reproduce it.

  To access the RadiSys web site, enter this URL in your web browser:
  `http://www.radisys.com`

  Requests for sales, service, and technical support information receive prompt response.

- **Other**: If you purchased your RadiSys product from a third-party vendor, you can contact that vendor for service and support.

About related RadiSys products

TASK software algorithms

The RadiSys TASK software algorithms extend the functionality of your TASK software, providing the tools you need for DSP programming. TASK software algorithms include:

Algorithm	Description
AGC/VOX software for TMS320C620x	Performs level estimation, normalization of PCM samples with respect to the user specified reference level and voice activity detection, which are part of the voice processing system.
Bad frame masking and comfort noise generator for TMS320C620x	Bad frame masking eliminates gaps in the received voice signal due to congested network conditions. The comfort noise generator engine generates comfort noise at the decoder of G.711 when no packets are received or when the received VAD (Voice Activity Detector) field indicates silence or background noise frames.
Call progress monitor for TMS320C620x	Performs CPT detection for high capacity trunks.
Line echo canceller for TMS320C620x	Provides: • Up to 32 ms echo span. • Robust double-talk detection. • Adaptation Non-linear processing control.
ITU-T G.711 Speech Coder for TMS320C620x	Encodes and decodes with an option to select A-Law/ μ-Law and multiple frame sizes at compilation/run time.
ITU-T G.723.1 Speech Coder for TMS320C620x[1]	Encodes and decodes based on 30 ms frames, as specified by the *ITU-T Recommendation G.723.1 Dual Rate Speech Coder for Multimedia Communications Transmitting at 5.3 and 6.3 kbits*.
ITU-T G.729 Speech Coder for TMS320C620x[1]	Encodes and decodes based on 10 ms frames, as specified by the *ITU-T Recommendation G.729 Coding of speech at 8 kbits/s using conjugate-structure algebraic-code-excited linear-prediction (CS-ACELP)*.

Algorithm	Description
ITU-T G.729A Speech Coder for TMS320C620x[1]	Encodes and decodes based on 10 ms frames, as specified by the *ITU-T Recommendation G.729 annex A, a low-complexity version of "Coding of speech at 8 kbits/s using conjugate-structure algebraic-code-excited linear-prediction (CS-ACELP).* A silence compression scheme specified by the *ITU-T Recommendation G.729 annex B, A silence compression scheme for G.729 optimized for terminals conforming to ITU-T V.70 digital simultaneous voice and data application*, operates with G.729A coder to reduce transmission rate.
Line/Register Signaling (R1/R2 MF) for TMS320C620x	Performs R1/R2 detection for high capacity trunks.
DTMF Detector/Suppressor for TMS320C620x	Performs DTMF detection and suppression for high-capacity trunks.
Tone Generator for TMS320C620x	Performs general call-progress and other telecom tones generation for high capacity trunks.
IETF RTP Protocol[1]	Performs RTP packetizing, depacketizing, and jitter buffering, as specified by the *IETF Request for Comments 1889*.

[1] This algorithm is not part of the TASK-6000 release. To purchase this algorithm, contact RadiSys as described in About TASK-6000 on page ii.

SPIRIT™ boards

The RadiSys SPIRIT boards are optimized for telecom OEM applications. The SPIRIT family currently includes the RadiSys SP6040E (SPIRIT-6040 CompactPCI† board), a high-performance single board voice gateway designed for telecom and datacom applications. Based on Texas Instruments† devices, the SP6040E has a 200MHz DSP engine with a PMC connector that provides either LAN/WAN interface or additional DSP resources. The SP6040 contains four TMS320C6201 digital signal processors for processing multi-channel telecommunications with Hot Swap support.

About other related products

TI tools

Software tools from Texas Instruments used to build DSP executables.

For more information about Texas Instruments products, enter this URL in your web browser:
http://www.ti.com/dsp

RadiSys Line/Register Signaling (R1/R2 MF) for TMS320C6201 User's Manual, RadiSys Corpration.

RadiSys Call Progress Tones Monitor for TMS320C6201 User's Manual, RadiSys Corporation.

ITU-T Rec. H.225, Media stream Packetization and synchronization on non-guaranteed quality of service LANs

RFC 1889 RTP - A Transport Protocol for Real Time Applications,

H.225 Real-time Transport Protocol / Real-time Transport Control Protocol API Document

Hughes Software Systems RTP/RTCP module specification

Contents

Chapter 1: Introducing TASK-6000 software

Product configurations 2

Chapter 2: Understanding TASK-6000 software architecture

Components 3
- TASK Host runtime library 3
  - taskhost.dll 3
  - NT Kernel Mode device driver (i960rp.sys) 4
  - Hot Swap Host library 4
- TASK IOP runtime library 4
  - VxWorks†-based library 4
  - Peripheral device driver libraries 5
- DSP application 6
- Utilities 6
  - sp6k_util.exe 6
  - rmondb.exe 6

Interfaces 6
- Internet Protocol (IP) 6
- Public Switched Telephone Network (PSTN) 7
- Application Programming Interface (API) 7
  - Objects 7
  - Services 8
  - Inter-processor communication 10

Chapter 3: Installing and configuring TASK-6000 software

Requirements 13

Before you begin 13

Running the install program 14

Uninstalling TASK software 17
- Automatic uninstall (recommended) 17
  - Uninstalling Hot Swap 18
- Manual uninstall procedure 19

TASK-6000 files 20
- Hot Swap 20
- Runtime kit 21
- Development kit 23
- DSP development kit 26

Chapter 4: Developing Host and IOP applications

Developing the Host application (IopAppLoader.c) 30
- Initialize the Host driver 30
- Set up message handlers 30
- Initialize UPA structures 30

Load and run applications 30
Developing the IOP application (UpaIopApp.c) 31
Initialize the IOP driver 31
Set up message handlers 31
Configure services 32
Create data paths 32
Building Host and IOP applications 33
Naming conventions 34
Host application development 34
IOP application development 34
Special note for IOP applications 35
Sample code: Host application (IopAppLoader.c) 36
Initializing the host driver 36
Setting up message handlers 36
Initializing UPA structures 36
Loading and running applications 37
Retrieving system information 37
Loading the VxWorks image on IOPs 37
Waiting for IOP response 38
Loading mulcoder.out 38
Sample code: IOP application (UpaIopApp.c) 39
Initializing the IOP drive 39
Calling iopInit 39
Calling upStart 39
Configuring on-board peripherals 40
Setting up message handlers 41
Installing event handlers (callback functions) 41
Connecting the TDM to IOPs 41
Enabling bi-directional voice data flow to and from the DSP 41
Configuring services 42
Configuring DSP services for outbound direction (toward the IP cloud) 42
Creating data paths 43
Creating a data path between an outgoing RTP channel running on a DSP and the IOP driver's network packet routing component 43
Enabling the receive direction on the channel 44
Initializing the RTP decoder 45
Creating a data path between an inbound RTP channel running on a DSP and the IOP driver's network packet routing component 45

Appendix A: Host functions

Overview 47
Message API 47
Function list 48
hostControlPeripheral 50
hostExit 54
hostGetBoardInfo 55
hostGetSystemInfo 57
hostInit 59
hostLoadDsp 60
hostLoadIop 61

hostResetBoard 62
hostResetDsp 63
hostRunDsp 64
hostRunLoadedIop 65
hostSetEventHandler 66
hostSetHotSwapHandler 67
hostSetPeripheralDataHandler 68
upConfigService 69
upConfigServiceGlobal 72
upConnectPktRecv 75
upConnectPktSend 77
upDisableService 79
upDisconnectPktRecv 82
upDisconnectPktSend 83
upEnableChannel 84
upEnableService 86
upQueryQOSReport 89
upSetEventHandler 90
upSetUserMsgHandler 91
upStart 92
hostGetNWPktBuf 94
hostJitterControl 95
hostReadIop 96
hostSendNWPktBuf 97
hostSendMsg 98
hostSetNWNotify 99
hostWriteIop 100
hostSetPollPeriod 101
hostSendPriorityMsg 102

Appendix B: IOP functions

Overview 103
Message API 103
Function list 104
getBoardInfo 106
iopControlPeripheral 108
iopInit 112
upConfigService 113
upConfigServiceGlobal 116
upConnectPktRecv 119
upConnectPktSend 121
upDisableService 123
upDisconnectPktRecv 126
upDisconnectPktSend 127
upEnableChannel 128
upEnableService 130
upQueryQOSReport 133
upSetEventHandler 136
upStart 139
iopGetNWPktBuf 141

iopJitterControl 142
iopSendNWPktBuf 143
iopSendMsg 144
iopSetNWNotify 145
upSetUserMsgHandler 146

Appendix C: HDLC driver library

Overview 147
Driver internals, data structures, and resources 147
Data structures 148
Processing modes 148
Processing packet transmission and reception 149
Sample HDLC driver sequence 150
Function list 151
Functions 153
HDLCInit 153
HDLCReset 154
HDLCCloseDriver 155
HDLCClosePort 156
HDLCConfigPort 157
HDLCConfigChannel 158
HDLCEnableChannel 159
HDLCDisableChannel 160
HDLCResetChannel 161
HDLCSendPacket 162
HDLCGetPacket 163
HDLCGetDeviceStatus 164
HDLCGetChannelStatus 165
HDLCSetTxPacketHandler 166
HDLCSetRxPacketHandler 167
HDLCSetDeviceErrorHandler 168
HDLCSetTxErrorHandler 169
HDLCSetRxErrorHandler 170
Type definitions 171
Structures 172
t_HDLC_port_config 173
t_HDLC_channel_config 175
t_HDLC_channel_status 176

Appendix D: T1/E1 library

Overview 177
Sample startup sequence 178
Function list 179
Functions 183
T1E1initCard 183
T1E1getBoardConfig 184
T1E1setLeds 185
setT1Config 186
setT1Signaling 187
setT1Command 188

setT1ClearChannels 189
setT1IdleChannels 190
setT1ChannelConfig 191
setE1Config 192
setE1Signaling 193
getT1Signaling 194
getT1Status 195
getT1SignalingRaw 196
getE1Signaling 197
setT1SignalingHandler 198
setT1StatusHandler 199
setE1SignalingHandler 200
Structures 201
t_T1E1_framer_id 203
t_T1E1_card_type 204
t_T1E1_led_state 205
t_T1E1_user_signaling_data 206
t_T1E1_BoardConfig 207
t_T1_line_coding 208
t_T1_framing_mode 209
t_T1_line_buildout 210
t_T1_user_config_struct 211
t_T1_user_signaling_data
t_T1_signaling_data 213
t_T1_user_command_data 214
t_T1_user_status_struct 215
t_T1_user_clear_channel_data 216
t_T1_user_idle_struct 217
t_T1_user_channel_config 218
t_T1_user_raw_signaling_struct 220
Example 220
T1StatusHandler 221
T1SignalingHandler 222
t_E1_line_coding 223
t_E1_signaling_mode 224
t_E1_line_buildout 225
t_E1_user_config_struct 226
t_E1_user_signaling_data 228
E1SignalingHandler 229

Appendix E: T8100 library

Overview 231
Making and breaking connections 231
Broadcasting 232
Sample startup sequence 234
Function list 235
Limitations 235
Functions 237
initT8100 237
setT8100ClockConfig 238

setT8100StreamConfig 239
setT8100SwitchConfig 240
clearT8100ClockFault 241
clearT8100MemoryFault 242
setT8100ClockFaultMask 243
getT8100ErrorStatus 244
setT8100Handler 245
Structures 246
t_ref_clk 247
t_fallback_clk 248
t_netref_clk 249
t_T8100ClockConfig 250
t_stream_rate 252
t_T8100StreamConfig 253
t_source_dest 256
t_T8100Connection 257
t_T8100SwitchConfig 258

Appendix F: Service descriptions

Codec 260
stCodec 260
Echo cancellation 263
stEchoCanc 263
Tone generation 264
stTdmToneGen 264
stPktToneGen 265
UP_TONEGEN_CONFIG_ST 265
Tone detection 266
stTdmDTMFDet 266
UP_DTMF_CONFIG_ST 266
UP_EVT_TDM_DTMF_DETECTED 266
UP_DTMF_DETECTED_DATA_ST 266
stPktDTMFDet 267
UP_DTMF_CONFIG_ST 267
UP_EVT_PKT_DTMF_DETECTED 267
UP_DTMF_DETECTED_DATA_ST 267
stCPTDet 268
UP_CPT_CONFIG_ST 268
UP_EVT_CPT_DETECTED
UP_CPT_DETECTED_DATA_ST 268
stMFDet 270
UP_MF_CONFIG_ST 270
UP_EVT_MF_DETECTED 270
UP_MF_DETECTED_DATA_ET 271
RTP packetization 272
stRtpEncode 272
RTP_HEADER_ST 272
UP_RTP_SEND_CONFIG_ST 272
stRtpDecode 274
UP_RTP_RECV_CONFIG_ST 274

UP_EVT_RTP_PT_CHANGE
UP_RTP_PT_CHANGE_DATA_ST 275
UP_EVT_RTP_SSRC_CHANGE
UP_RTP_SSRC_CHANGE_DATA_ST 275
Signaling 276
stCAS 276
UP_CAS_CONFIG_ST 276
UP_EVT_CAS_CHANGE
UP_CAS_CHANGE_DATA_ST 276
stQDS0Hdlc 277
Alarming 278
stEthernetAlarm 278
UP_EVT_ETHERNET_ALARM
UP_ETHERNET_ALARM_DATA_ST 278
stT1E1Alarm 279
UP_T1E1ALARM_CONFIG_ST 279
UP_EVT_T1E1_ALARM
UP_T1E1_ALARM_DATA_ST 279
Audio processing 281
stAGC 281
UP_AGC_CONFIG_ST 281
Internal 282
stPacketBuilder 282
UP_PACKET_BUILDER_CONFIG_ST 282
stPacketParser 283
UP_PACKET_PARSER_CONFIG_ST 283
Glossary 285
Index 293

Figures

Figure 1-1. TASK application distribution: IOP only 1
Figure 1-2. TASK application distribution: Host and IOP 2
Figure 2-1. DSP application on the DSP chip 6
Figure 2-2. Data path services 9
Figure 2-3. Using events and message handlers 11
Figure 4-1. Location of files required to compile the application 33
Figure A-1. Packet organization buffer 97
Figure B-1. Packet organization buffer 143

Tables

Table A-1. Host functions 48
Table B-1. IOP functions 104
Table C-1. HDLC driver functions 151
Table C-2. HDLC structures 172
Table D-1. T1/E1 functions 179
Table D-2. Time slot numbers 181
Table D-3. E1/T1 structures 201
Table E-1. T8100 functions 235
Table E-2. Time slot numbers 236
Table E-3. T8100 structures 246

1 Introducing TASK-6000 software

To create the complex applications needed by today's telecom service providers, system developers can benefit strongly from a proven software toolkit that abstracts the details of codecs, telephony algorithms, and network data connections. The RadiSys TASK-6000 software provides a powerful set of features that reduce development time without sacrificing flexibility, allowing the developer to cost-effectively add value at the application level.

Based on a powerful API, with a set of field-proven DSP algorithms and high-performance packet protocols under the hood, TASK forms the core of Media Gateways and other vital telecom applications. It takes its name from the underlying RadiSys "Telecom Application-Specific Kernel" that runs on the Digital Signal Processors (DSPs) and provides the flexible and powerful core of the telephony applications.

TASK components are distributed among the three hardware subsystems of the RadiSys family of Media Gateway products: DSP, IOP, and Host, according to the performance requirements dictated by modern VoIP and similar applications. A developer may choose to create applications whose functionality resides primarily on the IOP, primarily on the Host, or in a distributed fashion using both:

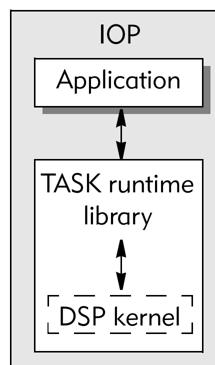
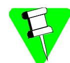


Figure 1-1. TASK application distribution: IOP only

Some applications will utilize multiple Hosts to meet high-availability requirements. Advanced developers can even create DSP-based applications that use the powerful RadiSys DSP library components directly.



For information about developing TASK applications, see *Chapter 4, Developing Host and IOP applications*.

Developing DSP-based applications using RadiSys DSP library components requires that you use the TASK-6000 DSP development kit. For more information about this kit, see *Product Configurations*, later in this chapter.

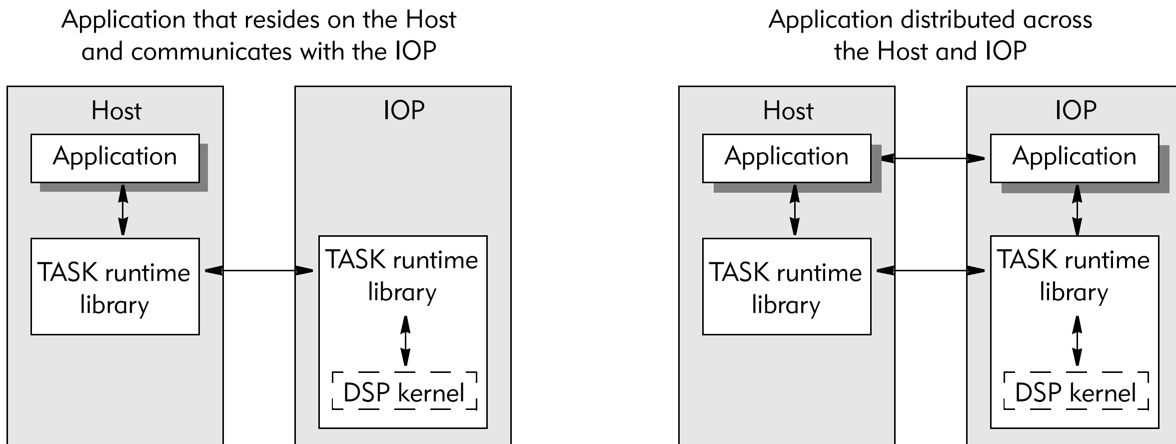


Figure 1-2. TASK application distribution: Host and IOP

TASK supports data and control flow between the Host- and IOP-based components, using the hardware PCI interconnection (for high throughput and low latency) or using a message-based interface over a protocol link via the IOP's optional LAN/WAN subsystems.

The TASK API creates abstractions for common telecom objects such as Slots, Units, Channels, and UDP Ports, and provides functions for the configuring, enabling, and disabling of a powerful set of services that operate on these objects. The API also provides abstractions for Fast Packet Routing, HDLC, T1/E1, and TDM (Time-Division Multiplex) Switch peripheral devices.

TASK-6000 software supports CompactPCI Hot Swap, as defined in the Hot Swap Specification [PICMG97c], for fail-over and high-availability applications.

Product configurations

You can order TASK software in these configurations:

- **TASK-6000 runtime kit**: Provides the software required to run TASK applications.
- **TASK-6000 development kit**: Provides the tools required to write and debug TASK Host and IOP applications. This software also includes the TASK-6000 runtime kit.
- **TASK-6000 DSP development kit**: Provides the tools required to write and debug TASK DSP applications. This software also includes the TASK-6000 development kit.

2 Understanding TASK-6000 software architecture

This chapter explains how the TASK-6000 software components and interfaces work together to provide the tools you need to create and run telephony applications.

When reading this file online, you can immediately view information about any topic by placing the mouse cursor over a topic name and clicking:

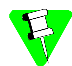
For information about...	Go to this page...
Components	3
TASK Host runtime library	3
TASK IOP runtime library	4
Utilities	6
Interfaces	6
Internet Protocol (IP)	6
Public Switched Telephone Network (PSTN)	7
Application Programming Interface (API)	7

Components

The TASK-6000 software architecture comprises several interdependent components: a Host Run-Time Library, an IOP Run-Time Library, and a standard DSP application. The following sections describe these components.

TASK Host runtime library

User Host application components interface to TASK through the Host runtime library. This comprises a static library linked to the Host application and a Win32 DLL (Dynamic Linked Library), which implements the functions of the API (Application Programming Interface).



For information about the Host API and functions, including syntax and parameter values, see Appendix A, Host functions.

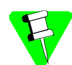
taskhost.dll

The taskhost.dll library creates several Win32 threads which poll the IOPs in all slots of a system to monitor for data and events. It resets, downloads, and monitors the IOPs, maintains message and data queues, communicating with each IOP via the NT Kernel Mode device driver. taskhost.dll includes API functions that implement callbacks to user-defined functions for event handling and data delivery.

NT Kernel Mode device driver (i960rp.sys)

The NT Kernel Mode device driver is a low-level driver that provides PIO (Programmed I/O) and DMA (Direct Memory Access) to the memory and control registers on all IOPs in the system. It also scans the PCI bus at system boot time, enumerates the IOPs and, through interaction with the Hot Swap Service, maintains slot state information.

Hot Swap Host library



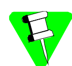
For detailed information about the RadiSys Hot Swap for Windows NT, see *Hot Swap for Windows NT* (PN 07-1080-00).

Hot Swap support is implemented as a Windows NT service and a device driver. These communicate with taskhost.dll to notify the user application of insertion and extraction events for all IOPs in the system. They communicate with the IOP through the TASK NT Kernel Mode device driver to detect the extraction tab flips and illuminate the blue quiescent LEDs on each IOP.

- **hsmgrint.dll**: Implements the Hot Swap Manager library and creates the Win32 threads necessary to poll the IOPs in the system for insertion and extraction events.
- **hbus.sys (NT Kernel Mode device driver)**: Enumerates the hot swap capable devices present in the system, monitors their hot swap state, alerts the Hot Swap Manager to changes in a device's hot swap state, and configures the device's hot swap configuration status register (HS_CSR) in response to requests from the Hot Swap Manager.

TASK IOP runtime library

TASK-6000 applications that reside on the IOP interface directly with the TASK IOP runtime library. This component serves command and event API functions to control on-board peripherals such as the T8100 TDM (Time Division Multiplex) switch, the optional LAN/WAN interface, and to load, reset and run the DSPs.



For information about the IOP API and functions, including syntax and parameter values, see Appendix B, IOP functions.

VxWorks†-based library

The current implementation of the IOP runtime library is based on Wind River's VxWorks RTOS (Real-Time Operating System), which it uses for multi-tasking, messaging, and IP (Internet Protocol) stacks. You can create user applications under the Tornado† 2.0 development system and link them with the components of the IOP runtime library to produce easily-debuggable applications.

The library includes the following:

- **Message dispatcher**: The IOP runtime library exposes its API to local user applications (in the same IOP), as well as to applications that run on a remote processor. For example, a Host application can exercise the API indirectly via

the Host runtime library, which forms messages and sends them to the IOP across the PCI bus.

The messages from remote API servers are acted upon in the same fashion as locally-originated function calls.

- **Event dispatcher**: Events generated locally on the IOP or arriving in messages from associated DSPs are forwarded by the IOP runtime library's event dispatch function. This makes a decision based on a runtime configuration variable of whether to route events to a remote API server (e.g. the Host), or to perform a callback to a local user application's function handler.
- **Fast packet router**: The heart of the TASK VoIP capability of the TASK-6000 software is the Fast Packet Router. This functionality is built in to the IOP runtime library with a custom Ethernet device driver and corresponding components in the DSP software. The Fast Packet Router can forward over twenty thousand packets per second of voice data to and from the DSPs on each IOP. It does this while consuming less than 40 percent of the CPU time and bus bandwidth of the IOP's processor. The DSPs build and parse Ethernet frames according to configuration parameters specified by the user application via specialized API functions.

Peripheral device driver libraries

TASK-6000 software provides a basic set of libraries for controlling the on-board peripheral devices, currently comprising the Siemens MUNICH128 128-channel HDLC (High-level Data Link Controller), the Siemens QFALC (Quad Framing And Line Interface Component), and the Lucent Technologies T8105 TDM (Time Division Multiplex) switch.

- **HDLC**: The MUNICH128 supports up to 128 channels of HDLC. The TASK-6000 driver library provides functions that configure, enable, and disable channels and an ISR (Interrupt Service Routine) that provides callback function hooks to the user application for transmitting and receiving frames, and handling errors.
- **E1/T1 line interface**: The TASK-6000 driver library provides functions that configure, enable, and disable each of the device's four E1/T1 ports. The device must be operated as all E1 or all T1; you cannot mix line types. The driver provides an ISR that provides callback function hooks to the user application for transmitting and receiving frames, and handling errors.
- **TDM (Time Division Multiplex) switch**: The TASK-6000 driver library provides functions that establish and tear down sets of half-duplex time slot connections from one to any of the TDM resources on the board. TDM resources include serial ports on the DSPs, E1/T1 framers, and the H.110 backplane bus. The driver provides an ISR that provides a callback function hook to the user application for handling errors.

DSP application

The standard TASK-6000 DSP application comes with a variety of voice codecs, telephony algorithms, including echo canceller and tone detector/generators. It also includes RTP (Real-time Transport Protocol) send and receive functions as well as a jitter buffer, and packet build and parse functions for the system's Fast Packet Routing.

The DSP application is built on top of the RadiSys TASK kernel for the C6x family of DSPs. The underlying kernel provides deterministic thread switching, RAM memory management, and inter-processor message-passing functions.

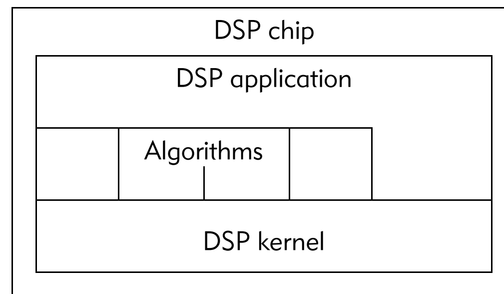


Figure 2-1. DSP application on the DSP chip

Utilities

TASK provides these tools that you use to develop TASK applications:

sp6k_util.exe

This is a development tool that provides host access to the memory and devices on the IOP. It includes functions to reset boards, read and write memory on the IOP, dump internal trace buffers, and so on.

rmondb.exe

rmondb, the RadiSys version of Intel's mondb.exe Win32 Console utility, connects a Windows NT host to the i960 Monitor over a PCI bus or serial port via the Host debugger interface. This program includes a GUI for use by diagnostics applications.

Interfaces

TASK-6000 software interfaces to the external environment using the methods described in this section.

Internet Protocol (IP)

The Fast Packet Router and the VxWorks IP stack provide flexible communication to other hosts and media gateways on an IP network. You can configure the Fast Packet Router to route UDP frames to individual DSPs based on their UDP port

number. All other traffic passes through to the VxWorks IP stack, so you can implement protocols such as RTCP in the IOP.

Applications on the IOP can communicate with remote hosts (or the local host) via IP, and the TASK-6000 IOP runtime library also contains functionality for extending its API to a remote IP host.

Public Switched Telephone Network (PSTN)

TASK-6000 software interfaces with the PSTN using the features of the E1/T1 line interface and HDLC controllers, which allow for implementations of ISDN and Robbed Bit call control applications. The DSP algorithms support R1/R2 signaling, DTMF generation and detection, echo cancellation, along with various industry-standard voice codecs.

Application Programming Interface (API)

The TASK-6000 API is presented at the IOP and is also reflected on the Host. In its baseline form the API provides the tools you use to:

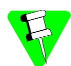
- Map a physical channel (T1 or E1 DS0 or H.110 channel) to a DSP's virtual channel.
- Configure a DSP channel with a particular codec, echo canceller, tone detectors and generators.
- RTP encode and decode a channel.
- Map an RTP-encoded channel to an RTP socket so that the socket is transparently (to the user) connected to the DSP channel.

The API also includes provisions for non-voice data types, such as T.38 traffic and V.90 data. Finally, the API is designed to be easily extensible so that you can add future services with minimal impact to existing applications.

Objects

TASK-6000 defines a hierarchy of objects that describe software operation:

- **Slot**: Refers to a given IOP in a system. Slots are numbered from zero.



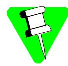
IOP functions do not specify a slot; all functions are assumed to operate on the current slot. Host functions include the slot parameter in all relevant functions.

- **Unit**: Usually refers to a DSP in a relative numbering system for each IOP in the system. Units are numbered from zero. For IOP-based services such as stCas, however, the unit parameter specifies the framer unit number.
- **Channels**: Usually refers to a virtual channel (see the following subsections). For IOP-based services such as stCas, however, the channel parameter specifies the physical channel. Channels are numbered from zero.

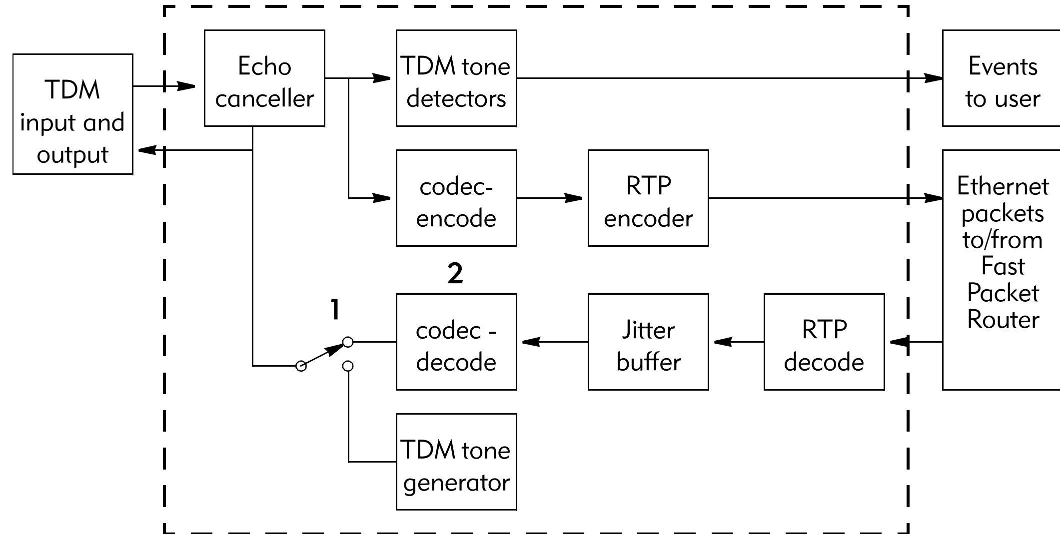
- **Physical channels:** Physical Channels are duplex TDM streams (DS0s) associated with a hardware port. A device type (T1, E1, or H.100), unit number, and timeslot uniquely identify a physical channel. Physical channels are referred to only when setting up a call (via) or when sending or receiving Channel Associated Signaling via the service.
- **Virtual channels:** Virtual Channels are duplex paths through a DSP. They are referenced by DSP unit number and channel. On one side of the DSP this refers to the TDM timeslot connected to a TDM switch, and on the other side to the channel number reported in network messages carrying packetized data. All of the TASK services except those that deal with physical channels (currently only stCas) are controlled and monitored via virtual channels.
- **Channel groups:** Virtual Channels are grouped by the DSP application based on their function. For example, voice and fax. Each TASK module operates on groups of virtual channels. The user application can change the channel grouping dynamically using TASK API functions.

Services

Services are processing entities that operate on a data or signaling stream. Services can be individually enabled and disabled, and have a particular position where they fit into the data stream when enabled. When disabled they are bypassed as necessary so that the data streams continue. Examples of services include voice codecs, tone detectors and generators, packet builders and parsers, and alarm detectors and generators.



For detailed information about TASK services, see Appendix F, Service descriptions.



1 If the tone generator is enabled and active (cadence in progress) then the tone generator output is connected to the TDM output. If the tone generator is inactive (disabled or cadence finished) then the codec is connected to the TDM output.

2 If the codec is disabled, a silent sound source is used in its place.

A second tone generator and tone detector exist in the opposite directions, but are omitted from this diagram for clarity.

Figure 2-2. Data path services

- **Setting up a service:** Most services must be configured before operation. To configure a service, the application fills in a configuration structure and passes the structure to the upConfigService function. This then uses the underlying message API to get the configuration data to the correct processor and set up the service.

  The configuration of a service is private to a particular slot, unit, and channel, and is persistent. The application can configure a service just once during initialization, and then enable and disable the service as needed, or the application can reconfigure the service every time it is needed. Configuring a service also enables it.

- **Caveats:** To ensure that your applications are compatible with future versions of TASK, take care to follow these guidelines:

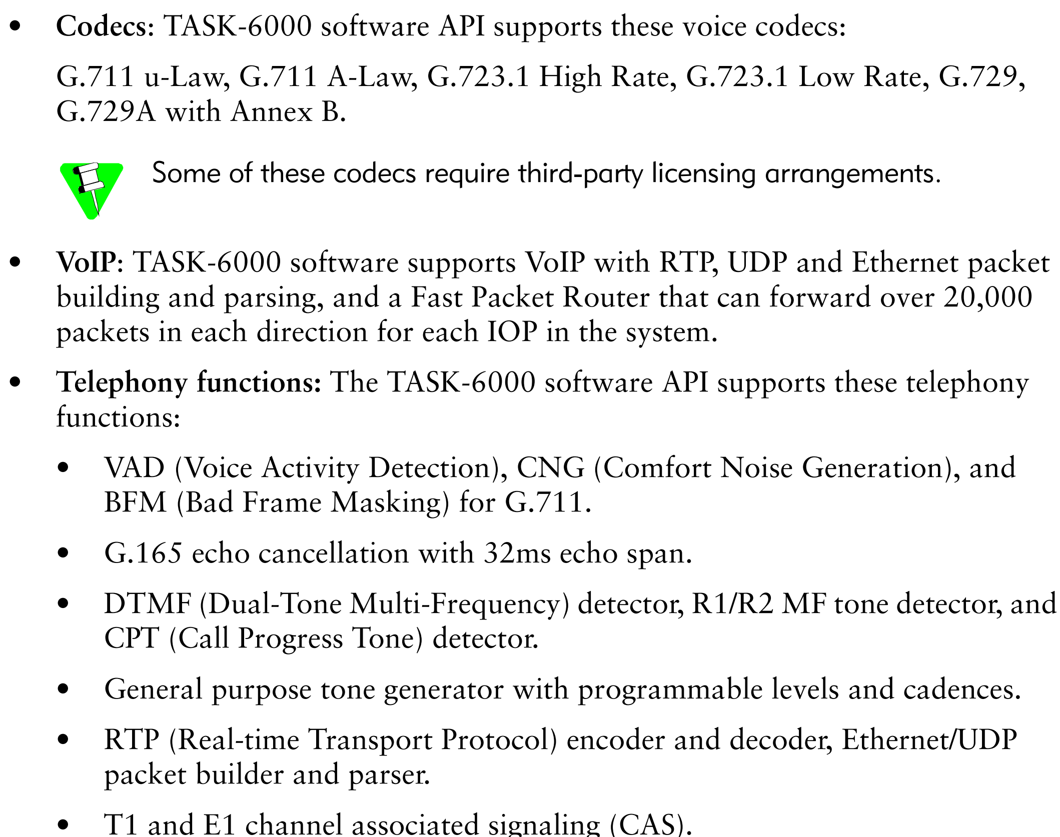
  - Reference data structure elements by name.
  - Allocate memory for data structures using the sizeof(data structure type) operator.
  - Clear the memory allocated for the data structure before filling in the elements, so that any new elements are set to zero.

Example

```
UP_CONFIG_SVC_MSG_UT *ptCfg;
ptCfg = calloc(sizeof(UP_CONFIG_SVC_MSG_UT));
ptCfg.tCodecConfig.eCodec = ctG711mu;
ptCfg.tCodecConfig.tCodecParams.tG711Params.eLaw = enumMULAW;
ptCfg.tCodecConfig.tCodecParams.tG711Params.eVadEnable = enumDisabled;
ptCfg.tCodecConfig.tCodecParams.tG711Params.eBfmEnable = enumEnabled;
```

- **Codecs**: TASK-6000 software API supports these voice codecs:

  G.711 u-Law, G.711 A-Law, G.723.1 High Rate, G.723.1 Low Rate, G.729, G.729A with Annex B.

  Some of these codecs require third-party licensing arrangements.

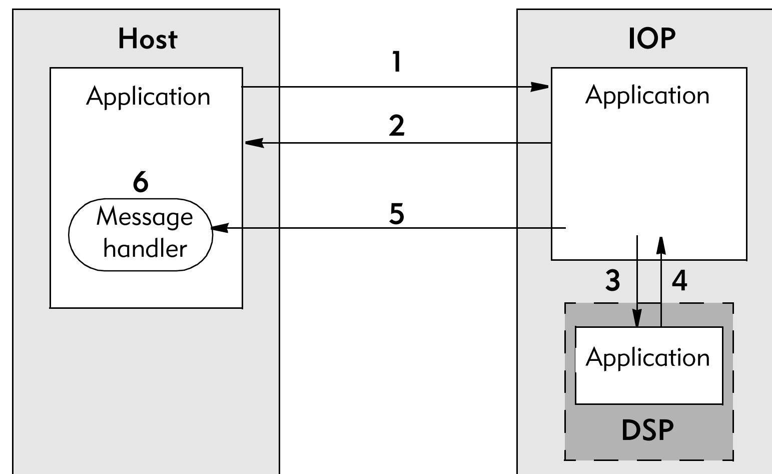
- **VoIP**: TASK-6000 software supports VoIP with RTP, UDP and Ethernet packet building and parsing, and a Fast Packet Router that can forward over 20,000 packets in each direction for each IOP in the system.
- **Telephony functions:** The TASK-6000 software API supports these telephony functions:
  - VAD (Voice Activity Detection), CNG (Comfort Noise Generation), and BFM (Bad Frame Masking) for G.711.
  - G.165 echo cancellation with 32ms echo span.
  - DTMF (Dual-Tone Multi-Frequency) detector, R1/R2 MF tone detector, and CPT (Call Progress Tone) detector.
  - General purpose tone generator with programmable levels and cadences.
  - RTP (Real-time Transport Protocol) encoder and decoder, Ethernet/UDP packet builder and parser.
  - T1 and E1 channel associated signaling (CAS).

Inter-processor communication

TASK-6000 software communicates between processors in the system, and in certain circumstances, between threads on the same processor via messages. The messages structures are defined in the API, and functions exist to send and receive messages on the Host, IOP, and DSP.

- **Messages:** Messages are generally of these types:
  - **Commands**: The application sends commands to configure, enable, and disable services, and to request status from them. Commands are also used to configure and operate the peripheral devices on the IOP.
  - **Events**: TASK software uses events, an asynchronous message mechanism, to prevent the application thread from blocking on API functions. When an API function requires another system processor to do some of the processing, the API function creates a message, sends it to the other processor using TASK's messaging feature, and immediately returns to the application thread. The message is delivered to the destination processor

(IOP or DSP), where it is interpreted and acted upon. In response to some of these messages, the destination processor communicates back to the application by means of an event.



1. A function from the Host application sends a message to the IOP.
2. The function immediately returns to the Host, reporting that the message was sent successfully.
3. The IOP forwards the message to the DSP.
4. The DSP receives the message, interprets it, performs required actions, and then reports the status as an event to the IOP.
5. The IOP forwards the event to the Host's Message Handler.
6. The Host's Message Handler receives and analyzes the event, performing any necessary action.

Figure 2-3. Using events and message handlers

The application must register an appropriate event handler callback function to receive the events. It is the responsibility of the user application to keep track of which commands are outstanding and to associate the returned events with their originating function calls. The events contain identification information that can be helpful for this purpose, such as the event message header which contains the slot, unit, and channel numbers, as well as the event.

Control: Some events notify the application of the deferred results of command messages. These can include acknowledgement or error reports.

Status: Events are also generated asynchronously by the operation of various services in the system. For example, a DTMF detector on a DSP generates events at the start and stop of a detected tone.

- **Callback functions**: TASK software uses a simple mechanism to send messages asynchronously to the user application. Messages are created by the TASK software and then a user-designated handler function in the application is called, passing a pointer to the message as a parameter. The user application must register a handler function for each subsystem that can generate a callback. This is done by means of API functions.

It is important to note that the user's event handler function executes in one of the TASK threads on the local processor (IOP or host). The user should ensure that:

- The handler function does not spend excessive amounts of time before returning to TASK.
- The function always returns.

It should also be noted that resource locks can occur if the user's handler function calls another TASK function that may contend for an internal TASK resource that is already held by the original TASK thread. This situation can be avoided by employing additional user threads and some form of signaling (using the appropriate Operating System features). The TASK API is "multi-thread safe" and, in general, telecom applications benefit from having as many threads as practical to accomplish their operation.

3 Installing and configuring TASK-6000 software

This chapter explains how to install your TASK-6000 software on a Windows NT system.

You can order TASK software in these configurations:

- **TASK-6000 runtime kit**: Provides the software required to run TASK applications.
- **TASK-6000 development kit**: Provides the tools required to write and debug TASK Host and IOP applications.
- **TASK-6000 DSP development kit**: Provides the tools required to write and debug TASK DSP applications. This software also includes the TASK-6000 development kit.

The installation program (t6kinst.exe) installs the components needed for your TASK software.

Requirements

Installing TASK software on your system requires:

- A PC with a processor similar to or better than Pentium II 266 MHz.
- A minimum 32 MB of DRAM.
- A minimum 20 MB disk space.
- A PC that runs Windows NT, version 4.0, with Service Pack 5.

Before you begin

- Ensure that you are logged on with Administrator privileges.
- Exit all programs prior to installing TASK software.
- If your system already has TASK software installed, you must uninstall it using one of these methods:
    - If your system has TASK software version 2.x, remove it when prompted during installation.
    - If the system has an earlier version of TASK software, follow the steps described in *Uninstalling TASK software* on page 17. After removing the software, continue with the installation.

Running the install program

To install TASK software:

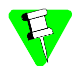
1. Insert the TASK software CD-ROM into the CD-ROM drive.
2. Double-click t6kinst.exe from the Windows NT Explorer, then select one of these:
   - **Read release notes**: Select this option to run Notepad and display relnotes.txt, the release notes file.

     This document contains the latest information about the release. You may find it useful to print the release notes for future reference. When you finish reading the file, select File>Exit. The install program displays.

     If you don't read the release notes at this time, you can access it at any time from the *InstallDir*\TASK6000\doc directory.
   - **Install TASK6000**: Select this option to install TASK software.

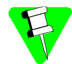
     If a version of TASK 2.x software is already installed, the install program prompts you to specify the installation method. To continue, go to step 3.

     Otherwise, the install program displays a welcome screen followed by the TASK software license agreement. To continue the installation, go to step 4.
   - **Exit Installation**: Select this option to leave the install program without installing TASK software.
3. Select one of these:
   - **Continue after Uninstall**: The install program uninstalls the software. The uninstall removes the TASK software.

     The install program does not remove Hot Swap software. For information about uninstalling Hot Swap software, see *Uninstalling Hot Swap* on page 18.

     When you uninstall TASK software using this method, Hot Swap should also be reinstalled from the CD-ROM. This is explained in step 6.

     If the uninstaller prompts to reboot, cancel the operation and proceed with the installation.

     After uninstalling TASK software, the installation continues.
   - **Install Additional components**: Installs the TASK software components that you select. Select this option to add components not selected during the original install.

     The install program copies any duplicate file (a file with the same name and path as a file to install) to the installation directory's BACKUP folder.
   - **Exit**: The install program exits without installing TASK software.

   If you selected either of the first two options, the install program displays a welcome screen, followed by the TASK software license agreement.

4. Read the agreement, then select one of these:

   - **Agree:** Selecting this option means you accept the terms of the software license agreement. The install program continues.
   - **Cancel:** Selecting this option means you do not accept the terms of the software license agreement. The install program exits.

   If you agreed with the software licensing agreement, the install program prompts you to identify a destination, or installation, directory.

5. Select a destination directory for TASK software files, then click the Next button. The default is C:\.

   Makefiles for Host and IOP development assume that the installation directory is C:\.

   When you choose C:\ as the installation directory, the install program extracts certain files into the C:\Tornado2 directory. It is assumed that the Tornado software is also installed under *InstallDir*\Tornado2 directory, such that the files from the CD-ROM are extracted into the Tornado installation directory.

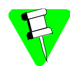
   If Tornado was installed in a directory other than C:\ (for example, D:\xyz\tornado2), or the software cannot install in the C:\ directory for other reasons (such as space constraints), choose another directory for the TASK6000 installation (for example, *InstallDir*) and copy the files installed under *InstallDir*\Tornado2 to the Tornado installation directory (for instance, copy files under C:\Tornado2 to D:\xyz\tornado2).

   For Makefile setting changes required when TASK6000 is installed in a directory other than C:\, see *IOP application development* on page 34.

   The install program prompts you to choose the TASK components you want to install.

6. Select the TASK components you want to install, then click the Next button:

   - **Hot Swap:** Select this option if your system is Hot Swap capable.

     
     Select this option if you are installing TASK software, and version 2.x was uninstalled prior to installation.
     The TASK uninstall program does not uninstall Hot Swap, but does remove certain registries required by TASK to support Hot Swap. Selecting the Hot Swap option re-installs these registries.

   - **Runtime kit:** Installs the software required to run TASK applications.
   - **Development kit:** Installs the files required to write and debug TASK Host and IOP applications. This option displays only in the TASK-6000 development kit configuration.
   - **DSP development kit:** Installs the files required to write and debug TASK DSP applications. This option includes the Development kit and displays only in the TASK-6000 DSP development kit configuration.

   The install program prompts you to select a program group name.

7. Do one of these:
   - Accept the default group name. This method may reduce confusion as the program name will match TASK documentation.
   - Enter a program group name. This method can provide a program name that has meaning to you.
8. When you click the Next button, install creates the directory you specified, then installs the TASK software files.

   If the installation program is cancelled at any point after this step, the installation program prompts you to perform rollback. Select "Yes" to clean up the partial installation.

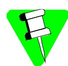
   If you chose to install RadiSys Hot Swap for Windows NT 4.0 software, the TASK installation program starts the Hot Swap installation program, and the Hot Swap installation welcome screen displays. To continue, go to step 9.

   Otherwise, the install prompts you to view the relnotes.txt file. This file contains the latest information about the TASK6000 2.x release. To continue the installation, go to step 13.
9. Select one of these:
   - **Next**: Installs Hot Swap software.
   - **Cancel**: Exits only the Hot Swap installation. The TASK installation continues.

   The install program prompts you to identify a destination directory.
10. Select a destination directory for Hot Swap software files, then click the Next button. The default is C:\Program Files\HotSwap.

    The install program prompts you to confirm your choices before installation.
11. Select one of these:
    - **Next**: Continues installing Hot Swap software.
    - **Back**: Returns to the previous screen where you can change installation information.
    - **Cancel**: Exits only the Hot Swap installation. The TASK installation continues.

    After selecting Next, the install program installs the Hot Swap software, and then displays the readme.txt file. Select next after reading the displayed text.

    The install program prompts you to restart the computer. Skip this step by selecting "Cancel" since the TASK installation program provides you with the same option at the end of installation.

    The installation is now complete, and the installation program prompts you to select "Finish".
12. Click the "Finish" button.

Next, install prompts you to view the relnotes.txt file. This file contains the latest information about the TASK6000 2.x release.

13. Choose one of these:
    - **View Release notes**: Notepad runs and the relnotes.txt file displays. You may find it useful to print the relnotes.txt file for future reference. When you finish reading the file, select File>Exit. After reading the document, you can choose to Exit the installation by selecting Exit Install.
    - **Exit install**: If you don't read the relnotes.txt file at this time, you can access it at any time from the *InstallDir*\TASK6000\doc directory.
14. Restart your system using one of these methods:
    - Select OK at the last screen. The install program restarts your system.
    - Select Cancel to restart your system at a later time.

You must restart your system before you can run TASK user applications.

If you installed the DSP Development kit, you must also define TASK_DIR, an environment variable, as one of the System variables. Set the variable to the *InstallDir*\TASK6000 directory path by selecting Start>Settings>Control Panel>System>Environment. If the variable is already defined, change it to the new installation path. Otherwise, add the variable with the current installation path under the values field. For example, if the installation path was at C:\, then set the TASK_DIR to C:\TASK6000.

Uninstalling TASK software

Automatic uninstall (recommended)

To automatically uninstall TASK software, use one of these methods:

- Go to the Control panel and select Add/Remove programs, then select one of these from the list that displays:
    - TASK6000 (displays if Task 2.0 was previously installed).
    - TASK-6000 and SPIRIT-6000 BSP (displays if Task 1.2 Beta 10 was previously installed).
- During installation, select Continue after Uninstall. This option is available only if the TASK software you want to uninstall is version 2.0 or later.

The uninstall program removes all the files installed by the CD-ROM, including files with same name as those installed by the CD-ROM—even if updated after installation. After uninstalling TASK software, only new files added to the installation directory remain.

The TASK uninstall program does not uninstall Hot Swap, but does remove certain registries required by TASK to support Hot Swap. If you install TASK again, you must select the Hot Swap option to re-install these registries.

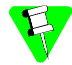
Uninstalling Hot Swap

If you installed Hot Swap, you must also complete these steps:

1. Select Start>Settings>Control Panel>Add\Remove Programs.
2. Select the Hot Swap 1.10 for Windows NT option.
3. Click the Add/Remove button.

Manual uninstall procedure

The TASK uninstall program relies on files created during installation. If these files are accidentally deleted, you must manually uninstall the software.



Use the manual uninstall only when you cannot uninstall the software with the Automatic Uninstall methods.

To manually uninstall TASK software:

1. Remove TASK software files, located in the directory you specified during installation.

   Two subdirectories, TASK6000 and Tornado2, must be removed. However, if these directories existed prior to installation and had some files already present in them, or were added after the installation, you should not remove those files.

   For a list of files added by the installation program, see *TASK-6000 files* on page 20.

2. Remove TASK entries from Start menu:

   A. Right-click the Start menu button, then select Explore all users. The Start Menu folder displays.

   B. Right-click the RadiSys TASK6000 icon and select Delete.

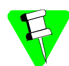
3. Stop the i960 driver:

   A. Select Start>Settings>Control Panel>Devices and select I960RP from the device list.

   B. Select Stop to halt the driver.

4. Remove the following registry keys using regedit:

   HKEY_CLASSES_ROOT\TASK6000
   HKEY_LOCAL_MACHINE\System\Services\CurrentControlSet\Services\i960RP
   HKEY_LOCAL_MACHINE\System\Services\ControlSet001\Services\i960RP
   HKEY_LOCAL_MACHINE\System\Services\ControlSet002\Services\i960RP
   HKEY_LOCAL_MACHINE\System\Services\ControlSet003\Services\i960RP
   HKEY_LOCAL_MACHINE\Software\RadiSys\HotSwap\Configuration\i960RP

   Your system may not include all these registries.

5. Remove this driver file:

   `%SystemRoot%\system32\drivers\i960rp.sys`

6. Remove Hot Swap as described in *Uninstalling Hot Swap* on page 18.

7. Reboot the system.

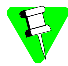
TASK-6000 files

This section lists all files installed as part of TASK-6000 software. All the file paths are listed relative to the directory of installation.

For a detailed list of files by product configuration, go to the topic below:

For information about...	Go to this page...
Hot Swap	20
Runtime kit	21
Development kit	23
DSP development kit	26

Hot Swap



For a complete description of RadiSys Hot Swap for Windows NT software, see *Hot Swap for Windows NT* (RadiSys part no. 07-1080-00). The Hot Swap install program copies this file to *InstallDir*\hotswap.pdf.

Type	Files
Documentation	*InstallDir*\readme.txt *InstallDir*\hotswap.pdf
Install utility	*InstallDir*\setup.exe
Uninstall utility	*InstallDir*\unwise.exe *InstallDir*\unwise.ini
Drivers	*InstallDir*\hbus.sys %SystemRoot%\system32\drivers\hsbus.sys
Services	*InstallDir*\bin\hsmgr.exe %SystemRoot%\system32\hsmgr.exe
Hot Swap API	*InstallDir*\hsmgrint.dll *InstallDir*\inc\hsmgrint.h *InstallDir*\lib\hsmgrint.lib %SystemRoot%\system32\hsmgrint.dll

InstallDir is the directory in which RadiSys Hot Swap software is installed.

Runtime kit

Type	Files
Documentation	*InstallDir*\TASK6000\Doc\Taskman.ico *InstallDir*\TASK6000\Doc\TaskMan.pdf *InstallDir*\TASK6000\Doc\sp6040_hwref.pdf *InstallDir*\TASK6000\Doc\relnotes.txt
Installation management information	*InstallDir*\TASK6000\misc\UNWISE.EXE *InstallDir*\TASK6000\misc\spdiag.ico *InstallDir*\TASK6000\misc\hwref.ico *InstallDir*\TASK6000\misc\hsetup.exe *InstallDir*\TASK6000\misc\u_guide.ico *InstallDir*\TASK6000\misc\uninst.ico *InstallDir*\TASK6000\misc\Taskdiag.ico *InstallDir*\TASK6000\misc\Taskman.ico *InstallDir*\TASK6000\misc\Taskact.ico *InstallDir*\TASK6000\misc\installed_comp *InstallDir*\TASK6000\misc\UNWISE.INI *InstallDir*\TASK6000\misc\INSTALL.LOG
DSP mulcoders	*InstallDir*\TASK6000\Dsp\UpaApp\mulcoder.ntb *InstallDir*\TASK6000\Dsp\UpaApp\mulcoder.out *InstallDir*\TASK6000\Dsp\UpaApp\mulcoder.ttb
IOP image	*InstallDir*\Tornado2\target\config\sp6k\vxWorks
Windows NT DLLs	*InstallDir*\TASK6000\Host\Nt\Bin\hsmgrint.dll *InstallDir*\TASK6000\Host\Nt\Bin\TASKHOST.dll
Utilities	*InstallDir*\TASK6000\Host\Nt\Bin\sp6k_util.exe
Windows NT driver	*InstallDir*\TASK6000\Host\Nt\Bin\I960RP.sys %SystemRoot%\system32\drivers\I960rp.sys

Type	Files
Diagnostics	*InstallDir*\TASK6000\Host\Nt\Bin\rmondb.exe
	InstallDir\TASK6000\Host\Nt\Bin\diag
	InstallDir\TASK6000\Host\Nt\Bin\diag\c6xcodec
	InstallDir\TASK6000\Host\Nt\Bin\diag\c6xcodec.hlp
	InstallDir\TASK6000\Host\Nt\Bin\diag\c6xmem
	InstallDir\TASK6000\Host\Nt\Bin\diag\c6xmem.hlp
	InstallDir\TASK6000\Host\Nt\Bin\diag\e1t1dq
	InstallDir\TASK6000\Host\Nt\Bin\diag\e1t1dq.hlp
	InstallDir\TASK6000\Host\Nt\Bin\diag\hs_adiag
	InstallDir\TASK6000\Host\Nt\Bin\diag\Hs_adiag.hlp
	InstallDir\TASK6000\Host\Nt\Bin\diag\hs_targ
	InstallDir\TASK6000\Host\Nt\Bin\diag\iopmem
	InstallDir\TASK6000\Host\Nt\Bin\diag\iopmem.hlp
	InstallDir\TASK6000\Host\Nt\Bin\diag\iopper
	InstallDir\TASK6000\Host\Nt\Bin\diag\iopper.hlp
	InstallDir\TASK6000\Host\Nt\Bin\diag\t8100
	InstallDir\TASK6000\Host\Nt\Bin\diag\t8100.hlp
	InstallDir\TASK6000\Host\Nt\Bin\diag\tiolan
	InstallDir\TASK6000\Host\Nt\Bin\diag\Tiolan.hlp
	InstallDir\TASK6000\Host\Nt\Bin\diag\tiowan
	InstallDir\TASK6000\Host\Nt\Bin\diag\Tiowan.hlp

InstallDir is the directory in which RadiSys TASK software is installed.

Development kit

Type	Files
Documentation	*InstallDir*\TASK6000\Doc\Taskman.ico
	InstallDir\TASK6000\Doc\TaskMan.pdf
	InstallDir\TASK6000\Doc\sp6040_hwref.pdf
	InstallDir\TASK6000\Doc\relnotes.txt
Installation management information	*InstallDir*\TASK6000\misc\UNWISE.EXE
	InstallDir\TASK6000\misc\spdiag.ico
	InstallDir\TASK6000\misc\hwref.ico
	InstallDir\TASK6000\misc\hsetup.exe
	InstallDir\TASK6000\misc\u_guide.ico
	InstallDir\TASK6000\misc\uninst.ico
	InstallDir\TASK6000\misc\Taskdiag.ico
	InstallDir\TASK6000\misc\Taskman.ico
	InstallDir\TASK6000\misc\Taskact.ico
	InstallDir\TASK6000\misc\installed_comp
	InstallDir\TASK6000\misc\UNWISE.INI
	InstallDir\TASK6000\misc\INSTALL.LOG
Example Applications: Host	*InstallDir*\TASK6000\Examples\Upa\UpaHostApp\Bin\UpaHostApp.exe
	InstallDir\TASK6000\Examples\Upa\UpaHostApp\UpaHostApp.h
	InstallDir\TASK6000\Examples\Upa\UpaHostApp\UpaHostApp.dsw
	InstallDir\TASK6000\Examples\Upa\UpaHostApp\UpaHostApp.dsp
	InstallDir\TASK6000\Examples\Upa\UpaHostApp\UpaHostApp.c
Example Applications: IOP	*InstallDir*\TASK6000\Examples\Upa\UpaIopApp\IopAppLoader.c
	InstallDir\TASK6000\Examples\Upa\UpaIopApp\UpaIopApp.c
	InstallDir\TASK6000\Examples\Upa\UpaIopApp\UpaIopApp.dsp
	InstallDir\TASK6000\Examples\Upa\UpaIopApp\UpaIopApp.dsw
	InstallDir\TASK6000\Examples\Upa\UpaIopApp\Bin\IopAppLoader.exe

Type	Files
IOP libraries and object files	*InstallDir*\TASK6000\Iop\VxWorks\Lib\DspIo.a
	InstallDir\TASK6000\Iop\VxWorks\Lib\E1t1.a
	InstallDir\TASK6000\Iop\VxWorks\Lib\Hdlc.a
	InstallDir\TASK6000\Iop\VxWorks\Lib\IopDma.a
	InstallDir\TASK6000\Iop\VxWorks\Lib\T8100.a
	InstallDir\TASK6000\Iop\VxWorks\Lib\TaskIop.a
	InstallDir\TASK6000\Iop\VxWorks\Lib\UpaIop.a
	InstallDir\Tornado2\target\config\sp6k\dataSegPad.o
	InstallDir\Tornado2\target\config\sp6k\sysALib.o
	InstallDir\Tornado2\target\config\sp6k\sysLib.o
	InstallDir\Tornado2\target\config\sp6k\usrConfig.o
	InstallDir\Tornado2\target\config\sp6k\if_fei.o
	InstallDir\Tornado2\target\config\sp6k\Rominit.s
	InstallDir\Tornado2\target\config\sp6k\Sysalib.s
	InstallDir\Tornado2\target\config\sp6k\Pciutil.o
	InstallDir\Tornado2\target\config\sp6k\Rppcilib.o
Host library	*InstallDir*\TASK6000\Host\Nt\lib\taskhost.lib

Type	Files
Sources	*InstallDir*\TASK6000\Iop\VxWorks\Application\taskIopApp.c
	InstallDir\Tornado2\target\config\sp6k\sysSerial.c (null file)
	InstallDir\Tornado2\target\config\sp6k\sysNetif.c (null file)
	InstallDir\Tornado2\target\config\sp6k\Syslib.c (null file)
	InstallDir\Tornado2\target\config\sp6k\sysEeprom.c (null file)
	InstallDir\Tornado2\target\config\sp6k\rpQueueLib.c (null file)
	InstallDir\Tornado2\target\config\sp6k\Rppcilib.c(null file)
	InstallDir\Tornado2\target\config\sp6k\Rpintlib.c (null file)
	InstallDir\Tornado2\target\config\sp6k\Rpdmalib.c (null file)
	InstallDir\Tornado2\target\config\sp6k\Pciutil.c (null file)
	InstallDir\Tornado2\target\config\sp6k\pciIomapLib.c (null file)
	InstallDir\Tornado2\target\config\sp6k\iopSupport.c (null file)
	InstallDir\Tornado2\target\config\sp6k\if_fei.c (null file)
	InstallDir\Tornado2\target\config\all\dataSegPad.c
	InstallDir\Tornado2\target\config\all\version.c
	InstallDir\Tornado2\target\config\all\usrConfig.c
	InstallDir\Tornado2\target\src\usr\usrLib.c
	InstallDir\Tornado2\target\config\sp6k\Target.bat
	InstallDir\Tornado2\target\config\sp6k\sysEeprom.h
	InstallDir\Tornado2\target\config\sp6k\startit.tcl
	InstallDir\Tornado2\target\config\sp6k\setit.tcl
	InstallDir\Tornado2\target\config\sp6k\sdm.h
	InstallDir\Tornado2\target\config\sp6k\rpQueueLib.h
	InstallDir\Tornado2\target\config\sp6k\Rppci.h
	InstallDir\Tornado2\target\config\sp6k\Rpint.h
	InstallDir\Tornado2\target\config\sp6k\Rpdmalib.h
	InstallDir\Tornado2\target\config\sp6k\Radtimer.h
	InstallDir\Tornado2\target\config\sp6k\Pcilib.h
	InstallDir\Tornado2\target\config\sp6k\pciIomapLib.h
	InstallDir\Tornado2\target\config\sp6k\Pci_devs.h
	InstallDir\Tornado2\target\config\sp6k\pc.h
	InstallDir\Tornado2\target\config\sp6k\I960rx.h
	InstallDir\Tornado2\target\config\sp6k\Dsputil.h
	InstallDir\Tornado2\target\config\sp6k\Config.h
	InstallDir\Tornado2\target\h\drv\netif\if_fei.h
	InstallDir\Tornado2\target\config\sp6k\Makefile.app
	InstallDir\Tornado2\target\config\sp6k\Makefile
	InstallDir\Tornado2\target\config\sp6k\saveobj
	InstallDir\Tornado2\target\h\make\defs.bsp

InstallDir is the directory in which RadiSys TASK software is installed.

DSP development kit

Type	Files
Documentation	*InstallDir*\TASK6000\Doc\Taskman.ico
	InstallDir\TASK6000\Doc\TaskMan.pdf
	InstallDir\TASK6000\Doc\sp6040_hwref.pdf
	InstallDir\TASK6000\Doc\relnotes.txt
Installation management information	*InstallDir*\TASK6000\misc\UNWISE.EXE
	InstallDir\TASK6000\misc\spdiag.ico
	InstallDir\TASK6000\misc\hwref.ico
	InstallDir\TASK6000\misc\hsetup.exe
	InstallDir\TASK6000\misc\u_guide.ico
	InstallDir\TASK6000\misc\uninst.ico
	InstallDir\TASK6000\misc\Taskdiag.ico
	InstallDir\TASK6000\misc\Taskman.ico
	InstallDir\TASK6000\misc\Taskact.ico
	InstallDir\TASK6000\misc\installed_comp
	InstallDir\TASK6000\misc\UNWISE.INI
	InstallDir\TASK6000\misc\INSTALL.LOG
Libraries	*InstallDir*\TASK6000\Dsp\RsysTask\Lib\kernel.lib
	InstallDir\TASK6000\Dsp\RsysTask\Lib\c6xrts.lib
	InstallDir\TASK6000\Dsp\UpaApp\Lib\Dtmf.lib
	InstallDir\TASK6000\Dsp\UpaApp\Lib\Tonedet.lib
	InstallDir\TASK6000\Dsp\UpaApp\Lib\rtp.lib
	InstallDir\TASK6000\Dsp\UpaApp\Lib\R1r2mf.lib
	InstallDir\TASK6000\Dsp\UpaApp\Lib\PktFifo.lib
	InstallDir\TASK6000\Dsp\UpaApp\Lib\G711.lib
	InstallDir\TASK6000\Dsp\UpaApp\Lib\Echocanc.lib
	InstallDir\TASK6000\Dsp\UpaApp\Lib\Cpm.lib
	InstallDir\TASK6000\Dsp\UpaApp\Lib\Tonegen.lib
	InstallDir\TASK6000\Dsp\UpaApp\Lib\Bfm.lib
	InstallDir\TASK6000\Dsp\UpaApp\Lib\Agcvox.lib

Type	Files
Sources	*InstallDir*\TASK6000\Dsp\UpaApp\src\include\VOXTypedef.h *InstallDir*\TASK6000\Dsp\UpaApp\src\include\Vadstr.h *InstallDir*\TASK6000\Dsp\UpaApp\src\include\UpaPacket.h *InstallDir*\TASK6000\Dsp\UpaApp\src\include\upadsp.h *InstallDir*\TASK6000\Dsp\UpaApp\src\include\UpaDef.h *InstallDir*\TASK6000\Dsp\UpaApp\src\include\typedef.h *InstallDir*\TASK6000\Dsp\UpaApp\src\include\Tongen.h *InstallDir*\TASK6000\Dsp\UpaApp\src\include\tonedet.h *InstallDir*\TASK6000\Dsp\UpaApp\src\include\tasktapi.h *InstallDir*\TASK6000\Dsp\UpaApp\src\include\task.h *InstallDir*\TASK6000\Dsp\UpaApp\src\include\Tapi.h *InstallDir*\TASK6000\Dsp\UpaApp\src\include\rtpDsp.h *InstallDir*\TASK6000\Dsp\UpaApp\src\include\r1r2proto.h *InstallDir*\TASK6000\Dsp\UpaApp\src\include\R1r2mf.h *InstallDir*\TASK6000\Dsp\UpaApp\src\include\R1r2def.h *InstallDir*\TASK6000\Dsp\UpaApp\src\include\Proto.h *InstallDir*\TASK6000\Dsp\UpaApp\src\include\IpUtil.h *InstallDir*\TASK6000\Dsp\UpaApp\src\include\G711.h *InstallDir*\TASK6000\Dsp\UpaApp\src\include\Ecproto.h *InstallDir*\TASK6000\Dsp\UpaApp\src\include\Eciostrc.h *InstallDir*\TASK6000\Dsp\UpaApp\src\include\Echostrc.h *InstallDir*\TASK6000\Dsp\UpaApp\src\include\Ecdefine.h *InstallDir*\TASK6000\Dsp\UpaApp\src\include\Dtmfu.h *InstallDir*\TASK6000\Dsp\UpaApp\src\include\DspFifo.h *InstallDir*\TASK6000\Dsp\UpaApp\src\include\defs.h *InstallDir*\TASK6000\Dsp\UpaApp\src\include\Def.h *InstallDir*\TASK6000\Dsp\UpaApp\src\include\cpmu.h *InstallDir*\TASK6000\Dsp\UpaApp\src\include\Cpmdef.h *InstallDir*\TASK6000\Dsp\UpaApp\src\include\Coder.h *InstallDir*\TASK6000\Dsp\UpaApp\src\include\BFMTypedef.h *InstallDir*\TASK6000\Dsp\UpaApp\src\include\BFMBasic_op.h *InstallDir*\TASK6000\Dsp\UpaApp\src\include\BFMasking.h

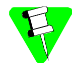
Type	Files
Sources (cont'd)	*InstallDir*\TASK6000\Dsp\UpaApp\src\include\basic_op.h
	InstallDir\TASK6000\Dsp\UpaApp\src\include\Agcvox.h
	InstallDir\TASK6000\Dsp\UpaApp\src\R1r2mf
	InstallDir\TASK6000\Dsp\UpaApp\src\R1r2mf\Zr1r2mf.c
	InstallDir\TASK6000\Dsp\UpaApp\src\R1r2mf\R1R2Tbl.h
	InstallDir\TASK6000\Dsp\UpaApp\src\R1r2mf\R1R2Stmchn.c
	InstallDir\TASK6000\Dsp\UpaApp\src\R1r2mf\R1r2findkey.c
	InstallDir\TASK6000\Dsp\UpaApp\src\msgDbg
	InstallDir\TASK6000\Dsp\UpaApp\src\msgDbg\msgDbg.c
	InstallDir\TASK6000\Dsp\UpaApp\src\Cpm
	InstallDir\TASK6000\Dsp\UpaApp\src\Cpm\Zcpm.c
	InstallDir\TASK6000\Dsp\UpaApp\src\Cpm\Cpmtbl.h
	InstallDir\TASK6000\Dsp\UpaApp\src\NoCoder
	InstallDir\TASK6000\Dsp\UpaApp\src\NoCoder\ZnoCoder.c
	InstallDir\TASK6000\Dsp\UpaApp\src\G711
	InstallDir\TASK6000\Dsp\UpaApp\src\G711\Zg711Enc.c
	InstallDir\TASK6000\Dsp\UpaApp\src\G711\Zg711Dec.c
	InstallDir\TASK6000\Dsp\UpaApp\src\G711\Vadtbl.dat
	InstallDir\TASK6000\Dsp\UpaApp\src\G711\law.dat
	InstallDir\TASK6000\Dsp\UpaApp\src\G711\Bfmtab.h
	InstallDir\TASK6000\Dsp\UpaApp\src\pkt
	InstallDir\TASK6000\Dsp\UpaApp\src\pkt\Zpktrecv.c
	InstallDir\TASK6000\Dsp\UpaApp\src\pkt\Zpktsend.c
	InstallDir\TASK6000\Dsp\UpaApp\src\Echo
	InstallDir\TASK6000\Dsp\UpaApp\src\Echo\zecho.c
	InstallDir\TASK6000\Dsp\UpaApp\src\Dtmf
	InstallDir\TASK6000\Dsp\UpaApp\src\Dtmf\Ztonedet.c
	InstallDir\TASK6000\Dsp\UpaApp\src\Dtmf\tonetbl.dat
	InstallDir\TASK6000\Dsp\UpaApp\src\Serial
	InstallDir\TASK6000\Dsp\UpaApp\src\Serial\zserial.c
	InstallDir\TASK6000\Dsp\UpaApp\src\Tonegen
	InstallDir\TASK6000\Dsp\UpaApp\src\Tonegen\dB2lin.h
	InstallDir\TASK6000\Dsp\UpaApp\src\Tonegen\Tonegen.dat
	InstallDir\TASK6000\Dsp\UpaApp\src\Tonegen\Ztonegen.c
	InstallDir\TASK6000\Dsp\UpaApp\src\UpaDspCtrl
	InstallDir\TASK6000\Dsp\UpaApp\src\UpaDspCtrl\UpaDspCtrl.c
Configuration files	*InstallDir*\TASK6000\Dsp\UpaApp\config.cfg
	InstallDir\TASK6000\Dsp\UpaApp\Board.cfg
Composer	*InstallDir*\TASK6000\Host\Nt\Bin\compose.exe

InstallDir is the directory in which RadiSys TASK software is installed.

4 Developing Host and IOP applications

This chapter describes how to write a simple Voice-Over-Internet-Protocol (VoIP) application. It explains what you must do to create IOP-based applications that run under the VxWorks real-time OS.

The sample application demonstrates how the SPIRIT board can conduct voice channels over a network and shows what's needed to establish a UPA channel with an enabled service, receive channels status, and control its behavior. Different DSP units, channel numbers and predefined set of codecs are used to demonstrate the SPIRIT board's ability.



For a copy of this application in ASCII format, see these files in the *InstallDir*/TASK6000/Examples/Upa/UpaIopApp directory:

- IopAppLoader.c
- UpaIopApp.c

Host-based application rules differ only slightly and are covered in a separate paragraph.

For information about... Go to this page...

Developing the Host application (IopAppLoader.c) 30
Initialize the Host driver 30
Set up message handlers 30
Initialize UPA structures 30
Load and run applications 30
Developing the IOP application (UpaIopApp.c) 31
Initialize the IOP driver 31
Set up message handlers 31
Configure services 32
Create data paths 32
Building Host and IOP applications 33
Naming conventions 34
Host application development 34
IOP application development 34
Sample code: Host application (IopAppLoader.c) 36
Sample code: IOP application (UpaIopApp.c) 39

Developing the Host application (IopAppLoader.c)

Initialize the Host driver

The Host software must first initialize the host driver, as shown in *Initializing the host driver* on page 36.

Set up message handlers

After initializing the host driver, the application sets a UPA event handler (a function to call whenever a UPA event occurs). In our sample application, the handler waits for an indication that the DSP is running (see the hostRunProg function description below).

The Host application creates Win32 events, ghIopRunning and ghDspRunning, to signal that a particular IOP or DSP is running, and then sets these flags:

- **etCommandAck**: Specifies whether DSP should acknowledge commands sent to it from IOP or host.
- **etEventForwarding**: Specifies whether IOP is to forward UPA events to the Host.
- **etLanControl**: Specifies whether the Host sends control messages to the IOP via the Local Area Network or via the PCI bus. You must set this field to `enumNoLanControl`; TASK does not support LAN-based messaging.

For sample code, see *Setting up message handlers* on page 36.

Initialize UPA structures

The host application initializes UPA structures on the host and informs the IOP of its IP address, as shown in *Initializing UPA structures* on page 36

Load and run applications

The host application retrieves system information, such as the number of boards plugged into the system, and the specifics of each board. For sample code, see *Loading and running applications* on page 37.

The VxWorks image resides in a file on the host. The host application loads each IOP with the same image. For sample code, see *Loading the VxWorks image on IOPs* on page 37.

The HOST waits for an indication from each IOP that it is running, as shown in *Waiting for IOP response* on page 38.

Finally the host loads the DSP image “mulcoder.out” to each DSP, sends a command to DSP to start the image, and waits until it receives feedback from DSP that it is running. For sample code, see *Loading mulcoder.out* on page 38.

Developing the IOP application (UpaIopApp.c)

The entry point for our sample IOP application is the function “idiag()”.

Initialize the IOP driver

idiag initializes the IOP driver by calling iopInit, which creates message queues, sets interrupt vectors, and so on, as shown in *Calling iopInit* on page 39

The upStart function initializes UPA structures on the IOP, as shown in *Calling upStart* on page 39

The IOP application then configures the on-board peripherals by doing the following:

- Assigns default values to T8100 registers.
- Sets T8100 clocks.
- Configures T1 structures.

For sample code, see *Configuring on-board peripherals* on page 40.

Set up message handlers

If the application is intended to react to various UPA events, it must install an event handler (callback function), as described in *Installing event handlers (callback functions)* on page 41.

In this case, CasEventMsgHandler is a function that establishes and tears down voice channels in response to OFF-Hook and ON-Hook events.

Upon UPA CAS event, a callback CasEventHandler() is called. It analyzes the ABCD bits of the CAS event to determine if it was an OFF HOOK or ON HOOK CAS event. Information about time-slot and the framer port on which the CAS event has happened is passed to CasEventHandler() function.

In the case of an OFF HOOK event, the functions CreateFastPacketOut and CreateFastPacketIn are called. CreateFastPacketOut enables transmit direction on a particular channel.

Since the SPIRIT board has four framer and four DSPs capable of running up to 24 channels, the sample application directly maps framers to DSPs and time-slots to virtual channel numbers. For example, if an OFF HOOK event happens on framer 1 time-slot 9, the application sets up channel 9 on DSP 1. This is accomplished by making the appropriate TDM connection using the IOP’s T8105 switch.

For sample code, see *Connecting the TDM to IOPs* on page 41.

Once the TDM connection is made, the application calls upEnableChannel to enable bidirectional voice data flow to and from the DSP, as described in *Enabling bi-directional voice data flow to and from the DSP* on page 41.

Configure services

At this point the application configures services on DSP for the outbound direction (toward the IP cloud). The sample application chooses a G.711 voice codec, Echo Cancellation, and RTP Encoder. For sample code, see *Configuring services* on page 42.

Create data paths

Once the services are configured, the application creates a data path between an outgoing RTP channel running on a DSP and the IOP driver's network packet routing component. For simplicity, the sample application assumes operation over a private LAN; each IOP assumes that its peer IP address is equal to its own address with the least-significant bit toggled. For sample code, see *Creating a data path between an outgoing RTP channel running on a DSP and the IOP driver's network packet routing component* on page 43.

The application calls CreateFastPacketIn to enable the receive direction on the channel. Time-slots are mapped to virtual channels in the same fashion as for CreateFastPacketOut and the T8105 switch makes another TDM connection for this direction of voice data flow. For sample code, see *Enabling the receive direction on the channel* on page 44.

The application then initializes the RTP decoder, as shown in *Initializing the RTP decoder* on page 45.

Finally the sample IOP application creates a data path between an inbound RTP channel running on a DSP and the IOP driver's network packet routing component, as shown in *Creating a data path between an inbound RTP channel running on a DSP and the IOP driver's network packet routing component* on page 45

The receive direction is now initialized.

When an OFF HOOK event occurs on both IOPs, the application establishes VoIP call across the Local Area Network.

When an ON HOOK event occurs on either IOP, the application tears down the channels by calling functions TerminateFastPacketOut and TerminateFastPacketIn, which disable services. As a final step, the application disables the virtual channel.

Building Host and IOP applications

The TASK6000 2.x distribution for Host and IOP development kit includes the Microsoft Visual C++ project files for Host applications and Makefiles for IOP applications. This section explains the development environment setup for Host and IOP applications. The next figure identifies the locations of the files that you need to use to compile the application.

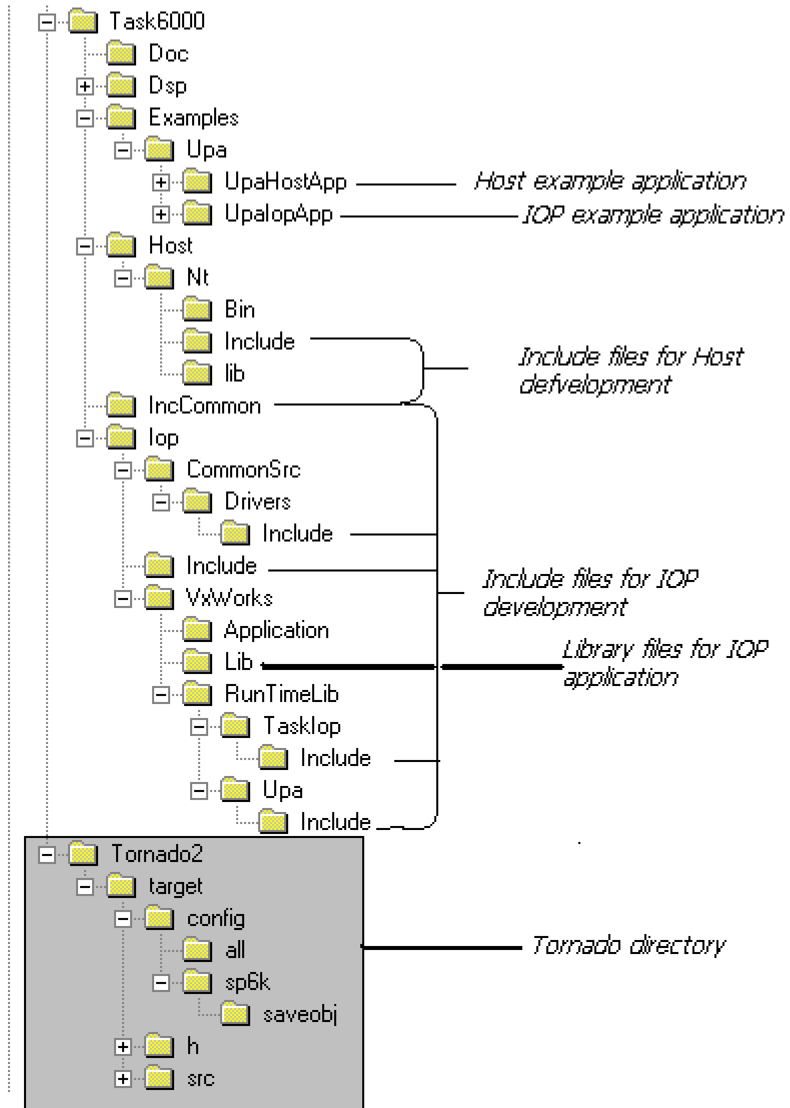


Figure 4-1. Location of files required to compile the application

Naming conventions

The directory under which TASK6000 and Tornado2 directories are installed is referred to as *InstallDir* in the following sections.

The directory under which tornado was installed, i.e., the directory under which the Tornado2 directory was installed by the original Tornado installation CD is referred to as *TornadoDir.*

Host application development

The host example application is available under *InstallDir*\TASK6000\Examples\Upa\UpaHostApp. The Microsoft Visual C++ project file contains the configurations required for making the build. Please make sure that you use similar configurations in the host application developed.

The include file directories are marked in the figure.

The following library files are available for Host application development

```
Taskhost.dll, hsmgrint.dll (InstallDir\TASK6000\Host\Nt\Bin)
Taskhost.lib (used to compile the application,
InstallDir\TASK6000\Host\Nt\lib)
```

The VxWorks image to be used with the host application is available in *InstallDir*\TASK6000\Iop\VxWorks\.

IOP application development

The application IopAppLoader.exe (located in *InstallDir*\TASK6000\Examples\Upa\UpaIopApp\) loads the bootable image (with the filename “vxworks”) to the IOP.

The library files required to make a bootable VxWorks image are available under *InstallDir*\Tornado2\target\config\sp6k.

A sample makefile is provided along with the example application (UpaIopApp). The sample Makefile provides example build and clean clauses for the example application. The installation CD extracts into the system a set of files meant to be under the Tornado distribution directory. The default setup expected from the installation system is that Tornado is installed in the C drive, and the Tornado2 directory path is C:\Tornado2.

The installation CD extracts files into this directory assuming that this directory already exists. If this directory is located in another location, say for example D:, install the software with destination directory as D: instead of C:. This ensures that the Tornado files from the CD are extracted to the right directory. If you cannot select a directory such that the Tornado files are extracted in the same path as the Tornado distribution files, you should manually copy these files from the installation directory into the Tornado distribution directory.

Copy all the files under *InstallDir*\Tornado2 to *TornadoDir*\Tornado2.

Change the makefiles such that PACE_PATH points to the location of *InstallDir*\TASK6000 and WIND_BASE as <TORNADO_ DIR>\Tornado2.

Special note for IOP applications

The CD contains certain object files extracted into the sp6k directory under *InstallDir*\tornado2\target\config. These files are required for building the VxWorks image. Use a clean clause similar to the one provided in the sample Makefile (task6000_clean). This backs up the object files before cleaning the build environment and the restores these files after the end of "clean" clause. This is required because the "clean" clause provided inside the makefiles of Tornado distribution clean all the object files including the one provided with the TASK6000 2.X distribution, and these object files cannot be re-built using the TASK6000 2.X distribution files.

Sample code: Host application (IopAppLoader.c)

Initializing the host driver

```
// Initialize host driver
   if(hostInit() != SUCCESS)
   {
      printf("\nUpaHostApp ==> ** ERROR ** Failed to initialize Host Driver - Exiting\n");
      return(UP_FAILURE);
   }
```

Setting up message handlers

```
// Set an event handler to process any events received from either the IOP or DSP
   upSetEventHandler(UpaEventHandler);

//  Set up an automatically resettable event, signalled when an individual IOP runs.
   ghIopRunning = CreateEvent( NULL, FALSE, FALSE, NULL );

//  Set up an automatically resettable event, signalled when an individual DSP runs.
   ghDspRunning = CreateEvent( NULL, FALSE, FALSE, NULL );

   memset(&stIopConfig, (int)NULL, sizeof(UP_IOPSYSCONFIG_ST));
   stIopConfig.etCommandAck      = enumEnabled;
   stIopConfig.etEventForwarding = enumEnabled;
   stIopConfig.etLanControl= enumNoLanControl;
```

Initializing UPA structures

```
// Call Up Start and pass any IOP Initialization code that may have been provided.
   if(upStart(&stIopConfig) != UP_SUCCESS)
   {
      printf("\nUpaHostApp==>  ** ERROR ** Unable to Start UPA\n");
   }
   else if(!iIOPS)
   {
      printf("\nUpaHostApp==>  UPA started with No Ethernet Adapters Initialized.");
   }
```

Loading and running applications

Retrieving system information

```
//  Get the System Information and Load and Run IOPs
    hostGetSystemInfo(&lIopCount, &lDspCount, &pstBoardInfo);
    printf("\nUpaHostApp ==>  System Information: %d IOP(s) Installed *** %d DSPs Installed\n", lIopCount, lDspCount);
```

Loading the VxWorks image on IOPs

```
    if(pstBoardInfo[lIOP].DeviceState == HS_DEVICE_NORMAL)
        {

            if(hostLoadIop(lIOP, "vxworks") != SUCCESS)
            {
                printf("\nUpaHostApp ==>  ** ERROR ** Unable to Load Program %s on IOP # %d\n", "VxWorks", lIOP);
                continue;
            }
            else
                printf("\nUpaHostApp ==>  Program %s loaded on IOP # %d\n", "VxWorks", lIOP);

        }

//      The board is either in a quiescent or powered off state.  Log that the board
//      cannot be loaded at this time and allow the board to be loaded when re-inserted.
        else
        {
            printf("\nUpaHostApp==>  ** ERROR ** IOP #%d Cannot be Loaded because it is in a %s state\n", lIOP,
(pstBoardInfo[lIOP].DeviceState == HS_DEVICE_QUIESCED ? "QUIESCENT" : "POWERED OFF"));
            continue;
//  The process of running an IOP is time consuming, so we poll the IOP several times
//  regarding its ready-to-run state.

        lIterations = 0;
        do
            {
//      Run the previously loaded IOP
            if(hostRunLoadedIops(lIOP) != SUCCESS)
                {
//          Catch a quick pause
                printf("\nUpaHostApp ==>  ** ERROR ** Unable to Run Program %s on IOP %d - RETRYING\n", "VxWorks", lIOP);
                Sleep(1000);  //wait 1s
```

```
                }
            else
                {
                printf("\nUpaHostApp ==>  IOP %d Running program %s!\n", lIOP, "VxWorks");
                bIOPRunning[lIOP] = TRUE;
                break;
                }

            }while(lIterations++ < RETRIES);
```

Waiting for IOP response

```
//      Wait for response from IOP before proceeding.
    WaitForSingleObject(ghIopRunning, UPA_WAIT_TIMEOUT);
```

Loading mulcoder.out

```
//  Go through the IOPs that are loaded and running and load and execute their
//  associated DSPs
    for(lIOP=0; lIOP < lIopCount; lIOP++)
    {
        if(!bIOPRunning[lIOP])
            continue;

//      Load and run the DSPs found on a specific IOP
        for(lDSP = 0; lDSP < pstBoardInfo[lIOP].NumberDSPs; lDSP++)
        {
//          Load the DSP using the program whose path was specified as an argument to this application
            if(hostLoadProg(lDSPOffset + lDSP, "mulcoder.out") != UP_SUCCESS)
            {
                printf("\nUpaHostApp ==>  ** ERROR ** Failed to load Program %s on Global DSP # %d\n","mulcoder.out",
lDSPOffset + lDSP);
                continue;
            }
            else
                printf("\nUpaHostApp ==>  Program %s loaded on Global DSP # %d\n", "mulcoder.out", lDSPOffset + lDSP);

//          Try to run the recently loaded DSP
            if(hostRunProg(lDSPOffset + lDSP) != UP_SUCCESS)
            {
```

```
              printf("\nUpaHostApp ==>  ** ERROR ** Unable to Run Program %s on Global DSP %d\n","mulcoder.out",
lDSPOffset + lDSP);
           }
           else if(WaitForSingleObject(ghDspRunning, UPA_WAIT_TIMEOUT) != WAIT_OBJECT_0)
            printf("\nUpaHostApp==>  ** ERROR ** DSP %d Timed Out when attenpting to run program %s!\n", lDSPOffset +
lDSP,"mulcoder.out" );

           else
           {
              printf("\nUpaHostApp ==>  Global DSP %d Running program %s!\n", lDSPOffset + lDSP,"mulcoder.out");

//            Increment the number of Startup Replies to be expected
              glStartReplies++;
           }
       } //for(lDSP = 0; lDSP < pstBoardInfo[lIOP].NumberDSPs; lDSP++)
//     Update the Global DSP offset
       lDSPOffset += pstBoardInfo[lIOP].NumberDSPs;

   }//for(lIOP=0; lIOP < lIopCount; lIOP++)
```

Sample code: IOP application (UpaIopApp.c)

Initializing the IOP drive

Calling iopInit

```
if (iopInit() != SUCCESS)
   {
        IopLogEvent(IOP_EVENT_LOG_TYPE_APP_HIGH, 0, 0, 0, "IopInit FAILED");
        return0(FAILURE);
   }
```

Calling upStart

```
if(upStart(NULL) != UP_SUCCESS)
   {
       IopLogEvent(IOP_EVENT_LOG_TYPE_APP_HIGH, 0, 0, 0, "upStart FAILED");
       return(FAILURE);
   }
```

Configuring on-board peripherals

```
iopControlPeripheral(TASK_T8100, CONFIG_T8100_DEFAULTS,(t_configArg *)&t8100ClkConfig);

// SET T8105 CLOCKS

   t8100ClkConfig.reference_clk_select = REF_LOCAL;
   t8100ClkConfig.netref_select = NETREF_LOCAL;
   t8100ClkConfig.fallback_clk_select = FB_LOCAL;

   t8100ClkConfig.netref_enable = TRUE;
   t8100ClkConfig.netref2_enable = FALSE;
   t8100ClkConfig.frame_clk_a_enable = TRUE;

   t8100ClkConfig.frame_clk_b_enable = TRUE;
   t8100ClkConfig.compat_clks_enable = TRUE;
   t8100ClkConfig.fallback_enable = FALSE;

   iopControlPeripheral(TASK_T8100, CONFIG_T8100_CLOCKS,(t_configArg *)&t8100ClkConfig);

// CONFIGURE T1

   T1_config_struct.line_coding= B8ZS;
   T1_config_struct.framing_mode= ESF;    /* extended superframe */
   T1_config_struct.line_build_out= DSX1_0_to_133_ft;
   T1_config_struct.idle_code= 0xff;  /* 255 */
   T1_config_struct.idle_channels= 0x00000000;
      /* do not insert idle code in any channels */
   T1_config_struct.payload_loopback_enable= FALSE;
   T1_config_struct.framer_loopback_enable= FALSE;
   T1_config_struct.local_loopback_enable= FALSE;
   T1_config_struct.remote_loopback_enable= FALSE;
   T1_config_struct.robbed_bit_signaling_enable= TRUE;

   for (framer_id = framer_1; framer_id<=framer_4; framer_id++ )
      // it has to a better way, like i<=num.ports
   {
   T1_config_struct.framer_id = framer_id;
   iopControlPeripheral(TASK_T1, CONFIG_T1,(t_configArg *)&T1_config_struct);
   }
```

Setting up message handlers

Installing event handlers (callback functions)

```
upSetEventHandler(&CasEventMsgHandler);
```

Connecting the TDM to IOPs

```
//  Setup path from T1 to DSP

     t8100SwitchCfg.number_of_connections = 1;
     t8100SwitchCfg.connections = path;

    path[0].connect_src.resource= T8100_T1;
    path[0].connect_src.mode= T8100_CONNECT_CONST_DELAY;
    path[0].connect_src.ctbus_connect_num = 0;
    path[0].connect_src.port= T1_Port[lUnit];
    path[0].connect_src.timeslot= lChannel;

    path[0].connect_dest.resource= T8100_DSP;
    path[0].connect_src.mode= T8100_CONNECT_CONST_DELAY;
    path[0].connect_dest.ctbus_connect_num = 0;
    path[0].connect_dest.port= DSP_Port[lUnit];
    path[0].connect_dest.timeslot= lChannel;

    iopControlPeripheral ( TASK_T8100, CONFIG_T8100_SWITCHING, (t_configArg *)&t8100SwitchCfg);
```

Enabling bi-directional voice data flow to and from the DSP

```
if(upEnableChannel(lUnit// DSP Unit #
                       lChannel,// Channel #
                       TRUE,// Tx Enabled
                       TRUE) != UP_SUCCESS)// Rx Enabled
    {
        IopLogEvent(IOP_EVENT_LOG_TYPE_APP_HIGH, lUnit, lChannel, 0, "CrFastPktOut::EnableChannel-Bad");
    }
    else
    {
        IopLogEvent(IOP_EVENT_LOG_TYPE_APP_HIGH, lUnit, lChannel, 0, "CrFastPktOut::EnableChannel-Good");
    }
```

Configuring DSP services for outbound direction (toward the IP cloud)

```
/  Setup G711 Service Parameters
    utConfigSvc.tCodecConfig.eCodec     = ctG711;    // Set Codec to G711
    utConfigSvc.tCodecConfig.eCodecParams.tG711Param.eBfmEnable  = FALSE;   / / BFM OFF
    utConfigSvc.tCodecConfig.eCodecParams.tG711Param.eLaw        = enumMULAW; / / U-Law
    utConfigSvc.tCodecConfig.eCodecParams.tG711Param.eVadEnable  = FALSE;  / / VAD OFF
    utConfigSvc.tCodecConfig.eCodecParams.tG711Param.lVadLowSigThreshold = -50;

// Configure Service
    if(upConfigService(lUnit,                       // DSP Unit #
                       lChannel,                    // Channel #
                       stCodec,                     // CODEC Service Type Enumeration
                       &utConfigSvc) != UP_SUCCESS) // Codec Configuration Settings
    {
        IopLogEvent(IOP_EVENT_LOG_TYPE_APP_HIGH, lUnit, lChannel, stCodec, "CrFastPktOut::ConfigSvc-Bad");
        return(UP_FAILURE);
    }
    else
    {
        IopLogEvent(IOP_EVENT_LOG_TYPE_APP_HIGH, lUnit, lChannel, stCodec, "CrFastPktOut::ConfigSvc-Good");
    }

//  Setup Echo Cancellation Service
    utConfigSvc.tEchoCancConfig.eTapLength          = TL8ms;
    utConfigSvc.tEchoCancConfig.lFreezeAdaptation   = enumDisabled;
    utConfigSvc.tEchoCancConfig.lNLPDisable         = enumDisabled;
    utConfigSvc.tEchoCancConfig.lNlpThreshold       = 10;
    utConfigSvc.tEchoCancConfig.lSlowAdaptation     = enumDisabled;

// Configure Service
    if(upConfigService(lUnit,              // DSP Unit #
                       lChannel,           // Channel #
                         stEchoCanc,         // Echo Cancellation Service Type Enumeration
                       &utConfigSvc) != UP_SUCCESS) // Codec Configuration Settings
    {
        IopLogEvent(IOP_EVENT_LOG_TYPE_APP_HIGH, lUnit, lChannel, stEchoCanc, "CrFastPktOut::ConfigSvc-Bad");
        return(UP_FAILURE);
    }
    else
```

```
    {
        IopLogEvent(IOP_EVENT_LOG_TYPE_APP_HIGH, lUnit, lChannel, stEchoCanc, "CrFastPktOut::ConfigSvc-Good");
    }

//  Setup RTP Encoder Service
   utConfigSvc.tRtpSendConfig.stRtpSendHeader.version       = 2;
    utConfigSvc.tRtpSendConfig.stRtpSendHeader.p            = 0;
    utConfigSvc.tRtpSendConfig.stRtpSendHeader.x            = 0;
    utConfigSvc.tRtpSendConfig.stRtpSendHeader.cc           = 0;
    utConfigSvc.tRtpSendConfig.stRtpSendHeader.m            = 0;
    utConfigSvc.tRtpSendConfig.stRtpSendHeader.pt           = 0;
    utConfigSvc.tRtpSendConfig.stRtpSendHeader.seq          = 1;
    utConfigSvc.tRtpSendConfig.stRtpSendHeader.ts           = 0;
    utConfigSvc.tRtpSendConfig.stRtpSendHeader.ssrc         = 0x12345;
    utConfigSvc.tRtpSendConfig.ulPaddingLen                 = 0;
    utConfigSvc.tRtpSendConfig.ulPayloadInterval            = 10;
    utConfigSvc.tRtpSendConfig.ulTimeElapsedForEachFrame    = 80;
    utConfigSvc.tRtpSendConfig.ulInitConfig                 = 0;

//  Configure Service
    if(upConfigService(lUnit,                    // DSP Unit #
                       lChannel,                 // Channel #
                       stRtpEncode,                 // RTP Encoder Service Type Enumeration
                       &utConfigSvc) != UP_SUCCESS) // Codec Configuration Settings
    {
        IopLogEvent(IOP_EVENT_LOG_TYPE_APP_HIGH, lUnit, lChannel, stRtpEncode, "CrFastPktOut::ConfigSvc-Bad");
        return(UP_FAILURE);
    }
    else
    {
        IopLogEvent(IOP_EVENT_LOG_TYPE_APP_HIGH, lUnit, lChannel, stRtpEncode, "CrFastPktOut::ConfigSvc-Good");
    }
```

Creating data paths

Creating a data path between an outgoing RTP channel running on a DSP and the IOP driver's network packet routing component

```
   ulRouterAddr = gulSrcAddress ^ 1 ;    //toggle least significant bit
   IopLogEvent(IOP_EVENT_LOG_TYPE_APP_HIGH, ulRouterAddr, 0, 0, "ulRouterAddr");

   stPktSendConfig.ulInterface     = 0; // Interface 0 (fei0)
```

```
    stPktSendConfig.ulRouterAddress = ulRouterAddr;  // IP address of Router
    stPktSendConfig.ulDestAddress   = ulRouterAddr;  // IP address of Destination

    stPktSendConfig.ulDestPort      = 6000 + ((32*lUnit + lChannel)<<1);// RTP Port

//  Configure Packet Builder Service
    if(upConnectPktSend(lUnit,                              // DSP #
                        lChannel,                           // Channel #
                        &stPktSendConfig) != UP_SUCCESS)    // Fast Pkt Send Structure
    {
        IopLogEvent(IOP_EVENT_LOG_TYPE_APP_HIGH, lUnit, lChannel, 0, "CrFastPktOut::ConPktSend-Bad");
        return(UP_FAILURE);
    }
    else
    {
        IopLogEvent(IOP_EVENT_LOG_TYPE_APP_HIGH, lUnit, lChannel, 0, "CrFastPktOut::ConPktSend-Good");
    }
Transmit direction is initialized !
```

Enabling the receive direction on the channel

```
//  Setup path from DSP to T1

    t8100SwitchCfg.number_of_connections = 1;
    t8100SwitchCfg.connections = path;

   path[0].connect_src.resource   = T8100_DSP;
   path[0].connect_src.mode       = T8100_CONNECT_CONST_DELAY;
   path[0].connect_src.ctbus_connect_num = 0;
   path[0].connect_src.port       = DSP_Port[lUnit];
   path[0].connect_src.timeslot   = lChannel;

   path[0].connect_dest.resource = T8100_T1;
   path[0].connect_src.mode       = T8100_CONNECT_CONST_DELAY;
   path[0].connect_dest.ctbus_connect_num = 0;
   path[0].connect_dest.port      = T1_Port[lUnit];
   path[0].connect_dest.timeslot = lChannel;

   iopControlPeripheral ( TASK_T8100, CONFIG_T8100_SWITCHING, (t_configArg *)&t8100SwitchCfg);
```

Initializing the RTP decoder

```
//  Setup RTP Decoder Service
    utConfigSvc.tRtpRecvConfig.ulAutoAdjustable          = TRUE;
    utConfigSvc.tRtpRecvConfig.ulMaxJitterBufferDly      = 200;
    utConfigSvc.tRtpRecvConfig.ulTargetJitterBufferDly   = 30;
    utConfigSvc.tRtpRecvConfig.ulMaxFrameSizeInBytes     = 80;
    utConfigSvc.tRtpRecvConfig.ulExtractDataLength       = 80;
    utConfigSvc.tRtpRecvConfig.ulInitConfig              = 0;

// Configure Service
    if(upConfigService(lUnit,                           // DSP Unit #
                       lChannel,                        // Channel #
                       stRtpDecode,                     // RTP Decoder Service Type Enumeration
                       &utConfigSvc) != UP_SUCCESS)     // Codec Configuration Settings
    {
        IopLogEvent(IOP_EVENT_LOG_TYPE_APP_HIGH, lUnit, lChannel, stRtpDecode, "CrFastPktIn::ConfigSvc-Bad");
        return(UP_FAILURE);
    }
    else
    {
        IopLogEvent(IOP_EVENT_LOG_TYPE_APP_HIGH, lUnit, lChannel, stRtpDecode, "CrFastPktIn::ConfigSvc-Good");    }
```

Creating a data path between an inbound RTP channel running on a DSP and the IOP driver's network packet routing component

```
//  Setup the Fast Packet Receiver Service
    stPktRecvConfig.ulInterface     = 0;      // Ethernet Adapter Interface (fei0)
    stPktRecvConfig.ulReceivePort   = 6000 + ((32*lUnit + lChannel)<<1); // Port Assignment (6000 - 7000 port range)
    stPktRecvConfig.eService = stRtpDecode;

// Setup Packet Parser Service
    if(upConnectPktRecv(lUnit,                                   // DSP #
                        lChannel,                                // Channel #
                        &stPktRecvConfig) != UP_SUCCESS)         // Fast Pkt Receive Structure
    {
        IopLogEvent(IOP_EVENT_LOG_TYPE_APP_HIGH, lUnit, lChannel, 0, "CrFastPktIn::ConPktRecv-Bad");
        return(UP_FAILURE);
    }
    else
    {
        IopLogEvent(IOP_EVENT_LOG_TYPE_APP_HIGH, lUnit, lChannel, 0, "CrFastPktIn::ConPktRecv-Good");
    }
```

A Host functions

The Host API provides a control interface for the allocation, configuration, and execution of Host resources. The function prototypes and definitions for this API are contained in these header files:

- **Host control peripherals**: sp6khostapi.h (anything that begins with host)
- **UPA calls**: upa.h.

The host application is developed by linking with the Host API library. The library is installed as part of the TASK-6000 development kit and resides in this file:

InstallDir`\host\nt\lib\taskhost.lib`

This appendix lists function descriptions alphabetically within the following groups:

- **Standard functions**: Includes Control and Message APIs.
- **Advanced functions**: Includes functions you use only in unusual, special-purpose circumstances.

You can find a function in this appendix either by locating it alphabetically within its group, or by using Table A-1 on the next page, which groups like functions together.

Overview

Message API

The Message API from the host is substantially the same as from the IOP with the addition of slot numbers to the function arguments.

Function list

Use this table to identify the Host functions you want to use. Use the function description later in this appendix to obtain detailed information, including syntax and parameter values.

Table A-1. Host functions

Function	Description
Standard functions	
hostControlPeripheral	Configures and controls peripherals on SPIRIT boards.
hostExit	Inform TASK of user application exit.
hostGetBoardInfo	Obtains information on a specific board.
hostGetSystemInfo	Obtains information about all boards in the system.
hostInit	Initializes the driver.
hostLoadDsp	Loads 'C6x application into 'C6x memory.
hostLoadIop	Loads and runs an IOP application.
hostResetBoard	Resets the SPIRIT board.
hostResetDsp	Reset and initialize a DSP.
hostRunDsp	Runs 'C6x program.
hostRunLoadedIop	Run all loaded IOPs.
hostSetEventHandler	Register a handler for IOP/DSP events.
hostSetHotSwapHandler	Set user handler for Hot Swap events.
hostSetPeripheralDataHandler	User handler for peripheral data callback.
Control API	
upEnableChannel	Enables transmit and receive paths the specified channel.
upEnableService	Enables a specified service based on the service type for a specific channel, and also implicitly enables a service when it is configured.
upDisableService	Disables a specified service based on service type for a specific channel.
upConfigService	Configures a specified service based on service type for a specific channel.
upConfigServiceGlobal	Configures a specified service based on service type for all channels.
upSetEventHandler	Sends notification of DSP events.
upQueryQOSReport	Causes the specified DSP to send a Quality of Service report as an UP_EVT_STATISTICS_RPT event.
upStart	Initializes the Universal Port subsystem
Message API	The Message API from the host is substantially the same as from the IOP with the addition of slot numbers to the function arguments.

Table A-1. Host functions

Function	Description
upConnectPktSend	Creates a data path connection between an RTP or T.38 coder that runs in a channel on a DSP and the IOP's networking hardware.
upConnectPktRecv	Creates a data path from a UDP receive port to an RTP or T.38 decoder that runs on the specified DSP channel.
upDisconnectPktSend	Stops the forwarding of packets from a DSP channel to an IP socket and deallocates all associated IOP and DSP resources.
upDisconnectPktRecv	Stops the forwarding of packets from an IP socket to a DSP channel and de-allocates all associated IOP and DSP resources.
Advanced functions	
hostGetNWPktBuf	Receives network packets from the DSP.
hostJitterControl	Updates jitter control parameters.
hostReadIop	Read from IOP memory.
hostSendNWPktBuf	Sends network packets to the DSP.
hostSendMsg	Sends a message to IOP/DSP.
hostSetNWNotify	Set user handler for network packets notification.
hostWriteIop	Write to IOP memory.
hostSetPollPeriod	Set time period for message and network data polling.
hostGetNWPktBuf	Receives network packets from the DSP.
hostJitterControl	Updates jitter control parameters.
hostSendPriorityMsg	Send a priority message to IOP.
upSetUserMsgHandler	The MsgHandlerFunc is called whenever a TASK user message is received by the host with a msgType not used by UPA.

hostControlPeripheral

Configures the specified peripheral.

Call this function to initialize, configure, and operate the T1/E1 framer and TDM switch peripherals.

Syntax

```
int hostControlPeripheral (
    IN int peripheral,
    IN int cmd,
    IN t_configArg *pArg
)
```

Parameters

peripheral

Peripheral. You can select one of these values:

TASK_E1
TASK_T1
TASK_T8100

cmd

Command to send to the peripheral. You can use one of these values:

CONFIG_E1
: Instructs the framer to use E1 line protocol.

CONFIG_T1
: Instructs the framer to use T1 line protocol.

CONFIG_T8100_DEFAULT
: Reserved.

CONFIG_T8100_SWITCHING
: A pointer to the structure that specifies T8100 switching:

t_T8100SwitchConfig structure

```
typedef struct {
    ulong               number_of_connections;
    t_T8100Connection   *connections;
} t_T8100SwitchConfig;
```

number_of_connections

The number of T8100 connections to be made. The maximum connections per call is 256.

**connections*

Pointer to a list that contains *number_of_connections* connections.

CONFIG_T8100_CLOCKS

A pointer to the structure that specifies the clock the TDM switch uses:

t_T8100ClockConfig

```
typedef struct {
    t_ref_clk          reference_clk_select;
    t_netref_clk       netref_select;
    t_fallback_clk     allback_clk_select;
    BOOL32             netref_enable;
    BOOL32             netref2_enable;
    BOOL32             frame_clk_a_enable;
    BOOL32             frame_clk_b_enable;
    BOOL32             compat_clks_enable;
    BOOL32             fallback_enable;
} t_T8100ClockConfig;
```

reference_clk_select

Specifies the primary clock source to be used by the T8100.

netref_select

Specifies the clock source used by the T8100 to generate its output CT_NETREF signal.

fallback_clk_select

Specifies the fallback clock source used by the T8100. That is, the clock used when the primary clock source selected via *reference_clk_select* fails.

netref_enable

Enables the CT_NETREF signal, generated by the T8100, onto the CT Bus.

netref2_enable

Enables the CT_NETREF_2 signal, generated by the T8100, onto the CT Bus.

framer_clk_a_enable

Enables the CT_C8_A (clock) and CT_FRAME_A (frame sync) signals, generated by the T8100, onto the bus.

framer_clk_b_enable

Enables the CT_C8_B (clock) and CT_FRAME_B (frame sync) signals, generated by the T8100, onto the bus.

compat_clks_enable

Enables the MVIP and SCSA compatibilty clock and strobe signals, generated by the T8100, onto the bus.

fallback_enable
Enables clock fallback. When a clock error occurs on the current clock reference, the T8100 will fall back to *fallback_clk_select*.

CONFIG_T8100_STREAMS
A pointer to the structure that specifies the T8100 stream parameters:

t_T8100StreamConfig structure

```
typedef struct {
    t_stream_rate    dsp_bsp0_rate;
    t_stream_rate    dsp_bsp1_rate;
    t_stream_rate    e1t1_rate;
    t_stream_rate    ct_bus_03_00_rate;
    t_stream_rate    ct_bus_07_04_rate;
    t_stream_rate    ct_bus_11_08_rate;
    t_stream_rate    ct_bus_15_12_rate;
    t_stream_rate    ct_bus_19_16_rate;
    t_stream_rate    ct_bus_23_20_rate;
    t_stream_rate    ct_bus_27_24_rate;
    t_stream_rate    ct_bus_31_28_rate;
} t_T8100StreamConfig;
```

dsp_bsp0_rate
Specifies the TDM stream rate for Buffered Serial Port 0 of the DSPs. Currently this value must be the same as *dsp_bsp1_rate*.

dsp_bsp1_rate
Specifies the TDM stream rate for Buffered Serial Port 1 of the DSPs. This value must be the same as *dsp_bsp0_rate*.

e1t1_rate Specifies the TDM stream rate for the E1/T1 framers. Under most conditions, this rate should be 2MHz.

ct_bus_03_00_rate
Specifies the TDM stream rate for CT bus streams 0 through 3.

ct_bus_07_04_rate
Specifies the TDM stream rate for CT bus streams 4 through 7.

ct_bus_11_08_rate
Specifies the TDM stream rate for CT bus streams 8 through 11.

ct_bus_15_12_rate
Specifies the TDM stream rate for CT bus streams 12 through 15.

ct_bus_19_16_rate

Specifies the TDM stream rate for CT bus streams 16 through 19.

ct_bus_23_20_rate
Specifies the TDM stream rate for CT bus streams 20 through 23.

ct_bus_27_24_rate
Specifies the TDM stream rate for CT bus streams 24 through 27.

ct_bus_31_28_rate
Specifies the TDM stream rate for CT bus streams 28 through 31.

pArg Pointer to the union of configuration structures. This union contains structures containing peripheral control information:

t_configArg structure

```
typedef union {
    t_T1_user_config_struct    t1Config;
    t_E1_user_config_struct    e1Config;
    t_T8100ClockConfig         t8100ClkCfg;
    t_T8100SwitchConfig        t8100SwitchCfg;
    t_T8100StreamConfig        t8100StreamCfg;
} t_configArg;
```

t_T1_user_config_struct
Configuration structure for T1.

t_E1_user_config_struct
Configuration structure for E1.

t_T8100ClockConfig
Configuration structure for the T8100 clock.

t_T8100SwitchConfig
Configuration structure for the T8100 switch.

t_T8100StreamConfig
Configuration structure for the t8100 stream.

Return values

SUCCESS Successful completion.

FAILURE Unsuccessful completion.

hostExit

Sets all SPIRIT boards to quiesced state.

Syntax

```
STATUS hostExit (
   IN char unused
)
```

Parameters

unused	Reserved.

Return values

SUCCESS	Successful completion.
FAILURE	Unsuccessful completion.

Comments

Call this function just prior to host application termination, if desired, to reset the state of all boards in the system. That is, the IOP and DSPs are reset.

hostGetBoardInfo

Provides a board's information structure, as defined by the IOP monitor's SP6K_BOARD_INFO_T structure.

This function reports the extents of the systemby identifying the number and type of processors, T1/E1 framers, HDLC codecs, and TDM switches on the board. Also, you can use hostGetSystemInfo to determine the number and type of SPIRIT boards in the system.

For more information, see the sp6k_board_info_tsp6k.h file.

Syntax

```
int hostGetBoardInfo(
    IN long iopNum,
    IN OUT SP6K_BOARD_INFO_T *pBoardInfo
)
```

Parameters

iopNum The IOP number of the board for which you want information.

pBoardInfo
A pointer to SP6K_BOARD_INFO_T, the structure that stores board information:

SP6K_BOARD_INFO_T structure

```
typedef struct {
    UINT32          post_results;
    UINT32          free_zone;
    UINT32          flash_size;
    UINT32          memory_size;
    UCHAR           board_revision;
    UCHAR           num_dsps;
    UCHAR           hmic_present;
    UCHAR           board_type;
    UINT32          error_code;
    C6X_INFO_T      dsp_info[ MAX_DSPS ];
} SP6K_BOARD_INFO_T;
```

post_results
POST results.

free_zone Address of host free memory zone.

flash_size
The Flash ROM size.

memory_size
The memory size.

board_revision
The board revision number.

num_dsps The number of DSPs in the configuration.

hmic_present

HMIC/T8100 presence. The lower nibble identifies the installed T810x, and the upper nibble identifies the installed option card, if one exists.

Since all deliverable SPIRIT boards have a T810x (HMIC) installed, this field is not used for additional information. The lower nibble identifies the type of T810x installed. The upper nibble identifies the type of option card (if any) installed.

board_type

Specifies the board type. You can select one of these:

Single

Dual

Quad

error_code

Error code storage (for IOPDRV).

dsp_info[MAX_DSPS]

DSP POST results and memory sizes.

Outputs

pBoardInfo

A buffer that stores board information.

Return values

SUCCESS Successful completion.

FAILURE Unsuccessful completion.

Comments

For more information, see *hostGetSystemInfo on page 57*.

hostGetSystemInfo

Provides a board's information structure as defined by the IOP monitor's SP6K_BOARD_INFO_T structure. For more information, see the sp6k.h file.

Syntax

```
int hostGetSystemInfo(
    OUT long *pTotalIops,
    OUT long *pTotalDsps,
    OUT PREPORTBOARDINFO *pBoardInfo
)
```

Parameters

None.

Outputs

pTotalIops
A pointer to the variable that stores the number of boards (IOPs).

pTotalDsps
A pointer to the variable that stores the total number of DSPs in system.

pBoardInfo
A pointer to REPORTBOARDINFO, the buffer that stores board driver information:

REPORTBOARDINFO structure

```
typedef struct _ReportBoardInfo {
    ULONG BusNumber;
    ULONG DeviceNumber;
    ULONG FunctionNumber;
    ULONG IopMemorySize;
    ULONG PostResults;
    UCHAR BoardRevision;
    UCHAR NumberDSPs;
    UCHAR H100Present;
    UCHAR BoardType;
    ULONG DspMemorySize;
    ULONG DeviceState;
    ULONG reserved[3];
} REPORTBOARDINFO, *PREPORTBOARDINFO;
```

BusNumber
The board's physical PCI bus number.

DeviceNumber
The board's physical PCI device number.

FunctionNumber
The board's physical PCI function number.

IopMemorySize
The number of bytes in the ATU memory block.

`PostResults`
The POST results bitmask.

`BoardRevision`
The board's revision number.

`NumberDSPs`
The number of DSPs in the system.

`H100Present`
Indicates whether the H100 switch is populated.

`BoardType`
The board type: SP-6000 or 6040.

`DspMemorySize`
The size of the DSP memory window. This parameter is valid only when the value of `BoardType` is SP-6040.

`DeviceState`
Contains Hot Swap state information.

Return values

SUCCESS Successful completion.

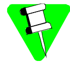
FAILURE Unsuccessful completion.

Comments

This function returns the information detected by the driver. The buffer should be capable of holding up to 8 REPORTBOARDINFO structures. You can use the pointer used in conjunction with an IOP number to index into a specific board's information, up to the given number of IOPs in the system. For more information, see *hostGetBoardInfo on page 55*.

hostInit

Initializes the Host API.



Call this function first, before calling any other functions, to initialize the module.

Syntax

```
int hostInit (void)
```

Parameters

None.

Return values

SUCCESS	Successful completion.
FAILURE	Unsuccessful completion.

hostLoadDsp

Loads a TASK-generated DSP program onto the specified DSP processor.

You can skip this step if an external program loads the cards before this application begins.

Syntax

```
int hostLoadDsp(
    IN long dspNum,
    IN char *sCoffFileName
)
```

Parameters

dspNum The number of the DSP onto which the program loads.

sCoffFileName
The DSP executable file, generated by the TASK composer.

Return values

SUCCESS Successful completion.

FAILURE Unsuccessful completion.

hostLoadIop

Loads the IOP with an application.

This function loads and executes the IOP application. Reloading the IOP resets the DSPs. DSPs must then be reloaded.

You can skip this step if an external program loads the IOP before this application begins.

Syntax

```
int hostLoadIop(
    IN UCHAR iopNum,
    IN char *app_name
)
```

Parameters

iopNum	The number of the IOP board onto which the program loads.
app_name	A pointer to a character string that specifies the IOP executable filename and path.

Return values

SUCCESS	Successful completion.
FAILURE	Unsuccessful completion.

hostResetBoard

Resets and initializes an IOP.

Syntax

```
int hostResetBoard
   IN long ulIopNum
)
```

Parameters

ulIopNum	The IOP number to reset and initialize.

Return values

SUCCESS	Successful completion.
FAILURE	Unsuccessful completion.

hostResetDsp

Resets and initializes a DSP.

Syntax

```
int hostResetDsp(
    IN long dspNum
)
```

Parameters

dspNum	The number of the DSP you want to reset and initialize.

Return values

SUCCESS	Successful completion.
FAILURE	Unsuccessful completion.

hostRunDsp

Runs the program downloaded through the hostLoadDsp function.

You can skip this step if an external program loads the IOP before this application begins.

Syntax

```
int hostRunDsp(
    IN long dspNum
)
```

Parameters

dspNum	The number of the DSP on which the program runs.

Return values

SUCCESS	Successful completion.
FAILURE	Unsuccessful completion.

hostRunLoadedIop

Initializes the message buffers which allow communication between the host and IOP, and sends handshake messages in both directions.

Syntax

```
int WINAPI hostRunLoadedIops(
    ulong ulIopNum
)
```

Parameters

ulIopNum	IOP number.

Return values

SUCCESS	The IOP is running.
FAILURE	Invalid IOP number, the IOP is not running, or internal API error occurred.

hostSetEventHandler

Specifies the user event handler for TASK events.

Syntax

```
int hostSetEventHandler (
   IN void (* eventHandler)(
   long src,
   long type,
   ulong count,
   long *pDataBuf
)
```

Parameters

eventHandler
: A pointer to the event handler.

src The message handler, in this format:

Device | *DeviceNum*

Device Enter either TASK_IOP or TASK_DSP.

DeviceNum Enter the device number.

Examples:

To invoke a handler from the second IOP, use (TASK_IOP | 2) as the destination.

To invoke a handler from the third DSP, use (TASK_DSP | 3) as destination.

type Message type. You can use any positive integer.

count The number of words in the message.

pDataBuf The buffer that contains the message.

Return values

SUCCESS Successful completion.

FAILURE Unsuccessful completion.

hostSetHotSwapHandler

Initializes the Hot Swap interface, and registers for notification of Hot Swap events with the Hot Swap service.

Syntax

```
STATUS WINAPI hostSetHotSwapHandler(
    IN void( *Handler) (
        IN long iopNum,
        IN long eventID
    )
)
```

Parameters

Handler A pointer to the message handler.

iopNum The number of the IOP board that registered for the Hot Swap Event.

eventID The event identification. You can enter one of these values:

HS_DEVICE_NORMAL
Indicates that the board is in the slot and activated.

HS_DEVICE_QUIESCED
Indicates that the board is removed from the slot and activity has ceased.

HS_DEVICE_POWERED_OFF
No board exists in the specified slot.

Return values

SUCCESS Successful completion.

FAILURE Unsuccessful completion.

Comments

Currently, only RadiSys SPIRIT-6040 boards support Hot Swap. For more information about these boards, see *SPIRIT™ boards* on page iv.

hostSetPeripheralDataHandler

Identifies the message handler to invoke for processing received messages.

Syntax

```
void hostSetPeripheralDataHandler(
   void(*peripheralDataHandler)
   (
      long src,
      long type,
      long *pDataBuf
   )
)
```

Parameters

peripheralDataHandler
A pointer to the message handler.

src The message's source, in this format:

Device | *DeviceNum*

Device Enter either TASK_IOP or TASK_DSP.

DeviceNum Enter the device number.

Examples:

To receive a message from the second IOP, use (TASK_IOP | 2) as the destination.

To receive a message from the third DSP, use (TASK_DSP | 3) as destination.

type Message type (currently only GET_T1_SIGNALING).

pDataBuf The buffer that contains the message.

Return values

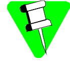
SUCCESS Successful completion.

FAILURE Unsuccessful completion.

upConfigService

Configures a specified service based on service type for a specific channel.

Also enables the service after configuration.



- Enabling the stCodec service disables the stTdmToneGen and stPktToneGen services.
- For descriptions of available services, see Appendix F, Service descriptions.

Syntax

```
UP_ERROR_ET upConfigService(
    RSYS_INT32 lSlot,
    RSYS_INT32 lUnit,
    RSYS_INT32 lChannel,
    UP_SERVICE_ET eService,
    UP_CONFIG_SVC_MSG_UT *puCfg
);
```

Parameters

lSlot The resource card unit number.

lUnit Specifies:

- For most services: the DSP unit number.
- For IOP-based services such as stCAS: the framer unit number.

lChannel Specifies:

- For most services: the virtual channel number.
- For IOP-based services such as stCAS: the timeslot number.

eService Identifies the service you want to configure:

stCodec Converts voice, fax, or modem data between TDM and a packetized form.

stEchoCanc Tries to remove time-delayed versions of a TDM channel's output from its input stream.

stAGC Provides Automatic Gain Control for TDM input data.

stTdmDTMFDet Performs DTMF detection on TDM input data.

stPktDTMFDet Performs DTMF detection on decompressed packet data before it is output to a TDM stream.

stCPTDet Performs Call Progress Tone detection on TDM input data.

stMFDet Performs MF Tone detection on TDM input data for R1 or R2 signaling.

stPktToneGen

Generates tones or tone-pairs to a TDM output with programmable cadence and cadence count.

stRtpEncode
: Performs RTP Packetization and, when used with the stJitterBuf service, depacketization.

stRtpDecode
: Implements both RTP decode functionality and a dynamic jitter buffer that feeds compressed frames to a codec for decompression on a strict schedule regardless of input variations.

stTdmToneGen
: Generates tones or tone-pairs to a TDM output with programmable cadence and cadence count.

stT1E1Alarm
: Notifies the user of an alarm signaled by a T1 or E1 framer, and allows control of the LEDs associated with each span.

stCAS
: Implements Channel Associated Signaling on T1 and E1 lines which are attached locally to framers control by the I/O Processor.

stPacketBuilder
: This is an embedded service controlled by the upConnectPktSend function.

stPacketParser
: Takes an IP/UDP packet from the Ethernet->DSP FIFO, checksums headers and data as appropriate, strips headers, and passes the payload to the next module, usually stRtpDecode or stT38, in TASK2.

stEthernetAlarm
: Detects Ethernet Link status changes and generates UP_EVT_ETHERNET_ALARM events, which notify a user application of the change.

stQDS0Hdlc
: Concatenates four sub-rate HDLC channels into a full DS0.

For detailed information about TASK services, see Appendix F, Service descriptions.

puCfg A pointer to the configuration data structure:

UP_CONFIG_SVC_UT structure

```
typedef union {
    UP_CODEC_CONFIG_ST          tCodecConfig;
    UP_ECHO_CONFIG_ST           tEchoCancConfig;
    UP_AGC_CONFIG_ST            tAGCConfig;
    UP_DTMF_CONFIG_ST           tDTMFConfig;
    UP_CPT_CONFIG_ST            tCPTConfig;
```

```
    UP_MF_CONFIG_ST                tMFConfig;
    UP_TONEGEN_CONFIG_ST           tToneGenConfig;
    UP_RTP_SEND_CONFIG_ST          tRtpSendConfig;
    UP_RTP_RECV_CONFIG_ST          tRtpRecvConfig;
    UP_T38_CONFIG_ST               tT38Config;
    UP_T1E1ALARM_CONFIG_ST         tT1E1AlarmConfig;
    UP_CAS_CONFIG_ST               tCasConfig;
    UP_PACKET_BUILDER_CONFIG_ST    tPBConfig;
    UP_PACKET_PARSER_CONFIG_ST     tPPConfig;
    UP_IOPINITCONFIG_ST            tIopInitData;
} UP_CONFIG_SVC_UT;
```

Return values

UP_SUCCESS

Successful completion.

If the function cannot return an error in real time (for example, the function sends a message to another processor to implement a configuration change), the function returns this value, indicating that the message was sent properly. The processor implementing the function returns one of these values:

UP_EVT_CONFIG_ACK

Occurs in response to this function if the service is configured.

UP_EVT_CONFIG_ERROR

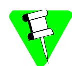
Occurs if the function fails, for instance if the specified service is not configured.

UP_FAILURE

General failure.

upConfigServiceGlobal

Configures a specified service based on service type for all channels.



For descriptions of available services, see Appendix F, Service descriptions.

Syntax

```
UP_ERROR_ET upConfigServiceGlobal(
    RSYS_INT32 lSlot,
    RSYS_INT32 lUnit,
    UP_SERVICE_ET eService,
    UP_GLOBALCONFIGDATA_UT *puCfg
)
```

Parameters

lSlot The resource card unit number.

lUnit Specifies:

- For most services: the DSP unit number.
- For IOP-based services such as stCAS: the framer unit number.

eService Identifies the service you want to configure:

stCodec Converts voice, fax, or modem data between TDM and a packetized form.

stEchoCanc
Tries to remove time-delayed versions of a TDM channel's output from its input stream.

stAGC Provides Automatic Gain Control for TDM input data.

stTdmDTMFDet
Performs DTMF detection on TDM input data.

stPktDTMFDet
Performs DTMF detection on decompressed packet data before it is output to a TDM stream.

stCPTDet Performs Call Progress Tone detection on TDM input data.

stMFDet Performs MF Tone detection on TDM input data for R1 or R2 signaling.

stPktToneGen
Generates tones or tone-pairs to a TDM output with programmable cadence and cadence count.

stRtpEncode
Performs RTP Packetization and, when used with the stJitterBuf service, depacketization.

stRtpDecode
Implements both RTP decode functionality and a dynamic

jitter buffer that feeds compressed frames to a codec for decompression on a strict schedule regardless of input variations.

stTdmToneGen
: Generates tones or tone-pairs to a TDM output with programmable cadence and cadence count.

stT1E1Alarm
: Notifies the user of an alarm signaled by a T1 or E1 framer, and allows control of the LEDs associated with each span.

stCAS
: Implements Channel Associated Signaling on T1 and E1 lines which are attached locally to framers control by the I/O Processor.

stPacketBuilder
: This is an embedded service controlled by the upConnectPktSend function.

stPacketParser
: Takes an IP/UDP packet from the Ethernet->DSP FIFO, checksums headers and data as appropriate, strips headers, and passes the payload to the next module, usually stRtpDecode or stT38, in TASK2.

stEthernetAlarm
: Detects Ethernet Link status changes and generates UP_EVT_ETHERNET_ALARM events, which notify a user application of the change.

stQDS0Hdlc
: Concatenates four sub-rate HDLC channels into a full DS0.

For detailed information about TASK services, see Appendix F, Service descriptions.

puCfg A pointer to the configuration data structure:

UP_GLOBALCONFIGDATA_UT structure

```
typedef union {
    UP_DSPINITCONFIG_ST tInitData;
} UP_GLOBALCONFIGDATA_UT;
```

This structure is currently only used by UPA initialization code.

Return values

UP_SUCCESS
: Successful completion.

If the function cannot return an error in real time (for example, the function sends a message to another processor to implement a configuration change), the function returns this value, indicating that the message was sent properly. The processor implementing the function returns one of these values:

UP_EVT_CONFIG_ACK
Occurs in response to this function if the service is configured.

UP_EVT_CONFIG_ERROR
Occurs if the function fails, for instance if the specified service is not configured.

UP_FAILURE
General failure.

upConnectPktRecv

Creates a data path from a UDP receive port to an RTP or T.38 decoder that runs on the specified DSP channel.

As each packet is received it is DMA'd to a FIFO in the DSP specified by *lUnit*. During the DSP's inter-frame time the packets in the FIFO are sorted by channel and their payloads passed either to the RTP Receive/Jitter Buffer service, or to the appropriate T.38 FIFO.

Syntax

```
UP_ERROR_ET upConnectPktRecv(
    RSYS_INT32 lSlot,
    RSYS_INT32 lUnit,
    RSYS_INT32 lChannel,
    UP_PKT_RECV_CONFIG_ST *pstPktRecvConfig
);
```

Parameters

lSlot The resource card unit number.

lUnit The DSP unit number.

lChannel The DSP virtual channel number.

pstPktRecvConfig
A pointer to the UP_PKT_RECV_CONFIG_ST structure:

UP_PKT_RECV_CONFIG_ST structure

```
typedef struct {
    RSYS_UINT32   ulReceivePort;
    RSYS_UINT32   ulInterface;
} UP_PKT_RECV_CONFIG_ST;
```

ulReceivePort
The 16-bit port number of the socket which receives the packets. You can use one of these values:

0 The port number is automatically assigned. An appropriate port number is chosen from the range of UP_PREALLOC_PORT_START through (UP_PREALLOC_PORT_START + UP_PREALLOC_RTP_PORTS) (defined as 5000–5999 in upa.h) and returned via an event.

Other You specify the port number. The number must fall in the range of (UP_PREALLOC_PORT_START + UP_PREALLOC_RTP_PORTS) through (UP_PREALLOC_PORT_START + UP_MAX_RTP_PORTS) (defined as 6000–6999 in upa.h), and must be an even number.

ulInterface

The unit number of the Ethernet interface on the IOP on which packets are received.

Return values

UP_SUCCESS

Successful completion.

If the function cannot return an error in real time (for example, the function sends a message to another processor to implement a configuration change), the function returns this value, indicating that the message was sent properly. The processor implementing the function returns one of these values:

UP_EVT_CONFIG_ACK

Occurs in response to this function if the connection is established.

UP_EVT_CONFIG_ERROR

Emitted when the port number could not be allocated (for example, the port is already assigned or the DSP channel is already in use).

UP_EVT_PORT_ASSIGNED

Emitted upon success to identify the UDP port number allocated to the connection.

UP_FAILURE

General failure.

upConnectPktSend

Creates a data path connection between an RTP or T.38 coder that runs in a channel on a DSP and the IOP's networking hardware.

After each packet is encoded, it is passed to the stPacketBuilder service which adds UDP, IP, and Ethernet headers to the packet, and then adds it to the DSP's Ethernet Transmit FIFO. On expiration of a timer tick, the packets in the FIFO are added to the transmit queue of the appropriate Ethernet controller.

Syntax

```
UP_ERROR_ET upConnectPktSend(
    RSYS_INT32 lSlot,
    RSYS_INT32 lUnit,
    RSYS_INT32 lChannel,
    UP_PKT_SEND_CONFIG_ST *pstPktSendConfig
);
```

Parameters

lSlot The resource card unit number.

lUnit The DSP unit number.

lChannel The DSP virtual channel number.

pstPktSendConfig
A pointer to the UP_PKT_SEND_CONFIG_ST structure:

UP_PKT_SEND_CONFIG_ST structure

```
typedef struct {
    RSYS_UINT32   ulInterface;
    RSYS_UINT32   ulRouterAddress;
    RSYS_UINT32   ulDestAddress;
    RSYS_UINT32   ulDestPort;
    RSYS_UINT32   ulServiceType;
    RSYS_UINT32   ulSrcPort;
} UP_PKT_SEND_CONFIG_ST;
```

ulInterface
The unit number of the Ethernet interface on which packets should be sent.

ulRouterAddress
The 32-bit IP address, in host order, of a router on the same network as the IOP which should be used to forward packets to *ulDestAddress*. If *ulDestAddress* is on the same network as the IOP and no router should be used, then *ulDestRouter* should be set to *ulDestAddress*.

ulDestAddress
The 32-bit IP address, in host order, of the RTP or T.38 packets' destination.

`ulDestPort`
The 16-bit UDP port number of the RTP or T.38 packets' destination.

`ulServiceType`
An 8-bit value used to specify the precedence, delay, throughput, and reliability of a message. This is used in making quality of service decisions in the delivery of a packet.

`ulSrcPort` The optional 16-bit UDP port number that identifies the RTP or T.38 source application peer. If the source port is not provided, the values of `ulUnit` and `lChannel` passed with the function are used to find the corresponding port ID registered to the inbound fast packet channel established through upConnectPktRecv. This approach assumes that upConnectPktSend is preceded by upConnectPktRecv, and that each path makes up one part of full-duplex connection on a specified DSP and channel. A source port ID of 0 is assigned if the user does not specify a source port in their call to upConnectPktSend and if an upConnectPktRecv was not successfully executed prior to calling upConnectPktSend on the specified DSP and Channel numbers.

Return values

UP_SUCCESS
Successful completion.

If the function cannot return an error in real time (for example, the function sends a message to another processor to implement a configuration change), the function returns this value, indicating that the message was sent properly. The processor implementing the function returns one of these values:

UP_EVT_CONFIG_ACK
Occurs in response to this function if the connection is established.

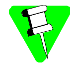
UP_EVT_CONFIG_ERROR
Occurs if the function fails, for instance if an ARP of the ulDestRouter (destination IP address) fails.

UP_FAILURE
General failure.

upDisableService

Disables a specified service based on service type for a specific channel.

Other services not dependent on this service are not affected. Major services (codecs, echo canceller), when disabled this way, consume no DSP resources. Minor services (tone detectors and generators, and so on) still load but do not run, reducing their resource load to a minimum, but not zero.



- Enabling the stCodec service disables the stTdmToneGen and stPktToneGen services.
- For descriptions of available services, see Appendix F, Service descriptions.

Syntax

```
UP_ERROR_ET upDisableService(
   RSYS_INT32 lSlot,
   RSYS_INT32 lUnit,
   RSYS_INT32 lChannel,
   UP_SERVICE_ET eService
);
```

Parameters

lSlot The resource card unit number.

lUnit Specifies:

- For most services: the DSP unit number.
- For IOP-based services such as stCAS: the framer unit number.

lChannel Specifies:

- For most services: the virtual channel number.
- For IOP-based services such as stCAS: the timeslot number.

eService Identifies the service you want to disable:

stCodec Converts voice, fax, or modem data between TDM and a packetized form.

stEchoCanc Tries to remove time-delayed versions of a TDM channel's output from its input stream.

stAGC Provides Automatic Gain Control for TDM input data.

stTdmDTMFDet Performs DTMF detection on TDM input data.

stPktDTMFDet Performs DTMF detection on decompressed packet data before it is output to a TDM stream.

stCPTDet Performs Call Progress Tone detection on TDM input data.

stMFDet Performs MF Tone detection on TDM input data for R1 or

R2 signaling.

stPktToneGen
: Generates tones or tone-pairs to a TDM output with programmable cadence and cadence count.

stRtpEncode
: Performs RTP Packetization and, when used with the stJitterBuf service, depacketization.

stRtpDecode
: Implements both RTP decode functionality and a dynamic jitter buffer that feeds compressed frames to a codec for decompression on a strict schedule regardless of input variations.

stTdmToneGen
: Generates tones or tone-pairs to a TDM output with programmable cadence and cadence count.

stT1E1Alarm
: Notifies the user of an alarm signaled by a T1 or E1 framer, and allows control of the LEDs associated with each span.

stCAS
: Implements Channel Associated Signaling on T1 and E1 lines which are attached locally to framers control by the I/O Processor.

stPacketBuilder
: This is an embedded service controlled by the upConnectPktSend function.

stPacketParser
: Takes an IP/UDP packet from the Ethernet->DSP FIFO, checksums headers and data as appropriate, strips headers, and passes the payload to the next module, usually stRtpDecode or stT38, in TASK2.

stEthernetAlarm
: Detects Ethernet Link status changes and generates UP_EVT_ETHERNET_ALARM events, which notify a user application of the change.

stQDS0Hdlc
: Concatenates four sub-rate HDLC channels into a full DS0.

For detailed information about TASK services, see Appendix F, Service descriptions.

Return values

UP_SUCCESS
: Successful completion.

If the function cannot return an error in real time (for example, the function sends a message to another processor to implement a configuration change), the function returns this value, indicating that the message was sent properly. The processor implementing the function returns one of these values:

UP_EVT_CONFIG_ACK
Occurs in response to this function if the service is disabled.

UP_EVT_CONFIG_ERROR
Occurs if the function fails, for instance if the specified service remains enabled.

UP_FAILURE
General failure.

upDisconnectPktRecv

Stops the forwarding of packets from an IP socket to a DSP channel and de-allocates all associated IOP and DSP resources.

Syntax

```
UP_ERROR_ET upDisconnectPktRecv(
   RSYS_INT32 lSlot,
   RSYS_INT32 lUnit,
   RSYS_INT32 lChannel
);
```

Parameters

lSlot	The resource card unit number.
lUnit	The DSP unit number.
lChannel	The DSP virtual channel number.

Return values

UP_SUCCESS

Successful completion.

If the function cannot return an error in real time (for example, the function sends a message to another processor to implement a configuration change), the function returns this value, indicating that the message was sent properly. The processor implementing the function returns one of these values:

UP_EVT_CONFIG_ERROR

UP_FAILURE

General failure.

upDisconnectPktSend

Stops the forwarding of packets from a DSP channel to an IP socket and deallocates all associated IOP and DSP resources.

Syntax

```
UP_ERROR_ET upDisconnectPktSend(
   RSYS_INT32 lSlot,
   RSYS_INT32 lUnit,
   RSYS_INT32 lChannel
);
```

Parameters

lSlot	The resource card unit number.
lUnit	The DSP unit number.
lChannel	The DSP virtual channel number.

Return values

UP_SUCCESS

Successful completion.

If the function cannot return an error in real time (for example, the function sends a message to another processor to implement a configuration change), the function returns this value, indicating that the message was sent properly. The processor implementing the function returns one of these values:

UP_EVT_CONFIG_ERROR

UP_FAILURE

General failure.

upEnableChannel

Enables transmit and receive paths the specified channel.

Syntax

```
UP_ERROR_ET upEnableChannel(
    RSYS_INT32 lSlot,
    RSYS_INT32 lUnit,
    RSYS_INT32 lChannel,
    RSYS_INT32 lTxEnable,
    RSYS_INT32 lRxEnable
);
```

Parameters

lSlot The resource card unit number.

lUnit The DSP unit number.

lChannel The DSP virtual channel number.

lTxEnable Controls a channel's transmit path processing.

- TRUE Enables transmit path processing.
- FALSE Disables transmit path processing.

lRxEnable Controls a channel's receive path processing.

- TRUE Enables receive path processing.
- FALSE Disables receive path processing.

Return values

UP_SUCCESS

Successful completion.

If the function cannot return an error in real time (for example, the function sends a message to another processor to implement a configuration change), the function returns this value, indicating that the message was sent properly. The processor implementing the function returns one of these values:

UP_EVT_CONFIG_ACK

Occurs in response to this function if the channel is enabled.

UP_EVT_CONFIG_ERROR

Occurs if the function fails, for instance if the specified channel is not enabled.

UP_FAILURE

General failure.

UP_INVALID_ARG

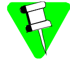
Invalid argument.

Comments

`lTxEnable` and `lRxEnable` may be used to separately control the transmit and receive path processing for a channel, e.g., a channel which generates tones may not need receive path processing. However, while this way the DSP processing time associated with transmit or receive algorithms, it does not eliminate the overhead of loading the algorithm into DSP memory. Only when `lTxEnable` and `lRxEnable` are both false is the channel entirely disabled, which removes its overhead entirely.

upEnableService

Enables a specified service based on the service type for a specific channel, and also implicitly enables a service when it is configured.



For descriptions of available services, see Appendix F, Service descriptions.

Syntax

```
UP_ERROR_ET upEnableService(
    RSYS_INT32 lSlot,
    RSYS_INT32 lUnit,
    RSYS_INT32 lChannel,
    UP_SERVICE_ET eService
);
```

Parameters

lSlot The resource card unit number.

lUnit Specifies:

- For most services: the DSP unit number.
- For IOP-based services such as stCAS or stT1E1Alarm: the framer unit number.

lChannel Specifies:

- For most services: the virtual channel number.
- For IOP-based services such as stCAS: the timeslot number.

eService Identifies the service you want to enable:

stCodec Converts voice, fax, or modem data between TDM and a packetized form.

stEchoCanc
Tries to remove time-delayed versions of a TDM channel's output from its input stream.

stAGC Provides Automatic Gain Control for TDM input data.

stTdmDTMFDet
Performs DTMF detection on TDM input data.

stPktDTMFDet
Performs DTMF detection on decompressed packet data before it is output to a TDM stream.

stCPTDet Performs Call Progress Tone detection on TDM input data.

stMFDet Performs MF Tone detection on TDM input data for R1 or R2 signaling.

stPktToneGen
Generates tones or tone-pairs to a TDM output with programmable cadence and cadence count.

stRtpEncode
: Performs RTP Packetization and, when used with the stJitterBuf service, depacketization.

stRtpDecode
: Implements both RTP decode functionality and a dynamic jitter buffer that feeds compressed frames to a codec for decompression on a strict schedule regardless of input variations.

stTdmToneGen
: Generates tones or tone-pairs to a TDM output with programmable cadence and cadence count.

stT1E1Alarm
: Notifies the user of an alarm signaled by a T1 or E1 framer, and allows control of the LEDs associated with each span.

stCAS
: Implements Channel Associated Signaling on T1 and E1 lines which are attached locally to framers control by the I/O Processor.

stPacketBuilder
: This is an embedded service controlled by the upConnectPktSend function.

stPacketParser
: Takes an IP/UDP packet from the Ethernet->DSP FIFO, checksums headers and data as appropriate, strips headers, and passes the payload to the next module, usually stRtpDecode or stT38, in TASK2.

stEthernetAlarm
: Detects Ethernet Link status changes and generates UP_EVT_ETHERNET_ALARM events, which notify a user application of the change.

stQDS0Hdlc
: Concatenates four sub-rate HDLC channels into a full DS0.

For detailed information about TASK services, see Appendix F, Service descriptions.

Return values

UP_SUCCESS
: Successful completion.

If the function cannot return an error in real time (for example, the function sends a message to another processor to implement a configuration change), the function returns this value, indicating that the message was sent properly. The processor implementing the function returns one of these values:

UP_EVT_CONFIG_ACK

Occurs in response to this function if the service is enabled.

UP_EVT_CONFIG_ERROR
Occurs if the function fails, for instance if the specified service is not enabled.

UP_FAILURE
General failure.

upQueryQOSReport

Causes the specified DSP to send a Quality of Service report as an UP_EVT_STATISTICS_RPT event.

An RTCP stack may use the provided information to create Sender and Receiver Reports.

Syntax

```
UP_ERROR_ET upQueryQOSReport(
   RSYS_INT32 lSlot,
   RSYS_INT32 lUnit,
   RSYS_INT32 lChannel
);
```

Parameters

lSlot The resource card unit number.

lUnit The DSP number.

lChannel The DSP virtual channel number.

Return values

UP_SUCCESS
: Successful completion. The function requests this event:

  UP_EVT_STATISTICS_RPT
  : Contains the statistics report requested by this function. The statistics are contained in the UP_STATISTICS_ST structure.

UP_FAILURE
: General failure.

UP_INVALID_ARG
: Invalid argument.

upSetEventHandler

Sends notification of DSP events.

Events can include tone detection, peripheral state changes such as hook and other CAS signals, and any other events any service generates, to EventHandlerFunc(UP_EVENT_DATA_ST*).

Syntax

```
UP_ERROR_ET upSetEventHandler(
    void (*eventHandlerFunc)(
        UP_EVENT_DATA_ST*
    )
);
```

Parameters

UP_EVENT_DATA_ST
: Receives notification of DSP events such as tone detection, peripheral state changes such as hook and other CAS signals, and any other events any service generates.

Return values

UP_SUCCESS
: The handler was installed.

upSetUserMsgHandler

The MsgHandlerFunc is called whenever a TASK user message is received by the host with a *msgType* not used by UPA.

Currently, UPA uses only message type 80 (decimal).

Syntax

```
void upSetUserMsgHandler (
    void (*MsgHandlerFunc)(
        IN long src,
        IN long msgType,
        IN ulong msgSzW,
        IN long *pBuf
    )
);
```

Parameters

MsgHandlerFunc
A pointer to the message handler.

src The message handler, in this format:

Device | *DeviceNum*

Device Enter either TASK_IOP or TASK_DSP.

DeviceNum Enter the device number.

Examples:

To invoke a handler from the second IOP, use (TASK_IOP | 2) as the destination.

To invoke a handler from the third DSP, use (TASK_DSP | 3) as destination.

msgType The message type. You can use any positive integer.

msgSzW The number of words in the message.

pBuf The buffer that contains the message.

Return values

None.

upStart

Initializes the Universal Port subsystem

To perform necessary system initialization, call this function once before calling any other UPA function on the Host. A controlling Host application populates the UP_IOPSYSCONFIG_ST structure when making its initial call to upStart.

To reconfigure network adapters or change VOIP control characteristics, call upStart again during runtime.

upStart also signals a change of control from a primary to a secondary Host Controller. This is a natural fit since the Host Controller's IP address is passed as part of the information that travels from Host Controller to the IOP. Under this model, a new controlling Host application simply has to call upStart to each IOP to inform it of the new IP address to use for control.

Syntax

```
UP_ERROR_ET upStart(
    UP_IOPSYSCONFIG_ST *ptIOPSysConfig
)
```

Parameters

ptIOPSysConfig

A pointer to a structure that configures the board during UPA initialization. This structure has the following format:

UP_IOPSYSCONFIG_ST structure

```
typedef struct {
    RSYS_UINT32          aulNICAddress[UP_MAX_SLOTS][MAX_NICS];
    RSYS_UINT32          ulHostIPAddress;
    UP_ENABLE_ET         etCommandAck;
    UP_ENABLE_ET         etEventForwarding;
    UP_LAN_CONTROL_ET    etLanControl;
} UP_IOPSYSCONFIG_ST;
```

aulNICaddress

An array element that carries the IP addresses (host byte ordered) to associate with Ethernet adapters of the installed IOPS. The IOP number serves as an index to de-reference the two installed Ethernet adapters (fei 0 and 1). You can enter one of these:

0 for either interface 0 or 1
: Indicates that you do not want UPA software to configure the Ethernet adapter and bind an IP address to it.

Other
: Causes UPA on the associated IOP to perform a vxWork's usrNetInit to initialize the adapter and a hostAdd to make a representative hostname to IP address mapping in the host table.

ulHostIPAddress

Identifies the IP Address assigned to each NIC. This informs an IOP of its controlling host's IP address. An IOP uses this IP address to establish a communication link to the host for event forwarding when either TCP or UDP based control is configured.

ulHostIPAddress

The IP Address of the controlling host.

etCommandAck

A flag that specifies whether UPA acknowledges each UPA command request with an asynchronous notification. You can select one of these:

`enumDisabled`

Commands are issued by an IOP based application.

`enumEnabled`

A host based application requires positive confirmation of command processing.

etEventForwarding

A flag that specifies whether the system forwards all events from each IOP to the host. You can select one of these:

`enumDisabled`

Events are forwarded from each IOP to the host.

`enumEnabled`

Events are not forwarded to the host. If you select this option, then a user application must be present on each IOP to receive most events.

etLanControl

A flag that specifies the transport to use for communicating control messages between a controlling host and an IOP. You can select one of these:

`enumNoLanControl`

The PCI bus communicates control messages between host and IOP. This limits host control to IOPs located within the same shelf.

Return values

UP_SUCCESS

Successful completion.

UP_FAILURE

General failure.

hostGetNWPktBuf

Receives the next buffer of packets from the DSP.

Syntax

```
int hostGetNWPktBuf (
    IN long dspNum,
    IN long *pBuf
)
```

Parameters

dspNum	The DSP number.
pBuf	A pointer to the buffer that receives network packets.

Return values

SUCCESS	Successful completion.
FAILURE	Unsuccessful completion.

Comments

The buffer contains one or more packets in the following format:

- Total buffer length
- Channel number (one word)
- Packet length (one word)
- One or more words of packet data

For details about the packet buffer contents, see *hostSendNWPktBuf* on page 97.

hostJitterControl

Updates jitter control parameters.

Syntax

```
int hostJitterControl (
    IN long dspNum,
    IN long ChnlNum,
    IN t_jitterParam *pJitter
)
```

Parameters

dspNum	The DSP number.
ChnlNum	The channel number.
pJitter	A pointer to a structure contains jitter parameters:

t_jitterParam structure

```
typedef struct {
    long nJitterBuf;
} t_jitterParam;
```

nJitterBuf
The number of jitter buffers between the Host and IOP/DSP.

Return values

SUCCESS	Successful completion.
FAILURE	Unsuccessful completion.

hostReadIop

Reads memory space from an IOP.

Syntax

```
int hostReadIop(
   UCHAR boardNum,
   long *iopSrcAddr,
   long *hstDstAddr,
   ULONG uCountW
```

Parameters

iopNum The number of the IOP you want to read memory from.

iopSrcAddr
The IOP external memory space address to read from.

hstDstAddr
The Host buffer address that stores read memory.

uCountW The number of words to read from the DSP.

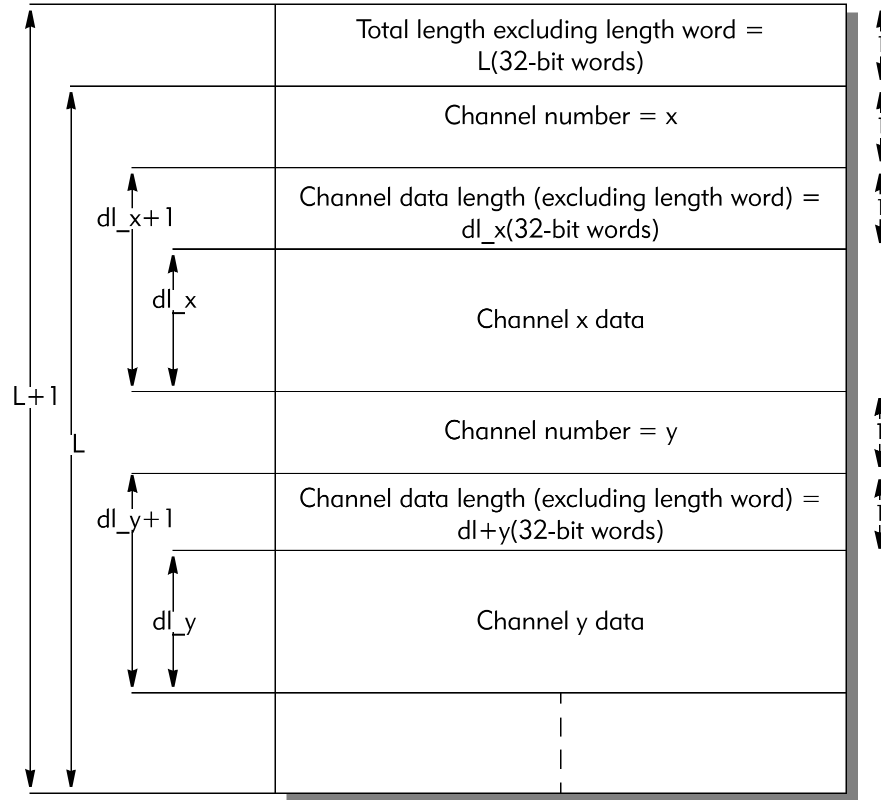
Return values

SUCCESS Successful completion.

FAILURE Unsuccessful completion.

hostSendNWPktBuf

Sends network packets to the DSP. The packets are organized in the buffer as follows:



All numbers and sizes are in 32-bit words

Figure A-1. Packet organization buffer

Syntax

```
int hostSendNWPktBuf (
    IN long dspNum,
    IN long *pBuf
)
```

Parameters

dspNum The number of the DSP to which you want to send packets.

pBuf A buffer that consists of one or more network packets, as shown in Figure A-1.

Return values

SUCCESS Successful completion.

FAILURE Unsuccessful completion.

hostSendMsg

Sends a message from the Host to the specified IOP or DSP. If the message is not delivered to the IOP within the specified time, the function returns failure.

Syntax

```
int hostSendMsg (
    IN long dst,
    IN long msgType,
    IN ulong msgSzW,
    IN long *pBuf,
    IN long waitTimeMsec
)
```

Parameters

dst The message's destination, in this format:

Device | *DeviceNum*

Device Enter either TASK_IOP or TASK_DSP.

DeviceNum Enter the device number.

Examples:

To send a message to the second IOP, use (TASK_IOP | 2) as the destination.

To send a message to the third DSP, use (TASK_DSP | 3) as destination.

msgType Message type. You can use any positive integer value.

msgSzW The number of words in a message.

pBuf The buffer that contains the message.

waitTimeMsec
The number of milliseconds that elapse before determining a message's delivered status.

Return values

SUCCESS Successful completion.

FAILURE Unsuccessful completion.

hostSetNWNotify

Identifies the handler to invoke to notify the arrival of network packets from the DSP. The handler must take the DSP number as the argument.

Syntax

```
void hostSetNWNotify(
    IN void (* NWNotifyHndlr) (IN long dspNum)
)
```

Parameters

NWNotifyHndlr
A pointer to the handler that notifies the Host application of the arrival of network packets from the DSP.

dspNum The number of the DSP from which network packets were sent.

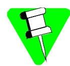
Return values

SUCCESS Successful completion.

FAILURE Unsuccessful completion.

hostWriteIop

Writes to the IOP memory space.



Syntax

```
int hostWriteIop(
    UCHAR boardNum,
    long *hstSrcAddr,
    long *iopDstAddr,
    ULONG uCountW
)
```

Parameters

iopNum The number of the IOP board to write memory from.

hstSrcAddr
A pointer to host buffer you want to write to.

iopDstAddr
The IOP destination address.

uCountW The number of words to write to the IOP.

Return values

SUCCESS Successful completion.

FAILURE Unsuccessful completion.

hostSetPollPeriod

Specifies the number of milliseconds that elapse before the host polls all IOPs for received network packets.

Syntax

```
int HostSetPollPeriod(
    ulong ulPollPeriod
)
```

Parameters

ulPollPeriod

The number of milliseconds that elapse before the host polls all IOPs for received network packets. The default is 10 ms.

Return values

SUCCESS Successful completion.

FAILURE Unsuccessful completion.

hostSendPriorityMsg

Sends a message from the Host to the specified IOP or DSP. If the message is not delivered to the IOP within the specified time, the function returns failure.

The message advances to the first position in the queue of messages destined for each IOP, and therefore pre-empts any other messages waiting in the queue.

Syntax

```
int hostSendPriorityMsg (
    IN long dst,
    IN long msgType,
    IN ulong msgSzW,
    IN long *pBuf,
    IN long waitTimeMsec
)
```

Parameters

dst The message's destination, in this format:

Device | *DeviceNum*

Device Enter either TASK_IOP or TASK_DSP.

DeviceNum Enter the device number.

Examples:

To send a message to the second IOP, use (TASK_IOP | 2) as the destination.

To send a message to the third DSP, use (TASK_DSP | 3) as destination.

msgType Message type. You can use any positive integer value.

msgSzW The number of words in a message.

pBuf The buffer that contains the message.

waitTimeMsec
The number of milliseconds that elapse before determining a message's delivered status.

Return values

SUCCESS Successful completion.

FAILURE Unsuccessful completion.

B IOP functions

The IOP API provides a control interface for the allocation, configuration, and execution of IOP resources. The function prototypes and definitions for this API are contained in the header filexyz.h.

You develop IOP applications by linking with the IOP API library.

This appendix lists function descriptions alphabetically within the following groups:

- **Standard functions**: Includes Control and Message APIs.
- **Advanced functions**: Includes functions you use only in unusual, special-purpose circumstances.

You can find a function in this appendix either by locating it alphabetically within its group, or by using Table B-1 on the next page, which groups like functions together.

Overview

Message API

You use these functions to configure UPA to transparently forward packets between an Ethernet interface and an RTP or T.38 transport service that runs on a RadiSys SPIRIT board's DSP.

The implementation of this functionality is very high performance, capable of moving many thousands of packets per second while consuming only a fraction of the i960 I/O processor's time. In brief, this is accomplished by bypassing the VxWorks TCP/IP stack and processing the majority of the Ethernet, IP, and UDP layers on the DSP.

The caveat is that no full IP stack proceses these packets, so "unusual" cases are not supported. For instance:

- Fragmentation/reassembly is not supported. Fragmented packets are silently discarded.
- Buffer sizes are based on typical RTP voice packets. Packets larger than a predefined maximum (initially 350 bytes) are silently discarded.
- VxWorks routing tables are not used; only a simple default route is currently supported.

Function list

Use this table to identify the IOP functions you want to use. Use the function description later in this appendix to obtain detailed information, including syntax and parameter values.

Table B-1. IOP functions

Function	Description
Standard functions	
iopControlPeripheral	Configures specified peripheral.
getBoardInfo	Provides a board's information structure, as defined by the IOP monitor's SP6K_BOARD_INFO_T structure.
iopInit	Initializes the IOP API.
Control API	
upStart	Initializes the Universal Port subsystem.
upEnableChannel	Enables transmit and receive paths of a specified channel.
upEnableService	Enables a specified service based on service type for the specified channel. upConfigService also implicitly enables a service when it is configured.
upDisableService	Disables a specified service based on service type for a specific channel.
upConfigService	Configures a specified service based on service type for a specific channel.
upConfigServiceGlobal	Configures a specified service based on service type for all channels.
upSetEventHandler	Passes information received from DSPs to the event handler.
upQueryQOSReport	Causes the specified DSP specified to send a Quality of Service report as an UP_EVT_STATISTICS_RPT event.
Message API	You use these functions to configure UPA to transparently forward packets between an Ethernet interface and an RTP or T.38 transport service that runs on a RadiSys SPIRIT board's DSP.
upConnectPktSend	Creates a data path between an RTP or T.38 coder running on a DSP channel and the IOP's Ethernet driver.
upConnectPktRecv	Creates a data path from a UDP receive socket to an RTP or T.38 decoder running on a DSP channel.

Table B-1. IOP functions

Function	Description
upDisconnectPktSend	Stops the forwarding of packets from a DSP channel to an IP socket and de-allocates all associated IOP and DSP resources.
upDisconnectPktRecv	Stops the forwarding of packets from an IP socket to a DSP channel and de-allocates all associated IOP and DSP resources.
Advanced functions	
iopGetNWPktBuf	Receives the next buffer of packets from the DSP.
iopJitterControl	Updates the IP network buffer control parameters.
iopSendNWPktBuf	Sends network packets to the DSP.
iopSendMsg	Sends a message from the Host to the specified IOP or DSP.
iopSetNWNotify	Sets handler to be invoked for notifying arrival of network packets from the DSP.
upSetUserMsgHandler	MsgHandlerFunc is called whenever a TASK user message is received by the IOP with a msgType not used by UPA.

getBoardInfo

Provides a board's information structure, as defined by the IOP monitor's SP6K_BOARD_INFO_T structure.

This function reports the extents of the system by identifying the number and type of processors, T1/E1 framers, HDLC codecs, and TDM switches on the board. Also, you can use hostGetSystemInfo to determine the number and type of SPIRIT boards in the system.

For more information, see the sp6k_board_info_tsp6k.h file.

Syntax

```
int getBoardInfo(
   INOUT SP6K_BOARD_INFO_T *pBoardInfo
)
```

Parameters

pBoardInfo

A pointer to SP6K_BOARD_INFO_T, the structure that stores board information:

SP6K_BOARD_INFO_T structure

```
typedef struct {
   UINT32       post_results;
   UINT32       free_zone;
   UINT32       flash_size;
   UINT32       memory_size;
   UCHAR        board_revision;
   UCHAR        num_dsps;
   UCHAR        hmic_present;
   UCHAR        board_type;
   UINT32       error_code;
   C6X_INFO_T   dsp_info[ MAX_DSPS ];
} SP6K_BOARD_INFO_T;
```

post_results
POST results.

free_zone Address of host free memory zone.

flash_size
Flash ROM size.

memory_size
Memory size.

board_revision
Board revision.

num_dsps The number of DSPs in the configuration.

hmic_present
HMIC/T8100 presence. The lower nibble identifies the installed T810x, and the upper nibble identifies the installed

option card, if one exists.

Since all deliverable SPIRIT boards have a T810x (HMIC) installed, this field is not used for additional information. The lower nibble identifies the type of T810x installed. The upper nibble identifies the type of option card (if any) installed.

board_type

Specifies the board type. You can select one of these:

Single

Dual

Quad

error_code

Error code storage (for IOPDRV).

dsp_info[MAX_DSPS]

DSP POST results and memory sizes.

Return values

SUCCESS Successful completion.

FAILURE Unsuccessful completion.

iopControlPeripheral

Configures specified peripheral.

Syntax

```
int iopControlPeripheral (
    IN int peripheral,
    IN int cmd,
    IN t_configArg *pArg
)
```

Parameters

peripheral
: The peripheral. You can select one of these values:

  TASK_E1

  TASK_T1

  TASK_T8100

cmd
: Command to send to the peripheral. You can use one of these values:

  CONFIG_E1
  : Instructs the framer to use E1 line protocol.

  CONFIG_T1
  : Instructs the framer to use T1 line protocol.

  CONFIG_T8100_DEFAULT
  : Reserved.

  CONFIG_T8100_SWITCHING
  : A pointer to the structure that specifies T8100 switching:

t_T8100SwitchConfig structure

```
typedef struct {
    ulong               number_of_connections;
    t_T8100Connection   *connections;
} t_T8100SwitchConfig;
```

number_of_connections
: The number of T8100 connections to be made. The maximum connections per call is 256.

**connections*
: Pointer to a list that contains *number_of_connections* connections.

CONFIG_T8100_CLOCKS
: A pointer to the structure that specifies the clock the TDM switch uses:

t_T8100ClockConfig structure

```
typedef struct {
    t_ref_clk        reference_clk_select;
    t_netref_clk     netref_select;
    t_fallback_clk   allback_clk_select;
    BOOL32           netref_enable;
    BOOL32           netref2_enable;
    BOOL32           frame_clk_a_enable;
    BOOL32           frame_clk_b_enable;
    BOOL32           compat_clks_enable;
    BOOL32           fallback_enable;
} t_T8100ClockConfig;
```

reference_clk_select
: Specifies the primary clock source to be used by the T8100.

netref_select
: Specifies the clock source used by the T8100 to generate its output CT_NETREF signal.

fallback_clk_select
: Specifies the fallback clock source used by the T8100. That is, the clock used when the primary clock source selected via *reference_clk_select* fails.

netref_enable
: Enables the CT_NETREF signal, generated by the T8100, onto the CT Bus.

netref2_enable
: Enables the CT_NETREF_2 signal, generated by the T8100, onto the CT Bus.

framer_clk_a_enable
: Enables the CT_C8_A (clock) and CT_FRAME_A (frame sync) signals, generated by the T8100, onto the bus.

framer_clk_b_enable
: Enables the CT_C8_B (clock) and CT_FRAME_B (frame sync) signals, generated by the T8100, onto the bus.

compat_clks_enable
: Enables the MVIP and SCSA compatibilty clock and strobe signals, generated by the T8100, onto the bus.

fallback_enable
Enables clock fallback. When a clock error occurs on the current clock reference, the T8100 will fall back to *fallback_clk_select*.

CONFIG_T8100_STREAMS
A pointer to the structure that specifies the T8100 stream parameters:

t_T8100StreamConfig structure

```
typedef struct {
    t_stream_rate    dsp_bsp0_rate;
    t_stream_rate    dsp_bsp1_rate;
    t_stream_rate    e1t1_rate;
    t_stream_rate    ct_bus_03_00_rate;
    t_stream_rate    ct_bus_07_04_rate;
    t_stream_rate    ct_bus_11_08_rate;
    t_stream_rate    ct_bus_15_12_rate;
    t_stream_rate    ct_bus_19_16_rate;
    t_stream_rate    ct_bus_23_20_rate;
    t_stream_rate    ct_bus_27_24_rate;
    t_stream_rate    ct_bus_31_28_rate;
} t_T8100StreamConfig;
```

dsp_bsp0_rate
Specifies the TDM stream rate for Buffered Serial Port 0 of the DSPs. Currently this value must be the same as *dsp_bsp1_rate*.

dsp_bsp1_rate
Specifies the TDM stream rate for Buffered Serial Port 1 of the DSPs. This value must be the same as *dsp_bsp0_rate*.

e1t1_rate Specifies the TDM stream rate for the E1/T1 framers. Under most conditions, this rate should be 2MHz.

ct_bus_03_00_rate
Specifies the TDM stream rate for CT bus streams 0 through 3.

ct_bus_07_04_rate
Specifies the TDM stream rate for CT bus streams 4 through 7.

ct_bus_11_08_rate
Specifies the TDM stream rate for CT bus streams 8 through 11.

ct_bus_15_12_rate
Specifies the TDM stream rate for CT bus streams 12 through 15.

ct_bus_19_16_rate

Specifies the TDM stream rate for CT bus streams 16 through 19.

ct_bus_23_20_rate
Specifies the TDM stream rate for CT bus streams 20 through 23.

ct_bus_27_24_rate
Specifies the TDM stream rate for CT bus streams 24 through 27.

ct_bus_31_28_rate
Specifies the TDM stream rate for CT bus streams 28 through 31.

pArg A pointer to the union of configuration structure. This union contains structures that have peripheral control information:

t_configArg structure

```
    typedef union {
    t_T1_user_config_struct     t1Config;
    t_E1_user_config_struct     e1Config;
    t_T8100ClockConfig          t8100ClkCfg;
    t_T8100SwitchConfig         t8100SwitchCfg;
    t_T8100StreamConfig         t8100StreamCfg;
} t_configArg;
```

t_T1_user_config_struct
Configuration structure for T1.

t_E1_user_config_struct
Configuration structure for e1.

t_T8100ClockConfig
Configuration structure for t8100 clock.

t_T8100SwitchConfig
Configuration structure for t8100 switch.

t_T8100StreamConfig
Configuration structure for t8100 stream.

Return values

SUCCESS Successful completion.

FAILURE Unsuccessful completion.

iopInit

Initializes the IOP API.

Syntax

```
int iopInit (void)
```

Parameters

None.

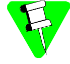
Return values

SUCCESS	Successful completion.
FAILURE	Unsuccessful completion.

upConfigService

Configures a specified service based on service type for a specific channel.

Also enables the service after configuration.



- Enabling the stCodec service disables the stTdmToneGen and stPktToneGen services.
- For descriptions of available services, see Appendix F, Service descriptions.

Syntax

```
UP_ERROR_ET upConfigService(
    RSYS_INT32 lUnit,
    RSYS_INT32 lChannel,
    UP_SERVICE_ET eService,
    UP_CONFIG_SVC_MSG_UT *putConfigService
);
```

Parameters

lUnit Specifies:

- For most services: the DSP unit number.
- For IOP-based services such as stCAS: the framer unit number.

lChannel Specifies:

- For most services: the virtual channel number.
- For IOP-based services such as stCAS: the timeslot number.

eService Identifies the service you want to configure:

stCodec Converts voice, fax, or modem data between TDM and a packetized form.

stEchoCanc
Tries to remove time-delayed versions of a TDM channel's output from its input stream.

stAGC Provides Automatic Gain Control for TDM input data.

stTdmDTMFDet
Performs DTMF detection on TDM input data.

stPktDTMFDet
Performs DTMF detection on decompressed packet data before it is output to a TDM stream.

stCPTDet Performs Call Progress Tone detection on TDM input data.

stMFDet Performs MF Tone detection on TDM input data for R1 or R2 signaling.

stPktToneGen
Generates tones or tone-pairs to a TDM output with programmable cadence and cadence count.

stRtpEncode
: Performs RTP Packetization and, when used with the stJitterBuf service, depacketization.

stRtpDecode
: Implements both RTP decode functionality and a dynamic jitter buffer that feeds compressed frames to a codec for decompression on a strict schedule regardless of input variations.

stTdmToneGen
: Generates tones or tone-pairs to a TDM output with programmable cadence and cadence count.

stT1E1Alarm
: Notifies the user of an alarm signaled by a T1 or E1 framer, and allows control of the LEDs associated with each span.

stCAS
: Implements Channel Associated Signaling on T1 and E1 lines which are attached locally to framers control by the I/O Processor.

stPacketBuilder
: This is an embedded service controlled by the upConnectPktSend function.

stPacketParser
: Takes an IP/UDP packet from the Ethernet->DSP FIFO, checksums headers and data as appropriate, strips headers, and passes the payload to the next module, usually stRtpDecode or stT38, in TASK2.

stEthernetAlarm
: Detects Ethernet Link status changes and generates UP_EVT_ETHERNET_ALARM events, which notify a user application of the change.

stQDS0Hdlc
: Concatenates four sub-rate HDLC channels into a full DS0.

For detailed information about TASK services, see Appendix F, Service descriptions.

putConfigService
A pointer to the configuration data structure:

UP_CONFIG_SVC_UT structure

```
typedef union {
    UP_CODEC_CONFIG_ST              tCodecConfig;
    UP_ECHO_CONFIG_ST               tEchoCancConfig;
    UP_AGC_CONFIG_ST                tAGCConfig;
    UP_DTMF_CONFIG_ST               tDTMFConfig;
    UP_CPT_CONFIG_ST                tCPTConfig;
    UP_MF_CONFIG_ST                 tMFConfig;
    UP_TONEGEN_CONFIG_ST            tToneGenConfig;
```

```
    UP_RTP_SEND_CONFIG_ST         tRtpSendConfig;
    UP_RTP_RECV_CONFIG_ST         tRtpRecvConfig;
    UP_T38_CONFIG_ST              tT38Config;
    UP_T1E1ALARM_CONFIG_ST        tT1E1AlarmConfig;
    UP_CAS_CONFIG_ST              tCasConfig;
    UP_PACKET_BUILDER_CONFIG_ST   tPBConfig;
    UP_PACKET_PARSER_CONFIG_ST    tPPConfig;
    UP_IOPINITCONFIG_ST           tIopInitData;
} UP_CONFIG_SVC_UT;
```

Return values

UP_SUCCESS

Successful completion.

If the function cannot return an error in real time (for example, the function sends a message to another processor to implement a configuration change), the function returns this value, indicating that the message was sent properly. The processor implementing the function returns one of these values:

UP_EVT_CONFIG_ERROR

P_EVT_CONFIG_ERROR

The new channel or service configuration is invalid.

UP_EVT_CONFIG_ACK

(for debugging) acknowleges each configuration action.

UP_CONFIG_ERR

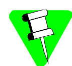
Configuration error.

UP_INVALID_ARG

Invalid argument.

upConfigServiceGlobal

Configures a specified service based on service type for all channels.



For descriptions of available services, see *Appendix F, Service descriptions*.

Syntax

```
UP_ERROR_ET upConfigServiceGlobal(
    RSYS_INT32 lUnit,
    UP_SERVICE_ET eService,
    UP_GLOBALCONFIGDATA_UT *putConfigGlobal
)
```

Parameters

lUnit Specifies:

- For most services: the DSP unit number.
- For IOP-based services such as stCAS: the framer unit number.

service Identifies the service you want to configure:

stCodec Converts voice, fax, or modem data between TDM and a packetized form.

stEchoCanc
Tries to remove time-delayed versions of a TDM channel's output from its input stream.

stAGC Provides Automatic Gain Control for TDM input data.

stTdmDTMFDet
Performs DTMF detection on TDM input data.

stPktDTMFDet
Performs DTMF detection on decompressed packet data before it is output to a TDM stream.

stCPTDet Performs Call Progress Tone detection on TDM input data.

stMFDet Performs MF Tone detection on TDM input data for R1 or R2 signaling.

stPktToneGen
Generates tones or tone-pairs to a TDM output with programmable cadence and cadence count.

stRtpEncode
Performs RTP Packetization and, when used with the stJitterBuf service, depacketization.

stRtpDecode
: Implements both RTP decode functionality and a dynamic jitter buffer that feeds compressed frames to a codec for decompression on a strict schedule regardless of input variations.

stTdmToneGen
: Generates tones or tone-pairs to a TDM output with programmable cadence and cadence count.

stT1E1Alarm
: Notifies the user of an alarm signaled by a T1 or E1 framer, and allows control of the LEDs associated with each span.

stCAS
: Implements Channel Associated Signaling on T1 and E1 lines which are attached locally to framers control by the I/O Processor.

stPacketBuilder
: This is an embedded service controlled by the upConnectPktSend function.

stPacketParser
: Takes an IP/UDP packet from the Ethernet->DSP FIFO, checksums headers and data as appropriate, strips headers, and passes the payload to the next module, usually stRtpDecode or stT38, in TASK2.

stEthernetAlarm
: Detects Ethernet Link status changes and generates UP_EVT_ETHERNET_ALARM events, which notify a user application of the change.

stQDS0Hdlc
: Concatenates four sub-rate HDLC channels into a full DS0.

For detailed information about TASK services, see Appendix F, Service descriptions.

putConfigGlobal
: A pointer to the configuration data structure:

UP_GLOBALCONFIGDATA_UT structure

```
typedef union {
    UP_DSPINITCONFIG_ST tInitData;
} UP_GLOBALCONFIGDATA_UT;
```

This structure is currently only used by UPA initialization code.

Return values

UP_SUCCESS
: Successful completion.

UP_CONFIG_ERR
: Configuration error.

UP_INVALID_ARG
Invalid argument.

upConnectPktRecv

Creates a data path from a UDP receive socket to an RTP or T.38 decoder running on a DSP channel.

As each packet is received it will be DMA'd to a FIFO in the DSP specified by *lUnit*. During the DSP's inter-frame time the packets in the FIFO are sorted by channel and their payloads are passed either to the RTP Receive/Jitter Buffer service, or to the appropriate T.38 FIFO.

An asynchronous UP_EVT_PORT_ASSIGNED event will be emitted on success to identify the UDP port number that was allocated to the connection. A UP_EVT_CONFIG_ERROR shall be emitted in situations where to port number could not be allocated (port already assigned or DSP channel already in use).

Syntax

```
UP_ERROR_ET upConnectPktRecv(
    RSYS_INT32 lUnit,
    RSYS_INT32 lChannel,
    UP_PKT_RECV_CONFIG_ST *pstPktRecvConfig
);
```

Parameters

`lUnit` The DSP unit number

`lChannel` The DSP virtual channel number

`pstPktRecvConfig`

A pointer to the UP_PKT_RECV_CONFIG_ST structure:

UP_PKT_RECV_CONFIG_ST structure

```
typedef struct {
    RSYS_UINT32  ulReceivePort;
    RSYS_UINT32  ulInterface;
} UP_PKT_RECV_CONFIG_ST;
```

`ulReceivePort`

The 16-bit port number of the socket which will receive the packets. If this number is specified as zero an appropriate port number will be chosen from the range of UP_PREALLOC_PORT_START through UP_PREALLOC_PORT_START + UP_PREALLOC_RTP_PORTS (defined as 5000 – 5999 in upa.h) and returned via an event. If this number is specified manually it must fall in the range through UP_PREALLOC_PORT_START + UP_PREALLOC_RTP_PORTS through UP_PREALLOC_PORT_START + UP_MAX_RTP_PORTS (defined as 6000 – 6999 in upa.h) and must be even.

ulReceivePort

The 16-bit port number of the socket which receives the packets. You can use one of these values:

0	An appropriate port number is chosen from the range of UP_PREALLOC_PORT_START through (UP_PREALLOC_PORT_START + UP_PREALLOC_RTP_PORTS) (defined as 5000–5999 in upa.h) and returned via an event.
Other	The number must fall in the range of (UP_PREALLOC_PORT_START + UP_PREALLOC_RTP_PORTS) through (UP_PREALLOC_PORT_START + UP_MAX_RTP_PORTS) (defined as 6000–6999 in upa.h), and must be an even number.

ulInterface

The unit number of the Ethernet interface on which to send packets.

Return values

UP_SUCCESS

Successful completion.

If the function cannot return an error in real time (for example, the function sends a message to another processor to implement a configuration change), the function returns this value, indicating that the message was sent properly. The processor implementing the function returns one of these values:

UP_EVT_CONFIG_ERROR

UP_EVT_CONNECT_RECV

UP_EVT_PORT_ASSIGNED

UP_CONFIG_ERR

Configuration error.

UP_INVALID_ARG

Invalid argument.

UP_DSP_ERR

DSP channel assignment error—fast packet.

upConnectPktSend

Creates a data path between an RTP or T.38 coder running on a DSP channel and the IOP's Ethernet driver.

As each packet is encoded by the codec, it will be passed to the stPacketBuilder service which will add UDP, IP, and Ethernet headers to the packet and then add it to a DSP->IOP FIFO. On the expiration of a timer tick the packets in the fifo will be added to the transmit queue of the appropriate Ethernet controller and transmitted to the network addressable device specified by the Ethernet frame's destination MAC address.

An asynchronous UP_EVT_CONFIG_ACK event will occur in response to this function if the connection is established. A UP_EVT_CONFIG_ERROR will occur if the function fails; for instance if an ARP of ulDestRouter (destination IP address) fails.

Syntax

```
UP_ERROR_ET upConnectPktSend(
    RSYS_INT32 lUnit,
    RSYS_INT32 lChannel,
    UP_PKT_SEND_CONFIG_ST *pstPktSendConfig
);
```

Parameters

lUnit The DSP unit number

lChannel The DSP virtual channel number

pstPktSendConfig
A pointer to the UP_PKT_SEND_CONFIG_ST structure:

UP_PKT_SEND_CONFIG_ST structure

```
typedef struct {
    RSYS_UINT32   ulInterface;
    RSYS_UINT32   ulRouterAddress;
    RSYS_UINT32   ulDestAddress;
    RSYS_UINT32   ulDestPort;
    RSYS_UINT32   ulServiceType;
    RSYS_UINT32   ulSrcPort;
} UP_PKT_SEND_CONFIG_ST;
```

ulInterface
The unit number of the Ethernet interface on which packets should be sent.

ulRouterAddress
The 32-bit IP address, in host order, of a router on the same network as the IOP which should be used to forward packets to `ulDestAddress`. If `ulDestAddress` is on the same network as the IOP and no router should be used, then `ulDestRouter` should be set to `ulDestAddress`.

ulDestAddress
The 32-bit IP address, in host order, of the RTP or T.38 packets' destination.

ulDestPort
The 16-bit UDP port number of the RTP or T.38 packets' destination.

ulServiceType
An 8-bit value used to specify the precedence, delay, throughput, and reliability of a message. This is used in making quality of service decisions in the delivery of a packet.

ulSrcPort The optional 16-bit UDP port number that identifies the RTP or T.38 source application peer. If the source port is not provided, the values of *ulUnit* and *lChannel* passed with the function are used to find the corresponding port ID registered to the inbound fast packet channel established through upDisconnectPktRecv. This approach assumes that upDisconnectPktSend is preceded by upConnectPktRecv, and that each path makes up one part of full-duplex connection on a specified DSP and channel. A source port ID of 0 is assigned if the user does not specify a source port in their call to upConnectPktSend and if an upConnectPktRecv was not successfully executed prior to calling upConnectPktSend on the specified DSP and Channel numbers.

Return values

UP_SUCCESS
Successful completion.

If the function cannot return an error in real time (for example, the function sends a message to another processor to implement a configuration change), the function returns this value, indicating that the message was sent properly. The processor implementing the function returns one of these values:

UP_EVT_CONFIG_ERROR

UP_EVT_CONNECT_SEND

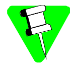
UP_FAILURE
General failure.

UP_INVALID_ARG
Invalid argument.

upDisableService

Disables a specified service based on service type for a specific channel.

Services not dependent on this service are not be affected. Major services (codecs, echo canceller) when disabled this way consume no DSP resources. Minor services (tone detectors and generators, etc.) still load but don't run, reducing their resource load to a minimum but not zero.



- Enabling the stCodec service disables the stTdmToneGen and stPktToneGen services.
- For descriptions of available services, see Appendix F, Service descriptions.

Syntax

```
UP_ERROR_ET upDisableService(
    RSYS_INT32 lUnit,
    RSYS_INT32 lChannel,
    UP_SERVICE_ET eService
);
```

Parameters

lUnit Specifies:

- For most services: the DSP unit number.
- For IOP-based services such as stCAS: the framer unit number.

lChannel Specifies:

- For most services: the virtual channel number.
- For IOP-based services such as stCAS: the timeslot number.

eService Identifies the service you want to disable:

stCodec Converts voice, fax, or modem data between TDM and a packetized form.

stEchoCanc
Tries to remove time-delayed versions of a TDM channel's output from its input stream.

stAGC Provides Automatic Gain Control for TDM input data.

stTdmDTMFDet
Performs DTMF detection on TDM input data.

stPktDTMFDet
Performs DTMF detection on decompressed packet data before it is output to a TDM stream.

stCPTDet Performs Call Progress Tone detection on TDM input data.

stMFDet Performs MF Tone detection on TDM input data for R1 or R2 signaling.

stPktToneGen
: Generates tones or tone-pairs to a TDM output with programmable cadence and cadence count.

stRtpEncode
: Performs RTP Packetization and, when used with the stJitterBuf service, depacketization.

stRtpDecode
: Implements both RTP decode functionality and a dynamic jitter buffer that feeds compressed frames to a codec for decompression on a strict schedule regardless of input variations.

stTdmToneGen
: Generates tones or tone-pairs to a TDM output with programmable cadence and cadence count.

stT1E1Alarm
: Notifies the user of an alarm signaled by a T1 or E1 framer, and allows control of the LEDs associated with each span.

stCAS
: Implements Channel Associated Signaling on T1 and E1 lines which are attached locally to framers control by the I/O Processor.

stPacketBuilder
: This is an embedded service controlled by the upConnectPktSend function.

stPacketParser
: Takes an IP/UDP packet from the Ethernet->DSP FIFO, checksums headers and data as appropriate, strips headers, and passes the payload to the next module, usually stRtpDecode or stT38, in TASK2.

stEthernetAlarm
: Detects Ethernet Link status changes and generates UP_EVT_ETHERNET_ALARM events, which notify a user application of the change.

stQDS0Hdlc
: Concatenates four sub-rate HDLC channels into a full DS0.

For detailed information about TASK services, see Appendix F, Service descriptions.

Return values

UP_SUCCESS
: Successful completion.

UP_CONFIG_ERR
: Configuration error.

UP_INVALID_ARG

Invalid argument.

upDisconnectPktRecv

Stops the forwarding of packets from an IP socket to a DSP channel and de-allocates all associated IOP and DSP resources.

Syntax

```
UP_ERROR_ET upDisconnectPktRecv(
    RSYS_INT32 lUnit,
    RSYS_INT32 lChannel
);
```

Parameters

lUnit The DSP unit number.

lChannel The DSP virtual channel number.

Return values

UP_SUCCESS
Successful completion.

UP_CONFIG_ERR
Configuration error.

UP_INVALID_ARG
Invalid argument.

UP_DSP_ERR
DSP channel assignment error—fast packet.

upDisconnectPktSend

Stops the forwarding of packets from a DSP channel to an IP socket and de-allocates all associated IOP and DSP resources.

Syntax

```
UP_ERROR_ET upDisconnectPktSend(
    RSYS_INT32 lUnit,
    RSYS_INT32 lChannel
);
```

Parameters

lUnit	The DSP unit number.
lChannel	The DSP virtual channel number.

Return values

UP_SUCCESS
: Successful completion.

UP_CONFIG_ERR
: Configuration error.

UP_INVALID_ARG
: Invalid argument.

upEnableChannel

Enables transmit and receive paths of a specified channel.

lChannel does not display in messages returned via upGetNWMsg or forwarded to an RTP socket unless the channel is enabled via this function.

Syntax

```
UP_ERROR_ET upEnableChannel(
   RSYS_INT32 lUnit,
   RSYS_INT32 lChannel,
   RSYS_INT32 lTxEnable,
   RSYS_INT32 lRxEnable
);
```

Parameters

lUnit The DSP unit number.

lChannel The DSP virtual channel number.

lTxEnable Controls a channel's transmit path processing.

- TRUE Enables transmit path processing.
- FALSE Disables transmit path processing.

lRxEnable Controls a channel's receive path processing.

- TRUE Enables receive path processing.
- FALSE Disables receive path processing.

Return values

UP_SUCCESS
: Successful completion.

If the function cannot return an error in real time (for example, the function sends a message to another processor to implement a configuration change), the function returns this value, indicating that the message was sent properly. The processor implementing the function returns one of these values:

UP_EVT_CONFIG_ERROR

UP_CONFIG_ERR
: Configuration error.

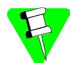
UP_INVALID_ARG
: Invalid argument.

Comments

`lTxEnable` and `lRxEnable` may be used to separately control the transmit and receive path processing for a channel, e.g., a channel which generates tones may not need any receive path processing. However, while this reduces the DSP processing time associated with transmit or receive algorithms, it does not eliminate the overhead of loading the algorithm into DSP memory. Only when `lTxEnable` and `lRxEnable` are both false is the channel entirely disabled, which removes its overhead entirely.

upEnableService

Enables a specified service based on service type for the specified channel. upConfigService also implicitly enables a service when it is configured.



- Enabling the stCodec service disables the stTdmToneGen and stPktToneGen services.
- For descriptions of available services, see *Appendix F, Service descriptions*.

Syntax

```
UP_ERROR_ET upEnableService(
    RSYS_INT32 lUnit,
    RSYS_INT32 lChannel,
    UP_SERVICE_ET eService
);
```

Parameters

lUnit Specifies:

- For most services: the DSP unit number.
- For IOP-based services such as stCAS or stT1E1Alarm: the framer unit number.

lChannel Specifies:

- For most services: the virtual channel number.
- For IOP-based services such as stCAS: the timeslot number.

eService Identifies the service you want to enable:

stCodec Converts voice, fax, or modem data between TDM and a packetized form.

stEchoCanc
Tries to remove time-delayed versions of a TDM channel's output from its input stream.

stAGC Provides Automatic Gain Control for TDM input data.

stTdmDTMFDet
Performs DTMF detection on TDM input data.

stPktDTMFDet
Performs DTMF detection on decompressed packet data before it is output to a TDM stream.

stCPTDet Performs Call Progress Tone detection on TDM input data.

stMFDet Performs MF Tone detection on TDM input data for R1 or R2 signaling.

stPktToneGen
Generates tones or tone-pairs to a TDM output with programmable cadence and cadence count.

stRtpEncode
Performs RTP Packetization and, when used with the stJitterBuf service, depacketization.

stRtpDecode
Implements both RTP decode functionality and a dynamic jitter buffer that feeds compressed frames to a codec for decompression on a strict schedule regardless of input variations.

stTdmToneGen
Generates tones or tone-pairs to a TDM output with programmable cadence and cadence count.

stT1E1Alarm
Notifies the user of an alarm signaled by a T1 or E1 framer, and allows control of the LEDs associated with each span.

stCAS
Implements Channel Associated Signaling on T1 and E1 lines which are attached locally to framers control by the I/O Processor.

stPacketBuilder
This is an embedded service controlled by the upConnectPktSend function.

stPacketParser
Takes an IP/UDP packet from the Ethernet->DSP FIFO, checksums headers and data as appropriate, strips headers, and passes the payload to the next module, usually stRtpDecode or stT38, in TASK2.

stEthernetAlarm
Detects Ethernet Link status changes and generates UP_EVT_ETHERNET_ALARM events, which notify a user application of the change.

stQDS0Hdlc
Concatenates four sub-rate HDLC channels into a full DS0.

For detailed information about TASK services, see Appendix F, Service descriptions.

Return values

UP_SUCCESS
Successful completion.

UP_CONFIG_ERR
Configuration error.

UP_FAILURE
General failure.

UP_INVALID_ARG
Invalid argument.

UP_RTP_ERR
RTP port assignment error—fast packet.

UP_DSP_ERR
DSP channel assignment error—fast packet.

UP_START_TIMEOUT
A timeout occurred on a blocking call to upStart.

UP_HDLC_PORT_ERR
HDLC port configuration error.

UP_HDLC_CHAN_ENABLE_ERR
HDLC channel enable error.

UP_HDLC_CHAN_CONFIG_ERR
HDLC channel configuration error.

upQueryQOSReport

Causes the specified DSP specified to send a Quality of Service report as an UP_EVT_STATISTICS_RPT event.

An RTCP stack can use the provided information to create Sender and Receiver Reports.

Syntax

```
UP_ERROR_ET upQueryQOSReport(
    RSYS_INT32 lUnit,
    SYS_INT32 lChannel
);
```

Parameters

lUnit	The DSP number.
lChannel	The DSP virtual channel number.

Return values

UP_SUCCESS

Successful completion.

If the function cannot return an error in real time (for example, the function sends a message to another processor to implement a configuration change), the function returns this value, indicating that the message was sent properly. The processor implementing the function returns one of these values:

UP_EVT_CONFIG_ERROR

UP_EVT_STATISTICS_RPT
UP_STATISTICS_ST

```
typedef struct {
    RSYS_UINT32     sender_packet_count;
    RSYS_UINT32     sender_octet_count;
    RSYS_UINT32     frac_lost;
    RSYS_UINT32     cum_packet_lost_count;
    RSYS_UINT32     cum_late_packet_count;
    RSYS_UINT32     ext_hi_seq_recv;
    RSYS_UINT32     interarrival_jitter;
    RSYS_INT32      lSilencePercentTx;
    RSYS_INT32      lSilencePercentRx;
    RSYS_UINT32     recv_packet_count;
    RSYS_UINT32     recv_octet_count;
    RSYS_UINT32     pkt_p_no_pkt_count;
    RSYS_UINT32     pkt_p_fifo_err1_count;
    RSYS_UINT32     pkt_p_fifo_err2_count;
    RSYS_UINT32     pkt_p_len_err_count;
    RSYS_UINT32     pkt_p_ip_cksum_count;
    RSYS_UINT32     pkt_p_udp_cksum_count;
    RSYS_UINT32     pkt_b_error_count;
    RSYS_UINT32     rtp_unsupport_pt_count;
    RSYS_UINT32     rtp_jb_resync_count;
```

```
    RSYS_UINT32       rtp_seq_dup_count;
    RSYS_UINT32       rtp_jb_full_count;
    RSYS_UINT32       rtp_ssrc_change_count;
    RSYS_UINT32       ulSysCycle;
    RSYS_UINT32       ulThreadAvg;
    RSYS_UINT32       ulThreadMax;
    RSYS_UINT32       ulUsage;
} UP_STATISTICS_ST;
```

Parameters

sender_packet_count
: The number of packets sent by the RTP encoder

sender_octet_count
: The number of bytes of payload sent by the RTP encoder

frac_lost
: The ratio packets_lost/packets_expected as an 8-bit fraction

cum_packet_lost_count
: The count of packets which were not available to play when required

cum_late_packet_count
: The count of packets which were not available to play when required but which subsequently arrived and were discarded

ext_hi_seq_recv
: The highest sequence number received thus far in an RTP session

interarrival_jitter
: The jitter in packet arrival times, computed according to H.225

lSilencePercentTx
: A number from 0 to 100 indicating the percentage silence suppression transmitted since the last upConfigService(stRTP,…)

lSilencePercentRx
: A number from 0 to 100 indicating the percentage silence suppression received since the last upConfigService(stRTP,…)

recv_packet_count
: The number of packets received by the RTP decoder

recv_octet_count
: The number of bytes of payload received by the RTP decoder

ulSysCycle
The timer count of DSP system cycle

ulThreadAvg
The timer counter in average spent by TDM thread

ulThreadMax
The maximum timer counter spent by TDM thread since the DSP is loaded.

ulUsage A number from 0 to 100 indicating *ulThreadMax/ ulSysCycle*.

All the other members of the structure are reserved for Radisys internal use only.

UP_CONFIG_ERR
Configuration error.

UP_INVALID_ARG
Invalid argument.

upSetEventHandler

Passes information received from DSPs to the event handler.

Syntax

```
UP_ERROR_ET upSetEventHandler(
    void (*eventHandlerFunc)(UP_EVENT_DATA_ST*)
);
```

Parameters

EventHandlerFunc(UP_EVENT_DATA_ST*)

Receives notification from the UP_EVENT_DATA_ST structure of DSP events such as tone detection, peripheral state changes such as hook and other CAS signals, and any other events any service generates.

UP_EVENT_DATA_ST structure

```
typedef struct
{
    UP_SERVICE_ET                 eService;
    UP_EVENT_ET                   eEventNum;
    RSYS_UINT32                   ulSlot;
    RSYS_UINT32                   ulUnit;
    RSYS_UINT32                   ulChannel;
    RSYS_UINT32                   ulEventDataLength;
    union
    {
        UP_DTMF_DETECTED_DATA_ST   sDtmfDetectedData;
        UP_CPT_DETECTED_DATA_ST    sCptDetectedData;
        UP_MF_DETECTED_DATA_ST     sMfDetectedData;
        UP_RTP_PT_CHANGE_DATA_ST   sRtpPtData;
        UP_RTP_SSRC_CHANGE_DATA_ST sRtpSsrcData;
        UP_CAS_CHANGE_DATA_ST      sCasChangeData;
        UP_STATISTICS_ST           sStatistics;
        UP_CONNECT_SEND_RPT_ST     sConnectSendRpt;
        UP_ERROR_ET                etErrorCode;
        RSYS_INT32                 lRtpPortID;
        UP_START_DSP_REPLY_ST      tStartReply;
        UP_T1E1_ALARM_DATA_ST      tT1E1AlarmData;
        UP_ETHERNET_ALARM_DATA_ST  tEthernetAlarmData;
        UP_STREAM_CONNECT_DATA_ST  tStreamConnectData;
        UP_HDLC_REPORT_DATA_ST     tHdlcReportData;
    } uEventData;
} UP_EVENT_DATA_ST;
```

eService	The service (from t_UP_SERVICE enum) which sent the event.
eEventNum	The event from this service.
ulUnit	The device unit number (i.e. for a DSP-based service this holds the DSP number).
ulChannel	The channel or timeslot within device `unit`.

ulEventDataLength

The length of the associated data.

uEventData

Further information about the event, when the event requires more than *eEventNum* to describe it.

Return values

UP_SUCCESS

Successful completion.

If the function cannot return an error in real time (for example, the function sends a message to another processor to implement a configuration change), the function returns this value, indicating that the message was sent properly. The processor implementing the function returns one of these values:

UP_EVT_CONFIG_ERROR

UP_EVENT_ET

```
typedef enum {
    UP_EVT_GENERAL_DSP_FAILURE = 1,
    UP_EVT_CONFIG_ERROR,
    UP_EVT_STATISTICS_RPT,
    UP_EVT_CONNECT_SEND,

    UP_EVT_CONNECT_RECV,
    UP_EVT_CONFIG_ACK,

    UP_EVT_TDM_DTMF_DETECTED,
    UP_EVT_PKT_DTMF_DETECTED,

    UP_EVT_CPT_DETECTED,
    UP_EVT_MF_DETECTED,

    UP_EVT_RTP_PT_CHANGE,
    UP_EVT_RTP_SSRC_CHANGE,

    UP_EVT_T1E1_ALARM
    UP_EVT_CAS_CHANGE
    UP_EVT_PORT_ASSIGNED,
    UP_EVT_FAILURE,
    UP_EVT_DSP_RUNNING,
    UP_EVT_ETHERNET_ALARM,
    UP_EVT_IOP_RUNNING,
    UP_EVT_ETHERNET_CONFIGURED,
    UP_EVT_STREAM_CONNECTED,
    UP_EVT_STREAM_FAILURE,
    UP_EVT_HDLC_REPORT
} UP_EVENT_ET;
```

UP_ERROR_ET

An enumerated data type passed in a UP_EVT_FAILURE event that contains the specific source of the UPA command processing failure.

P_START_DSP_REPLY_ST

A structure type passed in a UP_EVT_START_REPLY event that contains the FIFO address locations to use for setting up Fast Packet services. Receipt of the UP_EVT_START_REPLY event signals successful initialization of a loaded DSP.

UP_START_DSP_REPLY_ST structure

```
typedef struct {
   RSYS_UINT32      ulSendFifoAddr;
   RSYS_UINT32      ulSendFifoSize;
   RSYS_UINT32      ulRecvFifoAddr;
   RSYS_UINT32      ulRecvFifoSizePerChannel;
   RSYS_UINT32      ulRecvFifoChannels;
   RSYS_UINT32      ulAltSendFifoAddr;
} UP_START_DSP_REPLY_ST;
```

upStart

Initializes the Universal Port subsystem.

To perform necessary system initialization, call this function once before calling any other UPA function from the IOP. At this early stage of IOP initialization, IOP-based applications typically do not have access to information in the UP_IOPSYSCONFIG_ST structure. Therefore. IOP implementation typically pass a NULL pointer to upStart and allow the Host component, when making its call to upStart, to populate the contents of the UP_IOPSYSCONFIG_ST structure.

Syntax

```
UP_ERROR_ET upStart(
   UP_IOPSYSCONFIG_ST *ptIOPSysConfig
)
```

Parameters

ptIOPSysConfig

Points to the UP_IOPSYSCONFIG_ST structure, which configures the board during UPA initialization:

UP_IOPSYSCONFIG_ST structure

```
typedef struct {
   RSYS_UINT32        aulNICAddress[UP_MAX_SLOTS][MAX_NICS];
   RSYS_UINT32        ulHostIPAddress;
   UP_ENABLE_ET       etCommandAck;
   UP_ENABLE_ET       etEventForwarding;
   UP_LAN_CONTROL_ET  etLanControl;
} UP_IOPSYSCONFIG_ST;
```

aulNICaddress

Carries the IP addresses (host byte ordered) to associate with Ethernet adapters of installed IOPS. The IOP number is used as an index to de-reference the two Ethernet adapters (fei 0 and 1) installed. An IP address setting of zero for either interface 0 or 1 indicates that the user does not want UPA software to configure the Ethernet Adapter and bind an IP address to it. A non-zero entry causes UPA on the associated IOP to perform a vxWork's usrNetInit to initialize the adapter and a hostAdd to make a representative hostname to IP address mapping in the host table.

ulHostIPAddress

This element informs an IOP of its controlling host's IP address. This IP address is used by an IOP to establish a communication link to the host for event forwarding when either TCP or UDP based control has been configured.

etCommandAck

This enumeration specifies whether UPA should acknowledge each UPA command request with an asynchronous notification. This enumeration should be set to

'enumDisabled' when commands are issued by an IOP based application and should only be set to 'enumEnabled' when a host based application wants positive confirmation of command processing.

etEventForwarding

This enumeration specifies whether UPA should forward events emanating from an IOP to the Host. This forwarding is useful in determining the success or failure of an UPA command when issued from the host.

etLanControl

A flag that specifies the transport to use for communicating control messages between a controlling host and an IOP. You can select one of these:

`enumNoLanControl`

The PCI bus communicates control messages between host and IOP. This limits host control to IOPs located within the same shelf.

Return values

UP_SUCCESS

Successful completion.

UP_CONFIG_ERR

Configuration error.

iopGetNWPktBuf

Receives the next buffer of packets from the DSP.

The buffer contains one or more packets in the following format:

- Total buffer length
- Channel number (one word)
- Packet length (one word)
- One or more words of packet data

For details about the packet buffer contents, see *iopSendNWPktBuf* on page 143.

Syntax

```
int iopGetNWPktBuf (
    IN long dspNum,
    IN long *pBuf
)
```

Parameters

dspNum	The DSP number.
pBuf	A pointer to the buffer that receives network packets.

Return values

SUCCESS	Successful completion.
FAILURE	Unsuccessful completion.

iopJitterControl

Updates the IP network buffer control parameters.

Syntax

```
int iopJitterControl (
   IN long dspNum,
   IN long ChnlNum,
   IN t_jitterParam *pJitter
)
```

Parameters

dspNum The DSP number.

ChnlNum The channel number.

pJitter A pointer to the structure that contains jitter parameters:

t_jitterParam structure

```
typedef struct {
   long  nJitterBuf;
} t_jitterParam;
```

nJitterBuf
The number of jitter buffers between Host and IOP/DSP.

Return values

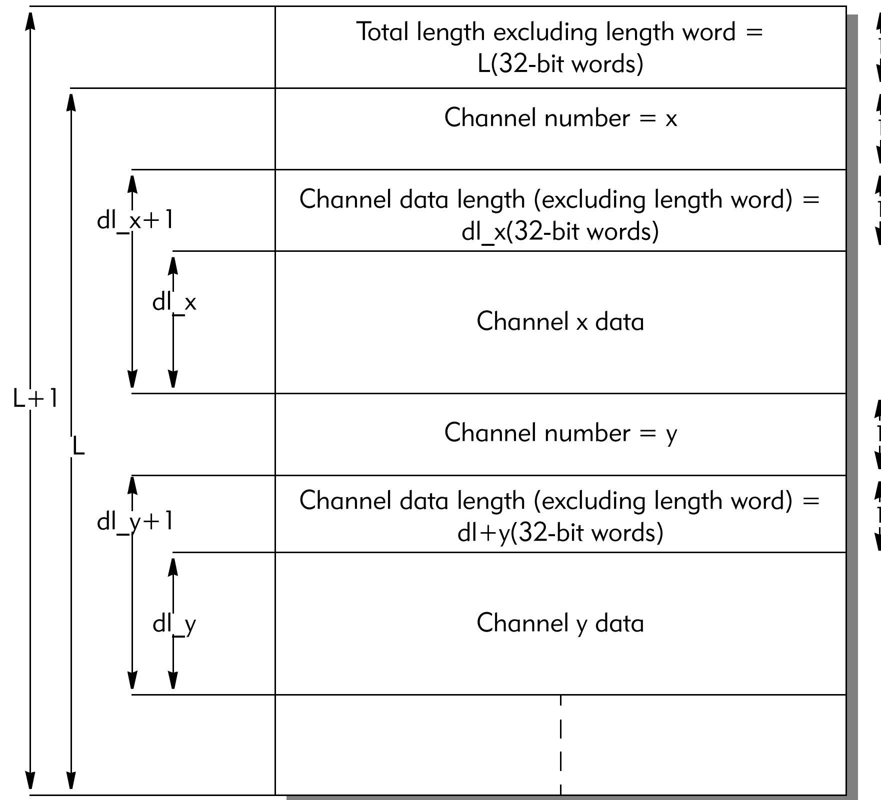
SUCCESS Successful completion.

FAILURE Unsuccessful completion.

iopSendNWPktBuf

Sends network packets to the DSP.

The packets are organized in the buffer as follows:



All numbers and sizes are in 32-bit words

Figure B-1. Packet organization buffer

Syntax

```
int iopSendNWPktBuf (
    IN long dspNum,
    IN long *pBuf
)
```

Parameters

dspNum The number of the DSP to which you want to send packets.

pBuf A buffer that consists of one or more network packets, as shown in Figure B-1.

Return values

SUCCESS Successful completion.

FAILURE Unsuccessful completion.

iopSendMsg

Sends a message from the Host to the specified IOP or DSP.

If the message is not delivered to the IOP within the specified time, the function returns failure.

Syntax

```
int iopSendMsg (
   IN long dst,
   IN long msgType,
   IN ulong msgSzW,
   IN long *pBuf,
   IN long waitTimeMsec
)
```

Parameters

dst The destination, in this format:

Device | *DeviceNum*

Device Enter either TASK_IOP or TASK_DSP.

DeviceNum Enter the device number.

Examples:

To invoke a handler from the second IOP, use (TASK_IOP | 2) as the source.

To invode a handler from the third DSP, use (TASK_DSP | 3) as source.

msgType The message type. You can use any positive integer.

msgSzW The number of words in the message.

pBuf The buffer that contains the message.

waitTimeMsec
The number of milliseconds that elapse before determining message status.

Return values

SUCCESS Successful completion.

FAILURE Unsuccessful completion.

iopSetNWNotify

Sets handler to be invoked for notifying arrival of network packets from the DSP.

The handler must take the DSP number as the argument.

Syntax

```
void iopSetNWNotify (
   IN void (* NWNotifyHndlr) (IN long dspNum)
)
```

Parameters

NWNotifyHndlr
A pointer to the handler that notifies the Host application of the arrival of network packets from the DSP.

dspNum The DSP number from which network packets were sent.

Return values

SUCCESS Successful completion.

FAILURE Unsuccessful completion.

upSetUserMsgHandler

MsgHandlerFunc is called whenever a TASK user message is received by the IOP with a msgType not used by UPA.

UPA currently uses only message type 80.

Syntax

```
void upSetUserMsgHandler (
   void (*MsgHandlerFunc)(
       IN long src,
       IN long msgType,
       IN ulong msgSzW,
       IN long *pBuf
   )
);
```

Parameters

MsgHandlerFunc
A pointer to the message handler.

src The message handler, in this format:

Device | *DeviceNum*

Device Enter either TASK_IOP or TASK_DSP.

DeviceNum Enter the device number.

Examples:

To invoke a handler from the second IOP, use (TASK_IOP | 2) as the source.

To invode a handler from the third DSP, use (TASK_DSP | 3) as source.

msgType The message type. You can use any positive integer.

msgSzW The number of words in the message.

pBuf The buffer that contains the message.

Return values

None.

C HDLC driver library

This appendix describes the HDLC driver library.

When reading this file online, you can immediately view information about any topic by placing the mouse cursor over a test name and clicking.

For information about...	Go to this page...
Overview	147
Driver internals, data structures, and resources	147
Sample HDLC driver sequence	150
Function list	151
Functions	153
Type definitions	171
Structures	172

Overview

The HDLC Status and Control API is the interface to the HDLC driver.

The HDLC driver provides the software interface to the HDLC controller of the LAN/WAN option card. The driver works under the VxWorks Real-Time operating system. The driver is not a “true” VxWorks driver in the sense that it is not built as part of the kernel. It does however make use of VxWorks system calls and therefore requires the O/S.

The driver provides control and monitoring of the HDLC controller of the LAN/WAN option card. The controller is a Siemens Munich128X. This document assumes that the reader has some understanding of the operation of the device.

The driver consists of these main parts:

- ISR (Interrupt Service Routine)
- IST (VxWorks Interrupt Service Task)
- Status and control API

Driver internals, data structures, and resources

This section describes some internal workings of the driver and issues related to resources.

Unlike the T1/E1 and T8100 drivers, which provide only control and status functionality, the HDLC driver includes data path functionality. That is, it is also responsible for the transmission and reception of data to and from the user application. Because of this, the HDLC driver requires board resources well beyond

what is used by the T1/E1 and T8100 drivers. The resources of concern are memory, on-board PCI bandwidth and processor power. These topics are discussed in great detail in the Munich128X documentation and are summarized here to illustrate how the driver code relates to the hardware.

Data structures

The HDLC driver maintains numerous data structures required by the HDLC controller to transmit and receive data. Of greatest interest are the transmit and receive descriptors and buffers and the interrupt queues.

- **Control Configuration Block (CCB)**: The primary data structure for the controller, the CCB is a relatively small block of data which contains setup information for the controller as well a table of pointers to transmit and receive descriptors. Each port and channel has one or more associated transmit and receive descriptors. The driver currently implements 32 descriptors per channel. Therefore with a maximum of 4 ports and 32 channels per port, there are a total of 4 * 32 * 32 = 4096 descriptors. The 32 descriptors per channel are configured as a circular chain. Each descriptor points to a data buffer as well as the next link in the chain. The data buffers are currently set to 512 bytes each. 512 was selected since it is sufficient to store the largest LAPD message.
- **Interrupt queues**: Another important data structure, the interrupt queues—one for transmit and one for receive—are used by the controller to store information on transmit and receive events. Each time a complete packet is sent or received or a transmit or receive error is detected, an entry or entries are written to the queue. These entries are later processed by the driver based on the mode (see mode description below).

Processing modes

Transmission and reception of packets requires various processing to be performed by the I/O processor (IOP). The following paragraphs describe the processing flow for transmission and reception as well as some of the build options (#define's) which are available to modify the driver operation to tailor it to the user's application. There are four modes in which the driver can work. These modes differ in the number of interrupts which are generated and the way in which Tx and Rx buffers are processed. Three of the four modes use polling for transmit, receive or both. Polling in this case refers to timer based polling in which an interrupt is generated by the Munich128X timer, and the transmit and receive descriptors are read (polled) in response to this timer interrupt. The interrupt/timer interval is a build option controlled by the HDLC_POLLING_INTERVAL #define.

- **Mode 1 (Fully interrupt driven)**: In this mode interrupts are generated for every packet which is transmitted and received. For applications which transmit many packets per second over many channels this mode results in high interrupt overhead and should not be used.
- **Mode 2 (Mixed mode 1—Tx polled)**: In this mode, packet transmission does not result in any interrupts but receive packets continue to generate interrupts. This mode generates approximately half the interrupts as compared to the first mode and still maintains very low latency on receive packets. It should be noted

that polling for completed packet transmission does NOT increase transmission latency. It does however increase the time before which the buffer for a transmitted packet is available for reuse by another packet. This gives the appearance of reducing the number of transmit buffers.

- **Mode 3** (**Mixed mode 2—Rx polled**): This mode is the opposite of the second mode. In this mode, interrupts are generated for transmitted packets while polling is used for receive packets.
- **Mode 4** (**Fully polled**): In this mode, no interrupts are generated as a result of transmitted or received packets. Timer based polling is used to determine when a packet has been transmitted and when one has been received. For applications which process many packets per second, this mode provides the least amount of processing overhead, however, the application must be able to tolerate the latency on receive packets. If this latency cannot be tolerated, Mode 2 should be used.

Processing packet transmission and reception

The descriptions below assume that the device, it ports and its channels are properly configured via the appropriate API calls.

Transmission of data begins with a call to the HDLCSendPacket function. This function takes the user data and copies it to the next available buffer associated with the next available descriptor for the port and channel being specified. Once the data has been copied to the buffer, it is marked as available for transmission. If the HDLC controller is idle, this wakes up the controller and begins transmission of the packet. If the controller is working on other links in the chain, marking the buffer ensures that the newly filled link gets transmitted after the others are completed. The HDLCSendPacket function returns immediately after the data is copied and the buffer is marked. It does NOT wait for transmission to begin.

When a packet has been completely transmitted, an interrupt is generated if the driver has been configured for Mode 1 or Mode 3 operation. As described previously, this interrupt results in a message to the driver task which picks up the message and processes the interrupt event. In this case, the processing involves reading the transmit interrupt queue, checking for any transmission errors and freeing the descriptor and buffer which held the packet that was transmitted. If Mode 2 or Mode 4 is used, no transmit interrupt is generated and the descriptor and buffer remains unavailable until the timer interrupt. When the timer interrupt occurs, the transmit interrupt queue is parsed and all descriptors and buffers associated with transmitted packets are freed. Note that although no per-packet transmit interrupt is generated for Modes 2 and 4, the controller continues to make entries in the interrupt queue for each packet.

During transmit packet processing, if the user has registered transmit handlers, the user is called back on two events. These events are a transmit error condition and/or a complete error-free packet transmission. Although not required, the later callback can be used to send another packet to the driver. Note that these callbacks occur in the context of the interrupt service task.

Once fully configured (device, port and channel), the controller constantly monitors the receive data stream awaiting a packet. When a packet is completely received, an

interrupt is generated if the driver was configured for Mode 1 or Mode 2 operation. As described previously, this interrupt with result in a message to the driver task which picks up the message and processes the interrupt event. In this case, the processing involves reading the receive interrupt queue, checking for any receive errors and marking the descriptor and buffer as having valid receive data. If Mode 3 or Mode 4 is used, no receive interrupt is generated and the descriptor and buffer remains unmarked until the timer interrupt. When the timer interrupt occurs, the receive interrupt queue is parsed and all descriptors and buffers associated with received packets are marked.

During receive packet processing, if the user has registered receive handlers, the user is be called back on two events. These events are a receive error condition and/or a complete error-free packet reception. In the error-free case, the user may call the HDLCGetPacket function to retrieve the received packet or as an alternative, the user can signal another task to read the packet. The user may also use a polling method to detect packet reception. In this case, the user can periodically poll for received packets using HDLCGetPacket. The function returns an indication if no packet is available, or copies the data to the user's space if a packet is available. HDLCGetPacket does not block when no packet is available. As in the interrupt versus polling case for the driver, polling for packet reception in this manner introduces additional latency. In either case, when HDLCGetPacket returns a valid packet, it copies the data to the user's space and frees the receive descriptor and buffer. As in the transmit case, the callbacks occur in the interrupt task context.

Since the HDLC controller user I/O processor memory for all its structures, memory bus and PCI bandwidth is also a consideration. This topic is discussed in detail in the device datasheets and user's manuals.

Sample HDLC driver sequence

Initialize the device and driver:

```
HDLCInit()
```

Set device-wide event handler:

```
HDLCSetDeviceErrorHandler(…….)
```

Setup port-wide event handlers.

```
HDLCSetTxPacketHandler(…….)
HDLCSetRxPacketHandler(…….)
HDLCSetTxErrorHandler(…….)
HDLCSetRxErrorHandler(…….)
```

Configure the port(s). Called for each port to be used.

```
HDLCConfigPort(…..…)
```

Configure the channel(s). Called for each channel of each port to be used.

```
HDLCConfigChannel(…..…)
```

Enable the channel(s). Called for each channel of each port to be used.

```
HDLCEnableChannel(…..…)
```

Channels are now enabled for transmission and reception of packets.

Transmit packets using:

```
HDLCSendPacket(…….)
```

Receive packets using:

```
HDLCGetPacket(…….)
```

This function may be used to poll for receipt of a packet or may be called in the user supplied callback function.

Close the driver before exiting the application

```
HDLCCloseDriver()
```

Function list

This appendix lists function descriptions alphabetically. You can find a function in this appendix either by locating it alphabetically within its group, or by using Table C-1 below, which groups like functions together.

Use this table to identify the HDLC driver functions you want to use. Use the function description later in this appendix to obtain detailed information, including syntax and parameter values.

Table C-1. HDLC driver functions

Type	Function	Description
Configuration		Configures each port and channel of the HDLC controller.
	HDLCInit	Initializes the HDLC hardware and software.
	HDLCReset	Resets the device and driver.
	HDLCCloseDriver	Shuts down the HDLC controller and device driver.
	HDLCClosePort	Shuts down an HDLC controller port.
	HDLCConfigPort	Specifies the configuration of any HDLC controller port, primarily mapping a port's timeslots to data channels.
	HDLCConfigChannel	Specifies the configuration of any channel of any HDLC controller port.
	HDLCEnableChannel	Enables a channel for operation.
	HDLCDisableChannel	Disables all status (interrupts and data) received from the specified channel.
	HDLCResetChannel	Resets the specified channel.
Packet transmission and reception		Queues packets for transmission and to read received packets.
	HDLCSendPacket	Sends an HDLC packet on the specified channel.
	HDLCGetPacket	Gets an HDLC packet from the driver.
Status monitoring		Polls the status of the controller and of each channel.

Table C-1. HDLC driver functions

Type	Function	Description
	HDLCGetDeviceStatus	Returns the status of the HDLC device and driver. **Note**: This function is not currently implemented.
	HDLCGetChannelStatus	Returns the status of the specified channel of the specified port. **Note**: This function is not currently implemented.
Interrupt/event control		Specifies handlers (callback functions) to call when an event occurs.
	HDLCSetTxPacketHandler	Specifies the user handler to call when packet transmission is complete.
	HDLCSetRxPacketHandler	Specifies the user handler to call when packet reception is complete.
	HDLCSetDeviceErrorHandler	Specifies the user handler to call when a device error is detected.
	HDLCSetTxErrorHandler	Specifies the user handler to call when a transmission error occurs.
	HDLCSetRxErrorHandler	Specifies the user handler to call when a message reception error occurs.

Function prototypes and type definitions for the interface are contained in this header file:

```
iop_to_hdlc.h
```

This document uses the term “Port” to refer to any one of the four TDM streams (ports) of the Siemens Munich128X HDLC controller. Ports are numbered 0 to 3. Each port consists of 32 64Kbps timeslots which are assigned to up to 32 “Channels”. The term “Channel” refers to logical (virtual) data channels of the HDLC controller. A channel consists of one or more bits from one or more TDM timeslots on a port. Therefore, each channel can have a data rate from 8Kbps to 2048Kbps. Each port is independent, so channels and timeslots of one port have no relationship or connection to channels and timeslots of another port. No two channels can share data from the same timeslot.

Functions

HDLCInit

Initializes the HDLC hardware and software.

Call this function before using any other HDLC driver functions.

Syntax

```
UINT32 HDLCInit(void)
```

Parameters

None.

Return values

HDLC_INIT_COMPLETE_32X
: The device is 32X; initialization succeeded.

HDLC_INIT_COMPLETE_128X
: The device is 128X; initialization succeeded.

HDLC_INIT_FAILURE
: An unspecified failure occurred.

HDLC_INIT_FAILURE_NO_DEV
: No device was found.

HDLC_INIT_INTR_FAILURE
: The device interrupt failed.

HDLC_INIT_CONFIG_FAILURE
: Reserved for future use.

HDLC_INIT_INTR_CONNECT_FAIL
: Failed to set the VxWorks interrupt handler.

HDLC_INIT_RESET_FAIL
: Cannot reset the device.

HDLCReset

Resets the device and driver.

Syntax

```
UINT32 HDLCReset(
   IN const UINT32 reset_type
)
```

Parameters

reset_type

Specifies the type of reset. You can use one of these values:

HDLC_HARD_RESET
: Resets the device (equivalent to a hardware reset).

HDLC_SOFT_RESET
: Resets only the driver.

In the current driver implementation, both reset types result in a hard reset.

Return values

SUCCESS Successful completion.

FAILURE Unsuccessful completion.

HDLCCloseDriver

Shuts down the HDLC controller and device driver.

Call this function prior to exiting the user application to properly quiesce the device.

Syntax

```
UINT32 HDLCCloseDriver(void)
```

Parameters

None.

Return values

SUCCESS	Successful completion.
FAILURE	Unsuccessful completion.

HDLCClosePort

Shuts down an HDLC controller port.

If multiple ports are in use, this function can shut down a port that generates excessive errors while keeping the remaining ports active.

This function is currently not implemented.

Syntax

```
UINT32 HDLCClosePort(IN const UINT port)
```

Parameters

port	The HDLC controller port.

Return values

SUCCESS	Successful completion.
FAILURE	Unsuccessful completion.

HDLCConfigPort

Specifies the configuration of any HDLC controller port, primarily mapping a port's timeslots to data channels.

Syntax

```
UINT32 HDLCConfigPort(
    IN const UINT32 port,
    IN const t_HDLC_port_config *port_config
)
```

Parameters

port The HDLC controller port.

**port_config*
For a detailed description of configuration options, see the data type section of the t_HDLC_port_config structure description.

Return values

SUCCESS Successful completion.

FAILURE Unsuccessful completion.

HDLCConfigChannel

Specifies the configuration of any channel of any HDLC controller port.

Configuration includes parameters such as interframe time-fill character, CRC type, enable/disabling CRC checking and channel loopbacks.

The current implementation of the driver does not support loopbacks.

Syntax

```
UINT32 HDLCConfigChannel(
   IN const UINT32 port,
   IN const UINT32 channel,
   IN const t_HDLC_channel_config *channel_config
)
```

Parameters

port The HDLC controller port.

channel The HDLC channel.

channel_config
For a detailed description of configuration options, see the data type section of the t_HDLC_channel_config structure description.

Return values

SUCCESS Successful completion.

FAILURE Unsuccessful completion.

HDLCEnableChannel

Enables a channel for operation.

The port and channel must be already configured. Once enabled, a channel can receive and accept packets for transmission.

Syntax

```
UINT32 HDLCEnableChannel(
   IN const UINT32 port,
   IN const UINT32 channel
)
```

Parameters

port	The HDLC controller port.
channel	The HDLC channel.

Outputs

Return values

SUCCESS	Successful completion.
FAILURE	Unsuccessful completion.

HDLCDisableChannel

Disables all status (interrupts and data) received from the specified channel.

Syntax

```
UINT32 HDLCDisableChannel(
    IN const UINT32 port,
    IN const UINT32 channel
)
```

Parameters

port	The HDLC controller port.
channel	The HDLC channel.

Return values

SUCCESS	Successful completion.
FAILURE	Unsuccessful completion.

HDLCResetChannel

Resets the specified channel.

A soft reset occurs on the channel, and the channel is disabled. Channel configuration resets and all allocated memory for that channel is deallocated within the driver.

Syntax

```
UINT32 HDLCResetChannel(
    IN const UINT32 port,
    IN const UINT32 channel
)
```

Parameters

port	The HDLC controller port.
channel	The HDLC channel.

Return values

SUCCESS	Successful completion.
FAILURE	Unsuccessful completion.

HDLCSendPacket

Sends an HDLC packet on the specified channel.

Syntax

```
UINT32 HDLCSendPacket(
    IN const UINT32 port,
    IN const UINT32 channel,
    IN const UINT8 *data_ptr,
    IN const UINT32 length
)
```

Parameters

port	The HDLC controller port.
channel	The HDLC channel.
data_ptr	A pointer to the data.
length	The number of bytes in the data. The current maximum packet length is 512 bytes.

Return values

SUCCESS	Successful completion.
FAILURE	Returns an error if the driver does not have sufficient memory to queue the message or if an invalid channel or port is specified.

HDLCGetPacket

Gets an HDLC packet from the driver.

The user was notified of the packet arrival by specifying a handler for received packets via HDLCSetRxPacketHandler. You can also use this function to poll for receipt of packet in which case the function returns a HDLC_NO_RX_DATA indication if no packet is available. The function does *not* wait for a packet, but instead returns immediately when no packet is available. This function returns only valid packets, that is, packets received without any errors (such as CRC).

Syntax

```
UINT32 HDLCGetPacket(
    IN const UINT32 port,
    IN const UINT32 channel,
    IN const UINT8 *data_ptr,
    OUT UINT32 *length
);
```

Parameters

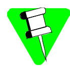
port	The HDLC controller port.
channel	The HDLC channel.
data_ptr	A pointer to the buffer from which to receive data.
length	The number of bytes in the buffer.

Return values

SUCCESS	Successful completion.
FAILURE	Unsuccessful completion.

HDLCGetDeviceStatus

Returns the status of the HDLC device and driver.



This function is currently not implemented.

Syntax

```
UINT32 HDCLGetDeviceStatus(void *status_ptr)
```

Parameters

status_ptr
No information available.

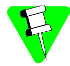
Return values

SUCCESS Successful completion.

FAILURE Unsuccessful completion.

HDLCGetChannelStatus

Returns the status of the specified channel of the specified port.



This function is currently not implemented.

Syntax

```
UINT32 HDLCGetChannelStatus(
    IN const UINT32 port,
    IN const UINT32 channel,
    OUT t_HDLC_channel_status *status_ptr
)
```

Parameters

port The HDLC controller port.

channel The HDLC channel.

status_ptr
A pointer to the t_HDLC_channel_status structure. For details about this structure, see *t_HDLC_channel_status* on page 176.

Return values

SUCCESS Successful completion.

FAILURE Unsuccessful completion.

HDLCSetTxPacketHandler

Specifies the user handler to call when packet transmission is complete.

Although not required, this can be used by the user code to send the next message. The current version of the driver queues up to 32 packets per channel. To disable this callback, the function can be called with a NULL function pointer.

Syntax

```
UINT32 HDLCSetTxPacketHandler(
    IN const UINT32 port,
    IN void (*UserHDLCTxPacketHandler) (UINT32 channel, UINT32 status)
)
```

Parameters

port The HDLC controller port.

UserHDLCTxPacketHandler
The user function that handles the event of packet transmission completion.

channel The HDLC channel.

status The HDLC transmission status.

Return values

SUCCESS Successful completion.

FAILURE Unsuccessful completion.

HDLCSetRxPacketHandler

Specifies the user handler to call when packet reception is complete.

When the driver performs the callback, the user may then call HDLCGetPacket to retrieve the data. Alternately the user may choose to poll for packet reception via repeated calls to HDLCGetPacket, or set a flag in the handler and read the packet at a later time. It is up to the user code to service the packet reception in a timely manner as to prevent overruns. The driver queues up to 32 received packets per channel. To disable this callback, the function can be called with a NULL function pointer.

Syntax

```
UINT32 HDLCSetRxPacketHandler(
    IN const UINT32 port,
    IN void (*UserHDLCRxPacketHandler) (UINT32 channel, UINT32 status)
)
```

Parameters

port The HDLC controller port.

UserHDLCRxPacketHandler
The user function that handles the event of packet receive completion.

channel The HDLC channel.

status The HDLC transmission status.

Return values

SUCCESS Successful completion.

FAILURE Unsuccessful completion.

HDLCSetDeviceErrorHandler

Specifies the user handler to call when a device error is detected.

Device errors can occur on events such are loss of frame sync or the device not being able to read IOP (i960) memory in a timely manner. To disable this callback, the function can be called with a NULL function pointer.

Syntax

```
UINT32 HDLCSetDeviceErrorHandler(
    IN const UINT32 port,
    IN void (*UserHDLCDeviceErrorHandler) (UINT32 error_code)
)
```

Parameters

port The HDLC controller port.

UserHDLCDeviceErrorHandler
The user function that handles device errors.

error_code
The HDLC error code.

Return values

SUCCESS Successful completion.

FAILURE Unsuccessful completion.

HDLCSetTxErrorHandler

Specifies the user handler to call when a transmission error occurs.

To disable this callback, the function can be called with a NULL function pointer.

Syntax

```
UINT32 HDLCSetTxErrorHandler(
    IN const UINT32 port,
    IN void (*UserHDLCTxErrorHandler) (UINT32 channel, UINT32 status)
)
```

Parameters

port The HDLC controller port.

UserHDLCTxErrorHandler
The user function that handles transmission errors.

channel The HDLC channel.

status The HDLC channel status.

Return values

SUCCESS Successful completion.

FAILURE Unsuccessful completion.

HDLCSetRxErrorHandler

Specifies the user handler to call when a message reception error occurs.

To disable this callback, the function can be called with a NULL function pointer.

Syntax

```
UINT32 HDLCSetRxErrorHandler(
    IN const UINT32 port,
    IN void (*UserHDLCRxErrorHandler) (UINT32 channel, UINT32 status)
)
```

Parameters

port The HDLC controller port.

UserHDLCRxErrorHandler
The user function that handles receive errors.

channel The HDLC channel.

status The HDLC channel status.

Return values

SUCCESS Successful completion.

FAILURE Unsuccessful completion.

Type definitions

(from IOP_TO_HDLC.H)

```
/* device limits */
#define HDLC_MAX_PORTS              4
#define HDLC_MAX_CHANNELS_PER_PORT 32
#define HDLC_MAX_TIMESLOTS_PER_PORT32

/* reset types */
#define HDLC_HARD_RESET             1
#define HDLC_SOFT_RESET             2

/* channel specifications */
#define HDLC_CHANNEL_NORMAL_MODE    1
#define HDLC_CHANNEL_LOOPBACK_INT   2
#define HDLC_CHANNEL_LOOPBACK_EXT   3

#define HDLC_IFTF_IS_7E0
#define HDLC_IFTF_IS_FF1

#define HDLC_FLAG_ADJUST_OFF        0 /* see Munich datasheet for */
#define HDLC_FLAG_ADJUST_ON         1 /* description of this mode */

#define HDLC_CRC_TYPE_16_BIT        0
#define HDLC_CRC_TYPE_32_BIT        1

#define HDLC_CRC_ENABLE             0
#define HDLC_CRC_DISABLE            1

#define HDLC_DONT_INVERT_DATA       0
#define HDLC_INVERT_DATA            1

/* code returned to device error handler */
#define HDLC_CMD_ERROR_NO_ACK       1
#define HDLC_CMD_ERROR_ACK_FAIL     2
#define HDLC_TX_SYNC_ERROR          3
#define HDLC_RX_SYNC_ERROR          4
#define HDLC_UNKNOWN_ERROR          5
#define HDLC_TX_BUFFER_ERROR        6

/* initialization return values */
#define HDLC_INIT_COMPLETE_32X      1 /* device is 32X, init successful */
#define HDLC_INIT_COMPLETE_128X     2 /* device is 128X, init successful */
#define HDLC_INIT_FAILURE3
#define HDLC_INIT_FAILURE_NO_DEV    4
#define HDLC_INIT_INTR_FAILURE      5
#define HDLC_INIT_CONFIG_FAILURE    6
#define HDLC_INIT_INTR_CONNECT_FAIL7
#define HDLC_INIT_RESET_FAIL        8

/* return status from HDLCGetPacket */
#define HDLC_NO_RX_DATA             1
#define HDLC_RX_DATA_RETURNED       2

/* return status from HDLCSendPacket */
#define HDLC_TX_CHAN_NOT_CONFIGURED1
#define HDLC_NO_TX_BUFFER_AVAIL     2
#define HDLC_PACKET_QUEUED          3

/* status word for Rx error handler */
#define HDLC_RX_BUFFER_ERROR        1

/* used in xx_channel_assignment field to indicate unassign timeslots */
#define HDLC_UNASSIGNED_TIMESLOT    0xff
```

Structures

Use this table to identify the HDLC driver structures. Use the structure description later in this appendix to obtain detailed information, including syntax and parameter values.

Table C-2. HDLC structures

Call	Description
t_HDLC_port_config	Specifies a port's configuration.
t_HDLC_channel_config	Specifies the configuration for each channel.
t_HDLC_channel_status	Specifies a channel's status. **Note**: This structure is not currently implemented.

t_HDLC_port_config

Specifies a port's configuration.

Syntax

```
typedef struct {
    UINT8  tx_channel_assignment[HDLC_MAX_TIMESLOTS_PER_PORT];
    UINT8  tx_enabled_bits[HDLC_MAX_TIMESLOTS_PER_PORT];
    UINT8  rx_channel_assignment[HDLC_MAX_TIMESLOTS_PER_PORT];
    UINT8  rx_enabled_bits[HDLC_MAX_TIMESLOTS_PER_PORT];
    bool   update_channel_assignments;
    bool   enable_port_loopback;
    bool   enable_frame_length_check;
    UINT32 max_frame_length;
} t_HDLC_port_config;
```

Parameters

tx_channel_assignment
tx_enabled_bits

Specifies the timeslot-to-data channel mapping for the port. The element indices correspond with timeslot numbers. These fields are an array of length 32. *tx_channel_assignment* specifies the data channel to which a timeslot belongs, while *tx_enabled_bits* is a bitmask that determines which bits in a timeslot are enabled.

Examples

`tx_channel_assignment[5] = 7`
Indicates that timeslot 5 belongs to data channel 7.

`tx_channel_assignment[5] = 7`
`tx_enabled_bits[5] = 0xFF`
Indicates that all 8 bits in timeslot 5 belong to data channel 7 forming a 64Kbps data channel.

`tx_channel_assignment[5] = 7`
`tx_enabled_bits[5] = 0x03`
Indicates that the first 2 bits of timeslot 5 belong to data channel 7, thus forming a 16Kbps channel.

rx_channel_assignment
rx_enabled_bits

Specifies the timeslot-to-data channel mapping for the port. The element indices correspond to timeslot numbers. These fields are an array of length 32. *rx_channel_assignment* specifies the data channel to which a timeslot belongs, while *rx_enabled_bits* is a bitmask that determines which bits in a timeslot are enabled.

Examples

`rx_channel_assignment[5] = 7`
Indicates that timeslot 5 belongs to data channel 7.

`rx_channel_assignment[5] = 7`
`rx_enabled_bits[5] = 0xFF`
Indicates that all 8 bits in timeslot 5 belong to data channel 7 forming a 64Kbps data channel.

`rx_channel_assignment[5] = 7`
`rx_enabled_bits[5] = 0x03`
Indicates that the first 2 bits of timeslot 5 belong to data channel 7, thus forming a 16Kbps channel.

`update_channel_assignments`
Indicates whether to process this structure's first four fields (channel/timeslot assignments and enables). You can select one of these:

FALSE — The first four fields are ignored and the structure's remaining fields are processed.

TRUE — The first four fields are valid and processed along with the structure's other fields.

`enable_port_loopback`
Not available. The current release of the driver does not support loopbacks. Loopback functionality can be accomplished by controlling the T8100 Timeslot switch on the SPIRIT board, to which all HDLC controller ports are connected.

`enable_frame_length_check`
Determines whether to use maximum frame length checking.

`max_frame_length`
Determines the maximum length for maximum frame length checking. The maximum length should not exceed 1024 in this version of the driver.

t_HDLC_channel_config

Specifies the configuration for each channel.

Syntax

```
typedef struct {
    UINT8  channel_mode;
    UINT8  interframe_timefill_char;
    UINT8  crc_type;
    UINT8  crc_enable;
    UINT8  invert_data;
    UINT8  protocol;
    UINT8  flag_adjust;
} t_HDLC_channel_config;
```

Parameters

channel_mode
: Specifies whether a channel is in normal or loopback mode. This version of the driver only supports normal mode.

interframe_timefill_char
: Specifies the octet to use for interframe time-fill (IFTF). You can select one of these:

HDLC_IFTF_IS_7E (IFTF = 0x7E)

HDLC_IFTF_IS_FF (IFTF = 0xFF)

crc_type
: Specifies whether the size of the CRC. You can select one of these:

16 bits (HDLC_CRC_TYPE_16_BIT)

32 bits (HDLC_CRC_TYPE_32_BIT)

crc_enable
: Enables and disables CRC checking (Rx) and generation (Tx).

invert_data
: Inverts transmitted and received data.

protocol
: Specifies the type of formatting protocol to use. The current driver supports only HDLC formatting and therefore this field is ignored

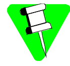
flag_adjust
: Adjusts the number of transmitted interframe time-fill characters. You can select one of these:

HDLC_FLAG_ADJUST_OFF
: Uses a minimum of three interframe time-fill characters between two consecutive frames.

HDLC_FLAG_ADJUST_ON
: Reduces the number of interface time-fill characters by 1/8 the number of zero insertions. See Munich128X datasheet for a complete description of this mode.

t_HDLC_channel_status

Specifies a channel's status.



This function is currently not implemented.

Syntax

```
typedef struct {
} t_HDLC_channel_status;
```

D T1/E1 library

This appendix describes the T1/E1 driver library.

When reading this file online, you can immediately view information about any topic by placing the mouse cursor over a test name and clicking.

For information about...	Go to this page...
Overview	177
Sample startup sequence	178
Function list	179
Functions	183
Structures	201

Overview

The T1/E1 Status and Control API is the user's interface to the T1/E1 driver.

The T1/E1 driver provides the software interface to the T1/E1 spans of the LAN/WAN option card. The driver is designed to work under the VxWorks Real-Time Operating System. The driver is not a "true" VxWorks driver in the sense that it is not built as part of the kernel. It does however make use of VxWorks system calls and therefore requires the O/S.

The driver provides control and monitoring of the two or four T1/E1 ports of the LAN/WAN option card. The ports use a Seimens Quad FALC LIU/Framer as their interface device. You do not need a working knowledge of this device to use T1/E1 functions.

The driver includes:

- Interrupt handling tools
- Function list

Sample startup sequence

```
T1E1initCard(&board_config);
```

Initializes the T1 card and driver. Not required if iopInit is being called.

```
setT1Config(&config_struct);
```

Sets the configuration of the T1 line.

```
setT1Signaling(&signaling_struct);
```

Sets the initial state of the transmitted signaling bits. Not required if signaling has not been enabled.

```
setT1SignalingHandler(&userT1SignalingHandler);
```

Sets a callback function for receive signaling bit changes.

```
setT1StatusHandler(&userT1StatusHandler);
```

Sets a callback function for receive line state/status changes.

Function list

This appendix lists function descriptions alphabetically. You can find a function in this appendix either by locating it alphabetically within its group, or by using Table D-1 below, which groups like functions together.

Use this table to identify the T1/E1 driver functions you want to use. Use the function description later in this appendix to obtain detailed information, including syntax and parameter values:

Table D-1. T1/E1 functions

Type	Function	Description
Configuration		Configure each port for T1 or E1 operation and configure the appropriate line parameters (line coding, framing mode, buildout, etc.). In addition, these functions allow for controlling of outbound robbed bit signaling, alarm generation, idle code insertion and loop-up/loop-down codes.
	T1E1initCard	Initializes the driver and initialize the framers on the option card.
	T1E1getBoardConfig	Gets information about the installed option card without performing the initialization done by T1E1initCard.
	T1E1setLeds	Controls the LEDs associated with each port of the card.
	setT1Config	Sets the card in T1 mode and sets the configuration of a specified port.
	setT1Signaling	Sets the values of the transmitted robbed bit signaling bits.
	setT1Command	Transmits “commands“ over the specified T1 port.
	setT1ClearChannels	Specifies which timeslots (DS0s) on the specified T1 port are clear channel (64Kbps) and which use robbed bit signaling and are therefore 56Kbps channels.
	setT1IdleChannels	Specifies which timeslots (DS0s) on the specified T1 port transmit the idle code and which transmit data “normally“.
	setT1ChannelConfig	Configures individual timeslots (DS0s) on the specified T1 port.
	setE1Config	Sets the card in E1 mode and set the configuration of a specified port.
	setE1Signaling	Sets the values of the transmitted CAS (channel associated signaling) bits.

Table D-1. T1/E1 functions

Type	Function	Description
Status monitoring		Polls the status of the framer and of each T1/E1 span. This status includes alarm conditions, loop code receipt and robbed bit signaling. These functions can also be used to collect status following the receipt of an interrupt event.
	getT1Signaling	Reads the values of the received robbed bit signaling bits on the specified T1 port.
	getT1Status	Reads the current receive and transmit status of a specified T1 port.
	getT1SignalingRaw	Reads the values of the received robbed bit signaling bits on the specified T1 port.
	getE1Signaling	Reads the values of the received CAS (channel associated signaling) bits on the specified E1 port.
Interrupt/event controls		Turns on and off alarm and signaling interrupt generation and allow the user to specify handlers (callback functions) to be called when the event occurs.
	setT1SignalingHandler	Specifies the event handler to call after detecting a change in receive T1 signaling state.
	setT1StatusHandler	Specifies the event handler to call after detecting a change in receive line state.
	setE1SignalingHandler	Specifies the event handler to call after detecting a change in receive E1 signaling state.

Function prototypes and type definitions for the driver interface are contained in the following header files.

- iop_to_t1.h
- iop_to_e1.h
- iop_to_t1e1.h
- t1e1_common_types.h

The functions and their data types are described in the pages that follow. The descriptions use the terms port, framer, span, line and DS1 interchangeably to indicate 1 of 4 T1/E1 connections available on the I/O board.

Functions which contain “T1E1” in their name are used for either T1 or E1 operation. Currently the driver does not support mixed operation on a single option card. Functions with only “T1” in their name are T1 specific while functions with only “E1” are E1 specific. Basic selection between T1 and E1 operating modes is made with the setT1Config or setE1Config API calls. These mutually exclusive

functions will configure the card for either T1 or E1 operation and must be called prior to calling other T1 or E1 specific functions.

Limitations

- **E1 support:** The API currently provides limited support for E1. Support is provided for span configuration and signaling generation and detection only. T1 support is much more comprehensive, however the T1 ESF Facility Data Link functionality is not supported.
- **Time slot assignments for T1 and E1 interfaces:** There are differences in the mapping of T8100 time slots depending on the interface used, as illustrated by the following table. The developer should be aware of these differences when assigning DSP data channel resources to time slots using the T8100 peripheral.

Table D-2. Time slot numbers

T8100	T1	E1
0	1 (data 0)	0 (signaling 0)
1	2 (data 1)	1 (data 0)
2	3 (data 2)	2 (data 1)
3	4 (data 3)	3 (data 2)
4	5 (data 4)	4 (data 3)
5	6 (data 5)	5 (data 4)
6	7 (data 6)	7 (data 6)
7	8 (data 7)	7 (data 6)
8	9 (data 8)	8 (data 7)
9	10 (data 9)	9 (data 8)
10	11 (data 10)	10 (data 9)
11	12 (data 11)	11 (data 10)
12	13 (data 12)	12 (data 11)
13	14 (data 13)	13 (data 12)
14	15 (data 14)	14 (data 13)
15	16 (data 15)	15 (data 14)
16	17 (data 16)	16 (signaling 1)
17	18 (data 17)	17 (data 15)
18	19 (data 18)	18 (data 16)
19	20 (data 19)	19 (data 17)
20	21 (data 20)	20 (data 18)
21	22 (data 21)	21 (data 19)
22	23 (data 22)	22 (data 20)
23	24 (data 23)	23 (data 21)
24	N/A	24 (data 22)
25	N/A	25 (data 23)

Table D-2. Time slot numbers

T8100	T1	E1
26	N/A	26 (data 24)
27	N/A	27 (data 25)
28	N/A	28 (data 26)
29	N/A	29 (data 27)
30	N/A	30 (data 28)
31	N/A	31 (data 29)

Functions

T1E1initCard

Initializes the driver and initialize the framers on the option card.

The function accepts a pointer to a structure and fills the structure with information about the installed card. You *must* call this function before using any other driver functions, however, the you do not need to call this function if iopInit is called, since iopInit also makes the call. As the name implies, this function is *not* specific to T1 or E1 operations.

Syntax

```
void T1E1initCard(
    t_T1E1_BoardConfig *board_config
)
```

Parameters

board_config

A pointer to the t_T1E1_BoardConfig structure. For more information about this structure, see *t_T1E1_BoardConfig on page 207*.

Return values

SUCCESS Successful completion.

FAILURE Unsuccessful completion.

T1E1getBoardConfig

Gets information about the installed option card without performing the initialization done by T1E1initCard.

Syntax

```
void T1E1getBoardConfig(
    t_T1E1_BoardConfig *board_config
)
```

Parameters

board_config
: A pointer to the t_T1E1_BoardConfig structure. For more information about this structure, see *t_T1E1_BoardConfig on page 207*.

Outputs

Return values

SUCCESS	Successful completion.
FAILURE	Unsuccessful completion.

T1E1setLeds

Controls the LEDs associated with each port of the card.

See the Type Definition section for a complete description of possible LED states.

Syntax

```
void T1E1setLeds(
    IN const t_T1E1_framer_id framer_id,
    IN const t_T1E1_led_state led_state
)
```

Parameters

framer_id	A pointer to the t_T1E1_framer_id structure. For more information about this structure, see *t_T1E1_framer_id on page 203*.
led_state	A pointer to the structure.

Outputs

Return values

SUCCESS	Successful completion.
FAILURE	Unsuccessful completion.

setT1Config

Sets the card in T1 mode and sets the configuration of a specified port.

You *must* call this function after calling T1E1initCard (for T1 operation).

Syntax

```
void setT1Config(
   IN const t_T1_user_config_struct *config_struct
)
```

Parameters

config_struct

A pointer to the t_T1_user_config_struct structure. For more information about this structure, see *t_T1_user_config_struct on page 211*.

Configuration includes:

- Line coding
- Framing mode
- Line buildout
- Idle channel and loopback specifications
- Enabling/disabling robbed bit signaling

Outputs

Return values

SUCCESS Successful completion.

FAILURE Unsuccessful completion.

setT1Signaling

Sets the values of the transmitted robbed bit signaling bits.

To transmit the bits, robbed bit signaling must be enabled via setT1Config. The *changed_bits* field of the signaling_struct determines which channel's (DS0's) bits are updated.

A 1 in Bit 0 of *changed_bits* corresponds to the first timeslot of the DS1 line. Bit 23 of *changed_bits* corresponds to the last timeslot of the DS1 line.

For details about each field, see the t_T1_user_signaling_data structure description.

Syntax

```
void setT1Signaling(
   IN const t_T1_user_signaling_data *signaling_struct
)
```

Parameters

signaling_struct

A pointer to the t_E1_user_signaling_data structure. For more information about this structure, see *t_T1_user_signaling_data t_T1_signaling_data on page 213.*

Outputs

Return values

SUCCESS Successful completion.

FAILURE Unsuccessful completion.

setT1Command

Transmits “commands” over the specified T1 port.

Syntax

```
void setT1Command(
    IN const t_T1_user_command_data *command_struct
)
```

Parameters

command_struct

A pointer to the t_T1_user_command_data structure. For more information about this structure, see *t_T1_user_command_data on page 214*

You can select one of these:

Transmit AIS

Transmit yellow (remote) alarm

Transmit loop up code

Transmit loop down code

Outputs

Return values

SUCCESS	Successful completion.
FAILURE	Unsuccessful completion.

setT1ClearChannels

Specifies which timeslots (DS0s) on the specified T1 port are clear channel (64Kbps) and which use robbed bit signaling and are therefore 56Kbps channels.

By default, channels are assigned to be non-clear channel (56Kbps). For any channels to be used for robbed bit signaling, the *robbed_bit_signaling_enable* field in the t_T1_user_config_struct configuration command sent via setT1Config must be set to TRUE. If it is set to FALSE, all channels are clear, regardless of the configuration specified through this function.

Syntax

```
void setT1ClearChannels(
   IN const t_T1_user_clear_channel_data clear_channels_struct
)
```

Parameters

clear_channels_struct

A pointer to the t_T1_user_clear_channel_data structure. For more information about this structure, see *t_T1_user_clear_channel_data on page 216*.

Outputs

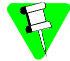
Return values

SUCCESS	Successful completion.
FAILURE	Unsuccessful completion.

setT1IdleChannels

Specifies which timeslots (DS0s) on the specified T1 port transmit the idle code and which transmit data “normally”.

By default, channels are assigned as non-idle. The function also specifies the 8-bit idle code.



Idle code data overwrites robbed bit signaling bits.

Syntax

```
void setT1IdleChannels(
   IN const t_T1_user_idle_struct *idle_channels_struct
)
```

Parameters

idle_channels_struct

A pointer to the t_T1_user_idle_struct structure. For more information about this structure, see *t_T1_user_idle_struct on page 217.*

Outputs

Return values

SUCCESS Successful completion.

FAILURE Unsuccessful completion.

setT1ChannelConfig

Configures individual timeslots (DS0s) on the specified T1 port.

You can specify these parameters:

- Insertion/removal of idle code
- Clear channel mode
- ABCD robbed bits (supervision)

Syntax

```
void setT1ChannelConfig(
    IN const t_T1_user_channel_config *channel_config
)
```

Parameters

channel_config

A pointer to the t_T1_user_channel_config structure. For more information about this structure, see *t_T1_user_channel_config on page 218*.

Outputs

Return values

SUCCESS	Successful completion.
FAILURE	Unsuccessful completion.

setE1Config

Sets the card in E1 mode and set the configuration of a specified port.

Configuration includes line coding, framing mode, line buildout, idle channel and loopback specifications and enabling/disabling channel associated signaling. This function *must* be called after calling T1E1initCard (for E1 operation).

Syntax

```
void setE1Config(
    IN const t_E1_user_config_struct *config_struct
)
```

Parameters

config_struct

A pointer to the t_E1_user_config_struct structure. For more information about this structure, see *t_E1_user_config_struct on page 226.*

Outputs

Return values

SUCCESS Successful completion.

FAILURE Unsuccessful completion.

setE1Signaling

Sets the values of the transmitted CAS (channel associated signaling) bits.

To transmit the bits, CAS must be enabled via setE1Config. The *changed_bits* field in signaling_struct determines which channel's (DS0's) bits to update. A 1 in Bit 0 of *changed_bits* corresponds to the first timeslot of the E1 line. Bit 31 of *changed_bits* corresponds to the last timeslot of the E1 line.

See t_E1_user_signaling_data structure description for details on each of the fields.

Syntax

```
void setE1Signaling(
    IN const t_E1_user_signaling_data *signaling_struct
)
```

Parameters

signaling_struct

A pointer to the t_E1_user_signaling_data structure. For more information about this structure, see *t_E1_user_signaling_data on page 228.*

Outputs

Return values

SUCCESS	Successful completion.
FAILURE	Unsuccessful completion.

getT1Signaling

Reads the values of the received robbed bit signaling bits on the specified T1 port.

To receive bits, robbed bit signaling must be enabled via setT1Config. When signaling is read in this fashion, the `changed_bits` and `timestamp` fields of the signaling_struct are invalid and not updated.

Syntax

```
void getT1Signaling(
    INOUT t_T1_user_signaling_data *signaling_struct
)
```

Parameters

signaling_struct

A pointer to the t_E1_user_signaling_data structure. For more information about this structure, see *t_T1_user_signaling_data t_T1_signaling_data on page 213*.

Outputs

Return values

SUCCESS Successful completion.

FAILURE Unsuccessful completion.

getT1Status

Reads the current receive and transmit status of a specified T1 port.

Status consists of the currently transmitted "commands" and the current receive conditions (alarms and errors).

Syntax

```
void getT1Status(
    INOUT t_T1_user_status_data *status_struct
)
```

Parameters

status_struct

A pointer to the t_T1_user_status_struct structure. For more information about this structure, see *t_T1_user_status_struct on page 215*.

Outputs

Return values

SUCCESS Successful completion.

FAILURE Unsuccessful completion.

getT1SignalingRaw

Reads the values of the received robbed bit signaling bits on the specified T1 port.

This function differs from the getT1Signaling function in the format in which data is presented to the user. This function is specifically designed for users who want to poll signaling bits rather than use an interrupt based method.

Syntax

```
void getT1SignalingRaw(
    INOUT t_T1_user_raw_signaling_struct *signaling_struct
)
```

Parameters

signaling_struct

A pointer to the t_T1_user_raw_signaling_struct structure. For more information about this structure, see *t_T1_user_raw_signaling_struct on page 220.*

Outputs

Return values

SUCCESS Successful completion.

FAILURE Unsuccessful completion.

getE1Signaling

Reads the values of the received CAS (channel associated signaling) bits on the specified E1 port.

To receive the bits, CAS must be enabled via setE1Config. When signaling is read in this fashion, the *changed_bits* and *timestamp* fields of *signaling_struct* are invalid and not updated.

Syntax

```
void getE1Signaling(
    INOUT t_E1_user_signaling_data *signaling_struct
)
```

Parameters

signaling_struct

A pointer to the t_E1_user_signaling_data structure. For more information about this structure, see *t_E1_user_signaling_data on page 228.*

Outputs

Return values

SUCCESS Successful completion.

FAILURE Unsuccessful completion.

setT1SignalingHandler

Specifies the event handler to call after detecting a change in receive T1 signaling state.

This function “hooks” in the user’s event handler and enables signaling event generation.

Syntax

```
void setT1SignalingHandler(
   T1SignalingHandler userT1SignalingHandler
)
```

Parameters

userT1SignalingHandler

You can select one of these:

Event handler name
: A user-specified signaling event handler. For more information about this event handler, see *T1SignalingHandler on page 222.*

NULL
: Disables status event generation.

Outputs

Return values

SUCCESS	Successful completion.
FAILURE	Unsuccessful completion.

setT1StatusHandler

Specifies the event handler to call after detecting a change in receive line state.

This function “hooks” in the user’s event handler and enables status event generation.

Syntax

```
void setT1StatusHandler(
   T1StatusHandler userT1StatusHandler
)
```

Parameters

userT1StatusHandler

A pointer to the structure. For more information about this structure, see *t_T1E1_BoardConfig on page 207*.

You can select one of these:

Event handler name

Passes a pointer to t_T1_user_status_data that indicates which framer had a state change or changes, which state bits changed value, the value of each state bit, and a timestamp that indicates when the change took place.

For more information about this event handler, see *T1StatusHandler on page 221*.

NULL — Disables status event generation.

Outputs

Return values

SUCCESS	Successful completion.
FAILURE	Unsuccessful completion.

Comments

Detected receive-line state changes include:

- AIS
- Yellow alarm
- LOF (Loss of frame sync)
- LOS (Loss of signal)

setE1SignalingHandler

Specifies the event handler to call after detecting a change in receive E1 signaling state.

This function "hooks" the user's event handler and enables signaling event.

Syntax

```
void setE1SignalingHandler(
    E1SignalingHandler userE1SignalingHandler
)
```

Parameters

userE1SignalingHandler

You can select one of these:

Event handler name
: A user-specified signaling event handler. For more information about this event handler, see *E1SignalingHandler* on page 229.

NULL
: Disables status event generation.

Outputs

Return values

SUCCESS Successful completion.

FAILURE Unsuccessful completion.

Structures

Use this table to identify the E1/T1 driver structures. Use the structure description later in this appendix to obtain detailed information, including syntax and parameter values.

Table D-3. E1/T1 structures

Type	Call	Description
Common (E1 and T1) definitions		
	t_T1E1_framer_id	Specifies a framer (port) for functions which can operate on any port.
	t_T1E1_card_type	Identifies the type of card installed.
	t_T1E1_led_state	Controls the LEDs for each port/span.
	t_T1E1_user_signaling_data	
	t_T1E1_BoardConfig	Indicates the installed card type.
T1 definitions		
	t_T1_line_coding	Specifies the line coding for a framer (port).
	t_T1_framing_mode	Specifies the line framing for a framer (port).
	t_T1_line_buildout	Specifies the line buildout for a framer (port).
	t_T1_user_config_struct	Specifies the complete configuration for a framer (port).
	t_T1_user_signaling_data t_T1_signaling_data	Reads and writes signaling data.
	t_T1_user_command_data	Sends alarm and loop up or down codes over the T1 line.
	t_T1_user_status_struct	Provides line status.
	t_T1_user_clear_channel_data	Indicates which channels are clear channels (64Kbps) and which are not (56Kbps).
	t_T1_user_idle_struct	Transmits an idle code on a selected DS0.
	t_T1_user_channel_config	Controls the configuration of a specified DS0 on a specified framer/port.
	t_T1_user_raw_signaling_struct	Reads the signaling bit (supervision bit) values for each channel on the specified T1 port.
E1 definitions		
	t_E1_line_coding	Enumerate the line coding for a framer (port).
	t_E1_signaling_mode	Enumerates the signaling mode for a framer (port).
	t_E1_line_buildout	Enumerates the line buildout for a framer (port).

Table D-3. E1/T1 structures

Type	Call	Description
	t_E1_user_config_struct	Specifies the complete configuration for a framer (port).
	t_E1_user_signaling_data	Reads and writes signaling data.

t_T1E1_framer_id

Specifies a framer (port) for functions which can operate on any port.

Syntax

```
typedef enum {
    framer_1 = 0,
    framer_2 = 1,
    framer_3 = 2,
    framer_4 = 3
} t_T1E1_framer_id;
```

Elements

framer_1 The first port on the framer.

framer_2 The second port on the framer.

framer_3 The third port on the framer.

framer_4 The fourth port on the framer.

t_T1E1_card_type

Identifies the type of card installed.

Syntax

```
typedef enum {
    E1,
    T1
} t_T1E1_card_type;
```

Elements

E1 The E1 card.

T1 The T1 card.

t_T1E1_led_state

Controls the LEDs for each port/span.

Each port has two associated LEDs. Each LED is bi-colored, with one LED either yellow or green, and the other either red or green.

Syntax

```
typedef enum {
    leds_off,
    leds_normal,
    leds_yellow_alarm,
    leds_red_alarm,
    leds_loopback,
    leds_line_fault,
    leds_gr_yellow,
    leds_red_gr,
    leds_off_gr,
} t_T1E1_led_state;
```

Elements

`leds_off` Sets both LEDs to off.

`leds_normal`
Sets one LED to off and sets the other to green.

`leds_yellow_alarm`
Sets one LED to off and sets the other to yellow.

`leds_red_alarm`
Sets one LED to off and sets the other to red.

`leds_loopback`
Sets both LEDs to green.

`leds_line_fault`
Sets one LED to red and sets the other to yellow.

`leds_gr_yellow`
Sets one LED to green and sets the other to yellow.

`leds_red_gr`
Sets one LED to red and sets the other to green.

`leds_off_gr`
Sets one LED to off and sets the other to green.

t_T1E1_user_signaling_data

A type definition for a function pointer that points to a user handler that will receive a t_T1E1_user_signaling_data structure pointer as a parameter.

Syntax

```
typedef void
    (*T1E1SignalingHandler)
    ( t_T1E1_user_signaling_data *signaling_struct);
```

Elements

T1E1SignalingHandler

A pointer to the T1/E1 signaling handler.

signaling_struct

A pointer to a t_T1E1_user_signaling_data structure.

t_T1E1_BoardConfig

Indicates the installed card type.

Calls to T1E1initCard and T1E1getBoardConfig fill in this structure.

Syntax

```
typedef struct {
   boolinit_passed;
   t_T1E1_card_typecard_type;
   ucharnumber_of_ports;
} t_T1E1_BoardConfig;
```

Elements

init_passed

Determines whether to send AIS. You can select one of these:

TRUE	Initialization was successful.
FALSE	The board was not intialized.

card_type

Identifies the type of card installed. For more information, see *t_T1E1_card_type* on page 204.

number_of_ports

The number of T1 or E1 ports on the board.

t_T1_line_coding

Specifies the line coding for a framer (port).

Syntax

```
typedef enum {
    AMI,
    B8ZS
} t_T1_line_coding;
```

Elements

AMI AMI (alternate mark inversion) line coding.

B8ZS B8ZS (binary 8 zeros substitution) line coding.

t_T1_framing_mode

Specifies the line framing for a framer (port).

Syntax

```
typedef enum {
    D4_SF,
    ESF
} t_T1_framing_mode;
```

Elements

D4_SF	D4 super frame.
ESF	Extended super frame.

t_T1_line_buildout

Specifies the line buildout for a framer (port).

Syntax

```
typedef enum {
    DSX1_0_to_133_ft,
    DSX1_133_to_266_ft,
    DSX1_266_to_399_ft,
    DSX1_399_to_533_ft,
    DSX1_533_to_655_ft,
    CSU_minus7p5_db,
    CSU_minus15_db,
    CSU_minus22p5_db
} t_T1_line_build_out;
```

Elements

DSX1 Sets line buildout to short-haul mode.

CSU Sets line buildout to long-haul mode.

t_T1_user_config_struct

Specifies the complete configuration for a framer (port).

Syntax

```
typedef struct {
    t_T1E1_framer_id     framer_id;
    t_T1_line_coding     line_coding;
    t_T1_framing_mode    framing_mode;
    t_T1_line_build_out  line_build_out;
    uchar                idle_code;
    ulong                idle_channels;
    BOOL32               payload_loopback_enable;
    BOOL32               framer_loopback_enable;
    BOOL32               local_loopback_enable;
    BOOL32               remote_loopback_enable;
    BOOL32               robbed_bit_signaling_enable;
} t_T1_user_config_struct;
```

Elements

`framer_id` Specifies a framer (port) for functions which can operate on any port. For more information, see *t_T1E1_framer_id on page 203*.

`line_coding`
: Specifies the line coding for a framer (port). For more information, see *t_T1_line_coding* on page 208.

`framing_mode`
: Specifies the line framing for a framer (port). For more information, see *t_T1_framing_mode* on page 209.

`line_build_out`
: Specifies the line buildout for a framer (port). For more information, see *t_T1_line_buildout* on page 210.

`idle_code` Specifies the pattern to inject into transmitted data.

`idle_channels`
: The lower 24 bits specify the channels to inject the pattern specified in `idle_code` into.

: A 1 in any bit indicates that the pattern is to be injected. Bit 0 is associated with the first DS0 of the DS1 line while bit 23 is associated with the last DS0.

`payload_loopback_enable`
: Instructs the framer to perform a line-side loopback. Setting any field to TRUE enables the loopback.

`framer_loopback_enable`
: Instructs the framer to perform a system-side loopback. Setting any field to TRUE enables the loopback.

local_loopback_enable
Instructs the framer to perform a system-side loopback. Setting any field to TRUE enables the loopback.

remote_loopback_enable
Instructs the framer to perform a line-side loopback. Setting any field to TRUE enables the loopback.

robbed_bit_signaling_enable
Turns on and off the robbed bit signaling. You can select one of these:

Disabled The T1 line operates in clear channel mode.

Enabled Robbed bit signaling bits are inserted into and extracted from the T1 data stream.

t_T1_user_signaling_data
t_T1_signaling_data

Reads and writes signaling data.

The first five fields of t_T1_signaling_data are bit masks where the 24 low-order bits represent the 24 DS0s of a DS1 line. These bits either provide status of (setT1Signaling or user callback) or allow control over the signaling bits (setT1Signaling). Bit 0 is used for timeslot 1, Bit 1 for timeslot 2 and so on to bit 23 which is used for timeslot 24.

Syntax

```
typedef struct {
    t_T1E1_framer_idframer_id;
    t_T1_signaling_datasignaling_data;
} t_T1_user_signaling_data;
```

```
typedef struct {
    ulong changed_bits;
    ulong a_bits;
    ulong b_bits;
    ulong c_bits;
    ulong d_bits;
    ulong timestamp;
} t_T1_signaling_data;
```

Elements

framer_id Specifies a framer (port) for functions which can operate on any port. For more information, see *t_T1E1_framer_id on page 203*.

signaling_data
Reads and writes signaling data. For more information, see *t_T1_user_signaling_data t_T1_signaling_data* on page 213.

changed_bits
Indicates which bits have changed or need to change state. When setting signaling bits, this mask indicates which bits should be updated. When reading signaling bits, this mask indicates which bits have changed state since the previous change. Any bit set to 1 indicates that a change has taken (or needs to take) place.

a_bits Rrepresents the 24 A signaling bits of the DS1 line.

b_bits Represents the 24 B signaling bits of the DS1 line.

c_bits Represents the 24 C signaling bits of the DS1 line. If the line is configured for D4/SF framing, these bits are not used.

d_bits Represents the 24 D signaling bits of the DS1 line. If the line is configured for D4/SF framing, these bits are not used.

timestamp A 1µS granularity (LSB=1 µS) timestamp that indicates when the signaling change took place. This field is useful when forwarding signaling information to other systems which may be interpreting signaling bit changes. The rollover value for this field is 0x07C1F080. This field is not used when setting signaling bits.

t_T1_user_command_data

Sends alarm and loop up or down codes over the T1 line.

Setting any field to TRUE sends the alarm or code. These fields are mutually exclusive.

Syntax

```
typedef struct {
    t_T1E1_framer_idframer_id;
    BOOL32send_AIS;
    BOOL32send_yellow_alarm;
    BOOL32send_loop_up;
    BOOL32send_loop_down;
} t_T1_user_command_data;
```

Elements

framer_id Specifies a framer (port) for functions which can operate on any port. For more information, see *t_T1E1_framer_id on page 203*.

send_AIS Determines whether to send AIS. You can select one of these:

TRUE Sends an AIS (unframed all 1s signal) alarm.

FALSE Does not send and AIS alarm.

send_yellow_alarm

Determines whether to send yellow alarms. You can select one of these:

TRUE Sends yellow alarms.

FALSE Does not send yellow alarms.

send_loop_up

Determines whether to send loop up. You can select one of these:

TRUE Sends loop up.

FALSE Does not send loop up.

send_loop_down

Determines whether to send loop down. You can select one of these:

TRUE Sends loop down.

FALSE Does not send loop down.

t_T1_user_status_struct

Provides line status.

Status is provided via one of these methods:

- Call getT1Status: The *framer_id* is specified by the caller and the *timestamp* and *change_word* fields are invalid.
- Use a callback function specified through setT1StatusHandler: A change in receive line status results in a callback to the user specified handler. In this case all four fields of t_T1_user_status_struct are valid.

Syntax

```
typedef struct {
   t_T1E1_framer_idframer_id;
   ulong state_word;
   ulong change_word;
   ulong timestamp;
} t_T1_user_status_data;
```

Elements

framer_id Identifies the framer that changed state.

state_word
A bit mask which indicates the current transmit commands and receive line status. It indicates the current line state (same as in the “get” case). The bit definitions for this field are contained in iop_to_t1.h.

change_word
A bit mask which indicates which bit (or bits) in *state_word* caused the event to occur.

timestamp A 1 µS granularity time value indicating when the state change took place.

t_T1_user_clear_channel_data

Indicates which channels are clear channels (64Kbps) and which are not (56Kbps).

Syntax

```
typedef struct {
    t_T1E1_framer_idframer_id;
    ulong clear_channels;
} t_T1_user_clear_channel_data;
```

Elements

framer_id Specifies the framer or port to configure.

clear_channels
A bit mask with the 24 lower bits representing the 24 DS0s.

1 (in any bit) configures that DS0 for clear channel operation.

0 (in any bit) configure that DS0 for non-clear channel.

Bit 0 is associated with the first DS0 of the DS1 line while bit 23 is associated with the last DS0.

t_T1_user_idle_struct

Transmits an idle code on a selected DS0.

Syntax

```
typedef struct {
    t_T1E1_framer_idframer_id;
    uchar idle_code;
    ulongidle_channels;
} t_T1_user_idle_struct;
```

Elements

framer_id Specifies the framer or port to configure.

idle_code An 8-bit value which represents the idle code to transmit.

idle_channels
A bit mask where the 24 lower bits represent the 24 DS0s.

1 (in any bit) configures that DS0 to transmit the *idle_code*.

0 (in any bit) configures that DS0 to transmit data received from the on-board TDM switch.

Bit 0 is associated with the first DS0 of the DS1 line while bit 23 is associated with the last DS0.

t_T1_user_channel_config

Controls the configuration of a specified DS0 on a specified framer/port.

Syntax

```
typedef struct {
    t_T1E1_framer_idframer_id;
    ulongchannel;
    BOOL32enable_idle_code;
    BOOL32disable_idle_code;
    BOOL32enable_clear_channel;
    BOOL32disable_clear_channel;
    BOOL32update_supervision;
    BOOL32a_signaling_bit;
    BOOL32b_signaling_bit;
    BOOL32c_signaling_bit;
    BOOL32d_signaling_bit;
} t_T1_user_channel_config;
```

Elements

framer_id Specifies the framer/port to configure.

channel Specifies the DS0 to configure (zero based).

enable_idle_code

You can select one of these:

TRUE Enables transmission of the idle code on the selected channel.

Do not set both this field and *disable_idle_code* to this value.

FALSE If both this field and *disable_idle_code* contain this value, transmission remains at its previous state.

disable_idle_code

You can select one of these:

TRUE Disables transmission of the idle code on the selected channel.

Do not set both this field and *enable_idle_code* to this value.

FALSE If both this field and *enable_idle_code* contain this value, transmission remains at its previous state.

enable_clear_channel

You can select one of these:

TRUE Configures the channel for eith 64Kbps (clear channel) or 56Kbps (non clear channel).

Do not set both this field and *disable_clear_channel* to this value.

FALSE If both this field and *disable_clear_channel* contain this value, the channel's clear/not-clear configuration remains at its previous state.

disable_clear_channel

You can select one of these:

TRUE Configures the channel for eith 64Kbps (clear channel) or 56Kbps (non clear channel).

Do not set both this field and *enable_clear_channel* to this value.

FALSE If both this field and *disable_clear_channel* contain this value, the channel's clear/not-clear configuration remains at its previous state.

update_supervision

You can select one of these:

TRUE Specifies the transmitted robbed bit signaling bits (supervision bits) by changing the ABCD bit values specified in *x_signaling_bit*.

Do not set both this field and *enable_clear_channel* to this value.

FALSE Supervision for the selected channel remains unchanged.

x_signaling_bit

You can select one of these:

TRUE Sets the signaling bit to 1.

Do not set both this field and *enable_clear_channel* to this value.

FALSE Sets the signaling bit to 0.

t_T1_user_raw_signaling_struct

Reads the signaling bit (supervision bit) values for each channel on the specified T1 port.

Use in conjunction with the getT1SignalingRaw function.

Syntax

```
typedef struct {
   t_T1E1_framer_idframer_id;
   ucharsignaling_bits[12];
} t_T1_user_raw_signaling_struct;
```

Elements

framer_id

Specifies which framer or port to read.

signaling_bits

An array of 12 bytes, each of which contain the signaling bits for two DS0s.

Example

The high-order nibble of each byte contains the signaling bits for the lower numbered channel. For example:

```
signaling_bits[0] bit 7 = DS0-0 A bit
signaling_bits[0] bit 6 = DS0-0 B bit
signaling_bits[0] bit 5 = DS0-0 C bit
signaling_bits[0] bit 4 = DS0-0 D bit
signaling_bits[0] bit 3 = DS0-1 A bit
signaling_bits[0] bit 2 = DS0-1 B bit
signaling_bits[0] bit 1 = DS0-1 C bit
signaling_bits[0] bit 0 = DS0-1 D bit
signaling_bits[11] bit 7 = DS0-22 A bit
signaling_bits[11] bit 6 = DS0-22 B bit
signaling_bits[11] bit 5 = DS0-22 C bit
signaling_bits[11] bit 4 = DS0-22 D bit
signaling_bits[11] bit 3 = DS0-23 A bit
signaling_bits[11] bit 2 = DS0-23 B bit
signaling_bits[11] bit 1 = DS0-23 C bit
signaling_bits[11] bit 0 = DS0-23 D bit
```

T1StatusHandler

A type definition for a function pointer that points to a user handler that will receive a t_T1_user_status_data structure pointer as a parameter.

Syntax

```
typedef void
   (*T1StatusHandler)
   (t_T1_user_status_data *status_struct);
```

Elements

status_struct

Provides line status. For more information, see *t_T1_user_status_struct on page 215*.

T1SignalingHandler

A type definition for a function pointer that points to a user handler that will receive a t_T1_user_signaling_data structure pointer as a parameter.

Syntax

```
typedef void
    (*T1SignalingHandler)
    (t_T1_user_signaling_data *signaling_struct);
```

Elements

signaling_struct

Reads and writes signaling data. For more information, see *t_T1_user_signaling_data t_T1_signaling_data on page 213*.

t_E1_line_coding

Enumerate the line coding for a framer (port).

Syntax

```
typedef enum {
    NON_HDB3
    HDB3
} t_E1_line_coding;
```

Elements

NON_HDB3	The framer uses non-HDB3 line coding.
HDB3	The framer uses HDB3 line coding.

t_E1_signaling_mode

Enumerates the signaling mode for a framer (port).

Syntax

```
typedef enum {
    CAS,
    CCS
} t_E1_signaling_mode;
```

Elements

CAS	Channel Associated Signaling.
CCS	Common Channel Signaling.

t_E1_line_buildout

Enumerates the line buildout for a framer (port).

Syntax

```
typedef enum {
    BUILDOUT_75_OHM,
    BUILDOUT_120_OHM
} t_E1_line_build_out;
```

Elements

BUILDOUT_75_OHM
Sets the framer (specifically the LIU portion) to 75OHM operation.

BUILDOUT_120_OHM
Sets the framer (specifically the LIU portion) to 120OHM operation.

t_E1_user_config_struct

Specifies the complete configuration for a framer (port).

Syntax

```
typedef struct {
    t_T1E1_framer_id    framer_id;
    t_E1_line_coding    line_coding;
    t_E1_signaling_mode signaling_mode;
    BOOL32              crc4_enable;
    t_E1_line_build_out line_build_out;
    uchar               idle_code;
    ulong               idle_channels;
    BOOL32              framer_loopback_enable;
    BOOL32              local_loopback_enable;
    BOOL32              remote_loopback_enable;
} t_E1_user_config_struct;
```

Elements

The first three fields and the line `line_build_out` field are described above in the type definitions.

`framer_id` Specifies a framer (port) for functions which can operate on any port. For more information, see *t_T1E1_framer_id on page 203*.

`line_coding`
Specifies the line coding for a framer (port). For more information, see *t_E1_line_coding* on page 223.

`signaling_mode`
Enumerates the signaling mode for a framer (port). For more information, see *t_E1_signaling_mode* on page 224.

`crc4_enable`
Sets the CRC4 multiframe mode. A value of TRUE sets the mode.

`line_build_out`
Specifies the line buildout for a framer (port). For more information, see *t_E1_line_buildout* on page 225.

`idle_code` Specifies the pattern to inject into transmitted data.

`idle_channels`
The lower 32 bits specify the channels to inject the pattern specified in `idle_code` into.

A 1 in any bit indicates that the pattern is to be injected. Bit 0 is associated with the first DS0 of the E1 line while bit 31 is associated with the last DS0.

`framer_loopback_enable`
Instructs the framer to perform a system-side loopback. Setting any field to TRUE enables the loopback.

local_loopback_enable

Instructs the framer to perform a system-side loopback. Setting any field to TRUE enables the loopback.

remote_loopback_enable

Instructs the framer to perform a line-side loopback. Setting any field to TRUE enables the loopback.

t_E1_user_signaling_data

Reads and writes signaling data.

This structure is identical to `t_T1_user_signaling_data` since the `t_E1_signaling_data` field is typedef'ed to `t_T1_signaling_data`. See description of the corresponding T1 structures. The only difference is that the E1 structure makes use of all 32 bits where as the T1 structure uses only 24, 1 per DS0.

Syntax

```
typedef struct {
    t_T1E1_framer_id framer_id;
    t_E1_signaling_data signaling_data;
} t_E1_user_signaling_data;
```

Elements

`framer_id` Specifies a framer (port) for functions which can operate on any port. For more information, see *t_T1E1_framer_id on page 203*.

`signaling_data`
Reads and writes signaling data. For more information, see *t_T1_user_signaling_data t_T1_signaling_data* on page 213.

E1SignalingHandler

Type definition for user-specified signaling event handler.

Syntax

```
typedef void
   (*E1SignalingHandler)
   ( t_E1_user_signaling_data *signaling_struct);
```

Elements

signaling_struct

Reads and writes signaling data. For more information, see *t_E1_user_signaling_data on page 228*.

E T8100 library

This appendix describes the T8100 driver library. This appendix assumes that you have a working understanding of the T8100 device family and of the H.100/H.110 Bus.

When reading this file online, you can immediately view information about any topic by placing the mouse cursor over a test name and clicking.

For information about...	Go to this page...
Overview	231
Making and breaking connections	231
Sample startup sequence	234
Function list	235
Functions	237
Structures	246

Overview

The T8100 Status and Control API is the interface to the T8100 driver.

The T8100 driver provides the software interface to the Lucent T8100 Time Slot Interchange device of the SPIRIT platform. The driver works under the VxWorks real-time OS. The driver is not a "true" VxWorks driver as it is not built as part of the kernel. It does use VxWorks system functions, and therefore requires the OS.

The driver controls and monitors the T8100. More specifically, the driver supports these devices in the T8100 family:

- T8100
- T8100A
- T8105

The driver consists of:

- Interrupt handling tools
- Function list

Making and breaking connections

setT8100SwitchConfig allows the user application to configure switching for one or all timeslots within the T8100. The function's parameter's structure type, t_T8100SwitchConfig, and its associated sub-types are defined in iop_to_t8100.h and were shown above for reference. Using this function, the user can specify local-to-local, local-to-CT-bus, and CT-bus-to-CT-bus connections.

Connections are specified with a source (input) and destination (output) for the connection. Sources and destinations are specified in terms of resources (DSP, T1, E1, HDLC, CT-Bus, etc), ports within those resources (for multiple port devices) and timeslot numbers. The range of timeslot values varies based on the stream rate for that resource's port.

Due to the T8100's limited number of CT-Bus connections (256 for T8100 and T8100A, 512 for T8105(5)) versus the number of switching combinations, the user application must assign each CT-Bus related[4] connection a unique connection number, ctbus_connect_num, in the range of 0..255 (0..511 for T8105). This number is used to identify the connection in the event that a connection must first be broken to make room for a new connection. In this case, the connection using the specified connection number is broken, then the newly specified connection is made and assigned its number. Connections not using the CT-Bus as either the source or destination of the connection need not specify a connection number.

Three special "resources" exist to support breaking connections and outputting patterns:

- T8100_CT_BUS_DISCONNECT: The source and destination resource for connections which are to be broken. The `ctbus_connect_num` field is used to specify the connection.
- T8100_LOCAL_DISCONNECT: The source resource to break a local connection. The port and timeslot fields associated with this resource are not used. However, the destination must be the actual resource, port and timeslot of the connection to be broken.
- T8100_PATTERN: A connection source resource when outputting a pattern to a destination resource timeslot. When T8100_PATTERN is specified as the source resource, the source timeslot field is used to specify the 8-bit pattern to be output on the destination resource, port and timeslot.

Broadcasting

The T8100 supports broadcasting of timeslots, that is, data from one input timeslot can be sent to multiple output timeslots. There are four possible broadcast paths:

- One (1) Local/On-board input to N Local/On-Board outputs
- One (1) Local/On-board input to N CT-Bus outputs
- One (1) CT-Bus input to N Local/On-Board outputs
- One (1) CT-Bus input to N CT-Bus outputs

For broadcast connections which do not involve the CT-Bus (case 1 above), commanding a broadcast connection to N output timeslots is identical to commanding N standard connections. To broadcast a timeslot to N outputs, specify N connections in which the source resource, port and timeslot for each connection is identical. The destination resources, ports, and timeslots must be unique.

For broadcast connections which involve CT-Bus timeslots (cases 2, 3 and 4 above) as either the source or destination of the connection, programming is slightly different from both the CT-Bus non-broadcast case and the Local-to-Local

broadcast case. For non-broadcast CT-Bus connections, specify a unique connection number for each connection. For a CT-Bus broadcast, specify the same connection number for each connection which is part of the broadcast. Like the Local-to-Local broadcast case, the source resource, port, and timeslot is fixed for each broadcast connection element.

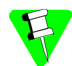
The following code samples show broadcast connections for the Local-to-Local and CT-Bus cases.

Example 1: Local-to-local broadcast 1 local timeslot (T1 port 0, timeslot 0) to 4 local timeslots (DSP A, port 0, timeslots 0..3)

```
configStruct.t8100SwitchCfg.number_of_connections = 4;
   configStruct.t8100SwitchCfg.connections = //ptr to connection array
(t_T8100Connection *)connections;

for(i=0;i<4;++i) {
   connections[i].connect_src.resource = T8100_T1;
   connections[i].connect_src.port = 0;
   connections[i].connect_src.timeslot = 0; // fixed
   connections[i].connect_dest.resource = T8100_DSP;
   connections[i].connect_dest.port = T8100_DSP_A_SP_0;
   connections[i].connect_dest.timeslot = i; // incrementing
   // ctbus_connect_num field for local-local connections is not used
}
hostControlPeripheral(uIopNum, TASK_T8100, CONFIG_T8100_SWITCHING,
&configStruct);
```

Example 2: Local-to-CT-Bus broadcast 1 local timeslot (T1 port 0, timeslot 0) to (applies to cases 2, 3, and 4) 4 CT-Bus timeslots (Stream 5, timeslots 0..3)

```
configStruct.t8100SwitchCfg.number_of_connections = 4;
   configStruct.t8100SwitchCfg.connections = //ptr to connection array
(t_T8100Connection *)connections;

for(i=0;i<4;++i) {
   connections[i].connect_src.resource = T8100_T1;
   connections[i].connect_src.port = 0;
   connections[i].connect_src.timeslot = 0; // fixed
   connections[i].connect_src.ctbus_connect_num = 0; // fixed
   connections[i].connect_dest.resource = T8100_CT_BUS;
   connections[i].connect_dest.port = 5;
   connections[i].connect_dest.timeslot = i; // incrementing
   connections[i].connect_dest.ctbus_connect_num = 0; // fixed
}
hostControlPeripheral(uIopNum, TASK_T8100, CONFIG_T8100_SWITCHING,
&configStruct);
```



The `ctbus_connect_num` for each of the broadcast members must be the same. However, this number must be different from the ctbus_connect_num of any other active connections.

(4) Either the source or destination (or both) of the connection is the CT-BUS

(5) This number represents the maximum number of CT-BUS related connections assuming that the source and destination of the connection are not both either even numbered CT-BUS streams or odd number CT-BUS streams. If all CT-BUS connections were to connect even streams to even streams or odd streams to odd streams, the number of connections would be reduced in half.

Sample startup sequence

```
void initT8100(void)
```

Initialize the T8100 card and driver. Not required if iopInit is being called.

```
int setT8100Handler(void (*UserT8100Handler) (int status) )
```

Set a callback function for errors/state changes.

```
void setT8100ClockConfig(IN const t_T8100ClockConfig *T8100ClockConfig)
```

Set the configuration of the T8100 reference and bus clocks.

```
void setT8100StreamConfig(IN const t_T8100StreamConfig *T8100StreamConfig)
```

Set the rates for the on-board and CT-Bus streams.

```
void setT8100SwitchConfig(IN const t_T8100SwitchConfig
*T8100SwitchConfig)
```

Make timeslot connections, both on-board and CT-Bus.

Function list

This appendix lists function descriptions alphabetically. You can find a function in this appendix either by locating it alphabetically within its group, or by using Table E-1 below, which groups like functions together.

Use this table to identify the T8100 driver functions you want to use. Use the function description later in this appendix to obtain detailed information, including syntax and parameter values:

Table E-1. T8100 functions

Type	Function	Description
Configuration		Configures the T8100.
	initT8100	Initializes the driver and the T8100, and identifies the installed device (T8100, T8100A, or T8105).
	setT8100ClockConfig	Configures the clocking of the T8100.
	setT8100StreamConfig	Sets the T8100's stream rates.
	setT8100SwitchConfig	Sets the T8100's switching.
	clearT8100ClockFault	Clears a clock fault in the T8100.
	clearT8100MemoryFault	Clears a memory (CAM) fault in the T8100.
	setT8100ClockFaultMask	Disables the monitoring of clock faults.
Status monitoring		Polls and reads the status of the T8100.
	getT8100ErrorStatus	Reads the status of the T8100 error status registers: SYSERR, CLKERR1, CLKERR2, and CLKERR3.
Interrupt/event controls		Specifies a handler (callback function) to call when an interrupt event occurs.
	setT8100Handler	Specifies the event handler to call when detecting a T8100 interrupt.

Function prototypes and type definitions for the driver interface are contained in the following header file:

```
iop_to_t8100.h
```

The functions and their data types are described in the pages that follow. The descriptions use the terms H.100 Bus, H.110 Bus and CT-Bus interchangeably.

Limitations

Time slot assignments for T1 and E1 interfaces: There are differences in the mapping of T8100 time slots depending on the interface used, as illustrated by the following table. The developer should be aware of these differences when assigning DSP data channel resources to time slots using the T8100 peripheral.

Table E-2. Time slot numbers

T8100	T1	E1
0	1 (data 0)	0 (signaling 0)
1	2 (data 1)	1 (data 0)
2	3 (data 2)	2 (data 1)
3	4 (data 3)	3 (data 2)
4	5 (data 4)	4 (data 3)
5	6 (data 5)	5 (data 4)
6	7 (data 6)	7 (data 6)
7	8 (data 7)	7 (data 6)
8	9 (data 8)	8 (data 7)
9	10 (data 9)	9 (data 8)
10	11 (data 10)	10 (data 9)
11	12 (data 11)	11 (data 10)
12	13 (data 12)	12 (data 11)
13	14 (data 13)	13 (data 12)
14	15 (data 14)	14 (data 13)
15	16 (data 15)	15 (data 14)
16	17 (data 16)	16 (signaling 1)
17	18 (data 17)	17 (data 15)
18	19 (data 18)	18 (data 16)
19	20 (data 19)	19 (data 17)
20	21 (data 20)	20 (data 18)
21	22 (data 21)	21 (data 19)
22	23 (data 22)	22 (data 20)
23	24 (data 23)	23 (data 21)
24	N/A	24 (data 22)
25	N/A	25 (data 23)
26	N/A	26 (data 24)
27	N/A	27 (data 25)
28	N/A	28 (data 26)
29	N/A	29 (data 27)
30	N/A	30 (data 28)
31	N/A	31 (data 29)

Functions

initT8100

Initializes the driver and the T8100, and identifies the installed device (T8100, T8100A, or T8105).

This function *must* be called before using any other driver functions, however, the user need not call this function if `iopInit` is being called, since the iopInit function makes the call. Following this function call, the T8100 and the CT-Bus are in the following state:

- No CT-Bus timeslots are driven, all connections are broken.
- All CT-Bus bus clocks are turned off, CT-Bus stream rates set to 8.192MHz.
- The T8100 reference clock is set to be the on-board oscillator.
- All local/on-board connections are broken.
- The local/on-board streams are set to a 2.048MHz rate (32 timeslots).
- On-board device clocks are set to a 2.048MHz rate (DSP clocks, framer clocks).

Syntax

```
void initT8100(void)
```

Parameters

None.

Return values

SUCCESS Successful completion.

FAILURE Unsuccessful completion.

setT8100ClockConfig

Configures the clocking of the T8100.

This includes selecting reference clocks for normal and fallback operation as well as determining which (if any) clocks to drive onto the bus. This function is typically called first following initT8100.

Syntax

```
void setT8100ClockConfig(
   IN const t_T8100ClockConfig *T8100ClockConfig
)
```

Parameters

T8100ClockConfig
A pointer to the t_T8100ClockConfig structure. For more information about this structure, see *t_T8100ClockConfig* on page 250.

Return values

SUCCESS	Successful completion.
FAILURE	Unsuccessful completion.

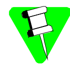
setT8100StreamConfig

Sets the T8100's stream rates.

You can configure stream rates for the three on-board stream groups as well as the CT-Bus. The 32 streams of the CT-Bus are broken up into eight groups of four streams each. Each group can be independently configured.

Available rates include:

- 8.192MHz
- 4.096MHz
- 2.048MHz
- 0MHz (off)



Call this function prior to making timeslots connections. Changing stream rates results in reconfiguration of T8100 internal switch memory and may result in undesired connectivity.

Syntax

```
void setT8100StreamConfig(
    IN const t_T8100StreamConfig *T8100StreamConfig
)
```

Parameters

T8100StreamConfig
A pointer to the t_T8100StreamConfig structure. For more information about this structure, see *t_T8100StreamConfig* on page 253.

Return values

SUCCESS	Successful completion.
FAILURE	Unsuccessful completion.

setT8100SwitchConfig

Sets the T8100's switching.

The function is passed the number of connections to make (or break), and the end points for those connections. These connections can be of any type: local-to-local, local-to-CT-Bus or CT-Bus to CT-Bus.

Syntax

```
void setT8100SwitchConfig(
    IN const t_T8100SwitchConfig *T8100SwitchConfig
)
```

Parameters

T8100SwitchConfig

A pointer to the t_T8100SwitchConfig structure. For more information about this structure, see *t_T8100SwitchConfig* on page 258.

Return values

SUCCESS	Successful completion.
FAILURE	Unsuccessful completion.

clearT8100ClockFault

Clears a clock fault in the T8100.

Syntax

```
int clearT8100ClockFault(void)
```

Parameters

None.

Return values

SUCCESS	Successful completion.
FAILURE	Unsuccessful completion.

clearT8100MemoryFault

Clears a memory (CAM) fault in the T8100.

Syntax

```
int clearT8100MemoryFault(void)
```

Parameters

None.

Return values

SUCCESS Successful completion.

FAILURE Unsuccessful completion.

setT8100ClockFaultMask

Disables the monitoring of clock faults.

This function allows you to disable monitoring of known bad clocks.

Syntax

```
int setT8100ClockFaultMask(
    IN const int clock_mask
)
```

Parameters

clock_mask

The lower 8 bits of *clock_mask* are inverted and and'ed with the contents of the T8100's CKW register. For the T8100A and T8105, bit 9 of *clock_mask* is inverted and and'ed with the LSB of the CLKERR3 register.

Return values

SUCCESS	Successful completion.
FAILURE	Unsuccessful completion.

getT8100ErrorStatus

Reads the status of the T8100 error status registers: SYSERR, CLKERR1, CLKERR2, and CLKERR3.

Syntax

```
int getT8100ErrorStatus(void)
```

Parameters

None.

Return values

SUCCESS	Return data is formatted as follows:
	Bits 07–00 SYSERR
	Bits 15–08 CLKERR1
	Bits 23–16 CLKERR2
	Bits 31–24 CLKERR3 (T8100A & T8105 only)
FAILURE	No information available.

setT8100Handler

Specifies the event handler to call when detecting a T8100 interrupt.

The value passed to the function is identical to the return parameter of getT8100ErrorStatus. setT8100Handler can be called with a NULL handler in order to disable callbacks.

Syntax

```
int setT8100Handler(
   void (*UserT8100Handler) (int status)
)
```

Parameters

UserT8100Handler
: The user call to handle T8100 interrupt events.

Return values

SUCCESS	Successful completion.
FAILURE	Unsuccessful completion.

Structures

Use this table to identify the T8100 driver structures. Use the structure description later in this appendix to obtain detailed information, including syntax and parameter values.

Table E-3. T8100 structures

Call	Description
t_ref_clk	Selects the primary reference of the T8100.
t_fallback_clk	Selects the fallback reference of the T8100.
t_netref_clk	Selects the clock reference used to generate the CT-NETREF bus clocks.
t_T8100ClockConfig	Used by the setT8100ClockConfig function to control the clocking of the T8100.
t_stream_rate	Selects the CT-BUS rate.
t_T8100StreamConfig	Used by the setT8100StreamConfig function to control the stream rates of the T8100.
t_source_dest	Specifies a connection's endpoint (source or destination).
t_T8100Connection	Specifies a single connection.
t_T8100SwitchConfig	Specifies a series (one or more) connections to make (or break).

t_ref_clk

Selects the primary reference of the T8100.

Syntax

```
typedef enum {
    REF_LOCAL= 0,
    REF_CT_NETREF= 1,
    REF_CT_C8_A= 2,
    REF_CT_C8_B= 3,
    REF_MVIP= 4,
    REF_HMVIP= 5,
    REF_SCSA_2M= 6,
    REF_SCSA_4_8M= 7,
    REF_T1E1_1= 8,
    REF_T1E1_2= 9,
    REF_T1E1_3= 10,
    REF_T1E1_4= 11,
    REF_CT_NETREF2= 12
} t_ref_clk;
```

t_fallback_clk

Selects the fallback reference of the T8100.

Syntax

```
typedef enum {
   FB_LOCAL= 0,
   FB_CT_C8_A= 1,
   FB_CT_C8_B= 2,
   FB_CT_NETREF= 3,
   FB_T1E1_1= 4,
   FB_T1E1_2= 5,
   FB_T1E1_3= 6,
   FB_CT_NETREF2= 7
} t_fallback_clk;
```

t_netref_clk

Selects the clock reference used to generate the CT-NETREF bus clocks.

Syntax

```
typedef enum {
   NETREF_LOCAL= 0,
   NETREF_T1E1_1= 1,
   NETREF_T1E1_2= 2,
   NETREF_T1E1_3= 3,
   NETREF_T1E1_4 = 4,
   NETREF_CT_NETREF= 5,
   NETREF_CT_NETREF2= 6
} t_netref_clk;
```

t_T8100ClockConfig

Used by the setT8100ClockConfig function to control the clocking of the T8100.

Syntax

```
typedef struct {
   t_ref_clkreference_clk_select;
   t_netref_clknetref_select;
   t_fallback_clkfallback_clk_select;
   bool netref_enable;
   bool netref2_enable;
   bool frame_clk_a_enable;
   bool frame_clk_b_enable;
   bool compat_clks_enable;
   boolfallback_enable;
} t_T8100ClockConfig;
```

Elements

reference_clk_select

Selects the primary reference clock for the T8100. The T8100 will lock to this clock (under non-fallback conditions) and generate both its internal references as well as any external clocks, based on this clock. This clock can be either the on-board reference, any of the H.100/110 bus clocks or one of the E1 or T1 loop timing clocks(1).

netref_select

Selects the reference clock for generation of the CT_NETREF signal. This field selects the reference but does not enable the output.

fallback_clk_select

Selects the reference clock to which the T8100 will switch when an error is detected on the signal used as the *reference_clk_select*. Fallback must be enabled for this to occur.

netref_enable

Determines whether the T8100's CT_NETREF output goes onto the H.100/110 bus. You can select one of these:

TRUE	CT_NETREF output goes onto the H.100/110 bus.
FALSE	CT_NETREF output does not go onto the H.100/110 bus.

netref2_enable

Determines whether the T8100's CT_NETREF_2 output goes onto the H.100/110 bus. You can select one of these:

TRUE	CT_NETREF_2 is available only for H.110 applications.
FALSE	CT_NETREF_2 is not available for H.110 applications.

frame_clk_a_enable

Determines whether the T8100's CT_C8_A and /CT_FRAME_A outputs go onto the H.100/110 bus. You can select one of these:

TRUE	CT_C8_A and /CT_FRAME_A output goes onto the H.100/110 bus.
FALSE	CT_C8_A and /CT_FRAME_A output does not go onto the H.100/110 bus.

frame_clk_b_enable

Determines whether the T8100's CT_C8_B and /CT_FRAME_B outputs go onto the H.100/110 bus. You can select one of these:

TRUE	CT_C8_B and /CT_FRAME_B output goes onto the H.100/110 bus.
FALSE	CT_C8_B and /CT_FRAME_B output does not go onto the H.100/110 bus.

compat_clks_enable

Determines whether the T8100's compatibility clocks and frame strobes go onto the H.100/110 bus. These signals include /FR_COMP,SCLK,SCLKx2,C2,/C4,/C16+,/C16-[(2)]. You can select one of these:

TRUE	Compatibility clocks and frame strobes go onto the H.100/110 bus.
FALSE	Compatibility clocks and frame strobes do not go onto the H.100/110 bus.

fallback_enable

Determines whether to engage the T8100's fallback (switch) mode. You can select one of these:

(1)	Enables the fallback mode. The T8100 will fallback (switch) to the reference clock specified in *fallback_clk_select* when an error is detected on the primary reference clock specified in *reference_clk_select*. Select this option if E1 or T1 option card is installed.
(2)	Disables the fallback mode. Only SCLK, SCLKx2 and /FR_COMP are defined for use in H.110 systems.

t_stream_rate

Selects the CT-BUS rate.

Syntax

```
typedef enum {
   TWO_MHZ= 0,
   FOUR_MHZ= 1,
   EIGHT_MHZ= 2,
   DISABLE_GROUP= 3
} t_stream_rate;
```

t_T8100StreamConfig

Used by the setT8100StreamConfig function to control the stream rates of the T8100.

On-board/local stream rates are controlled in groups:

- **BSP 0**: DSP Buffered Serial Port (BSP) 0 group.
- **BSP 1**: DSP Buffered Serial Port (BSP) 1 group.
- **E1/T1**: SPIRIT board's option card site group

For the H.100/110 bus, each group is a set of four consecutive H.100/110 streams. The streams rates correspond to 32, 64 and 128 timeslots respectively.

Syntax

```
typedef struct {
    t_stream_rate dsp_bsp0_rate;
    t_stream_rate dsp_bsp1_rate;
    t_stream_rate e1t1_rate;
    t_stream_rate ct_bus_03_00_rate;
    t_stream_rate ct_bus_07_04_rate;
    t_stream_rate ct_bus_11_08_rate;
    t_stream_rate ct_bus_15_12_rate;
    t_stream_rate ct_bus_19_16_rate;
    t_stream_rate ct_bus_23_20_rate;
    t_stream_rate ct_bus_27_24_rate;
    t_stream_rate ct_bus_31_28_rate;
} t_T8100StreamConfig;
```

Elements

dsp_bsp0_rate

Sets the stream rate for the streams which are connected to BSP 0 of each DSP.

dsp_bsp1_rate

Sets the stream rate for the streams which are connected to BSP 1 of each DSP.

e1t1_rate

Sets the stream rate for the streams which are connected to the SPIRIT board's option card site. You can set the rate to one of these:

2.048MHz Sets the stream rate to 2.048MHz. If the RadiSys E1 or T1 option card is being used, you must select this option.

4.096MHz Sets the stream rate to 4.096MHz.

8.192MHz Sets the stream rate to 8.192MHz.

Disabled Disables the stream.

ct_bus_yy_xx_rate

Sets the stream rates for the H.100/110 streams xx thru yy.

Restrictions

The H.100/110 streams rates can be configured without restriction, that is, any combination of rates is allowed. The on-board/local streams have the following restrictions:

- If `e1t1_rate` is set to EIGHT_MHZ, the other 2 on-board/local stream groups (the DSP BSP streams) are disabled.
- If `dsp_bsp1_rate` is set to EIGHT_MHZ, the on-board/local stream connected to BSP 0 of each DSP are disabled.

  In general the following must hold true:

  $8*TS_{E1T1} + 4*TS_{DSPBSP1} + 4*TS_{DSPBSP0} <= 1024$

  Where:

  - TS_{E1T1} is the number of timeslots for the E1T1 streams(3)
  - $TS_{DSPBSP1}$ is the number of timeslots for the DSP BSP1 streams
  - TS DSPBSP0 is the number of timeslots for the DSP BSP0 streams

    2.048MHz rate = 32 timeslots
    4.096MHz rate = 64 timeslots
    8.192MHz rate = 128 timeslots

- When a group is disabled via the DISABLE_GROUP command, its rate is still considered to be 2.048MHz, although the T8100's output for that group is disabled.

```
/* list of all possible TDM resources on a SPIRIT board */
#define T8100_DSP0
#define T8100_T11
#define T8100_E12
#define T8100_HDLC3
#define T8100_CODEC4
#define T8100_CT_BUS5
#define T8100_CT_BUS_DISCONNECT6
#define T8100_LOCAL_DISCONNECT7
#define T8100_PATTERN8

/* constants used to specify DSP and it serial port in the port field of the*/
/* t_source_dest structure */
#define T8100_DSP_A_SP_0 0x0/* DSP A (0) Serial port 0 */
#define T8100_DSP_A_SP_1 0x1/* DSP A (0) Serial port 1 */
#define T8100_DSP_B_SP_0 0x4/* DSP B (1) Serial port 0 */
#define T8100_DSP_B_SP_1 0x5/* DSP B (1) Serial port 1 */
#define T8100_DSP_C_SP_0 0x8/* DSP C (2) Serial port 0 */
#define T8100_DSP_C_SP_1 0x9/* DSP C (2) Serial port 1 */
#define T8100_DSP_D_SP_0 0xC/* DSP D (3) Serial port 0 */
#define T8100_DSP_D_SP_1 0xD/* DSP D (3) Serial port 1 */

#define T8100_DSP_E_SP_0 0x2/* DSP E (4) Serial port 0 - future use */
#define T8100_DSP_E_SP_1 0x3/* DSP E (4) Serial port 1 - future use */
#define T8100_DSP_F_SP_0 0x6/* DSP F (5) Serial port 0 - future use */
#define T8100_DSP_F_SP_1 0x7/* DSP F (5) Serial port 1 - future use */
#define T8100_DSP_G_SP_0 0xa/* DSP G (6) Serial port 0 - future use */
#define T8100_DSP_G_SP_1 0xb/* DSP G (6) Serial port 1 - future use */
```

```
#define T8100_DSP_H_SP_0 0xe/* DSP H (7) Serial port 0 - future use */
#define T8100_DSP_H_SP_1 0xf/* DSP H (7) Serial port 1 - future use */

#define T8100_CONNECT_CONST_DELAY0 /* used to specify constant delay */
#define T8100_CONNECT_MIN_DELAY1 /* used to specify minimum delay */
```

t_source_dest

Specifies a connection's endpoint (source or destination).

Syntax

```
typedef struct {
 unsigned int mode;
 unsigned int resource;
 unsigned int port;
 unsigned int timeslot;
 unsigned int ctbus_connect_num
} t_source_dest;
```

Elements

mode Specifies the connection to be either constant delay or minimum delay.

resource Specifies the connection's resource number. For example: CT-Bus, DSP, T1, or HDLC.

Most resources have multiple ports. In the case of the CT-Bus, each stream is a port. In the case of an E1 or T1 card, the port corresponds to the framer.

port Specifies the connection's port number. For example: CT-Bus, DSP, T1, or HDLC.

timeslot Specifies the connection's timeslot number. The maximum value for this field is a function of the stream rate selected.

ctbus_connect_num
Used for CT-Bus connection only and is described in the "Making and Breaking Connections" section

t_T8100Connection

Specifies a single connection.

Syntax

```
typedef struct {
    t_source_dest connect_src;
    t_source_dest connect_dest;
} t_T8100Connection;
```

Elements

connect_src
The connection's source.

connect_dest
The connection's destination.

t_T8100SwitchConfig

Specifies a series (one or more) connections to make (or break).

Syntax

```
typedef struct {
   ulongnumber_of_connections;
   t_T8100Connection*connections;
} t_T8100SwitchConfig;
```

Elements

number_of_connections

Indicates the total number of connections to be made or broken. The maximum value is 512 (256 for T8100 and T8100A), but is further limited by the maximum number of connections supported by the device.

**connections*

A pointer to number_of_connections number of connections, each of type t_T8100Connection. Each of these connections is made up of a connect_src and connect_dest. Each connect_src and connect_dest is in turn made up of a mode, resource, port, timeslot and ctbus_connect_num (CT-BUS Connections only)

F Service descriptions

This appendix describes TASK services.

This appendix lists structures in the order listed in the table below. Use this table to identify the service or services you want to use. Use the service description later in this appendix to obtain detailed information, including syntax and parameter values.

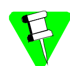
For information about...	Go to this page...
Codec	260
stCodec	260
Echo cancellation	263
stEchoCanc	263
Tone generation	264
stTdmToneGen	264
stPktToneGen	265
Tone detection	266
stTdmDTMFDet	266
stPktDTMFDet	267
stCPTDet	268
stMFDet	270
RTP packetization	272
stRtpEncode	272
stRtpDecode	274
Signaling	276
stCAS	276
stQDS0Hdlc	277
Alarming	278
stEthernetAlarm	278
stT1E1Alarm	278
Audio processing	278
stAGC	281
Internal	282
stPacketBuilder	282
stPacketParser	283

Codec

stCodec

Converts voice, fax, or modem data between TDM and a packetized form.

In the voice case, it takes a frame (often 10ms) of TDM audio data, compresses it according to one of the voice codecs such as G.711, G.723, G.729 etc., and outputs the compressed result. In the reverse direction it takes a compressed packet corresponding to an audio frame and decompresses it into the full data frame for playout via a TDM channel. In addition to the standard compression/decompression functions, some codecs also support Voice Activity Detection and Comfort Noise Generation, which allows suppression of packetized data during periods of silence.



upEnableService and upConfigService disables the stTdmToneGen service, if it is currently running on the same channel.

Config data

UP_CODEC_CONFIG_ST

```
typedef struct
{
    UP_CODEC_ET eCodec;
    UP_CODEC_PARAM_UT tCodecParams;
} UP_CODEC_CONFIG_ST;
```

eCodec The codec to enable and configure.

tCodecParams
The parameters for this codec; may be NULL for codecs which are not configured.

UP_CODEC_ET

```
typedef enum {
    ctG729A = 1,
    ctG723,
    ctG711Mu,
    ctG711A,
    ctG726_32,
    ctG3Fax,
    ctV90,
    ctNoCoder, /* disable codec (used internally) */
    ctG729,
    ctGSM
} UP_CODEC_ET;
```

UP_CODEC_PARAM_UT

```
typedef union
{
    UP_CODEC_G711_PARAM_ST tG711Param;
    UP_CODEC_G729_PARAM_ST tG729Param;
    UP_CODEC_G723_PARAM_ST tG723Param;
} UP_CODEC_PARAM_UT;
```

UP_CODEC_G711_PARAM_ET

```
typedef struct {
    enum
    {
        enumMULAW=0,
        enumALAW
    } eLaw;
    UP_ENABLE_ET eVadEnable;
    RSYS_INT32 lVadLowSigThreshold;
    UP_ENABLE_ET eBfmEnable;
} UP_CODEC_G711_PARAM_ST;
```

eLaw Selects between G.711 µ-Law and A-Law versions. You can select one of these:

enumMULAW Selects µ-Law coding.

enumALAW Selects A-Law coding.

eVadEnable

Allows the Voice Activity Detector to suppress packet generation on the encoder side. You can select one of these values:

enumEnabled

Enables VAD.

enumDisabled

Always generates packets.

lVadLowSigThreshold

VAD noise threshold in dBm.

eBfmEnable

Enables and disables bad frame masking.

UP_CODEC_G729_PARAM_ET

```
typedef struct {
    UP_ENABLE_ET eVadEnable;
} UP_CODEC_G729_PARAM_ST;
```

eVadEnable

Allows the Voice Activity Detector to suppress packet generation on the encoder side. You can select one of these:

enumEnabled

Enables VAD.

enumDisabled

Always generates packets.

UP_CODEC_G723_PARAM_ST

```
typedef struct {
    enum {
        enumRate63=0,/* 6300 bps*/
        enumRate53/* 5300 bps*/
    } eRate;
    UP_ENABLE_ET UseHp;/* High pass filter enable*/
    UP_ENABLE_ET UsePf;/* Post filter enable*/
    UP_ENABLE_ET UseVx;/* VAD enable*/
} UP_CODEC_G723_PARAM_ST;
```

`UseVx` Allows the Voice Activity Detector to suppress packet generation on the encoder side. You can select one of these:

`enumEnabled`
Enables VAD.

`enumDisabled`
Always generates packets.

Events There are no events associated with this service.

Echo cancellation

stEchoCanc

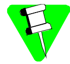
Tries to remove time-delayed versions of a TDM channel's output from its input stream.

These echoes are typically introduced when the TDM channel is translated into a two-wire analog line somewhere outside the gateway. The echo canceller uses an adaptive filter with a 32 millisecond buffer to eliminate near-end echoes.

Config data

UP_ECHO_CONFIG_ST

```
typedef struct {
   RSYS_INT32lNlpThreshold;/* Non-Linear Processor Threshold*/
   enum   {
      TL8ms = 1,
      TL16ms,
      TL24ms,
      TL32ms
   } eTapLength;
   UP_ENABLE_ET lSlowAdaptation;
   UP_ENABLE_ET lFreezeAdaptation;
   UP_ENABLE_ET lNLPDisable;
} UP_ECHO_CONFIG_ST;
```



The echo canceller's initial version includes:

- A fixed *lNlpThreshold* value; this flag currently has no effect.
- A fixed *eTapLength* value of 32ms; this flag currently has no effect.

Events

There are no events associated with this service.

Tone generation

stTdmToneGen

Generates tones or tone-pairs to a TDM output with programmable cadence and cadence count.

While a cadence is in progress, this service overrides the decompression side of any codec concurrently enabled on the same channel. Once the cadence ends the channel automatically reverts to the codec output.

Config data

UP_TONEGEN_CONFIG_ST

```
typedef struct {
    RSYS_INT32 lFreq1;
    RSYS_INT32 lFreq2;
    RSYS_INT32 lAmplitudedB1;
    RSYS_INT32 lAmplitudedB2;
    RSYS_INT32 lCadenceOn;
    RSYS_INT32 lCadenceOff;
    RSYS_INT32 lRepeat;
} UP_TONEGEN_CONFIG_ST;
```

`lFreq1` The primary tone's frequency in Hz.

`lFreq2` The secondary tone's frequency in Hz, or zero if no secondary tone is required.

`lAmplitudedB1` and `lAmplitudedB2`
Each tone's amplitude. If both tones are used, the total amplitude is the sum of the two individual amplitudes.

`lCadenceOn`
The tone or tone-pair's on-time, in ms. This number is rounded up to the nearest frame, i.e. 30ms. `CadenceOn=0` means leave the tone on forever, or until the tone generator is reconfigured or disabled.

`lCadenceOff`
The tone or tone-pair's off-time, in ms, rounded up to the nearest frame.

`lRepeat` The number of times to repeat the cadence; –1 means repeat forever.

Events

There are no events associated with this service.

stPktToneGen

Generates tones or tone-pairs to the input of a codec with programmable cadence and cadence count.

While a cadence is in progress the stTdmToneGen service overrides the TDM channel's input. Once the cadence ends the codec input automatically reverts to the TDM input.

Config data

UP_TONEGEN_CONFIG_ST

```
typedef struct {
   RSYS_INT32 lFreq1;
   RSYS_INT32 lFreq2;
   RSYS_INT32 lAmplitudedB1;
   RSYS_INT32 lAmplitudedB2;
   RSYS_INT32 lCadenceOn;
   RSYS_INT32 lCadenceOff;
   RSYS_INT32 lRepeat;
} UP_TONEGEN_CONFIG_ST;
```

lFreq1 The primary tone's frequency in Hz.

lFreq2 The secondary tone's frequency in Hz, or zero if no secondary tone is required.

lAmplitudedB1 and *lAmplitudedB2*
Each tone's amplitude. If both tones are used, the total amplitude is the sum of the two individual amplitudes.

lCadenceOn
The tone or tone-pair's on-time, in ms. This number is rounded up to the nearest frame, i.e. 30ms. *CadenceOn*=0 means leave the tone on forever, or until the tone generator is reconfigured or disabled.

lCadenceOff
The tone or tone-pair's off-time, in ms, rounded up to the nearest frame.

lRepeat The number of times to repeat the cadence; –1 means repeat forever.

Events

There are no events associated with this service.

Tone detection

stTdmDTMFDet

Performs DTMF detection on TDM input data.

It generates UP_EVT_DTMF_DETECTED events to alert the application of the digit detected.

Config data

UP_DTMF_CONFIG_ST

```
typedef struct {
   RSYS_INT32 lLowSigThreshold;
   *UP_ENABLE_ET supressOnToneDet;
} UP_DTMF_CONFIG_ST;
```

lLowSigThreshold
Absolute low-signal threshold.

supressOnToneDet
Supress audio during DTMF.

Events

UP_EVT_TDM_DTMF_DETECTED

UP_DTMF_DETECTED_DATA_ST

```
typedef struct {
   RSYS_UINT32 ulTimeStamp;
   RSYS_INT32 lDigit;
} UP_DTMF_DETECTED_DATA_ST;
```

ulTimeStamp
Contains the RTP timestamp when the digit was detected.

lDigit Contains the ASCII code for the digit detected {'0'-'9','A'-'D', '*', '#'} or zero when tone detection ends.

stPktDTMFDet

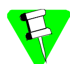
Performs DTMF detection on decompressed packet data before it is output to a TDM stream.

It generates UP_EVT_DTMF_DETECTED events to alert the application of the digit detected.

Config data

UP_DTMF_CONFIG_ST

```
typedef struct {
   RSYS_INT32 lLowSigThreshold;
   UP_ENABLE_ET supressOnToneDet;
} UP_DTMF_CONFIG_ST;
```



DTMF's initial version:

- Includes a fixed *lLowSigThreshold* value; this flag currently has no effect.
- Does not incorporate the *supressOnToneDet* feature; this flag currently has no effect.

lLowSigThreshold
: Absolute low-signal threshold.

supressOnToneDet
: Supress audio during DTMF.

Events

UP_EVT_PKT_DTMF_DETECTED

UP_DTMF_DETECTED_DATA_ST

```
typedef struct {
   RSYS_UINT32 ulTimeStamp;
   RSYS_INT32 lDigit;
} UP_DTMF_DETECTED_DATA_ST;
```

ulTimeStamp
: Contains the RTP timestamp when the digit was detected.

lDigit
: Contains the ASCII code for the digit detected {'0'-'9','A'-'D', '*', '#'} or zero when tone detection ends.

stCPTDet

Performs Call Progress Tone detection on TDM input data.

It generates UP_EVT_CPT_DETECTED events to alert the application of the call progress indication detected.

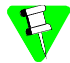
Config data

UP_CPT_CONFIG_ST

```
typedef struct {
   RSYS_INT32 lLowSigThreshold;
   RSYS_UINT32 ulDetFlags;
} UP_CPT_CONFIG_ST;
```

lLowSigThreshold
Absolute low-signal threshold.

ulDetFlags A bit field that designates which call progress tones are detected. In this field, tone 0 detection is enabled by bit 0, tone 1 by bit 1, etc. Set this field to 0xffffffff to enable all 32 possible tone/cadence detections, though only 22 are currently defined. See UP_CPM_TONES_ET in section 3.4.3.1 for the tone number definitions.



CPT's initial version has a fixed *lLowSigThreshold* value; this flag currently has no effect.

Events

UP_EVT_CPT_DETECTED
UP_CPT_DETECTED_DATA_ST

```
typedef enum {
   enumCptDial = 0,/* 0th combination (350+440Hz dial tone) */
   enumCptRecallDial = 1, /* 1st combination (350+440Hz Recall Dial tone) */
   enumCptConfirm = 2, /* 2nd combination (350+440Hz Conformation tone) */
   enumCptStutterDial = 3, /* 3rd combination (350+440Hz Stutter Dial tone)*/
   enumCptBusy = 4, /* 4th combination (480+620Hz Busy tone) */
   enumCptReorder = 5, /* 5th combination (480+620 Reorder tone) */
   enumCptRing = 6, /* 6th combination (440+480 Audible ring tone) */
   enumCptSpecialRing = 7, /* 7th comb (440+480Hz Special Audible ring tone) */
   enumCptCallWaiting = 8, /* 1st single (440Hz Call Waiting tone) */
   enumCptBusyVerify = 9, /* 2nd single (440Hz Busy Verification tone) */
   enumCptExecOverride = 10,/* 3rd single (440Hz Executive Override tone) */
   enumCptIntercept440 = 11,/* 4th single tone (440Hz Intercept tone) */
   enumCptIntercept620 = 12,/* 5th single tone (620Hz Intercept tone) */
   enumCptOffHook1040 = 13,/* 6th single tone (1400Hz Off Hook) */
   enumCptOffHook2060 = 14,/* 7th single tone (2060Hz Off Hook) */
   enumCptOffHook2450 = 15,/* 8th single tone  (2450Hz Off Hook) */
   enumCptOffHook2600 = 16,/* 9th single tone  (2600Hz Off Hook) */
   enumCptFaxTx = 17, /* 10th single tone (1100Hz fax TX tone) */
   enumCptFaxRx = 18, /* 11th single tone (2100Hz fax rx tone) */
   enumCptDataModem = 19, /* 12th single tone (2100Hz data modem tone) */
   enumCptLineTest = 20, /* 13th single tone (1004Hz line test tone) */
   enumCptSS7 = 21 /* 14th single tone (2010Hz SS7 tone) */
} UP_CPM_TONES_ET;
```

```
typedef struct {
    RSYS_UINT32 ulTimeStamp;
    UP_CPM_TONES_ET eTone;
} UP_CPT_DETECTED_DATA_ST;
```

ulTimeStamp
: Contains the RTP timestamp when the tone was detected. This denotes when the required cadence for this tone was established, not when the tone began.

eTone
: The detector value for the detected tone. Note that the listed enumerated tones are valid only for the CPM module's default configuration. The CPM module may be configured to detect many other combinations and cadences of tones and tone-pairs and the *tone* value simply reflects the entry matched in the CPM module's configuration.

stMFDet

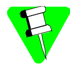
Performs MF Tone detection on TDM input data for R1 or R2 signaling.

It generates UP_EVT_MF_DETECTED events to alert the application of the digit/signal detected. Since the tone sets overlap, you must configure this service for the expected signaling type.

Config data

UP_MF_CONFIG_ST

```
typedef struct {
    RSYS_INT32 lLowSigThreshold;
    enum {
        R1 = 1,
        R2F,
        R2R
    } eMfTones;
} UP_MF_CONFIG_ST;
```



MF's initial version has a fixed *lLowSigThreshold* value; this flag currently has no effect.

lLowSigThreshold
Absolute low-signal threshold.

R1 R1 signaling.

R2F R2 forward signaling.

R2R R2 reverse signaling.

eMfTones
The set of tones to detect.

Events

UP_EVT_MF_DETECTED

For details about MF tone detector operation, see the *Line/Register Signaling (R1/R2 MF) for TMS320C6201 User's Manual.*

```
typedef enum {
    /* MF tones for R1 signaling by function */
    MFR1_LINE, MFR1_KP, MFR1_1, MFR1_2, MFR1_3,
    MFR1_4, MFR1_5, MFR1_6, MFR1_7, MFR1_8, MFR1_9,
    MFR1_0, MFR1_ST, MFR1_ST1, MFR1_ST2, MFR1_ST3,
    /* MF tones for R2 forward signaling by register */
    MFR2F1, MFR2F2, MFR2F3, MFR2F4, MFR2F5, MFR2F6,
    MFR2F7, MFR2F8, MFR2F9, MFR2F10, MFR2F11, MFR2F12,
    MFR2F13, MFR2F14, MFR2F15,
    /* MF tones for R2 reverse signaling by register */
    MFR2R1, MFR2R2, MFR2R3, MFR2R4, MFR2R5,MFR2R6,
    MFR2R7, MFR2R8, MFR2R9, MFR2R10, MFR2R11, MFR2R12,
    MFR2R13, MFR2R14, MFR2R15
} UP_R1R2_TONES_ET;
```

UP_MF_DETECTED_DATA_ET

```
typedef struct {
    RSYS_UINT32 ulTimeStamp;
    UP_R1R2_TONES_ET eTone;
} UP_MF_DETECTED_DATA_ST;
```

ulTimeStamp
Contains the RTP timestamp when the tone or tone-pair was detected.

eTone Contains the enumerated value for the tone, or -1 when the end of the tone is detected.

RTP packetization

stRtpEncode

Performs RTP Packetization and, when used with the stJitterBuf service, depacketization.

The stRTP service accumulates a specified number of codec frames into a payload before wrapping them with an RTP header. This service also keeps statistics that an RTCP stack can use to generate Sender Reports.

Config data

RTP_HEADER_ST

```
typedef struct {
    RSYS_UINT32 version;
    RSYS_UINT32 p;
    RSYS_UINT32 x;
    RSYS_UINT32 cc;
    RSYS_UINT32 m;
    RSYS_UINT32 pt;
    RSYS_UINT32 seq;
    RSYS_UINT32 ts;
    RSYS_UINT32 ssrc;
    RSYS_UINT32 csrc[15];
} RTP_HEADER_ST;
```

version	Protocol version (2 bits).
p	Padding type (1 bit).
x	Header extension (1 bit).
cc	CSRC count (4 bits).
m	Marker bit (1 bit).
pt	Payload type (7 bits).
seq	Sequence number (16 bits).
ts	Time stamp.
ssrc	Synchronization source.
csrc	Optional CSRC list.

UP_RTP_SEND_CONFIG_ST

```
typedef struct {
    RTP_HEADER_ST stRtpSendHeader;
    RSYS_UINT32 ulPaddingLen;
    RSYS_UINT32 ulPayloadInterval;
    RSYS_UINT32 ulTimeElapsedForEachFrame;
    UP_RTP_SEND_CFG_FLAG_ET ulInitConfig;
} UP_RTP_SEND_CONFIG_ST;
```

stRtpSendHeader

The RTP sender header.

`ulPaddingLen`

Desired length of the RTP packet. Set the padding bit in stRtpHeader appropriately. You can select one of these:

0	No padding.
Other	The number of bits in the RTP packet.

`ulPayloadInterval`

The interval, in milliseconds, for payload duration. For example, if the `ulPayloadInterval` is 15ms, each RTP packet carries 15ms of voice. `ulPayloadInterval` must be chosen based on the codec in use. Frame-based codecs (G.723.1, G.729, etc.) require that the payload interval be an integer multiple of the codec frame size. Sample-based codecs (G.711) require that the payload interval be an integer number of milliseconds.

`ulTimeElapsedForEachFrame`

The frame duration of the codec in use, in samples. G.711, G.729, and G.729A encode 10ms frames, so this element should be set to 80. G.723 encodes 30ms frames, so this element should be set to 240.

`ulInitConfig`

Specifies whether the service is being setup for the first time within a call or whether it is a modification to a previously configured UP_RTP_SEND_CONFIG_ST.

For details regarding the RTP Header structure's content, see RFC 1899.

Events

There are no events associated with this service.

stRtpDecode

Implements both RTP decode functionality and a dynamic jitter buffer that feeds compressed frames to a codec for decompression on a strict schedule regardless of input variations.

The RTP decoder places the packet's payload into the proper position in the jitter buffer according to the RTP timestamp The jitter buffer can accommodate packet streams with missing, silence suppressed, and out-of-order packets.

The stRtpDecode service also keeps statistics that an RTCP stack can use to generate Receiver Reports and to control the jitter buffer's automatic size adjustment.

Config data

UP_RTP_RECV_CONFIG_ST

```
typedef struct {
   RSYS_UINT32 ulAutoAdjustable;
   RSYS_UINT32 ulMaxJitterBufferDly;/* Maximum jitter buf depth in ms*/
   RSYS_UINT32 ulTargetJitterBufferDly;/* Target jitter buffer depth in ms*/
   RSYS_UINT32 ulMaxFrameSizeInBytes;/* user expected max frame size*/
      /* in jitter buf*/
   RSYS_UINT32 ulExtractDataLength;/* Number of samples output*/
   RSYS_UINT32 ulInitConfig;/* Indicate init config or update config*/
} UP_RTP_RECV_CONFIG_ST;
```

ulAutoAdjustable

You can use one of these values:

FALSE — Controls the jitter buffer's target delay through the *ulTargetJitterBufferDly* element of the configuration structure.

TRUE — Overrides the *ulTargetJitterBufferDly* element and automatically controls target jitter buffer delay based on the jitter measurements of arriving packets.

Note: This element is not currently implemented. Jitter does not automatically adjust, regardless of the value you enter; the jitter buffer's target delay is always controlled through the *ulTargetJitterBufferDly* element of the configuration structure.

ulMaxJitterBufferPDly

The play-out jitter buffer's maximum length, in RTP packets. The jitter buffer induced delay can never exceed this number. The maximum usable value for this number depends on the memory allocated to the Jitter Buffer in the DSP application and the payload size of the codec in use; the standard maximum is 120ms of G.711 data.

ulTargetJitterBufferDly

The play-out jitter buffer's target length, in milliseconds. Over long time periods, the jitter buffer tries to average this many ms of delay, and so can cope with this many milliseconds of packet arrival time jitter. For best results this element should be set to an integer multiple of the codec frame size—30mS for G.723, and 10ms otherwise.

ulInitConfig

Specifies whether the service is being setup for the first time within a call or whether it is a modification to a previously configured UP_RTP_RECV_CONFIG_ST.

Events

UP_EVT_RTP_PT_CHANGE
UP_RTP_PT_CHANGE_DATA_ST

```
typedef struct {
   RSYS_UINT32 ulTimeStamp;
   RSYS_UINT32 ucNewPTValue;
} UP_RTP_PT_CHANGE_DATA_ST;
```

ulTimeStamp

Contains the RTP timestamp in the packet where the change was detected.

ucNewPTValue

Contains the new 6-bit PT value

UP_EVT_RTP_SSRC_CHANGE
UP_RTP_SSRC_CHANGE_DATA_ST

```
typedef struct {
   RSYS_UINT32 ulTimeStamp;
   RSYS_UINT32 ucNewSsrcValue;
} UP_RTP_SSRC_CHANGE_DATA_ST;
```

ulTimeStamp

Contains the RTP timestamp in the packet where the change was detected.

ucNewSsrcValue

Contains the new SSRC value.

Signaling

stCAS

Implements Channel Associated Signaling on T1 and E1 lines which are attached locally to framers control by the I/O Processor.

It allows control of two-bit SF or four-bit ESF CAS on the framer's transmission side and, if configured to do so, generates UP_EVT_CAS_CHANGE events whenever received bit values change.

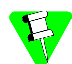
Config data

UP_CAS_CONFIG_ST

```
typedef struct {
    RSYS_INT32 lABCD;
    RSYS_INT32 lEventFlag;
} UP_CAS_CONFIG_ST;
```

lABCD The four-bit value to assign to the timeslot, with a in bit 3. In SF mode only a and b are used; b and c are ignored.

lEventFlag
Non-zero means generate an event each time the received bits for this channel change.



When upEnableService, upDisableService, or upConfigService is called for the stCas service, the *dsp* and *channel* arguments are used as framer unit number and timeslot number, respectively.

Events

UP_EVT_CAS_CHANGE
UP_CAS_CHANGE_DATA_ST

```
typedef struct {
    RSYS_UINT32 ulTimeStamp;
    RSYS_INT32 lABCD;
} UP_CAS_CHANGE_DATA_ST;
```

ulTimeStamp
Contains the microsecond timestamp when the change was detected. This timestamp value rolls over at 0x07C1F080.

lABCD Holds the ABCD bits' four-bit value with A in bit 3 to indicate the current CAS data for this channel. In SF framing only the A and B bits are valid.

stQDS0Hdlc

Concatenates four sub-rate HDLC channels into a full DS0.

DSP virtual channels 16 through 31 are reserved for Quarter DS0 support, with a maximum of 16 Quarter DS0 channels per DSP. The starting channel must be either 16, 20, 24, or 28. Enabling the first channel in this block concatenates that channel with the remaining three channels in the block, and output the results in the last channel of the block.

For example, the following concatenates channels 20, 21, 22, and 23 into channel 23:

```
UP_ERROR_ET upEnableService(lSlot, lUnit, 20, stQDS0Hdlc);
```

Channels 16 through 31 are reserved for Q-DS0 traffic to prevent the user from accidently interleaving voice channels with Q-DS0 channel groups. This service enforces the reservation and checks that the first channel within the grouping (‘ulChannel’) is a factor of 4 (e.g. 16, 20, 24, 28).

Config data

There are no data structures associated with this service.

Events

There are no events associated with this service.

Alarming

stEthernetAlarm

Detects Ethernet Link status changes and generates UP_EVT_ETHERNET_ALARM events, which notify a user application of the change.

A 'UP_EVT_ETHERNET_ALARM' event with a link status of 'enumEthernetUp' is emitted when the Ethernet Adapter is in an active state and detects a valid LAN connection. This state occurs when the Ethernet Adapter is first initialized when connected to a network or after initialization following reconnection to the network.

A 'UP_EVT_ETHERNET_ALARM' event with a link status of 'enumEthernetDown' is also emitted when the Ethernet Adapter becomes inactive or is disconnected from the network.

Config data

There are no data structures associated with this service.

Events

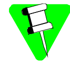
UP_EVT_ETHERNET_ALARM
UP_ETHERNET_ALARM_DATA_ST

A structure type passed in a UP_EVT_ETHERNET_ALARM event containing state change information associated with a specific adapter located on a specific IOP.

```
typedef enum {
   enumEthernetUp = 1,/* This ethernet link is available */
   enumEthernetDown/* This ethernet link is unavailable */
} UP_ETHERNET_ALARM_ET;

typedef struct {
   RSYS_UINT32 ulTimeStamp;
   UP_ETHERNET_ALARM_ET eEthernetState;
} UP_ETHERNET_ALARM_DATA_ST;
UP_EVT_ETHERNET_ALARM

enumEthernetUp


enumEthernetDown
```

stT1E1Alarm

Notifies the user of an alarm signaled by a T1 or E1 framer, and allows control of the LEDs associated with each span.



The user application must cause the LEDs to reflect the alarm status of the span.

Config data

UP_T1E1ALARM_CONFIG_ST

```
typedef enum {
    enumLedsOff= 0x05,/* both LEDs are off */
    enumLedsNormal= 0x07,/* one LED is off, other is green */
    enumLedsYellowAlarm= 0x04,/* one LED is off, other is yellow */
    enumLedsRedAlarm= 0x01,/* one LED is off, other is red */
    enumLedsLoopback= 0x0f,/* both LEDs are green */
    enumLedsLineFault= 0x00,/* one LED is red, other is yellow */
           /* Other possible combinations */
    enumLedsGrYellow= 0x0c,/* one LED is green, other is yellow */
    enumLedsRedGr= 0x03,/* one LED is red, other is green */
    enumLedsOffGr= 0x0d/* one LED is off, other is green */
} UP_T1E1_LED_STATE_ET;

typedef enum {
    enumSendNothing = 0,
    enumSendAIS,
    enumSendYellowAlm,
    enumSendLoopUp,
    enumSendLoopDown
} UP_T1E1_SEND_CMD_ET;

typedef struct {
    UP_T1E1_LED_STATE_ETtLedState;
    UP_T1E1_SEND_CMD_ETtSendCmd;
    RSYS_INT32lEventFlag;
} UP_T1E1ALARM_CONFIG_ST;
```

tLedState Controls the display of LEDs for this span.

tSendCmd Causes the specified state to be sent on the span. Set this to enumSendNothing for normal operation.

lEventFlag
Non-zero means generate a UP_EVT_T1E1_ALARM event each time the alarm status of this span changes.

Since the T1/E1 alarm status is per span rather than per channel, the `lChannel` argument is ignored when a function enables, disables, or configures this service, and has no meaning in a received alarm event message.

Events

UP_EVT_T1E1_ALARM
UP_T1E1_ALARM_DATA_ST

```
typedef struct {
    RSYS_UINT32 ulTimeStamp;
    RSYS_UINT32 ulStateWord;
    RSYS_UINT32 ulChangeWord;
} UP_T1E1_ALARM_DATA_ST;
```

ulTimeStamp

Contains the microsecond timestamp when the status change was detected. This timestamp value rolls over at 0x07C1F080.

ulStateWord

Indicates which bits of *ulStateWord* changed since the last notification.

ulChangeWord

May be interpreted bit by bit according to the following macros, provided in iop_to_t1.h:

```
#define T1_RX_AIS_BIT(1<<0)/* bit 0 of T1 status is Rx AIS state*/
#define T1_RX_YELLOW_BIT(1<<1)/* bit 1 of T1 status is Rx yellow*/
       /* alarm state*/
#define T1_RX_LOS_BIT(1<<2)/* bit 2 of T1 status is Rx loss of*/
       /* signal state*/
#define T1_RX_LOF_BIT(1<<3)/* bit 3 of T1 status is Rx loss of*/
       /* frame state*/

#define T1_TX_AIS_BIT(1<<16)/* bit 16 of T1 status indicates */
       /* AIS transmission*/
#define T1_TX_YELLOW_BIT(1<<17)/* bit 17 of T1 status indicates */
       /* yellow alm transmission */
#define T1_TX_LOOP_UP_BIT(1<<18)/* bit 18 of T1 status indicates */
       /* loop up code transmission */
#define T1_TX_LOOP_DOWN_BIT(1<<19)/* bit 19 of T1 status indicates */
       /* loop down code transmission */
```

Audio processing

stAGC

Provides Automatic Gain Control for TDM input data.

It normalizes input data to the level specified in the service configuration.

Config data

UP_AGC_CONFIG_ST

```
typedef struct {
   RSYS_INT32 lTargetValdB;/* AGCTargetvalue in dB*/
   RSYS_INT32 lInScaleFact;/* absolute scalefactor for input*/
   RSYS_INT32 lOutScaleFact;/* absolute scalefactor for output*/
} UP_AGC_CONFIG_ST;
```

Events

There are no events associated with this service.

Internal

These are services which are implemented in one part of the system for the internal use of other parts of the system.

stPacketBuilder

This is an embedded service controlled by the upConnectPktSend function.

It takes the output of the stT38 or stRTP service, adds UDP, IP, and Ethernet headers, and submits the result to the DSP's Ethernet Transmit Queue.

Config data

UP_PACKET_BUILDER_CONFIG_ST

```
typedef struct {
    RSYS_UINT32 ulDestAddress;
    RSYS_UINT32 ulDestPort;
    RSYS_UINT32 ulSrcAddress;
    RSYS_UINT32 ulSrcPort;
    RSYS_UINT32 ulRouterEtherAddr[2];
    RSYS_UINT32 ulDestEtherAddr[2];
    RSYS_UINT32 ulServiceType;
} UP_PACKET_BUILDER_CONFIG_ST;
```

Elements

ulDestAddress
The destination IP address.

ulDestPort The destination UDP port.

ulSrcAddress
The source IP address.

ulSrcPort The source UDP port.

ulRouterEtherAddr
The router Ethernet address.

ulDestEtherAddr
The destination Ethernet address.

ulServiceType;
A bit that indicates priority, delay, thruput and reliability.

This structure is internally used by UPA on the IOP to configure the stPacketBuilder on a particular DSP channel. This structure is not directly accessed by the user but is populated with the contents of a UP_PKT_SEND_CONFIG_ST passed as an argument to the upConnectPktSend function.

Events

There are no events associated with this service.

stPacketParser

Takes an IP/UDP packet from the *Ethernet->DSP* FIFO, checksums headers and data as appropriate, strips headers, and passes the payload to the next module, usually stRtpDecode or stT38, in TASK2.

This is an embedded service controlled by the upConnectPktRecv function. Generally, user applications should not directly access this service.

Config data

UP_PACKET_PARSER_CONFIG_ST

```
typedef struct {
   RSYS_UINT32 ulReceivePort;
   RSYS_UINT32 ulInterface;
   UP_SERVICE_ET eService;
} UP_PACKET_PARSER_CONFIG_ST;
```

This structure is internally used by UPA on the IOP to configure the stPacketParser on a particular DSP channel. This structure is not directly accessed by the user but is populated with the contents of a UP_PKT_RECV_CONFIG_ST passed as an argument to the upConnectPktRecv function.

Events

There are no events associated with this service.

Glossary

Address A number that identifies the location of a word in memory. Each word in a memory storage device or system has a unique address.

AGC (Automatic Gain Control) An electronic circuit or software algorithm used to maintain signal level.

ANSI (American National Standards Institute) An organization dedicated to advancement of national standards related to product manufacturing.

BFM (Bad Frame Masking) A software algorithm that hides transmission losses in a packetized voice communication system where the input signal is encoded and packetized at a transmitter, sent over a network, and received at a receiver that decodes the packet and plays out the output.

BIOS Basic Input/Output System) Firmware in a PC-compatible computer that runs when the computer is powered up. The BIOS initializes the computer hardware, allows the user to configure the hardware, boots the operating system, and provides standard mechanisms that the operating system can use to access the PC's peripheral devices.

Bit A binary digit.

Boot The process of starting a microprocessor and loading the operating system from a powered down state (cold boot) or after a computer reset (warm boot). Before the operating system loads, the computer performs a general hardware initialization and resets internal registers.

Boot Device The storage device from which the computer boots the operating system.

Byte A group of 8 bits.

CAS

1. (Channel Associated Signaling) Repetitively sending one or more bits of signaling status associated with the specified circuit to indicate circuit state.
2. (Column Address Strobe) An input signal from the DRAM controller to an internal DRAM latch register specifying the column at which to read or write data. The DRAM requires a column address and a row address to define a memory address. Since both parts of the address are applied at the same DRAM inputs, use of column addresses and row addresses in a multiplexed array allows use of half as many pins to define an address location in a DRAM device as would otherwise be required.

CNG (Comfort Noise Generator) A software algorithm that creates a background audio signal to replace the silence created by some software CODECs during the time no speech data is transmitted over the telephone line.

CODEC (COder/DECoder) Converts voice signals from analog form to digital signals acceptable to modern digital PBXs and digital transmission systems. It then converts

those digital signals back to analog so that you can hear and understand what the other person is saying.

COFF (Common Object File Format)

CPT (Call Progress Tones) Call Progress is a tonal-signaling standard used to acquire connections between subscribers in telephone network systems. Call Progress Tones indicate a call's status. CPT signals consist of single frequency and dual frequency combinations of sinusoidal signals with specific ON-OFF patterns.

CPU (Central Processing Unit) A semiconductor device which performs the processing of data in a computer. The CPU, also referred to as the microprocessor, consists of an arithmetic/logic unit to perform the data processing, and a control unit which provides timing and control signals necessary to execute instructions in a program.

COM Port A bi-directional serial communication port which implements the RS-232 specification.

CSMA/CD (Carrier-Sense Multiple Access with Collision Detect) A method whereby workstations on a network listen for transmission in progress (carrier sense) before starting to transmit (multiple access). If two or more workstations transmit at the same time, each workstation stops transmitting (collision detection) for a different amount of time before trying to transmit again.

CSU (Channel Service Unit) A device to terminate a digital channel on a customer's premises. It performs certain line coding, line-conditioning and equalization functions, and responds to loopback commands sent from the central office.

CT bus (Computer Telephony Bus) An auxiliary bus used in computer systems. This bus is dedicated to carrying telecom data between the system components.

Default The state of all user-changeable hardware and software settings as they are originally configured before any changes are made.

Driver A software component of the operating system which directs the computer interface with a hardware device. The software interface to the driver is standardized such that application software calling the driver requires no specific operational information about the hardware device.

DRAM. (Dynamic Random Access Memory) Semiconductor RAM memory devices in which the stored data does not remain permanently stored, even with the power applied, unless the data are periodically rewritten into memory during a refresh operation.

DSP (Digital Signal Processor) A high-speed computer chip that performs real-time signal manipulation. DSPs are used extensively in telecommunications for tasks such as echo cancellation, audio and video processing.

DSX (Digital System Cross-connect Frame) A bay or panel to which T-1 lines and DS1 circuit packs are wired and that permits cross-connections by patch cords and plugs. A DSX panel is used in small office applications where only a few digital trunks are installed.

DSX-1 (Digital Signal Cross-connect Level 1) The set of parameters for cross connecting DS-1 lines.

DTMF (Dual-Tone Multi-Frequency) Push button or Touchtone dialing, where touching a button on a push button pad makes two tones, one high frequency and one low frequency.

EEPROM (Electrically Erasable Programmable ROM) EPROMs that can be erased electrically as compared to other erasing methods.

EDO (Extended Data Out) A type of DRAM that allows higher memory system performance since the data pins are still driven when CAS# is de-asserted. This allows the next DRAM address to be presented to the device sooner than with Fast Page Mode DRAM.

ESF (Extended Super Frame or Extended Superframe Format) A T-1 format that uses the 193rd bit as a framing bit. ESF provides frame synchronization, cyclic redundancy checking and data link bits. Frames consist of 24 bits instead of the previous standard 12 bits as in the D4 format. The standard allows error information to be stored and retrieved easily, facilitating network performance monitoring and maintenance.

Flash Memory A fast EEPROM semiconductor memory typically used to store firmware such as the computer BIOS. Flash memory also finds general application where a semiconductor non-volatile storage device is required.

Flash Recovery A process whereby an existing, corrupt BIOS image in the flash boot device is overwritten with a new image. Also referred to as a flash recovery.

Flash Update A process whereby an existing, uncorrupted BIOS image in the flash boot device is overwritten with a new image. Also referred to as a flash update.

FPM (Fast Page Mode) A “standard” type of DRAM that is lower performance than EDO.

FPGA (Field Programmable Gate Array) A large, general-purpose logic device that is programmed at power-up to perform specific logic functions.

Framers A device used in digital communication systems to create parellel data frames from a serial data stream.

GB or GByte (Gigabyte) Approximately one billion (US) or one thousand million (Great Britain) bytes. 2^30 = 1,073,741,824 bytes exactly.

Hang A condition where the system microprocessor suspends processing operations due to an anomaly in the data or an illegal instruction.

HDLC (High Level Data Link Control) An ITU-TSS link layer protocol standard for point-to-point and multi-point communications.

Header A mechanical pin and sleeve style connector on a circuit board. The header may exist in either a male or female configuration. For example, a male header has a number and pattern of pins which corresponds to the number and pattern of sleeves on a female header plug.

h (Hexadecimal) A base 16 numbering system using numeric symbols 0 through 9 plus alpha characters A, B, C, D, E, and F as the 16 digit symbols. Digits A through F are equivalent to the decimal values 10 through 15.

IMI (Initial Memory Image) A structure in memory the i960 core requires to initialize internal registers before normal operation.

INT	(Interrupt Request) A software-generated interrupt request.
IOP	(Input/Output Processor) A processing element on intelligent adapter boards used to off-load a host processor.
I/O	(Input/Output) The communication interface between system components and between the system and connected peripherals.
IRQ	(Interrupt Request). In ISAbus systems, a microprocessor input from the control bus used by I/O devices to interrupt execution of the current program and cause the microprocessor to jump to a special program called the interrupt service routine. The microprocessor executes this special program, which normally involves servicing the interrupting device. When the interrupt service routine is completed, the microprocessor resumes execution of the program it was working on before the interruption occurred.
ISR	(Interrupt Service Routine) A program executed by the microprocessor upon receipt of an interrupt request from an I/O device and containing instructions for servicing of the device.
Jumper	A set of male connector pins on a circuit board over which can be placed coupling devices to electrically connect pairs of the pins. By electrically connecting different pins, a circuit board can be configured to function in predictable ways to suit different applications.
KB or KByte	(Kilobyte) Approximately one thousand bytes. 2^{10} = 1024 bytes exactly.
LIU	(Line Interface Unit)
Logical Address	The memory-mapped location of a segment after application of the address offset to the physical address.
MAC	(Media Access Controller) A media-specific access control protocol within IEEE 802 specifications for the lower half of the data link layer (layer 2) that defines topology dependent access control protocols for IEEE LAN specifications.
MB or MByte	(Megabyte) Approximately one million bytes. 2^20 = 1,048,576 bytes exactly.
Memory	A designated system area to which data can be stored and from which data can be retrieved. A typical computer system has more than one memory area.
Memory shadowing	Copying information from an extension ROM into DRAM and accessing it in this alternate memory location.
Offset	The difference in location of memory-mapped data between the physical address and the logical address.
PAL	(Programmable Array Logic) A semiconductor programmable ROM which accepts customized logic gate programming to produce a desired sum-of-products output function.
PCI	(Peripheral Component Interconnect) A popular microcomputer bus standard used extensively in personal computer architecture. This 32-bit local bus, used inside PCs, was designed by Intel.
Peripheral Device	An external device connected to the system for the purpose of transferring data into or out of the system.

PHY (Physical Layer) An ATM layer whose functionality loosely corresponds to the OSI physical layer (Layer 1). ATM Physical Layer functionality includes the Physical Medium sublayer (PM) and the Transmission Convergence (TC) sublayer.

PLL (Phase-Locked Loop) A semiconductor device which functions as an electronic feedback control system to maintain a closely regulated output frequency from an unregulated input frequency. The typical PLL consists of an internal phase comparator or detector, a low pass filter, and a voltage controlled oscillator which function together to capture and lock onto an input frequency. When locked onto the input frequency, the PLL can maintain a stable, regulated output frequency (within bounds) despite frequency variance at the input.

Physical Address The address or location in memory where data is stored before it is moved as memory remapping occurs. The physical address is that which appears on the computer's address bus when the CPU requests data from a memory address. When remapping occurs, the data can be moved to a different memory location or logical address.

Pinout A diagram or table describing the location and function of pins on an electrical connector.

PMC (PCI Mezzanine Card) A new standard form factor for PCI add-in modules. PMCs mate with their respective connectors on the motherboard and are secured with screws.

POST (Power On Self Test) A diagnostic routine which a board runs at power up. Along with other testing functions, this comprehensive test initializes the system chipset and hardware, resets registers and flags, performs ROM checksums, and checks disk drive devices and the keyboard interface.

PQFP (Plastic Quad Flat Pack) A popular package design for integrated circuits of high complexity.

Program A set of instructions a computer follows to perform specific functions relative to user need or system requirements. In a broad sense, a program is also referred to as a software application, which can actually contain many related, individual programs.

PSTN (Public Switched Telephone Network) The worldwide voice telephone networks and services accessible to all who have telephones and access privileges.

R1/R2 Multi-frequency signaling for PSTN trunk lines. R1/R2 signaling consists of register signaling for address signals, and line signaling for line and supervisory signals.

RAM (Random Access Memory) Memory in which the actual physical location of a memory word has no effect on how long it takes to read from or write to that location. In other words, the access time is the same for any address in memory. Most semiconductor memories are RAM.

ROM (Read Only Memory) A broad class of semiconductor memories designed for applications where the ratio of read operations to write operations is very high. Technically, a ROM can be written to (programmed) only once, and this operation is normally performed at the factory. Thereafter, information can be read from the memory indefinitely.

Reflashing The process of replacing a BIOS image, in binary format, in the flash boot device.

Register An area typically inside the microprocessor where data, addresses, instruction codes, and information on the status on various microprocessor operations are stored. Different types of registers store different types of information.

Reset A signal delivered to the microprocessor by the control bus, which causes a halt to internal processing and resets most CPU registers to 0. The CPU then jumps to a starting address vector to begin the boot process.

RS-232 A popular asynchronous bi-directional serial communication protocol. Among other things, the RS-232 standard defines the interface cabling and electrical characteristics, and the pin arrangement for cable connectors.

RAS (Row Address Strobe) An input signal to an internal DRAM latch register specifying the row at which to read or write data. The DRAM requires a row address and a column address to define a memory address. Since both parts of the address are applied at the same DRAM inputs, use of row addresses and column addresses in a multiplexed array allows use of half as many pins to define an address location in a DRAM device as would otherwise be required.

RTOS (Real Time Operating System) An operating system that performs tasks at specified times.

Sample rate generator Software or firmware that the number of times per second that an analog signal is measured and converted to a binary number -- the purpose being to convert the analog signal to a digital analog. The most common digital signal—PCM—samples voice 8,000 times a minute.

SBSRAM Synchronous Burst Static Random Access Memory.

Serial Port A physical connection with a computer for the purpose of serial data exchange with a peripheral device. The port requires an I/O address, a dedicated IRQ line, and a name to identify the physical connection and establish serial communication between the computer and a connected hardware device. A serial port is often referred to as a COM port.

SF (Super Frame) A DS1 framing format in which 24 DSO timeslots plus a coded framing bit are organized into a frame which is repeated 12 times to form the ssuperframe.

Shadow Memory RAM in the address range 0xC000h through 0xFFFFFh used for shadowing. Shadowing is the process of copying BIOS extensions from ROM into DRAM for the purpose of faster CPU access to the extensions when the system requires frequent BIOS calls. Typically, system and video BIOS extensions are shadowed in DRAM to increase system performance.

SODIMM (Small Outline Dual Inline Memory Module) A new form factor for memory modules that is smaller and denser than SIMMs.

Standoff A mechanical device, typically constructed of an electrically non-conductive material, used to fasten a circuit board to the bottom, top, or side of a protective enclosure.

SRAM (Static Random Access Memory) A semiconductor RAM device in which the data remains permanently stored as long as power is applied, without the need for periodically rewriting the data into memory.

Symmetrically Addressable SIMM A SIMM, the memory content of which is configured as two independent banks. Each 16-bit wide bank contains an equal number of rows and columns and is independently addressable by the CPU via twin row address strobe registers in the DRAM controller.

TDM (Time Division Multiplex) A technique originated in satellite communications to interweave multiple conversations into one transponder so as to appear to get simultaneous conversations. This technicque is used on the H.110 bus and the internal DSP serial buses.

TSI Time-slot interchange. A way of temporarily storing data bytes so they can be sent in a different order than they were received. TSI is a way to switch calls.

UART (Universal Asynchronous Receiver/Transmitter) A device, usually an integrated circuit chip, that converts digital data to transmit from parallel to serial, and transmitted digital data from serial to parallel. The UART converts incoming serial data from the device connected to the serial port (typically a modem) into the parallel form that your computer handles. UART also converts the computer's parallel data into serial data suitable for asynchronous transmission on phone lines.

VAD (Voice Activity Detector) A signal classifier used to distinguish between active voice and inactive voice (silence and background noise).

VoIP (Voice over Internet Protocol) A protocol that enables devices of disparate manufactures to support voice communication over packet networks such as the Internet.

Wait State A period of one or more microprocessor clock pulses during which the CPU suspends processing while waiting for data to be transferred to or from the system data or address buses.

Index

A

addresses
- defined *285*
- logical, defined *288*
- physical, defined *289*

ANSI, defined *285*
API *7*
application distribution
- Host and IOP *2*
- IOP *1*

Application Programming Interface (API) *7*
applications
- DSP *1*, *6*
- Host
  - initializing the Host driver *30*
  - initializing UPA structures *30*
  - loading and running applications *30*
  - sample code *36*
  - setting up message handlers *30*
- IOP
  - configuring services *32*
  - creating data paths *32*
  - initializing the IOP driver *31*
  - sample code *39*
  - setting up message handlers *31*

B

BIOS, defined *285*
boot device, defined *285*
broadcasting *232*

C

callback functions *11*
channels
- defined *7*
- groups *8*
- physical *8*
- virtual *8*

clearT8100ClockFault *241*
clearT8100MemoryFault *242*
command messages *10*
communication, inter-processor *10*
components, TASK *3*
configuring services *32*
connections, making and breaking *231*
conventions, notational *ii*
creating data paths *32*

D

data path services *9*
data structures, HDLC *148*
developing
- Host applications *30*
- IOP applications *31*

device drivers
- libraries, peripheral *5*
- NT Kernel Mode *4*

dispatchers
- event *5*
- message *4*

driver sequence, sample *150*
driver, defined *286*
DSP application *6*
DSP applications *1*
Dynamic Random Access Memory (DRAM), defined *286*

E

E1/T1 functions
- E1 and T1
  - T1E1getBoardConfig *184*
  - T1E1initCard *183*
  - T1E1setLeds *185*
- E1 only
  - getE1Signaling *197*
  - setE1Config *192*
  - setE1Signaling *193*
  - setE1SignalingHandler *200*
- T1 only
  - getT1Signaling *194*
  - getT1SignalingRaw *196*
  - getT1Status *195*
  - setT1ChannelConfig *191*
  - setT1ClearChannels *189*
  - setT1Command *188*
  - setT1Config *186*
  - setT1IdleChannels *190*
  - setT1Signaling *187*
  - setT1SignalingHandler *198*

setT1StatusHandler 199
E1/T1 library 5
function list 179
sample startup sequence 178
structures 201
E1/T1 structures
E1 only
E1SignalingHandler 229
t_E1_line_buildout 225
t_E1_line_coding 223
t_E1_signaling_mode 224
t_E1_user_config_struct 226
t_E1_user_signaling_data 228
T1 and E1
t_T1E1_BoardConfig 207
t_T1E1_card_type 204
t_T1E1_framer_id 203
t_T1E1_led_state 205
t_T1E1_user_signaling_data 206
T1 only
t_T1_framing_mode 209
t_T1_line_buildout 210
t_T1_line_coding 208
t_T1_signaling_data 213
t_T1_user_channel_config 218
t_T1_user_clear_channel_data 216
t_T1_user_command_data 214
t_T1_user_config_struct 211
t_T1_user_idle_struct 217
t_T1_user_raw_signaling_struct 220
t_T1_user_signaling_data 213
t_T1_user_status_struct 215
T1SignalingHandler 222
T1StatusHandler 221
E1SignalingHandler 229
EDO DRAMs, defined 287
e-mail address, RadiSys ii
event dispatcher 5
event messages 10

F

fast packet router 5
Fast Page Mode DRAMs, defined 287
Flash
recovery, defined 287
functions
callback 11
E1/T1 function list 179
HDLC function list 151
Host function list 48
T8100 function list 235

G

getE1Signaling 197
getT1Signaling 194
getT1SignalingRaw 196
getT1Status 195
getT8100ErrorStatus 244
glossary 285

H

hbus.sys 4
HDLC functions
HDLCCloseDriver 155
HDLCClosePort 156
HDLCConfigChannel 158
HDLCConfigPort 157
HDLCDisableChannel 160
HDLCEnableChannel 159
HDLCGetChannelStatus 165
HDLCGetDeviceStatus 164
HDLCGetPacket 163
HDLCInit 153
HDLCReset 154
HDLCResetChannel 161
HDLCSendPacket 162
HDLCSetDeviceErrorHandler 168
HDLCSetRxErrorHandler 170
HDLCSetRxPacketHandler 167
HDLCSetTxErrorHandler 169
HDLCSetTxPacketHandler 166
HDLC library 5
data structures 148
function list 151
overview 147
processing modes 148
processing packet transmission and reception 149
sample driver sequence 150
structure list 172
type definitions 171
HDLC structures
t_HDLC_channel_config 175
t_HDLC_channel_status 176
t_HDLC_port_config 173
HDLCCloseDriver 155
HDLCClosePort 156
HDLCConfigChannel 158
HDLCConfigPort 157
HDLCDisableChannel 160
HDLCEnableChannel 159
HDLCGetChannelStatus 165
HDLCGetDeviceStatus 164
HDLCGetPacket 163
HDLCInit 153

HDLCReset *154*
HDLCResetChannel *161*
HDLCSendPacket *162*
HDLCSetDeviceErrorHandler *168*
HDLCSetRxErrorHandler *170*
HDLCSetRxPacketHandler *167*
HDLCSetTxErrorHandler *169*
HDLCSetTxPacketHandler *166*
header, defined *287*
help *ii*
Host and IOP application distribution *2*
Host applications, developing *30*
  initializing the Host driver *30*
  initializing UPA structures *30*
  loading and running applications *30*
  setting up message handlers *30*
Host applications,sample code *36*
Host files
  Hot Swap
    hbus.sys *4*
    Host library *4*
    hsmgrint.dll *4*
  i960rp.sys *4*
  taskhost.dll *3*
host functions
  function list *48*
  hostControlPeripheral *50*
  hostExit *54*
  hostGetBoardInfo *55*
  hostGetNWPktBuf *94*
  hostGetSystemInfo *57*
  hostInit *59*
  hostJitterControl *95*
  hostLoadDsp *60*
  hostLoadIop *61*
  hostReadIop *96*
  hostResetBoard *62*
  hostResetDsp *63*
  hostRunDsp *64*
  hostRunLoadedIop *65*
  hostSendMsg *98*
  hostSendNWPktBuf *97*
  hostSetEventHandler *66*
  hostSetHotSwapHandler *67*
  hostSetNWNotify *99*
  hostSetPeripheralDataHandler *68*
  hostWriteIop *100*
  upConfigService *69*
  upConfigServiceGlobal *72*
  upConnectPktRecv *75*
  upConnectPktSend *77*
  upDisableService *79*
  upDisconnectPktRecv *82*
  upDisconnectPktSend *83*
  upEnableChannel *84*
  upEnableService *86*
  upQueryQOSReport *89*
  upSetEventHandler *90*
  upSetUserMsgHandler *91*
  upStart *92*
Host runtime library *3*
host structures
  REPORTBOARDINFO *57*
  SP6K_BOARD_INFO_T *55*
  t_configArg *53*
  t_T8100StreamConfig *52*
  t_T8100SwitchConfig *50*
  UP_CONFIG_SVC_UT *70*
  UP_GLOBALCONFIGDATA_UT *73*
  UP_PKT_RECV_CONFIG_ST *75*
hostControlPeripheral *50*
hostExit *54*
hostGetBoardInfo *55*
hostGetNWPktBuf *94*
hostGetSystemInfo *57*
hostInit *59*
hostJitterControl *95*
hostLoadDsp *60*
hostLoadIop *61*
hostReadIop *96*
hostResetBoard *62*
hostResetDsp *63*
hostRunDsp *64*
hostRunLoadedIop *65*
hostSendMsg *98*
hostSendNWPktBuf *97*
hostSetEventHandler *66*
hostSetHotSwapHandler *67*
hostSetNWNotify *99*
hostSetPeripheralDataHandler *68*
hostWriteIop *100*
Hot Swap *4*
  hbus.sys file *4*
  hsmgrint.dll file *4*
Hot Swap files
  hsmgrint.dll *4*
hsmgrint.dll *4*

I

i960rp.sys *4*
initialize the Host driver *30*
initialize the IOP driver *31*
initializing UPA structures *30*
initT8100 *237*

interfaces *6*
  Application Programming Interface (API) *7*
  Internet Protocol (IP) *6*
  PSTN *7*
Internet Protocol *6*
inter-processor communication *10*
interrupt
  request (IRQ), defined *288*
IOP application distribution *1*
IOP applications, developing
  initializing the IOP driver *31*
  setting up message handlers *31*
IOP applications,configuring services *32*
IOP applications,creating data paths *32*
IOP applications,sample code *39*
IOP functions
  iopControlPeripheral *108*
  iopGetNWPktBuf *141*
  iopInit *112*
  iopJitterControl *142*
  iopSendMsg *144*
  iopSendNWPktBuf *143*
  iopSetNWNotify *145*
  upConfigService *113*
  upConfigServiceGlobal *116*
  upConnectPktRecv *119*
  upConnectPktSend *121*
  upDisableService *123*
  upDisconnectPktRecv *126*
  upDisconnectPktSend *127*
  upEnableChannel *128*
  upEnableService *130*
  upQueryQOSReport *133*
  upSetEventHandler *136*
  upSetUserMsgHandler *146*
  upStart *139*
IOP runtime library *4*
IOP structures
  SP6K_BOARD_INFO_T *106*
  t_configArg *111*
  t_jitterParam *142*
  t_T8100ClockConfig *109*
  t_T8100StreamConfig *110*
  t_T8100SwitchConfig *108*
  UP_CONFIG_SVC_UT *114*
  UP_EVENT_DATA_ST *136*
  UP_GLOBALCONFIGDATA_UT *117*
  UP_IOPSYSCONFIG_ST *139*
  UP_PKT_RECV_CONFIG_ST *119*
  UP_PKT_SEND_CONFIG_ST *121*
  UP_START_DSP_REPLY_ST *138*
IopAppLoader.c *30*, *36*
iopControlPeripheral *108*
iopGetNWPktBuf *141*
iopInit *112*
iopJitterControl *142*
iopSendMsg *144*
iopSendNWPktBuf *143*
iopSetNWNotify *145*
IP *6*

J

jumpers
  defined *288*

L

libraries
  E1/T1 *5*
  HDLC *5*
  Hot Swap Host *4*
  peripheral device driver *5*
  TDM switch *5*
loading and running applications *30*
logical address, defined *288*

M

memory
  random access, defined *289*
message dispatcher *4*
messages
  callback functions *11*
  command *10*
  event *10*

N

notational conventions *ii*
NT Kernel Mode device driver *4*

O

objects *7*
  channels *7*
  slot *7*
  unit *7*
offset, defined *288*
operating system, defined *290*

P

packets
  fast packet router *5*
  transmission and reception, HDLC *149*
peripheral device driver libraries *5*
physical address, defined *289*
physical channels *8*
POST *289*
Power-On Self Test (POST), defined *289*

processing modes, HDLC *148*
protocols
  Internet Protocol (IP) *6*
PSTN *7*
Public Switch Telephone Network (PSTN) *7*

R

RadiSys, contacting *ii*
RAM, defined *289*
Random Access Memory (RAM), defined *289*
README file *ii*
reflashing, defined *289*
REPORTBOARDINFO structure *57*
requests
  IRQ, defined *288*
reset, defined *290*
rmondb.exe *6*
router, fast packet *5*
runtime libraries
  Host *3*
  IOP *4*

S

sample code
  Host application *36*
  IOP application *39*
services *8*
  data path *9*
  stAGC *281*
  stCodec *260*
  stCPTDet *268*
  stEchoCanc *263*
  stMFDet *270*
  stPacketBuilder *282*
  stPacketParser *283*
  stPktDTMFDet *267*
  stPktToneGen *265*
  stRtpDecode *274*
  stRtpEncode *272*
  stT1E1Alarm *279*
  stTdmDTMFDet *266*
  stTdmToneGen *264*
setE1Config *192*
setE1Signaling *193*
setE1SignalingHandler *200*
setT1ChannelConfig *191*
setT1ClearChannels *189*
setT1Command *188*
setT1Config *186*
setT1IdleChannels *190*
setT1Signaling *187*
setT1SignalingHandler *198*
setT1StatusHandler *199*
setT8100ClockConfig *238*
setT8100ClockFaultMask *243*
setT8100Handler *245*
setT8100StreamConfig *239*
setT8100SwitchConfig *240*
setting up message handlers on an IOP *31*
setting up message handlers on the Host *30*
SIMMs
  symmetrically addressable, defined *291*
slot, defined *7*
SP6K_BOARD_INFO_T structure *55*, *106*
sp6k_util.exe *6*
stAGC *281*
startup sequence, E1/T1 *178*
startup sequence, T8100 *234*
stCodec *260*
stCPTDet *268*
stEchoCanc *263*
stMFDet *270*
stPacketBuilder *282*
stPacketParser *283*
stPktDTMFDet *267*
stPktToneGen *265*
stRtpDecode *274*
stRtpEncode *272*
structure list, HDLC *172*
structures
  E1/T1 *201*
  E1SignalingHandler *229*
  REPORTBOARDINFO *57*
  SP6K_BOARD_INFO_T *55*, *106*
  t_configArg *53*, *111*
  t_E1_line_buildout *225*
  t_E1_line_coding *223*
  t_E1_signaling_mode *224*
  t_E1_user_config_struct *226*
  t_E1_user_signaling_data *228*
  t_fallback_clk *248*
  t_jitterParam structure *142*
  t_netref_clk *249*
  t_ref_clk *247*
  t_source_dest *256*
  t_stream_rate *252*
  t_T1_framing_mode *209*
  t_T1_line_buildout *210*
  t_T1_line_coding *208*
  t_T1_signaling_data *213*
  t_T1_user_channel_config *218*
  t_T1_user_clear_channel_data *216*
  t_T1_user_command_data *214*
  t_T1_user_config_struct *211*
  t_T1_user_idle_struct *217*
  t_T1_user_raw_signaling_struct *220*

t_T1_user_signaling_data 213
t_T1_user_status_struct 215
t_T1E1_BoardConfig 207
t_T1E1_card_type 204
t_T1E1_framer_id 203
t_T1E1_led_state 205
t_T1E1_user_signaling_data 206
t_T8100ClockConfig 51, 109, 250
t_T8100Connection 257
t_T8100StreamConfig 52, 110, 253
t_T8100SwitchConfig 50, 108, 258
T1SignalingHandler 222
T1StatusHandler 221
T8100 structure list 246
UP_CONFIG_SVC_UT 70, 114
UP_EVENT_DATA_ST 136
UP_GLOBALCONFIGDATA_UT 73, 117
UP_IOPSYSCONFIG_ST 139
UP_PKT_RECV_CONFIG_ST 75, 119
UP_PKT_SEND_CONFIG_ST 121
UP_START_DSP_REPLY_ST 138
stT1E1Alarm 279
stTdmDTMFDet 266
stTdmToneGen 264
support ii
Symmetrically Addressable SIMM, defined 291

T

t_configArg structure 53, 111
t_E1_line_buildout 225
t_E1_line_coding 223
t_E1_signaling_mode 224
t_E1_user_config_struct 226
t_E1_user_signaling_data 228
t_HDLC_channel_config 175
t_HDLC_channel_status 176
t_HDLC_port_config 173
t_jitterParam structure 142
t_T1_framing_mode 209
t_T1_line_buildout 210
t_T1_line_coding 208
t_T1_signaling_data 213
t_T1_user_channel_config 218
t_T1_user_clear_channel_data 216
t_T1_user_command_data 214
t_T1_user_config_struct 211
t_T1_user_idle_struct 217
t_T1_user_raw_signaling_struct 220
t_T1_user_signaling_data 213
t_T1_user_status_struct 215
t_T1E1_BoardConfig 207
t_T1E1_card_type 204
t_T1E1_framer_id 203
t_T1E1_led_state 205
t_T1E1_user_signaling_data 206
t_T8100ClockConfig structure 51, 109
t_T8100StreamConfig structure 52, 110
t_T8100SwitchConfig structure 50, 108
T1E1getBoardConfig 184
T1E1initCard 183
T1E1setLeds 185
T1SignalingHandler 222
T1StatusHandler 221
T8100 functions
  clearT8100ClockFault 241
  clearT8100MemoryFault 242
  getT8100ErrorStatus 244
  initT8100 237
  setT8100ClockConfig 238
  setT8100ClockFaultMask 243
  setT8100Handler 245
  setT8100StreamConfig 239
  setT8100SwitchConfig 240
T8100 library
  broadcasting 232
  function list 235
  making and breaking connections 231
  sample startup sequence 234
T8100 structure list 246
T8100 structures
  t_fallback_clk 248
  t_netref_clk 249
  t_ref_clk 247
  t_source_dest 256
  t_stream_rate 252
  t_T8100ClockConfig 250
  t_T8100Connection 257
  t_T8100StreamConfig 253
  t_T8100SwitchConfig 258
TASK
  application distribution, Host and IOP 2
  application distribution, IOP 1
  interfaces 6
  objects 7
  runtime libraries
    Host 3
    IOP 4
  software components 3
taskhost.dll 3
TDM switch library 5
technical support ii
Tornado 4
troubleshooting ii
type definitions, HDLC 171

U

unit, defined *7*
UP_CONFIG_SVC_UT structure *70*, *114*
UP_EVENT_DATA_ST structure *136*
UP_GLOBALCONFIGDATA_UT structure *73*, *117*
UP_IOPSYSCONFIG_ST structure *139*
UP_PKT_RECV_CONFIG_ST structure *75*, *119*
UP_PKT_SEND_CONFIG_ST structure *121*
UP_START_DSP_REPLY_ST structure *138*
UpaIopApp.c *31*, *39*
upConfigService *69*, *113*
upConfigServiceGlobal *72*, *116*
upConnectPktRecv *75*, *119*
upConnectPktSend *77*, *121*
upDisableService *79*, *123*
upDisconnectPktRecv *82*, *126*
upDisconnectPktSend *83*, *127*
upEnableChannel *84*, *128*
upEnableService *86*, *130*
upQueryQOSReport *89*, *133*
upSetEventHandler *90*, *136*
upSetUserMsgHandler *91*, *146*
upStart *92*, *139*
URL, RadiSys *ii*, *iv*
utilities
    rmondb.exe *6*
    sp6k_util.exe *6*

V

virtual channels *8*
VxWorks *4*

W

World-Wide Web URLs
    RadiSys *iv*
World-Wide Web, accessing RadiSys *ii*