



AltaAPI™

Software User's Manual



Part Number: 14301-00000-J7
Cage Code: 4RK27 • NAICS: 334118

Alta Data Technologies LLC
4901 Rockaway Blvd, Building A
Rio Rancho, NM 87124 USA
(tel) 505-994-3111 • www.altadt.com

CUSTOMER NOTES:

Document Information:

Alta Software Version: 3.3.0.0

Rev J6 Release Date: January 16, 2019

Note to the Reader and End-User:

This document is provided for information only and is copyright by © Alta Data Technologies LLC. While Alta strives to provide the most accurate information, there may be errors and omissions in this document. Alta disclaims all liability in document errors and any product usage. By using an Alta product, the customer or end user agrees (1) to accept Alta's Standard Terms and Conditions of Sale, Standard Warranty and Software License and (2) to not hold Alta Members, Employees, Contractors or Sales & Support Representatives responsible for any loss or legal liability, tangible or intangible, from any document errors or any product usage.

The product described in this document is not US ITAR controlled. Use of Alta products or documentation in violation of local usage, waste discard and export control rules, or in violation of US ITAR regulations, voids product warranty and shall not be supported. This document may be distributed to support government programs and projects. Third party person, company or consultant distribution is not allowed without Alta's written permission.

AltaCore, AltaCore-1553, AltaCore-ARINC, AltaAPI, AltaAPI-LV, AltaView, AltaRTVal, ENET-1553, ENET-A429 & ENET-1553-EBR are Trademarks of Alta Data Technologies LLC, Rio Rancho, New Mexico USA

Contact:

We welcome comments and suggestions. Please contact us at 888-429-1553 (toll free in US) or 505-994-3111 or visit our web site for support submit forms at www.altadt.com or email us at alta.info@altadt.com or alta.support@altadt.com.

Table of Contents

Table of Contents.....	iii
The AltaAPI Software Model.....	1
Layer 0 API Modules	3
Layer 1 API Module.....	4
Layer 2 API Modules	5
List of Acronyms.....	6
The Layer 0 API	8
Layer 0 API Constants	8
The Device Identifier (Device ID – or DEVID).....	9
#Define Device ID Examples for Layer 1 Programs (from ADT_L0.h):	13
Layer 0 API Files	14
Layer 0 API Type Definitions.....	14
ADT_L0_UINT32	14
ADT_L0_UINT16	14
ADT_L0_UINT8	14
Error Code Constants	15
Layer 0 API Functions.....	16
Low Level Functions	16
ADT_L0_AttachIntHandler.....	16
ADT_L0_DetachIntHandler.....	16
ADT_L0_MapMemory	17
ADT_L0_MapMemory_pcInfo	18
ADT_L0_UnmapMemory.....	18
ADT_L0_msSleep	19
ADT_L0_ReadMem16.....	19
ADT_L0_ReadMem32.....	20
ADT_L0_ReadSetupMem32	20
ADT_L0_WriteMem16.....	22
ADT_L0_WriteMem32.....	22

ADT_L0_WriteSetupMem32	23
ADT_L0_ENET_ADCP_Reset	23
ADT_L0_ENET_ADCP_GetStatistics	24
ADT_L0_ENET_ADCP_ClearStatistics.....	24
The Layer 1 API	25
Channels & Devices – Basic Definition Reviewed	26
Table Layer 1 API - 1: ARINC/A429 Device ID (DEVID) Bank Channel Assignments	27
Common Layer 1 API Functions & Discussion	28
ADT_L1.h File & 1553 or ARINC Quick Reference Guides	28
Basic Data Types.....	28
Data Structures.....	29
General Programming Flow	29
Performance Optimization	29
Initializing and Closing the API	31
Advanced Concepts on Device Initialization	31
Closing the Device – Last Step of an Application.....	32
Memory Management	32
Multi-Threaded Applications.....	33
Built-In-Test (BIT) Operation.....	34
Power-On Self Test.....	34
Periodic BIT.....	34
Initiated BIT	34
BIT Status Register.....	34
Using IRIG-B Time.....	35
IRIG Calibration.....	35
Verifying IRIG Lock.....	36
Reading the IRIG time from the Global Device.....	37
Reading the LATCHED IRIG and Internal Time from a 1553/A429 Device	37
Converting Internal Time-Stamps to IRIG Time.....	38
ENET Device and Software Overview.....	39
ENET Device Programming: ADCP	40

Alta Passive Monitor Protocol (APMP)	41
AltaAPI and ENET Programming	42
Assigning IP Addresses	42
Initialize the Device	42
Checking the ADCP Connection Status.....	42
System and Network Performance with ENET Devices.....	43
ENET Performance.....	45
ENET Application Performance Optimization	45
ENET Ethernet UDP Protocols	46
ALTA DEVICE CONTROL PROTOCOL (ADCP)	46
UDP Port Numbers	48
ADCP Packet Format.....	48
Auto 1553/ARINC Ethernet BM/RX Bridging: ALTA PASSIVE MONITOR PROTOCOL (APMP)	52
UDP Port Numbers	52
APMP Packet Format.....	53
APMP 1553 CDP Packet Format - CDP.....	55
How to Start/Control 1553 APMP	55
ARINC APMP Packet Format.....	57
How to Start/Control ARINC APMP.....	58
MIL-STD-1553 Layer 1 Programming	60
1553 (M1553) Device & Protocol Initialization & Closing	62
Advance 1553 Initialization Options – Manual Device & Protocol Setup (1553A and Other Non 1553B Variants).....	64
“RT Live” Initialization – MIL-STD-1760 Startup.....	64
Closing the Device – Last Step of an Application.....	65
Overview of MIL-STD-1553 Functions	65
The 1553 Common Data Packet (CDP).....	65
Error Injection/Detection for Data Words of BC or RT Messages	68
Setting Message (CDP) Interrupts, Triggers, etc.....	69
1553 Bus Monitor Operation	70
Figure BM-1: Basic Programming Flow	70

BM Filters.....	71
BM Buffer (CDP) Allocation	72
BM Control Functions.....	73
BM Message Read Functions.....	73
1553 Remote Terminal Operation.....	74
Figure RT-1: Basic RT Programming Flow.....	74
RT Initialization	75
Single RT and Multiple RT Configurations	76
BROADCAST Messages – Single RT and Multiple RT	77
Allocating and Managing RT/SA and Mode Code Buffers (CDPs).....	77
RT Command Legalization	78
RT Status and Last Command Words	79
RT Status Response Time.....	79
Error Injection on the RT Status Word	79
RT Control Functions	79
1553 Bus Controller (BC) Operation	80
Figure BC-1: Basic BC Programming Flow.....	80
Figure BC-2: BCCB Data Structure	83
BC Initialization.....	84
Allocating BCCBs and CDP Buffers.....	84
Reading and Writing BC Data (and reviewing message results)	85
Defining BC Messages (1553 Message Types).....	86
BC-RT	87
RT-BC	87
RT-RT.....	87
Mode Codes without Data	88
Mode Codes with Data	88
Broadcast BC-RT	89
Broadcast RT-RT	89
Broadcast Mode Codes without Data	89
Broadcast Mode Codes with Data	90
Other BC Message Types: NOPs, Delays & Branches	90

ADDRBRANCH.....	91
DELAYONLY.....	92
Note on Inter-Message Gap Time & Frame Message Scheduling	92
NOP.....	93
Starting and Stopping the BC & Synchronizing Stop	93
BC Frame Operation	93
Note on Inter-Message Gap time & Frame Message Scheduling	95
Advanced BC Frame Operation	95
Aperiodic Messages.....	97
BC Retry Operation.....	99
Error Injection on Command Words	100
1553 Playback Operation	101
Playback Relative verses Absolute (AT) Time Options	103
Playback Start/Stop Control Functions.....	104
1553 Signal Generator Operation	105
SG Start/Stop Control Functions	106
Signal Capture	106
1553 Device Interrupts.....	107
1553 Hardware Interrupts.....	107
1553 Software Polled Interrupts	107
ENET-1553 Devices and Interrupts.....	108
The Interrupt Queue.....	108
General Device Interrupt Functions	109
1553 Device Interrupt Functions	110
RT Interrupt Info Word	111
BC Interrupts	112
BCCB Complete verses BCCB CDP Interrupt.....	112
BM CDP Interrupt Info Word.....	113
ARINC 429 Device Operation.....	114
A429 Device Initialization Functions	114
A429 Channel Configuration	115

Closing the ARINC Device	116
A429 Receive (RX) Operation	117
Receive Channel Operation.....	118
Multichannel Receive Operation.....	119
Figure ARINC-RX-1: Basic Flow for Channel Level RX	120
Label LSB/MSB for ARINC-429 RXPs and TXPs.....	122
A429 Transmit (TX) Operation.....	123
Figure ARINC-TX-1: Basic TX Scheduling API Flow.....	126
Sending Aperiodic TXCB/TXP Label Lists	128
Label LSB/MSB for ARINC-429 RXPs and TXPs.....	129
A429 Playback Operation	130
Playback Relative verses Absolute (AT) Time Options	132
Playback Start/Stop Control Functions.....	133
A429 Signal Generator Operation.....	134
SG Start/Stop Control Functions	135
A429 Device Interrupt Functions.....	136
A429 Hardware Interrupts.....	136
A429 Software Polled Interrupts	136
ENET-A429 Devices and Interrupts.....	137
General A429 Interrupt Functions.....	137
Other ARINC Setup & Usage (ARINC 717)	140
Layer 1 API Files	142
Layer 1 Example Programs	142
Layer 1 API – Key Type Definitions.....	143
ADT_L1_1553_CDP	143
ADT_L1_1553_BC_CB.....	143
ADT_L1_A429_RXP	144
ADT_L1_A429_TXP	144
Label LSB/MSB for ARINC-429 RXPs and TXPs.....	144
ADT_L1_A429_TXCB.....	145
Layer 1 API Constants	146

Error Code Constants	146
Layer 1 API Functions	147
General Functions	147
ADT_L1_DevicePresent	147
ADT_L1_DevicePresent_pcilInfo	148
ADT_L1_InitDevice	149
ADT_L1_CloseDevice	150
ADT_L1_GetBoardInfo.....	151
ADT_L1_GetVersionInfo	152
ADT_L1_ProgramBoardFlash.....	153
ADT_L1_ReadDeviceMem32.....	153
ADT_L1_WriteDeviceMem32.....	154
ADT_L1_msSleep	154
ADT_L1_ENET_SetIpAddr	155
ADT_L1_ENET_GetIpAddr	156
ADT_L1_ENET_ADCP_Reset	156
ADT_L1_ENET_ADCP_GetStatistics	157
ADT_L1_ENET_ADCP_ClearStatistics.....	157
ADT_L1_Error_to_String.....	158
Board Global Functions	159
ADT_L1_Global_TimeClear.....	159
ADT_L1_Global_ConfigExtClk.....	159
ADT_L1_Global_I2C_ReadTemp.....	160
ADT_L1_Global_I2C_SetIrigDac.....	160
ADT_L1_Global_I2C_SetVVdac	161
ADT_L1_Global_CalibrateIrigDac	161
ADT_L1_Global_CalibrateIrigDacOptions.....	162
ADT_L1_Global_ReadIrigTime.....	163
Memory Management Functions	164
ADT_L1_InitMemMgmt.....	164
ADT_L1_CloseMemMgmt.....	164

ADT_L1_GetMemoryAvailable	164
ADT_L1_MemoryAlloc.....	165
ADT_L1_MemoryFree.....	166
BIT Functions.....	167
ADT_L1_BIT_MemoryTest.....	167
ADT_L1_BIT_InitiatedBIT.....	168
ADT_L1_BIT_PeriodicBIT	168
Interrupt Functions	171
ADT_L1_INT_HandlerAttach.....	171
ADT_L1_INT_HandlerDetach.....	171
1553 General Functions	173
ADT_L1_1553_InitDefault	173
ADT_L1_1553_InitDefault_ExtendedOptions	174
ADT_L1_1553_GetConfig	176
ADT_L1_1553_GetPEInfo	177
ADT_L1_1553_InitChannel	177
ADT_L1_1553_InitChannelLive.....	178
ADT_L1_1553_SetConfig	179
ADT_L1_1553_TimeClear	180
ADT_L1_1553_TimeGet.....	180
ADT_L1_1553_TimeSet	181
ADT_L1_1553_IrigLatchedTimeGet.....	182
ADT_L1_1553_PBTimeGet	183
ADT_L1_1553_PBTimeSet	183
ADT_L1_1553_UseExtClk.....	184
ADT_L1_1553_ForceTrgOut	184
ADT_L1_1553_SC_ArmTrigger	185
ADT_L1_1553_SC_ReadBuffer	185
ADT_L1_1553_CDP_Calculate_1760_Checksum	186
ADT_L1_1553_IntervalTimerGet.....	187
ADT_L1_1553_IntervalTimerSet.....	188

1553 Interrupt Functions	189
ADT_L1_1553_INT_CheckChannelIntPending.....	189
ADT_L1_1553_INT_DisableInt.....	189
ADT_L1_1553_INT_EnableInt.....	190
ADT_L1_1553_INT_GenInt.....	190
ADT_L1_1553_INT_GetIntSeqNum	190
ADT_L1_1553_INT_IQ_ReadEntry.....	190
ADT_L1_1553_INT_IQ_ReadNewEntries.....	192
ADT_L1_1553_INT_IQ_ReadRawEntry.....	193
ADT_L1_1553_INT_IQ_ReadNewRawEntries.....	194
ADT_L1_1553_INT_SetIntSeqNum.....	195
1553 Bus Monitor Functions	196
ADT_L1_1553_BM_BufferCreate	196
ADT_L1_1553_BM_BufferFree.....	196
ADT_L1_1553_BM_CDPRead	197
ADT_L1_1553_BM_CDPWrite	197
ADT_L1_1553_BM_Clear.....	198
ADT_L1_1553_BM_Config.....	198
ADT_L1_1553_BM_FilterRead	199
ADT_L1_1553_BM_FilterWrite	199
ADT_L1_1553_BM_ReadNewMsgs	200
ADT_L1_1553_BM_ReadNewMsgsDMA.....	200
ADT_L1_1553_BM_Start	201
ADT_L1_1553_BM_Stop.....	201
1553 Remote Terminal Functions.....	203
ADT_L1_1553_RT_Close	203
ADT_L1_1553_RT_Disable.....	203
ADT_L1_1553_RT_Enable	204
ADT_L1_1553_RT_GetExternalRTAddr	204
ADT_L1_1553_RT_GetLastCmd.....	205
ADT_L1_1553_RT_GetOptions.....	205

ADT_L1_1553_RT_GetRespTime.....	206
ADT_L1_1553_RT_GetSingleRTAddr.....	206
ADT_L1_1553_RT_Init.....	207
ADT_L1_1553_RT_InjStsWordError	207
ADT_L1_1553_RT_MC_CDPAllocate	208
ADT_L1_1553_RT_MC_CDPFree	208
ADT_L1_1553_RT_MC_CDPRead	208
ADT_L1_1553_RT_MC_CDPReadWords	210
ADT_L1_1553_RT_MC_CDPWrite	211
ADT_L1_1553_RT_MC_CDPWriteWords	212
ADT_L1_1553_RT_MC_LegalizationRead.....	213
ADT_L1_1553_RT_MC_LegalizationWrite.....	214
ADT_L1_1553_RT_Monitor	215
ADT_L1_1553_RT_ReadStsWordError	216
ADT_L1_1553_RT_SA_CDPAllocate.....	217
ADT_L1_1553_RT_SA_CDPFree	217
ADT_L1_1553_RT_SA_CDPRead.....	218
ADT_L1_1553_RT_SA_CDPReadWords.....	219
ADT_L1_1553_RT_SA_CDPWrite	220
ADT_L1_1553_RT_SA_CDPWriteWords.....	221
ADT_L1_1553_RT_SA_LegalizationRead	222
ADT_L1_1553_RT_SA_LegalizationWrite	223
ADT_L1_1553_RT_SetOptions.....	224
ADT_L1_1553_RT_SetRespTime	224
ADT_L1_1553_RT_SetSingleRTAddr.....	225
ADT_L1_1553_RT_Start.....	225
ADT_L1_1553_RT_StatusRead	226
ADT_L1_1553_RT_StatusWrite	226
ADT_L1_1553_RT_Stop	227
ADT_L1_1553_RT_SA_CDP_GetAddr.....	227
ADT_L1_1553_RT_MC_CDP_GetAddr.....	229
1553 Bus Controller Functions.....	231

ADT_L1_1553_BC_AperiodicIsRunning.....	231
ADT_L1_1553_BC_AperiodicSend.....	232
ADT_L1_1553_BC_CB_CDPAllocate	233
ADT_L1_1553_BC_CB_CDPFree	233
ADT_L1_1553_BC_CB_CDPRead	234
ADT_L1_1553_BC_CB_CDPReadWords.....	235
ADT_L1_1553_BC_CB_CDPWrite	236
ADT_L1_1553_BC_CB_CDPWriteWords	237
ADT_L1_1553_BC_CB_Read.....	238
ADT_L1_1553_BC_CB_ReadWords	239
ADT_L1_1553_BC_CB_Write.....	240
ADT_L1_1553_BC_CB_WriteWords	241
ADT_L1_1553_BC_SetAddressBranchValues.....	242
ADT_L1_1553_BC_Close.....	243
ADT_L1_1553_BC_GetFrameCount	244
ADT_L1_1553_BC_Init.....	244
ADT_L1_1553_BC_InjCmdWordError	245
ADT_L1_1553_BC_IsRunning	246
ADT_L1_1553_BC_ReadCmdWordError	247
ADT_L1_1553_BC_Start	248
ADT_L1_1553_BC_Stop.....	248
ADT_L1_1553_BC_CB_GetAddr	249
ADT_L1_1553_BC_CB_CDP_GetAddr.....	249
1553 Signal Generator Functions	251
ADT_L1_1553_SG_AddVectors	251
ADT_L1_1553_SG_Configure	251
ADT_L1_1553_SG_CreateSGCB	252
ADT_L1_1553_SG_Free	252
ADT_L1_1553_SG_Start	253
ADT_L1_1553_SG_Stop.....	253
ADT_L1_1553_SG_IsRunning	253
ADT_L1_1553_SG_WordToVectors.....	254

1553 Playback Functions.....	255
ADT_L1_1553_PB_Allocate	255
ADT_L1_1553_PB_CDPWrite	255
ADT_L1_1553_PB_Free	256
ADT_L1_1553_PB_GetRtResponse	256
ADT_L1_1553_PB_IsRunning	257
ADT_L1_1553_PB_SetRtResponse	257
ADT_L1_1553_PB_Start.....	257
ADT_L1_1553_PB_Stop.....	257
A429 General Functions	258
ADT_L1_A429_InitDefault.....	258
ADT_L1_A429_InitDefault_ExtendedOptions	259
ADT_L1_A429_InitDevice	261
ADT_L1_A429_GetConfig	262
ADT_L1_A429_GetPEInfo	262
ADT_L1_A429_TimeClear	263
ADT_L1_A429_UseExtClk	263
ADT_L1_A429_TimeGet	264
ADT_L1_A429_TimeSet	264
ADT_L1_A429_IrigLatchedTimeGet	265
ADT_L1_A429_PBTIMEGet	266
ADT_L1_A429_PBTIMESet.....	266
ADT_L1_A429_SC_ArmTrigger	267
ADT_L1_A429_SC_ReadBuffer	267
ADT_L1_A429_IntervalTimerGet.....	268
ADT_L1_A429_IntervalTimerSet	269
A429 Interrupt Functions	270
ADT_L1_A429_INT_CheckChannelIntPending	270
ADT_L1_A429_INT_DisableInt	270
ADT_L1_A429_INT_EnableInt	271
ADT_L1_A429_INT_GetIntSeqNum.....	271

ADT_L1_A429_INT_IQ_ReadEntry	272
ADT_L1_A429_INT_IQ_ReadNewEntries	273
ADT_L1_A429_INT_IQ_ReadRawEntry	274
ADT_L1_A429_INT_IQ_ReadNewRawEntries	275
ADT_L1_A429_INT_SetIntSeqNum	276
A429 Multichannel Receive Functions	277
ADT_L1_A429_RXMC_BufferCreate.....	277
ADT_L1_A429_RXMC_BufferFree	277
ADT_L1_A429_RXMC_ReadNewRxPs.....	278
ADT_L1_A429_RXMC_ReadNewRxPsDMA	278
ADT_L1_A429_RXMC_ReadRxP	279
ADT_L1_A429_RXMC_WriteRxP	280
A429 Receive Functions	281
ADT_L1_A429_RX_Channel_Init	281
ADT_L1_A429_RX_Channel_Close	282
ADT_L1_A429_RX_Channel_Start	282
ADT_L1_A429_RX_Channel_Stop	283
ADT_L1_A429_RX_Channel_ReadNewRxPs.....	283
ADT_L1_A429_RX_Channel_ReadNewRxPsDMA.....	284
ADT_L1_A429_RX_Channel_ReadRxP.....	285
ADT_L1_A429_RX_Channel_WriteRxP.....	285
ADT_L1_A429_RX_Channel_CVTReadRxP	286
ADT_L1_A429_RX_Channel_CVTWriteRxP	286
ADT_L1_A429_RX_Channel_SetConfig	287
ADT_L1_A429_RX_Channel_GetConfig.....	288
ADT_L1_A429_RX_Channel_SetMaskCompare	289
ADT_L1_A429_RX_Channel_GetMaskCompare.....	290
ADT_L1_A717_RX_Channel_SetConfig	291
ADT_L1_A717_RX_Channel_GetConfig.....	292
A429 Transmit Functions	293
ADT_L1_A429_TX_Channel_Init.....	293

ADT_L1_A429_TX_Channel_Close	294
ADT_L1_A429_TX_Channel_CB_TXPAllocate.....	294
ADT_L1_A429_TX_Channel_CB_TXPFree.....	295
ADT_L1_A429_TX_Channel_CB_Write.....	296
ADT_L1_A429_TX_Channel_CB_Read.....	297
ADT_L1_A429_TX_Channel_CB_TXPWrite.....	298
ADT_L1_A429_TX_Channel_CB_TXPRead.....	299
ADT_L1_A429_TX_Channel_Start	300
ADT_L1_A429_TX_Channel_Stop.....	300
ADT_L1_A429_TX_Channel_IsRunning	301
ADT_L1_A429_TX_Channel_AperiodicIsRunning.....	302
ADT_L1_A429_TX_Channel_SendLabel	303
ADT_L1_A429_TX_Channel_SendLabelBlock.....	304
ADT_L1_A429_TX_Channel_AperiodicSend.....	305
ADT_L1_A429_TX_Channel_SetConfig.....	306
ADT_L1_A429_TX_Channel_GetConfig.....	307
ADT_L1_A429_TX_ChannelGetTxpCount.....	308
ADT_L1_A429_TX_Channel_CB_GetAddr	308
ADT_L1_A429_TX_Channel_CB_TXP_GetAddr.....	309
A429 Signal Generator Functions.....	311
ADT_L1_A429_SG_AddVectors	311
ADT_L1_A429_SG_Configure	311
ADT_L1_A429_SG_CreateSGCB.....	312
ADT_L1_A429_SG_Free.....	312
ADT_L1_A429_SG_Start	313
ADT_L1_A429_SG_Stop.....	313
ADT_L1_A429_SG_IsRunning	313
ADT_L1_A429_SG_WordToVectors	314
A429 Playback Functions	315
ADT_L1_A429_TX_Channel_PB_Init	315
ADT_L1_A429_TX_Channel_PB_Close	316
ADT_L1_A429_TX_Channel_PB_CB_PXPAllocate	316

ADT_L1_A429_TX_Channel_PB_CB_PXPFree	317
ADT_L1_A429_TX_Channel_PB_CB_Write	318
ADT_L1_A429_TX_Channel_PB_CB_Read	319
ADT_L1_A429_TX_Channel_PB_CB_PXPWrite	320
ADT_L1_A429_TX_Channel_PB_CB_PXPRead	321
ADT_L1_A429_TX_Channel_PB_Start	322
ADT_L1_A429_TX_Channel_PB_Stop	322
ADT_L1_A429_PB_Start	323
ADT_L1_A429_PB_Stop	323
ADT_L1_A429_TX_Channel_PB_IsRunning	324
ADT_L1_A429_TX_Channel_PB_SetConfig	324
ADT_L1_A429_TX_Channel_PB_GetConfig	325
ADT_L1_A429_TX_Channel_PB_RXPWrite	326
The Layer 2 APIs	327
Layer 2 API for Microsoft Windows .NET 2.0	327
Appendix A – ENET-ONLY BSD API	328
Appendix B – Microsoft Windows.....	333
Windows Device Drivers	333
Windows System Requirements.....	334
Supported Alta Products	334
Supported Compilers and Development Environments.....	334
Windows Power Settings.....	334
Installation of Alta Products.....	335
Anti-Virus Note	338
Windows Layer 1 API Files	338
Layer 1 API Example Programs – **READ ME**	339
Using MSVS 2005/2008/2010/2012/2013/2015 C++ with the Layer 1 API....	339
Windows .NET Support	346
Layer 1 .NET API Example Programs – **READ ME**	346
Using MSVS 2005/2008 C# with the Layer 1 .NET 2.0 API	347
Using MSVS 2010 C# with the Layer 1 .NET 4.0 API	352

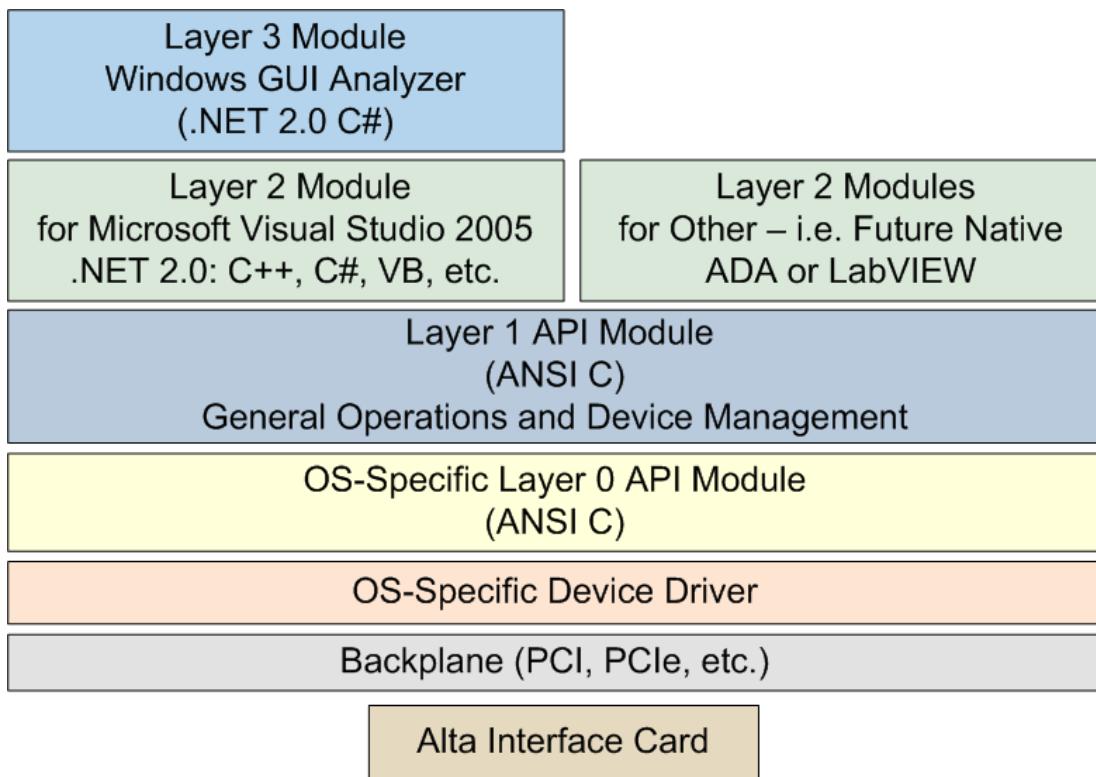
Using MSVS 2012 C# with the Layer 1 .NET 4.5 API	360
Using MSVS 2013 C# with the Layer 1 .NET 4.51 API	368
Using LabWindows/CVI™ with the Layer 1 API	375
Using LabVIEW™ and LabVIEW Real-Time™ with the Layer 1 API.....	378
Using LabVIEW™ with the Layer 1 .NET 2.0 API	378
Layer 1 .NET 2.0 API Classes and Methods	382
CLASS_1553_CDP	382
CLASS_1553_BC_CB	382
CLASS_A429_RXP	382
CLASS_A429_TXP.....	382
ADT_L1	383
Using AltaAPI with TenAsys INtime™	396
Using AltaAPI with IntervalZero RTX™	397
Installation.....	397
Configure the Alta Board as an RTX Device.....	397
Configuration for Interrupts	404
Setup an RTX Application Project in MSVS.....	404
Appendix C – Linux	412
Linux Distribution/Kernel Versions.....	412
Supported Alta Products	412
Supported C compilers.....	412
Prior to installation.....	413
Installing AltaAPI	415
Building and installing AltaDriver	415
Building and Installing the Kernel PlugIn	416
Building the AltaAPI Example Programs	417
Symbol Setup	419
Uninstalling the Kernel PlugIn and Driver	420
Appendix D – VxWorks	421
Supported VxWorks Versions.....	421
Supported Alta Products	421

Supported Development Environments	421
Hardware Installation.....	422
System Configuration	422
Software Installation	423
Testing the Installation and L0 API.....	423
Building and Testing the L1 API and Example Programs	424
Appendix E – LynxOS.....	430
Supported LynxOS Versions	430
Supported Alta Products	430
Supported C Compilers	430
Hardware Installation.....	430
Software Installation.....	430
System Configuration	431
Building and Testing the L0 and L1 API and Example Programs	432
Note on Interrupts.....	432
Appendix F – GHS INTEGRITY.....	434
Supported INTEGRITY Versions.....	434
Supported Alta Products	434
Supported Development Environments.....	434
Hardware Installation.....	434
Kernel/Driver Configuration	434
Software Installation	434
Testing the Installation and L0 API.....	435
Building and Testing the L1 API and Example Programs	435
Appendix G – Solaris	437
Supported Environments.....	437
Supported Alta Products	437
Installation on SPARC 64-bit systems.....	437
Installing the Driver on SPARC 64-bit systems	438
Installing the Kernel Plugin on SPARC 64-bit systems.....	440

Building and Testing the API and Example Programs on SPARC 64-bit Systems	440
Installation on x86 32-bit Systems.....	442
Installing the Kernel Plugin on x86 32-bit Systems.....	442
Installing the Driver on x86 32-bit Systems	443
Building and Testing the API and Example Programs on x86 32-bit Systems	444
Manual Revision Information	447

The AltaAPI Software Model

Alta uses a layered approach to structure the AltaAPI (application program interface – API), as shown below:



Each layer of the API (Layer 0, 1 and 2) is modular and can be built and tested independently. This architecture limits the impact of changes to the module where they occur and make it easier to test and troubleshoot problems.

There are many example programs provided (over 80 Layer 1 programs in the full Windows installation) and some people can jump right in to one of the example programs and cut/paste for their application. You should not have to start from scratch for almost any application – start with one of the example programs and jump-start your development. There is a README description file provided with the examples.

For Windows systems, these example programs are found at:

C:\Program Files\Alta Data Technologies\Alta
Software\ADT_L1_API\examples\M1553 Examples

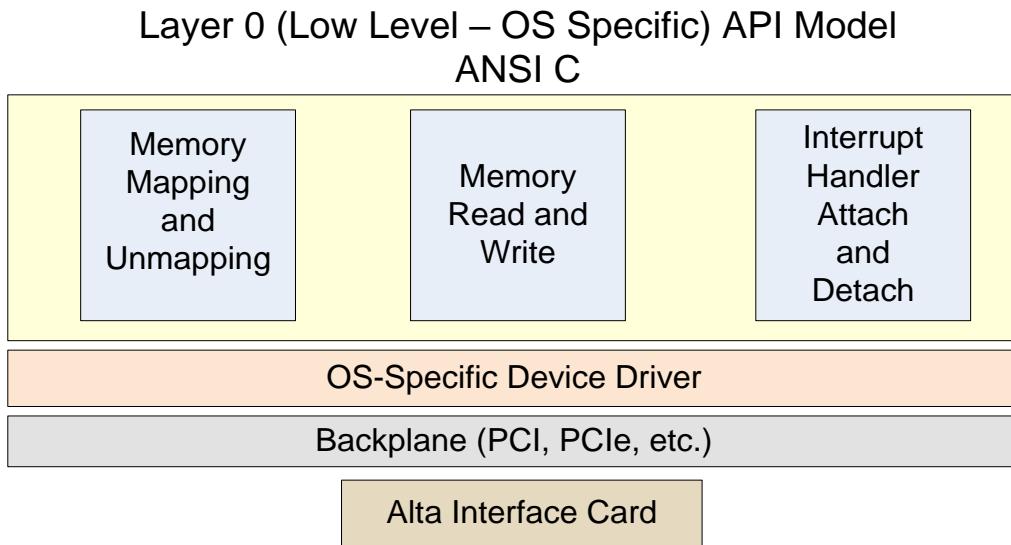
Other operating systems (OSes) have example programs in the appropriate L1 folder of their OS distribution on the CD. The example programs are generally the same regardless of the OS (Layer 1 applications are generally portable across OSes with little or no modification).

You should be able to easily browse the CD folders to find the various files and documentation. The Appendices of this manual provide installation instructions for the supported Operating Systems.

On Windows systems, the C:\Program Files\Alta Data Technologies folder is the root and it should be obvious to drill-down to the appropriate folder of interest, including Documentation and Manuals and the Various Layers of the AltaAPI per the diagram above.

Layer 0 API Modules

All operating system and platform dependencies are kept in the Layer 0 API. Therefore porting to a new OS or platform only requires changes to Layer 0. A new Layer 0 module should be created for any new OS/platform to be supported.

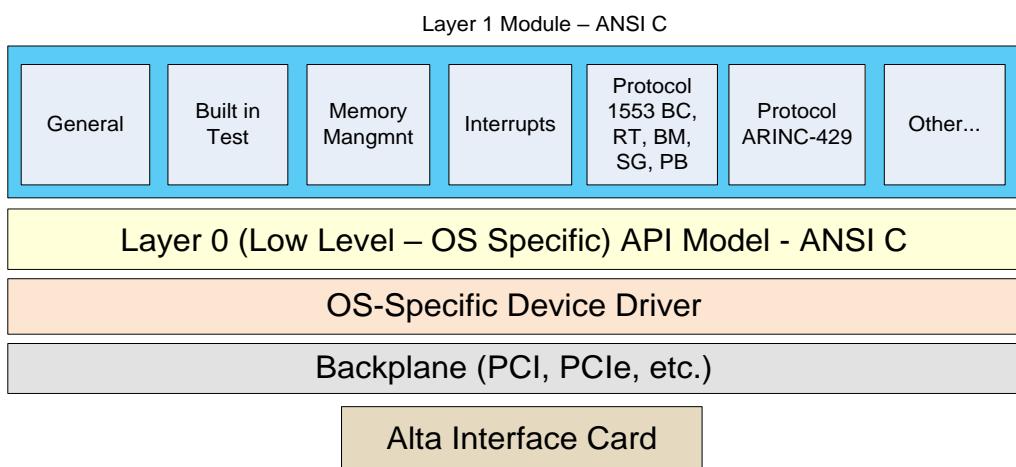


Layer 0 only contains the basic functions needed to communicate with the device driver and the Alta card. This layer is kept as simple as possible to minimize the complexity of porting to new environments.

User applications will not normally deal with the Layer 0 API directly. The user applications will typically interface with Layer 1.

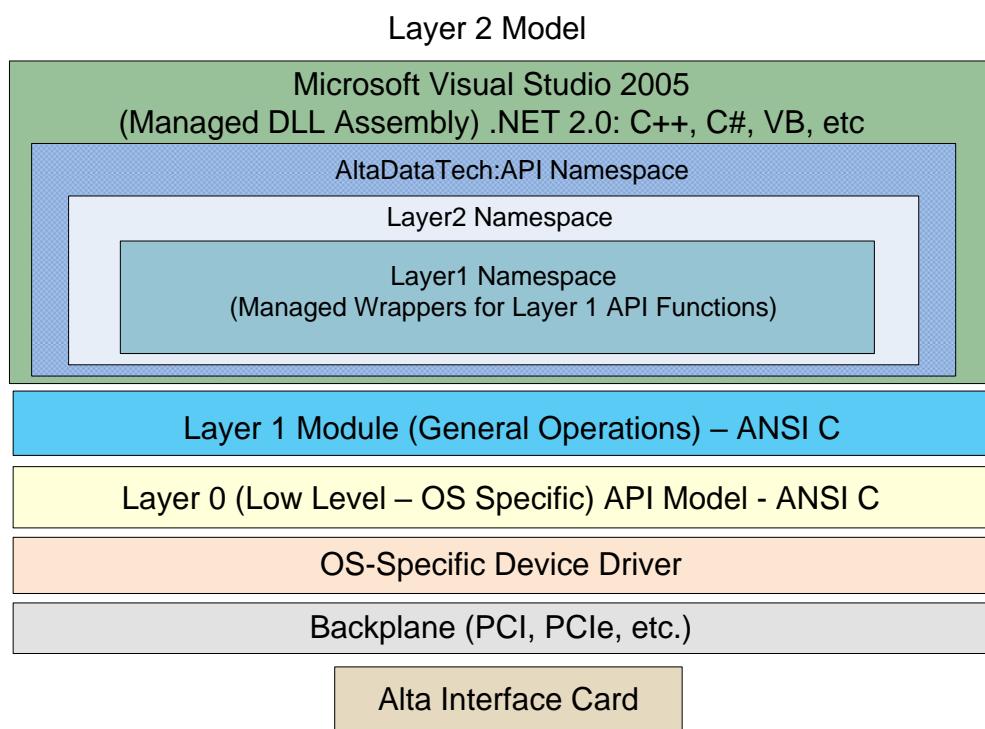
Layer 1 API Module

Layer 1 is the functional core of the API. This layer is written in ANSI C for portability to any environment and provides all functions needed to control the Alta hardware. Layer 1 uses the Layer 0 module for ALL communication with the Alta hardware. Many applications will use only Layer 1 and will not need any higher layers of the API.



Layer 2 API Modules

Layer 2 uses the Layer 1 API (which uses the Layer 0 API) to communicate with the Alta hardware. This layer uses higher-level programming languages to encapsulate the Layer 1 functions for object-oriented programming.



The ADT_L1_NET20 .NET module is a Layer 2 module that allows .NET environments like C# to use the Alta API. This is discussed in **Appendix A** in the following sections:

Using MSVS 2005/2008 C# with the Layer 1 .NET 2.0 API Layer 1 .NET 2.0 API Classes and Methods

Layer 2 modules could be added to support any object-oriented programming language, such as Java, C++, ADA, etc.

List of Acronyms

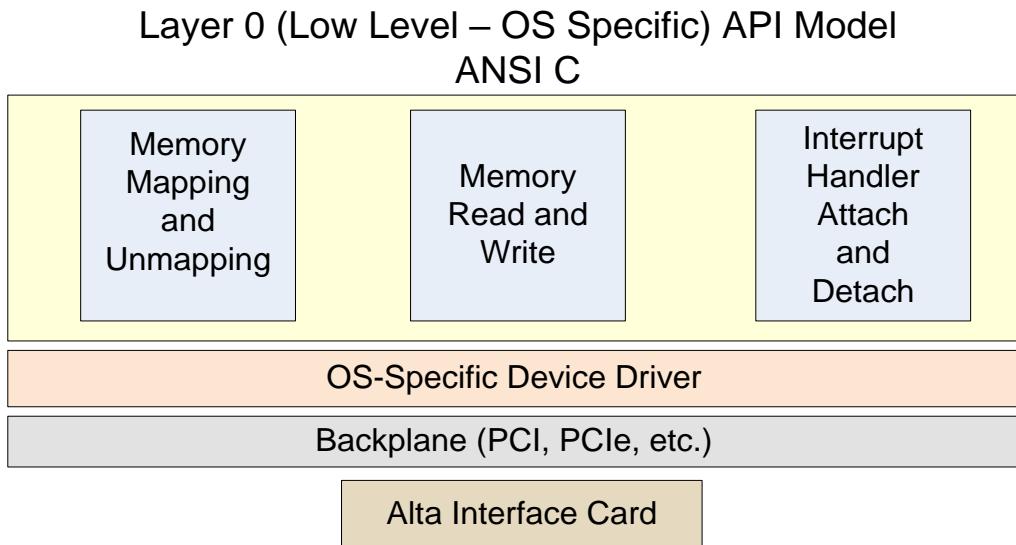
This section defines some of the acronyms that may be used in this manual.

Acronym	Meaning ("A" = ARINC; "M" = MIL-STD-1553)
ADC	Analog-Digital Converter
API	Application Programming Interface
BC	Bus Controller - M
BCCB	Bus Controller Control Block - M
BIT	Built In Test
BM	Bus Monitor
BPS	Bits Per Second
CDP	Common Data Packet - M
CSR	Control & Status Register
DAC	Digital-Analog Converter
DF	Dual-Function - M
FF	Full-Function - M
IBIT	Initiated Built In Test
IMG	Inter-Message Gap (time) - M
I2C	Inter-Integrated Circuit (serial bus)
IQ	Interrupt Queue
IQP	Interrupt Queue Packet
IRIG	Inter-Range Instrumentation Group (time code)
MC	Mode Code - M
MCRX	Multi-Channel Receive - A
OS	Operating System
PB	Playback
PBIT	Periodic Built In Test
PE	Protocol Engine
POST	Power-On Self Test
PxP	Playback Transmit Packet - A
RT	Remote Terminal - M
RX	Receive
RxP	Receive Packet - A

SA	Sub-Address - M
SC	Signal Capture
SG	Signal Generator
SGCB	Signal Generator Control Block
SW	Software
TR	Transmit/Receive - M
TX	Transmit
TXCB	Transmit Control Block - A
TxP	Transmit Packet - A
VV	Variable Voltage - M
WC	Word Count - M

The Layer 0 API

This section discusses the Layer 0 API in detail.



The Layer 0 API is the fundamental interface between the application software (and higher API layers) and the platform-specific device driver and hardware. This API layer is intended to encapsulate ALL operating-system or platform dependencies and only provides basic functions to open/map memory, close/unmap memory, read and write from/to memory, and to attach/detach interrupt handlers.

ALL higher software layers must go through the Layer 0 API functions to access the hardware.

Separate Layer 0 modules are defined for different operating-systems/platforms. For example, one Layer 0 module is used for Windows platforms and a completely different Layer 0 module is used for Linux platforms. All platforms use the same Layer 1 module.

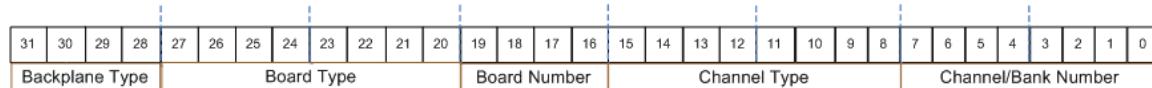
Layer 0 API Constants

The top-level header file (ADT_L0.h) defines the constants used in this API layer. These constants are used by higher API layers to interface with Layer 0.

The Device Identifier (Device ID – or DEVID)

The Layer 0 API (and higher layers of the API) uses a 32-bit unsigned integer value as the Device Identifier (or Device ID - DEVID). It is very important to understand what the Device ID is and what information it provides to the API.

First of all, we shall define a “device” as a functional hardware module. For 1553, this will be a CHANNEL (including both Bus A and Bus B and all associated functionality – BM, RT, BC, etc.). For other protocols, the definition of a “device” may vary based on what makes sense for that protocol. For example, an ARINC-429 “device” will consist of a set of transmit and receive channels (“Bank”).



As shown above, the Device ID consists of five fields – Backplane Type, Board Type, Board Number, Channel Type, and Channel Number. The following shows examples of their ADT_L0.h definitions. More definitions can be added from time to time so review the ADT_L0.h file if there are questions.

The **Backplane Type** tells us the type of interface to the board. Currently defined Backplane types are 0x0 for SIMULATED (where the API simulates the board with an internal array representing board memory), 0x1 for PCI and PCIe, and 0x2 for Ethernet.

```
/* Device ID Constants - Backplane Type (4 bits) - (DEVID & 0xF0000000) */
#define ADT_DEVID_BACKPLANETYPE_SIMULATED      0x00000000
#define ADT_DEVID_BACKPLANETYPE_PCI             0x10000000
#define ADT_DEVID_BACKPLANETYPE_ENET            0x20000000
```

WARNING – The simulated devices (SIM1553 and SIMA429) are only intended for use with the AltaView software. The simulation is very limited and is performed in AltaView. Do NOT use simulated devices in your applications.

The **Board Type** tells us what product the board is. The Test1553 and TestA429 board types are reserved for internal Alta use only.

```
/* Device ID Constants - Board Type (8 bits) - (DEVID & 0xFF000000) */
#define ADT_DEVID_BOARDDTYPE_SIM1553           0x00000000
#define ADT_DEVID_BOARDDTYPE_TEST1553          0x00100000
#define ADT_DEVID_BOARDDTYPE_PMC1553           0x00200000
#define ADT_DEVID_BOARDDTYPE_PCI104P1553       0x00300000
#define ADT_DEVID_BOARDDTYPE_PCI1553           0x00400000
#define ADT_DEVID_BOARDDTYPE_PCCD1553          0x00500000
#define ADT_DEVID_BOARDDTYPE_PCI104E1553       0x00600000
#define ADT_DEVID_BOARDDTYPE_XMC1553           0x00700000
```

```

#define ADT_DEVID_BOARDTYPE_ECD54_1553          0x00800000
#define ADT_DEVID_BOARDTYPE_PCIE4L1553           0x00900000
#define ADT_DEVID_BOARDTYPE_PCIE1L1553           0x00A00000
#define ADT_DEVID_BOARDTYPE_MPCIE1553            0x00B00000
#define ADT_DEVID_BOARDTYPE_XMCMW                0x00C00000

#define ADT_DEVID_BOARDTYPE_SIMA429              0x01000000
#define ADT_DEVID_BOARDTYPE_TESTA429              0x01100000
#define ADT_DEVID_BOARDTYPE_PMCA429              0x01200000
#define ADT_DEVID_BOARDTYPE_PC104PA429            0x01300000
#define ADT_DEVID_BOARDTYPE_PCIA429              0x01400000
#define ADT_DEVID_BOARDTYPE_PCCDA429             0x01500000
#define ADT_DEVID_BOARDTYPE_PCI104EA429           0x01600000
#define ADT_DEVID_BOARDTYPE_XMCA429              0x01700000
#define ADT_DEVID_BOARDTYPE_ECD54_A429            0x01800000
#define ADT_DEVID_BOARDTYPE_PCIE4LA429             0x01900000
#define ADT_DEVID_BOARDTYPE_PCIE1LA429             0x01A00000
#define ADT_DEVID_BOARDTYPE_MPCIEA429             0x01B00000
#define ADT_DEVID_BOARDTYPE_PC104PA429LTV         0x01C00000
#define ADT_DEVID_BOARDTYPE_PMC429HD              0x01D00000

#define ADT_DEVID_BOARDTYPE_PMCMA4                0x02200000
#define ADT_DEVID_BOARDTYPE_PC104PMA4              0x02300000
#define ADT_DEVID_BOARDTYPE_PC104EMA4              0x02600000
#define ADT_DEVID_BOARDTYPE_XMCM4                 0x02700000
#define ADT_DEVID_BOARDTYPE_TBOLTMA4              0x02800000

#define ADT_DEVID_BOARDTYPE_ENET1553              0x03100000
#define ADT_DEVID_BOARDTYPE_PMCE1553              0x03200000
#define ADT_DEVID_BOARDTYPE_ENETA429              0x03300000
#define ADT_DEVID_BOARDTYPE_ENET1A1553             0x03400000
#define ADT_DEVID_BOARDTYPE_ENET485               0x03500000
#define ADT_DEVID_BOARDTYPE_ENET1553EBR            ADT_DEVID_BOARDTYPE_ENET485
#define ADT_DEVID_BOARDTYPE_ENET2_1553             0x03600000
#define ADT_DEVID_BOARDTYPE_ENET_MA4               0x03700000
#define ADT_DEVID_BOARDTYPE_ENETX_MA4              0x03800000

#define ADT_DEVID_BOARDTYPE_PMCWMUX              0x04200000

```

We have defined a set of “product type” constants that combine backplane and board type.

```

/* Device ID Constants - Product (backplane | board type)
 * For defining DEVID in user applications.
 */
#define ADT_PRODUCT_SIM1553          ADT_DEVID_BACKPLANETYPE_SIMULATED |
ADT_DEVID_BOARDTYPE_SIM1553       ADT_DEVID_BACKPLANETYPE_PCI |
#define ADT_PRODUCT_TEST1553          ADT_DEVID_BACKPLANETYPE_PCI |
ADT_DEVID_BOARDTYPE_TEST1553      ADT_DEVID_BACKPLANETYPE_PCI |
#define ADT_PRODUCT_PMC1553           ADT_DEVID_BACKPLANETYPE_PCI |
ADT_DEVID_BOARDTYPE_PMC1553       ADT_DEVID_BACKPLANETYPE_PCI |
#define ADT_PRODUCT_PC104P1553        ADT_DEVID_BACKPLANETYPE_PCI |
ADT_DEVID_BOARDTYPE_PC104P1553    ADT_DEVID_BACKPLANETYPE_PCI |
#define ADT_PRODUCT_PCI1553           ADT_DEVID_BACKPLANETYPE_PCI |
ADT_DEVID_BOARDTYPE_PCI1553       ADT_DEVID_BACKPLANETYPE_PCI |
#define ADT_PRODUCT_PCCD1553          ADT_DEVID_BACKPLANETYPE_PCI |
ADT_DEVID_BOARDTYPE_PCCD1553      ADT_DEVID_BACKPLANETYPE_PCI |
#define ADT_PRODUCT_PCI104E1553        ADT_DEVID_BACKPLANETYPE_PCI |
ADT_DEVID_BOARDTYPE_PCI104E1553   ADT_DEVID_BACKPLANETYPE_PCI |
#define ADT_PRODUCT_XMC1553           ADT_DEVID_BACKPLANETYPE_PCI |
ADT_DEVID_BOARDTYPE_XMC1553       ADT_DEVID_BACKPLANETYPE_PCI |
#define ADT_PRODUCT_ECD54_1553         ADT_DEVID_BACKPLANETYPE_PCI |
ADT_DEVID_BOARDTYPE_ECD54_1553    ADT_DEVID_BACKPLANETYPE_PCI |
#define ADT_PRODUCT_PCIE4L1553         ADT_DEVID_BACKPLANETYPE_PCI |
ADT_DEVID_BOARDTYPE_PCIE4L1553    ADT_DEVID_BACKPLANETYPE_PCI |
#define ADT_PRODUCT_PCIE1L1553         ADT_DEVID_BACKPLANETYPE_PCI |
ADT_DEVID_BOARDTYPE_PCIE1L1553    ADT_DEVID_BACKPLANETYPE_PCI |
#define ADT_PRODUCT_MPCIE1553          ADT_DEVID_BACKPLANETYPE_PCI |
ADT_DEVID_BOARDTYPE_MPCIE1553     ADT_DEVID_BACKPLANETYPE_PCI |

```

```

#define ADT_PRODUCT_XMCMW
ADT_DEVID_BOARDTYPE_XMCMW

#define ADT_PRODUCT_SIMA429
ADT_DEVID_BOARDTYPE_SIMA429
#define ADT_PRODUCT_TESTA429
ADT_DEVID_BOARDTYPE_TESTA429
#define ADT_PRODUCT_PMCA429
ADT_DEVID_BOARDTYPE_PMCA429
#define ADT_PRODUCT_PC104PA429
ADT_DEVID_BOARDTYPE_PC104PA429
#define ADT_PRODUCT_PCIA429
ADT_DEVID_BOARDTYPE_PCIA429
#define ADT_PRODUCT_PCCDA429
ADT_DEVID_BOARDTYPE_PCCDA429
#define ADT_PRODUCT_PCI104EA429
ADT_DEVID_BOARDTYPE_PCI104EA429
#define ADT_PRODUCT_XMCA429
ADT_DEVID_BOARDTYPE_XMCA429
#define ADT_PRODUCT_ECD54_A429
ADT_DEVID_BOARDTYPE_ECD54_A429
#define ADT_PRODUCT_PCIE4LA429
ADT_DEVID_BOARDTYPE_PCIE4LA429
#define ADT_PRODUCT_PCIE1LA429
ADT_DEVID_BOARDTYPE_PCIE1LA429
#define ADT_PRODUCT_MPCIEA429
ADT_DEVID_BOARDTYPE_MPCIEA429
#define ADT_PRODUCT_PC104PA429LTV
ADT_DEVID_BOARDTYPE_PC104PA429LTV
#define ADT_PRODUCT_PMCA429HD
ADT_DEVID_BOARDTYPE_PMCA429HD

#define ADT_PRODUCT_PMCMA4
ADT_DEVID_BOARDTYPE_PMCMA4
#define ADT_PRODUCT_PC104PMA4
ADT_DEVID_BOARDTYPE_PC104PMA4
#define ADT_PRODUCT_PC104EMA4
ADT_DEVID_BOARDTYPE_PC104EMA4
#define ADT_PRODUCT_XMCMA4
ADT_DEVID_BOARDTYPE_XMCMA4
#define ADT_PRODUCT_TBOLTM4A4
ADT_DEVID_BOARDTYPE_TBOLTM4A4

#define ADT_PRODUCT_ENET1553
ADT_DEVID_BOARDTYPE_ENET1553
#define ADT_PRODUCT_PMCE1553
ADT_DEVID_BOARDTYPE_PMCE1553
#define ADT_PRODUCT_ENETA429
ADT_DEVID_BOARDTYPE_ENETA429
#define ADT_PRODUCT_ENET1A1553
ADT_DEVID_BOARDTYPE_ENET1A1553
#define ADT_PRODUCT_ENET485
ADT_DEVID_BOARDTYPE_ENET485
#define ADT_PRODUCT_ENET1553EBR
#define ADT_PRODUCT_ENET2_1553
ADT_DEVID_BOARDTYPE_ENET2_1553
#define ADT_PRODUCT_ENET_MA4
ADT_DEVID_BOARDTYPE_ENET_MA4
#define ADT_PRODUCT_ENETX_MA4
ADT_DEVID_BOARDTYPE_ENETX_MA4

#define ADT_PRODUCT_PMCWMUX
ADT_DEVID_BOARDTYPE_PMCWMUX

ADT_DEVID_BACKPLANETYPE_PCI |
ADT_DEVID_BACKPLANETYPE_SIMULATED |
ADT_DEVID_BACKPLANETYPE_PCI |
ADT_DEVID_BACKPLANETYPE_ENET |
ADT_DEVID_BACKPLANETYPE_PCI |

```

The Board Number is just that – the number of the specific board in the system of that backplane and product type. The first board is board number 0, the second board is board number 1, and so on.

```
/* Device ID Constants - Board Number (4 bits) - (DEVID & 0x000F0000) */
```

```

#define ADT_DEVID_BOARDNUM_01      0x00000000
#define ADT_DEVID_BOARDNUM_02      0x00010000
#define ADT_DEVID_BOARDNUM_03      0x00020000
#define ADT_DEVID_BOARDNUM_04      0x00030000
#define ADT_DEVID_BOARDNUM_05      0x00040000
#define ADT_DEVID_BOARDNUM_06      0x00050000
#define ADT_DEVID_BOARDNUM_07      0x00060000
#define ADT_DEVID_BOARDNUM_08      0x00070000
#define ADT_DEVID_BOARDNUM_09      0x00080000
#define ADT_DEVID_BOARDNUM_10      0x00090000
#define ADT_DEVID_BOARDNUM_11      0x000A0000
#define ADT_DEVID_BOARDNUM_12      0x000B0000
#define ADT_DEVID_BOARDNUM_13      0x000C0000
#define ADT_DEVID_BOARDNUM_14      0x000D0000
#define ADT_DEVID_BOARDNUM_15      0x000E0000
#define ADT_DEVID_BOARDNUM_16      0x000F0000

```

The **Channel Type** tells us what the channel is – this could be a 1553 channel, global registers, etc. The channel types currently defined are 0x01 for the board-level global registers, 0x10 for 1553 channels, 0x20 for A429 devices, and 0x30 for WMUX devices.

```

/* Device ID Constants - Channel Type (8 bits) - (DEVID & 0x0000FF00) */
#define ADT_DEVID_CHANNELTYPE_GLOBALS      0x00000100
#define ADT_DEVID_CHANNELTYPE_1553         0x00001000
#define ADT_DEVID_CHANNELTYPE_A429         0x00002000
#define ADT_DEVID_CHANNELTYPE_WMUX        0x00003000

```

The **Channel/Bank Number** is an index representing multiple devices on the same board. For example, a 1553 board can have multiple dual redundant channels and each 1553 channel is considered a “device” by the API.

For ARINC a “Bank” of channels is considered a “device” - a Bank usually consists of 8, 14 or 16 ARINC channels. There are typically one or two banks per card and the value for Bank 1 would be zero and Bank 2 would be one.

The use of the Channel/Bank Number field may vary for different protocols, or may always be zero for product types that have only one “device” per board. For remainder of this document, the reference to “Channel” may also refer to a “Bank” for ARINC devices.

```

/* Device ID Constants - Channel Number (8 bits) - (DEVID & 0x000000FF) */
#define ADT_DEVID_CHANNELNUM_01          0x00000000
#define ADT_DEVID_CHANNELNUM_02          0x00000001
#define ADT_DEVID_CHANNELNUM_03          0x00000002
#define ADT_DEVID_CHANNELNUM_04          0x00000003
#define ADT_DEVID_CHANNELNUM_05          0x00000004
#define ADT_DEVID_CHANNELNUM_06          0x00000005
#define ADT_DEVID_CHANNELNUM_07          0x00000006
#define ADT_DEVID_CHANNELNUM_08          0x00000007
#define ADT_DEVID_CHANNELNUM_09          0x00000008
#define ADT_DEVID_CHANNELNUM_10          0x00000009
#define ADT_DEVID_CHANNELNUM_11          0x0000000A
#define ADT_DEVID_CHANNELNUM_12          0x0000000B
#define ADT_DEVID_CHANNELNUM_13          0x0000000C
#define ADT_DEVID_CHANNELNUM_14          0x0000000D
#define ADT_DEVID_CHANNELNUM_15          0x0000000E
#define ADT_DEVID_CHANNELNUM_16          0x0000000F

/* Device ID Constants - ARINC Bank Number (8 bits) - (DEVID & 0x000000FF) */
#define ADT_DEVID_BANK_01              0x00000000

```

```

#define ADT_DEVID_BANK_02          0x00000001
#define ADT_DEVID_BANK_03          0x00000002
#define ADT_DEVID_BANK_04          0x00000003

```

Raw Hex Device ID Examples:

A Device ID of 0x10201000 indicates Channel 1 of Board 1, PMC-1553
A Device ID of 0x10201001 indicates Channel 2 of Board 1, PMC-1553
A Device ID of 0x10211000 indicates Channel 1 of Board 2, PMC-1553
A Device ID of 0x10211001 indicates Channel 2 of Board 2, PMC-1553
A Device ID of 0x10200200 indicates Bank 1 of Board 1, PMC-A429
A Device ID of 0x10200201 indicates Bank 2 of Board 1, PMC-A429
A Device ID of 0x10210200 indicates Bank 1 of Board 2, PMC-A429
A Device ID of 0x10210201 indicates Bank 2 of Board 2, PMC-A429

#Define Device ID Examples for Layer 1 Programs (from ADT_L0.h):

```

// #define examples for other cards & channels - change for your card!!
// PCI-1553 Channel 1
#define DEVID (ADT_PRODUCT_PCI1553 | ADT_DEVID_BOARDNUM_01 | ADT_DEVID_CHANNELTYPE_1553 |ADT_DEVID_CHANNELNUM_01)

//PMC-1553 Channel 2
#define DEVID (ADT_PRODUCT_PMC1553 | ADT_DEVID_BOARDNUM_01 | ADT_DEVID_CHANNELTYPE_1553 |ADT_DEVID_CHANNELNUM_02)

//PCCD-1553 Channel 2
#define DEVID (ADT_PRODUCT_PCCD1553 | ADT_DEVID_BOARDNUM_01 | ADT_DEVID_CHANNELTYPE_1553 |ADT_DEVID_CHANNELNUM_02)

//PMCMA4 (1553 and ARINC card) 1553 Channel 1
#define DEVID (ADT_PRODUCT_PMCMA4 | ADT_DEVID_BOARDNUM_01 | ADT_DEVID_CHANNELTYPE_1553 |ADT_DEVID_CHANNELNUM_01)

//PMCMA4 (1553 and ARINC card) ARINC Bank 1
#define DEVID (ADT_PRODUCT_PMCMA4 | ADT_DEVID_BOARDNUM_01 | ADT_DEVID_CHANNELTYPE_A429 |ADT_DEVID_BANK_01)

//PCI-A429 Bank 1
#define DEVID (ADT_PRODUCT_PCIA429 | ADT_DEVID_BOARDNUM_01 | ADT_DEVID_CHANNELTYPE_A429 |ADT_DEVID_BANK_01)

//PCCD-A429 Bank 1
#define DEVID (ADT_PRODUCT_PCCD429 | ADT_DEVID_BOARDNUM_01 | ADT_DEVID_CHANNELTYPE_A429 |ADT_DEVID_BANK_01)

```

Layer 0 API Files

The Layer 0 API consists of a top-level header file (ADT_L0.h) and at least one C file that implements the Layer 0 functions. Additional files may be added for internal functions. The operating-system specific appendices discuss the specific files included with each Layer 0 module.

Layer 0 API Type Definitions

The top-level header file (ADT_L0.h) defines a few basic types that are used by higher API layers. These are provided in type definitions in Layer 0 so that they can be defined in one place appropriately for the target operating system and platform.

ADT_L0_UINT32

This is the **32-bit unsigned integer** type. This is the primary base type used throughout the higher layers of the API.

ADT_L0_UINT16

This is the **16-bit unsigned integer** type.

ADT_L0_UINT8

This is the **8-bit unsigned integer** type.

Error Code Constants

The Layer 0 API reserves the error code range 0 to 999. These constants define integer (32-bit) values that will be returned by the API functions to indicate either SUCCESS or an error code indicating why the function failed.

```
/* Layer 0 Error Codes (1 to 999) */
#define ADT_SUCCESS 0      /*!< \brief Function call completed without error. */
#define ADT_FAILURE 1      /*!< \brief Function call completed with error. */
#define ADT_ERR_MEM_MAP_SIZE 2  /*!< \brief Invalid memory map size. */
#define ADT_ERR_NO_DEVICE 3  /*!< \brief Device not found */
#define ADT_ERR_CANT_OPEN_DEV 4  /*!< \brief Can't open device */
#define ADT_ERR_DEV_NOT_INITED 5  /*!< \brief Device not initialized */
#define ADT_ERR_DEV_ALREADY_OPEN 6  /*!< \brief Device already open */
#define ADT_ERR_UNSUPPORTED_BACKPLANE 7  /*!< \brief Unsupported backplane in DevID */
#define ADT_ERR_UNSUPPORTED_BOARDTYPE 8  /*!< \brief Unsupported board type in DevID */
#define ADT_ERR_UNSUPPORTED_CHANNELTYPE 9  /*!< \brief Unsupported channel type in DevID */
#define ADT_ERR_CANT_OPEN_DRIVER 10  /*!< \brief Can't open driver */
#define ADT_ERR_CANT_SET_DRV_OPTIONS 11  /*!< \brief Can't set driver options */
#define ADT_ERR_CANT_GET_DEV_INFO 12  /*!< \brief Can't get device info */
#define ADT_ERR_INVALID_BOARD_NUM 13  /*!< \brief Invalid board number */
#define ADT_ERR_INVALID_CHANNEL_NUM 14  /*!< \brief Invalid channel number */
#define ADT_ERR_DRIVER_READ_FAIL 15  /*!< \brief Driver read memory failure */
#define ADT_ERR_DRIVER_WRITE_FAIL 16  /*!< \brief Driver write memory failure */
#define ADT_ERR_DEVICE_CLOSE_FAIL 17  /*!< \brief Device close failure */
#define ADT_ERR_DRIVER_CLOSE_FAIL 18  /*!< \brief Driver close failure */
#define ADT_ERR_KP_OPEN_FAIL 19  /*!< \brief Kernel Plug-In Open failure */
#define ADT_ERR_ENET_NO_PORT_AVAILABLE 100  /*!< \brief No UDP port available */
#define ADT_ERR_ENET_READ_FAIL 101  /*!< \brief ENET Read failure */
#define ADT_ERR_ENET_WRITE_FAIL 102  /*!< \brief ENET Write failure */
#define ADT_ERR_ENET_NOTRUNNING 103  /*!< \brief ENET Sockets not running */
#define ADT_ERR_ENET_INVALID_SIZE 104  /*!< \brief ENET Invalid payload size */
#define ADT_ERR_ENET_SENDFAIL 105  /*!< \brief ENET Send failure */
#define ADT_ERR_ENET_SELECTFAIL 106  /*!< \brief ENET Select failure */
#define ADT_ERR_ENET_SELECTTIMEOUT 107  /*!< \brief ENET Select timeout */
#define ADT_ERR_ENET_BADSEQNUM 108  /*!< \brief ENET Bad sequence number */
#define ADT_ERR_ENET_SRVSTSFAIL 109  /*!< \brief ENET Server Status Code indicate FAILURE */
#define ADT_ERR_ENET_BADPRODUCTID 110  /*!< \brief ENET Bad Product ID */
#define ADT_ERR_CACHEDDMAFAIL 111  /*!< \brief DMA Cache Malloc Fail */
```

NOTE: `ADT_ERR_NO_DEVICE` is the most common Layer 0 error returned. This usually means the wrong DEVID has been defined (or not changed from a Layer 1 example program – you must change the DEVID parameter to match your card type).

An error of `ADT_ERR_CANT_OPEN_DEV` would indicate a driver problem. And an error of `ADT_ERR_DEV_ALREADY_OPEN` probably indicates the device was not properly closed (probably an application crash or improper shutdown).

The function `ADT_L1_Error_to_String` can be used to convert an error/status code to a string.

Layer 0 API Functions

Each of the Layer 0 API functions is described below.

Low Level Functions

ADT_L0_AttachIntHandler

```
ADT_L0_UINT32 ADT_L0_AttachIntHandler (ADT_L0_UINT32 devID,  
                                      ADT_L0_UINT32 chanRegOffset,  
                                      void * pUserISR,  
                                      void * pUserData)
```

This function attaches an interrupt handler function. If a pointer to user context data is provided in the pUserData parameter, this will be passed back to the user interrupt handler function when the interrupt occurs.

NOTE: AltaAPI versions prior to v2.2.1.0 did not include the pUserData parameter. This needs to be added when porting code from older API versions.

Parameters:

devID - 32-bit Device Identifier.
chanRegOffset is the byte offset to the PE registers for the channel.
pUserISR is a pointer to the function to attach as an ISR.
pUserData is a pointer to user context data – set to NULL if not used.

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_UNSUPPORTED_BACKPLANE - Unsupported backplane
ADT_FAILURE - Completed with error

ADT_L0_DetachIntHandler

```
ADT_L0_UINT32 ADT_L0_DetachIntHandler (ADT_L0_UINT32 devID)
```

This function detaches an interrupt handler function.

Parameters:

devID - 32-bit Device Identifier.

Returns:

ADT_SUCCESS - Completed without error
ADT_FAILURE - Completed with error

ADT_L0_MapMemory

```
ADT_L0_UINT32 ADT_L0_MapMemory (ADT_L0_UINT32 devID,  
                                ADT_L0_UINT32 startupOptions,  
                                ADT_L0_UINT32 clientIpAddress,  
                                ADT_L0_UINT32 serverIpAddress);
```

This function maps the device memory and stores the memory pointer internally for use with the read/write functions.

Parameters:

devID - 32-bit Device Identifier.

startupOptions are user selected startup options defined as follows:

```
#define ADT_L1_API_DEVICEINIT_NOKP 0x00000004
```

Note: the “ADT_L1_API_DEVICEINIT_NOKP” startup option only applies to platforms that use the Jungo WinDriver software for the device driver (Windows, Linux, Solaris). If this option is selected then hardware interrupts cannot be used, because the kernel plug-in is required for hardware interrupts. This option is used for Alta internal testing and rare cases where the kernel plug-in cannot be installed.

clientIpAddress is the 32-bit IP address of the client computer where the AltaAPI program is running. THIS IS ONLY USED FOR ENET DEVICES. For all other devices this parameter is ignored and can be set to zero.

serverIpAddress is the 32-bit IP address of the server ENET device. THIS IS ONLY USED FOR ENET DEVICES. For all other devices this parameter is ignored and can be set to zero.

Returns:

ADT_SUCCESS - Completed without error

ADT_FAILURE - Completed with error

ADT_L0_MapMemory_pcilInfo

```
ADT_L0_UINT32 ADT_L0_MapMemory_pcilInfo (ADT_L0_UINT32 devID,
                                         ADT_L0_UINT32 startupOptions,
                                         ADT_L0_UINT32 *pciBus,
                                         ADT_L0_UINT32 *pciDevice,
                                         ADT_L0_UINT32 *pciFunc);
```

This function maps the device memory and stores the memory pointer internally for use with the read/write functions. Also returns PCI info – Bus/Device/Function.

Parameters:

devID - 32-bit Device Identifier.

startupOptions are user selected startup options defined as follows:

```
#define ADT_L1_API_DEVICEINIT_NOKP 0x00000004
```

Note: the “ADT_L1_API_DEVICEINIT_NOKP” startup option only applies to platforms that use the Jungo WinDriver software for the device driver (Windows, Linux, Solaris). If this option is selected then hardware interrupts cannot be used, because the kernel plug-in is required for hardware interrupts. This option is used for Alta internal testing and rare cases where the kernel plug-in cannot be installed.

pciBus is the pointer to store the PCI Bus number.

pciDevice is the pointer to store the PCI Device number.

pciFunc is the pointer to store the PCI Function number.

Returns:

ADT_SUCCESS - Completed without error

ADT_FAILURE - Completed with error

ADT_L0_UnmapMemory

```
ADT_L0_UINT32 ADT_L0_UnmapMemory (ADT_L0_UINT32 devID)
```

This function un-maps and releases a previously mapped block of memory.

Parameters:

devID - 32-bit Device Identifier.

Returns:

ADT_SUCCESS - Completed without error

ADT_FAILURE - Completed with error

ADT_L0_msSleep

void ADT_L0_msSleep (ADT_L0_UINT32 *msDelay*)

This function waits for the requested number of milliseconds.

Parameters:

msDelay is the number of milliseconds to wait.

ADT_L0_ReadMem16

ADT_L0_UINT32 ADT_L0_ReadMem16 (ADT_L0_UINT32 *devID*,
ADT_L0_UINT32 *offset*,
ADT_L0_UINT16 * *data*,
ADT_L0_UINT32 *count*)

This function reads the requested number of 16-bit words from memory.

This function is only used when programming the PE/firmware to an Alta device.

Parameters:

devID - 32-bit Device Identifier.
offset - BYTE offset to the first word to read
data - pointer to store the words read
count - number of 16-bit words to read

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_DEV_NOT_INITED - Device not initialized
ADT_ERR_UNSUPPORTED_BACKPLANE - Unsupported backplane
ADT_FAILURE - Completed with error

ADT_L0_ReadMem32

```
ADT_L0_UINT32 ADT_L0_ReadMem32 (ADT_L0_UINT32 devID,  
                                ADT_L0_UINT32 offset,  
                                ADT_L0_UINT32 * data,  
                                ADT_L0_UINT32 count)
```

This function reads the requested number of 32-bit words from memory.

Parameters:

devID - 32-bit Device Identifier.
offset - BYTE offset to the first word to read
data - pointer to store the words read
count - number of 32-bit words to read

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_DEV_NOT_INITED - Device not initialized
ADT_ERR_UNSUPPORTED_BACKPLANE - Unsupported backplane
ADT_FAILURE - Completed with error

ADT_L0_ReadMem32DMA

```
ADT_L0_UINT32 ADT_L0_ReadMem32DMA (ADT_L0_UINT32 devID,  
                                    ADT_L0_UINT32 offset,  
                                    ADT_L0_UINT32 * data,  
                                    ADT_L0_UINT32 count)
```

This function uses DMA to read the requested number of 32-bit words from memory.
DMA read is limited to 400 bytes (100 words).

Parameters:

devID - 32-bit Device Identifier.
offset - BYTE offset to the first word to read
data - pointer to store the words read
count - number of 32-bit words to read

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_DEV_NOT_INITED - Device not initialized
ADT_ERR_UNSUPPORTED_BACKPLANE - Unsupported backplane
ADT_ERR_UNSUPPORTED_BOARDTYPE - Board does not support DMA
ADT_FAILURE - Completed with error

ADT_L0_ReadSetupMem32

```
ADT_L0_UINT32 ADT_L0_ReadSetupMem32 (ADT_L0_UINT32 devID,  
                                      ADT_L0_UINT32 offset,
```

```
ADT_L0_UINT32 * data,  
ADT_L0_UINT32 count)
```

This function reads the requested number of 32-bit words from board setup memory.
This function is only used when programming the PE/firmware to an Alta device.

Parameters:

devID - 32-bit Device Identifier.
offset - BYTE offset to the first word to read
data - pointer to store the words read
count - number of 32-bit words to read

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_UNSUPPORTED_BACKPLANE - Unsupported backplane
ADT_FAILURE - Completed with error

ADT_L0_WriteMem16

```
ADT_L0_UINT32 ADT_L0_WriteMem16 (ADT_L0_UINT32 devID,  
                                ADT_L0_UINT32 offset,  
                                ADT_L0_UINT16 * data,  
                                ADT_L0_UINT32 count)
```

This function writes the requested number of 16-bit words to memory.

This function is only used when programming the PE/firmware to an Alta device.

Parameters:

devID - 32-bit Device Identifier.
offset - BYTE offset to the first word to write
data - pointer to the words to write
count - number of 16-bit words to write

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_DEV_NOT_INITED - Device not initialized
ADT_ERR_UNSUPPORTED_BACKPLANE - Unsupported backplane
ADT_FAILURE - Completed with error

ADT_L0_WriteMem32

```
ADT_L0_UINT32 ADT_L0_WriteMem32 (ADT_L0_UINT32 devID,  
                                ADT_L0_UINT32 offset,  
                                ADT_L0_UINT32 * data,  
                                ADT_L0_UINT32 count)
```

This function writes the requested number of 32-bit words to memory.

Parameters:

devID - 32-bit Device Identifier.
offset - BYTE offset to the first word to write
data - pointer to the words to write
count - number of 32-bit words to write

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_DEV_NOT_INITED - Device not initialized
ADT_ERR_UNSUPPORTED_BACKPLANE - Unsupported backplane
ADT_FAILURE - Completed with error

ADT_L0_WriteSetupMem32

```
ADT_L0_UINT32 ADT_L0_WriteMem32 (ADT_L0_UINT32 devID,  
                                ADT_L0_UINT32 offset,  
                                ADT_L0_UINT32 * data,  
                                ADT_L0_UINT32 count)
```

This function writes the requested number of 32-bit words to board setup memory.
This function is only used when programming the PE/firmware to an Alta device.

Parameters:

devID - 32-bit Device Identifier.
offset - BYTE offset to the first word to write
data - pointer to the words to write
count - number of 32-bit words to write

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_UNSUPPORTED_BACKPLANE - Unsupported backplane
ADT_FAILURE - Completed with error

ADT_L0_ENET_ADCP_Reset

```
ADT_L0_UINT32 ADT_L0_ENET_ADCP_Reset (ADT_L0_UINT32 devID)
```

This function is only used with ENET devices and transmits an ADCP RESET command to the device. This resets the entire ENET device and must use the device ID for the Global device.

Parameters:

devID - 32-bit Device Identifier.

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_UNSUPPORTED_BACKPLANE - Unsupported backplane, must be ENET device
ADT_ERR_UNSUPPORTED_CHANNEL - Unsupported channel type, must be Global device
ADT_FAILURE - Completed with error

ADT_L0_ENET_ADCP_GetStatistics

```
ADT_L0_UINT32 ADT_L0_ENET_ADCP_GetStatistics (ADT_L0_UINT32 devID,
                                              ADT_L0_UINT32 *pPortNum,
                                              ADT_L0_UINT32 *pTransactions,
                                              ADT_L0_UINT32 *pRetries,
                                              ADT_L0_UINT32 *pFailures)
```

This function is only used with ENET devices and gets statistical information on the ADCP communications with the device. This returns the transaction count, retry count, and failure count. A transaction is a command packet and response packet pair (normally a memory read or write operation). A retry is anything that causes a retry (bad sequence number in response packet, bad status value in response packet, or timeout waiting for response packet). A failure is any transaction where all retries were attempted but did not succeed.

Parameters:

- devID* - 32-bit Device Identifier.
- pPortNum* - pointer to store the UDP port number for the device.
- pTransactions* - pointer to store the number of transactions.
- pRetries* - pointer to store the number of retries.
- pFailures* - pointer to store the number of failures.

Returns:

- ADT_SUCCESS** - Completed without error
- ADT_ERR_UNSUPPORTED_BACKPLANE** - Unsupported backplane, must be ENET device
- ADT_ERR_ENET_NOTRUNNING** – ENET services not running, device not initialized

ADT_L0_ENET_ADCP_ClearStatistics

```
ADT_L0_UINT32 ADT_L0_ENET_ADCP_ClearStatistics
                           (ADT_L0_UINT32 devID)
```

This function is only used with ENET devices and clears the statistical information on the ADCP communications with the device. This clears the transaction count, error count, and failure count.

Parameters:

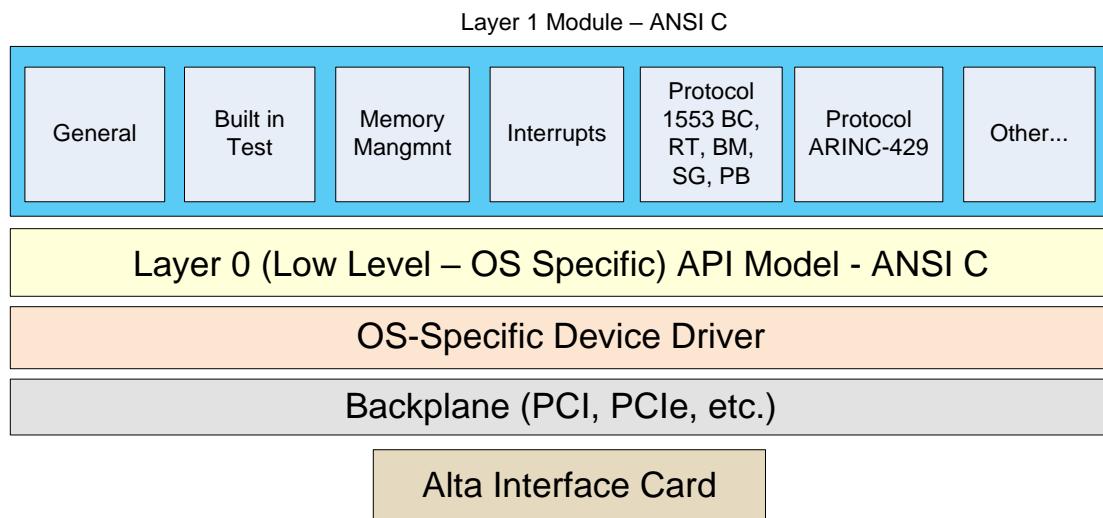
- devID* - 32-bit Device Identifier.

Returns:

- ADT_SUCCESS** - Completed without error
- ADT_ERR_UNSUPPORTED_BACKPLANE** - Unsupported backplane, must be ENET device
- ADT_ERR_ENET_NOTRUNNING** – ENET services not running, device not initialized

The Layer 1 API

This section discusses the Layer 1 API in detail.



The Layer 1 API is a general ANSI “C” API that provides all the major functions needed to work with Alta products. This API layer is portable to any platform/operating-system/environment that supports ANSI “C” programming. This layer contains NO platform-specific code. Only one Layer 1 (L1) API module is needed for ALL operating-systems and platforms.

Portability and Reliability

Many applications will use only the Layer 1 API to control the Alta boards and will not use any higher API layers. Windows user generally link in the provided DLL/Lib files and Linux and Unix based operating systems will build the desired source files into their application and link in the provided shared object file for the layer 0 API. The Appendices at the end of this manual provide details on installation and setup for the supported operating systems. Once the Alta software is installed and the appropriate development project is setup, then the user will be using the same Layer 1 functions regardless of the OS – this makes for a highly portable application.

Another key point of the Layer 1 API is to abstract and manage the low level driver and device-level hardware control registers. Most competitors’ APIs require the application to manage low-level registers and raw memory pointers, which is not reliable and not portable. Alta’s Layer 1 API will abstract and manage the registers and pointers and provide the user’s application access via

simple index numbers for data buffers and message structures. This can greatly simplify application program and makes the application more reliable and portable.

Channels & Devices – Basic Definition Reviewed

The AltaAPI and card level Protocol Engines (PEs) are unique in the industry where each 1553 channel or **bank** of ARINC channels is a unique logical **device** (in the API referenced as a “DEVID”). Separate logical devices allow multi-application support, usually one application thread per device. This is unique and powerful as most other products on the market do not support multiple applications natively in their design and force the user’s application to manage different applications. Alta devices even have their device reset so that respective application control does not affect another device’s application thread, even on the same card. In most cases, each device application can even have its own interrupt handler.

Each 1553 channel is a separate logical device.

For ARINC/A429 products, channels are grouped into logical devices called a **Bank**. Each channel within a bank has independent controls. An ARINC/A429 device **Bank** has a maximum of 16 channels and only PCI, PMC, XMC, PCI Express type cards (with larger board area) have more than 16 channels, and thus, two logical device banks. This bank number is fundamental in controlling the desired channel of the card.

The following table shows ARINC/A429 product channel assignments for each bank. Your hardware manual will provide pin-outs for the channels. Remember, many of the first channels of each bank are shared RX and TX functions, which can be very handy for self-monitoring of TX applications (the pin-outs in the respective hardware manual mark these channels).

ARINC/A429 Product	Bank1 (Channels 1-16 max)	Bank2 (Channels 17-30 max)
PCI-A429	Channels 1-16	Channels 17-30
PCIE4L-A429	Channels 1-16	Channels 17-30
PCIE1L-A429	Channels 1-8	NA
PCCD-A429	Channels 1-8	NA
ECD54-A429	Channels 1-8	NA
PMC-A429 (including cPCI and PCI Carriers)	Channels 1-16	Channels 17-30
PMC-MA4 (including cPCI and PCI Carriers)	Channels 1-8	NA
XMC-A429	Channels 1-16	Channels 17-30
XMC-MA4	Channels 1-8	NA
PC104P-A429	Channels 1-16	Channels 17-24
MPCIE-A429	Channels 1-6	NA
ENET-A429	Channels 1-8	NA
ENET-A429P	Channels 1-8	NA

Table Layer 1 API - 1: ARINC/A429 Device ID (DEVID) Bank Channel Assignments

All Layer 1 (L1) functions start with a Device ID parameter – this parameter is usually referenced as the “DEVID” parameter in the documents and sample programs (the DEVID value is derived from ADT_L0.h defines as described in the previous section).

Each of these devices can be controlled separately, even if the devices are on the same board. For example, you can run one of the API example programs on one 1553 channel device and run the AltaView bus analyzer software on another 1553 channel device on the same board and use AltaView to verify that your API program is performing correctly.

Common Layer 1 API Functions & Discussion

The following sections will detail how to use the Layer 1 API and will explain the architecture and operation of the API. Prior to starting the discussion, we should review a few Layer 1 basics that apply to any Alta device (1553 or ARINC – A429).

ADT_L1.h File & 1553 or ARINC Quick Reference Guides

The ADT_L1.h file is a key reference file for API programming. Data structures, bit definitions, L1 error codes (status/error code range 1000-1999) and prototypes of function calls are defined in this file. You will want to have this file handy to reference key API definitions.

For most applications, you can match up the ADT_L1.h file with the 1553 or ARINC Quick Reference Guide found in the Documents/Manuals folder of the CD (or in the “Alta Data Technologies\Alta Documentation\Manuals” path of your MS Windows installation). The Quick Reference Guide and the ADT_L1.h file combine to provide a great reference for detailed programming of data structures and bit control/status options.

If you need detailed reference to 1553 or ARINC Protocol Engine (PE) data structures and PE rules, then reference the appropriate AltaCore manual (1553 or ARINC).

Basic Data Types

The fundamental data type used by the API is an unsigned 32-bit word. Because different operating-systems or platforms may have different definitions of data types, we use the Layer 0 API module (which contains all OS/platform specifics) to define the unsigned 32-bit data type appropriately for the OS/platform. The Layer 0 API module defines (in the file ADT_L0.h) the following type for an unsigned 32-bit word:

ADT_L0_UINT32

This is the building block for all higher-level data structures used in the API. You will see that most API function parameters are of this type or are data structures containing fields of this type. There are few references to ADT_L0_UINT16/UINT8, but the ADT_L0_UNIT32 data type is by far the most referenced.

Data Structures

There are several data structures defined in the ADT_L1.h file that are used in the API. These structures will provide control, status and data values that you set or read (from card reads). It is a good idea to zero/clear, memset(), these structures to zero prior to reference.

General Programming Flow

All applications will follow a simple execution flow as follows:

1. Start the API: Connect the Device to the Driver and Initialize API and Device Memory – Single Function call for most applications.
2. Initialize the Desired Protocol Engine (PE) Operation – Call an Init Function for 1553 or ARINC Operations
 - a. Allocate Buffers or Set Defaults Values for the Operation. (You might repeat this Step 2 for Multi Operation Applications).
3. Start the Operation
4. **Read and Write Data for the Operation** – This step is usually repeated at some application control frequency through system (or application timer events) or through interrupt events. This step is the main part of the application (the whole purpose of the Alta network interface card is to read and write data from the 1553 or ARINC networks).
5. Stop the Operation
6. Close (and de-allocate) Operation Memory
7. Close the API

Most of the steps above are each performed in a single function call. There are several example programs provided and all of them follow the general flow above. Most applications behave in the manner above: Initialize the connection, allocate buffers, read/write data, free the buffers, and close the connection. The following figures and manual sections will detail each of these steps specific for your card's 1553 or ARINC operation.

Performance Optimization

Applications that need to operate at very fast update rates, may need to optimize their application to use direct data structure access (use of on-board memory addresses instead of abstract data structure references) in order to bypass many of the pointer lookups in the AltaAPI L1 calls. MOST applications DO NOT need this step. Direct access control requires the developer to be familiar with the low level definitions of AltaCore structures and some of the inner workings of the AltaAPI source. Alta has example programs on how to perform this for most

applications. Please contact Alta and we'll be glad to work with you for the best approach to building your application.

For information on ENET device-specific performance considerations, refer to section 'System and Network Performance with ENET Devices'.

Initializing and Closing the API

The first and last step of any program will be the start (Initialize) and close the device with the API. Initialization connects the device to the device driver and initializes memory for the API and device. The Close function will do the opposite: Close will disconnect the device from the driver and destroy or de-allocate memory associated with the device and 1553 or ARINC operation for the respective device.

Most applications will choose one the following initialization functions:

- **ADT_L1_1553/A429_InitDefault()**
- **ADT_L1_1553/A429_InitDefault_ExtendedOptions()**

There are many different 1553 and ARINC variants used on older or custom networks – but 95% of applications want to use standard protocol setup.

These “InitDefault” functions combine low level L1 setup (memory mapping) and device/protocol steps for standard 1553 and ARINC 429 protocols to make a simple, one function call method for Initializing a 1553 or ARINC device.

Most applications do not need to use/read the following paragraph, but please review the “Closing the Device” paragraph on the next page.

Advanced Concepts on Device Initialization

General functions to open and close the API are provided in the file

ADT_L1_General.c. The InitDefault functions above call this base function:

ADT_L1_InitDevice(DEVID, options);

This function maps memory to the selected device, initializes the API memory management structures, and has options to perform a memory test, perform hard PE resets and disabling of Windows/Solaris/Linux Jungo Kernel Plug-In for the Jungo driver (ignore this for most applications). If any of these steps fail then the ADT_L1_InitDevice function will return an appropriate error code.

Note that only ONE application can control a given device/channel. If a second application tries to initialize a device that has already been initialized the initialization function will return the ADT_ERR_DEVICEINUSE error code. The device in use flag will only be cleared when the device is closed (with the ADT_L1_CloseDevice function described below).

WARNING: If an application exits without closing the device then the device will be unavailable because it will still be marked as “in use”. The “options” parameter of the ADT_L1_InitDevice function can be set to a non-zero value to override the “in use” error.

Closing the Device – Last Step of an Application

At the end of an application, or when the application no longer needs to use the Alta device it can close the API as follows:

ADT_L1_CloseDevice(DEVID);

This function frees resources, closes memory management, un-maps memory, and detaches from the device. No API calls should be made for the device after the ADT_L1_CloseDevice call has been made. The device must be initialized again before use.

Memory Management

The API provides memory management functions (in the file ADT_L1_MemMgmt.c) to allocate and free blocks of memory on the Alta board. These functions are normally only used internally by the API to manage board memory for the data structures created, maintained, and destroyed by other API functions. User applications will not normally use any of the memory management functions directly.

Many of the API functions come in pairs – one function creates/allocates data structures, control blocks, buffers, etc. and a corresponding function closes/frees the data structures. This allows the user application to dynamically create and destroy data structures as needed without re-initializing the device in order to re-use the same board memory.

The ADT_L1_GetMemoryAvailable function is provided to allow the user to see how much memory is available on the device:

```
status = ADT_L1_GetMemoryAvailable(DEVID, &memAvailable);
if (status == ADT_SUCCESS)
    printf("%d bytes of memory available.\n", memAvailable);
```

This function returns the number of bytes available. This value will be a multiple of 4 because memory is allocated in 32-bit words.

WARNING:

The functions that allocate and free board memory are not thread-safe. Multi-threaded applications should use a single thread for a single device (1553 channel or ARINC 429 device/bank). See the section on multi-threaded applications below for more information.

Multi-Threaded Applications

There are parts of the Alta API that are not thread-safe. This section discusses what can and cannot be done with the Alta API in multi-threaded applications.

The Alta API memory management module uses an internal global array of “device memory management” structure pointers, each of which provides a linked-list of structures that track the free memory on a given device (1553 channel or A429 bank of channels). This global array is the main concern for multi-threaded applications using the Alta API.

Because each device ID has its own entry in the memory manager array, it is safe to have different threads for different devices on a given Alta board. Likewise, it is safe to have separate threads for each Alta board in the system (because each board is a collection of devices).

It MAY NOT be safe to use multiple threads on a single device. For example, a 1553 application might want to use a single 1553 channel device with separate threads for BC, RT, and BM operation. If these threads are simultaneously allocating or freeing board memory for BC messages, RT buffers, etc. then it is possible to have conflicts in the memory management module of the Alta API.

That being said, it is possible to use multiple threads on a single device with careful application design. If the application firsts performs all allocation of data structures in the main thread, then starts threads that only read or write packet data without allocating or freeing memory on the board, then this can work.

Alta does not guarantee thread-safety with the Alta API. Multi-threaded applications should be restricted to ONE THREAD for ONE DEVICE (remember for ARINC devices are a bank of RX/TX channels).

Built-In-Test (BIT) Operation

The API provides BIT functions in the file ADT_L1_BIT.c. BIT functions are used at the logical device level (1553 channel device or A429 bank device). If you are using multiple 1553 channels or A429 banks you will run BIT separately for each logical device (as specified by the Device Identifier).

Built-In-Tests are classified as “Power-On Self Test” (POST) which is performed on initialization, “Periodic BIT” (PBIT) which is run periodically during normal operation of the device, and “Initiated BIT” (IBIT) which is performed only when initiated by a user command.

Power-On Self Test

The device will perform internal tests on power-up and the results will be stored in the BIT Status Register. The POST results can be checked by using the ADT_L1_BIT_PeriodicBIT function to read the BIT Status Register. When the board is initialized, the API maps memory and performs an additional memory test using the ADT_L1_BIT_MemoryTest function.

Periodic BIT

Periodic BIT is safe test that is run by the FPGA during normal operation to verify the basic health of the device. The ADT_L1_BIT_PeriodicBIT function is used to check the status of PBIT.

Initiated BIT

Initiated BIT is safe test that may be run at any time during normal operation to verify the basic health of the device. It runs the same tests that Periodic BIT runs, but may be initiated at any time using the ADT_L1_BIT_InitiatedBIT function.

If additional testing is desired, POST may be run after power up by resetting the device via bit 31 of the Root PE Control Word. The ADT_L1_BIT_MemoryTest function may also be called as part of a user IBIT process to test all or part of the device’s memory if desired. Both of these methods include destructive memory tests that will overwrite data structures in board memory and will require the device to be re-initialized to work properly after they are run.

BIT Status Register

The BIT Status Register should be zero if all tests pass. A non-zero value indicates a failure. Please refer to the appropriate **AltaCore Users Manual** (1553 or A429) for details on the BIT Status Register.

Using IRIG-B Time

The Alta boards are capable of decoding IRIG-B time-code signals.

Alta products use **IRIG Standard 200-04** and support these IRIG time codes:

- **IRIG-B002** (Format B, DC Level Shift, no carrier, BCD Time Of Year)
- **IRIG-B006** (Format B, DC Level Shift, no carrier, BCD Time Of Year, BCD Year)
- **IRIG-B122** (Format B, AM, 1KHz carrier, BCD Time Of Year)
- **IRIG-B126** (Format B, AM, 1KHz carrier, BCD Time Of Year, BCD Year)

The Alta IRIG decoder does not use the coded expressions for **Control Flags (CF)** or **Straight Binary Seconds (SBS)**.

WARNING: The older **IRIG Standard 200-98** did not include **BCD Year** and used the bits in P5 through P8 for **Control Flags (CF)**. The newer **IRIG Standard 200-04** reserves P5 through P6 for **BCD Year** and restricts Control Flags to P6 through P8. Non-zero Contol Flag bits in P5 through P6 will be seen as BCD Year by the Alta IRIG decoder.

More information on IRIG time-code formats can be found here:

<http://www.irigb.com/pdf/wp-irig-200-04.pdf>

This section explains how IRIG-B time can be correlated to time-stamps from the internal 50MHz clock on Alta boards. The example program **ADT_L1_1553_ex_bm2_IRIGsync.c** demonstrates the API usage discussed here.

You must first connect the IRIG-B signal to the board. Refer to the Hardware Manual for your specific Alta board type for information on connector pin-outs.

IRIG Calibration

Once the signal is connected to the board, the first step is to calibrate the IRIG decoder on the Alta board to your IRIG-B signal. This is done at the **Global device** level – this calibration applies for all 1553 channels and/or A429 banks on the board.

```
status = ADT_L1_Global_CalibrateIrigDac(DEVID_GLOB);
```

This process will take several seconds to complete. If the return status is API_SUCCESS then the calibration was successful and the board should now be locked to the IRIG signal.

Verifying IRIG Lock

You can check the IRIG lock status by reading the Global CSR for the board. Bit 4 will be set if you have a good lock to the IRIG signal.

```
status = ADT_L1_ReadDeviceMem32(DEVID_GLOB,  
                                ADT_L1_GLOBAL_CSR, &globalCSR, 1);  
  
if (globalCSR & ADT_L1_GLOBAL_CSR_IRIG_LOCK)  
    printf("IRIG signal is locked.\n");  
else  
    printf("IRIG signal is NOT locked.\n");
```

Reading the IRIG time from the Global Device

You can read the IRIG time for the last one-second sync from the Global device for the board.

```
status = ADT_L1_Global_ReadIrigTime(DEVID_GLOB,  
    &timeHigh, &timeLow);
```

The “timeHigh” and “timeLow” words will contain the IRIG time in BCD format as shown below.

IRIG Time High – 0x000000C8																																
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
Reserved												BCD Years												BCD Days								
Reserved												BCD Hours												BCD Minutes								BCD Seconds
Reserved												BCD Hours												BCD Minutes								BCD Seconds

IRIG Time Low – 0x000000CC

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
Reserved												BCD Hours												BCD Minutes								BCD Seconds
Reserved												BCD Hours												BCD Minutes								BCD Seconds

NOTE:

IRIG Standard 200-98 – The IRIG-B time code does not include years.

IRIG Standard 200-04 – The IRIG-B time code can include years.

NOTE:

The IRIG time registers on the board will contain the time for the PREVIOUS one-second sync, so if you read the raw value from the board the time will be one second behind. The API corrects for this so the ADT_L1_Global_ReadIrigTime function returns the correct time.

Reading the LATCHED IRIG and Internal Time from a 1553/A429 Device

The firmware on the Alta board will latch the IRIG time and the corresponding internal 50MHz clock time on every one-second IRIG sync (every one-second boundary). **Therefore, the latched time is updated once per second.** These latched values can be used to convert message time-stamps from internal time to IRIG time.

This is done at the 1553 channel or A429 bank level (rather than the Global device) because each channel/bank device has its own internal clock registers that may have different values for each channel/bank on the board. The following function can be used for 1553 channel devices.

```
status = ADT_L1_1553_IrigLatchedTimeGet(DEVID,  
    &irigTimeHigh, &irigTimeLow,
```

```
&intTimeHigh, &intTimeLow,  
&deltaTimeHigh, &deltaTimeLow);
```

This function returns the latched IRIG time (in BCD format), the latched internal 50MHz time (20ns LSB), and the delta time – this is the difference between the latched IRIG time (converted from BCD to 64-bit binary with 20ns LSB) and the latched internal time.

NOTE: The IRIG time registers on the board will contain the time for the PREVIOUS one-second sync, so if you read the raw value from the board the time will be one second behind. The API corrects for this so the ADT_L1_1553_IrigLatchedTimeGet function returns the correct time.

Converting Internal Time-Stamp to IRIG Time

The delta time can be used to convert time-stamps from the internal 50MHz clock to IRIG time. For example, let's say we want to read 1553 CDP message buffers from the 1553 Bus Monitor and convert the time-stamps on these messages from internal time to IRIG time.

```
/* Convert the delta time to a 64-bit value */  
timeDelta = ((unsigned long long int) deltaTimeHigh << 32) |  
            (unsigned long long int) deltaTimeLow;  
  
/* Get the internal timestamp from the CDP as 64-bit value */  
timeCDP = ((unsigned long long int) bmMessages[i].TimeHigh << 32) |  
           (unsigned long long int) bmMessages[i].TimeLow;  
  
/* Convert to the equivalent IRIG time (64-bit math) */  
timelrig = timeCDP + timeDelta;  
  
/* Write the equivalent IRIG time back to the CDP */  
bmMessages[i].TimeHigh = (ADT_L0_UINT32) (timelrig >> 32);  
bmMessages[i].TimeLow =  
           (ADT_L0_UINT32) (timelrig & 0x00000000FFFFFFFF);
```

Now the CDP time-stamp contains the 64-bit binary IRIG time for the message. If the CDPs are saved to a file, then the file can be opened with the AltaView bus analyzer software (using the 1553 BM File Viewer) and the messages will be displayed with IRIG time-stamps.

ENET Device and Software Overview

Alta's ENET (ENET) device is the industry's first and only real-time Ethernet to 1553/ARINC devices. ENET is a small, rugged Ethernet appliance that provides unprecedented real-time access to setup, transmission and monitor/receive buffers for 1553 and ARINC applications. **This discussion also applies to PMCE and other Alta products that are Ethernet based – they will be referred to as ENET going forward.**

The ENET device essentially works the same as any of Alta's cards. Alta has replaced the normal PCI/PCI Express backplane interface with a real-time UDP server. For most applications, you can run the same AltaAPI code on an ENET device as you can any of our cards. Note that Ethernet does not provide a hardware interrupt like PCI/PCI Express does, therefore ENET applications using interrupts must use software polled interrupts.

All API accesses/functions are distilled down to a couple low-level ADT_L0 read/write memory accesses. A typical single memory transfer for a PCI card will take ~two usec and this same transfer an Ethernet packet with your computer's Internet Protocol (IP) operating system stack can take 50-200+ uSec. For hard real-time applications/simulations, Ethernet cannot replace PCI/PCI Express cards that are installed in the backplane of a computer. Your computer's operating system (OS) Internet Protocol (IP) processing stack will cause much more delay in memory accesses than a single PCI/PCI Express memory access (~2 uSec) across a computer's backplane.

ENET is not a significant source of Ethernet transfer delay. This is what is unique about ENET and all other products on the market. ENET has a real-time UDP server to accept memory read/write accesses in less than 10 uSec. Your computer's OS IP stack is the source of any delays in device memory accesses. This Ethernet access delay is called "Path Delay". Each computer is different and Path Delay results can vary greatly: Contact Alta if you want to review if ENET would be practical for your application configuration.

For most applications, ENET provides unparalleled flexibility in configuring a system for 1553 and ARINC connectivity. For example, unlike PCI or USB devices, most OSes already support Ethernet socket programming, so ENET can drop in almost any computer system.

ENET Device Programming: ADCP

ENET products are controlled through an Alta UDP protocol called Alta Device Control Protocol (**ADCP**) that provides device memory reads/writes in essentially the same manner as PCI or PCIe backplane actions (but over Ethernet instead of a computer backplane). UDP is inherently a “connectionless” IP protocol, so Alta developed a small, fast UDP handshake protocol that ensures packet delivery to the ENET device through a series of simple retries and failure indications (and is much faster than TCP).

The Layer 0 and Layer 1 API manages the ADCP UDP handshake just like PCI/PCIe transfers to make the Ethernet interface transparent to the customer’s application verses a normal card PCI/PCIe transaction. You can even mix and match card devices and ENET devices and the AltaAPI code will automatically determine if a PCI access is needed or a UDP ADCP is performed (all based on the Device ID, “DEVID,” parameter).

Alta’s example programs show how to write your application so it can run the application with PCI/PCI Express cards or with ENET devices without even changing the executable (by adding a couple lines of code)! This provides excellent portability and flexibility.

The API provides some additional functions for managing ENET devices. The example program ***ADT_L1_1553_ex_bcbm1_Timing_Test.c*** demonstrates the usage of some of these functions. The API functions specific to ENET devices are:

- ADT_L1_ENET_SetIpAddr
- ADT_L1_ENET_GetIpAddr
- ADT_L1_ENET_ADCP_GetStatistics
- ADT_L1_ENET_ADCP_ClearStatistics
- ADT_L1_ENET_ADCP_Reset

WARNING – The ENET devices and API are NOT designed to recover from a total loss of connection. All bets are off if you disconnect and reconnect the Ethernet cable while your application is running – DO NOT DO THIS. If you do, your application may lock up completely and you will have to exit the application, reset the system, and restart your application.

Alta Passive Monitor Protocol (APMP)

In addition to ADCP device control mode, ENET devices can be optionally configured to simultaneously broadcast Ethernet packets containing 1553 or ARINC receive data. This uses the Alta Passive Monitor Protocol (**APMP**) and details are provided at the end of this section. Essentially, you can setup a unique UDP packet on the ENET device to auto transmit 1553/ARINC packets as they arrive so that any computer on the LAN can monitor or record data. There are also several example programs you can cut/paste to implement a simple UDP APMP monitor. This requires VERY little code and could be done in almost any language or computer system.

AltaAPI and ENET Programming

There is only one additional AltaAPI call required to make an application run with a PCI/PCI Express card or with ENET.

Assigning IP Addresses

You must give the API the Server (ENET device) and Client (controlling computer) IP Addresses BEFORE calling the normal initialization functions. This is done with the ADT_L1_ENET_SetIpAddr function. **This must be executed before initializing the device.**

```
status = ADT_L1_ENET_SetIpAddr(DEVID, serverIpAddr, clientIpAddr);
```

The IP Address parameters are 32-bit values. An IP Address of 192.168.0.5 would be represented as 0xC0A80005 (192 = 0xC0, 168 = 0xA8, 0 = 0x00, 5 = 0x05).

The Alta example programs show how to check the DEVID parameter so that you only need to make this function call if an ENET device is defined in the DEVID parameter (a simple “IF” statement check will allow your application to run with ENET or any other Alta device). Also, the example programs also show a simple Macro to convert individual IP octets to the single ADT_L0_UINT32 for serverIpAddr/clientIpAddr parameters.

Initialize the Device

After the IP Addresses have been set, the ENET device is initialized the same way any other Alta device is initialized. Individual reads/writes on Ethernet are slower than on PCI, so performing a full memory test on every initialization can cause a delay. The code below demonstrates how to initialize the device without running a full memory test on the device.

```
printf("Initializing . . . ");
status = ADT_L1_1553_InitDefault_ExtendedOptions(DEVID, 10,
                                                ADT_L1_API_DEVICEINIT_NOMEMTEST);
```

Checking the ADCP Connection Status

The Alta API provides a function to check the status of the ADCP communications with the ENET device. The function ADT_L1_ENET_ADCP_GetStatistics returns the number of ADCP transactions, retries, and failures. A “transaction” is a read or write operation consisting of a command packet and a response packet. If there is a problem with the transaction (error or timeout waiting for response), the transaction is retried. If the transaction is not successful after three attempts then this is a failure.

```

/* For ENET devices - Get and display the ENET ADCP statistics */
if ((DEVID & 0xF0000000) == ADT_DEVID_BACKPLANETYPE_ENET) {
    printf("\n*****\n");
    printf("\nGetting ENET ADCP Statistics . . . ");
    status = ADT_L1_ENET_ADCP_GetStatistics(DEVID, &portnum,
                                            &transactions, &retries, &failures);
    if (status == ADT_SUCCESS) {
        printf("Success.\n");
        printf("UDP Port %d: %d trans, %d retries, %d failures\n",
               portnum, transactions, retries, failures);
    }
    else printf("FAILED! Error = %d\n", status);
    printf("\n*****\n");
}

```

If you have no retries or failures then you have a perfect connection to the ENET device. It is normal to see some retries. If you see some retries but no failures this is acceptable. If you see failures then you have a problem with your connection to the ENET device – this can be due to heavy loading of the network, using a slow network connection, or other problems on the network. The next section discusses system and network performance issues.

The function ADT_L1_ENET_ADCP_ClearStatistics can be used to reset the counts of ADCP transactions, retries, and failures to zero.

System and Network Performance with ENET Devices

Network conditions can vary widely from one system to another and performance is also affected by the type of network interface, the client computer operating system, and other factors. ENET devices are real-time, meaning Ethernet UDP packets are being turned-around/transmitted in 10-20 uSec! Your client computer will not be that fast and your computer and LAN configuration will be the limiting factor with high packet rate applications! (USB offers even worse limitations as most computer systems limit USB polling rates to 1-8 mSec where Ethernet is usually 10x or faster for packet rate processing).

The recommended configuration is a point-to-point connection (for any real-time Ethernet application) or a good clean LAN system between the client computer and the ENET device. Managed switches often need to setup to handle ENET static IP addressing and provide a high priority for ENET UDP port (see UDP Protocols later in the manual for port information). Gigabit Ethernet provide the best results for your computer. It is NOT recommended to use anything slower than 100Mbit Ethernet. Wireless connections will

usually drop packets and this is NOT recommended unless you characterize your LAN/WIFI system.

The example program ***ADT_L1_1553_ex_bcbm1_Timing_Test.c*** can be used to characterize the performance of your system and network. Note that this program can be used with any Alta device, not just ENET devices. Therefore you can measure the performance with PCI or PCI Express devices if desired. This program calculates the following metrics:

- 32-bit Word Read Time – this is “Path Delay” for ENET devices
- 32-bit Word Write Time
- BC CDP Read Time
- BC CDP Write Time
- BM CDP Read Time

The key value is the 32-bit Word Read Time or **Path Delay** for ENET devices. This Path Delay is the total time it takes for the user program to initiate a read operation, which the API executes as an ADCP transaction, where a command packet is sent through the operating system network stack to the NIC to be transmitted on the network, then waits for the response packet from the ENET device, which must be received by the NIC, processed back up through the operating system network stack and returned to the API, which extracts the read data and returns it to the user application.

The other metrics (32-bit Word Write Time, BC CDP Read Time, BC CDP Write Time, and BM CDP Read Time) are informative and can be useful, but will not be discussed in detail here. We will focus on Path Delay for the discussion of system and network performance with ENET devices.

The Path Delay for your system and network will determine how fast you can get data from the ENET device. For the ENET-1553, this determines how many 1553 messages per second you can keep up with. For 1553 the most demanding area is the Bus Monitor because CDP message buffers must be read as fast as possible to prevent data loss. Remote Terminal and Bus Controller applications are generally not as demanding in terms of the amount of data that needs to be moved.

Note that the **AltaView** bus analyzer software measures Path Delay for ENET devices and displays Path Delay and ENET statistics in the Device Info form.

Measuring the Path Delay of your system and network gives you a snapshot of your system performance at the time the measurement was performed. This does NOT account for varying conditions on the network, periods of heavy network traffic, or other variables that affect the network. Therefore the Path Delay is a GENERAL INDICATOR of the performance you can expect with the ENET device on your system and network. ALTA CANNOT AND DOES NOT GUARANTEE ERROR-FREE OPERATION ON YOUR NETWORK.

A majority of network performance is based on packets per second (so bit rate is a factor, but computer systems usually are limited to how many packets per second they can process). The following formula can be used to estimate the number of 1553 messages per second that you can expect to keep up with (as Bus Monitor) on your network with an ENET-1553 device:

$$\text{MsgsPerSecond} = 1 / (7 * \text{PathDelayInSeconds})$$

For example, if we have a Path Delay of 100 microseconds then we can expect to keep up with a message rate of 1400 messages per second. Note that this is a general, **very** conservative estimate, not an absolute boundary – you can monitor data at higher message rates but you may start to drop messages if you cannot read the messages faster than they are received on the 1553 bus. High message rates can also cause the software to slow down or lock up because it is continuously processing UDP packets trying to read the messages.

ENET Performance

Alta has provided a lot of guidance and warnings on your computer's path delay and LAN delays or dropped WIFI packets, but in most applications, ENET provides a fantastic, reliable, real-time Ethernet to 1553/ARINC interface. 95% of applications will not even notice that ENET is connected to their application verses a normal PCI or PCI Express card interface. And with complete code portability and elimination of device driver issues, ENET is a game changer for flexible system configurations.

ENET Application Performance Optimization

ENET applications that need to work around a long Ethernet path delay, or operate at fast update rates, may need to optimize their application to use direct data structure access (use of on-board memory addresses instead of abstract data structure references) in order to bypass many of the pointer lookups in the AltaAPI L1 calls. MOST applications DO NOT need this step. Direct access control requires the developer to be familiar with the low level definitions of AltaCore structures and some of the inner workings of the AltaAPI source. Alta

has example programs on how to perform this for most applications. Please contact Alta and we'll be glad to work with you for the best approach to building your application.

Advanced ENET Auto-Setup and Control without API Control

Most Alta products contain boot-up flash (please check your product as PCMCIA and Expresscards do not have flash) that can auto-setup BC, RT and BM data structures (buffers). This can be very useful with ENET where you want to remote the device and have it auto-boot with a setup image. Then you can use AltaAPI or straight UDP accesses (BSD sockets) to read/write memory. For example, it is common to have a simple RT with only a handful of subaddress buffers needed. ENET could auto-boot with these buffers pre-defined and then your code could use straight OS UDP services to read/write buffers. This feature can greatly simplify system architectures and sometimes remove expensive code verification steps. The flash auto-image boot requires detailed knowledge of card memory setup and Alta would be glad to assist in setting up your device. Please contact alta.support@altadt.com for more information.

ENET Ethernet UDP Protocols

ENET devices utilize real-time Alta UDP protocols used with Alta ENET products: Alta Device Control Protocol (**ADCP**) and Alta Passive Monitor Protocol (**APMP**). These protocols are used to communicate with Alta ENET devices over an Ethernet network (instead of a computer backplane like PCI or PCIe).

Both UDP protocols are “real-time” meaning that the ENET device is servicing and transmitting UDP packets in less than 20 uSec on the Ethernet wire. The customer’s client computer network interface, OS IP stack and LAN configuration is the source of all other possible “path delays” in the communication link. Alta has sample API programs that the user can run to model their network system.

REFERENCES

- AltaAPI Users Manual
- IETF RFC 768 User Datagram Protocol
- IETF RFC 791 Internet Protocol v4
- IEEE 802.3 Ethernet

ALTA DEVICE CONTROL PROTOCOL (ADCP)

The Alta Device Control Protocol (ADCP) is used by the AltaAPI to control ENET devices, primarily through memory read/write accesses (that mimic backplane memory transfers). This is accomplished by a simple handshake of UDP packets

where an Alta header in the the UDP payload provides memory read and write control and status information. This ADCP UPD protocol is detailed in the following paragraphs.

NOTE: The ADCP protocol is abstracted/managed in the AltaAPI and for most customers they do not need to the lower level details. These ADCP paragraphs are provided for reference only and most customers can skip this section.

The Alta ENET device is an ADCP Server (memory server for ENET protocol engine memory – described in AltaCore manuals), and only responds to commands from a controller Client (customer's computer). The Client initiates all transactions and the Server ENET device only sends packets in response to commands from the Client. The Client sends only one command per transaction and the transaction is only completed by a response or time-out action from the Server's response

The ENET device only buffers one UDP packet at a time to ensure an handshake cannot get interrupted (at the ENET device – it is up to the Client software to ensure it processes the correct handshake packet, which can be checked with an Alta header sequence number in the packet).

Basic Operations

The controller client performs all operations with single UDP packets, where each packet is a command to the device. The basic operations are:

1. Device Reset
2. Read from device main memory (16-bit or 32-bit)
3. Write to device main memory (16-bit or 32-bit)
4. Read from device setup memory (32-bit)
5. Write to device setup memory (32-bit)

The 32-bit reads/writes in main memory are used for normal ENET device operation (as described in the respective AltaCore manuals). The 16-bit reads/writes in main memory and the 32-bit reads/writes in setup memory are only used when programming the Protocol Engine firmware load into flash memory. For comparison to Alta board-level products, the Alta PCI/PCIe devices map main memory to BAR2 and map setup memory to BAR0.

The ENET device provides the echo handshake request (with or without memory requested) in less than 20 uSec on the Ethernet wire. The Client's computer or network configuration is the source of all other "path delays" in the handshake.

Error Detection and Error Handling

The Ethernet packet includes a Frame Check Sequence using a 32-bit Cyclic Redundancy Check (CRC). This is generated for transmitted packets and checked for received packets at the MAC level. If the Server detects any MAC errors (including CRC errors) in the command packet from the Client, the Server will not process the command and it will send a response packet to the Client with a failure status code. Note that this CRC checksum is present in the top-level TCP-IP header and is not a checksum for the UDP data only.

The ADCP packet header contains a Sequence Number which is generated by the controller Client. The sequence number is key to the ADCP protocol and is verified by the client to detect dropped packets. The Server ENET device responds to a command by sending a response packet back to the Client with the same Sequence Number that it received in the command packet. This response packet is the acknowledgement from the Server to the Client to indicate that the memory command was received.

If the Client receives a response from the Server with the wrong Sequence Number, with a failure status code, or if the Client does not receive a response at all (and times-out), it can send the command again.

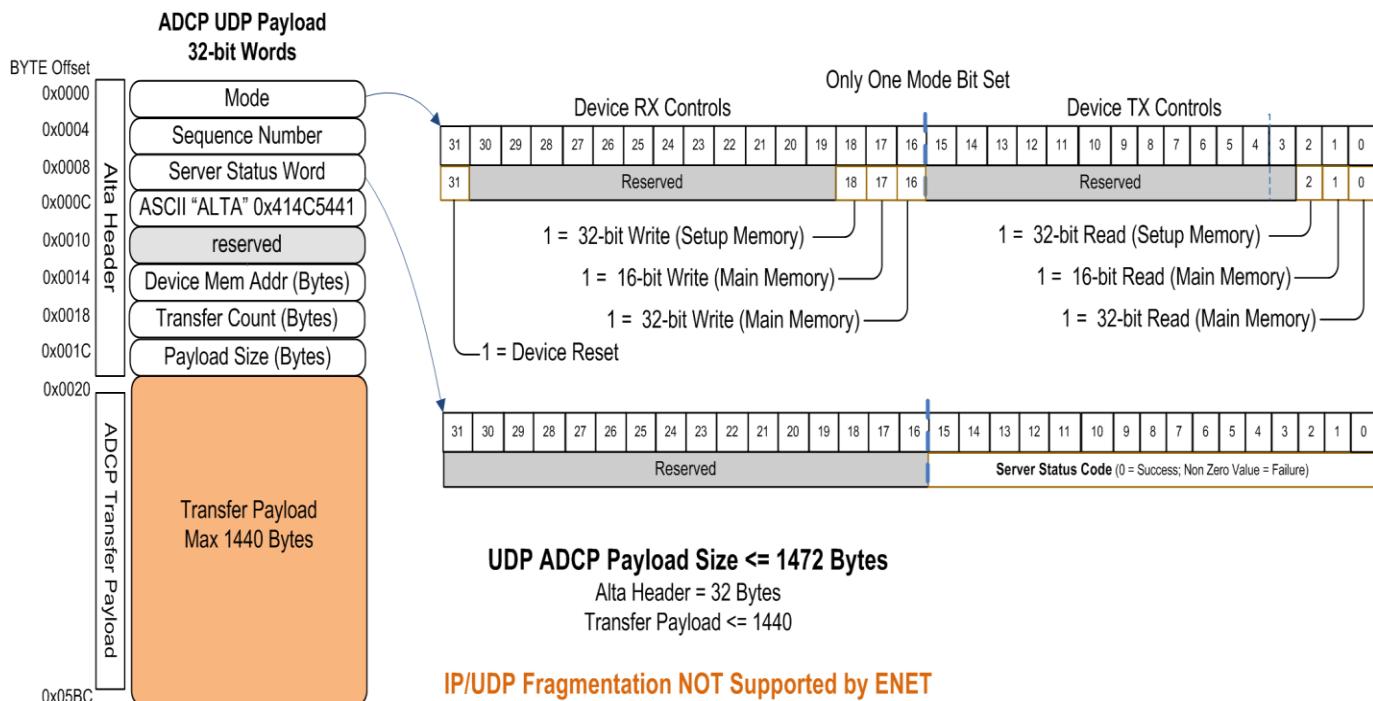
The time-out period to wait for a response from the Server is determined by the client application, but a recommended value is a minimum of 50 milliseconds. Likewise, the maximum number of times that the client will retry sending a command packet is determined by the application, but a recommended number is three retries. Alta provides the C source code for the client Layer 0 API to show this standard BSD socket management for client computers.

UDP Port Numbers

The ADCP Server will always be UDP port number 55512. ADCP Clients will start at UDP port number 55513. The AltaAPI, if the port is already in use the Client will increment the port number until it finds one that is available.

ADCP Packet Format

The following diagram shows the UDP ADCP packet format:



The first 8 words (16 bytes) make up the Alta Header, and consist of a Mode word, a Sequence Number, Server Status Word, an alignment/market word that is ASCII “ALTA”, a reserved word and then the Device Memory Byte Address (Addr), Transfer Byte Count, and Payload (raw memory data) Byte Count. The Transfer Payload is the raw memory that is read/written per the definition of the Alta Header words.

All values in the packet are in “Big-Endian” (network) format. (The client will want to use standard ntohs or htonl type function calls to provide proper Endian formats for their computer/OS).

The 32-bit value of the ASCII Alta marker word is 0xA1B2C3D4 and would be stored as follows:

First Byte: 0xA1
Second Byte: 0xB2
Third Byte: 0xC3
Fourth Byte: 0xD4

The controller Client (user’s computer probably using the AltaAPI functions) sends memory read or write command packets to the Server (ENET). The Server performs the requested operation (as specified in the Mode word) and

sends a response packet back to the Client. The response packet includes the Server Status Word, which indicates (AltaAPI) ADT_SUCCESS (zero) or ADT_FAILURE (one). If the Server detected any errors in the command packet it will indicate ADT_FAILURE in the response packet.

The ENET Server response packet shall use the same Sequence Number that it received in the corresponding command packet (this should be verified to ensure packet sequence integrity – and is checked the AltaAPI Layer 0).

For memory read/write operations, the ADCP header includes the starting byte address (Device Mem Addr) and the number of bytes (Transfer Count). For memory write operations, the Client sends the words to be written in the Transfer Payload of the command packet. For memory read operations, the Server sends the words read in the Transfer Payload of the response packet. As with all words in the packet, the Transfer Payload will store 32-bit and 16-bit words in “Big-Endian” format.

Here are some examples:

(Please remember that this process is abstracted in the Layer 1 and 0 of the AltaAPI and most customers will not need to know these low level processes).

If the Client wants to read 49 32-bit words from main memory starting at byte address 0x10000 it would send a command packet with a Mode word of 0x00000001 (32-bit read from main memory), the next sequence number (let's use 0x12345678 for this example), zero in the next four words of the header, a Device Mem Addr of 0x00010000, and a Transfer Count of 0x000000C4 ($49 * 4 = 196 = 0xC4$). There would be no words in the Transfer Payload of the command packet, so the Payload Size will be zero.

When the Server receives this packet it will perform the read operation and send a response packet back to the Client. The Mode word will be 0x00000001 (same as command packet), the Sequence Number will be 0x12345678 (same as command packet), assuming there were no errors the Server Status will be 0x00000000 (indicating SUCCESS), the Device Mem Addr will be 0x00010000 (same as command packet), and the Transfer Count will be 0x000000C4 (same as command packet). The Payload Size will also be 0x000000C4 and the Transfer Payload will contain the words read from memory.

If the Client wants to write 49 32-bit words to main memory starting at byte address 0x10000 it would send a command packet with a Mode Word of 0x00010000 (32-bit write to main memory), the next sequence number (let's use 0x12345679), zero in the next four words of the header, a Device Mem Addr of 0x00010000, a Transfer Count of 0x000000C4, and a Payload Size of 0x000000C4, with the data to be written in the Transfer Payload.

When the Server receives this packet it will perform the write operation and send a response packet back to the Client. The Mode word will be 0x00010000 (same as command packet), the Sequence Number will be 0x12345679 (same as command packet), assuming there were no errors the Server Status will be 0x00000000 (indicating SUCCESS), the Device Mem Addr will be 0x00010000 (same as command packet), and the Transfer Count will be 0x000000C4 (same as command packet). The Payload Size will be zero because there are no words in the Transfer Payload.

Auto 1553/ARINC Ethernet BM/RX Bridging: ALTA PASSIVE MONITOR PROTOCOL (APMP)

The Alta Passive Monitor Protocol (APMP) is provided by ENET devices to auto transmit 1553 bus monitor CDPs or ARINC RXP (Receive) data to Ethernet (thus, provides an auto bridge of time-stamped 1553 or ARINC data to Ethernet). This is an automatic action from the ENET device, and once setup by the customer (via ADCP), this action does not require any client handshake (even from power-up).

The Alta ENET device is the APMP Server, which transmits UDP packets as it receives messages on the data bus (1553, A429, etc.). Clients (any computer on the Ethernet LAN) only receive packets. The ENET server only transmits APMP packets, it does not receive packets. The AltaAPI is not used in this mode to receive auto broadcast APMP packets – simply straight BSD sockets (or whatever computer utility) that can read UDP packets.

UDP Port Numbers

Each ENET device will use a fixed pair of Server and Client port numbers. These port numbers will be programmed into flash memory on the device when it is configured for APMP operation (these values can be changed if needed – contact Alta support for more information).

	Device 1	Device 2	Device 3	Device 4
ENET-1553 APMP Server Port Number	56512	56514	56516	56518
ENET-1553 APMP Client Port Number	56513	56515	56517	56519
ENET-A429 APMP Server Port Number	56768	56770	56772	56774
ENET-A429 APMP Client Port Number	56769	56771	56773	56775

The 2-channel ENET2-1553 will be TWO devices (one for each channel). Each device needs to be configured as a different device number.

There is only one operation for APMP: transmit a UDP packet when either a 1553 message or group of ARINC RXPs (Receive Packets) has occurred. By

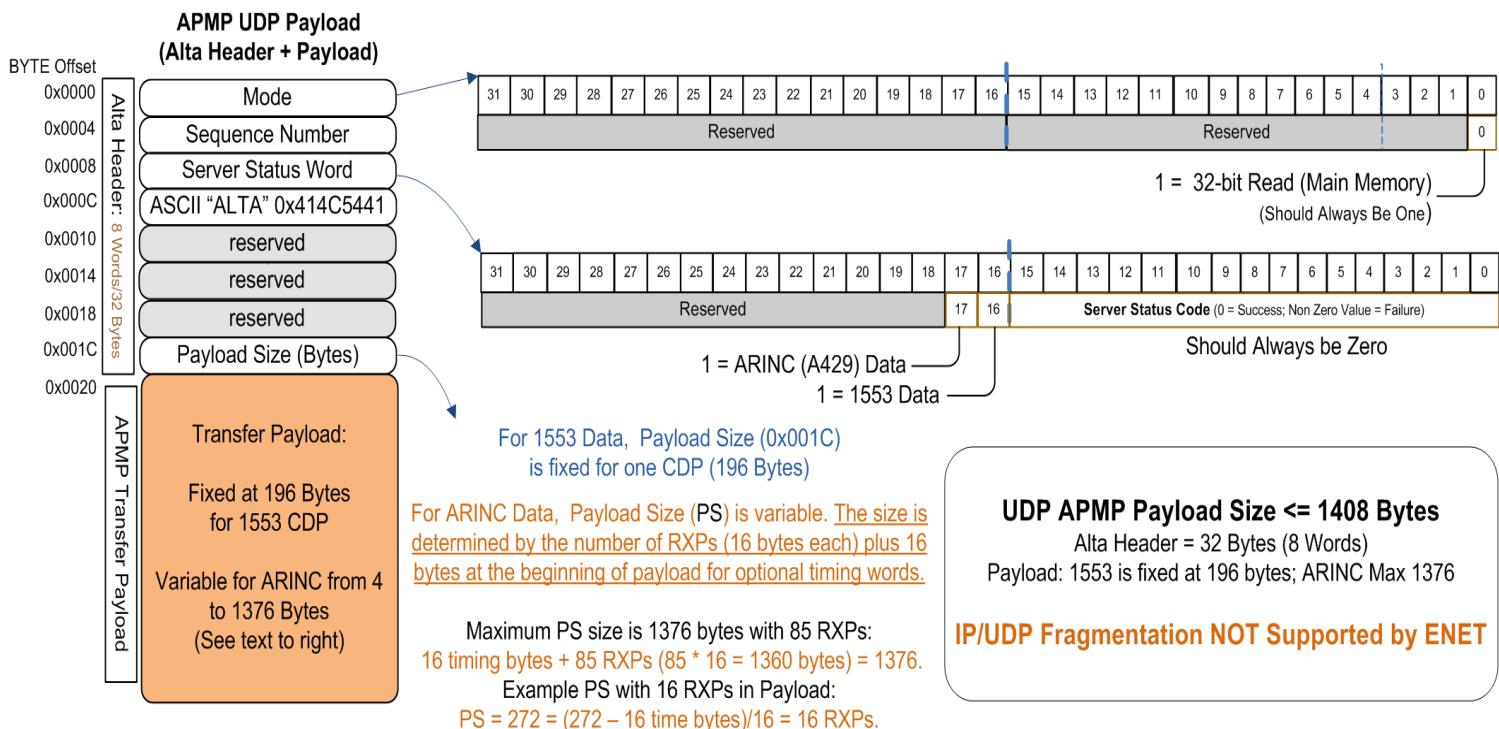
default, the ENET Server sends the APMP UDP broadcast with IP destination address of 255.255.255.255. This is a value stored in flash and can be changed by the user with the AltaView ENET Config tool (as a Global device).

Error Detection and Error Handling

The Ethernet packet includes a Frame Check Sequence using a 32-bit Cyclic Redundancy Check (CRC). This is generated for transmitted packets and checked for received packets at the MAC level. If the Client detects any MAC errors (including CRC errors) in the APMP packet from the Server, the Client will discard the packet and will not use the data.

APMP Packet Format

The following diagrams show the general APMP packet:



The APMP packet format is very similar to the ADCP packet format. The Mode word should always be 0x00000001 (32-bit Read of main memory). The Server Status Word should always have 0x0000 (Success) in the Server Status Code and indicates the Payload Data Type (1553 message or ARINC/A429) in the upper 16 bits. The Payload Size word contains the number of bytes in the Transfer Payload.

All values in the packet are in “Big-Endian” format.

The 32-bit value ASCII ALTA word is 0xA1B2C3D4 would be stored as follows:

First Byte: 0xA1

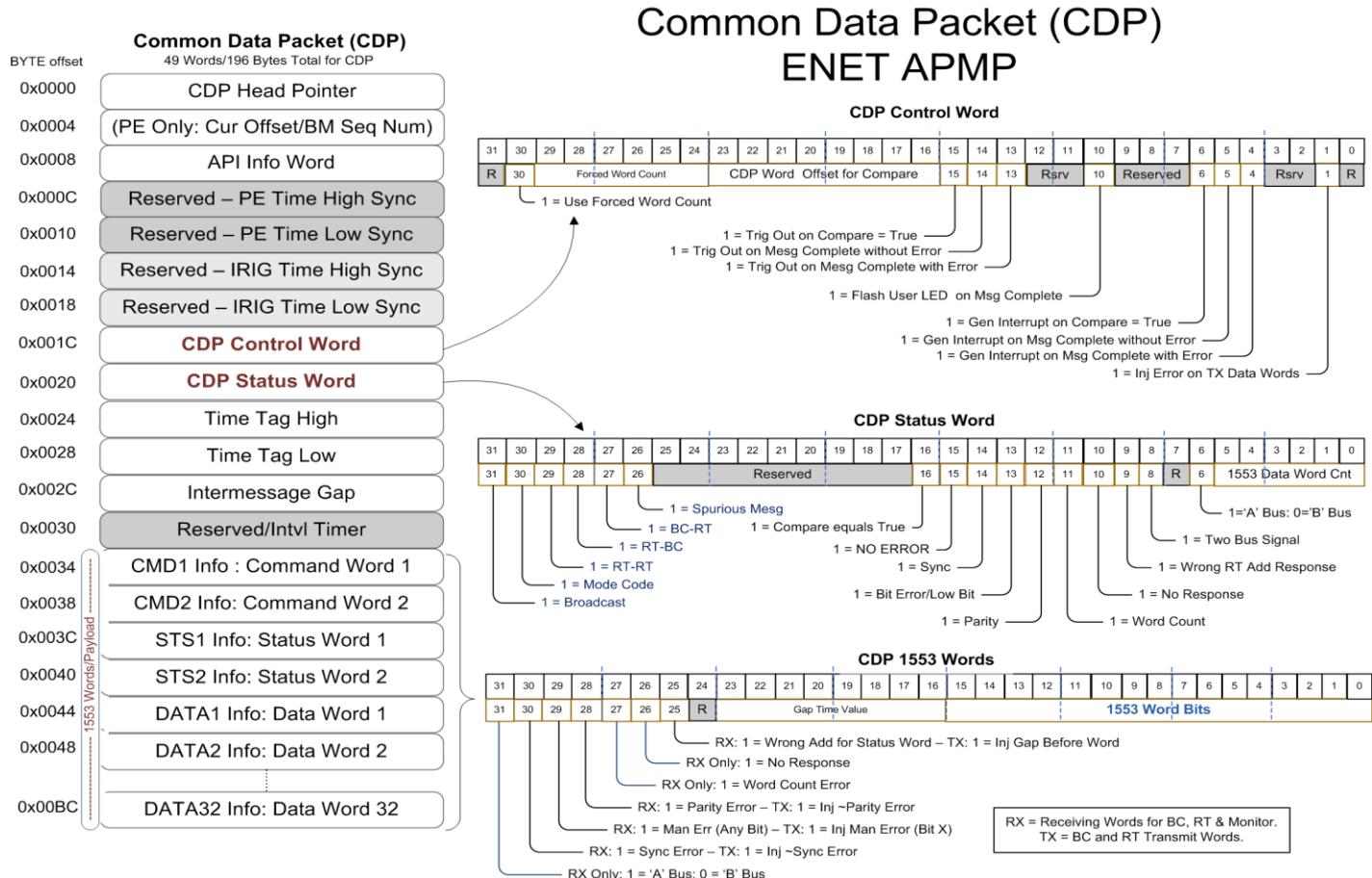
Second Byte: 0xB2

Third Byte: 0xC3

Fourth Byte: 0xD4

APMP 1553 CDP Packet Format - CDP

For 1553 APMP packets, the APMP Tranfer payload contains a 196 byte/49 word Common Data Packet (CDP) per the following figure.



How to Start/Control 1553 APMP

The APMP mode is controlled through the Sequential Monitor Control Word (API Function: **ADT_L1_1553_BM_Config**) by seting the following definitions.

ADT_L1_1553_BM_CSR_ENET_APMP_ENABLE 0x00000100 ENET APMP Enable

This bit is set by the user to enable the APMP auto CDP/UDP broadcasting. When set, the ENET-1553 will auto broadcast CDP/UDP packets on Ethernet connection

ADT_L1_1553_BM_CSR_ENET_APMP_PEIRIG 0x00000200 ENET APMP Insert PE/IRIG Time

This bit is set by the user to direct the PE to insert the last latched PE and IRIG Time values in to APMP CDP word offsets 0x0C through 0x18. This allows the users Ethernet programs listening to APMP packets to have PE+IRIG absolute

time along with the normal CDP Time High/Low relative time (to time sync data capture). This bit must be set with Bit

ADT_L1_1553_BM_CSR_ENET_APMP_ENABLE. The time values would be the same as setting the Read IRIG control bit in the Root PE Control Word (0x0000).

ADT_L1_1553_BM_CSR_ENET_APMP_PEINTV 0x00000400

ENET APMP Insert PE/Interval Timer

This bit is set by the user to direct the PE to insert the last latched PE and Interval Timer (user PPS or clock) in to APMP CDP word offsets 0x0C through 0x18. Ethernet programs listening to APMP packets to have PE+ Interval/PPS absolute time along with the normal CDP Time High/Low relative time (to time sync data capture). This bit must be set with Bit

ADT_L1_1553_BM_CSR_ENET_APMP_ENABLE. The time values would be the same as setting the Read Time control bit in the Root PE Interval Timer register (0x004C).

Once the BM is started, then the ENET will automatically start sending APMP UDP CDP packets within 10 uSec after the completion of the 1553 message.

There are example programs provided to show setup and reading of APMP packets.

ENET APMP 1553 CDP Format: PE + IRIG or Interval Timer Inserts

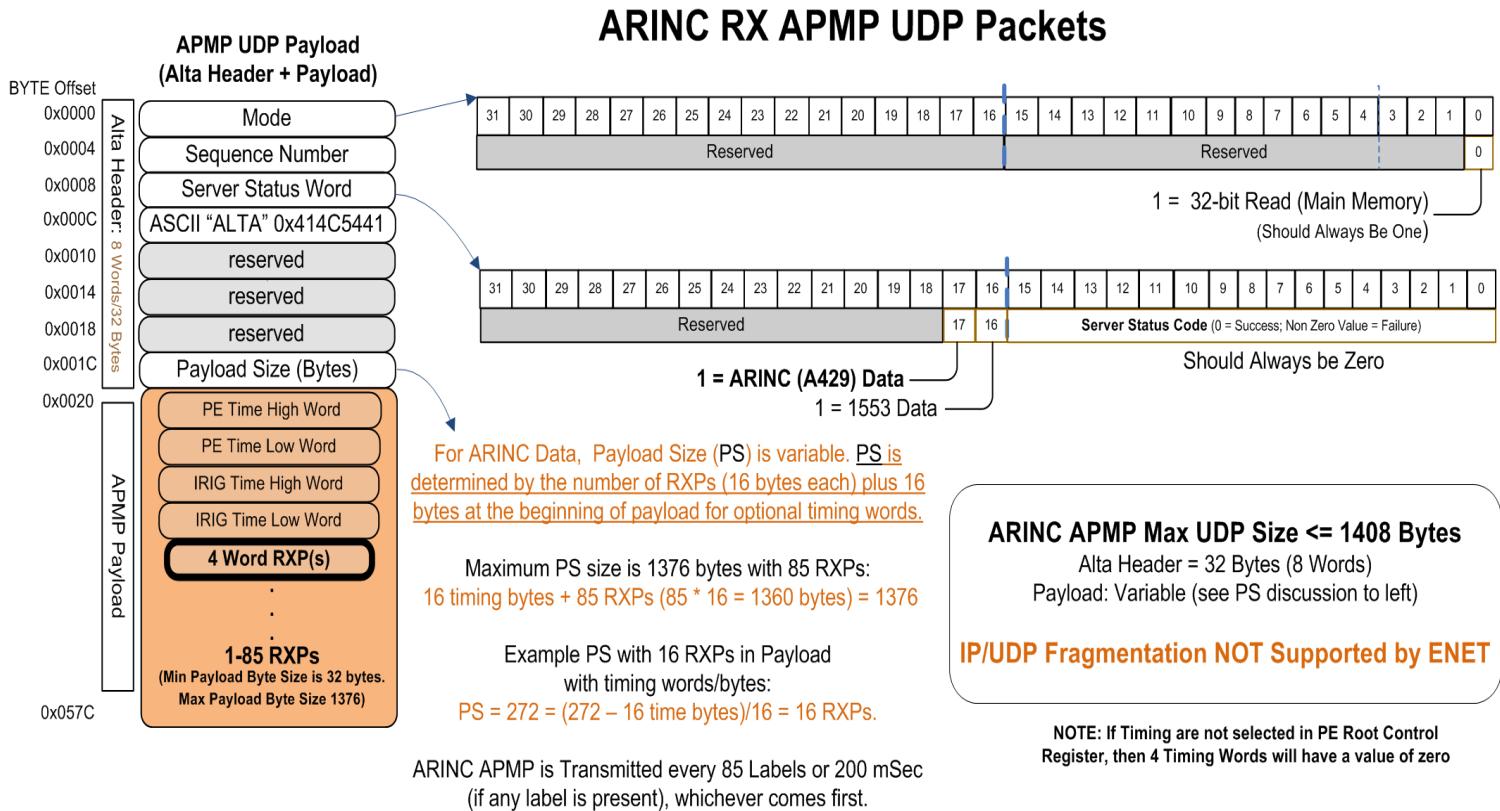
For ENET-1553 product, the CDP can have a different format if the user selects the APMP Insert PE/Intrvl or PE/IRIG Time options in the Root SM CSR registers. These time formats are the SAME as if IRIG time is read from the Root PE Control register (for PE + IRIG Time) or the Read Timer in the Root Interval Timer Control Register (Interval Timer word would in place of IRIG Time High). All other CDP values are the same as normal SM/BM functions. Please see the following bookmarks in the AltaCore-1553 manual for full details.

Root IRIG Time High & Low: 0x0024/28

Root Interval Timer: 0x004C

ARINC APMP Packet Format

The following figure shows the ARINC APMP Packet format.



For A429 payload, the Transfer Payload will contain the Alta A429 Receive Packet (RXP), which is 4 words (16 bytes) plus 4 fixed time words at the beginning of the payload. The time words are selected in AltaCore-ARINC Root PE Control Word for PE + IRIG. If not selected, these words are set to zero. ARINC APMP will sent every 200 mSec or 85 labels from all selected channels, whichever comes first.

Please see the AltaCore-ARINC Receive section for details on ARINC RXPs (simple 4 word array that includes a control word, 2 time-stamp words and one word with raw ARINC data).

How to Start/Control ARINC APMP

The APMP mode is controlled through the Root PE CSR Register (0x0000 offset). The user should use an ADT_L1_A429_Read/WriteDeviceMem32() function to set the following option bits. In addition, the customer will set the `ADT_L1_A429_RXREG_SETUP1_MCRX` option when the Receive channel is initialization (in `ADT_L1_A429_RX_Channel_Init()` function).

ADT_L1_A429_PE_CSR_ENET_APMP_ENABLE 0x00000100 ENET APMP Enable

This bit is set by the user to enable the APMP auto CDP/UDP broadcasting. When set, the ENET-1553 will auto broadcast CDP/UDP packets on Ethernet connection

ADT_L1_A429_PE_CSR_ENET_APMP_PEIRIG 0x00000200 ENET APMP Insert PE/IRIG Time

This bit is set by the user to direct the PE to insert the last latched PE and IRIG Time the first 4 words of the APMP Transfer Payload (shown above). This allows the users Ethernet programs listening to APMP packets to have PE+IRIG absolute time along with the normal RXP Time High/Low relative time (to time sync data capture). This bit must be set with

ADT_L1_A429_PE_CSR_ENET_APMP_ENABLE. The time values would be the same as setting the Read IRIG control bit in the Root PE CSR Word.

ADT_L1_A429_PE_CSR_ENET_APMP_PEINTV 0x00000400 ENET APMP Insert PE/Interval Timer

This bit is set by the user to direct the PE to insert the last latched PE and Interval Timer (user PPS or clock) in to first 3 words of the APMP Transfer Payload (shown above – Interval Value replaces IRIG Time High). Ethernet programs listening to APMP packets to have PE+ Interval/PPS absolute time along with the normal RXP Time High/Low relative time (to time sync data capture). This bit must be set with

ADT_L1_A429_PE_CSR_ENET_APMP_ENABLE. The time values would be the same as setting the Read Time control bit in the Root PE Interval Timer register.

Once the receive channel(s) are started, then the ENET will automatically start sending APMP UDP RXP packets within 10 uSec after the completion of the last RXP of the APMP packet. **There are example programs provided to show setup and reading of APMP packets.**

Here is a code snippet for updating the ARINC Device CSR with APMP options:

```
ADT_L0_UINT32 data, status;

/* Get Current CSR */
status = ADT_L1_ReadDeviceMem32(DEVID, ADT_L1_A429_PE_ROOT_CSR, &data, 1);
if (status == ADT_SUCCESS)
{
    /* Update with APMP Options - "OR" options together */
    data |= ADT_L1_1553_BM_CSR_ENET_APMP_ENABLE;

    /* Update CSR */
    status = ADT_L1_WriteDeviceMem32(DEVID, ADT_L1_A429_PE_ROOT_CSR, &data, 1);
    if (status != ADT_SUCCESS)
        printf("\nERROR on WRITE %d", status);
}
```

ENET APMP ARINC Format: PE + IRIG or Interval Timer Inserts

For ENET-A429 product, the first 4 words of the APMP Transfer Payload (shown above and enabled with the CSR bits described on the previous page) can have the PE 64-bit time and either IRIG or Interval Timing word (word 3 only in place of IRIG Time High). These time formats are the SAME as if IRIG time is read from the Root PE Control register (for PE + IRIG Time) or the Read Timer in the Root Interval Timer Control Register. . Please see the following bookmarks in the AltaCore-ARINC manual for full details.

Root IRIG Time High & Low: 0x0024/28

Root Interval Timer: 0x004C

MIL-STD-1553 Layer 1 Programming

AltaAPI Layer 1 (L1) programming follows the same steps as described in the last section. A brief example for an RT function will be provided and then this section will detail L1 programming of each major function of Alta 1553 Devices, including:

- 1553 Device Initialization and Closure
- Bus Controller (BC)
- Remote Terminal (RT)
- Sequential Monitoring (Bus Monitor - BM)
- Playback (PB)
- Interrupt Handling
- Signal Generator (SG)
- Signal Capture (SC)

Here is a simple program (without error checking) of an RT that would have all Receive and Transmit Subaddresses and Mode Code buffers automatically wrapped.

```
#include <stdio.h>
#include "ADT_L1.h"

/* DevID = 1553 Device Channel Code from ADT_L0.h */
#define DEVID      (ADT_PRODUCT_PMC1553 |  
                 ADT_DEVID_BOARDNUM_01 |  
                 ADT_DEVID_CHANNELTYPE_1553 |  
                 ADT_DEVID_CHANNELNUM_01)

int main()  
{  
    ADT_L0_UINT32      status, RTnum = 1;  
  
    status = ADT_L1_1553_InitDefault(DEVID, 10);  
    status = ADT_L1_1553_RT_Init(DEVID, RTnum);  
    status = ADT_L1_1553_RT_Enable(DEVID, RTnum);  
    status = ADT_L1_1553_RT_Start(DEVID);  
    /* Your application here */  
    status = ADT_L1_1553_RT_Stop(DEVID);  
    status = ADT_L1_1553_RT_Close(DEVID, RTnum);  
    status = ADT_L1_CloseDevice(DEVID);  
}
```

This program would not really do anything except initialize the device, initialize the RT, enable and start the RT and then close it – this is not a practical

application, but these are the basic steps for most 1553 functions (just substitute RT with BC or BM, etc...). The following sections will detail each step.

A key data structure for any RT, BC, BM or PB function is the Common Data Packet (CDP) – you should review this data structure in the ADT_L1.h and AltaCore-1553 manual. The CDP is reviewed in later 1553 sections of this manual. This is powerful part of the Alta design – one common data structure for any 1553 function (a first in the industry and can greatly simplify 1553 programming to more resemble a general network device).

1553 (M1553) Device & Protocol Initialization & Closing

Initialization functions specific to the MIL-STD-1553 protocol are provided in the file ADT_L1_1553_General.c.

The L1 API provides the low level steps to allow customization of the initialization and protocol setup, but most customers will use one of two functions: These functions combine low level setup (memory mapping) and device/protocol steps for standard 1553B protocol in a single function call (these are used by 99% of 1553 applications).

- **ADT_L1_1553_InitDefault()** – **This is the recommend function!**

This function performs memory tests and will error out if the 1553 device has not been closed properly. The memory test of this function can take seconds, but overall, this function is the safest and best to use for test applications.

```
status = ADT_L1_1553_InitDefault(DEVID, 10);
if (status == ADT_SUCCESS)
{
    // Continue Application
}
else printf("FAILURE - Error = %d\n", status);
```

- **ADT_L1_1553_InitDefault_ExtendedOptions()**

This function provides the user options to control initialization with options to hard reset the PE (device channel) low-level registers, override previous bad shutdowns and bypass memory tests. This function can provide very fast startup (usually <50-100 msec), but can also require the application to better manage shutdowns and memory de-allocations.

```
status = ADT_L1_1553_InitDefault_ExtendedOptions(DEVID,
ADT_L1_API_DEVICEINIT_FORCEINIT |
ADT_L1_API_DEVICEINIT_NOMEMTEST |
ADT_L1_API_DEVICEINIT_ROOTPERESET);
if (status == ADT_SUCCESS)
{
    // Continue Application
}
else printf("FAILURE - Error = %d\n", status);
```

Note: All the initialization function call examples allocate memory for an interrupt queue with a depth of **10** entries. The interrupt queue depth is set by the user and is determined by how often the application expects the device to generate interrupts and how

quickly the application will be able to service interrupts. This is discussed further in the section on interrupt operation.

The Extended “options” are defined as follows:

```
#define ADT_L1_API_DEVICEINIT_FORCEINIT      0x00000001
    Forces Initialization Regardless of Current API State. Often
    used from application crashes or incorrect closing of
    application.

#define ADT_L1_API_DEVICEINIT_NOMEMTEST      0x00000002
    Skips API Memory Test and Initialization (that can take
    several seconds).

#define ADT_L1_API_DEVICEINIT_ROOTPERESET     0x80000000
    Forces a hard reset of the device channel (not the card).
    This clears all 1553 low level control registers and halts any
    transmission (BC or RT Responses) and reception of data.

#define ADT_L1_API_DEVICEINIT_NOKP           0x00000004
    Skips loading of the interrupt kernel plug-in. Not
    recommended for most applications and not shown.
```

The ADT_L1_API_DEVICEINIT_NOKP option can be used to bypass loading the driver kernel plug-in but in most cases this option should NOT be set. This option only applies to platforms that use the Jungs WinDriver software for the device driver (Windows, Linux, Solaris). If this option is used then the application cannot use hardware interrupts. The kernel plug-in is required for hardware interrupts.

The ADT_L1_API_DEVICEINIT_FORCEINIT option should ONLY be used in development and testing. This option is provided for cases where the device may not have been closed properly and is used to override the ADT_ERR_DEVICEINUSE error. This option should NOT be used as the normal initialization method for your application, because it bypasses protection against two applications using the same device.

Advance 1553 Initialization Options – Manual Device & Protocol Setup (1553A and Other Non 1553B Variants).

The “Default” functions described above provide the setup for standard 1553B protocols, and these functions actually call the following 3 functions for you to simplify setup:

- ADT_L1_InitDevice() – Basic Device Connection to Driver
- ADT_L1_1553_InitChannel() – Initialization of 1553 Protocol Engine
- ADT_L1_1553_SetConfig() – Sets key 1553 Protocol Settings

After the API is initialized (ADT_L1_InitDevice) for the device, the 1553 channel must be initialized and configured. The ADT_L1_1553_InitChannel function initializes the 1553 channel registers and allocates the interrupt queue.

```
/* Initialize the CHANNEL, allocate interrupt queue */
printf("Initializing Channel and Interrupt Queue. . . ");
status = ADT_L1_1553_InitChannel(DEVID, 10);
if (status == ADT_SUCCESS) printf("Success.\n");
else printf("FAILURE - Error = %d\n", status);
```

The ADT_L1_1553_SetConfig function sets protocol options, selects internal or external bus, and sets the status response timeout value (used by BC and BM).

```
/* Set 1553B, RT31 is BROADCAST, external bus, 14us timeout */
printf("Setting options . . . ");
status = ADT_L1_1553_SetConfig(DEVID, 1, 1, 1, 0, 14);
if (status == ADT_SUCCESS) printf("Success.\n");
else printf("FAILURE - Error = %d\n", status);
```

These functions perform the low level initialization, 1553 channel setup and custom protocol configuration. If you need to initialize with non 1553B protocol settings (such as older 1553A), then please review these functions in the reference section of this L1 section to decided what values to provide these functions.

“RT Live” Initialization – MIL-STD-1760 Startup

If the channel is configured to come up “live” as a remote terminal (using the external RT address lines), then use the ADT_L1_1553_InitChannelLive function. This function is basically identical to the ADT_L1_1553_InitChannel function, but does not clear the RT control blocks, so there will be a much shorter transition period (when the RT will stop responding) as the application takes over as the RT. There is an example program of RT Live Startup provided in the distribution.

Closing the Device – Last Step of an Application

At the end of an application, or when the application no longer needs to use the Alta device it can close the API as follows:

```
ADT_L1_CloseDevice(DEVID);
```

This function frees resources, closes memory management, un-maps memory, and detaches from the device. No API calls should be made for the device after the ADT_L1_CloseDevice call has been made. The device must be initialized again before use.

Overview of MIL-STD-1553 Functions

Once the 1553 device (channel) has been initialized per the previous pages, the user's application is ready to setup 1553 functions. One of the best references to get started quickly are the numerous MIL-STD-1553 (M1553) example programs provided – please see these programs for quick start of your application (locations given in previous pages).

The following pages will detail each of the major 1553 functions and how to setup data structures, read and write data and manage timing. Before starting each section, though, we should introduce a key data structure that is used for all BC, RT, BM and Playback functions – the Common Data Packet (CDP).

The 1553 Common Data Packet (CDP)

Alta 1553 devices use a “Common Data Packet” (CDP) data structure for Bus Monitor (BM), Remote Terminal (RT), Bus Controller (BC) data buffers and Playback API functions. This offers a huge advantage over older 1553 designs as the one common data structure is utilized for data buffering of BC, RT and Monitor functions – no longer do you need to have 6, 10 or more different data structures for each function; this can greatly simplify your software management and makes 1553 packets more like a standard network UDP type packet.

Each CDP structure contains information on one 1553 message. The CDP controls most buffer functions such as error injection/detection, interrupts (including mask/compare interrupts per CDP), CDP and intermessage time stamps. This complete information allows the RT, BC or Monitor application to have a complete snapshot of all message information/status (most older designs would only provide partial message information depending on which mode was being used)

The Layer 1 API represents the CDP with the structure ADT_L1_1553_CDP, which is defined in the ADT_L1.h header file.

```
typedef struct adt_l1_1553_cdp {
    ADT_L0_UINT32 NextPtr;                                /*!< \brief CDP next pointer */
    ADT_L0_UINT32 BMCCount;                             /*!< \brief BM message count */
    ADT_L0_UINT32 APInfo;                               /*!< \brief Reserved for API */
    ADT_L0_UINT32 Rsvd1;                                /*!< \brief Reserved */
    ADT_L0_UINT32 Rsvd2;                                /*!< \brief Reserved */
    ADT_L0_UINT32 MaskValue;                            /*!< \brief Mask value */
    ADT_L0_UINT32 MaskCompare;                          /*!< \brief Mask compare value */
    ADT_L0_UINT32 CDPCControlWord;                     /*!< \brief CDP control word */
    ADT_L0_UINT32 CDPStatusWord;                        /*!< \brief CDP status word */
    ADT_L0_UINT32 TimeHigh;                            /*!< \brief Timestamp, upper 32-bits */
    ADT_L0_UINT32 TimeLow;                             /*!< \brief Timestamp, lower 32-bits */
    ADT_L0_UINT32 IMGap;                               /*!< \brief Intermessage gap, 100ns LSB */
    ADT_L0_UINT32 Rsvd3;                                /*!< \brief Reserved */
    ADT_L0_UINT32 CMD1Info;                            /*!< \brief Command 1 info */
    ADT_L0_UINT32 CMD2Info;                            /*!< \brief Command 2 info */
    ADT_L0_UINT32 STS1Info;                            /*!< \brief Status 1 info */
    ADT_L0_UINT32 STS2Info;                            /*!< \brief Status 2 info */
    ADT_L0_UINT32 DATAInfo[32];                         /*!< \brief Data word info */
} ADT_L1_1553_CDP;
```

The fields in this structure correspond directly to the words in the CDP structure used by the 1553 Protocol Engine (PE) on the board (see next figure). This is discussed in detail in the **AltaCore** 1553 Protocol Engine Specification/User's Manual.

The ADT_L1.h also provide bit field definitions for your application to set and mask settings as needed:

```
/* 1553 CDP Offsets (BYTE offsets) */
/* *** AltaCore-1553 Manual: Common Data Packet (CDP) *** */
#define ADT_L1_1553_CDP_NEXT          0x0000
#define ADT_L1_1553_CDP_BMCOUNT       0x0004
#define ADT_L1_1553_CDP_RESV_API     0x0008
#define ADT_L1_1553_CDP_RAPI_RT_ID   0x80000000
#define ADT_L1_1553_CDP_RAPI_MC_ID   0x40000000
#define ADT_L1_1553_CDP_RAPI_RT_RTADDR 20
#define ADT_L1_1553_CDP_RAPI_RT_TR   18
#define ADT_L1_1553_CDP_RAPI_RT_SA   12
#define ADT_L1_1553_CDP_RAPI_RT_BUFSIZE 0
#define ADT_L1_1553_CDP_RAPI_BC_ID   0x20000000
#define ADT_L1_1553_CDP_RAPI_BC_MSGNUM 12
#define ADT_L1_1553_CDP_RAPI_BC_BUFSIZE 0
#define ADT_L1_1553_CDP_RAPI_BM_ID   0x10000000
#define ADT_L1_1553_CDP_RAPI_BM_BUFSIZE 0
#define ADT_L1_1553_CDP_RSVD1        0x000C
#define ADT_L1_1553_CDP_RSVD2        0x0010
#define ADT_L1_1553_CDP_MASKVALUE    0x0014
#define ADT_L1_1553_CDP_MASKCOMPARE  0x0018
#define ADT_L1_1553_CDP_CONTROL      0x001C
#define ADT_L1_1553_CDP_CONTROL_TXERRINJ 0x00000002
#define ADT_L1_1553_CDP_CONTROL_INTERR 0x00000010
#define ADT_L1_1553_CDP_CONTROL_INTNOERR 0x00000020
```

```

#define      ADT_L1_1553_CDP_CONTROL_INTCMPTRUE      0x00000040
#define      ADT_L1_1553_CDP_CONTROL_LEDONCMPLT      0x000000400
#define      ADT_L1_1553_CDP_CONTROL_TRGOUTERR 0x00002000
#define      ADT_L1_1553_CDP_CONTROL_TRGOUTNOERR    0x00004000
#define      ADT_L1_1553_CDP_CONTROL_TRGOUTCMPTR    0x00008000
#define      ADT_L1_1553_CDP_CONTROL_CDPCMP       16
#define      ADT_L1_1553_CDP_CONTROL_FRCDWCONUM     24
#define      ADT_L1_1553_CDP_CONTROL_FRCDWCON 0x40000000
#define ADT_L1_1553_CDP_STATUS      0x0020
#define      ADT_L1_1553_CDP_STATUS_MSGWC          0x0000003F
#define      ADT_L1_1553_CDP_STATUS_BUSAB          0x00000040
#define      ADT_L1_1553_CDP_STATUS_TWOBUSERR    0x00000100
#define      ADT_L1_1553_CDP_STATUS_STSWRNGADD 0x00000200
#define      ADT_L1_1553_CDP_STATUS_NORESP        0x00000400
#define      ADT_L1_1553_CDP_STATUS_WCERR         0x00000800
#define      ADT_L1_1553_CDP_STATUS_PARERR        0x00001000
#define      ADT_L1_1553_CDP_STATUS_BITERR        0x00002000
#define      ADT_L1_1553_CDP_STATUS_SYNCERR       0x00004000
#define      ADT_L1_1553_CDP_STATUS_NOERR         0x00008000
#define      ADT_L1_1553_CDP_STATUS_CMPTURE       0x00010000
#define      ADT_L1_1553_CDP_STATUS_SPURMSG      0x04000000
#define      ADT_L1_1553_CDP_STATUS_BCRTMSG      0x08000000
#define      ADT_L1_1553_CDP_STATUS_RTBCMSG      0x10000000
#define      ADT_L1_1553_CDP_STATUS_RTRTMMSG     0x20000000
#define      ADT_L1_1553_CDP_STATUS_MCMMSG        0x40000000
#define      ADT_L1_1553_CDP_STATUS_BRDCSTMSG    0x80000000
#define ADT_L1_1553_CDP_TIMEHIGH      0x0024
#define ADT_L1_1553_CDP_TIMELOW 0x0028
#define ADT_L1_1553_CDP_IMGAP        0x002C
#define ADT_L1_1553_CDP_RSVD3       0x0030
#define ADT_L1_1553_CDP_CMD1INFO    0x0034
#define ADT_L1_1553_CDP_CMD2INFO    0x0038
#define ADT_L1_1553_CDP_STS1INFO    0x003C
#define ADT_L1_1553_CDP_STS2INFO    0x0040
#define ADT_L1_1553_CDP_DATA1INFO   0x0044
#define      ADT_L1_1553_CDP_WRDINFO_1553BITS    0x0000FFFF
#define      ADT_L1_1553_CDP_WRDINFO_GAPTIME     0x00FF0000
#define      ADT_L1_1553_CDP_WRDINFO_GAPTIMEOSET 16
#define      ADT_L1_1553_CDP_WRDINFO_RXADDTXGAP   0x020000000
#define      ADT_L1_1553_CDP_WRDINFO_RXNORESP     0x040000000
#define      ADT_L1_1553_CDP_WRDINFO_RXWCERR      0x080000000
#define      ADT_L1_1553_CDP_WRDINFO_RXTXPARERR  0x100000000
#define      ADT_L1_1553_CDP_WRDINFO_RXTXMANERR   0x200000000
#define      ADT_L1_1553_CDP_WRDINFO_RXTXSYNCERR  0x400000000
#define      ADT_L1_1553_CDP_WRDINFO_RXBUSAB     0x800000000

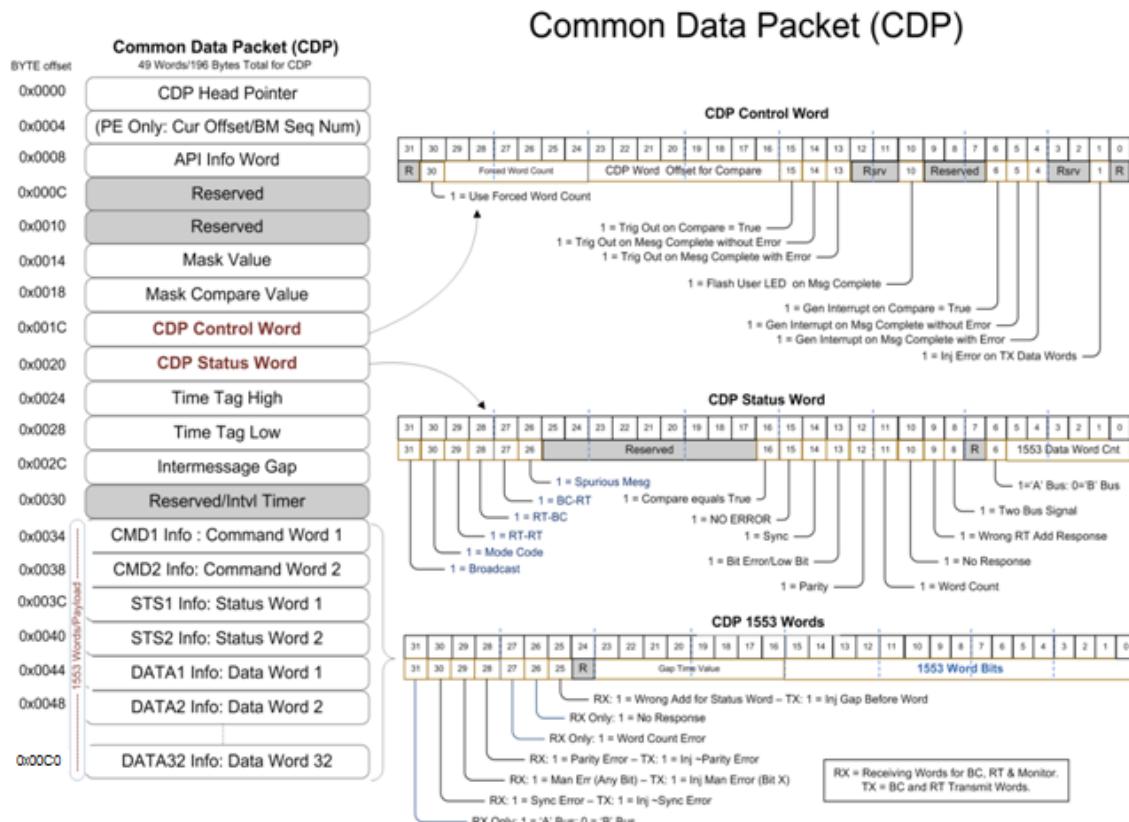
```

The L1 API provides read and write functions for CDPs for each 1553 operational mode (BC, RT, BM). Each BC, RT or BM function can usually define one to N buffers for each Subaddress/mode code or message as appropriate. Also, the API provides functions to access only sub-portions of the CDP for applications that only need to update small areas like a couple data words or a single control word (without having to write/read all 49 words of the structure).

Error Injection/Detection for Data Words of BC or RT Messages

The CDP provides low level error detection status and error injection options for any transmit data words. This can be very useful for development testing for design verification. The CDP Control Word provides on/off bits for error injection on transmit data words, and then each data word itself has the low-level controls for the type of error to be injected on the data word. Command Words and Status Word error injection is provided at the BC Control Block (BCCB) and RT Control Block data structure level, respectively.

Please review the following figure and #define definitions above to see error injection settings for transmit data words in CDPs. You should also review the AltaCore-1553 manual, CDP section for details on these advanced settings. Please see the BC or RT discussions for further discussion on Command or Status Word error injection.



Setting Message (CDP) Interrupts, Triggers, etc...

The CDP Control Word provides many different options for setting trigger, interrupts, Flashing the user LED, etc. A common application is setting the interrupt bit to signal the application that the CDP (and thus the desired BM, BC or RT message) has been received. The trigger output on error could be useful to set an external trigger to an oscilloscope to look for bus error conditions.

1553 Bus Monitor Operation

The 1553 Bus Monitor (BM) functions are defined in the file ADT_L1_1553_BM.c. The BM API functions allocate and manage a circular list of CDP structures (making a sequential monitor). Each CDP stores one 1553 message. The following figure shows basic BM programming.

AltaAPI Operational Flow – 1553 Bus Monitor

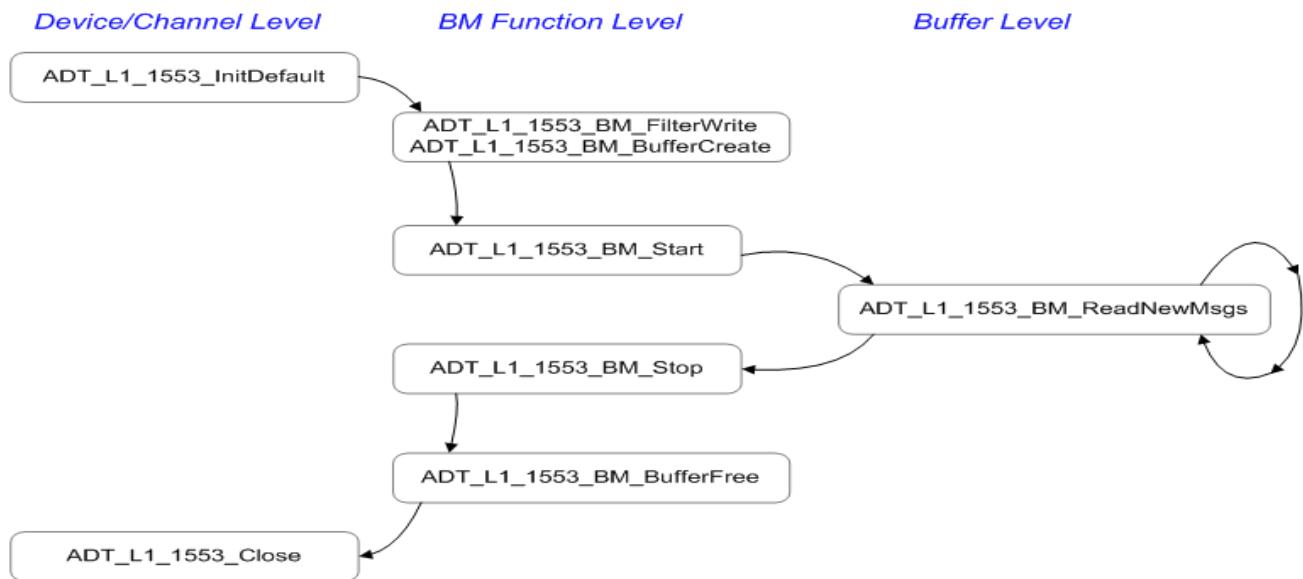


Figure BM-1: Basic Programming Flow

The following sample code shows the basic BM function calls (there are more complete BM example programs provided in the software distribution – including one simple example that archives all messages to a file):

```
#include <stdio.h>
#include "ADT_L1.h"

#define DEVID (ADT_PRODUCT_PMC1553 | ADT_DEVID_BOARDNUM_01 |
           ADT_DEVID_CHANNELTYPE_1553 | ADT_DEVID_CHANNELNUM_01)

int main() {
    ADT_L0_UINT32 status, rtAddr, numMsgs, i;
    ADT_L1_1553_CDP bmMessages[10];

    status = ADT_L1_1553_InitDefault(DEVID, 10);
    // Set the BM Subaddress Filters for Each RT
    for (rtAddr=0; rtAddr<32; rtAddr++)
    {
        status = ADT_L1_1553_BM_FilterWrite(DEVID, rtAddr,
                                           0xFFFFFFFF, 0xFFFFFFFF);
    }
    status = ADT_L1_1553_BM_BufferCreate(DEVID, 20);
    status = ADT_L1_1553_BM_Start(DEVID);
    // Repeat this next function to keep reading BM CDPs
    status = ADT_L1_1553_BM_ReadNewMsgs(DEVID, 10, &numMsgs,
                                         bmMessages);
    status = ADT_L1_1553_BM_Stop(DEVID);
    status = ADT_L1_CloseDevice(DEVID);
}
```

The following paragraphs will detail each step above.

BM Filters

Prior to starting BM storage, the BM provides filters to specify which messages to store by RT address, transmit/receive, and sub-address. The filters are configured using the ADT_L1_1553_BM_FilterWrite function.

```
printf("Writing BM Filters (capture all) . . . ");
for (rtAddr=0; rtAddr<32; rtAddr++) {
    status = ADT_L1_1553_BM_FilterWrite(DEVID, rtAddr, 0xFFFFFFFF, 0xFFFFFFFF);
}
```

The example above configures the BM filters to capture all possible messages. The RT address is the second parameter. The third parameter specifies the RECEIVE sub-addresses to capture. Each bit in the 32-bit word corresponds to a receive sub-address – for example if bit 5 is set then messages for receive sub-address 5 will be captured, if bit 5 is clear then messages for receive sub-

address 5 will not be captured. The fourth parameter specifies the TRANSMIT sub-addresses to capture and the bits are used the same way.

BM Buffer (CDP) Allocation

The function ADT_L1_1553_BM_BufferCreate allocates CDP buffers that the BM uses to store messages. The function ADT_L1_1553_BM_BufferFree frees the BM buffer memory and un-initializes the BM.

```
status = ADT_L1_1553_BM_BufferCreate(DEVID, 20);
```

The example above allocates 20 CDP buffers for the BM in a circular list. This allows the BM to store the 20 most recent messages. After 20 messages are received the BM will overwrite old messages as more messages are captured.

The number of BM buffers to allocate is determined by how much message traffic is expected and by how often the application software can read messages out of the buffer. For example, the worst-case (fastest) 1553 message traffic would be a series of broadcast mode codes without data (one word per message) with a minimum inter-message gap (4 microsecond gap) – this would give us a message every 24 microseconds. For easy math we will round this up to one message every 25 microseconds, or 40,000 messages per second. Let's assume that our software can check the device for new messages every 10 milliseconds (100 times per second). We could get up to 400 new messages in 10 milliseconds if we are getting 40,000 messages per second. Now we want to add a safety margin just in case we don't always get around to reading the buffer every 10 milliseconds, so we could round this up to 500 message buffers needed on the board. If we REALLY want to make sure we won't miss any messages we could double this and allocate 1000 message buffers for the BM.

This example demonstrates that 1000 (or even 500) CDP buffers is all you need to allocate on the device for any possible 1553 traffic as long as you can check for new messages every 10 milliseconds or so. The **AltaView** bus analyzer software allocates 1000 CDP buffers for the bus monitor and checks for new messages every 10 milliseconds.

The CDP structure provides a field called "BMCount". This gives us a "sequence number" for each message received by the BM. This number should increment by one on each new message in the BM buffer. Therefore we can check this field as we read messages from the BM to tell us if the software has fallen behind and missed any messages.

BM Control Functions

Bus monitoring can be started and stopped with the functions ADT_L1_1553_BM_Start and ADT_L1_1553_BM_Stop.

BM Message Read Functions

Messages can be read from the BM buffer with the function:

ADT_L1_1553_BM_ReadNewMsgs()

This is a VERY powerful, yet, simple function. This function will read up to N CDPs (in the example code above, 10 messages where the maximum), and then tells the application how many CDPs where actually read (numMsgs) and then populates the CDP array (bmMessages[]) with the read CDPs.

For example, from the “bm2.c” example program we have a simple loop to read CDPs and save to file:

```
totalMsgCount = 0;
while (totalMsgCount < maxMsgs)
{
    /* This will read up to 1000 messages from the board (which is the
maximum our bmMessages array can hold in this example).
*/
    status = ADT_L1_1553_BM_ReadNewMsgs(DEVID, 1000, &numMsgs,
                                         bmMessages);
    if (status == ADT_SUCCESS) {
        totalMsgCount += numMsgs;

        /* Write the messages to the CDP file */
        for (i=0; i<numMsgs; i++)
        {
            fwrite(&bmMessages[i], sizeof(ADT_L1_1553_CDP), 1,
                  fp);
        }
    }
}
```

That's it! This simple code snippet would write all 1553 messages (CDPs) to a file.

1553 Remote Terminal Operation

The 1553 Remote Terminal functions are defined in the file ADT_L1_1553_RT.c. There are numerous RT example programs provided that show a wide range of simple to more complex RT operations. Use one of these programs to jump-start your application.

On a 1553 network, there are 32 possible RT addresses (0-31). Each RT can have up to 32 receive sub-addresses (0-31) and up to 32 transmit sub-addresses (0-31), and a number of pre-defined message modes called “mode codes”. Each sub-address or mode code can have one or more CDP buffers in a circular link-list where each CDP is for one 1553 message. The following figure shows basic RT program flow.

AltaAPI Operational Flow – 1553 Remote Terminal

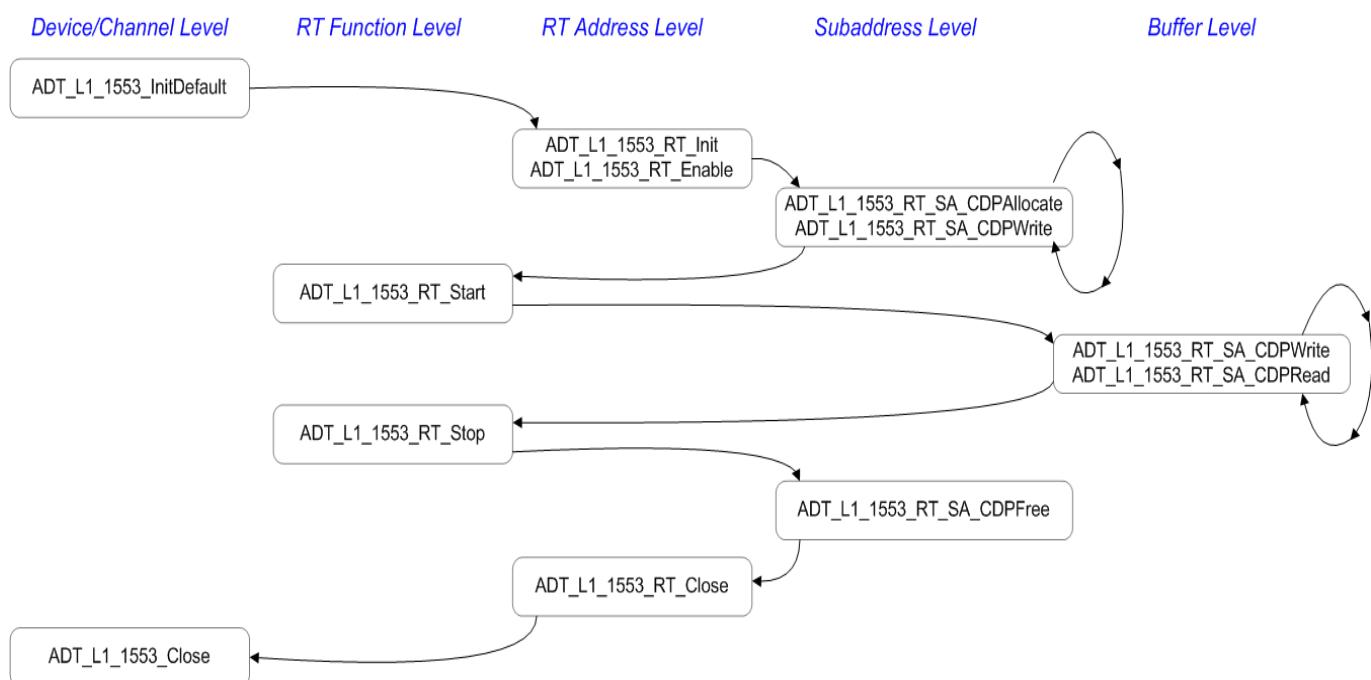


Figure RT-1: Basic RT Programming Flow

Here is a simple sample code snippet to setup an RT:

```
#include <stdio.h>
#include "ADT_L1.h"

/* DevID = 1553 Device Channel Code from ADT_L0.h */
#define DEVID      (ADT_PRODUCT_PMC1553 |
                  ADT_DEVID_BOARDNUM_01 |
                  ADT_DEVID_CHANNELTYPE_1553 |
                  ADT_DEVID_CHANNELNUM_01)

int main()
{
    ADT_L0_UINT32      status, RTnum=1, tr=0, saAdd=1,
                      modeCode=2, CDPnum=0;
    ADT_L1_1553_CDP  myCdp;

    status = ADT_L1_1553_InitDefault(DEVID, 10);
    status = ADT_L1_1553_RT_Init(DEVID, RTnum);
    status = ADT_L1_1553_RT_Enable(DEVID, RTnum);
    status = ADT_L1_1553_RT_Start(DEVID);
    /* Your application here */
    status = ADT_L1_1553_RT_SA_CDPRead(DEVID, RTnum, tr,
                                         subAdd, CDPnum, &myCdp);
    status = ADT_L1_1553_RT_SA_CDPWrite (DEVID, RTnum, tr,
                                         subAdd, CDPnum, &myCdp);
    status = ADT_L1_1553_RT_MC_CDPRead (DEVID, RTnum, tr,
                                         modeCode, CDPnum, &myCdp);
    status = ADT_L1_1553_RT_MC_CDPWrite (DEVID, RTnum, tr,
                                         modeCode, CDPnum, &myCdp);

    status = ADT_L1_1553_RT_Stop(DEVID);
    status = ADT_L1_1553_RT_Close(DEVID, RTnum);
    status = ADT_L1_CloseDevice(DEVID);
}
```

This program would not really do anything except initialize the device, initialize the RT, enable and start the RT and then close it – this is not a practical application, but these are the basic steps for many RT applications. Let's now review these steps in detail.

RT Initialization

Each RT address to be used must be initialized using the ADT_L1_1553_RT_Init function.

```
printf("Initializing RT 1 . . . ");
status = ADT_L1_1553_RT_Init(DEVID, 1);
if (status == ADT_SUCCESS) printf("Success.\n");
else printf("FAILURE - Error = %d\n", status);
```

This function initializes the data structures for the RT address and allocates a default CDP data buffer that is used for all sub-addresses. Therefore, if we enabled the RT (ADT_L1_1553_RT_Enable) and started RT operation (ADT_L1_1553_RT_Start), the RT would respond to commands and all sub-addresses would be “wrapped” (if the BC sends a BC-RT message with a given set of data, then sends a RT-BC command, it will get the same set of data back in the RT-BC message).

Single RT and Multiple RT Configurations

(Most applications skip this paragraph – this step is only needed for Single RT and External RT Address Configurations).

Alta 1553 products that support RT functionality are available in single or multiple RT configurations. Multiple RT products can operate in either single or multiple RT mode. Single or multiple RT mode is selected by the function ADT_L1_1553_SetConfig.

```
/* Set single RT, 1553B, RT31 is BROADCAST, external bus, 14us timeout */
printf("Setting options . . . ");
status = ADT_L1_1553_SetConfig(DEVID, 0, 1, 1, 0, 14);
if (status == ADT_SUCCESS) printf("Success.\n");
else printf("FAILURE - Error = %d\n", status);
```

The second parameter to ADT_L1_1553_SetConfig selects the mode – if this parameter is zero then single RT mode is selected, otherwise multiple RT mode is selected.

If single RT mode is selected the application must assign an RT address. This can be done by hardware inputs (external RT address inputs to the board) or by software. The hardware external RT address signals can be read using the ADT_L1_1553_RT_GetExternalRTAddr function. The single RT address can be set by software with the ADT_L1_1553_RT_SetSingleRTAddr function.

```
/* Our RT will be RT Address 1 */
printf("Setting Single RT Address . . . ");
status = ADT_L1_1553_RT_SetSingleRTAddr(DEVID, 1);
if (status == ADT_SUCCESS) printf("Success.\n");
else printf("FAILURE - Error = %d\n", status);
```

BROADCAST Messages – Single RT and Multiple RT

Broadcast (RT31) messages are handled differently for Single-RT mode and Multiple-RT (default) mode. In Single-RT mode the Broadcast messages will go to the same CDP buffer that would be used for non-broadcast messages. In Multiple-RT mode you must setup RT31 with the desired subaddress buffers to receive Broadcast messages when operating as an RT.

The example program ADT_L1_1553_ex_rt2_bcast_srt.c demonstrates this for Single-RT mode.

The example program ADT_L1_1553_ex_rt2_bcast_mrt.c demonstrates this for Multiple-RT mode.

Allocating and Managing RT/SA and Mode Code Buffers (CDPs)

If we want to do anything meaningful with the sub-address data then we need to allocate dedicated (non-default) buffers for the sub-addresses of interest. This is done using the ADT_L1_1553_RT_SA_CDPAllocate function.

NOTE: For efficiency if multiple CDPs are allocated for a given RT/SA Buffer they are assumed to be ordered sequentially in the software. If the user wishes to reorder the CDP buffers by changing the NextPtr values of the ADT_L1_1553_CDP structure, the ADT_1553_USE_NONCONTIGUOUS_CDP #define must be uncommented in the ADT_L1.h file located just after the ADT_L1_API_VERSION #define. Otherwise the various functions which read and write to the CDPs will merely use the CDP number as an offset in the CDP list instead of following the pointers.

Data is read and written to SA buffers using the ADT_L1_1553_RT_SA_CDPRead and ADT_L1_1553_RT_SA_CDPWrite functions. These functions read the entire message Common Data Packet (CDP). There are also two functions to CDPReadWords/CDPWriteWords to read/write individual word(s), but the user must be careful to know and access proper word offsets of the CDP (these functions can save a lot of PCI access time for reading/writing a small number of CDP words). Most applications should just read the whole CDP and not worry about word offset locations (and you would avoid possibly stepping on the CDP Head Pointer).

NOTE: If you read a CDP buffer while the firmware is in the middle of processing a message for that buffer, then the CDP Status Word will be 0xFFFFFFFF. If you see this value, then you should read the buffer again until the CDP Status Word is NOT 0xFFFFFFFF – you will then have a

complete CDP buffer. If you use interrupts to synchronize buffer reads to messages on the bus then you should not see this case.

Most applications only use a single CDP buffer for a given SA. It is best to read and write buffers synchronously with messages on the bus – the usual approach is to enable an interrupt on each CDP, when the application gets the interrupt it reads or writes the appropriate CDP buffer (see example program ADT_L1_1553_ex_rt3int.c).

The Alta architecture allows multiple buffers per SA – these are configured as a circular linked-list where the firmware will fill/send the current buffer then automatically move to the next buffer. If an interrupt is used on each CDP then the information passed to the interrupt handler will identify the CDP that generated the interrupt and the handler can then read/write the appropriate CDP. If your application does NOT use interrupts, then your code will have to determine which buffer to read or write – for example, you could check the time-stamps in each buffer to find the CDP that was filled/sent most recently, then handle appropriately.

SA buffers can be freed using the ADT_L1_1553_RT_SA_CDPFree function. This will reset the SA to use the default buffer.

Mode codes allocate buffers by the mode code number, so there is a different set of functions for mode code buffers. Dedicated mode code buffers are allocated using the ADT_L1_1553_RT_MC_CDPAllocate function.

Data is read and written to mode code buffers using the ADT_L1_1553_RT_MC_CDPRead and ADT_L1_1553_RT_MC_CDPWrite functions.

Mode code buffers can be freed using the ADT_L1_1553_RT_MC_CDPFree function. This will reset the mode code to use the default buffer.

RT Command Legalization

Legal and illegal commands can be specified for sub-addresses and mode codes, as per the 1553 protocol illegal command option.

Sub-address legalization settings can be read with the ADT_L1_1553_RT_SA_LegalizationRead function and written with the ADT_L1_1553_RT_SA_LegalizationWrite function.

Mode code legalization settings can be read with the ADT_L1_1553_RT_MC_LegalizationRead function and written with the ADT_L1_1553_RT_MC_LegalizationWrite function.

RT Status and Last Command Words

The RT status word can be read with the ADT_L1_1553_RT_StatusRead function and written with the ADT_L1_1553_RT_StatusWrite function.

The last command word received by the RT can be read with the ADT_L1_1553_RT_GetLastCmd function.

RT Status Response Time

The RT status response time can be read with the ADT_L1_1553_RT_GetRespTime function and written with the ADT_L1_1553_RT_SetRespTime function.

Error Injection on the RT Status Word

Errors can be injected on the RT status word using the ADT_L1_1553_RT_InjStsWordError function. The error settings for the status word can be read using the ADT_L1_1553_RT_ReadStsWordError function.

Note that error injection on data words is defined at the CDP level and works the same way for RT or BC data.

RT Control Functions

Remote terminal operation can be started and stopped with the functions ADT_L1_1553_RT_Start and ADT_L1_1553_RT_Stop.

Specific remote terminals can be enabled with the ADT_L1_1553_RT_Enable function or disabled with the ADT_L1_1553_RT_Disable function.

RT operation in response to specific mode codes (Dynamic Bus Control and Transmit Vector Word) and enable/disable of transmission on specific busses (A or B) can be done with the ADT_L1_1553_RT_SetOptions function.

1553 Bus Controller (BC) Operation

The 1553 Bus Controller functions are defined in the file ADT_L1_1553_BC.c. There are 15+ BC example programs in the distribution and you should use one of these to jump-start your application.

The Bus Controller (BC) defines 1553 messages using BC Control Blocks (BCCB). The BCCB provides controls and transmission timing for BC messages. Each BCCB will have one or more Common Data Packets (CDP) to store transmit or received data for the message.

Most applications use one CDP data buffer for each BCCB message. Multiple CDP data buffers can be used to pre-load a set of data buffers to be transmitted or to store the last N sets of data received for a given message.

The following paragraphs and figure will detail BC setup and usage.

AltaAPI Operational Flow – 1553 Bus Controller

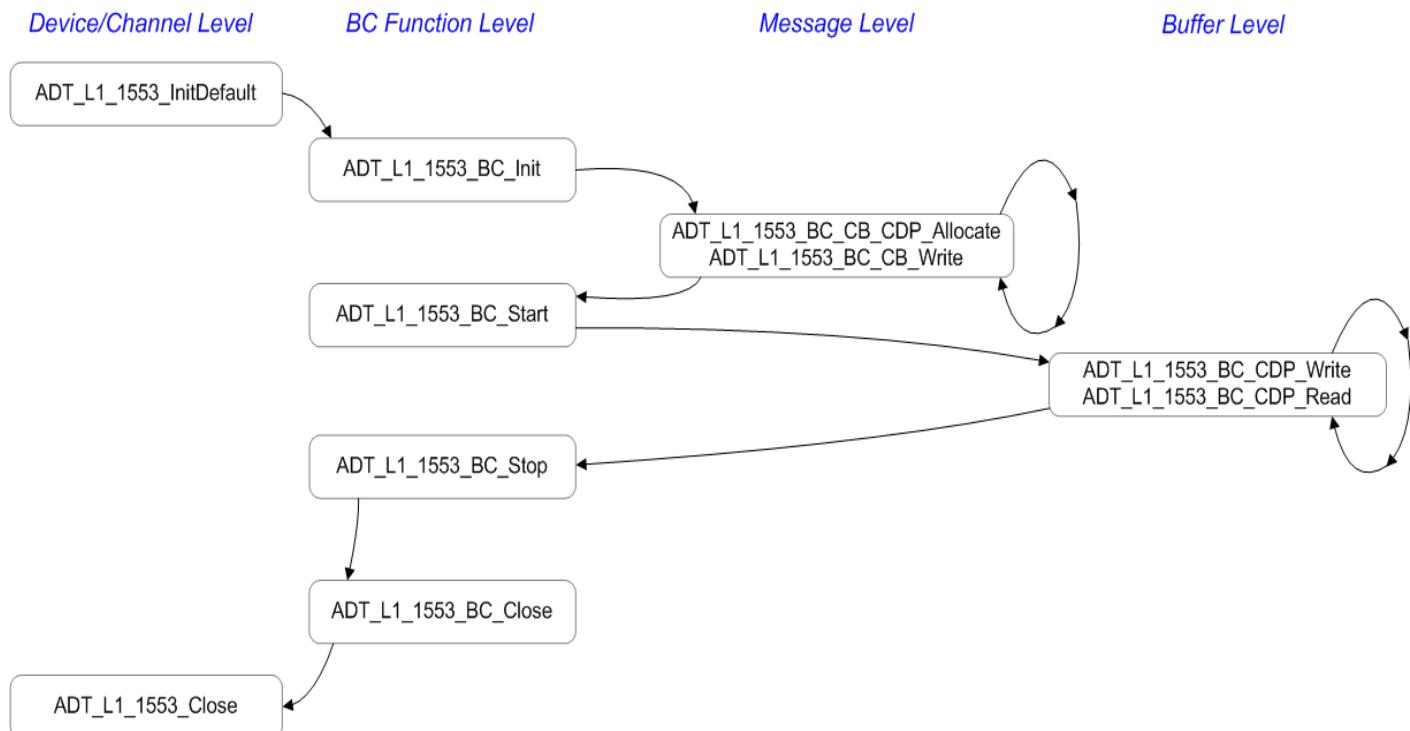


Figure BC-1: Basic BC Programming Flow

The following sample program provides a rudimentary setup and execution of a BC message (derived from example “bc1.c”).

```
#include <stdio.h>
#include <memory.h>
#include "ADT_L1.h"

#define DEVID (ADT_PRODUCT_PMC1553 | ADT_DEVID_BOARDNUM_01 |
           ADT_DEVID_CHANNELTYPE_1553 |ADT_DEVID_CHANNELNUM_01)

int main()
{
    ADT_L0_UINT32      status, i;
    ADT_L1_1553_BC_CB myBCCB;
    ADT_L1_1553_CDP   myCdp;

    status = ADT_L1_1553_InitDefault(DEVID, 10);
    status = ADT_L1_1553_BC_Init(DEVID, 10, 1, 0);
    status = ADT_L1_1553_BC_CB_CDPAllocate(DEVID, 0, 1);

    /* Define the BCCB for message 0 */
    memset(&myBCCB, 0, sizeof(myBCCB));
    myBCCB.CMD1Info = 0x0820;
    myBCCB.Csr = ADT_L1_1553_BC_CB_CSR_TYPE_BCRT |
                  ADT_L1_1553_BC_CB_CSR_BUSA;
    myBCCB.DelayTime = 1000;
    myBCCB.NextMsgNum = ADT_L1_1553_BC_NO_NEXT_MSG;
    status = ADT_L1_1553_BC_CB_Write(DEVID, 0, &myBCCB);

    /* Write the data buffer (CDP) for message 0 */
    memset(&myCdp, 0, sizeof(myCdp));
    for (i=0; i<32; i++)
        myCdp.DATAinfo[i] = 0x00001100 + i;
    status = ADT_L1_1553_BC_CB_CDPWrite(DEVID, 0, 0, &myCdp);

    status = ADT_L1_1553_BC_Start(DEVID, 0);
    status = ADT_L1_msSleep(1);
    status = ADT_L1_1553_BC_Stop(DEVID);
    status = ADT_L1_CloseDevice(DEVID);
}
```

The API represents BCCBs with the structure ADT_L1_1553_BC_CB. This structure is defined in the ADT_L1.h header file.

```

typedef struct adt_l1_1553_bc_cb {
    ADT_L0_UINT32 NextMsgNum;           /*!< \brief Next message number */
    ADT_L0_UINT32 Retry;               /*!< \brief BC Retry word */
    ADT_L0_UINT32 Csr;                /*!< \brief BC CB CSR */
    ADT_L0_UINT32 CMD1Info;            /*!< \brief Command word 1 info */
    ADT_L0_UINT32 CMD2Info;            /*!< \brief Command word 2 info */
    ADT_L0_UINT32 FrameTime;           /*!< \brief Frame time, 100ns LSB, applies if SOF */
    ADT_L0_UINT32 DelayTime;           /*!< \brief Delay time, 100ns LSB, IM gap */
    ADT_L0_UINT32 BranchMsgNum;        /*!< \brief Branch message number */
    ADT_L0_UINT32 StartFrame;          /*!< \brief Start frame number */
    ADT_L0_UINT32 StopFrame;           /*!< \brief Stop frame number */
    ADT_L0_UINT32 FrameRepRate;        /*!< \brief Frame repetition rate */
    ADT_L0_UINT32 MsgNum;              /*!< \brief Message number for this BCCB */
    ADT_L0_UINT32 NumBuffers;          /*!< \brief Number of CDPs allocated to this BCCB */
} ADT_L1_1553_BC_CB;

/* 1553 BC Control Block Offsets (BYTE offsets) */
/* *** AltaCore-1553 Manual: Bus Controller (BC) *** */
#define ADT_L1_1553_BC_CB_NEXTPTR      0x0000
#define ADT_L1_1553_BC_CB_CDPPTR       0x0004
#define ADT_L1_1553_BC_CB_RETRY        0x0008
#define ADT_L1_1553_BC_CB_RETRY_ENABLEONERR 0x00000001
#define ADT_L1_1553_BC_CB_RETRY_ENABLEONBSY 0x00000002
#define ADT_L1_1553_BC_CB_RETRY_NUMATTEMPTD 0x000000F0
#define ADT_L1_1553_BC_CB_RETRY_CURPENUM 0x00000F00
#define ADT_L1_1553_BC_CB_RETRY_MAXNUMSET 0x0000F000
#define ADT_L1_1553_BC_CB_RETRY_BUSPATTERN 0xFFFF0000
#define ADT_L1_1553_BC_CB_CSR          0x000C
#define ADT_L1_1553_BC_CB_CSR_HALTONERROR 0x00000001
#define ADT_L1_1553_BC_CB_CSR_BUSA     0x00000002
#define ADT_L1_1553_BC_CB_CSR_BUSB     0x00000000
#define ADT_L1_1553_BC_CB_CSR_STARTFRAME 0x00000004
#define ADT_L1_1553_BC_CB_CSR_ENDFRAME 0x00000008
#define ADT_L1_1553_BC_CB_CSR_SCHEDTIMING 0x00000010
#define ADT_L1_1553_BC_CB_CSR_WAITFORTRG 0x00000020
#define ADT_L1_1553_BC_CB_CSR_GENEXTTRG 0x00000040
#define ADT_L1_1553_BC_CB_CSR_INTMSGCOMP 0x00000100
#define ADT_L1_1553_BC_CB_CSR_ADDRBRANCH 0x00100000
#define ADT_L1_1553_BC_CB_CSR_CDPBRANCHONLY 0x00200000
#define ADT_L1_1553_BC_CB_CSR_DELAYONLY 0x00400000
#define ADT_L1_1553_BC_CB_CSR_BRNCHONVALUE 0x00800000
#define ADT_L1_1553_BC_CB_CSR_BRNCHRETURN 0x01000000
#define ADT_L1_1553_BC_CB_CSR_TYPE_NOP 0x02000000
#define ADT_L1_1553_BC_CB_CSR_TYPE_BCRT 0x04000000
#define ADT_L1_1553_BC_CB_CSR_TYPE_RTBC 0x08000000
#define ADT_L1_1553_BC_CB_CSR_TYPE_RTRT 0x10000000
#define ADT_L1_1553_BC_CB_CSR_TYPE_MCDATA 0x20000000
#define ADT_L1_1553_BC_CB_CSR_TYPE_MCNODATA 0x40000000
#define ADT_L1_1553_BC_CB_CMD1INFO      0x0010
#define ADT_L1_1553_BC_CB_CMD2INFO      0x0014
#define ADT_L1_1553_CMD_WRDINFO_1553BITS 0x0000FFFF
#define ADT_L1_1553_CMD_WRDINFO_GAPTIME 0x00FF0000
#define ADT_L1_1553_CMD_WRDINFO_TXGAP   0x02000000
#define ADT_L1_1553_CMD_WRDINFO_TXPARERR 0x10000000
#define ADT_L1_1553_CMD_WRDINFO_TXMANERR 0x20000000
#define ADT_L1_1553_CMD_WRDINFO_TXSYNCERR 0x40000000
#define ADT_L1_1553_BC_CB_FRAME TIME 0x0018
#define ADT_L1_1553_BC_CB_DELAYTIME    0x001C
#define ADT_L1_1553_BC_CB_BRANCHADD    0x0020
#define ADT_L1_1553_BC_CB_STARTFRM     0x0024
#define ADT_L1_1553_BC_CB_STOPFRM      0x0028

```

```

#define ADT_L1_1553_BC_CB_REPRT 0x002C
#define ADT_L1_1553_BC_CB_NXTXMIT 0x0030
#define ADT_L1_1553_BC_CB_APIMSGNUM 0x0034
#define ADT_L1_1553_BC_CB_APINUMCDP 0x0038
#define ADT_L1_1553_BC_CB_API1STCDP 0x003C

```

1553 BC Control Block (BCCB)

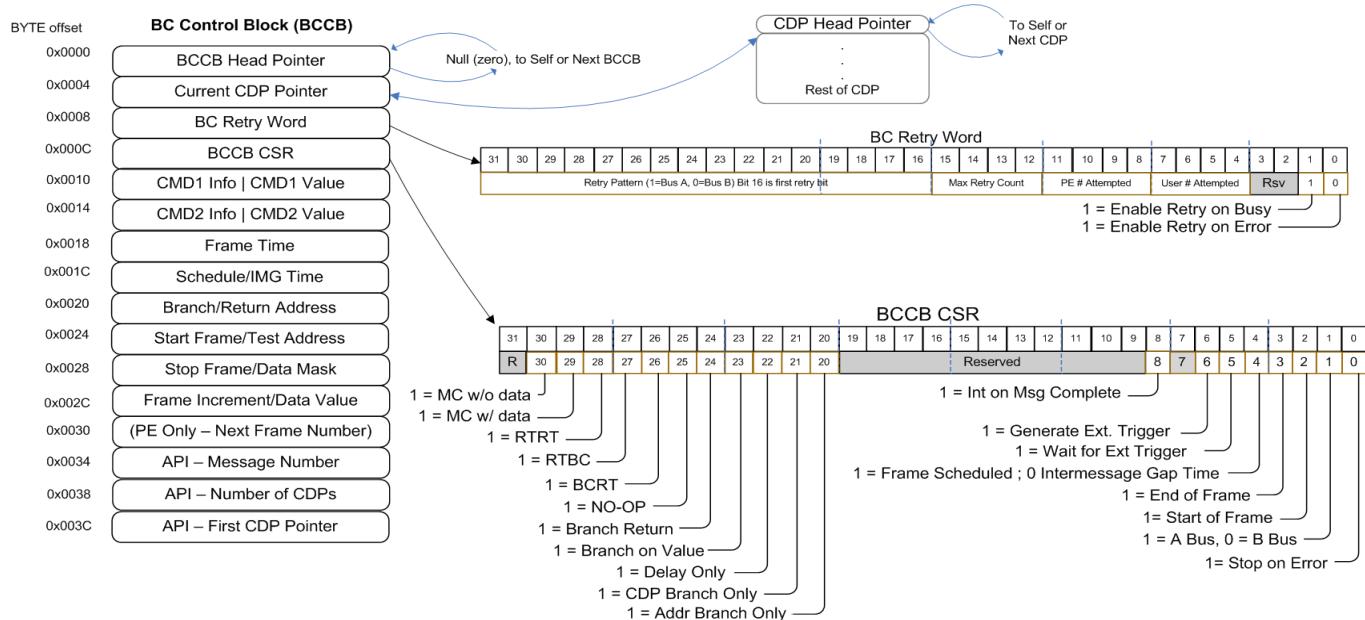


Figure BC-2: BCCB Data Structure

Note that the fields in the `adt_11_1553_bc_cb` structure correspond directly to words in the BCCB structure used by the 1553 Protocol Engine (PE) on the board (shown above), but the PE BCCB structure contains additional words which are used exclusively by the PE to process the BCCB. This is discussed in detail in the **AltaCore 1553 Protocol Engine Specification/User's Manual**.

The API identifies messages by “message numbers” where the BCCBs used by the PE on the board use memory pointers. The API manages the memory pointers by message number so the user application does not need to deal with PE memory pointers.

BC Initialization

The BC is initialized with the function ADT_L1_1553_BC_Init. The BC is closed with the function ADT_L1_1553_BC_Close.

```
/* BC Initialization – max 100 messages, 1 minor per major, BC CSR 0 */
printf("Initializing BC . . . ");
status = ADT_L1_1553_BC_Init(DEVID, 100, 1, 0);
if (status == ADT_SUCCESS) printf("Success.\n");
else printf("FAILURE - Error = %d\n", status);
```

The second parameter to the ADT_L1_1553_BC_Init function is the maximum number of BC messages that will be used. The API allocates a table in the device memory that contains one 32-bit word for each possible message number (in this example we have a maximum of 100 messages so we can have message numbers 0 through 99). When a BCCB is allocated for a given message number the API will store the memory pointer to the BCCB in the table entry word for that message number. This is how the API relates message numbers to PE BCCB pointers.

The third parameter is the number of minor frames per major frame. This will be discussed in the section for BC Frame Operation.

The fourth parameter is the value to write to the PE Root BC CSR. The ADT_L1.h header file defines constants for each of the bits/options in the CSR. These are listed below:

ADT_L1_1553_BC_CSR_STOPONFLOW	Stop BC if there is a frame overflow.
ADT_L1_1553_BC_CSR_EN_SUBFRAMES	Enable subframes (see BC Frame Opn).
ADT_L1_1553_BC_CSR_RTRYBUSY	Enable retry on busy (see BC Retry Opn).
ADT_L1_1553_BC_CSR_RTRYERROR	Enable retry on error (see BC Retry Opn).
ADT_L1_1553_BC_CSR_INTONFRMOWFLOW	Interrupt on frame overflow.
ADT_L1_1553_BC_CSR_INTONSTOP	Interrupt on BC stop.

Refer to the **AltaCore** 1553 Protocol Engine Specification/User's Manual for details on the PE Root BC CSR.

Allocating BCCBs and CDP Buffers

You **must** allocate BCCB and CDP buffers before you can read, write, or otherwise do anything with the BC messages and buffers. This should be done for all message numbers you plan to use because this allocation step writes the BCCB memory pointers to the API BCCB table, thus allowing the API to relate the message number to a BCCB in memory.

You cannot use a message number (for a NEXT or BRANCH message number) until a BCCB has been allocated for it.

BCCBs and CDP buffers are allocated for a message number using the ADT_L1_1553_BC_CB_CDPAllocate function. This memory can be freed using the ADT_L1_1553_BC_CB_CDPFree function.

```
***** Allocate BCCB and CDP for each message we plan to use *****
for (i=0; i<10; i++) {
    /* Allocate BCCB and one CDPs for each message */
    printf("Allocating BCCB and one CDP for msg %d . . . ", i);
    status = ADT_L1_1553_BC_CB_CDPAllocate(DEVID, i, 1);
    if (status == ADT_SUCCESS) printf("Success.\n");
    else printf("FAILURE - Error = %d\n", status);
}
```

The above example allocates a BCCB and one CDP buffer for 10 messages (message numbers 0 through 9).

It is imperative that memory be allocated on the Alta device for BCCBs and CDP buffers before attempting to define the messages or reference them from other BCCBs.

NOTE: For efficiency if multiple CDPs are allocated for a given BCCB Buffer they are assumed to be ordered sequentially in the software. If the user wishes to reorder the CDP buffers by changing the NextPtr values of the ADT_L1_1553_CDP structure, the ADT_1553_USE_NONCONTIGUOUS_CDP #define must be uncommented in the ADT_L1.h file located just after the ADT_L1_API_VERSION #define. Otherwise the various functions which read and write to the CDPs will merely use the CDP number as an offset in the CDP list instead of following the pointers.

Reading and Writing BC Data (and reviewing message results)

There are two main functions to read/write CDP buffer data and control/status information: ADT_L1_1553_BC_CB_CDPRead and ADT_L1_1553_BC_CB_CDPWrite. These functions read the entire BCCB message content of the Common Data Packet (CDP). There are also two functions to ADT_L1_1553_BC_CB_CDPReadWords() and ADT_L1_1553_BC_CB_CDPWriteWords() to read/write individual word(s), but the user must be careful to know and access proper word offsets of the CDP.

Defining BC Messages (1553 Message Types)

1553 BC Messages are defined by the Bus Controller Control Block (BCCB) and CDP as discussed in previous paragraphs. Here is the BCCB data structure again:

```
memset(&myBCCB, 0, sizeof(myBCCB));
typedef struct adt_L1_1553_bc_cb {
    ADT_L0_UINT32 NextMsgNum;           /*!< \brief Next message number */
    ADT_L0_UINT32 Retry;               /*!< \brief BC Retry word */
    ADT_L0_UINT32 Csr;                /*!< \brief BC CB CSR */
    ADT_L0_UINT32 CMD1Info;            /*!< \brief Command word 1 info */
    ADT_L0_UINT32 CMD2Info;            /*!< \brief Command word 2 info */
    ADT_L0_UINT32 FrameTime;           /*!< \brief Frame time, 100ns LSB, applies if SOF */
    ADT_L0_UINT32 DelayTime;           /*!< \brief Delay time, 100ns LSB, IM gap */
    ADT_L0_UINT32 BranchMsgNum;        /*!< \brief Branch message number */
    ADT_L0_UINT32 StartFrame;          /*!< \brief Start frame number */
    ADT_L0_UINT32 StopFrame;           /*!< \brief Stop frame number */
    ADT_L0_UINT32 FrameRepRate;         /*!< \brief Frame repetition rate */
    ADT_L0_UINT32 MsgNum;              /*!< \brief Message number for this BCCB */
    ADT_L0_UINT32 NumBuffers;          /*!< \brief Number of CDPs allocated to this BCCB */
} ADT_L1_1553_BC_CB;
```

The user will set values in these BCCB words to format the BC message transmission. The Command Words for the message are set in the **CMD1 & CMD2** (CMD2 for RT-RT second command word). The **BCCB Control & Status Register (CSR)** is a key word that must be programmed with the message type bit flag and other message options.

The other words in the BCCB do not need to be set for basic transfers, but must be defined if more advanced BC options such as Framing, SubFraming, Branching and Retries are selected.

The **memset()** C function is shown above as a reminder that it is a good idea to zero out the BCCB before definition as most options must be defined with non-zero (1's) and initializing the field to zero will generally provide a good default value.

The **MsgNum** field defines the message number (0-N) for this message. The **NextMsgNum** field specifies the next message number in the list of BC messages. These fields are used to link messages into lists of messages for transmission.

The following paragraphs show examples to set BCCBs for different basic 1553 message types. The key difference between message types is setting the correct BCCB CSR bit flag and setting the Command Word value(s). The Alta design tries to abstract the various 1553 message structures so you only have to

define simple bit and word definitions and not worry about 1553 message formats. The example program “bc3.c” show basic setup for all 1553 message types:

BC-RT

Below is an example that defines a BCRT message (1-R-1-32) and writes data words to the CDP buffer:

```
***** MESSAGE 0 - BCRT Message *****
memset(&myBCCB, 0, sizeof(myBCCB));

myBCCB.CMD1Info = 0x0820;          /* BCRT 1-R-1-32 on Bus A */
myBCCB.Csr = ADT_L1_1553_BC_CB_CSR_TYPE_BCRT | ADT_L1_1553_BC_CB_CSR_BUSA;
myBCCB.DelayTime = 1000; /* 100.0 us inter-message gap (dead-bus time) */
myBCCB.NextMsgNum = 1;           /* Go to message 1 */

printf("Writing BCCB 0 . . . ");
status = ADT_L1_1553_BC_CB_Write(DEVID, 0, &myBCCB);
if (status == ADT_SUCCESS) printf("Success.\n");
else printf("FAILURE - Error = %d\n", status);

/* Write the data buffer (CDP) for message 0 */
memset(&myCdp, 0, sizeof(myCdp));
printf("Writing msg 0 buffer 0 . . . ");
for (i=0; i<32; i++)
    myCdp.DATAinfo[i] = 0x00001100 + i;
status = ADT_L1_1553_BC_CB_CDPWrite(DEVID, 0, 0, &myCdp);
if (status == ADT_SUCCESS) printf("Success.\n");
else printf("FAILURE - Error = %d\n", status);
```

RT-BC

Here is another example that defines a RTBC message (1-T-2-32):

```
***** MESSAGE 1 - RTBC Message *****
memset(&myBCCB, 0, sizeof(myBCCB));

myBCCB.CMD1Info = 0x0C40;          /* BCRT 1-T-2-32 on Bus B */
myBCCB.Csr = ADT_L1_1553_BC_CB_CSR_TYPE_RTBC | ADT_L1_1553_BC_CB_CSR_BUSB;
myBCCB.DelayTime = 1000; /* 100.0 us inter-message gap (dead-bus time) */
myBCCB.NextMsgNum = 2;           /* Go to message 2 */

printf("Writing BCCB 1 . . . ");
status = ADT_L1_1553_BC_CB_Write(DEVID, 1, &myBCCB);
if (status == ADT_SUCCESS) printf("Success.\n");
else printf("FAILURE - Error = %d\n", status);
```

RT-RT

The following example defines a RTRT message (2-R-3-16, 1-T-7-16):

```
***** MESSAGE 2 - RTRT Message *****
memset(&myBCCB, 0, sizeof(myBCCB));

myBCCB.CMD1Info = 0x1060;          /* RTRT 2-R-3-32 1-T-7-32 on Bus A */
myBCCB.CMD2Info = 0x0CE0;
myBCCB.Csr = ADT_L1_1553_BC_CB_CSR_TYPE_RTRT | ADT_L1_1553_BC_CB_CSR_BUSA;
```

```

myBCCB.DelayTime = 1000; /* 100.0 us inter-message gap (dead-bus time) */
myBCCB.NextMsgNum = 3; /* Go to message 3 */

printf("Writing BCCB 2 . . . ");
status = ADT_L1_1553_BC_CB_Write(DEVID, 2, &myBCCB);
if (status == ADT_SUCCESS) printf("Success.\n");
else printf("FAILURE - Error = %d\n", status);

```

Mode Codes without Data

The next example defines a MODE CODE message without data (1-T-0-1 Synchronize without Data):

```

***** MESSAGE 3 - MODE WITHOUT DATA Message *****
memset(&myBCCB, 0, sizeof(myBCCB));

myBCCB.CMD1Info = 0x0C01; /* MODE 1-T-0-1 (Sync without data) on Bus B */
myBCCB.Csr = ADT_L1_1553_BC_CB_CSR_TYPE_MCNODATA | ADT_L1_1553_BC_CB_CSR_BUSB;
myBCCB.DelayTime = 1000; /* 100.0 us inter-message gap (dead-bus time) */
myBCCB.NextMsgNum = 4; /* Go to message 4 */

printf("Writing BCCB 3 . . . ");
status = ADT_L1_1553_BC_CB_Write(DEVID, 3, &myBCCB);
if (status == ADT_SUCCESS) printf("Success.\n");
else printf("FAILURE - Error = %d\n", status);

```

Mode Codes with Data

The next example defines a MODE CODE message with receive data (1-R-31-17 Synchronize with Data) and writes a data word to the CDP buffer:

```

***** MESSAGE 4 - MODE WITH RECEIVE DATA Message *****
memset(&myBCCB, 0, sizeof(myBCCB));

myBCCB.CMD1Info = 0x0BF1; /* MODE 1-R-31-17 (Sync with data) on Bus A */
myBCCB.Csr = ADT_L1_1553_BC_CB_CSR_TYPE_MCDATA | ADT_L1_1553_BC_CB_CSR_BUSA;
myBCCB.DelayTime = 1000; /* 100.0 us inter-message gap (dead-bus time) */
myBCCB.NextMsgNum = 5; /* Go to message 5 */

printf("Writing BCCB 4 . . . ");
status = ADT_L1_1553_BC_CB_Write(DEVID, 4, &myBCCB);
if (status == ADT_SUCCESS) printf("Success.\n");
else printf("FAILURE - Error = %d\n", status);

/* Write the data buffer (CDP) for message 4 */
memset(&myCdp, 0, sizeof(myCdp));
printf("Writing msg 4 buffer 0 . . . ");
myCdp.DATAInfo[0] = 0x0000ABCD;
status = ADT_L1_1553_BC_CB_CDPWrite(DEVID, 4, 0, &myCdp);
if (status == ADT_SUCCESS) printf("Success.\n");
else printf("FAILURE - Error = %d\n", status);

```

The next example defines a MODE CODE message with transmit data (1-T-0-18 Transmit Last Command):

```

***** MESSAGE 5 - MODE WITH TRANSMIT DATA Message *****
memset(&myBCCB, 0, sizeof(myBCCB));

myBCCB.CMD1Info = 0x0C12; /* MODE 1-T-0-18 (Transmit Last Command) on Bus A */
myBCCB.Csr = ADT_L1_1553_BC_CB_CSR_TYPE_MCDATA | ADT_L1_1553_BC_CB_CSR_BUSA;
myBCCB.DelayTime = 1000; /* 100.0 us inter-message gap (dead-bus time) */
myBCCB.NextMsgNum = 6; /* Go to message 6 */

```

```

printf("Writing BCCB 5 . . . ");
status = ADT_L1_1553_BC_CB_Write(DEVID, 5, &myBCCB);
if (status == ADT_SUCCESS) printf("Success.\n");
else printf("FAILURE - Error = %d\n", status);

```

Broadcast BC-RT

The next example defines a BROADCAST BCRT message (31-R-5-32) and writes data to the CDP buffer:

```

***** MESSAGE 6 - BROADCAST BCRT Message *****/
memset(&myBCCB, 0, sizeof(myBCCB));

myBCCB.CMD1Info = 0xF8A0;           /* BCRT 31-R-5-32 on Bus B */
myBCCB.Csr = ADT_L1_1553_BC_CB_CSR_TYPE_BCRT | ADT_L1_1553_BC_CB_CSR_BUSB;
myBCCB.DelayTime = 1000; /* 100.0 us inter-message gap (dead-bus time) */
myBCCB.NextMsgNum = 7;             /* Go to message 7 */

printf("Writing BCCB 6 . . . ");
status = ADT_L1_1553_BC_CB_Write(DEVID, 6, &myBCCB);
if (status == ADT_SUCCESS) printf("Success.\n");
else printf("FAILURE - Error = %d\n", status);

/* Write the data buffer (CDP) for message 6 */
memset(&myCdp, 0, sizeof(myCdp));
printf("Writing msg 6 buffer 0 . . . ");
for (i=0; i<32; i++)
    myCdp.DATAInfo[i] = 0x0000FE00 + i;
status = ADT_L1_1553_BC_CB_CDPWrite(DEVID, 6, 0, &myCdp);
if (status == ADT_SUCCESS) printf("Success.\n");
else printf("FAILURE - Error = %d\n", status);

```

Broadcast RT-RT

The next example defines a BROADCAST RTRT message (31-R-6-16, 1-T-7-16):

```

***** MESSAGE 7 - BROADCAST RTRT Message *****/
memset(&myBCCB, 0, sizeof(myBCCB));

myBCCB.CMD1Info = 0xF8D0;           /* BROADCAST RTRT 31-R-6-16 1-T-7-16 on Bus A */
myBCCB.CMD2Info = 0x0CF0;
myBCCB.Csr = ADT_L1_1553_BC_CB_CSR_TYPE_RTRT | ADT_L1_1553_BC_CB_CSR_BUSA;
myBCCB.DelayTime = 1000; /* 100.0 us inter-message gap (dead-bus time) */
myBCCB.NextMsgNum = 8;             /* Go to message 8 */

printf("Writing BCCB 7 . . . ");
status = ADT_L1_1553_BC_CB_Write(DEVID, 7, &myBCCB);
if (status == ADT_SUCCESS) printf("Success.\n");
else printf("FAILURE - Error = %d\n", status);

```

Broadcast Mode Codes without Data

The next example defines a BROADCAST MODE CODE message without data (31-T-0-1 Synchronize without Data):

```

***** MESSAGE 8 - BROADCAST MODE WITHOUT DATA Message *****/
memset(&myBCCB, 0, sizeof(myBCCB));

```

```

myBCCB.CMD1Info = 0xFC01;           /* BRDCAST MODE 31-T-0-1 (Sync w/o data), Bus B */
myBCCB.Csr = ADT_L1_1553_BC_CB_CSR_TYPE_MCNO DATA | ADT_L1_1553_BC_CB_CSR_BUSB;
myBCCB.DelayTime = 1000; /* 100.0 us inter-message gap (dead-bus time) */
myBCCB.NextMsgNum = 9;             /* Go to message 9 */

printf("Writing BCCB 8 . . . ");
status = ADT_L1_1553_BC_CB_Write(DEVID, 8, &myBCCB);
if (status == ADT_SUCCESS) printf("Success.\n");
else printf("FAILURE - Error = %d\n", status);

```

Broadcast Mode Codes with Data

The final example defines a BROADCAST MODE CODE message with receive data (31-R-0-17 Synchronize with Data) and writes a data word to the CDP buffer:

```

***** MESSAGE 9 - BROADCAST MODE WITHOUT DATA Message *****
memset(&myBCCB, 0, sizeof(myBCCB));

myBCCB.CMD1Info = 0xF811;           /* BRDCAST MODE 31-R-0-17 (Sync w/ data), Bus A */
myBCCB.Csr = ADT_L1_1553_BC_CB_CSR_TYPE_MCDATA | ADT_L1_1553_BC_CB_CSR_BUSA;
myBCCB.DelayTime = 1000; /* 100.0 us inter-message gap (dead-bus time) */
myBCCB.NextMsgNum = ADT_L1_1553_BC_NO_NEXT_MSG; /* Stop BC after this message */

printf("Writing BCCB 9 . . . ");
status = ADT_L1_1553_BC_CB_Write(DEVID, 9, &myBCCB);
if (status == ADT_SUCCESS) printf("Success.\n");
else printf("FAILURE - Error = %d\n", status);

/* Write the data buffer (CDP) for message 9 */
memset(&myCdp, 0, sizeof(myCdp));
printf("Writing msg 9 buffer 0 . . . ");
myCdp.DATAinfo[0] = 0x0000BEEF;
status = ADT_L1_1553_BC_CB_CDPWrite(DEVID, 9, 0, &myCdp);
if (status == ADT_SUCCESS) printf("Success.\n");
else printf("FAILURE - Error = %d\n", status);

```

Note that the “NextMsgNum” field is set to ADT_L1_1553_BC_NO_NEXT_MSG. This tells the BC to stop after this message is sent. Therefore when the BC is started the BC will send this list of nine messages once and stop. If we wanted the list to run continuously we would set the “NextMsgNum” field of the last message to message number 0, thus pointing back to the first message.

Other BC Message Types: NOPs, Delays & Branches

Not only do BCCBs control transmission of 1553 commands, but BCCBs also provide timing and decision/branching control within a list of BCCBs. Although a total of six control options are provided, we have found that the three listed below meet the needs of most applications:

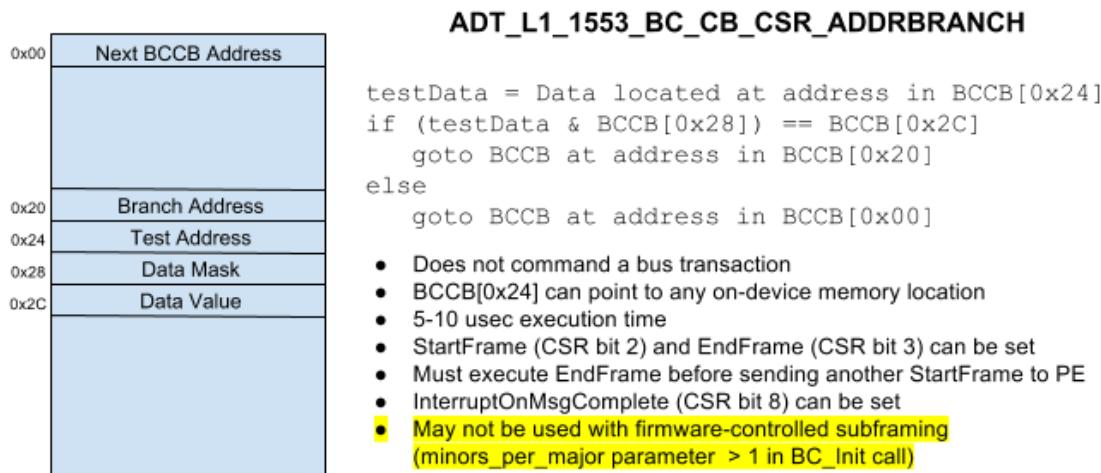
#define ADT_L1_1553_BC_CB_CSR_ADDRBRANCH	0x00100000
#define ADT_L1_1553_BC_CB_CSR_DELAYONLY	0x00400000
#define ADT_L1_1553_BC_CB_CSR_TYPE_NOP	0x02000000

Please see the example programs provided with the AltaAPI release for suggestions on using BCCB branching.

Three additional branching methods are also available: CDPBRANCHONLY, BRNCHONVALUE and BRNCHRETURN. These three options are legacy items that tend to be more complicated to use. Refer to the AltaCore-1553 User's Manual for details on the setup and operation of these three branching options.

ADDRBRANCH

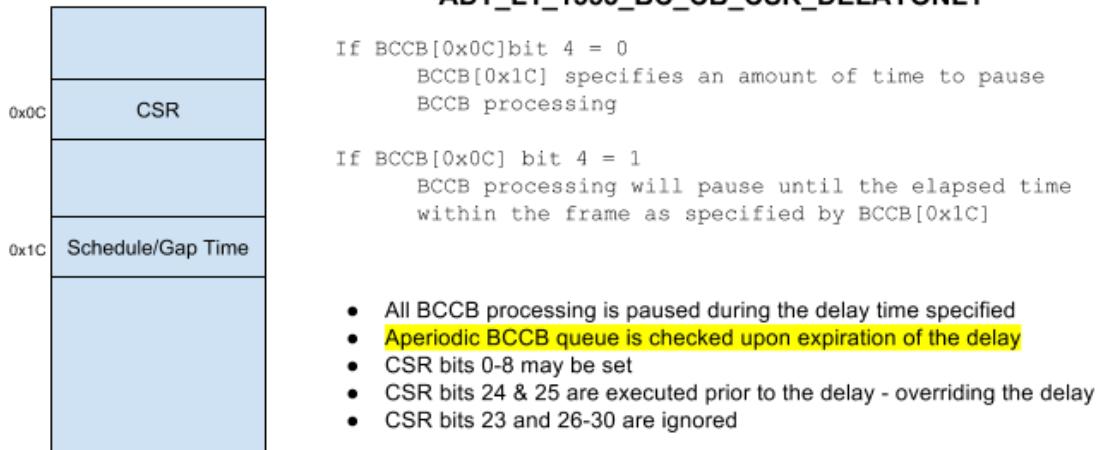
The ADDRBRANCH option (bit 20 in the BCCB CSR) allows BCCB list execution to branch based on a data value stored at any location of on-board memory. The diagram below illustrates the processing logic:



DELAYONLY

A DELAYONLY BCCB (BCCB CSR bit 22) causes the firmware's protocol engine to halt execution of the BCCB list for the specified delay time, or until the specified time since StartFrame. A DELAYONLY BCCB is primarily used to provide entry points within a periodic BCCB list where aperiodic messages can be processed (refer to section 'Aperiodic Messages' for detailed information).

ADT_L1_1553_BC_CB_CSR_DELAYONLY



Note on Inter-Message Gap Time & Frame Message Scheduling

The "DelayTime" field defines the Inter-Message Gap (IMG) time or a Frame Schedule Time (depending on the bit setting in the BCCB CSR). For IMG default setting, this is the delay from the end of the previous message to the start of this message. The delay time has a LSB value of 100 nanoseconds and is DEAD-BUS time (two microseconds will be added when measuring the gap from mid-parity to mid-sync).

For "Frame Schedule Time", which is only valid for Framed Messages (see later paragraphs), this 100 nSec time is from the beginning of the frame and the BCCB will execute when the time has expired. This can be VERY beneficial to reduce jitter in message transmissions in frames. There are several example programs that show both methods.

**DO NOT SET BOTH ADT_L1_1553_BC_CB_CSR_STARTFRAME AND
ADT_L1_1553_BC_CB_CSR_SCHEDTIMING ON THE SAME BCCB! IF
ADT_L1_1553_BC_CB_CSR_STARTFRAME IS SET DO NOT SET
ADT_L1_1553_BC_CB_CSR_SCHEDTIMING! You cannot schedule from the
start of frame on the frame start message.**

NOP

Setting bit 25 in a BCCB's CSR indicates the BCCB should not be executed, although processing a NOP'd BCCB can take 10 to 15 microseconds. The NOP bit can be turned off or on at any time. An interrupt may be generated from a NOP BCCB by setting bit 8 (Int on Message Complete) in the BCCB's CSR word.

Starting and Stopping the BC & Synchronizing Stop

We can start the BC with the ADT_L1_1553_BC_Start function:

```
/* Start BC */
printf("Starting BC . . . ");
status = ADT_L1_1553_BC_Start(DEVID, 0);
if (status == ADT_SUCCESS) printf("Success.\n");
else printf("FAILURE - Error = %d\n", status);
```

This tells the API to start the BC at message number 0. Note that we could specify a different initial message number in the second parameter.

The BC can be stopped using the ADT_L1_1553_BC_Stop function.

```
/* Stop BC */
printf("Stopping BC . . . ");
status = ADT_L1_1553_BC_Stop(DEVID);
if (status == ADT_SUCCESS) printf("Success.\n");
else printf("FAILURE - Error = %d\n", status);
```

We can check to see if the BC is currently running with the ADT_L1_1553_BC_IsRunning function. This can be very useful to see if the BC is still executing so you can synchronize future transmissions.

```
/* Wait for BC to stop */
isRunning = 1;
while (isRunning) {
    status = ADT_L1_1553_BC_IsRunning(DEVID, &isRunning);
}
printf("BC stopped.\n");
```

BC Frame Operation

So far we have only discussed simple BC message lists that use the BCCB "DelayTime" field to specify the inter-message gap time. The BC frame timer can be used to implement more advanced message timing options.

The simplest usage of the frame timer is to send one or more messages in a cyclic frame where the frame repeats at a constant rate regardless of the number of messages, response or no response conditions, etc. For example, we could define a frame of one or more messages that would be transmitted every 100 milliseconds. The first message in the frame would always start at a 10Hz frequency (100 millisecond period) even if we added more messages to a frame, changed inter-message gap times on messages in the frame, etc.

If the STARTFRAME flag (ADT_L1_1553_BC_CB_CSR_STARTFRAME) is set in the BCCB CSR for a message, the PE waits until the frame timer has expired before sending the message. The PE then loads the value from the “FrameTime” field of the BCCB into the frame timer as the new frame time. Therefore you can have different frame times on a frame by frame basis if desired.

When using BC frame timing, you should use both STARTFRAME (ADT_L1_1553_BC_CB_CSR_STARTFRAME) and ENDFRAME (ADT_L1_1553_BC_CB_CSR_ENDFRAME) markers. The first message in the frame should be the STARTFRAME and the last message in the frame should be the ENDFRAME. The PE uses the ENDFRAME marker to increment the PE frame counter and to mark when it is safe to send low-priority aperiodic messages.

The following example sets up a single message in a 100 millisecond frame. We mark this message as both STARTFRAME and ENDFRAME because it is the only message in the frame.

```
/* Define the BCCB for message 0 */
memset(&myBCCB, 0, sizeof(myBCCB));

myBCCB.CMD1Info = 0x0820;           /* BCRT 1-R-1-32 on Bus A */
myBCCB.Csr = ADT_L1_1553_BC_CB_CSR_TYPE_BCRT |
    ADT_L1_1553_BC_CB_CSR_BUSA |
    ADT_L1_1553_BC_CB_CSR_STARTFRAME |          /* Start of frame */
    ADT_L1_1553_BC_CB_CSR_ENDFRAME;            /* End of frame */
myBCCB.FrameTime = 1000000;         /* 100 millisecond frame time (100ns LSB) */
myBCCB.NextMsgNum = 0;             /* Set NEXT msg number to point to this message */

printf("Writing BCCB . . . ");
status = ADT_L1_1553_BC_CB_Write(DEVID, 0, &myBCCB);
if (status == ADT_SUCCESS) printf("Success.\n");
else printf("FAILURE - Error = %d\n", status);

/* Write the data buffer (CDP) for message 0 */
memset(&myCdp, 0, sizeof(myCdp));
printf("Writing msg 0 buffer 0 . . . ");
for (i=0; i<32; i++)
```

```

myCdp.DATAinfo[i] = 0x00001100 + i;
status = ADT_L1_1553_BC_CB_CDPWrite(DEVID, 0, 0, &myCdp);
if (status == ADT_SUCCESS) printf("Success.\n");
else printf("FAILURE - Error = %d\n", status);

```

This message will be transmitted once every 100 milliseconds (msec) using the PE frame timer.

The PE frame counter is incremented every time the PE sees the ENDFRAME marker. This counter can be read with the ADT_L1_1553_BC_GetFrameCount function.

Note that you can have frames of varied length from one another. For example, the first frame could be 100 msec and the second frame could be 200 msec as you program the BCCB Frame Time with the STARTFRAME flag in the first message of the frame. Most systems have common length frames, but you still must set the correct, desired value in the BCCB Frame Time on the first message of every frame.

Note on Inter-Message Gap time & Frame Message Scheduling

The “DelayTime” field defines the Inter-Message Gap (IMG) time or a Frame Schedule Time (depending on the bit setting in the BCCB Control Word). For IMG default setting, this is the delay from the end of the previous message to the start of this message. The delay time has a LSB value of 100 nanoseconds and is DEAD-BUS time (two microseconds will be added when measuring the gap from mid-parity to mid-sync).

For “Frame Schedule Time”, which is only valid for Framed Messages, this 100 nSec time is from the beginning of the frame and the BCCB will execute when the time has expired. This can be VERY beneficial to reduce jitter in message transmissions in frames. There are several example programs that show both methods.

**DO NOT SET BOTH ADT_L1_1553_BC_CB_CSR_STARTFRAME AND
ADT_L1_1553_BC_CB_CSR_SCHEDTIMING ON THE SAME BCCB! IF
ADT_L1_1553_BC_CB_CSR_STARTFRAME IS SET DO NOT SET
ADT_L1_1553_BC_CB_CSR_SCHEDTIMING! You cannot schedule from the
start of frame on the frame start message.**

Advanced BC Frame Operation

The BC frame capabilities allow us to define MAJOR and MINOR frames with messages being sent at different repetition rates. For example, we may need to

send one message at 10Hz, another message at 5Hz, and yet another message at 1Hz. In this case, we would use a 1Hz major frame that contains ten 10Hz minor frames. When we initialize the BC we would specify 10 minor frames per major frame and we would enable subframes.

```
/* BC Initialization - max 100 messages 10 minors per major, enable subframes */
printf("Initializing BC . . . ");
status = ADT_L1_1553_BC_Init(DEVID, 100, 10, ADT_L1_1553_BC_CSR_EN_SUBFRAMES);
if (status == ADT_SUCCESS) printf("Success.\n");
else printf("FAILURE - Error = %d\n", status);
```

Now we will define three messages. We use the “StartFrame” and “FrameRepRate” fields to specify which minor frame the message is first sent on (start frame) and how often (in frames) the message will repeat (FrameRepRate). The StopFrame also needs to be specified to signify the last frame to transmit in (usually set to the last frame number).

```
***** MESSAGE 0 - MODE WITHOUT DATA Message *****
memset(&myBCCB, 0, sizeof(myBCCB));

myBCCB.CMD1Info = 0x0C01;           /* MODE 1-T-0-1 (Sync without data) on Bus B */
myBCCB.Csr = ADT_L1_1553_BC_CB_CSR_TYPE_MCNODATA |
    ADT_L1_1553_BC_CB_CSR_BUSB |
    ADT_L1_1553_BC_CB_CSR_STARTFRAME; /* Start of frame */
myBCCB.FrameTime = 1000000;         /* 100 millisecond frame time (100ns LSB) */
myBCCB.StartFrame = 1;
myBCCB.FrameRepRate = 1;
myBCCB.StopFrame = 10;
myBCCB.NextMsgNum = 1;             /* Go to message 1 */

printf("Writing BCCB 0 . . . ");
status = ADT_L1_1553_BC_CB_Write(DEVID, 0, &myBCCB);
if (status == ADT_SUCCESS) printf("Success.\n");
else printf("FAILURE - Error = %d\n", status);

***** MESSAGE 1 - BCRT Message *****
memset(&myBCCB, 0, sizeof(myBCCB));

myBCCB.CMD1Info = 0x0820;           /* BCRT 1-R-1-32 on Bus A */
myBCCB.Csr = ADT_L1_1553_BC_CB_CSR_TYPE_BCRT |
    ADT_L1_1553_BC_CB_CSR_BUSA;
myBCCB.DelayTime = 1000;            /* 100 microsecond intermessage gap (100ns LSB) */
myBCCB.StartFrame = 1;
myBCCB.FrameRepRate = 2;
myBCCB.StopFrame = 10;
myBCCB.NextMsgNum = 2;             /* Go to message 2 */

printf("Writing BCCB 1 . . . ");
status = ADT_L1_1553_BC_CB_Write(DEVID, 1, &myBCCB);
if (status == ADT_SUCCESS) printf("Success.\n");
else printf("FAILURE - Error = %d\n", status);

/* Write the data buffer (CDP) for message 1 */
memset(&myCdp, 0, sizeof(myCdp));
```

```

printf("Writing msg 1 buffer 1 . . . ");
for (i=0; i<32; i++)
    myCdp.DATAinfo[i] = 0x00001100 + i;
status = ADT_L1_1553_BC_CB_CDPWwrite(DEVID, 1, 0, &myCdp);
if (status == ADT_SUCCESS) printf("Success.\n");
else printf("FAILURE - Error = %d\n", status);

***** MESSAGE 2 - RTTR Message *****
memset(&myBCCB, 0, sizeof(myBCCB));

myBCCB.CMD1Info = 0x1060;           /* RTTR 2-R-3-32 1-T-7-32 on Bus A */
myBCCB.CMD2Info = 0x0CE0;
myBCCB.Csr = ADT_L1_1553_BC_CB_CSR_TYPE_RTTR |
    ADT_L1_1553_BC_CB_CSR_BUSA |
    ADT_L1_1553_BC_CB_CSR_ENDFRAME;      /* End of frame */
myBCCB.DelayTime = 1000; /* 100.0 us inter-message gap (dead-bus time) */
myBCCB.StartFrame = 1;
myBCCB.FrameRepRate = 10;
myBCCB.StopFrame = 10;
myBCCB.NextMsgNum = 0;             /* Go to message 0 */

printf("Writing BCCB 2 . . . ");
status = ADT_L1_1553_BC_CB_Write(DEVID, 2, &myBCCB);
if (status == ADT_SUCCESS) printf("Success.\n");
else printf("FAILURE - Error = %d\n", status);

```

Note that we have highlighted the settings for “StartFrame” and “FrameRepRate”. All three messages have a “StartFrame” value of 1, which means that the message will first be sent on the first frame. Message 0 has a “FrameRepRate” of 1, which means that it will be sent on every frame – therefore this message will be sent at a rate of 10Hz (100 millisecond period). Message 1 has a “FrameRepRate” of 2, which means that it will be sent on every second frame – therefore this message will be sent at a rate of 5Hz (200 millisecond period). Message 2 has a “FrameRepRate” of 10, which means that it will be sent on every tenth frame – therefore this message will be sent at a rate of 1 Hz (1 second period).

NOTE: Branching is not allowed with Subframing.

Aperiodic Messages

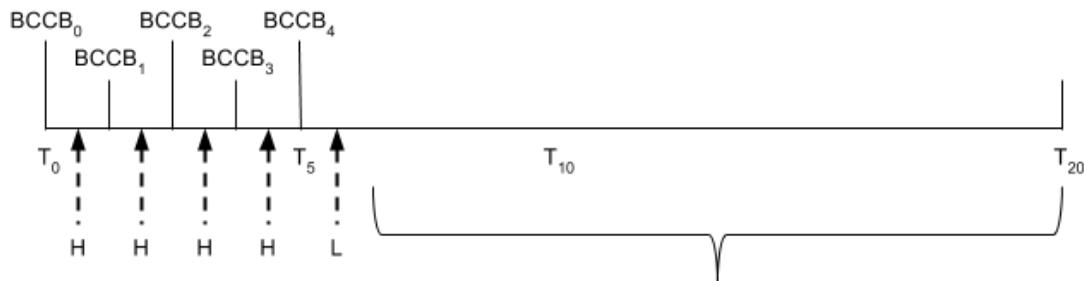
Aperiodic messages are messages that are injected into a cyclic BC frame on demand with the **ADT_L1_1553_BC_AperiodicSend** function.

Aperiodic messages can be specified as low-priority or high-priority. It is important to understand how aperiodic messages are processed in relation to periodic message processing.

As the PE completes processing of each BCCB in a periodic frame list, the high-priority aperiodic message flag is checked for waiting messages. If the flag is set

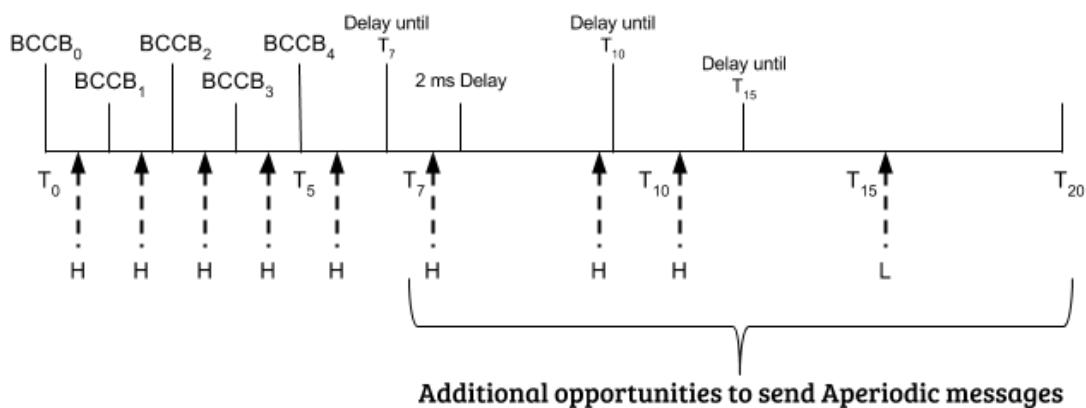
then the high-priority message list is processed before processing the next BCCB in the frame list (thus delaying any other messages in the frame).

After processing the BCCB with the ENDFRAME bit set in its BCCB CSR the PE checks the low-priority aperiodic message flag. Low-priority messages are processed if a list is waiting and the remaining time in the frame is greater than or equal to the amount of time specified in the last parameter of the **ADT_L1_1553_BC_AperiodicSend** function. Upon completion of the low-priority aperiodic message list the PE then halts BCCB processing until the frame time expires. This is illustrated in the diagram below:



H - Check for High Priority Aperiodic BCCB List
L - Check for Low Priority Aperiodic BCCB List

To expedite processing of aperiodic messages over the entire span of a cyclic frame, DELAYONLY BCCBs can be linked to the end of a periodic message list to provide aperiodic processing entry points into the spare time at the end of a frame as shown in the diagram below:



H - Check for High Priority Aperiodic BCCB List
 L - Check for Low Priority Aperiodic BCCB List

BC Retry Operation

The BC can be configured to automatically retry messages if errors are detected in the message or if the RT responds with the BUSY bit set in the status word. Retries are enabled in the BC Control Block using the “retry” word. This word uses the following format:

BC Retry Word																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Retry Pattern (1=Bus A, 0=Bus B) Bit 16 is first retry bit																Retry Count				PE Retry # Attempts				Retry # Attempts				Rsvd			
																1 = Enable Retry on Busy								1 = Enable Retry on Error							

If bit 0 is set this enables retry on error conditions (no response, parity error, etc.). If bit 1 is set this enables retry when the RT responds with the BUSY bit set in the status word. Bits 4-7 are set by the PE to show the number of retries that were attempted for the last retry instance. Bits 8-11 are used by the PE to show the number of retries that were attempted. Bits 12-15 are set by the user to select the maximum number of retries (0 to 15). Bits 16 to 31 are used to select the retry bus pattern – the first retry corresponds to bit 16 and the fifteenth retry

corresponds to bit 30. If the bit is set then the retry message will be sent on Bus A, otherwise it will be sent on Bus B.

For example, if we want to enable retries on errors or on a RT BUSY bit with a maximum of 15 retries with the first retry on Bus B and alternating between busses thereafter, we would program the retry word as follows:

```
myBCCB.Retry = 0xAAAAF003;
```

There is an example program that shows how this can be setup for messages.

Error Injection on Command Words

Errors can be injected on command words using the ADT_L1_1553_BC_InjCmdWordError function. The error settings for command words can be read using the ADT_L1_1553_BC_ReadCmdWordError function.

Note that error injection on data words is defined at the CDP level and works the same way for RT or BC data.

1553 Playback Operation

The 1553 Playback functions are defined in the file ADT_L1_1553_PB.c.

Playback takes recorded BM messages (CDP records) and uses them to regenerate the recorded messages on the 1553 bus. It can be configured to playback RT status words or not on an RT by RT basis.

Note that playback is intended for “normal” messages without protocol errors – it is difficult to accurately playback messages with errors. Abnormal conditions, like broadcast RTBC messages for example (things like this occur in AS4111 RT validation testing), may be discarded as invalid CDP records and the message will be skipped by playback. **Messages with errors or abnormal conditions may play back inaccurately or may be skipped altogether.**

Playback is initialized with the ADT_L1_1553_PB_Allocate function. This function allocates the requested number of Playback Control Blocks (PCB) in board memory. Each PCB corresponds to one message (one CDP record). The PCBs are linked in a circular list. The Protocol Engine on the board will maintain a “current” or “head” pointer to the active PCB. The API will maintain a “tail” pointer to the next Playback Control Block to be written. The ADT_L1_1553_PB_CDPWrite function discussed later uses these two pointers to write new PCBs and to determine when the buffer is full.

```
printf("Allocating PB buffers . . . ");
status = ADT_L1_1553_PB_Allocate(DEVID, PB_BUFFER_SIZE);
if (status == ADT_SUCCESS) printf("Success.\n");
else printf("FAILURE - Error = %d\n", status);
```

The playback buffer size should be large enough that it will not empty faster than your software can write new messages to it. This is determined by the expected message rate and by how often the application can refresh the playback buffer.

Playback can be configured to playback RT status words or not on an RT by RT basis. This is done with the ADT_L1_1553_PB_SetRtResponse function.

```
printf("\nEnabling RT response for all RT addresses . . . ");
status = ADT_L1_1553_PB_SetRtResponse(DEVID, 0xFFFFFFFF);
if (status == ADT_SUCCESS) printf("Success.\n");
else printf("FAILURE - Error = %d\n", status);
```

The example above enables playback of RT status words for all RT addresses. The second parameter is a 32-bit word with one bit for each possible RT address (0-31). If the bit is set then playback of the status word for that RT address is

enabled. If the bit is clear then playback of the status word and transmit data for that RT address is disabled.

It is useful to disable playback of the RT status word and transmit data for a specific RT address in cases where you need to test a specific RT device in a recorded scenario. The RT under test is connected to the bus with the playback device, which acts as the BC and all other RTs. By disabling playback of the RT status word and transmit data for the RT address of the RT under test we allow the UUT to respond to commands for that RT address.

BM CDP records are converted to Playback Control Blocks and written to board memory by the ADT_L1_1553_PB_CDPWrite function. This function will return ADT_SUCCESS or it will return ADT_ERR_BUFFER_FULL if there is no room in the buffer.

```
printf("Writing first CDP . . . ");
status = ADT_L1_1553_PB_CDPWrite(DEVID, &myCdp, 0, 1);
if (status == ADT_SUCCESS) printf("Success.\n");
else printf("FAILURE - Error = %d\n", status);
```

The above example demonstrates writing the FIRST CDP to the playback buffer. The fourth parameter is the key here – a non-zero value tells the API that this is the first CDP record and so the API will allow writing past the playback head pointer. If this parameter is zero, this tells the API that this is NOT the first message and if it sees that the tail pointer equals the head pointer then the buffer is full and it will not accept the message. This “isFirstMsg” parameter allows us to write to the playback buffer to fill it up initially before starting playback. Only the first message should have this parameter set to a non-zero value – all subsequent ADT_L1_1553_PB_CDPWrite calls should set this parameter to zero.

```
printf("Writing last CDP . . . ");
status = ADT_L1_1553_PB_CDPWrite(DEVID, &myCdp, ADT_L1_1553_PBP_CONTROL_STOP, 0);
if (status == ADT_SUCCESS) printf("Success.\n");
else printf("FAILURE - Error = %d\n", status);
```

This example demonstrates writing the LAST CDP to the playback buffer. The third parameter provides options for the PCB being written. The ADT_L1.h header file defines constants for options that can be set in this parameter. These options are:

ADT_L1_1553_PBP_CONTROL_STOP	Stop playback after this PCB
ADT_L1_1553_PBP_CONTROL_LED	Flash user LED on this PCB
ADT_L1_1553_PBP_CONTROL_TRGOUT	Generate TRG OUT on this PCB
ADT_L1_1553_PBP_CONTROL_INT	Generate interrupt on this PCB

ADT_L1_1553_API_PB_CDPWRITE_ATON Directs Playback for Absolute Timing

We want playback to stop after completion of this PCB because this is the last record to be played back, therefore we set the ADT_L1_1553_PBP_CONTROL_STOP option.

Playback Relative verses Absolute (AT) Time Options

The ADT_L1_1553_API_PB_CDPWRITE_ATON option is required for playback sessions that are using Absolute Timing (AT) on the PCBs. Without this option, which is the default setting and used by AltaView, PCBs Time Stamps (64-bit, 20 nanosecond time stamp of the CDP) are transmitted to an offset of zero to the first PCB (which means the AltaCore protocol engine resets its' playback clock to zero and uses the first time-stamp as a relative offset, subtraction, to all PCBs in the session stream – so PCBs start transmitting immediately and then the first CDP time stamp is subtracted to all other PCB time stamps to determine their relative transmission time).

With the AT option, the playback clock does not reset and the absolute time stamp of the CDP (copied to the PCB) is used to determine when the PCB is transmitted. This requires the user to set the Playback clock to a known start value using the ADT_L1_1553_PBS setTime function. There is also a Root PE Playback CSR option for not starting the playback clock until a trigger is received – and there is a Root PE Control Word option to force a trigger through software. The AT method may be the preferred method of playback for multi channel/card systems that want to synchronize clocks and playback streams to known system time values.

For AT to be active, the user must also set the ADT_L1_1553_PB_CSR_NOCLKRST and ADT_L1_1553_PB_CSR_SKPPCBTMBKUP Root Playback Control Bits to not have the clock reset and to decide if PCB times less than current time (time back-ups) are transmitted or skipped. Time back-up PCBs will be immediately transmitted on the wire if the skip option is not selected.

For large playback sessions (you could playback large BM CDP files with thousands of messages) you will not have a “last” CDP in the playback buffer when you first fill the buffer before starting playback. After you start playback you will want to periodically write new CDP records to the playback buffer – you will call the function ADT_L1_1553_PB_CDPWrite repeatedly until it returns ADT_ERR_BUFFER_FULL. This allows you to maintain a continuous playback

of messages. Only when you reach the end of the CDP records to be played back will you set the ADT_L1_1553_PBP_CONTROL_STOP flag.

You can release the resources used by playback with the ADT_L1_1553_PB_Free function.

Playback Start/Stop Control Functions

Playback can be started and stopped with the functions ADT_L1_1553_PB_Start and ADT_L1_1553_PB_Stop.

1553 Signal Generator Operation

The 1553 Signal Generator functions are defined in the file ADT_L1_1553_SG.c. There is an extensive example program provided to show how to setup BC and RT type of signal generation messages. Also, please read the AltaCore 1553 Signal Generator manual section for details for this advanced feature.

The Signal Generator (SG) provides very precise control of the 1553 transceiver output. We first generate a stream of “vectors” where each vector is a two-bit pattern – “10” is high, “01” is low, and “00” or “11” is ground. Each two-bit vector represents 20 nanoseconds.

The Signal Generator is initialized with the ADT_L1_1553_SG_Configure function.

```
status = ADT_L1_1553_SG_Configure(DEVID);
```

The following demonstrates building vectors for a 1-R-1-32 command word and 32 data words:

```
/* Command Word = 0x0820 (RT 1 RECEIVE SA 1 32 words) */
status = ADT_L1_1553_SG_WordToVectors(0x80000820, vectors,
                                         VECARRAYSIZE, &numVectors);

/* 32 Data Words = 0xAB00 to 0xAB1F */
for (i=0; i<32; i++)
    status = ADT_L1_1553_SG_WordToVectors(0x0000AB00 + i, vectors,
                                         VECARRAYSIZE, &numVectors);

/* We will send this on bus A */
status = ADT_L1_1553_SG_CreateSGCB(DEVID, 'A', 0, 0, vectors, numVectors);
```

In this example, “vectors” is an array of ADT_L0_UINT32 with VECARRAYSIZE entries. We use the function ADT_L1_1553_SG_WordToVectors to build vectors for a standard 1553 word, including sync and parity. The first parameter contains the 1553 word in bits 0-15 and bit 31 is the sync type (1 for command sync or 0 for data sync). We first generate vectors for the command word and then generate vectors for 32 data words. All of these vectors are stored in the “vectors” array. When we have all the vectors for the 1553 message, we use the function ADT_L1_1553_SG_CreateSGCB to create a Signal Generator Control Block (SG CB) in board memory. The second parameter to this function is a character ('A' or 'B') that selects the bus to use. The third parameter is reserved and should always be zero.

If we create additional SG Control Blocks they are linked together in sequence as they are created. The fourth parameter for the ADT_L1_1553_SG_CreateSGCB function is the dead-bus gap time from the end of the previous SG CB. This time value has a 100 nanosecond LSB.

The function ADT_L1_1553_SG_Free will free all memory used for SG Control Blocks and reset the signal generator.

The API provides a function for vector-level control rather than 1553 word level control. The function ADT_L1_1553_SG_AddVectors can be used to add individual high, low, or ground vectors (where each vector represents 20 nanoseconds). This gives almost unlimited flexibility to define any vector pattern desired.

SG Start/Stop Control Functions

The Signal Generator can be started and stopped with the functions ADT_L1_1553_SG_Start and ADT_L1_1553_SG_Stop.

Signal Capture

Many Alta cards offer 8-bit A/D Signal Trigger and Capturing on the first channel of 1553 cards (most cards except PMC-1553 and PC104P-1553 have signal capture, but see your hardware manual for details). This is a powerful and unique feature to only Alta standard cards – an industry first!

There are two simple functions to setup and read signal values:

```
status = ADT_L1_1553_SC_ArmTrigger(DEVID, scBus, scCsr, maskValue);  
status = ADT_L1_1553_SC_ReadBuffer(DEVID, scBus, scBuffer1);
```

The SC_ReadBuffer() function is one of the few functions that uses the ADT_L0_UINT8 data type – the above scBuffer1 is a:

```
ADT_L0_UINT8           scBuffer1[2048]
```

Please see the L1 function references, the AltaCore 1553 manual and the example program to see how to setup and use the trigger and reading of values.

1553 Device Interrupts

Interrupts are events programmed in various 1553 data structures (usually a BC, RT or BM CDP) that can signal the user that a message or execution event has occurred. To use interrupts the user really should know the low-level details on how they work with the Alta 1553 device – please see the AltaCore-1553 manual ‘Interrupt Functions’ section.

There are also several sample programs that show how to use interrupts - this is probably the best starting point (the interrupt example programs demonstrate reading and servicing most 1553 device interrupts, including BC, RTs and BM). Essentially any CDP (RT, BC or BM message) or major function can have an option flag bit set in their Control Word to set and log an interrupt event.

1553 Hardware Interrupts

The device is enabled to generate a hardware interrupt signal to the host computer when an interrupt event occurs on the device. The operating system and/or device driver acknowledges the interrupt and signals the API. The API calls the user’s interrupt handler function.

The hardware interrupt approach is preferred when the software needs to perform other tasks and cannot continually poll the board to check for interrupts. This approach does require that the device driver and operating system are properly configured to handle interrupts from the device.

1553 Software Polled Interrupts

The application software periodically calls the user’s interrupt handler function to read new entries from the interrupt queue.

The software polling approach is simpler and more portable because it does not require interaction with the operating system, but is more CPU-intensive because the software must periodically read the board to check for interrupt queue entries. If the CPU cost is acceptable, this approach can provide faster interrupt latency than the hardware interrupt approach because there is no context-switching required to handle the interrupt.

For polling, the developer will need to decide when and how to call the ISR, usually through some sort of software timer. Remember that you should probably poll at a Nyquist rate (usually 2x the event rate that you are trying to capture). Example program ADT_L1_1553_ex_bc2int_polled.c implements a

simple software polling scheme. Contact Alta if you have questions and would like to review the proper approach to the application.

ENET-1553 Devices and Interrupts

All communications to ENET devices are performed exclusively over an Ethernet connection. Subsequently, ENET devices cannot generate a hardware interrupt signal to the application's host computer since there is no connection to the host computer's backplane. In order to capture interrupt events, ENET devices will need to implement a software polling scheme as discussed above to query the Interrupt Queue at some periodic rate.

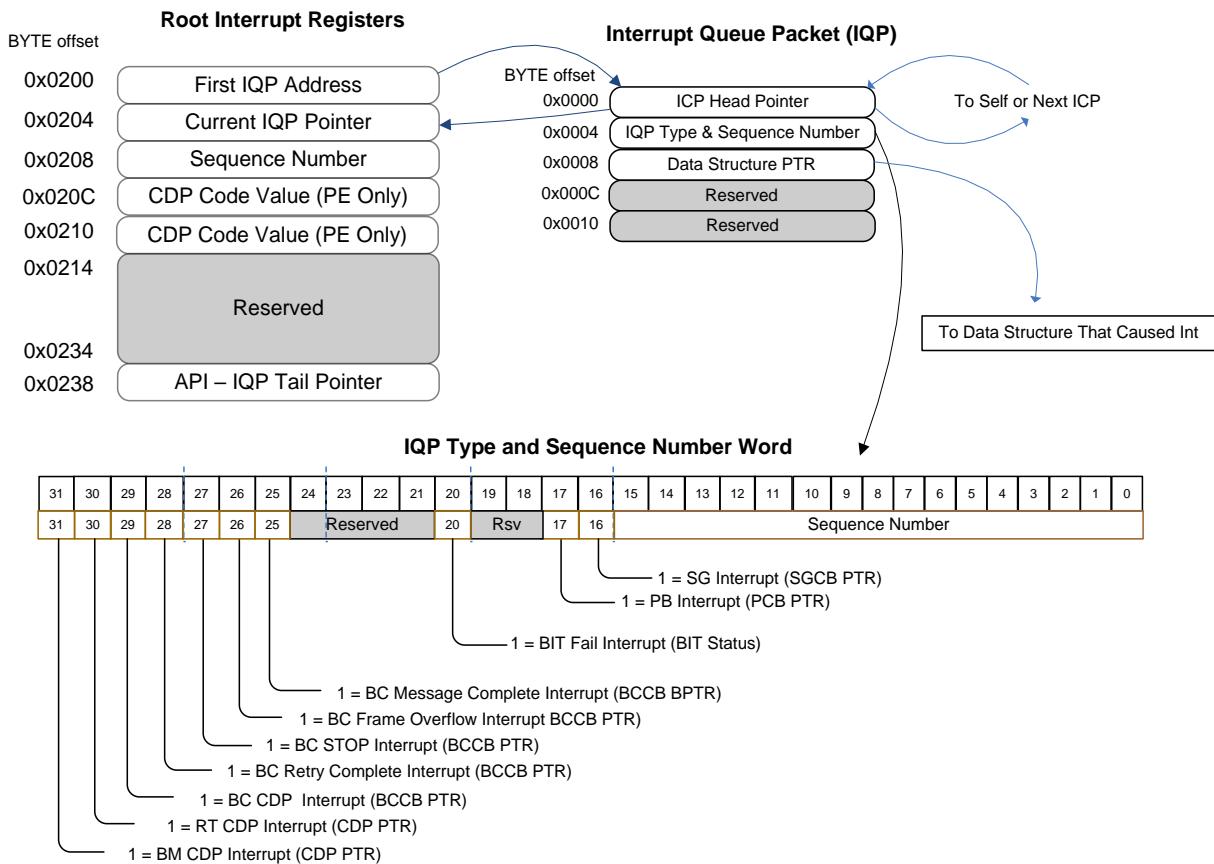
The Interrupt Queue

Alta devices use an "interrupt queue" to store information on interrupt events in the device's on-board memory. The device can be configured to generate interrupt events when a message has been received, when a buffer is full, etc. In some cases, interrupt events can occur faster than the software can detect and process them. The interrupt queue stores information on these events until the software can handle them.

The device initialization functions allow the user to specify the number of entries to allocate for the interrupt queue. For example, the interrupt queue size is set by the user by the ADT_L1_1553_InitDefault() functions. The interrupt queue size is determined by the application's requirements, but even if you do not intend to use interrupt events in your application you must allocate an interrupt queue with at least one entry. A good general practice is to allocate a minimum queue depth of 10 entries. If the application requires more, then allocate a larger queue depth.

The interrupt queue depth needed is determined by the rate at which the device is expected to generate interrupts and by how quickly the software will be able to detect and process interrupts. Real-time operating systems can provide very fast, deterministic, and documented interrupt latencies. Operating systems like Microsoft Windows can have widely varying interrupt latencies based on what the system is doing and the programmer needs to plan for the worst-case latency that the operating system will guarantee (assuming the operating system vendor documents guaranteed interrupt latency).

1553 Interrupt Queue Data Structures



General Device Interrupt Functions

The API provides general interrupt functions in the file ADT_L1_INT.c.

The ADT_L1_INT_HandlerAttach function is used to assign a user interrupt handler function for the device. This function calls the Layer 0 ADT_L0_AttachIntHandler function which configures the device driver and operating system to handle interrupts from the device and to call the user interrupt handler function when an interrupt occurs. This function includes an optional “pUserData” parameter (void *). This pointer will be passed back to the user interrupt handler function when the interrupt occurs and can be used to provide context data. If not used just set this parameter to NULL.

The ADT_L1_INT_HandlerDetach function is used to detach the user interrupt handler function for the device. This function calls the Layer 0 ADT_L0_DetachIntHandler function which configures the device driver and operating system and removes the connection to the user interrupt handler function.

1553 Device Interrupt Functions

Interrupt functions specific to the MIL-STD-1553 protocol are provided in the file ADT_L1_1553_INT.c. Refer to the 1553 example programs for examples demonstrating the use of these functions.

The functions ADT_L1_1553_INT_EnableInt and ADT_L1_1553_INT_DisableInt enable or disable interrupts at the 1553 channel level.

The function ADT_L1_1553_INT_GenInt will cause the 1553 channel to generate a test interrupt (but does not put anything in the interrupt queue). This function is useful for testing interrupts when developing a device driver and Layer 0 API module for a new system.

The function ADT_L1_1553_INT_CheckChannelIntPending is for applications using the “software polling” method to detect interrupts. The application can call this function periodically to determine if there are any new interrupt events for the 1553 channel.

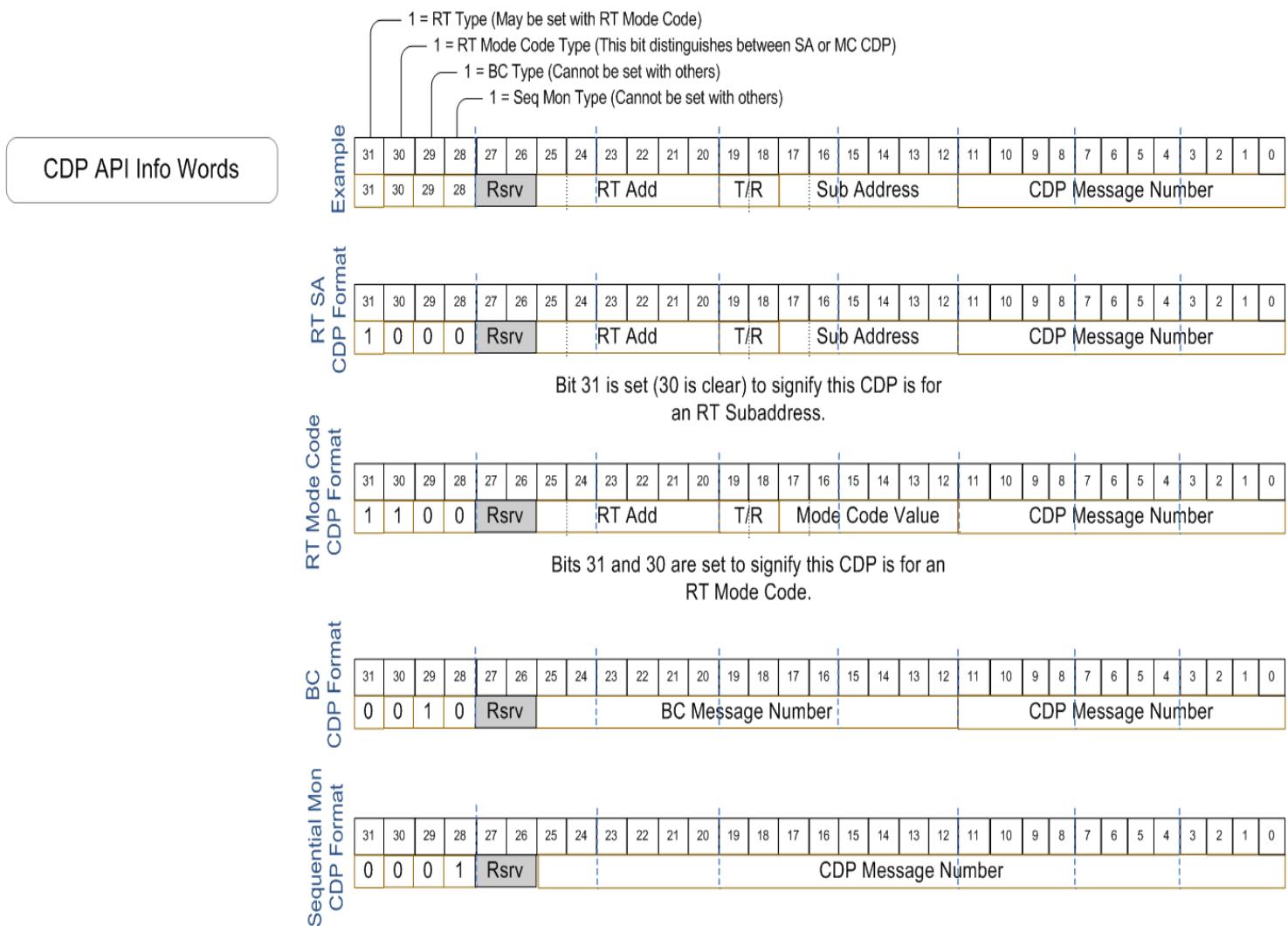
The ADT_L1_1553_IntervalTimerGet and ADT_L1_1553_IntervalTimerSet functions can be used to configure the 1553 device to generate an interrupt at regular time intervals. This can be useful for applications that need to perform actions at specific times that may or may not correspond to 1553 bus messages or events. The example program ADT_L1_1553_ex_bm2int_tmr.c demonstrates usage of the interval timer interrupt.

The function ADT_L1_1553_INT_IQ_ReadNewEntries reads ALL new entries from the interrupt queue into an array (**THIS IS THE RECOMMENDED METHOD**). The function ADT_L1_1553_INT_IQ_ReadEntry will read one new entry from the interrupt queue. This function can be called by the application’s interrupt handler until all new entries in the interrupt queue have been processed. These functions will return two values for each interrupt queue entry: **interrupt type** and **interrupt info** words. These words are derived from the interrupt queue structures on the board, which is shown in the diagram below.

The **interrupt type** (intType) word tells us what kind of interrupt event occurred and therefore tells us how to interpret the **interrupt info** word. This is the “IQP Type and Sequence Number” word shown above.

If the interrupt type is BM CDP, RT CDP, or BC CDP then the “Data Structure PTR” in the firmware interrupt queue entry points to a CDP (Common Data Packet).

CDP/Interrupt API Info Word Format



RT Interrupt Info Word

For RT interrupts, this word contains fields for the RT Address, Transmit/Receive, and Sub Address or Mode Code Value. Note that each of these fields has one more bit than you might expect – we have 6 bits for RT Address (only need 5 bits to represent 0-31), we have 2 bits for T/R (only need 1 bit to represent 0-1), and we have 6 bits for Sub Address (only need 5 bits for 0-31). The reason we have an extra bit for these fields is so we can indicate when

multiple messages are using the same CDP buffer so the user knows that more than one command word can cause this CDP buffer to generate an interrupt.

For example, once you have the RT Address, T/R bit, Sub Address/Mode Code Value, and CDP Buffer Number you can use these as parameters to the ADT_L1_1553_RT_SA_CDPRead or ADT_L1_1553_RT_MC_CDPRead (Mode Codes) function to read the CDP buffer to see the complete message that caused the interrupt. (There is also an API ...CDPReadWords() functions that allow the user to index directly to desired words in the CDP – this can save time in only reading a data word of interest instead of the whole CDP to retrieve one word).

When an RT is first created in memory it uses a single default buffer for all T/R and SA & MCs so everything is “wrapped” until the user allocates specific buffers for the desired sub addresses. If the CDP buffer is used by multiple sources then the high bit in the RT, T/R, and Sub Address/Mode Code field is set in the API info word to indicate this. MOST APPLICATIONS WILL NOT CARE AND CAN DROP THE EXTRA BIT – use the lower 5 bits for RT Address, use the low bit for T/R, and use the lower 5 bits for Sub Address.

BC Interrupts

For BCCB interrupts (Retry, Frame Overflow, Stop and BCCB Complete), the API Info Word is simply the BCCB Message number which caused the interrupt. For the BCCB CDP Interrupt, the API Info Word is described in the figure above, and is a split word to provide the BCCB Message number and CDP buffer number.

BCCB Complete verses BCCB CDP Interrupt

Many customers running BC applications with interrupts ask which of these two interrupts is best for BC applications. In general, the BC CDP interrupt is the most common because this would provide an interrupt when the individual BCCB message is complete, AND the user would get the CDP index number in the Interrupt Queue so they can query the message result.

The BCCB Complete interrupt is used to signify the BCCB transmission is complete, but without regard to which CDP for that BCCB was completed. If there is only one CDP for the BCCB, then setting either interrupt yields the same result – the user was notified that this one message was sent and there is only one CDP buffer. You should only need to use the BCCB Complete Interrupt if you want to be signaled that the respective BCCB message completed and you do not care about which CDP for the BCCB was executed.

BM CDP Interrupt Info Word

Monitor interrupts just provide the “CDP Message Number” (see figure for Sequential Monitor CDP Info Word above), which is the CDP buffer number. Bus Monitor Interrupts come from the user setting the interrupt option for the respective Sequential Monitor CDP. Most interrupt driven BM applications will allocate N CDPs for the BM and then set interrupt on the mid and last CDP of the link list. This would allow “ping pong” buffers to be setup for monitoring. The user can then use the ADT_L1_1553_BM_ReadNewMesgs() functions to retrieve fresh messages.

ARINC 429 Device Operation

This section will discuss the usage of Alta A429 devices. Refer to the ARINC 429 example programs for specific examples showing how the API is used with these devices.

A429 Device Initialization Functions

Initialization functions specific to the ARINC 429 protocol are provided in the file ADT_L1_A429_General.c.

Initialization functions specific to a bank of ARINC channels (device) are provided in the file ADT_L1_A429_General.c. The L1 API provides the low-level steps to allow customization of the initialization and word/label setup, but most customers will use one of two functions:

- **ADT_L1_A429_InitDefault**
- **ADT_L1_A429_InitDefault_ExtendedOptions**

These functions combine low level setup (memory mapping) and bank/channel steps for standard ARINC 429 Label definitions and bit/baud rates.

The ExtendedOptions functions allows the user to further specify if BIT Memory Tests should be run (some applications prefer to have a fast start up without memory BIT test) and if the device should have a bank level (not card) reset (which is useful for application that stop/crash without proper shutdown).

The ADT_L1_API_DEVICEINIT_NOKP option can be used to bypass loading the driver kernel plug-in but in most cases this option should NOT be set. This option only applies to platforms that use the Jungo WinDriver software for the device driver (Windows, Linux, Solaris).

These options are defined as follows:

```
#define ADT_L1_API_DEVICEINIT_FORCEINIT      0x00000001
    Forces Initialization Regardless of Current API State. Often used from application
    crashes or incorrect closing of application
#define ADT_L1_API_DEVICEINIT_NOMEMTEST       0x00000002
    Skips API Memory Test and Initialization (that can take several seconds).
#define ADT_L1_API_DEVICEINIT_NOKP            0x00000004
    Skips loading of the interrupt kernel plug-in. Not recommended for most applications.
#define ADT_L1_API_DEVICEINIT_ROOTRESET       0x80000000
    Forces a hard reset of the device channels. This clears all ARINC low level control
    registers and halts any transmission and reception of data.
```

The ADT_L1_API_DEVICEINIT_NOKP option can be used to bypass loading the driver kernel plug-in but in most cases this

option should NOT be set. This option only applies to platforms that use the Jungo WinDriver software for the device driver (Windows, Linux, Solaris). If this option is used then the application cannot use hardware interrupts. The kernel plug-in is required for hardware interrupts.

The ADT_L1_API_DEVICEINIT_FORCEINIT option should ONLY be used in development and testing. This option is provided for cases where the device may not have been closed properly and is used to override the ADT_ERR_DEVICEINUSE error. This option should NOT be used as the normal initialization method for your application, because it bypasses protection against two applications using the same device.

See the example programs provided to jump start the programming process.

The **ADT_L1_A429_InitDefault** and **ADT_L1_A429_InitDefault_ExtendedOptions** functions can be used to simplify initialization of an A429 device. These function calls the following functions:

ADT_L1_InitDevice
ADT_L1_A429_InitDevice

The ADT_L1_A429_InitDevice function initializes memory and allocates memory for the interrupt queue for an A429 device.

```
status = ADT_L1_A429_InitDevice(DEVID, 10);
```

This function call initializes the A429 device registers and allocates memory for an interrupt queue with a depth of 10 entries. The interrupt queue depth is determined by how often the user expects the device to generate interrupts and how quickly the application will be able to service interrupts. This is discussed further in the section on interrupt operation.

The caller provides the Device ID and the number of interrupt queue entries to allocate and the ADT_L1_A429_InitDefault function will perform the initialization and set the device configuration to the standard settings for the ARINC 429 protocol.

A429 Channel Configuration

The ADT_L1_A429_GetConfig function retrieves the TX-RX channel selection for the device. This RX and TX configuration for the bank device of channels is located at Root PE Offset from ADT_L1.h:

```
#define ADT_L1_A429_PE_TXRX_CHANCONFIG      0x000C
```

Here is the Data Structure from the AltaCore ARINC manual:

Root PE Register 0x000C

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Bank TX Enables (Bit 16 = TX0)																Bank RX Enables (Bit 0 = RX0)															

*Note: The number of Tx and Rx Channels Varies Depending on Order Configuration – See Your Getting Started and Hardware Manual for Details on Configurations and Pin-Outs.

Closing the ARINC Device

At the end of an ARINC (A429) application, or when the application no longer needs to use the Alta device it can close the API as follows:

ADT_L1_CloseDevice(DEVID);

This function frees resources, closes memory management, un-maps memory, and detaches from the device. No API calls should be made for the device after the ADT_L1_CloseDevice call has been made. The device must be initialized again before use.

A429 Receive (RX) Operation

The ARINC Receive (RX) functions are defined in the file ADT_L1_A429_RX.c.

The A429 device has three modes of receive operation – “Channel” and “Multichannel” multi-buffer and Channel Label CVT. At the channel level you can have a multi-buffer for labels received by the channel, and a single current value table buffer (CVT) to read the latest label/word for the channel (for many applications, only a CVT value is required and this can greatly simplify reading the latest value for specific label/word of a channel).

You can also configure each receive channel to be included (or not) in the multichannel receive buffer. The multichannel receive buffer allows you to see labels from any or all receive channels in one buffer. All modes of operation store received labels in “receive packets” (RXP). The API represents a receive packet with the following structure.

```
/*! \brief A429 Receive Packet structure */
typedef struct adt_l1_a429_rxp {
    ADT_L0_UINT32 Control;      /*!< \brief Control Word */
    ADT_L0_UINT32 TimeHigh;    /*!< \brief Timestamp, upper 32-bits */
    ADT_L0_UINT32 TimeLow;     /*!< \brief Timestamp, lower 32-bits */
    ADT_L0_UINT32 Data;        /*!< \brief ARINC word */
} ADT_L1_A429_RXP;
```

The RXP control word contains three key items:

1. Bit 31 (mask 0x80000000) is the “decode error” flag. This bit will be set if there was any error detected for the label (parity error, encoding error, etc.)
2. Bits 24 to 27 (mask 0x0F000000) are a 4-bit field containing the RX channel number (0-15, corresponding to channel number 1-16).
3. Bit 15 is set by the API to one for Multi-Channel RXPs only. This bit is set to zero for normal channel RXPs.

The next two words are the 64-bit, 20ns LSB time stamp.

The Data word is the 32-bit A429 word/label (left justified if using less than 32 bits – the MSB will always be in bit position 31).

RX Packet (RxP)

RxP – Control/Status Word

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
31	30	29	R	RX Channel Number				Channel RX Sequence Number				15	API Info: RXP Number																		



RxP – Time High

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
32-bit, 20 nsec Time High Stamp																															

RxP – Time Low

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
32-bit, 20 nsec Time Low Stamp																															

RxP – Data Word

MSB	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1-32-bit RX Label/Word																																

Receive Channel Operation

Individual receive channels are initialized with the function

ADT_L1_A429_RX_Channel_Init. This function sets the bit-rate and allocates the requested number of receive packets (RXPs). This function also determines whether or not to include the channel in the multichannel (MC) receive (MCRX) buffer AND selects if the Label CVT buffer should be allocated (as an options parameter per below).

```
/* A429 API Init Option Values for ADT_L1_A429_RX_Channel_Init() */
#define      ADT_L1_A429_API_RX_MCON      0x00000001
#define      ADT_L1_A429_API_RX_LABELCVTON 0x00000002
```

The channel is started with the ADT_L1_A429_RX_Channel_Start function.

The ADT_L1_A429_RX_Channel_ReadNewRxPs function reads all new receive packets (since the last time this function was called) from the channel. The user must be careful to allocate enough RXPs to buffer between function calls. If only one RXP is allocated (by the function ADT_L1_A429_RX_Channel_Init) for a data table, then this function will always return the one RXP regardless if the RXP is fresh or not.

There are also a read/write CVT functions to access the single-buffer RXP location (if setup) for each label value (first 8 bits of the label/word – the label

index into the CVT is raw 8 bits, so the user will need to reverse the 8 bit value from standard ARINC-429 label values, which are transmitted in reverse order to the rest of the label).

The channel is stopped with the ADT_L1_A429_RX_Channel_Stop function.

The ADT_L1_A429_RX_Channel_Close function frees memory used by the receive channel and closes it.

Two sets of “mask/compare” interrupt registers are provided to allow the user to interrupt on two different conditions for each receive channel. The function ADT_L1_A429_RX_Channel_SetMaskCompare writes these settings and the function ADT_L1_A429_RX_Channel_GetMaskCompare reads these settings.

Multichannel Receive Operation

The multichannel receive buffer is initialized with the function ADT_L1_A429_RXMC_BufferCreate. This function allocates the requested number of receive packets (RXPs).

Start the individual receive channels with the ADT_L1_A429_RX_Channel_Start function.

The ADT_L1_A429_RXMC_ReadNewRxPs function reads all new receive since the last time this function was called) from the MCRX buffer. The user must be careful to allocate enough RXPs to buffer between function calls. If only one RXP is allocated for a data table, then this function will always return the one RXP regardless if the RXP is fresh or not.

Stop the individual receive channels with the ADT_L1_A429_RX_Channel_Stop function.

The ADT_L1_A429_RXMC_BufferFree function frees memory used by the multichannel receive buffer.

The following basic API flow figure and example code provides a crude example to initialize a ARINC device bank (of channels) and setup RX Channel 0 for receiving labels. The user has three options for receiving labels in channel buffers, but you must call the RX_Channel_Init() function and should allocate at least 1 RXP at the Channel Level.

AltaAPI Operational Flow – ARINC Receive

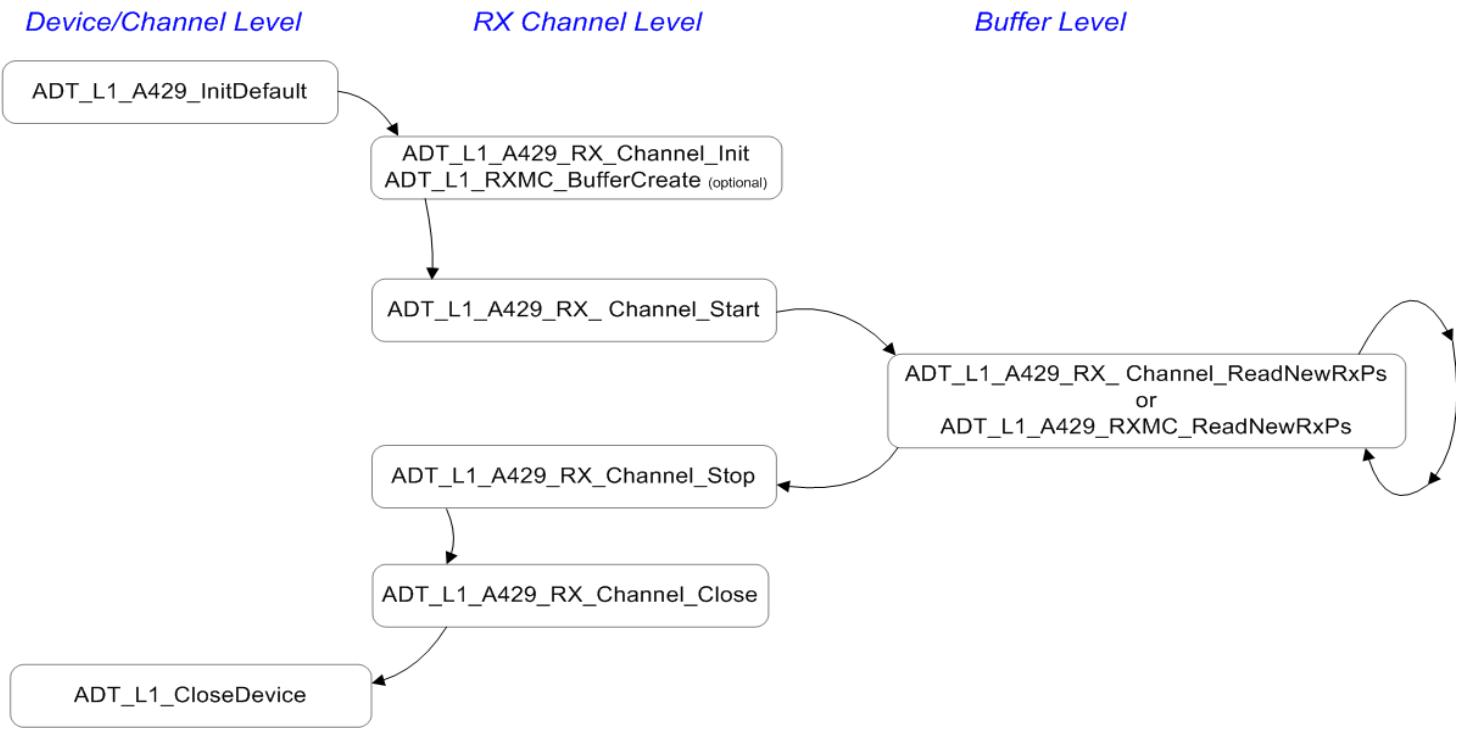


Figure ARINC-RX-1: Basic Flow for Channel Level RX

```

#include <stdio.h>
#include "ADT_L1.h"

#define DEVID (ADT_PRODUCT_PMCA429 | ADT_DEVID_BOARDNUM_01 |
ADT_DEVID_CHANNELTYPE_A429 | ADT_DEVID_BANK_01)

int main()
{
    ADT_L0_UINT32      status, API_MC_CVT_Options, rxChan=0; label=0;
    ADT_L1_A429_RXP   myRXP, chRXPs[50], mcRXPs[200];

    status = ADT_L1_A429_InitDefault(DEVID, 10);

    /* This is only setting needed to configure CVT – Optional RX Method */
    API_MC_CVT_Options = ADT_L1_A429_API_RX_LABELCVTON;
    /* Ths following channel init for RX Channel 0 with 100K baud, 100 RXP buffers for the
    channel and turns on the CVT Table for the Channel.
    */
    status = ADT_L1_A429_RX_Channel_Init(DEVID, rxChan, 100000, 100,
API_MC_CVT_Options); // Allocates 100 RXPs for Channel Buffer

    /* Optional Multi Channel RX of all Channels */
    status = ADT_L1_A429_RXMC_BufferCreate(DEVID, 80);

    status = ADT_L1_A429_RX_Channel_Start(DEVID, 0);

    /* The user needs to decide which read function to use */
    /* User RX Application goes after one of the reads below */
    status = ADT_L1_A429_RX_Channel_CVTReadRxP(DEVID,
                                                rxChan, label, &myRxp);
    status = ADT_L1_A429_RX_Channel_ReadNewRxPs(DEVID,
                                                rxChan, 50, chRXPs);
    status = ADT_L1_A429_RXMC_ReadNewRxPs(DEVID, 200, mcRXPs);

    status = ADT_L1_A429_RX_Channel_Stop(DEVID, 0);
    status = ADT_L1_A429_RX_Channel_Close(DEVID, 0);
    status = ADT_L1_CloseDevice(DEVID);
}

```

Label LSB/MSB for ARINC-429 RXPs and TXPs

The “Data” word of the RXP or TXP is a raw 32-bit left justified word. ARINC-429 labels (the first 8 bits of the 32-bit word – Bits 0-7) are normally LSB first for bit 7, which is reversed from the rest of the word. The user must reverse the label field for transmission or reception as appropriate.

For example: a label of 0x02 would be need to flipped to 0x40. The following code snippet would flip a label from an RXP (reverse the logic for TXP).

```
ADT_L0_UINT32      RXPlabel;
ADT_L1_A429_RXP   myRXP_buffer;
/* Flip label MSB->LSB */
RXPlabel = myRXP_buffer.Data & 0x000000FF;
tempLabel |= (RXPlabel & 1) << 7;
tempLabel |= (RXPlabel & 2) << 5;
tempLabel |= (RXPlabel & 4) << 3;
tempLabel |= (RXPlabel & 8) << 1;
tempLabel |= (RXPlabel & 16) >> 1;
tempLabel |= (RXPlabel & 32) >> 3;
tempLabel |= (RXPlabel & 64) >> 5;
tempLabel |= (RXPlabel & 128) >> 7;
```

A429 Transmit (TX) Operation

ARINC label/words can be transmitted by simple ARINC 429 label transmissions controlled by the application and through more advanced periodic and aperiodic structures to automate/repeat transmission schedules.

Prior to any transmission, the user will need to call the ADT_L1_A429_TX_Channel_Init to setup bit rate and the number of TXCBs (set at least one TXCB even if only using the simple one-shot method) – this function setups the protocol engine for standard ARINC-429 bit encoding/decoding parameters.

The API also provides ADT_L1_A429_TX_Channel_GetConfig and ADT_L1_A429_TX_Channel_SetConfig to setup for non ARINC-429 style transmissions (these calls are not necessary for standard ARINC-429 communications, which 90% of customers use). There are several example programs that show various transmission options with standard 429, 717 and other communication settings.

To use the simple, one-shot transmission method, the following function call after ADT_L1_A429_TX_Channel_Init:

- The ADT_L1_A429_TX_Channel_SendLabel function is used to send a single “one-shot” label.
- The ADT_L1_A429_TX_Channel_SendLabelBlock function is used to send a block of labels.

For more advanced transmission control and for periodic setups, your application will need to define Transmit Control Blocks (TXCB) and Transmit Packets (TXPs).

The following paragraphs detail API calls when using TXCB transmission options. **Note that when using periodic transmission of labels you should add the faster labels to the transmit list before slower labels – this allows the firmware to process the faster labels first. If slower labels are in the list before faster labels you can see more jitter in the faster labels.**

The API represents the TXCB with the following structure:

```
/*! \brief A429 Transmit Control Block structure */
typedef struct adt_l1_a429_txcb {
    ADT_L0_UINT32 TxcbNum;           /*!< \brief TXCB number */
    ADT_L0_UINT32 NextTxcbNum;       /*!< \brief Next TXCB number */
    ADT_L0_UINT32 Control;           /*!< \brief Control Word */
    ADT_L0_UINT32 TotalTxpCount;     /*!< \brief Num TxP for the TXCB */
    ADT_L0_UINT32 CurrTxpIndex;      /*!< \brief Index to current TxP */
    ADT_L0_UINT32 TxPeriod500us;      /*!< \brief TX Period, 500us LSB */
} ADT_L1_A429_TXCB;
```

All bits in the TXCB control word are reserved except for:

ADT_L1_A429_TXCB_CONTROL_STOPONTXBCOMP 0x00000010

This bit can be set to stop transmission when this TXCB completes execution.

This can allow one-shot TXCB(s).

ADT_L1_A429_TXCB_CONTROL_INTONTXBCOMP 0x000000100

This bit can be set to register an interrupt event when the TXCB has finished transmitting.

The TxPeriod500us field sets the transmit period for the block (with a 500us LSB). For example, if you want this TXCB to transmit every 100ms, set this value to 200.

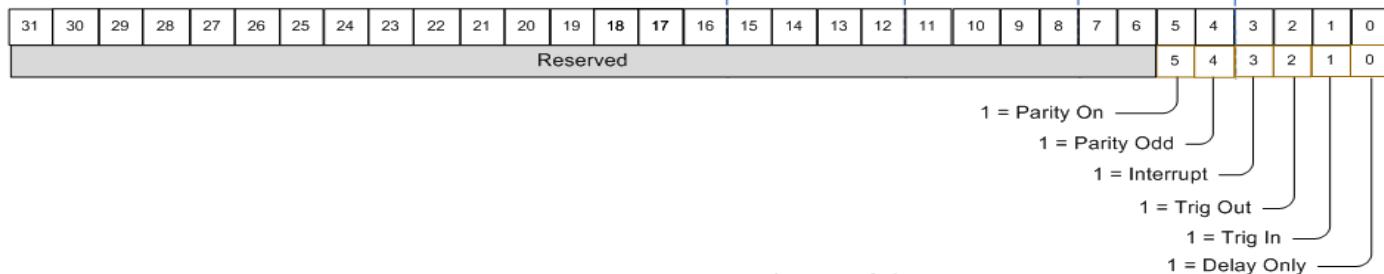
The API represents the TXP with the following structure:

```
/*! \brief A429 Transmit Packet structure */
typedef struct adt_l1_a429_txp {
    ADT_L0_UINT32 Control;           /*!< \brief Control Word */
    ADT_L0_UINT32 Reserved;         /*!< \brief Reserved Word */
    ADT_L0_UINT32 Delay;            /*!< \brief Delay Word (100ns LSB) */
    ADT_L0_UINT32 Data;             /*!< \brief ARINC word */
} ADT_L1_A429_TXP;
```

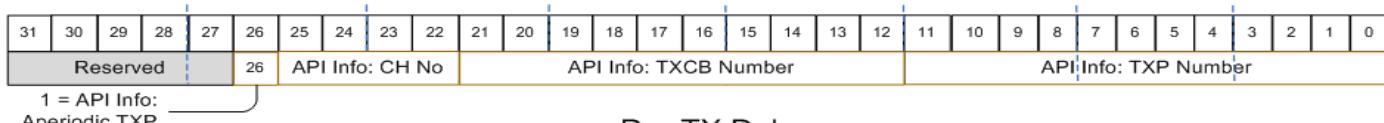
The TXP words are shown below:

TX Packet (TxP)

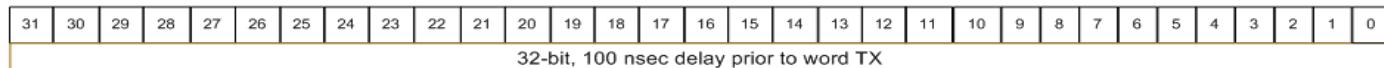
TxP – Control Word



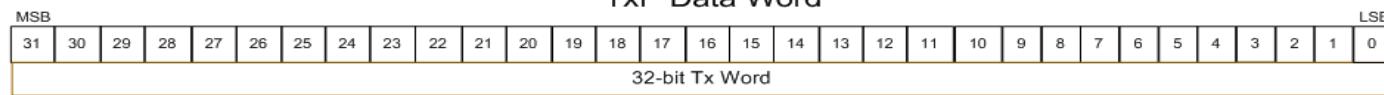
TxP – Reserved (API Info)



Pre-TX Delay



TxP Data Word



The API initializes the TXP Control Word to use odd parity (standard for A429).

The Pre-TX Delay field is used to set the minimum inter-label gap time (100ns LSB). The A429 specification defines the minimum gap as four bit-times for the selected bit-rate. For example, at a 100KHz bit-rate, each bit takes 10 microseconds so four bit-times would be 40 microseconds, or 400 with a 100ns LSB.

Each transmit channel is initialized with the ADT_L1_A429_TX_Channel_Init function. This function sets the bit rate and allocates the requested number of transmit control blocks (TXCB) for the channel.

The ADT_L1_A429_TX_Channel_SendLabel function is used to send a single “one-shot” label (if no periodic labels are defined) or to inject a single “aperiodic” label (if periodic labels are defined using TXCBs and TXPs below).

The ADT_L1_A429_TX_Channel_SendLabelBlock function is used to send a block of “one-shot” labels (if no periodic labels are defined) or to inject a block of “aperiodic” labels (if periodic labels are defined using TXCBs and TXPs below).

The ADT_L1_A429_TX_Channel_CB_TXPAllocate function allocates the requested number of transmit packets (TXP) for a specific TXCB. This function should be done for ALL TXCBs planned for future use BEFORE any TXCB definitions (or before any ADT_L1_A429_TX_Channel_CB_Write calls). When this allocation call is made, the API build a TXCB table of pointers so that TXCB message reference numbers can be indexed and referenced for later use.

The ADT_L1_A429_TX_Channel_CB_Read function is used to read a TXCB. The ADT_L1_A429_TX_Channel_CB_Write function is used to write a TXCB.

The ADT_L1_A429_TX_Channel_CB_TXPRead function is used to read a TXP. The ADT_L1_A429_TX_Channel_CB_TXPWrite function is used to write a TXP.

The ADT_L1_A429_TX_Channel_Start function starts a transmit channel. The ADT_L1_A429_TX_Channel_Stop function stops a transmit channel.

The ADT_L1_A429_TX_Channel_IsRunning function can be used to determine if a transmit channel is running.

The ADT_L1_A429_TX_Channel_CB_TXPFree function frees memory for a specific TXCB.

The function ADT_L1_A429_TX_Channel_Close frees memory for a transmit channel and closes it.

The following basic API Flow figure and example programs show basic calls for setting up an ARINC device bank, TX channel zero. The first program setups one TXP label for transmitting at a 100 millisecond repeated period. A second program follows that shows using the SendLabel() function. The SendLabel function is much easier, but does not allow you to build lists and control the transmission frequency with fine granularity (the SendLabel feature allows you to send labels in between TXCB periods).

Several programs in the distribution provide more detailed and practical examples.

AltaAPI Operational Flow – ARINC Transmit with Frequency Scheduling

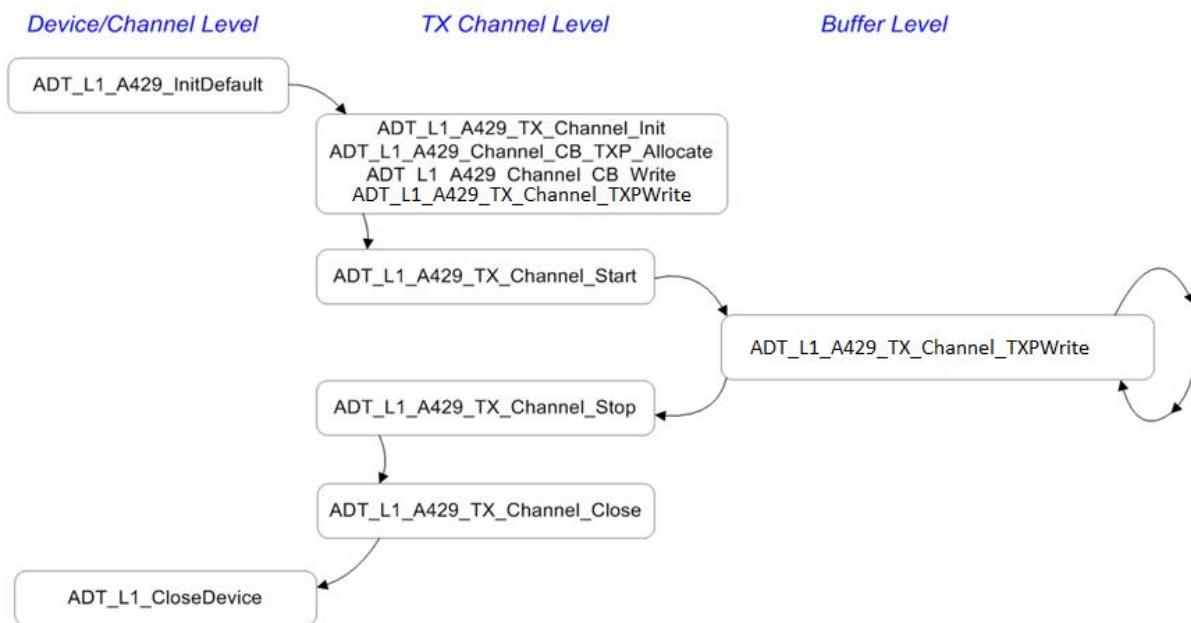


Figure ARINC-TX-1: Basic TX Scheduling API Flow

```

/* Basic API Calls for TXCB-TXP Scheduling Setup */
#include <stdio.h>
#include <memory.h>
#include "ADT_L1.h"

#define DEVID (ADT_PRODUCT_PMCA429 | ADT_DEVID_BOARDNUM_01 |
           ADT_DEVID_CHANNELTYPE_A429 | ADT_DEVID_BANK_01)
int main()
{
    ADT_L0_UINT32      status, API_MC_CVT_Options, txChan=0;

    status = ADT_L1_A429_InitDefault(DEVID, 10);

    /* Inits TX Channel 0 to 100K Baud and 10 TXCBs */
    status = ADT_L1_A429_TX_Channel_Init(DEVID, txChan, 100000, 10);
    /* Allocates 1 TXP to TXCB (base offset 0) */
    /* Remember to make ALL Allocate Calls Prior to Setup of TXCBs */
    status = ADT_L1_A429_TX_Channel_CB_TXPAllocate(DEVID, txChan, 0, 1);

    /* Setup the TXCB Period and Other Options */
    memset(&myTXCB, 0, sizeof(myTXCB));
    myTXCB.NextTxcbNum = ADT_L1_A429_TXCB_NO_NEXT_TXCB;
    myTXCB.Control = 0;
    /* 100ms period (10Hz) 500us LSB=200 */
    myTXCB.TxPeriod500us = 200;
    status = ADT_L1_A429_TX_Channel_CB_Write(DEVID, txChan, 0,      &myTXCB);

    /* Setup the TXP Value, Parity and Gap for transmission */
    myTXP.Control =      ADT_L1_A429_TXP_CONTROL_PARITYON |
                         ADT_L1_A429_TXP_CONTROL_PARITYODD;
    /* A429 gap of 4 bit-times, at 100KHz, this is 40us.
       At 100ns LSB, this is 400 */
    myTXP.Delay = 400;
    myTXP.Data = 0x12345678; /* the LSB byte (0x8) is the RAW A429 Label */
    status = ADT_L1_A429_TX_Channel_CB_TXPWrite(DEVID, txChan, 0, 0, &myTXP);

    status = ADT_L1_A429_TX_Channel_Start(DEVID, txChan, 0);
    /* Do User Application Here */
    status = ADT_L1_A429_TX_Channel_Stop(DEVID, txChan);
    status = ADT_L1_msSleep(100); // Let TX finish

    status = ADT_L1_A429_TX_Channel_Close(DEVID, txChan);
    status = ADT_L1_CloseDevice(DEVID);
}

```

```

/* Basic TX with SendLabel() Function */
#include <stdio.h>
#include <memory.h>
#include "ADT_L1.h"

#define DEVID (ADT_PRODUCT_PMCA429 | ADT_DEVID_BOARDNUM_01 |
ADT_DEVID_CHANNELTYPE_A429 | ADT_DEVID_BANK_01)
int main()
{
    ADT_L0_UINT32 status, txChan=0;

    status = ADT_L1_A429_InitDefault(DEVID, 10);

    /* Inits TX Channel 0 to 100K Baud and 10 TXCBs */
    /* Ignore TXCBs for Send Label Only Applications */
    status = ADT_L1_A429_TX_Channel_Init(DEVID, txChan, 100000, 10);

    status = ADT_L1_A429_TX_Channel_SendLabel(DEVID, txChan, 0x12345678);
    status = ADT_L1_msSleep(1); // Let TX finish

    status = ADT_L1_A429_TX_Channel_Close(DEVID, txChan);
    status = ADT_L1_CloseDevice(DEVID);
}

```

Sending Aperiodic TXCB/TXP Label Lists

The user may want to send a label(s) list while periodic TXCBs are executing and the SendLabel/Block() functions do not provide a low enough level of TXP control. You can use a spare TXCB/TXP block to define execute an aperiodic list (the SendLabel/Block() functions actually use this method, but hide all the low level setup and control is handled for application). Aperiodic transmission of a TXCB/TXP list can be done through the **ADT_L1_A429_TX_Channel_AperiodicSend** and **ADT_L1_A429_TX_Channel_AperiodicIsRunning** functions. A list of TXPs in a single TXCB can be posted during periodic TXCB execution for execution after the current TXCB is being executed. Please see these function calls and the example programs for simple setup methods.

Label LSB/MSB for ARINC-429 RXPs and TXPs

The “Data” word of the RXP or TXP is a raw 32-bit left justified word. ARINC-429 labels (the first 8 bits of the 32-bit word – Bits 0-7) are normally LSB first for bit 7, which is reversed from the rest of the word. The user must reverse the label field for transmission or reception as appropriate.

For example: a label of 0x02 would be need to flipped to 0x40. The following code snippet would flip a label from an RXP (reverse the logic for TXP).

```
ADT_L0_UINT32      RXPlabel;
ADT_L1_A429_RXP   myRXP_buffer;
/* Flip label MSB->LSB */
RXPlabel = myRXP_buffer.Data & 0x000000FF;
tempLabel |= (RXPlabel & 1) << 7;
tempLabel |= (RXPlabel & 2) << 5;
tempLabel |= (RXPlabel & 4) << 3;
tempLabel |= (RXPlabel & 8) << 1;
tempLabel |= (RXPlabel & 16) >> 1;
tempLabel |= (RXPlabel & 32) >> 3;
tempLabel |= (RXPlabel & 64) >> 5;
tempLabel |= (RXPlabel & 128) >> 7;
```

A429 Playback Operation

The A429 Playback functions are defined in the file ADT_L1_A429_PB.c.

Playback takes recorded RXPs and uses them to regenerate the recorded messages on an A429 TX Channel. The respective TX channels word/label definitions (specialized parity/word lengths, encoding – for example ARINC 717 word formats) must be setup by the application prior to starting playback (default is standard ARINC-429 labels with a programmed bit rate).

Playback is initialized with the ADT_L1_A429_PB_Init and ADT_L1_A429_PB_CB_PXPAllocate functions. The allocate function allocates the requested number of Playback Control Blocks (PBCB) and Playback Transmit Packets (PXPs) in board memory. Each PBCB can have 1-N PXPs where a PXP corresponds to a RXP input. The PBCBs need to be in a circular list. The Protocol Engine on the board will maintain a “current” or “head” pointer to the active PCB. The API will maintain a “tail” pointer to the next Playback Control Block to be written. The ADT_L1_A429_PB_RXPWrite function discussed later uses these two pointers to write new (refresh) PBCBs/PXPs to determine when the buffer is full.

ARINC Playback uses the same TX low level protocol engine structures for a given TX channel. Root TX protocol engine data registers and the same (playback does not use transmission periods as transmission is determined by the RXP/PXP time stamp), and Transmit Control Blocks and Playback Control Blocks data structures are essentially the same. So TXCBs and PBCB can be used interchangeably.

```
/* Initialize TX channel 1 for playback with a bit-rate of 100KHz */
status = ADT_L1_A429_TX_Channel_PB_Init(DEVID, 0, 100000, PB_BUFFER_SIZE);
if (status != ADT_SUCCESS) printf("ERROR %d on ADT_L1_A429_TX_Channel_PB_Init\n", status);
else
{
    /* Allocate PBCB and one PXP for each message */
    for (i=0; i<PB_BUFFER_SIZE; i++)
    {
        status = ADT_L1_A429_TX_Channel_PB_CB_PXPAllocate(DEVID, 0, i, 1);
    }

    /* Link the PBCB messages in a circular list */
    for (i=0; i<PB_BUFFER_SIZE; i++)
    {
        status = ADT_L1_A429_TX_Channel_PB_CB_Read(DEVID, 0, i, &txcb);
        txcb.TxcbNum = i;
        if (i < (PB_BUFFER_SIZE - 1))
            txcb.NextTxcbNum = i+1;
        else
            txcb.NextTxcbNum = 0;
        status = ADT_L1_A429_TX_Channel_PB_CB_Write(DEVID, 0, i, &txcb);
    }
}
```

The number of PBCBs and PXP perPBCB should be large enough that they will not empty faster than your software can write new messages to it. This is determined by the expected message rate and by how often the application can refresh the playback buffer.

RXPs records are converted to PXPs and written to board memory by the ADT_L1_A429_PB_RXPWrite function. This function will return ADT_SUCCESS or it will return ADT_ERR_BUFFER_FULL if there is no room in the buffer (there are no empty PBCBs).

```
printf("Writing first RXP Block . . . ");
status = ADT_L1_A429_PB_RXPWrite(DEVID, txChan, numOfRXPs, &RXPbuffer, pbOptions, isFirst);
if (status == ADT_SUCCESS) printf("Success.\n");
else printf("FAILURE - Error = %d\n", status);
```

The above example demonstrates writing the FIRST PXP block to the playback buffer (an empty PBCB). The user passes in the DEVID, the TX Channel Number, the number of PXPs to be written to a PBCB buffer, the starting address of the RXP buffer, and options variable for setting Stop and Absolute Timing options (discussed in subsequent paragraphs, and a flag for designating if this RXP write is the first of the playback session.

This “isFirst” parameter is the key here – a non-zero value tells the API that this is the first RXP/PBCB in the session and allows the API to write past the playback head pointer. If this parameter is zero, this tells the API that this is NOT the first message and if it sees that the tail pointer equals the head pointer then the buffer is full and it will not accept the message. This “isFirstMsg” parameter allows us to write to the playback buffer to fill it up initially before starting playback. Only the first message should have this parameter set to a non-zero value – all subsequent ADT_L1_A429_PB_RXPWrite calls should set this parameter to zero.

```
printf("Writing Last RXP Block . . . ");
pbOptions |= ADT_L1_A429_PBCB_CONTROL_STOPONPBCBCOMP;
status = ADT_L1_A429_PB_RXPWrite(DEVID, txChan, numOfRXPs, &RXPbuffer, pbOptions, isFirst);
if (status == ADT_SUCCESS) printf("Success.\n");
else printf("FAILURE - Error = %d\n", status);
```

This example demonstrates writing the LAST PBCB/RXP buffer to the playback buffer. The pbOptions parameter provides options for the RXP block being written.

The ADT_L1.h header file defines constants for options that can be set in this parameter. These options are:

ADT_L1_A429_PBCB_CONTROL_STOPONPBCBCOMP	Stop playback after this PCB
ADT_L1_A429_PBCB_CONTROL_INTONPBCBCOMP	Generate interrupt PBCB Complete
ADT_L1_A429_PB_API_ATON	Directs Playback for Absolute Timing

We want playback to stop after completion of this PCB because this is the last record to be played back, therefore we set the
ADT_L1_A429_PBCB_CONTROL_STOPONPBCBCOMP.

Playback Relative verses Absolute (AT) Time Options

The ADT_L1_A429_API_ATON option is required for playback sessions that are using Absolute Timing (AT) for transmitting RXPs. Without this option, which is the default setting and used by AltaView, RXPs Time Stamps (64-bit, 20 nSec time stamp of the RXP) are transmitted to an offset of zero to the first RXP (which means the AltaCore protocol engine resets its' playback clock to zero and uses the first time-stamp as a relative offset, subtraction, to all RXPs in the session stream – so RXPs start transmitting immediately and then the first RXP time stamp is subtracted to all other RXP time stamps to determine their relative transmission time).

With the AT option, the playback clock does not reset and the absolute time stamp of the RXP (copied to a PXP) is used to determine when the PXP is transmitted. This requires the user to set the Playback clock to a known start value using the ADT_L1_A429_PBSetTime function. The AT method may be the preferred method of playback for multi channel/card systems that want to synchronize clocks and playback streams to known system time values

For AT to be active, the user must also set the
ADT_L1_A429_PECSR_NOCLKRST and
ADT_L1_A429_PECSR_SKPPCBTMBKUP Root PE Control Bits to not have the clock reset and to decide if PXP times less than current times (time back-ups) are transmitted or skipped. Time back-up PXPs will be immediately transmitted on the wire if the skip option is not selected.

For large playback sessions, you probably only need about 10 PBCBs with about 200 PXPs; The application will probably only need to refresh RXPs at a $\frac{1}{2}$ second or less rate. Again, the number of PBCBs and PXPs is determined by the playback RXP rates and your applications ability to periodically refresh new RXP packets. After you start playback you will want to periodically write new RXP blocks to the playback buffer – you will call the function
ADT_L1_A429_PB_RXPWrite repeatedly until it returns
ADT_ERR_BUFFER_FULL. This allows you to maintain a continuous playback

of messages. Only when you reach the end of the RXP records (end of playback file) will you set the ADT_L1_A429_PBCB_CONTROL_INTONPBCBCOMP flag.

There are sample playback example programs that describe the above logic and should greatly speed-up the code development.

You can release the resources used by playback with the ADT_L1_A429_PB_Free function.

Playback Start/Stop Control Functions

Playback can be started and stopped with the functions ADT_L1_A429_PB_Start and ADT_L1_A429_PB_Stop.

A429 Signal Generator Operation

The A429 Signal Generator functions are defined in the file ADT_L1_A429_SG.c.

The Signal Generator (SG) provides very precise control of the A429 transmitter output for one selected transmit channel. We first generate a stream of “vectors” where each vector is a two-bit pattern – “10” is high, “01” is low, and “00” or “11” is ground. Each two-bit vector represents 100 nanoseconds.

The Signal Generator is initialized with the ADT_L1_A429_SG_Configure function. This selects TX Channel 1 (0 in second parameter) and high-speed slew rate (1 in third parameter).

```
status = ADT_L1_A429_SG_Configure(DEVID, 0, 1);
```

The following demonstrates building vectors for an A429 label word (0x12345678):

```
/* Label word = 0x12345678, 100KHz bit rate (5us half bit time) */
status = ADT_L1_A429_SG_WordToVectors(0x12345678, 50, vectors,
                                         2000, &numVectors);
```

In this example, “vectors” is an array of ADT_L0_UINT32 with VECARRAYSIZE entries. We use the function ADT_L1_A429_SG_WordToVectors to build vectors for a standard A429 word. All of these vectors are stored in the “vectors” array.

The second parameter to this function is the half-bit-time with a 100ns LSB. In this example, we want a bit-rate of 100KHz, which has a bit-time of 10us. Therefore the half-bit-time is 5us, which is 50 with a 100ns LSB.

When we have all the vectors for the A429 word, we use the function ADT_L1_A429_SG_CreateSGCB to create a Signal Generator Control Block (SG CB) in board memory.

```
status = ADT_L1_A429_SG_CreateSGCB(DEVID, 0, 0, vectors, numVectors);
```

If we create additional SG Control Blocks they are linked together in sequence as they are created.

The function ADT_L1_A429_SG_Free will free all memory used for SG Control Blocks and reset the signal generator.

The API provides a function for vector-level control rather than A429 word level control. The function ADT_L1_A429_SG_AddVectors can be used to add individual high, low, or ground vectors (where each vector represents 100 nanoseconds). This gives almost unlimited flexibility to define any vector pattern desired.

SG Start/Stop Control Functions

The Signal Generator can be started and stopped with the functions ADT_L1_A429_SG_Start and ADT_L1_A429_SG_Stop.

A429 Device Interrupt Functions

Interrupts are events programmed in various A429 data structures (usually a TxCB, TxP or RxP) that can signal the user that a message or execution event has occurred. To use interrupts the user really should know the low-level details on how they work with the Alta A429 device – please see the AltaCore-ARINC manual ‘Interrupt Functions’ section.

There are also several sample programs that show how to use interrupts - this is probably the best starting point. Essentially any A429 data structure or major function can have an option flag bit set in their Control Word to set and log an interrupt event.

A429 Hardware Interrupts

The device is enabled to generate a hardware interrupt signal to the host computer when an interrupt event occurs on the device. The operating system and/or device driver acknowledges the interrupt and signals the API. The API calls the user’s interrupt handler function.

The hardware interrupt approach is preferred when the software needs to perform other tasks and cannot continually poll the board to check for interrupts. This approach does require that the device driver and operating system are properly configured to handle interrupts from the device.

A429 Software Polled Interrupts

The application software periodically calls the user’s interrupt handler function to read new entries from the interrupt queue.

The software polling approach is simpler and more portable because it does not require interaction with the operating system, but is more CPU-intensive because the software must periodically read the board to check for interrupt queue entries. If the CPU cost is acceptable, this approach can provide faster interrupt latency than the hardware interrupt approach because there is no context-switching required to handle the interrupt.

For polling, the developer will need to decide when and how to call the ISR, usually through some sort of software timer. Remember that you should probably poll at a Nyquist rate (usually 2x the event rate that you are trying to capture). Contact Alta if you have questions and would like to review the proper approach to the application.

ENET-A429 Devices and Interrupts

All communications to ENET devices are performed exclusively over an Ethernet connection. Subsequently, ENET devices cannot generate a hardware interrupt signal to the application's host computer since there is no connection to the host computer's backplane. In order to capture interrupt events, ENET devices will need to implement a software polling scheme as discussed above to query the Interrupt Queue at some periodic rate.

General A429 Interrupt Functions

Interrupt functions specific to the ARINC 429 protocol are provided in the file ADT_L1_A429_INT.c. Refer to the A429 example programs for examples demonstrating the use of these functions.

The functions ADT_L1_A429_INT_EnableInt and ADT_L1_A429_INT_DisableInt enable or disable interrupts at the A429 device level.

The function ADT_L1_A429_INT_CheckDeviceIntPending is for applications using the “software polling” method to detect interrupts. The application can call this function periodically to determine if there are any new interrupt events for the A429 device.

The function ADT_L1_A429_INT_IQ_ReadEntry will read one new entry from the interrupt queue. This function can be called by the application's interrupt handler until all new entries in the interrupt queue have been processed.

The function ADT_L1_A429_INT_IQ_ReadNewEntries reads ALL new entries from the interrupt queue into an array. This provides another method for the application's interrupt handler to read interrupt queue entries.

ADT_L1_A429_INT_IQ_ReadEntry and
ADT_L1_A429_INT_IQ_ReadNewEntries return two types of interrupt words returned by these functions; interrupt type and interrupt info.

The **interrupt type** word tells us what kind of interrupt event occurred. This is the “IQP Type and Sequence Number” word shown above.

The **interrupt info** word returned from these functions contains the following:

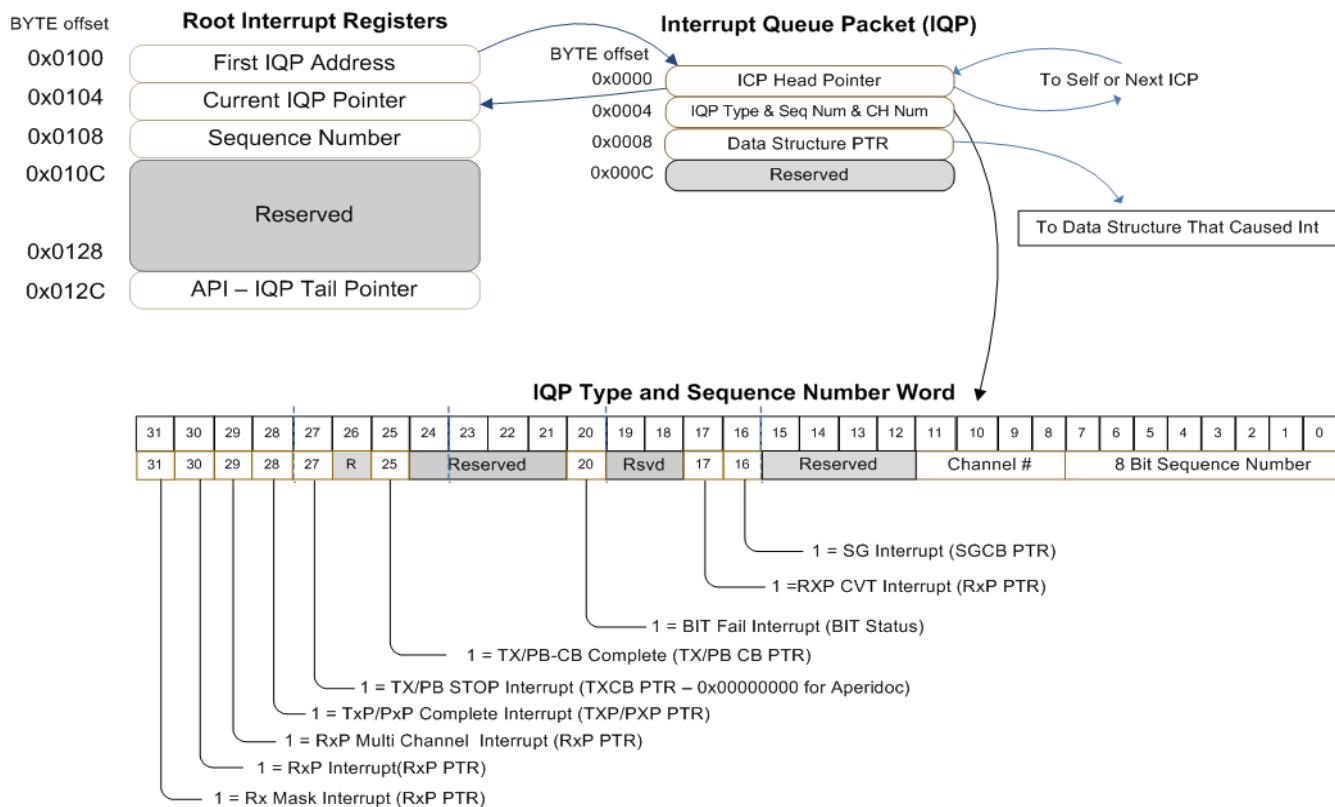
- For TXP/PXP Interrupt: the pInfo value will be a copy of the respective TXP/PXP Reserved-API Info Word (2nd Word of the TXP/PXP). For a TXP this is Channel # in bits 22-31, TXCB/message # in bits 12-21, TXP # in bits 0-11.

- For RXP Interrupts: the pInfo value will be a copy of the respective RXP Control Word (1st Word of the RXP). For Multi Channel (MC) RXPs, Bit 15 of the RXP Control Word is set to differentiate between MC and straight channel RXP data tables.
- For TX/PB-CB Interrupt Complete: the pInfo value will be the respective API TXCB/message Number of the TX/PB-CB (Offset 0x002C).
- For TX/PB Stop Interrupt:
 - For Aperiodic pInfo will be 0xFFFFFFFF if the TX Stop was from a Aperiodic TXP (PB does not have aperiodics)
 - For TX Stop from a TX-CB or PB-CB, then the pInfo value will be the respective API TX/PB-CB number (Offset 0x002C).
- All other interrupt types will return zero for pInfo

The pInfo values allow access to TXP, RXP and TX/PB-CB messages using standard, non-pointer calls in the API.

Make sure to review the TXP and RXP data structures in the ARINC Quick Reference or AltaCore-ARINC documents. This review will help in understanding the information passed in the interrupt queue for parsing/processing entries.

ARINC Interrupt Queue Data Structures



Other ARINC Setup & Usage (ARINC 717)

The Alta A429 product has universal encoder/decoder capability for baud rates from 500 to 200K bits per second (baud symbols). The user can also define the bit encoding/decoding and bits per word. Further, some RX and TX channel pairs can be set for 2 channels of 5V differential input/output for ARINC 717 and other interfaces like single bit triggers. (Refer to the Hardware User's Manual for your specific device for details.)

Each RX and TX channel has two control registers (Setup1 & 2 for RX and CSR1 and CSR2 for TX) that set the universal encoding and decoding values for the channel. Please see the AltaCore ARINC manual for details on these registers. There are sample programs that show how to setup and use these channels for ARINC 717 and for one-bit triggering. Once the encoder/decoder PE registers are setup, then the user simply uses normal RX or TX setup and control data structures for buffering and transmission control.

Here is a code snippet from example program "717rxtx1.c" for ARINC 717 setup.

```
/* Adjust RX Channel Setup for 717 Parameters */
/* This will force the PE to look for SF Sync "Lock" (4 word sync pattern found prior to saving
RXPs) */
status = ADT_L1_A429_RX_Channel_GetConfig(DEVID, RX_CH, &RXsetup1, &RXsetup2);
if (status != ADT_SUCCESS) printf("\nERROR %d on
ADT_L1_A429_RX_Channel_GetConfig\n", status);

/* Update RX Setup1 Reg with 717 Settings - bitRate should already be set */
RXsetup1 = (RXsetup1 & 0x03FF0000) | ((BitsPerWord << 26) |
(ADT_L1_A429_RXREG_SETUP1_MODEBITS_5V717<<6) |
ADT_L1_A429_RXREG_SETUP1_717SYNCON);
status = ADT_L1_A429_RX_Channel_SetConfig(DEVID, RX_CH, RXsetup1, RXsetup2);
if (status != ADT_SUCCESS) printf("\nERROR %d on ADT_L1_A429_RX_Channel_SetConfig\n",
status);

/* Setup RX CH 717 CSR Word & Sync Words*/
status = ADT_L1_A717_RX_Channel_SetConfig(DEVID, RX_CH, wordsPerSF, frameSyncs[0]
<< (32-BitsPerWord), frameSyncs[1] << (32-BitsPerWord),

frameSyncs[2] << (32-BitsPerWord), frameSyncs[3] << (32-BitsPerWord));
if (status != ADT_SUCCESS) printf("\nERROR %d on ADT_L1_A717_RX_Channel_SetConfig\n",
status);

/* TX CHANNEL SETUP */
/* Init TX Channel */
status = ADT_L1_A429_TX_Channel_Init(DEVID, TX_CH, bitRate, 1);
if (status != ADT_SUCCESS) printf("\nERROR %d on ADT_L1_A429_TX_Channel_Init %d\n",
status, TX_CH);

/* Adjust Root TX Channel CSRs for 717 Parameters */
status = ADT_L1_A429_TX_Channel_GetConfig(DEVID, TX_CH, &TXcsr1, &TXcsr2);
```

```
if (status != ADT_SUCCESS) printf("\nERROR %d on ADT_L1_A429_TX_Channel_GetConfig\n",  
status);  
  
/* Update TX CSR2 Reg with 717 Settings - bitRate should already be set */  
TXcsr2 = (TXcsr2 & 0x03FF0000) | ((BitsPerWord << 26) |  
(ADT_L1_A429_RXREG_SETUP1_MODEBITS_5V717<<6));  
status = ADT_L1_A429_TX_Channel_SetConfig(DEVID, TX_CH, TXcsr1, TXcsr2);  
if (status != ADT_SUCCESS) printf("\nERROR %d on ADT_L1_A429_TX_Channel_SetConfig\n",  
status);
```

Layer 1 API Files

The Layer 1 API consists of a top-level header file (ADT_L1.h) and a set of C files that implement the Layer 1 functions.

The Layer 1 API module consists of the following files:

- ADT_L1.h** – Top-level header file for Layer 1.
- ADT_L1_General.c** – Layer 1 general functions.
- ADT_L1_MemMgmt.c** – Layer 1 memory management functions.
- ADT_L1_BIT.c** – Layer 1 built-in-test functions.
- ADT_L1_INT.c** – Layer 1 interrupt functions.
- ADT_L1_BoardGlobal.c** – Layer 1 board-level global device functions.
- ADT_L1_1553_General.c** – Layer 1 1553 general functions.
- ADT_L1_1553_INT.c** – Layer 1 1553 interrupt functions.
- ADT_L1_1553_RT.c** – Layer 1 1553 remote terminal functions.
- ADT_L1_1553_BM.c** – Layer 1 1553 bus monitor functions.
- ADT_L1_1553_BC.c** – Layer 1 1553 bus controller functions.
- ADT_L1_1553_SG.c** – Layer 1 1553 signal generator functions.
- ADT_L1_1553_PB.c** – Layer 1 1553 playback functions.
- ADT_L1_A429_General.c** – Layer 1 A429 general functions.
- ADT_L1_A429_INT.c** – Layer 1 A429 interrupt functions.
- ADT_L1_A429_RX.c** – Layer 1 A429 receive functions.
- ADT_L1_A429_RX_MC.c** – Layer 1 A429 multichannel receive functions.
- ADT_L1_A429_TX.c** – Layer 1 A429 transmit functions.
- ADT_L1_A429_SG.c** – Layer 1 A429 signal generator functions.
- ADT_L1_A429_PB.c** – Layer 1 A429 playback functions.

Layer 1 Example Programs

In the ADT_L1/Examples directory (windows installation), there are two folders that provide numerous examples of 1553 (m1553) and ARINC (A429) Layer 1 programs. Most customers will start with one of these programs to start building their application. The programs also apply to most other operating systems.

Layer 1 API – Key Type Definitions

ADT_L1_1553_CDP

The structure ADT_L1_1553_CDP defines the “Common Data Packet” used for 1553 BC, RT, and BM data buffers.

```
/*! \brief BC/RT/BM CDP structure */
typedef struct adt_l1_1553_cdp {
    ADT_L0_UINT32 NextPtr;           /*!< \brief CDP next pointer */
    ADT_L0_UINT32 BMCount;          /*!< \brief BM message count */
    ADT_L0_UINT32 APIInfo;          /*!< \brief Reserved for API */
    ADT_L0_UINT32 Rsvd1;            /*!< \brief Reserved */
    ADT_L0_UINT32 Rsvd2;            /*!< \brief Reserved */
    ADT_L0_UINT32 MaskValue;        /*!< \brief Mask value */
    ADT_L0_UINT32 MaskCompare;      /*!< \brief Mask compare value */
    ADT_L0_UINT32 CDPControlWord;   /*!< \brief CDP control word */
    ADT_L0_UINT32 CDPStatusWord;    /*!< \brief CDP status word */
    ADT_L0_UINT32 TimeHigh;         /*!< \brief Timestamp, upper 32-bits */
    ADT_L0_UINT32 TimeLow;          /*!< \brief Timestamp, lower 32-bits */
    ADT_L0_UINT32 IMGap;            /*!< \brief Intermessage gap, 100ns LSB */
    ADT_L0_UINT32 Rsvd3;            /*!< \brief Reserved */
    ADT_L0_UINT32 CMD1info;         /*!< \brief Command 1 info */
    ADT_L0_UINT32 CMD2info;         /*!< \brief Command 2 info */
    ADT_L0_UINT32 STS1info;         /*!< \brief Status 1 info */
    ADT_L0_UINT32 STS2info;         /*!< \brief Status 2 info */
    ADT_L0_UINT32 DATAinfo[32];     /*!< \brief Data word info */
} ADT_L1_1553_CDP;
```

ADT_L1_1553_BC_CB

The structure ADT_L1_1553_BC_CB defines the “BC Control Block” used to define 1553 BC message blocks.

```
/*! \brief BC Control Block structure */
typedef struct adt_l1_1553_bc_cb {
    ADT_L0_UINT32 NextMsgNum;       /*!< \brief Next message number */
    ADT_L0_UINT32 Retry;            /*!< \brief BC Retry word */
    ADT_L0_UINT32 Csr;              /*!< \brief BC CB CSR */
    ADT_L0_UINT32 CMD1Info;         /*!< \brief Command word 1 info */
    ADT_L0_UINT32 CMD2Info;         /*!< \brief Command word 2 info */
    ADT_L0_UINT32 FrameTime;        /*!< \brief Frame time, 100ns LSB, applies if start of frame */
    ADT_L0_UINT32 DelayTime;        /*!< \brief Delay time, 100ns LSB, IM gap or delay from SOF */
    ADT_L0_UINT32 BranchMsgNum;     /*!< \brief Branch message number */
    ADT_L0_UINT32 StartFrame;       /*!< \brief Start frame number */
    ADT_L0_UINT32 StopFrame;        /*!< \brief Stop frame number */
    ADT_L0_UINT32 FrameRepRate;     /*!< \brief Frame repetition rate */
    ADT_L0_UINT32 MsgNum;           /*!< \brief Message number for this BCCB */
    ADT_L0_UINT32 NumBuffers;       /*!< \brief Number of CDPs allocated to this BCCB */
} ADT_L1_1553_BC_CB;
```

ADT_L1_A429_RXP

The structure ADT_L1_A429_RXP defines the “Receive Packet” used to store labels (and raw data) received on ARINC 429 receive channels. This is also used to read an RXP archive file for Playback operations.

```
/*! \brief A429 Receive Packet structure */
typedef struct adt_l1_a429_rxp {
    ADT_L0_UINT32 Control;           /*!< \brief Control Word */
    ADT_L0_UINT32 TimeHigh;          /*!< \brief Timestamp, upper 32-bits */
    ADT_L0_UINT32 TimeLow;           /*!< \brief Timestamp, lower 32-bits */
    ADT_L0_UINT32 Data;              /*!< \brief ARINC word */
} ADT_L1_A429_RXP;
```

ADT_L1_A429_TXP

The structure ADT_L1_A429_TXP defines the “Transmit Packet” used to store labels (and raw data) to be transmitted on ARINC 429 transmit (including playback) channels.

```
/*! \brief A429 Transmit Packet structure */
typedef struct adt_l1_a429_txp {
    ADT_L0_UINT32 Control;           /*!< \brief Control Word */
    ADT_L0_UINT32 Reserved;          /*!< \brief Reserved Word */
    ADT_L0_UINT32 Delay;             /*!< \brief Delay Word (100ns LSB) */
    ADT_L0_UINT32 Data;              /*!< \brief ARINC word */
} ADT_L1_A429_TXP;
```

Label LSB/MSB for ARINC-429 RXPs and TXPs

The “Data” word of the RXP or TXP is a raw 32-bit left justified word. ARINC-429 labels (the first 8 bits of the 32-bit word – Bits 0-7) are normally LSB first for bit 7, which is reversed from the rest of the word. The user must reverse the label field for transmission or reception as appropriate.

For example: a label of 0x02 would be need to flipped to 0x40. The following code snippet would flip a label from an RXP (reverse the logic for TXP).

```
ADT_L0_UINT32      RXPlabel;
ADT_L1_A429_RXP   myRXP_buffer;
/* Flip label MSB->LSB */
RXPlabel = myRXP_buffer.Data & 0x000000FF;
tempLabel |= (RXPlabel & 1) << 7;
tempLabel |= (RXPlabel & 2) << 5;
tempLabel |= (RXPlabel & 4) << 3;
tempLabel |= (RXPlabel & 8) << 1;
tempLabel |= (RXPlabel & 16) >> 1;
tempLabel |= (RXPlabel & 32) >> 3;
tempLabel |= (RXPlabel & 64) >> 5;
tempLabel |= (RXPlabel & 128) >> 7;
```

ADT_L1_A429_TXCB

The structure ADT_L1_A429_TXCB defines the “Transmit Control Block” (TXCB) used to control transmissions on ARINC 429 transmit channels. This is also used for Playback Control Blocks (PBCB)

```
/*! \brief A429 Transmit Control Block structure */
typedef struct adt_l1_a429_txcb {
    ADT_L0_UINT32 TxcbNum;          /*!< \brief TXCB number */
    ADT_L0_UINT32 NextTxcbNum;     /*!< \brief Next TXCB number */
    ADT_L0_UINT32 Control;         /*!< \brief Control Word */
    ADT_L0_UINT32 TotalTxpCount;   /*!< \brief Number of TxP for this block */
    ADT_L0_UINT32 CurrTxpIndex;    /*!< \brief Index to current TxP */
    ADT_L0_UINT32 TxPeriod500us;   /*!< \brief TX Period, 500us LSB */
} ADT_L1_A429_TXCB;
```

Layer 1 API Constants

Error Code Constants

The Layer 1 API reserves the error code range 1000 to 1999. These constants define integer (32-bit) values that will be returned by the API functions to indicate either SUCCESS or an error code indicating why the function failed.

```
***** Layer 1 Error Codes (1000 to 1999) *****/
#define ADT_ERR_BAD_INPUT          1000  /*!< \brief Bad input parameters. */
#define ADT_ERR_MEM_TEST_FAIL      1001  /*!< \brief Failed memory test. */
#define ADT_ERR_MEM_MGT_NO_INIT    1002  /*!< \brief Memory Management not initialized for the device ID */
#define ADT_ERR_MEM_MGT_INIT       1003  /*!< \brief Memory Management already initialized for device ID */
#define ADT_ERR_MEM_MGT_NO_MEM     1004  /*!< \brief Not enough memory available */
#define ADT_ERR_BAD_DEV_TYPE       1005  /*!< \brief Bad device type in device ID */
#define ADT_ERR_RT_FT_UNDEF        1006  /*!< \brief RT Filter Table not defined */
#define ADT_ERR_RT_SA_UNDEF        1007  /*!< \brief RT Subaddress not defined */
#define ADT_ERR_RT_SA_CDP_UNDEF   1008  /*!< \brief RT SA CDP not defined */
#define ADT_ERR_IQ_NO_NEW_ENTRY    1009  /*!< \brief No new entry in interrupt queue */
#define ADT_ERR_NO_BCCB_TABLE      1010  /*!< \brief BCCB Table Pointer is zero */
#define ADT_ERR_BCCB_ALREADY_ALLOCATED 1011  /*!< \brief BCCB already allocated */
#define ADT_ERR_BCCB_NOT_ALLOCATED 1012  /*!< \brief BCCB has not been allocated */
#define ADT_ERR_BUFFER_FULL        1013  /*!< \brief 1553-ARINC PB (CDP/PCB or RXP/PXP) buf full */
#define ADT_ERR_TIMEOUT            1014  /*!< \brief Timeout error */
#define ADT_ERR_BAD_CHAN_NUM       1015  /*!< \brief Bad channel number or channel does not exist */
#define ADT_ERR_BITFAIL            1016  /*!< \brief Built-In Test failure */
#define ADT_ERR_DEVICEINUSE         1017  /*!< \brief Device in use already */
#define ADT_ERR_NO_TXCB_TABLE      1018  /*!< \brief TXCB Table Pointer is zero */
#define ADT_ERR_TXCB_ALREADY_ALLOCATED 1019  /*!< \brief TXCB already allocated */
#define ADT_ERR_TXCB_NOT_ALLOCATED 1020  /*!< \brief TXCB has not been allocated */
#define ADT_ERR_PBCB_TOOMANYPXPS   1021  /*!< \brief PBCB Too Many PXPs For PCBC Allocation */
#define ADT_ERR_NORXCHCVT_ALLOCATED 1022  /*!< \brief RX CH - No CVT Option Defined at Init */
#define ADT_ERR_NO_DATA_AVAILABLE   1023  /*!< \brief No Data Available */
```

Many more constants are defined in the ADT_L1.h header file – most of these are used internally by the API and they will not be described in detail here.

The function ADT_L1_Error_to_String can be used to convert an error/status code to a string.

Layer 1 API Functions

Each of the Layer 1 API functions is described below.

General Functions

This section describes the Layer 1 API General functions. These functions are defined in the file ADT_L1_General.c.

ADT_L1_DevicePresent

```
ADT_L0_UINT32 ADT_L1_DevicePresent (ADT_L0_UINT32 devID,  
                                     ADT_L0_UINT32 *config,  
                                     ADT_L0_UINT32 *serNum)
```

This function determines if a given device is present in the system. If the device is present it reads the configuration register and serial number for the board. This function can be used to scan for Alta devices installed in the system and can be called without first initializing a device.

This function should NOT be called on devices/channels that are already initialized and operational, because when the DevicePresent function finishes it unmaps memory (and closes the socket for ENET devices).

A recommended way to use this function is to always look for the GLOBAL device on the desired board type (rather than a 1553 channel device or A429 bank device).

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).
config – pointer to store the value of the configuration register.
serNum – pointer to store the serial number.

Returns:

ADT_SUCCESS - Device is present
ADT_FAILURE - Device is NOT present

ADT_L1_DevicePresent_pcilInfo

```
ADT_L0_UINT32 ADT_L1_DevicePresent_pcilInfo (ADT_L0_UINT32 devID,  
                                              ADT_L0_UINT32 *config,  
                                              ADT_L0_UINT32 *serNum,  
                                              ADT_L0_UINT32 *pciBus,  
                                              ADT_L0_UINT32 *pciDevice,  
                                              ADT_L0_UINT32 *pciFunc)
```

This function determines if a given device is present in the system and returns PCI information – bus/device/function. If the device is present it reads the configuration register and serial number for the board. This function can be used to scan for Alta devices installed in the system and can be called without first initializing a device.

This function should NOT be called on devices/channels that are already initialized and operational, because when the DevicePresent function finishes it unmaps memory (and closes the socket for ENET devices).

A recommended way to use this function is to always look for the GLOBAL device on the desired board type (rather than a 1553 channel device or A429 bank device).

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).
config – pointer to store the value of the configuration register.
serNum – pointer to store the serial number.
pciBus – pointer to store the PCI bus number.
pciDevice – pointer to store the PCI device number.
pciFunc – pointer to store the PCI function number.

Returns:

ADT_SUCCESS - Device is present
ADT_FAILURE - Device is NOT present

ADT_L1_InitDevice

ADT_L0_UINT32 ADT_L1_InitDevice (ADT_L0_UINT32 *devID*,
 ADT_L0_UINT32 *startupOptions*)

This function initializes a device - maps memory and initializes memory management.

Additional startup options are provided to allow the user to specify to force API initialization (overrides check for device in use, needed for previous soft application crashes), skip Memory Test (for fast startups), and perform a hard reset on the device (a 1553 channel or A429 device bank of channels – this does not reset the entire card – only the device). The ADT_L1_API_DEVICEINIT_NOKP option can be used to bypass loading the driver kernel plug-in but in most cases this option should NOT be set.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).
startupOptions - user selected startup options defined as follows:

```
#define ADT_L1_API_DEVICEINIT_FORCEINIT 0x00000001  
#define ADT_L1_API_DEVICEINIT_NOMEMTEST 0x00000002  
#define ADT_L1_API_DEVICEINIT_NOKP 0x00000004  
#define ADT_L1_API_DEVICEINIT_ROOTPERESET 0x80000000
```

The ADT_L1_API_DEVICEINIT_NOKP option can be used to bypass loading the driver kernel plug-in but in most cases this option should NOT be set. This option only applies to platforms that use the Jungs WinDriver software for the device driver (Windows, Linux, Solaris). If this option is used then the application cannot use hardware interrupts. **The kernel plug-in is required for hardware interrupts.**

The ADT_L1_API_DEVICEINIT_FORCEINIT option should ONLY be used in development and testing. This option is provided for cases where the device may not have been closed properly and is used to override the ADT_ERR_DEVICEINUSE error. This option should NOT be used as the normal initialization method for your application, because it bypasses protection against two applications using the same device.

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_BAD_CHAN_NUM - Bad channel number or channel does not exist
ADT_ERR_BITFAIL - Failed Built-In Test
ADT_ERR_DEVICEINUSE - Device in use (see explanation in the Layer 1 API Operational Discussion under “Initializing and Closing the API”)

ADT_FAILURE - Completed with error

ADT_L1_CloseDevice

ADT_L0_UINT32 ADT_L1_CloseDevice (ADT_L0_UINT32 *devID*)

This function un-maps memory and closes the device.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).

Returns:

ADT_SUCCESS - Completed without error

ADT_FAILURE - ADT_L0_UnmapMemory failed

ADT_L1_GetBoardInfo

```
ADT_L0_UINT32 ADT_L1_GetBoardInfo (ADT_L0_UINT32 devID,  
                                  ADT_L0_UINT32 * prodIDandRev,  
                                  ADT_L0_UINT32 * capabilitiesReg,  
                                  ADT_L0_UINT32 * serialNumber,  
                                  ADT_L0_UINT32 * alignCheck,  
                                  ADT_L0_UINT32 * memorySize)
```

This function gets the BOARD-LEVEL information from the global registers on the board. These are described in detail in the AltaCore 1553 Protocol Engine Specification/User's Manual. **ADT_DEVID_CHANNELTYPE_GLOBALS must be specified in devID.**

Product ID and Revision - Product ID in upper 16-bits, revision in lower 16-bits. This is represented as two Hex values, for example product ID 6000, rev 0502.

Capabilities Register - See **AltaCore** 1553 Protocol Engine Specification/User's Manual.

Serial Number - This is the board serial number. The normal representation of this displays the upper 16-bits in Hex (date code) and the lower 16-bits in decimal (serial number). For example, 0903-00123.

Align Check Word - This should always be 0x12345678. This value can be read to check for problems with byte or word swapping when porting to a new OS or platform.

Memory Size - This provides the total memory size for the board in kilobytes.

Parameters:

devID - device identifier (*Backplane, Board Type, Board #, Channel Type, Channel #*).
prodIDandRev - pointer to store the product ID and Revision.
capabilitiesReg - pointer to store the capabilities register.
serialNumber - pointer to store the serial number.
alignCheck - pointer to store the align check word.
memorySize - pointer to store the memory size.

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_UNSUPPORTED_CHANNELTYPE - Not a Global device in devID
ADT_FAILURE - Completed with error

ADT_L1_GetVersionInfo

```
ADT_L0_UINT32 ADT_L1_GetVersionInfo (ADT_L0_UINT32 devID,
                                     ADT_L0_UINT16 * peVersion,
                                     ADT_L0_UINT32 * layer0ApiVersion,
                                     ADT_L0_UINT32 * layer1ApiVersion)
```

This function gets PE (firmware version and the Layer 0 and Layer 1 API version numbers.

The PE version is a 16-bit unsigned integer interpreted as a hexadecimal version number.

For API version numbers, the returned value is a 32-bit unsigned integer. This is interpreted in octets similar to an IP address. For example, a value of 0x12345678 would be interpreted by first separating the octets (or bytes) to 0x12.0x34.0x56.0x78, then converting these octets to decimal values resulting in version 18.52.86.120.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).
peVersion - pointer to store the PE (firmware) version.
layer0ApiVersion - pointer to store the L0 API version.
layer1ApiVersion - pointer to store the L1 API version.

Returns:

ADT_SUCCESS - Completed without error
ADT_FAILURE - Completed with error

Example Code:

```
printf("Checking Protocol Engine (PE) and API versions . . . ");
status = ADT_L1_GetVersionInfo(DEVID, &peVersion, &l0Version, &l1Version);
if (status == ADT_SUCCESS)
{
    printf("Success.\n");
    printf(" PE version = %04X\n", peVersion);
    printf(" L0 API version = %.d.%d.%d.%d\n",
           (l0Version & 0xFF000000) >> 24, (l0Version & 0x00FF0000) >> 16,
           (l0Version & 0x0000FF00) >> 8, l0Version & 0x000000FF);
    printf(" L1 API version = %.d.%d.%d.%d\n",
           (l1Version & 0xFF000000) >> 24, (l1Version & 0x00FF0000) >> 16,
           (l1Version & 0x0000FF00) >> 8, l1Version & 0x000000FF);
}
else printf("FAILURE - Error = %d\n", status);
```

ADT_L1_ProgramBoardFlash

ADT_L0_UINT32 ADT_L1_ProgramBoardFlash (ADT_L0_UINT32 *devID*,
 ADT_L0_UINT32 *numBytes*,
 ADT_L0_UINT8 * *fpga_load_bytes*)

This function programs the FLASH memory for the BOARD. WARNING - This affects ALL channels on the board. Do not use this function if other applications are using other channels on the board.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).
numBytes - number of bytes to load.
fpga_load_bytes - pointer to the bytes to load.

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_BAD_DEV_TYPE - Unsupported board type
ADT_FAILURE - Completed with error

ADT_L1_ReadDeviceMem32

ADT_L0_UINT32 ADT_L1_ReadDeviceMem32 (ADT_L0_UINT32 *devID*,
 ADT_L0_UINT32 *offset*,
 ADT_L0_UINT32 * *pData*,
 ADT_L0_UINT32 *count*)

This function reads memory from a device in 32-bit words. The API determines the offset to the selected channel internally, so the memory offset to read provided by the caller is an offset in CHANNEL memory rather than BOARD memory.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).
offset - BYTE offset from the start of CHANNEL memory (not an offset from start of board memory).
pData - pointer to store the word(s) read.
count - number of 32-bit words to read. FOR ENET DEVICES THIS CANNOT BE MORE THAN 360.

Returns:

ADT_SUCCESS - Completed without error
ADT_FAILURE - Completed with error
ADT_ERR_ENET_INVALID_SIZE - Count too large (>360) for ENET device.

ADT_L1_WriteDeviceMem32

ADT_L0_UINT32 ADT_L1_WriteDeviceMem32 (ADT_L0_UINT32 *devID*,
 ADT_L0_UINT32 *offset*,
 ADT_L0_UINT32 * *pData*,
 ADT_L0_UINT32 *count*)

This function writes memory to a device in 32-bit words. The API determines the offset to the selected channel internally, so the memory offset to write provided by the caller is an offset in CHANNEL memory rather than BOARD memory.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).
offset - BYTE offset from the start of CHANNEL memory (not an offset from start of board memory).
pData - pointer to the word(s) to be written.
count - number of 32-bit words to read. FOR ENET DEVICES THIS CANNOT BE MORE THAN 360.

Returns:

ADT_SUCCESS - Completed without error
ADT_FAILURE - Completed with error
ADT_ERR_ENET_INVALID_SIZE - Count too large (>360) for ENET device.

ADT_L1_msSleep

ADT_L0_UINT32 ADT_L1_msSleep(ADT_L0_UINT32 *milliSecTick*)

This function calls the ADT_L0 operating system (OS) sleep function. Accuracy is OS dependent.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).
milliSecTick - sleep time in milliseconds.

Returns:

ADT_SUCCESS - Completed without error

ADT_L1_ENET_SetIpAddr

```
ADT_L0_UINT32 ADT_L1_ENET_SetIpAddr (ADT_L0_UINT32 devID,  
                                     ADT_L0_UINT32 ServerIpAddr,  
                                     ADT_L0_UINT32 ClientIpAddr)
```

This function is only used for ENET devices and assigns the client and server IP addresses for the ENET device. These settings apply to the entire ENET device and are assigned by the BoardType and BoardNumber fields of the device ID.

The Client IP Address is the IP address of the host computer (where the API application is running) for the appropriate adapter.

The Server IP Address is the IP address of the ENET device.

THIS FUNCTION MUST BE CALLED BEFORE ATTEMPTING TO INITIALIZE THE ENET DEVICE. This function provides the IP addresses needed to communicate with the device.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).
ServerIpAddr – 32-bit IP address of the ENET device.
ClientIpAddr – 32-bit IP address of the host system/adapter.

Returns:

ADT_SUCCESS – Completed successfully
ADT_ERR_BAD_INPUT – Not an ENET device

ADT_L1_ENET_GetIpAddr

```
ADT_L0_UINT32 ADT_L1_ENET_GetIpAddr (ADT_L0_UINT32 devID,  
                                     ADT_L0_UINT32 *pServerIpAddr,  
                                     ADT_L0_UINT32 *pClientIpAddr)
```

This function is only used for ENET devices and gets the client and server IP addresses for the ENET device, as previously assigned by a call to ADT_L1_ENET_SetIpAddr. These settings apply to the entire ENET device and are assigned by the BoardType and BoardNumber fields of the device ID.

The Client IP Address is the IP address of the host computer (where the API application is running) for the appropriate adapter.

The Server IP Address is the IP address of the ENET device.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).
pServerIpAddr – pointer to store the 32-bit IP address of the ENET device.
pClientIpAddr – pointer to store the 32-bit IP address of the host system/adapter.

Returns:

ADT_SUCCESS – Completed successfully
ADT_ERR_BAD_INPUT – Not an ENET device

ADT_L1_ENET_ADCP_Reset

```
ADT_L1_UINT32 ADT_L1_ENET_ADCP_Reset (ADT_L0_UINT32 devID)
```

This function is only used with ENET devices and transmits an ADCP RESET command to the device. This resets the entire ENET device and must use the device ID for the Global device.

Parameters:

devID - 32-bit Device Identifier.

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_UNSUPPORTED_BACKPLANE - Unsupported backplane, must be ENET device
ADT_ERR_UNSUPPORTED_CHANNEL - Unsupported channel type, must be Global device
ADT_FAILURE - Completed with error

ADT_L1_ENET_ADCP_GetStatistics

```
ADT_L0_UINT32 ADT_L1_ENET_ADCP_GetStatistics (ADT_L0_UINT32 devID,
                                              ADT_L0_UINT32 *pPortNum,
                                              ADT_L0_UINT32 *pTransactions,
                                              ADT_L0_UINT32 *pRetries,
                                              ADT_L0_UINT32 *pFailures)
```

This function is only used with ENET devices and gets statistical information on the ADCP communications with the device. This returns the transaction count, error count, and failure count. A transaction is a command packet and response packet pair (normally a memory read or write operation). A retry is anything that causes a retry (bad sequence number in response packet, bad status value in response packet, or timeout waiting for response packet). A failure is any transaction where all retries were attempted but did not succeed.

Parameters:

- devID* - 32-bit Device Identifier.
- pPortNum* - pointer to store the UDP port number for the device.
- pTransactions* - pointer to store the number of transactions.
- pRetries* - pointer to store the number of retries.
- pFailures* - pointer to store the number of failures.

Returns:

- ADT_SUCCESS** - Completed without error
- ADT_ERR_UNSUPPORTED_BACKPLANE** - Unsupported backplane, must be ENET device
- ADT_ERR_ENET_NOTRUNNING** – ENET services not running, device not initialized

ADT_L1_ENET_ADCP_ClearStatistics

```
ADT_L0_UINT32 ADT_L1_ENET_ADCP_ClearStatistics
                           (ADT_L0_UINT32 devID)
```

This function is only used with ENET devices and clears the statistical information on the ADCP communications with the device. This clears the transaction count, error count, and failure count.

Parameters:

- devID* - 32-bit Device Identifier.

Returns:

- ADT_SUCCESS** - Completed without error
- ADT_ERR_UNSUPPORTED_BACKPLANE** - Unsupported backplane, must be ENET device
- ADT_ERR_ENET_NOTRUNNING** – ENET services not running, device not initialized

ADT_L1_Error_to_String

```
char * ADT_L1_Error_to_String(ADT_L0_UINT32 err_status)
```

This function converts an error/status code to a string.

Parameters:

err_status - error/status code to convert to a string.

Returns:

String corresponding to the error/status code.

Board Global Functions

This section describes the Layer 1 API functions that use the board-level global registers. **These functions are ONLY usable for Device IDs with the channel type field set to ADT_DEVID_CHANNELTYPE_GLOBALS**. These functions are defined in the file ADT_L1_BoardGlobal.c.

ADT_L1_Global_TimeClear

ADT_L0_UINT32 ADT_L1_Global_TimeClear (ADT_L0_UINT32 devID)

This function clears the time-tag registers for all channels on the board.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).

Returns:

ADT_SUCCESS - Completed without error
ADT_BAD_INPUT - Invalid device ID
ADT_FAILURE - Completed with error

ADT_L1_Global_ConfigExtClk

This function configures the Global CSR external clock settings. If both clkSrcIn and clkSrcOut are set to NONE, this completely disables external clock input and output.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).
clkFreq must be 1, 5, or 10 (1MHz, 5MHz, or 10MHz).
clkSrcIn selects RS485, TTL, or NONE for the ext clock input (1=RS485, 2=TTL, else NONE).
clkSrcOut selects RS485, TTL, or NONE for the ext clock output (1=RS485, 2=TTL, else NONE).

Returns:

ADT_SUCCESS - Completed without error
ADT_BAD_INPUT - Invalid device ID or invalid clkFreq
ADT_FAILURE - Completed with error

ADT_L1_Global_I2C_ReadTemp

```
ADT_L0_UINT32 ADT_L1_Global_I2C_ReadTemp (ADT_L0_UINT32 devID,  
                                         ADT_L0_UINT32 temp_addr,  
                                         ADT_L0_UINT32 *temp)
```

This function reads a temperature from the I2C bus.

If the temp data MSB bit D10 = 0 then the temperature is positive and
Temp value ($^{\circ}$ C) = + (Temp data) * 0.125 $^{\circ}$ C

If the temp data MSB bit D10 = 1 then the temperature is negative and
Temp value ($^{\circ}$ C) = - (2's complement of Temp data) * 0.125 $^{\circ}$ C

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).
temp_addr - offset to the temp sensor to read (ADT_L1_GLOBAL_I2C_TS_FPGA_ADDR
or ADT_L1_GLOBAL_I2C_TS_XCVR_ADDR).
temp - pointer to store the temperature value.

Returns:

ADT_SUCCESS - Completed without error
ADT_BAD_INPUT - Invalid device ID
ADT_FAILURE - Completed with error

ADT_L1_Global_I2C_SetIrigDac

```
ADT_L0_UINT32 ADT_L1_Global_I2C_SetIrigDac (ADT_L0_UINT32 devID,  
                                              ADT_L0_UINT32 value)
```

This function writes a voltage value to the IRIG DAC on the I2C bus.

The LSB of the value represents 3.22266mV (3.3V / 1024).

To set the DAC output to 1.65V, the value should be set to 512 (0x200).

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).
value - value to set for the IRIG DAC (0 to 1023).

Returns:

ADT_SUCCESS - Completed without error
ADT_BAD_INPUT - Invalid device ID
ADT_FAILURE - Completed with error

ADT_L1_Global_I2C_SetVVdac

ADT_L0_UINT32 ADT_L1_Global_I2C_SetVVdac (ADT_L0_UINT32 *devID*,
 ADT_L0_UINT32 *channel*,
 ADT_L0_UINT32 *value*)

This function writes a voltage value to the Variable Voltage DAC for the specified 1553 or ARINC channel on the I2C bus.

The LSB of the value represents 4.8828125mV (5V / 1024) for 1553 devices, and represents 46.875mV (12V/256) for ARINC devices.

Setting 5V (1023) on the DAC selects MAXIMUM voltage output from the 1553 transceiver (about 22V peak to peak).

Setting 12V (255) on the DAC selects MAXIMUM voltage output from the ARINC transmitter (about +/- 12V differential).

Setting 3V (64) on the DAC selects MINIMUM voltage output from the ARINC transmitter (about +/- 3V differential).

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).
channel - zero-indexed 1553 or ARINC channel (0=CH1, 1=CH2, 2=CH3, 3=CH4).
value - value to set for the VV DAC (0 to 1023 for 1553 devices, 64 to 255 for ARINC devices).

Returns:

ADT_SUCCESS - Completed without error
ADT_BAD_INPUT - Invalid device ID or invalid value
ADT_FAILURE - Completed with error

ADT_L1_Global_CalibrateIrigDac

ADT_L0_UINT32 ADT_L1_Global_CalibrateIrigDac (ADT_L0_UINT32 *devID*)

This function calibrates the IRIG DAC for the input IRIG signal.

NOTE: There must be a valid IRIG signal to the board for this function to succeed. This function may take several seconds/minutes to complete depending on your machine.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).

Returns:

ADT_SUCCESS - Completed without error
ADT_BAD_INPUT - Invalid device ID
ADT_FAILURE - Completed with error

ADT_L1_Global_CalibrateIrigDacOptions

```
ADT_L0_UINT32 ADT_L1_Global_CalibrateIrigDacOptions (
    ADT_L0_UINT32 devID,
    ADT_L0_UINT32 options)
```

This function calibrates the IRIG DAC for the input IRIG signal. The non-option CalibrateIrigDac() function above adjusts the calibration in 1/1024 steps, which can take many seconds/minutes depending on the machine. This function provides an options field so the user can program the calibration value from with a 8-bit value of (1-255)/1024. The user may want to experiment with the value that is best for their system and IRIG signal to best achieve lock. Starting with an options value of 10 decimal would be best. The 8-bit calibration is located in the 8 MSB of the options word, so shift 24 bits left to set the value (CalValue << 24).

NOTE: There must be a valid IRIG signal to the board for this function to succeed. This function may take several seconds/minutes to complete depending on your machine and options value setting.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).
options - 32-bit UINT parameter where the 8-bit calibration setting is located in bits 24-31.

Returns:

ADT_SUCCESS - Completed without error
ADT_BAD_INPUT - Invalid device ID
ADT_FAILURE - Completed with error

ADT_L1_Global_ReadIrigTime

```
ADT_L0_UINT32 ADT_L1_Global_ReadIrigTime (ADT_L0_UINT32 devID,  
                                         ADT_L0_UINT32 *irigHigh,  
                                         ADT_L0_UINT32 *irigLow)
```

If the device has locked onto a valid IRIG signal this function reads the Global IRIG time registers. If IRIG LOCK is not present, ADT_FAILURE is returned.

These registers contain the following information:

IRIG Time High– 0x000000C8

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved																						BCD Years		BCD Days							

IRIG Time Low – 0x000000CC

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved								BCD Hours								BCD Minutes								BCD Seconds							

WARNING – The IRIG registers contain the time of the PREVIOUS 1-second IRIG sync, so the time in the registers is always ONE SECOND BEHIND the actual time. This function corrects for this by adding one second to the time in the IRIG time registers – this function returns the corrected time (API version 2.4.1.8 and later).

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).
irigHigh - pointer to store the IRIG high 32-bits.
irigLow - pointer to store the IRIG low 32-bits.

Returns:

ADT_SUCCESS - Completed without error
ADT_BAD_INPUT - Invalid device ID
ADT_FAILURE - Completed with error

Memory Management Functions

This section describes the Layer 1 API Memory Management functions. These functions are defined in the file ADT_L1_MemMgmt.c.

ADT_L1_InitMemMgmt

ADT_L0_UINT32 ADT_L1_InitMemMgmt (ADT_L0_UINT32 *devID*,
 ADT_L0_UINT32 startupOptions)

This function initializes a memory management for a device - initializes the internal data structures for memory management, determines the size of user memory by product type, and performs a memory test on user memory.

A user option parameter is provided to designate if ADT_L1_BIT_MemoryTest should be called. If startupOptions is zero (default), then BIT_MemoryTest will be called. Memory test takes several seconds and may not be desirable for embedded applications where a fast startup is desired.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).

startupOptions - user selected startup options defined as follows:

#define ADT_L1_API_DEVICEINIT_NOMEMTEST 0x00000002

Returns:

ADT_SUCCESS - Completed without error

ADT_ERR_MEM_MGT_INIT - Mem management already initialized for the devID

ADT_ERR_BAD_DEV_TYPE - Unknown device type in devID

ADT_L1_CloseMemMgmt

ADT_L0_UINT32 ADT_L1_CloseMemMgmt (ADT_L0_UINT32 *devID*)

This function frees resources and closes a memory management for a device.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).

Returns:

ADT_SUCCESS - Completed without error

ADT_ERR_MEM_MGT_NO_INIT - Memory management has not been initialized for the device ID

ADT_L1_GetMemoryAvailable

ADT_L0_UINT32 ADT_L1_GetMemoryAvailable (ADT_L0_UINT32 *devID*,

`ADT_L0_UINT32 * memAvailable)`

This function gets the amount of memory available (in bytes) in the data structure memory area.

Parameters:

`devID` - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).
`memAvailable` - pointer to store the number of BYTES available.

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_MEM_MGT_NO_INIT - Memory management has not been initialized for the device ID
ADT_FAILURE - Completed with error

ADT_L1_MemoryAlloc

`ADT_L0_UINT32 ADT_L1_MemoryAlloc (ADT_L0_UINT32 devID,
 ADT_L0_UINT32 memSize,
 ADT_L0_UINT32 * pMemOffset)`

This function allocates memory from the data structure memory area.

Parameters:

`devID` - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).
`memSize` - size in BYTES of memory requested.
`pMemOffset` - pointer to store the starting BYTE offset of the allocated block of memory.

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_MEM_MGT_NO_MEM - Requested memory is not available
ADT_ERR_MEM_MGT_NO_INIT - Memory management has not been initialized for the device ID

ADT_L1_MemoryFree

```
ADT_L0_UINT32 ADT_L1_MemoryFree (ADT_L0_UINT32 devID,  
                                ADT_L0_UINT32 memStart,  
                                ADT_L0_UINT32 memSize)
```

This function frees previously allocated memory in the data structure memory area.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).

memStart - starting BYTE offset of the memory block.

memSize - size in BYTES of the memory block.

Returns:

ADT_SUCCESS - Completed without error

**ADT_ERR_MEM_MGT_NO_INIT - Memory management has not been initialized for
the device ID**

ADT_FAILURE - Completed with error

BIT Functions

This section describes the Layer 1 API Built-In-Test functions. These functions are defined in the file ADT_L1_BIT.c.

ADT_L1_BIT_MemoryTest

```
ADT_L0_UINT32 ADT_L1_BIT_MemoryTest (ADT_L0_UINT32 devID,  
                                     ADT_L0_UINT32 start,  
                                     ADT_L0_UINT32 end,  
                                     ADT_L0_UINT32 * addr,  
                                     ADT_L0_UINT32 * exp,  
                                     ADT_L0_UINT32 * act)
```

This function performs a memory test on the selected range of memory. The memory is tested five times with different values (0x55555555, 0xAAAAAAA, 0xFFFFFFFF, value=offset, and 0x00000000) where first the value is written to the entire range of memory and then each word is read and checked to see if it matches the expected value. If a failure occurs, the function provides the memory offset, the expected value, and the actual value read. This test can take several seconds.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).
start - starting offset (BYTE offset)
end - ending offset (BYTE offset)
addr - pointer to store the memory offset (BYTE offset) on failure
exp - pointer to store the expected value on failure
act - pointer to store the actual value on failure

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_BAD_INPUT - Invalid device number
ADT_ERR_MEM_TEST_FAIL - Completed with error

ADT_L1_BIT_InitiatedBIT

ADT_L0_UINT32 ADT_L1_BIT_InitiatedBIT (ADT_L0_UINT32 *devID*,
 ADT_L0_UINT32 **bitStatus*)

This function performs INITIATED BIT.

Initiated BIT is safe test that may be run at any time during normal operation to verify the basic health of the device. It runs the same tests that Periodic BIT runs, but may be initiated at any time using the ADT_L1_BIT_InitiatedBIT or ADT_L1_BIT_PeriodicBIT functions.

If additional testing is desired, POST may be run after power up by resetting the device via bit 31 of the Root PE Control Word (or by re-initializing the device using ADT_L1_1553_InitDefault_ExtendedOptions with the ADT_L1_API_DEVICEINIT_ROOTPERESET option). The ADT_L1_BIT_MemoryTest function may also be called as part of a user IBIT process to test all or part of the device's memory if desired. Both of these methods include destructive memory tests that will overwrite data structures in board memory and will require the device to be re-initialized to work properly after they are run.

If this function returns ADT_FAILURE you can examine the value of the BIT Status Register to see what failed.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).
bitStatus - pointer to store the value of the BIT Status Register.

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_BAD_INPUT - Invalid device number
ADT_ERR_TIMEOUT - Timeout error
ADT_FAILURE - Completed with error

ADT_L1_BIT_PeriodicBIT

ADT_L0_UINT32 ADT_L1_BIT_PeriodicBIT (ADT_L0_UINT32 *devID*,
 ADT_L0_UINT32 **bitStatus*)

This function checks PERIODIC BIT. This can be safely run during normal operation. If this function returns ADT_FAILURE you can examine the value of the BIT Status Register to see what failed.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).
bitStatus - pointer to store the value of the BIT Status Register.

Returns:

- ADT_SUCCESS** - Completed without error
- ADT_ERR_BAD_INPUT** - Invalid device number
- ADT_FAILURE** - Completed with error

Interrupt Functions

This section describes the Layer 1 API Interrupt functions. These functions are defined in the file ADT_L1_INT.c.

ADT_L1_INT_HandlerAttach

```
ADT_L0_UINT32 ADT_L1_INT_HandlerAttach (ADT_L0_UINT32 devID,  
                                         void * pUserISR,  
                                         void * pUserData)
```

This function attaches an interrupt handler function for the device. If a pointer to user context data is provided in the pUserData parameter, this will be passed back to the user interrupt handler function when the interrupt occurs.

NOTE: AltaAPI versions prior to v2.2.1.0 did not include the pUserData parameter. This needs to be added when porting code from older API versions.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).
pUserISR - pointer to the user int handler function.
pUserData - pointer to the user context information – set to NULL if not used.

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_UNSUPPORTED_BACKPLANE - Unsupported backplane
ADT_FAILURE - Completed with error

ADT_L1_INT_HandlerDetach

```
ADT_L0_UINT32 ADT_L1_INT_HandlerDetach (ADT_L0_UINT32 devID)
```

This function detaches the interrupt handler function for the device.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).

Returns:

ADT_SUCCESS - Completed without error
ADT_FAILURE - Completed with error

1553 General Functions

This section describes the Layer 1 API 1553 General functions. These functions are defined in the file ADT_L1_1553_General.c.

ADT_L1_1553_InitDefault

```
ADT_L0_UINT32 ADT_L1_1553_InitDefault (ADT_L0_UINT32 devID,  
                                         ADT_L0_UINT32 numIQEntries)
```

This function initializes a 1553 channel, allocates interrupt queue, and sets defaults for external bus and 1553B protocol. This function calls ADT_L1_1553_InitChannel and ADT_L1_1553_SetConfig.

Also see the next function, ADT_L1_1553_InitDefault_ExtentedOptions. This function provides more options for startup of the device channel for resets and forcing API initialization, which may be useful for development and embedded applications.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).
numIQEntries - number of interrupt queue entries to allocate.

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_BAD_INPUT - Invalid number of IQ entries
ADT_ERR_BAD_CHAN_NUM - Bad channel number or channel does not exist
ADT_ERR_BITFAIL - Failed Built-In Test
ADT_ERR_DEVICEINUSE - Device in use (see explanation in the Layer 1 API
 Operational Discussion under “Initializing and Closing the API”)
ADT_FAILURE - Completed with error

ADT_L1_1553_InitDefault_ExtendedOptions

```
ADT_L0_UINT32 ADT_L1_1553_InitDefault_ExtendedOptions  
    (ADT_L0_UINT32 devID,  
     ADT_L0_UINT32 numIQEntries,  
     ADT_L0_UINT32 startupOptions)
```

This function initializes a 1553 channel, allocates interrupt queue, and sets defaults for external bus and 1553B protocol. This function calls ADT_L1_1553_InitChannel and ADT_L1_1553_SetConfig.

Additional startup options are provided to allow the user to specify to skip Force API initialization (need for previous soft application crashes), skip Memory Test (for fast startups), and perform a hard reset on the 1553 channel (not the card – this clears key channel protocol engine registers). The ADT_L1_API_DEVICEINIT_NOKP option can be used to bypass loading the driver kernel plug-in but in most cases this option should NOT be set.

This function may be preferred over ADT_L0_UINT32 ADT_L1_1553_InitDefault where the user always wants the 1553 channel to initialize regardless of previous application or hardware states.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).

numIQEntries - number of interrupt queue entries to allocate.

startupOptions - user selected startup options defined as follows:

#define	ADT_L1_API_DEVICEINIT_FORCEINIT	0x00000001
#define	ADT_L1_API_DEVICEINIT_NOMEMTEST	0x00000002
#define	ADT_L1_API_DEVICEINIT_NOKP	0x00000004
#define	ADT_L1_API_DEVICEINIT_ROOTPERESET	0x80000000

The ADT_L1_API_DEVICEINIT_FORCEINIT option should ONLY be used in development and testing. This option is provided for cases where the device may not have been closed properly and is used to override the ADT_ERR_DEVICEINUSE error. This option should NOT be used as the normal initialization method for your application, because it bypasses protection against two applications using the same device.

The ADT_L1_API_DEVICEINIT_NOMEMTEST option skips the default memory test on initialization.

The ADT_L1_API_DEVICEINIT_ROOTPERESET option resets the protocol engine and firmware to power on state upon initialization.

The ADT_L1_API_DEVICEINIT_NOKP option can be used to bypass loading the driver kernel plug-in but in most cases this option should NOT be set. This option only applies to platforms that use the Jungo WinDriver software for the device driver (Windows, Linux, Solaris). If this option is used then the application cannot use hardware interrupts. The kernel plug-in is required for hardware interrupts.

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_BAD_INPUT - Invalid number of IQ entries
ADT_ERR_BAD_CHAN_NUM - Bad channel number or channel does not exist
ADT_ERR_BITFAIL - Failed Built-In Test
ADT_FAILURE - Completed with error

ADT_L1_1553_GetConfig

```
ADT_L0_UINT32 ADT_L1_1553_GetConfig (ADT_L0_UINT32 devID,  
                                     ADT_L0_UINT32 * pIsMultRT,  
                                     ADT_L0_UINT32 * pIs1553B,  
                                     ADT_L0_UINT32 * pIsRt31Bcast,  
                                     ADT_L0_UINT32 * pUseIntBus,  
                                     ADT_L0_UINT32 * pStsRespTimeout_us)
```

This function gets the 1553 options for a channel. These options are:

Multiple RT - 1 for multiple RT, 0 for single RT.

1553B Protocol - 1 for 1553B protocol, 0 for 1553A protocol.

RT31 Broadcast - 1 if RT31 is broadcast, 0 if RT31 is a normal RT address.

Use Internal Bus - 1 to use internal bus, 0 to use external bus.

Status Response Timeout - time (in microseconds) that the BC and BM will allow for an RT to respond with a status word before declaring a "no-response" condition.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).

pIsMultRT - pointer to store the MRT setting.

pIs1553B - pointer to store the protocol setting.

pIsRt31Bcast - pointer to store the RT31 setting.

pUseIntBus - pointer to store the internal bus setting.

pStsRespTimeout_us - pointer to store the timeout setting.

Returns:

ADT_SUCCESS - Completed without error

ADT_FAILURE - Completed with error

ADT_L1_1553_GetPEInfo

ADT_L0_UINT32 ADT_L1_1553_GetPEInfo (ADT_L0_UINT32 *devID*,
 ADT_L0_UINT32 * *peInfo*)

This function gets 1553 PE ID and Version information. Also See:
[ADT_L1_GetVersionInfo\(\)](#)

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).
peInfo - pointer to store the PE ID and Version.

Returns:

ADT_SUCCESS - Completed without error
ADT_FAILURE - Completed with error

ADT_L1_1553_InitChannel

ADT_L0_UINT32 ADT_L1_1553_InitChannel (ADT_L0_UINT32 *devID*,
 ADT_L0_UINT32 *numIQEntries*)

This function initializes a 1553 channel and allocates interrupt queue.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).
numIQEntries - number of interrupt queue entries to allocate.

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_BAD_INPUT - Invalid number of IQ entries
ADT_FAILURE - Completed with error

ADT_L1_1553_InitChannelLive

ADT_L0_UINT32 ADT_L1_1553_InitChannelLive (ADT_L0_UINT32 *devID*,
 ADT_L0_UINT32 *numIQEntries*)

This function initializes a 1553 channel and allocates interrupt queue, but does NOT clear the RT control blocks. Use this function instead of ADT_L1_1553_InitChannel if the channel is configured to come up “live” as a Remote Terminal (using the external RT address lines).

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).
numIQEntries - number of interrupt queue entries to allocate.

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_BAD_INPUT - Invalid number of IQ entries
ADT_FAILURE - Completed with error

ADT_L1_1553_SetConfig

```
ADT_L0_UINT32 ADT_L1_1553_SetConfig (ADT_L0_UINT32 devID,  
                                     ADT_L0_UINT32 isMultRT,  
                                     ADT_L0_UINT32 is1553B,  
                                     ADT_L0_UINT32 isRt31Bcast,  
                                     ADT_L0_UINT32 useIntBus,  
                                     ADT_L0_UINT32 stsRespTimeout_us)
```

This function sets the 1553 options for a channel. These options are:

Multiple RT - 1 for multiple RT, 0 for single RT.

1553B Protocol - 1 for 1553B protocol, 0 for 1553A protocol.

RT31 Broadcast - 1 if RT31 is broadcast, 0 if RT31 is a normal RT address.

Use Internal Bus - 1 to use internal bus, 0 to use external bus.

Status Response Timeout - time (in microseconds) that the BC and BM will allow for an RT to respond with a status word before declaring a "no-response" condition.

NOTE – Broadcast (RT31) messages are handled differently for Single-RT mode and Multiple-RT (default) mode. In Single-RT mode the Broadcast messages will go to the same CDP buffer that would be used for non-broadcast messages. In Multiple-RT mode you must setup RT31 with the desired subaddress buffers to receive Broadcast messages when operating as an RT.

The example program ADT_L1_1553_ex_rt2_bcast_srt.c demonstrates this for Single-RT mode.

The example program ADT_L1_1553_ex_rt2_bcast_mrt.c demonstrates this for Multiple-RT mode.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).
isMultRT - set to 1 to enable multiple RT mode (ZERO = Single RT).
is1553B - set to 1 for 1553B handling of mode codes (ZERO = 1553A).
isRt31Bcast - set to 1 for RT31 to be BROADCAST (ZERO = RT31 is a valid RT address).
useIntBus - set to 1 to use INTERNAL bus (ZERO = External bus).
stsRespTimeout_us - BC/BM status response timeout in microseconds.

Returns:

ADT_SUCCESS - Completed without error
ADT_FAILURE - Completed with error

ADT_L1_1553_TimeClear

ADT_L0_UINT32 ADT_L1_1553_TimeClear (ADT_L0_UINT32 *devID*)

This function clears the time tag value for the channel.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_TIMEOUT - Timeout waiting for CSR bit to clear
ADT_FAILURE - Completed with error

ADT_L1_1553_TimeGet

ADT_L0_UINT32 ADT_L1_1553_TimeGet (ADT_L0_UINT32 *devID*,
 ADT_L0_UINT32 * *pTimeHigh*,
 ADT_L0_UINT32 * *pTimeLow*)

This function gets the time tag value for the channel.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).
pTimeHigh - pointer to store the upper 32-bits of the 64-bit time tag value.
pTimeLow - pointer to store the lower 32-bits of the 64-bit time tag value.

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_TIMEOUT - Timeout waiting for CSR bit to clear
ADT_FAILURE - Completed with error

ADT_L1_1553_TimeSet

ADT_L0_UINT32 ADT_L1_1553_TimeSet (ADT_L0_UINT32 *devID*,
 ADT_L0_UINT32 *timeHigh*,
 ADT_L0_UINT32 *timeLow*)

This function sets the time tag value for the channel.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).
timeHigh - upper 32-bits of the 64-bit time tag value.
timeLow - lower 32-bits of the 64-bit time tag value.

Returns:

ADT_SUCCESS - Completed without error
ADT_FAILURE - Completed with error

ADT_L1_1553_IrigLatchedTimeGet

```
ADT_L0_UINT32 ADT_L1_1553_IrigLatchedTimeGet (ADT_L0_UINT32 devID,  
                                              ADT_L0_UINT32 * pIrigTimeHigh, ADT_L0_UINT32 * pIrigTimeLow,  
                                              ADT_L0_UINT32 * pIntTimeHigh, ADT_L0_UINT32 * pIntTimeLow,  
                                              ADT_L0_UINT32 * pDeltaTimeHigh, ADT_L0_UINT32 * pDeltaTimeLow)
```

If the device has locked onto a valid IRIG signal, this function gets the latched IRIG and internal time value for the channel. If IRIG LOCK is not present, ADT_FAILURE is returned. These values are latched by the firmware on every 1 second IRIG sync. Use the ADT_L1_Global_CalibrateIrigDac function to calibrate to the IRIG signal.

The latched IRIG time is in IRIG BCD format, as shown below:

IRIG Time High—0x000000C8																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved										BCD Years										BCD Days											

IRIG Time Low – 0x000000CC																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved										BCD Hours										BCD Minutes				BCD Seconds							

NOTE:

IRIG Standard 200-98 – The IRIG-B time code does not include years.

IRIG Standard 200-04 – The IRIG-B time code can include years.

The latched internal time is in 64-bit binary format (not BCD), with a 20ns LSB. The delta time is the difference between the IRIG time (converted from BCD to binary) and the internal time. Add the delta time to internal time values (like the time stamp on a CDP) to convert them to binary IRIG time.

Parameters:

- devID* - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).
- pIrigTimeHigh* - pointer to store the upper 32-bits of the 64-bit IRIG BCD time (latched).
- pIrigTimeLow* - pointer to store the lower 32-bits of the 64-bit IRIG BCD time (latched).
- pIntTimeHigh* - pointer to store the upper 32-bits of the 64-bit internal time (latched).
- pIntTimeLow* - pointer to store the lower 32-bits of the 64-bit internal time (latched).
- pDeltaTimeHigh* - pointer to store the upper 32-bits of the 64-bit delta time.
- pDeltaTimeLow* - pointer to store the lower 32-bits of the 64-bit delta time.

Returns:

- ADT_SUCCESS** - Completed without error
- ADT_ERR_TIMEOUT** - Timeout waiting for CSR bit to clear
- ADT_FAILURE** - Completed with error

ADT_L1_1553_PBTimeGet

```
ADT_L0_UINT32 ADT_L1_1553_PBTimeGet (ADT_L0_UINT32 devID,  
                                     ADT_L0_UINT32 * pTimeHigh,  
                                     ADT_L0_UINT32 * pTimeLow)
```

This function gets the Playback time tag value for the channel.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).
pTimeHigh - pointer to store the upper 32-bits of the 64-bit time tag value.
pTimeLow - pointer to store the lower 32-bits of the 64-bit time tag value.

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_TIMEOUT - Timeout waiting for CSR bit to clear
ADT_FAILURE - Completed with error

ADT_L1_1553_PBTimeSet

```
ADT_L0_UINT32 ADT_L1_1553_TimeSet (ADT_L0_UINT32 devID,  
                                     ADT_L0_UINT32 timeHigh,  
                                     ADT_L0_UINT32 timeLow)
```

This function sets the Playback time tag value for the channel.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).
timeHigh - upper 32-bits of the 64-bit time tag value.
timeLow - lower 32-bits of the 64-bit time tag value.

Returns:

ADT_SUCCESS - Completed without error
ADT_FAILURE - Completed with error

ADT_L1_1553_UseExtClk

ADT_L0_UINT32 ADT_L1_1553_UseExtClk (ADT_L0_UINT32 *devID*,
 ADT_L0_UINT32 *useExtClk*,
 ADT_L0_UINT32 *clkFreq*)

This function enables the channel to use an external clock.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).
useExtClk - enables or disables external clock (1=enable, 0=disable).
clkFreq - must be 1, 5, or 10 (1MHz, 5MHz, or 10MHz).

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_BAD_INPUT - Invalid value for *clkFreq*
ADT_FAILURE - Completed with error

ADT_L1_1553_ForceTrgOut

ADT_L0_UINT32 ADT_L1_1553_ForceTrgOut (ADT_L0_UINT32 *devID*)

This function causes the channel to generate an output trigger.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).

Returns:

ADT_SUCCESS - Completed without error
ADT_FAILURE - Completed with error

ADT_L1_1553_SC_ArmTrigger

ADT_L0_UINT32 ADT_L1_1553_SC_ArmTrigger (ADT_L0_UINT32 *devID*,
 ADT_L0_UINT32 *bus*,
 ADT_L0_UINT32 *trigger*,
 ADT_L0_UINT32 *mask_value*)

This function arms the Signal Capture for a trigger.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).
bus - identifies which bus to use (zero for bus A, non-zero for bus B).
trigger - value to write to the Signal Capture CSR.
mask_value - value to write to the Signal Capture Mask\Value Register.

Returns:

ADT_SUCCESS - Completed without error
ADT_FAILURE - Completed with error

ADT_L1_1553_SC_ReadBuffer

ADT_L0_UINT32 ADT_L1_1553_SC_ReadBuffer (ADT_L0_UINT32 *devID*,
 ADT_L0_UINT32 *bus*,
 ADT_L0_UINT8 **buffer*)

This function reads the Signal Capture data buffer for the specified bus.

This reads 2048 unsigned 8-bit values.

Each value represents a voltage where **0 is -15 volts** and **255 is +15 volts**. Therefore the voltage range is 30 volts over 256 possible values, or 0.117 volts per LSB.

Each of the 2048 values represents a time period of 50 nanoseconds. Therefore the entire buffer represents 102.4 microseconds of time.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).
bus - identifies which bus to use (zero for bus A, non-zero for bus B).
buffer - array of 2048 unsigned 8-bit values.

Returns:

ADT_SUCCESS - Completed without error
ADT_FAILURE - Completed with error

ADT_L1_1553_CDP_Calculate_1760_Checksum

```
ADT_L0_UINT32 ADT_L1_1553_CDP_Calculate_1760_Checksum  
    (ADT_L1_1553_CDP *pCdp,  
     ADT_L0_UINT32 wordCount,  
     ADT_L0_UINT16 *pChecksum)
```

This function calculates the MIL-STD-1760 checksum for a CDP buffer. Refer to MIL-STD-1760C, Appendix B, section B.4.1.5.2.1 for a description of the MIL-STD-1760 checksum algorithm.

The user passes in a pointer to the CDP structure containing the data words, a word count value, and a pointer to store the 16-bit checksum result. The word count parameter is the number of data words to include in the checksum calculation (which will be the message word count minus 1 because the checksum value will go in the last data word).

For example, a four word message will use the fourth data word to contain the checksum, so the first three data words will be used to calculate the checksum. If the first three data words are 0x0001, 0xC000, and 0x0F00, we will pass the CDP containing these data words to this function with a wordCount parameter of 3. The function will calculate the checksum and pass it back in the pChecksum parameter. For the three data words we have specified, the resulting checksum should be 0x1E0B. The user can now put this value in the fourth data word and write the CDP to device memory for transmission.

Parameters:

pCdp is a pointer to the CDP structure.
wordCount - number of data words (1 to 31) to include in the checksum.
pChecksum - pointer to store the checksum value.

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_BAD_INPUT - Invalid word count or null pointer
ADT_FAILURE - Completed with error

ADT_L1_1553_IntervalTimerGet

```
ADT_L0_UINT32 ADT_L1_1553_IntervalTimerGet(ADT_L0_UINT32 devID,  
                                         ADT_L0_UINT32 *pIntvltmrReg);
```

This function retrieves the Root PE Interval Timer Register. The register is comprised of 24 MSB, 1 usec time value and 5 control and status bits in the 8 LSB. The following provides the ADT_L1.h definitions:

```
#define ADT_L1_1553_PE_INTVLTMR          0x004C  
#define ADT_L1_1553_PE_INTVLTMR_STARTSTOP 0x00000001  
#define ADT_L1_1553_PE_INTVLTMR_STARTONTRIG 0x00000002  
#define ADT_L1_1553_PE_INTVLTMR_EXTTRIG   0x00000008  
#define ADT_L1_1553_PE_INTVLTMR_SETINT    0x00000010  
#define ADT_L1_1553_PE_INTVLTMR_TIMECOMP  0x00000080  
#define ADT_L1_1553_PE_INTVLTMR_TIME24    0xFFFFFFF00  
#define ADT_L1_1553_PE_INTVLTMR_TIMESHIFT 8
```

Also see the AltaCore-1553 manual for more detail definitions and actions of the register bits.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).
pIntvltmrReg - pointer to the register value result.

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_BAD_INPUT - Not a 1553 device
ADT_FAILURE - Completed with error

ADT_L1_1553_IntervalTimerSet

```
ADT_L0_UINT32 ADT_L1_1553_IntervalTimerGet(ADT_L0_UINT32 devID,
                                            ADT_L0_UINT32 intvltmrReg);
```

This function writes a user value to the Root PE Interval Timer Register. The register is comprised of 24 MSB, 1 usec time value and 5 control and status bits in the 8 LSB. The following provides the ADT_L1.h definitions:

```
#define ADT_L1_1553_PE_INTVLTMR          0x004C
#define ADT_L1_1553_PE_INTVLTMR_STARTSTOP  0x00000001
#define ADT_L1_1553_PE_INTVLTMR_STARTONTRIG 0x00000002
#define ADT_L1_1553_PE_INTVLTMR_EXTTRIG    0x00000008
#define ADT_L1_1553_PE_INTVLTMR_SETINT     0x00000010
#define ADT_L1_1553_PE_INTVLTMR_TIMECOMP   0x00000080
#define ADT_L1_1553_PE_INTVLTMR_TIME24     0xFFFFFFF00
#define ADT_L1_1553_PE_INTVLTMR_TIMESHIFT  8
```

Also see the AltaCore-1553 manual for more detail definitions and actions of the register bits.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).
intvltmrReg - user value to write to the Interval Register.

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_BAD_INPUT - Not a 1553 device
ADT_FAILURE - Completed with error

1553 Interrupt Functions

This section describes the Layer 1 API 1553 interrupt functions. These functions are defined in the file ADT_L1_1553_INT.c.

ADT_L1_1553_INT_CheckChannelIntPending

ADT_L0_UINT32 ADT_L1_1553_INT_CheckChannelIntPending
(ADT_L0_UINT32 *devID*,
ADT_L0_UINT32 * *pIsIntPending*)

This function checks to see if an interrupt is pending for a given 1553 channel.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).
pIsIntPending - pointer to store the result (0 = no int pending, 1 = int pending).

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_BAD_INPUT - Invalid channel number
ADT_FAILURE - Completed with error

ADT_L1_1553_INT_DisableInt

ADT_L0_UINT32 ADT_L1_1553_INT_DisableInt (ADT_L0_UINT32 *devID*)

This function disables interrupts for the channel.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).

Returns:

ADT_SUCCESS - Completed without error
ADT_FAILURE - Completed with error

ADT_L1_1553_INT_EnableInt

ADT_L0_UINT32 ADT_L1_1553_INT_EnableInt (ADT_L0_UINT32 *devID*)

This function enables interrupts for the channel.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).

Returns:

ADT_SUCCESS - Completed without error

ADT_FAILURE - Completed with error

ADT_L1_1553_INT_GenInt

ADT_L0_UINT32 ADT_L1_1553_INT_GenInt (ADT_L0_UINT32 *devID*)

This function generates a test interrupt. This does NOT put an entry in the interrupt queue. This function is useful for testing interrupts when developing a device driver for a new operating system.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).

Returns:

ADT_SUCCESS - Completed without error

ADT_FAILURE - Completed with error

ADT_L1_1553_INT_GetIntSeqNum

ADT_L0_UINT32 ADT_L1_1553_INT_GetIntSeqNum (ADT_L0_UINT32 *devID*,
ADT_L0_UINT32 * *pSeqNum*)

This function reads the interrupt sequence number for the channel.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).

pSeqNum - pointer to store the sequence number.

Returns:

ADT_SUCCESS - Completed without error

ADT_FAILURE - Completed with error

ADT_L1_1553_INT_IQ_ReadEntry

ADT_L0_UINT32 ADT_L1_1553_INT_IQ_ReadEntry (ADT_L0_UINT32 *devID*,

```
ADT_L0_UINT32 * pType,  
ADT_L0_UINT32 * pInfo)
```

This function reads one new entry from the interrupt queue. The pType word is the “IQP Type and Sequence Number” word from the Interrupt Queue Packet, where the upper 16-bits provide the interrupt type and the lower 16-bits provide a sequence number. The pInfo word is one of the following:

For CDP interrupts, this is the “API Info Word” from the CDP that caused the interrupt.

For BCCB interrupts, this is the “API Message Number” word from the BCCB that caused the interrupt.

For PB and SG interrupts, this is the memory address (byte offset) to the PBCB or SGCB that caused the interrupt.

For more information on interrupt queue entries refer to the “Layer 1 API Operational Discussion” section, under “Interrupt Operation” in the section for “1553 Device Interrupt Functions”.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).

pType - pointer to store the IQ type/seqnum.

pInfo - pointer to store the CDP API INFO word, BCCB message number, PBCB pointer, or SGCB pointer.

Returns:

ADT_SUCCESS - Completed without error

ADT_FAILURE - Completed with error

ADT_ERR_IQ_NO_NEW_ENTRY - No new entry in the queue

ADT_L1_1553_INT_IQ_ReadNewEntries

```
ADT_L0_UINT32 ADT_L1_1553_INT_IQ_ReadNewEntries  
    (ADT_L0_UINT32 devID,  
     ADT_L0_UINT32 MaxNumEntries,  
     ADT_L0_UINT32 * pNumEntries,  
     ADT_L0_UINT32 * pType,  
     ADT_L0_UINT32 * pInfo)
```

This function reads all new entries from the interrupt queue. . The pType word is the “IQP Type and Sequence Number” word from the Interrupt Queue Packet, where the upper 16-bits provide the interrupt type and the lower 16-bits provide a sequence number. The pInfo word is one of the following:

For CDP interrupts, this is the “API Info Word” from the CDP that caused the interrupt.

For BCCB interrupts, this is the “API Message Number” word from the BCCB that caused the interrupt.

For PB and SG interrupts, this is the memory address (byte offset) to the PBCB or SGCB that caused the interrupt.

For more information on interrupt queue entries refer to the “Layer 1 API Operational Discussion” section, under “Interrupt Operation” in the section for “1553 Device Interrupt Functions”.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).
maxNumEntries - maximum number of entries to read (size of buffer).
pNumEntries - pointer to store the number of messages read.
pType - pointer to store the IQ type/seqnums (array sized by maxNumEntries).
pInfo - pointer to store the CDP API INFO words (array sized by maxNumEntries).

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_BAD_INPUT - invalid pointer
ADT_FAILURE - Completed with error

ADT_L1_1553_INT_IQ_ReadRawEntry

```
ADT_L0_UINT32 ADT_L1_1553_INT_IQ_ReadEntry (ADT_L0_UINT32 devID,  
                                         ADT_L1_1553_INT *intbuffer)
```

This function reads one new raw entry from the interrupt queue. ADT_L1_1553_INT has the following structure:

```
typedef struct adt_l1_1553_int {  
    ADT_L0_UINT32 NextPtr;  
    ADT_L0_UINT32 Type_SeqNum;  
    ADT_L0_UINT32 IntData;  
} ADT_L1_1553_INT;
```

NextPtr contains a pointer to the next interrupt queue entry. Type_SeqNum is the “IQP Type and Sequence Number” word from the Interrupt Queue Packet, where the upper 16-bits provide the interrupt type and the lower 16-bits provide a sequence number. IntData is a pointer to the data structure that caused the interrupt.

- For CDP interrupts, IntData points to the CDP that caused the interrupt.
- For BCCB interrupts, IntData points to the BCCB that caused the interrupt.
- For PB and SG interrupts, IntData points to the PBCB or SGCB that caused the interrupt.

For more information on interrupt queue entries refer to the “Layer 1 API Operational Discussion” section, under “Interrupt Operation” in the section for “1553 Device Interrupt Functions”.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).
int_buffer - pointer to the ADT_L1_1553_INT data buffer type where the interrupt info is stored.

Returns:

ADT_SUCCESS - Completed without error
ADT_FAILURE - Completed with error
ADT_ERR_IQ_NO_NEW_ENTRY - No new entry in the queue

ADT_L1_1553_INT_IQ_ReadNewRawEntries

```
ADT_L0_UINT32 ADT_L1_1553_INT_IQ_ReadNewEntries  
    (ADT_L0_UINT32 devID,  
     ADT_L0_UINT32 MaxNumEntries,  
     ADT_L0_UINT32 * pNumEntries,  
     ADT_L1_1553_INT *intbuffer)
```

This function reads all new entries from the interrupt queue into an array of type ADT_L1_1553_INT. ADT_L1_1553_INT has the following structure:

```
typedef struct adt_l1_1553_int {  
    ADT_L0_UINT32 NextPtr;  
    ADT_L0_UINT32 Type_SeqNum;  
    ADT_L0_UINT32 IntData;  
} ADT_L1_1553_INT;
```

NextPtr contains a pointer to the next interrupt queue entry. Type_SeqNum is the “IQP Type and Sequence Number” word from the Interrupt Queue Packet, where the upper 16-bits provide the interrupt type and the lower 16-bits provide a sequence number. IntData is a pointer to the data structure that caused the interrupt.

- For CDP interrupts, IntData points to the CDP that caused the interrupt.
- For BCCB interrupts, IntData points to the BCCB that caused the interrupt.
- For PB and SG interrupts, IntData points to the PBCB or SGCB that caused the interrupt.

For more information on interrupt queue entries refer to the “Layer 1 API Operational Discussion” section, under “Interrupt Operation” in the section for “1553 Device Interrupt Functions”.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).
maxNumEntries - maximum number of entries to read (size of buffer).
pNumEntries - pointer to store the number of messages read.
intbuffer - pointer to store the raw interrupt data array (array sized by maxNumEntries).

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_BAD_INPUT - invalid pointer
ADT_FAILURE - Completed with error

ADT_L1_1553_INT_SetIntSeqNum

ADT_L0_UINT32 ADT_L1_1553_INT_SetIntSeqNum (ADT_L0_UINT32 *devID*,
ADT_L0_UINT32 *seqNum*)

This function writes the interrupt sequence number for the channel.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).
seqNum - sequence number to write.

Returns:

ADT_SUCCESS - Completed without error
ADT_FAILURE - Completed with error

1553 Bus Monitor Functions

This section describes the Layer 1 API 1553 Bus Monitor functions. These functions are defined in the file ADT_L1_1553_BM.c.

ADT_L1_1553_BM_BufferCreate

ADT_L0_UINT32 ADT_L1_1553_BM_BufferCreate (ADT_L0_UINT32 *devID*,
 ADT_L0_UINT32 *numMsgs*)

This function allocates and links the requested number of BM buffers.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).
numMsgs - number of BM CDP buffers requested.

Returns:

ADT_SUCCESS - Completed without error
ADT_FAILURE - Completed with error

ADT_L1_1553_BM_BufferFree

ADT_L0_UINT32 ADT_L1_1553_BM_BufferFree (ADT_L0_UINT32 *devID*)

This function frees all board memory used for BM buffers.

WARNING - THIS COMPLETELY UN-INITIALIZES THE BM.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).

Returns:

ADT_SUCCESS - Completed without error
ADT_FAILURE - Completed with error

ADT_L1_1553_BM_CDPRead

```
ADT_L0_UINT32 ADT_L1_1553_BM_CDPRead (ADT_L0_UINT32 devID,  
                                     ADT_L0_UINT32 cdpNum,  
                                     ADT_L1_1553_CDP * pCdp)
```

This function reads a CDP for the BM.

NOTE: If you read a CDP buffer while the firmware is in the middle of processing a message for that buffer, then the CDP Status Word will be 0xFFFFFFFF. If you see this value, then you should read the buffer again until the CDP Status Word is NOT 0xFFFFFFFF – you will then have a complete CDP buffer.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).
cdpNum - CDP number.
pCdp - pointer to store the CDP structure.

Returns:

ADT_SUCCESS - Completed without error
ADT_FAILURE - Completed with error

ADT_L1_1553_BM_CDPWrite

```
ADT_L0_UINT32 ADT_L1_1553_BM_CDPWrite (ADT_L0_UINT32 devID,  
                                         ADT_L0_UINT32 cdpNum,  
                                         ADT_L1_1553_CDP * pCdp)
```

This function writes a CDP for the BM.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).
cdpNum - CDP number.
pCdp - pointer to the CDP structure.

Returns:

ADT_SUCCESS - Completed without error
ADT_FAILURE - Completed with error

ADT L1 1553 BM Clear

ADT_L0_UINT32 ADT_L1_1553_BM_Clear (ADT_L0_UINT32 devID)

This function clears BM message counter for the device. This will cause the BM to reset to zero for the message numbers used in the “BM Count” field of the CDP for each message.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_BAD_INPUT - Invalid device number
ADT_FAILURE - Completed with error

APT L1 1553 BM Config

This function configures BM options for the device by setting/clearing bits in the Root BM CSR. The *options* parameter can be used to set or clear any of the following Root BM CSR bits:

ADT_L1_1553_BM_CSR_MSGTRGON	0x00000002
ADT_L1_1553_BM_CSR_WAITEXTTRG	0x00000004
ADT_L1_1553_BM_CSR_STORESPURDATA	0x00000008
ADT_L1_1553_BM_CSR_INTVLTMRSSVD3	0x00000010
ADT_L1_1553_BM_CSR_FILTER_RT_RT	0x00000020
ADT_L1_1553_BM_CSR_ENET_APMP_ENABLE	0x00000100
ADT_L1_1553_BM_CSR_ENET_APMP_PEIRIG	0x00000200
ADT_L1_1553_BM_CSR_ENET_APMP_PIENTV	0x00000400

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).
options - value to write to the Root BM CSR.

Returns:

ADT_SUCCESS - Completed without error
ADT_FAILURE - Completed with error

ADT_L1_1553_BM_FilterRead

```
ADT_L0_UINT32 ADT_L1_1553_BM_FilterRead (ADT_L0_UINT32 devID,  
                                         ADT_L0_UINT32 rtAddr,  
                                         ADT_L0_UINT32 * pRxFilters,  
                                         ADT_L0_UINT32 * pTxFilters)
```

This function reads BM filter settings for the device. Each bit in a filter word (rx filters or tx filters) corresponds to a subaddress. Bit 0 is SA 0, bit 1 is SA 1, etc. If the bit is set then the BM will capture messages for that subaddress.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).
rtAddr - RT address to apply the filter for.
pRxFilters - pointer to the filter word for RECEIVE subaddresses.
pTxFilters - pointer to the filter word for TRANSMIT subaddresses.

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_BAD_INPUT - Invalid RT address
ADT_FAILURE - Completed with error

ADT_L1_1553_BM_FilterWrite

```
ADT_L0_UINT32 ADT_L1_1553_BM_FilterWrite (ADT_L0_UINT32 devID,  
                                         ADT_L0_UINT32 rtAddr,  
                                         ADT_L0_UINT32 rxFilters,  
                                         ADT_L0_UINT32 txFilters)
```

This function writes BM filter settings for the device. Each bit in a filter word (rx filters or tx filters) corresponds to a subaddress. Bit 0 is SA 0, bit 1 is SA 1, etc. If the bit is set then the BM will capture messages for that subaddress.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).
rtAddr - RT address to apply the filter for.
rxFilters - filter word for RECEIVE subaddresses.
txFilters - filter word for TRANSMIT subaddresses.

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_BAD_INPUT - Invalid RT address
ADT_FAILURE - Completed with error

ADT_L1_1553_BM_ReadNewMsgs

ADT_L0_UINT32 ADT_L1_1553_BM_ReadNewMsgs (ADT_L0_UINT32 *devID*,
 ADT_L0_UINT32 *maxNumMsgs*,
 ADT_L0_UINT32 * *pNumMsgs*,
 ADT_L1_1553_CDP * *pMsgBuffer*)

This function reads all new messages from the BM buffer.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).
maxNumMsgs - maximum number of messages to read (size of buffer).
pNumMsgs - pointer to store the number of messages read.
pMsgBuffer - pointer to store the message CDP records.

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_BAD_INPUT - invalid pointer
ADT_FAILURE - Completed with error

ADT_L1_1553_BM_ReadNewMsgsDMA

ADT_L0_UINT32 ADT_L1_1553_BM_ReadNewMsgsDMA
 (ADT_L0_UINT32 *devID*,
 ADT_L0_UINT32 *maxNumMsgs*,
 ADT_L0_UINT32 * *pNumMsgs*,
 ADT_L1_1553_CDP * *pMsgBuffer*)

This function reads all new messages from the BM buffer using DMA.

NOTE THAT ONLY SELECTED LAYER 0 SUPPORT DMA (currently Windows, VxWorks, Linux) AND ONLY SELECTED BOARD TYPES SUPPORT DMA.

PCI devices with PLX9056 – PMC, PC104P, PCI

NOT supported on PCCD products

PCIE devices with 1553 PE version 0707 or later

This function is not multiple-application/thread safe. This function should only be used if one thread controls all channels on a given board that are using DMA.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).
maxNumMsgs - maximum number of messages to read (size of buffer).
pNumMsgs - pointer to store the number of messages read.
pMsgBuffer - pointer to store the message CDP records.

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_BAD_INPUT - invalid pointer
ADT_FAILURE - Completed with error

ADT_L1_1553_BM_Start

ADT_L0_UINT32 ADT_L1_1553_BM_Start (ADT_L0_UINT32 *devID*)

This function starts BM operation for the device.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).

Returns:

ADT_SUCCESS - Completed without error

ADT_FAILURE - Completed with error

ADT_L1_1553_BM_Stop

ADT_L0_UINT32 ADT_L1_1553_BM_Stop (ADT_L0_UINT32 *devID*)

This function stops BM operation for the device.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).

Returns:

ADT_SUCCESS - Completed without error

ADT_ERR_BAD_INPUT - Invalid device number

ADT_FAILURE - Completed with error

1553 Remote Terminal Functions

This section describes the Layer 1 API 1553 Remote Terminal functions. These functions are defined in the file ADT_L1_1553_RT.c.

ADT_L1_1553_RT_Close

ADT_L0_UINT32 ADT_L1_1553_RT_Close (ADT_L0_UINT32 *devID*,
 ADT_L0_UINT32 *rtAddr*)

This function un-initializes data structures and frees memory for the RT.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).
rtAddr - RT address.

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_BAD_INPUT - Invalid RT address
ADT_FAILURE - Completed with error

ADT_L1_1553_RT_Disable

ADT_L0_UINT32 ADT_L1_1553_RT_Disable (ADT_L0_UINT32 *devID*,
 ADT_L0_UINT32 *rtAddr*)

This function disables a specific RT. Delays 1ms to allow the firmware to finish processing any currently executing message for the RT.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).
rtAddr - RT address.

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_BAD_INPUT - Invalid RT address
ADT_FAILURE - Completed with error

ADT_L1_1553_RT_Enable

This function enables a specific RT.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).
rtAddr - RT address.

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_BAD_INPUT - Invalid RT address
ADT_FAILURE - Completed with error

ADT L1 1553 RT GetExternalRTAddr

`ADT_L0_UINT32 ADT_L1_1553_RT_GetExternalRTAddr
 (ADT_L0_UINT32 devID,
 ADT_L0_UINT32 * pRtAddr)`

This function gets the external RT address for the device. Note that there must be a valid external RT address set, with correct parity, and the external RT address enable signal must be low (GND). Refer to the hardware manual for your specific board type for information on these signals and pin-outs.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).
pRtAddr - pointer to store the external RT address.

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_BAD_INPUT - Invalid parity on external RT address
ADT_FAILURE - Completed with error

ADT_L1_1553_RT_GetLastCmd

```
ADT_L0_UINT32 ADT_L1_1553_RT_GetLastCmd (ADT_L0_UINT32 devID,  
                                         ADT_L0_UINT32 rtAddr,  
                                         ADT_L0_UINT32 * pLastCmd)
```

This function gets the last command seen by the RT.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).
rtAddr - RT address.
pLastCmd - pointer to store the last command word.

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_BAD_INPUT - Invalid RT address
ADT_FAILURE - Completed with error

ADT_L1_1553_RT_GetOptions

```
ADT_L0_UINT32 ADT_L1_1553_RT_GetOptions (ADT_L0_UINT32 devID,  
                                         ADT_L0_UINT32 rtAddr,  
                                         ADT_L0_UINT32 * pAllowDBC,  
                                         ADT_L0_UINT32 * pTxInh_A,  
                                         ADT_L0_UINT32 * pTxInh_B,  
                                         ADT_L0_UINT32 * pClrSRonTxVectorWd)
```

This function gets the settings for a RT CB for transmitter inhibits and mode code options.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).
rtAddr - RT address.
pAllowDBC - pointer to store DBC setting (1 to ALLOW DYNAMIC BUS CONTROL or 0 to IGNORE DYNAMIC BUS CONTROL).
pTxInh_A - pointer to store BUS A INHIBIT setting (1 to INHIBIT BUS A or 0 to ENABLE BUS A).
pTxInh_B - pointer to store BUS B INHIBIT setting (1 to INHIBIT BUS B or 0 to ENABLE BUS B).
pClrSRonTxVectorWd - pointer to store CLEAR SR setting (1 to CLEAR SR BIT ON TX VECTOR WORD or 0 to NOT CHANGE SR ON TX VECTOR WORD).

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_BAD_INPUT - Invalid RT address
ADT_FAILURE - Completed with error

ADT_L1_1553_RT_GetRespTime

ADT_L0_UINT32 ADT_L1_1553_RT_GetRespTime (ADT_L0_UINT32 *devID*,
 ADT_L0_UINT32 *rtAddr*,
 ADT_L0_UINT32 * *pRespTime100ns*)

This function gets the Status Response time (100ns LSB) for the specified RT.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).
rtAddr - RT address.
pRespTime100ns - pointer to store the response time (100ns LSB).

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_BAD_INPUT - Invalid RT address
ADT_FAILURE - Completed with error

ADT_L1_1553_RT_GetSingleRTAddr

ADT_L0_UINT32 ADT_L1_1553_RT_GetSingleRTAddr
 (ADT_L0_UINT32 *devID*,
 ADT_L0_UINT32 * *pRtAddr*)

This function gets the single RT address for the device.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).
pRtAddr - pointer to store the single RT address.

Returns:

ADT_SUCCESS - Completed without error
ADT_FAILURE - Completed with error

ADT_L1_1553_RT_Init

ADT_L0_UINT32 ADT_L1_1553_RT_Init (ADT_L0_UINT32 *devID*,
 ADT_L0_UINT32 *rtAddr*)

This function initializes data structures for the RT.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).
rtAddr - RT address.

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_BAD_INPUT - Invalid RT address
ADT_FAILURE - Completed with error

ADT_L1_1553_RT_InjStsWordError

ADT_L0_UINT32 ADT_L1_1553_RT_InjStsWordError (ADT_L0_UINT32 *devID*,
 ADT_L0_UINT32 *rtAddr*,
 ADT_L0_UINT32 *syncErr*,
 ADT_L0_UINT32 *manchesterErr*,
 ADT_L0_UINT32 *parityErr*)

This function configures a RT for errors on the status word. There are three parameters that determine the error settings. These are syncErr, manchesterErr, and parityErr. These parameters should be used in one of the following ways: If syncErr, manchesterErr, and parityErr are all zero, no error will be injected. If syncErr is non-zero, the sync pattern will be inverted. If manchesterErr is non-zero, a manchester (zero-crossing) error will be injected on bit 3 of the word. If parityErr is non-zero, the parity bit will be inverted. If both manchesterErr and parityErr are non-zero, a manchester (zero-crossing) error will be injected on the parity bit.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).
rtAddr - RT address.
syncErr - set to 1 for sync error or 0 for no sync error.
manchesterErr - set to 1 for manchester error or 0 for no manchester error.
parityErr - set to 1 for parity error or 0 for no parity error.

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_BAD_INPUT - Invalid RT address
ADT_FAILURE - Completed with error

ADT_L1_1553_RT_MC_CDPAllocate

```
ADT_L0_UINT32 ADT_L1_1553_RT_MC_CDPAllocate  
        (ADT_L0_UINT32 devID,  
         ADT_L0_UINT32 rtAddr,  
         ADT_L0_UINT32 tr,  
         ADT_L0_UINT32 modeCode,  
         ADT_L0_UINT32 numCDP)
```

This function allocates and links the requested number of CDPs for the RT MODE CODE.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).
rtAddr - RT address.
tr - transmit(1) or receive(0).
modeCode - mode code.
numCDP - requested number of CDP buffers.

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_BAD_INPUT - Invalid RT address, TR, subaddress, or numCDP
ADT_FAILURE - Completed with error

ADT_L1_1553_RT_MC_CDPFree

This function frees the RT MC Control Block and CDPs for the RT/MC and resets to default buffer.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).
rtAddr - RT address.
tr - transmit(1) or receive(0).
modeCode - mode code.

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_BAD_INPUT - Invalid RT address, TR, subaddress, or numCDP
ADT_FAILURE - Completed with error

ADT_L1_1553_RT_MC_CDPRead

ADT_L0_UINT32 ADT_L1_1553_RT_MC_CDPRead (ADT_L0_UINT32 devID,

```
ADT_L0_UINT32 rtAddr,  
ADT_L0_UINT32 tr,  
ADT_L0_UINT32 modeCode,  
ADT_L0_UINT32 cdpNum,  
ADT_L1_1553_CDP * pCdp)
```

This function reads a CDP for the RT MODE CODE.

NOTE: If you read a CDP buffer while the firmware is in the middle of processing a message for that buffer, then the CDP Status Word will be 0xFFFFFFFF. If you see this value, then you should read the buffer again until the CDP Status Word is NOT 0xFFFFFFFF – you will then have a complete CDP buffer. If you use interrupts to synchronize buffer reads to messages on the bus then you should not see this case.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).
rtAddr - RT address.
tr - transmit(1) or receive(0).
modeCode - mode code.
cdpNum - index of the CDP to write.
pCdp - pointer to the CDP structure to be read.

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_BAD_INPUT - Invalid RT address, TR, mode code, or cdpNum
ADT_FAILURE - Completed with error

ADT_L1_1553_RT_MC_CDPReadWords

```
ADT_L0_UINT32 ADT_L1_1553_RT_MC_CDPRead (ADT_L0_UINT32 devID,
                                         ADT_L0_UINT32 rtAddr,
                                         ADT_L0_UINT32 tr,
                                         ADT_L0_UINT32 modeCode,
                                         ADT_L0_UINT32 cdpNum,
                                         ADT_L0_UINT32 wordOffset,
                                         ADT_L0_UINT32 numOfWords,
                                         ADT_L0_UINT32 *pWords)
```

This function reads consecutive words from a CDP for the RT MODE CODE. The user must reference a valid CDP wordOffset – see AltaCore manual or 1553 Quick Reference for CDP structure. This function can save backplane cycles by reading only selected words from an RT CDP.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).
rtAddr - RT address.
tr - transmit(1) or receive(0).
modeCode - mode code.
cdpNum - CDP buffer number.
wordOffset - word offset in to the CDP.
numOfWords - number of words to read from the wordOffset of the CDP.
pWords - pointer to the UINT32 array of words.

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_BAD_INPUT - Invalid RT address, TR, modeCode, cdpNum, wordOffset or numOfWords
ADT_FAILURE - Completed with error

ADT_L1_1553_RT_MC_CDPWrite

```
ADT_L0_UINT32 ADT_L1_1553_RT_MC_CDPWrite (ADT_L0_UINT32 devID,  
                                         ADT_L0_UINT32 rtAddr,  
                                         ADT_L0_UINT32 tr,  
                                         ADT_L0_UINT32 modeCode,  
                                         ADT_L0_UINT32 cdpNum,  
                                         ADT_L1_1553_CDP * pCdp)
```

This function writes a CDP for the RT MODE CODE.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).
rtAddr - RT address.
tr - transmit(1) or receive(0).
modeCode - mode code.
cdpNum - index of the CDP to write.
pCdp - pointer to the CDP structure to be written.

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_BAD_INPUT - Invalid RT address, TR, mode code, or cdpNum
ADT_FAILURE - Completed with error

ADT_L1_1553_RT_MC_CDPWriteWords

```
ADT_L0_UINT32 ADT_L1_1553_RT_MC_CDPWriteWords (
    ADT_L0_UINT32 devID,
    ADT_L0_UINT32 rtAddr,
    ADT_L0_UINT32 tr,
    ADT_L0_UINT32 modeCode,
    ADT_L0_UINT32 cdpNum,
    ADT_L0_UINT32 wordOffset,
    ADT_L0_UINT32 numOfWords,
    ADT_L0_UINT32 *pWords)
```

This function writes consecutive words from a CDP for the RT MODE CODE. The user must reference a valid CDP wordOffset – see AltaCore manual or 1553 Quick Reference for CDP structure. This function can save backplane cycles by writing only selected words from an RT CDP.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).
rtAddr - RT address.
tr - transmit(1) or receive(0).
modeCode - mode code.
cdpNum - CDP buffer number.
wordOffset - word offset in to the CDP.
numOfWords - number of words to write to the wordOffset of the CDP.
pWords - pointer to the UINT32 array of words.

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_BAD_INPUT - Invalid RT address, TR, modeCode, cdpNum, wordOffset or numOfWords
ADT_FAILURE - Completed with error

ADT_L1_1553_RT_MC_LegalizationRead

```
ADT_L0_UINT32 ADT_L1_1553_RT_MC_LegalizationRead (
    ADT_L0_UINT32 devID,
    ADT_L0_UINT32 rtAddr,
    ADT_L0_UINT32 tr,
    ADT_L0_UINT32 modeCode,
    ADT_L0_UINT32 * pIllegalBits)
```

This function reads the legalization settings for the RT MODE CODE.

Mode codes are legalized in the mode code (MC) control blocks (NOT in the SA0 or SA31 structures).

In the legalization word for mode codes, the only bit that counts is the bit that corresponds to the mode code number (for example, bit 17 for mode code 17).

Mode code 2 (transmit status) and mode code 18 (transmit last command) are ALWAYS legal and cannot be illegalized.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).
rtAddr - RT address.
tr - transmit(1) or receive(0).
modeCode - mode code.
pIllegalBits - pointer to store the value read.

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_BAD_INPUT - Invalid RT address or subaddress
ADT_FAILURE - Completed with error

ADT_L1_1553_RT_MC_LegalizationWrite

```
ADT_L0_UINT32 ADT_L1_1553_RT_MC_LegalizationWrite (
    ADT_L0_UINT32 devID,
    ADT_L0_UINT32 rtAddr,
    ADT_L0_UINT32 tr,
    ADT_L0_UINT32 modeCode,
    ADT_L0_UINT32 illegalBits)
```

This function writes the legalization settings for the RT MODE CODE.

Mode codes are legalized in the mode code (MC) control blocks (NOT in the SA0 or SA31 structures).

In the legalization word for mode codes, the only bit that counts is the bit that corresponds to the mode code number (for example, bit 17 for mode code 17). The easiest thing to do is to set ALL bits in the legalization word to the same value - 0 for legal or 1 for illegal. The following sets mode code 16 to be illegal:

```
status = ADT_L1_1553_RT_MC_LegalizationWrite(DEVID, 1, 1, 16, 0xFFFFFFFF);
```

Mode code 2 (transmit status) and mode code 18 (transmit last command) are ALWAYS legal and cannot be illegalized.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).
rtAddr - RT address.
tr - transmit(1) or receive(0).
modeCode - mode code.
illegalBits - contains one bit for each word count, 0=legal, 1=illegal.

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_BAD_INPUT - Invalid RT address or subaddress
ADT_FAILURE - Completed with error

ADT_L1_1553_RT_Monitor

ADT_L0_UINT32 ADT_L1_1553_RT_Monitor (ADT_L0_UINT32 *devID*,
 ADT_L0_UINT32 *rtAddr*,
 ADT_L0_UINT32 *isMonitor*)

This function configures a RT to either MONITOR or RESPOND.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).
rtAddr - RT address.
isMonitor - set to 1 for MONITOR or 0 for RESPOND.

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_BAD_INPUT - Invalid RT address
ADT_FAILURE - Completed with error

ADT_L1_1553_RT_ReadStsWordError

ADT_L0_UINT32 ADT_L1_1553_RT_ReadStsWordError

```
(ADT_L0_UINT32 devID,  
ADT_L0_UINT32 rtAddr,  
ADT_L0_UINT32 * pSyncErr,  
ADT_L0_UINT32 * pManchesterErr,  
ADT_L0_UINT32 * pParityErr)
```

This function reads the settings for a RT for errors on the status word. There are three parameters that determine the error settings. These are syncErr, manchesterErr, and parityErr. These parameters should be used in one of the following ways: If syncErr, manchesterErr, and parityErr are all zero, no error will be injected. If syncErr is non-zero, the sync pattern will be inverted. If manchesterErr is non-zero, a manchester (zero-crossing) error will be injected on bit 3 of the word. If parityErr is non-zero, the parity bit will be inverted. If both manchesterErr and parityErr are non-zero, a manchester (zero-crossing) error will be injected on the parity bit.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).
rtAddr - RT address.
pSyncErr - ptr to store sync error (1 for sync error or 0 for no sync error).
pManchesterErr – ptr to store manch error (1 for manch error or 0 for no manch error).
pParityErr - ptr to store parity error (1 for parity error or 0 for no parity error).

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_BAD_INPUT - Invalid RT address
ADT_FAILURE - Completed with error

ADT_L1_1553_RT_SA_CDPAllocate

```
ADT_L0_UINT32 ADT_L1_1553_RT_SA_CDPAllocate (ADT_L0_UINT32 devID,  
                                              ADT_L0_UINT32 rtAddr,  
                                              ADT_L0_UINT32 tr,  
                                              ADT_L0_UINT32 subAddr,  
                                              ADT_L0_UINT32 numCDP)
```

This function allocates and links the requested number of CDPs for the RT/SA.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).
rtAddr - RT address.
tr - transmit(1) or receive(0).
subAddr - subaddress.
numCDP - requested number of CDP buffers.

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_BAD_INPUT - Invalid RT address, TR, subaddress, or numCDP
ADT_FAILURE - Completed with error

ADT_L1_1553_RT_SA_CDPFree

```
ADT_L0_UINT32 ADT_L1_1553_RT_SA_CDPFree (ADT_L0_UINT32 devID,  
                                           ADT_L0_UINT32 rtAddr,  
                                           ADT_L0_UINT32 tr,  
                                           ADT_L0_UINT32 subAddr)
```

This function frees the RT SA Control Block and CDPs for the RT/SA and resets to default buffer.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).
rtAddr - RT address.
tr - transmit(1) or receive(0).
subAddr - subaddress.

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_BAD_INPUT - Invalid RT address, TR, subaddress, or numCDP
ADT_FAILURE - Completed with error

ADT_L1_1553_RT_SA_CDPRead

```
ADT_L0_UINT32 ADT_L1_1553_RT_SA_CDPRead (ADT_L0_UINT32 devID,
                                         ADT_L0_UINT32 rtAddr,
                                         ADT_L0_UINT32 tr,
                                         ADT_L0_UINT32 subAddr,
                                         ADT_L0_UINT32 cdpNum,
                                         ADT_L1_1553_CDP * pCdp)
```

This function reads a CDP for the RT/SA.

NOTE: If you read a CDP buffer while the firmware is in the middle of processing a message for that buffer, then the CDP Status Word will be 0xFFFFFFFF. If you see this value, then you should read the buffer again until the CDP Status Word is NOT 0xFFFFFFFF – you will then have a complete CDP buffer. If you use interrupts to synchronize buffer reads to messages on the bus then you should not see this case.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).
rtAddr - RT address.
tr - transmit(1) or receive(0).
subAddr - subaddress.
cdpNum - index of the CDP to write.
pCdp - pointer to the CDP structure to be read.

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_BAD_INPUT - Invalid RT address, TR, subaddress, or cdpNum
ADT_FAILURE - Completed with error

ADT_L1_1553_RT_SA_CDPReadWords

```
ADT_L0_UINT32 ADT_L1_1553_RT_SA_CDPReadWords (  
    ADT_L0_UINT32 devID,  
    ADT_L0_UINT32 rtAddr,  
    ADT_L0_UINT32 tr,  
    ADT_L0_UINT32 subAddr,  
    ADT_L0_UINT32 cdpNum,  
    ADT_L0_UINT32 wordOffset,  
    ADT_L0_UINT32 numWords,  
    ADT_L0_UINT32 *pWords)
```

This function reads consecutive words from a CDP for the RT SA. The user must reference a valid CDP wordOffset – see AltaCore manual or 1553 Quick Reference for CDP structure. This function can save backplane cycles by reading only selected words from an RT CDP.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).
rtAddr - RT address.
tr - transmit(1) or receive(0).
subAddr - subaddress.
cdpNum - CDP buffer number.
wordOffset - word offset in to the CDP.
numWords - number of words to read from the wordOffset of the CDP.
pWords - pointer to the UINT32 array of words.

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_BAD_INPUT - Invalid RT address, TR, subaddress, cdpNum, wordOffset or numWords
ADT_FAILURE - Completed with error

ADT_L1_1553_RT_SA_CDPWrite

```
ADT_L0_UINT32 ADT_L1_1553_RT_SA_CDPWrite (ADT_L0_UINT32 devID,  
                                         ADT_L0_UINT32 rtAddr,  
                                         ADT_L0_UINT32 tr,  
                                         ADT_L0_UINT32 subAddr,  
                                         ADT_L0_UINT32 cdpNum,  
                                         ADT_L1_1553_CDP * pCdp)
```

This function writes a CDP for the RT/SA.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).
rtAddr - RT address.
tr - transmit(1) or receive(0).
subAddr - subaddress.
cdpNum - index of the CDP to write.
pCdp - pointer to the CDP structure to be written.

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_BAD_INPUT - Invalid RT address, TR, subaddress, or cdpNum
ADT_FAILURE - Completed with error

ADT_L1_1553_RT_SA_CDPWriteWords

```
ADT_L0_UINT32 ADT_L1_1553_RT_SA_CDPWriteWords (
    ADT_L0_UINT32 devID,
    ADT_L0_UINT32 rtAddr,
    ADT_L0_UINT32 tr,
    ADT_L0_UINT32 subAddr,
    ADT_L0_UINT32 cdpNum,
    ADT_L0_UINT32 wordOffset,
    ADT_L0_UINT32 numOfWords,
    ADT_L0_UINT32 *pWords)
```

This function writes consecutive words from a CDP for the RT SA. The user must reference a valid CDP wordOffset – see AltaCore manual or 1553 Quick Reference for CDP structure. This function can save backplane cycles by writing only selected words from an RT CDP.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).
rtAddr - RT address.
tr - transmit(1) or receive(0).
subAddr - subaddress.
cdpNum - CDP buffer number.
wordOffset - word offset in to the CDP.
numOfWords - number of words to write to the wordOffset of the CDP.
pWords - pointer to the UINT32 array of words.

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_BAD_INPUT - Invalid RT address, TR, subaddress, cdpNum, wordOffset or numOfWords
ADT_FAILURE - Completed with error

ADT_L1_1553_RT_SA_LegalizationRead

ADT_L0_UINT32 ADT_L1_1553_RT_SA_LegalizationRead

(ADT_L0_UINT32 *devID*,
ADT_L0_UINT32 *rtAddr*,
ADT_L0_UINT32 *tr*,
ADT_L0_UINT32 *subAddr*,
ADT_L0_UINT32 * *pIllegalBits*)

This function reads the legalization settings for the RT/SA.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).
rtAddr - RT address.
tr - transmit(1) or receive(0).
subAddr - subaddress.
pIllegalBits - pointer to store the value read.

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_BAD_INPUT - Invalid RT address, T/R, or subaddress
ADT_FAILURE - Completed with error

ADT_L1_1553_RT_SA_LegalizationWrite

ADT_L0_UINT32 ADT_L1_1553_RT_SA_LegalizationWrite
(ADT_L0_UINT32 *devID*,
ADT_L0_UINT32 *rtAddr*,
ADT_L0_UINT32 *tr*,
ADT_L0_UINT32 *subAddr*,
ADT_L0_UINT32 *illegalBits*)

This function writes the legalization settings for the RT/SA.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).
rtAddr - address.
tr - transmit(1) or receive(0).
subAddr - subaddress.
illegalBits - contains one bit for each word count, 0=legal, 1=illegal.

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_BAD_INPUT - Invalid RT address, T/R, or subaddress
ADT_FAILURE - Completed with error

ADT_L1_1553_RT_SetOptions

```
ADT_L0_UINT32 ADT_L1_1553_RT_SetOptions (ADT_L0_UINT32 devID,  
                                         ADT_L0_UINT32 rtAddr,  
                                         ADT_L0_UINT32 allowDBC,  
                                         ADT_L0_UINT32 txInh_A,  
                                         ADT_L0_UINT32 txInh_B,  
                                         ADT_L0_UINT32 clrSRonTxVectorWd)
```

This function configures a RT CB for transmitter inhibits and mode code options.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).
rtAddr - RT address.
allowDBC - set to 1 to ALLOW DYNAMIC BUS CONTROL or 0 to IGNORE DYNAMIC BUS CONTROL.
txInh_A - set to 1 to INHIBIT BUS A or 0 to ENABLE BUS A.
txInh_B - set to 1 to INHIBIT BUS B or 0 to ENABLE BUS B.
clrSRonTxVectorWd - set to 1 to CLEAR SR BIT ON TX VECTOR WORD or 0 to NOT CHANGE SR ON TX VECTOR WORD.

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_BAD_INPUT - Invalid RT address
ADT_FAILURE - Completed with error

ADT_L1_1553_RT_SetRespTime

```
ADT_L0_UINT32 ADT_L1_1553_RT_SetRespTime (ADT_L0_UINT32 devID,  
                                         ADT_L0_UINT32 rtAddr,  
                                         ADT_L0_UINT32 respTime100ns)
```

This function sets the Status Response time (100ns LSB) for the specified RT.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).
rtAddr - RT address.
respTime100ns - response time (100ns LSB, max value allowed 4095 = 409.5us).

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_BAD_INPUT - Invalid RT address or resp time
ADT_FAILURE - Completed with error

ADT_L1_1553_RT_SetSingleRTAddr

ADT_L0_UINT32 ADT_L1_1553_RT_SetSingleRTAddr
(ADT_L0_UINT32 *devID*,
ADT_L0_UINT32 *rtAddr*)

This function sets the single RT address for the device.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).
rtAddr - RT address (0-31).

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_BAD_INPUT - Invalid RT address
ADT_FAILURE - Completed with error

ADT_L1_1553_RT_Start

ADT_L0_UINT32 ADT_L1_1553_RT_Start (ADT_L0_UINT32 *devID*)

This function starts the RT function of the channel.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).

Returns:

ADT_SUCCESS - Completed without error
ADT_FAILURE - Completed with error

ADT_L1_1553_RT_StatusRead

```
ADT_L0_UINT32 ADT_L1_1553_RT_StatusRead (ADT_L0_UINT32 devID,  
                                         ADT_L0_UINT32 rtAddr,  
                                         ADT_L0_UINT32 * pStsWord)
```

This function reads the Status word for the RT.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).
rtAddr - RT address.
pStsWord - pointer to store the status word.

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_BAD_INPUT - Invalid RT address
ADT_FAILURE - Completed with error

ADT_L1_1553_RT_StatusWrite

```
ADT_L0_UINT32 ADT_L1_1553_RT_StatusWrite (ADT_L0_UINT32 devID,  
                                         ADT_L0_UINT32 rtAddr,  
                                         ADT_L0_UINT32 stsWord)
```

This function writes the Status word for the RT. Note that you CANNOT change the RT Address (bits 11-15) or the Message Error (ME) bit (bit 10). The ME bit is strictly defined by the MIL-STD-1553 specification and is controlled by the Alta Protocol Engine.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).
rtAddr - RT address.
stsWord - status word.

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_BAD_INPUT - Invalid RT address
ADT_FAILURE - Completed with error

ADT_L1_1553_RT_Stop

ADT_L0_UINT32 ADT_L1_1553_RT_Stop (ADT_L0_UINT32 *devID*)

This function stops the RT function of the channel.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).

Returns:

ADT_SUCCESS - Completed without error

ADT_FAILURE - Completed with error

ADT_L1_1553_RT_SA_CDP_GetAddr

ADT_L0_UINT32 ADT_L1_1553_RT_SA_CDP_GetAddr(
 ADT_L0_UINT32 *devID*,
 ADT_L0_UINT32 *rtAddr*,
 ADT_L0_UINT32 *tr*,
 ADT_L0_UINT32 *subAddr*,
 ADT_L0_UINT32 *cdpNum*,
 ADT_L0_UINT32 **pAddr*)

This function gets the byte address of a RT/SA CDP – see AltaCore-1553 manual or 1553 Quick Reference for the CDP structure. This address can be used to directly read or write words in a CDP using the functions `ADT_L1_ReadDeviceMem32` and `ADT_L1_WriteDeviceMem32`.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).

rtAddr - RT Address.

tr - Transmit (1) or Receive (0).

subAddr - Sub Address.

cdpNum - buffer (CDP) number.

pAddr - pointer to store the address.

Returns:

ADT_SUCCESS - Completed without error

ADT_ERR_BAD_INPUT - Invalid input parameter

ADT_FAILURE - Completed with error

ADT_L1_1553_RT_MC_CDP_GetAddr

```
ADT_L0_UINT32 ADT_L1_1553_RT_MC_CDP_GetAddr(  
    ADT_L0_UINT32 devID,  
    ADT_L0_UINT32 rtAddr,  
    ADT_L0_UINT32 tr,  
    ADT_L0_UINT32 modeCode,  
    ADT_L0_UINT32 cdpNum,  
    ADT_L0_UINT32 *pAddr)
```

This function gets the byte address of a RT/MC (mode code) CDP – see AltaCore-1553 manual or 1553 Quick Reference for the CDP structure. This address can be used to directly read or write words in a CDP using the functions `ADT_L1_ReadDeviceMem32` and `ADT_L1_WriteDeviceMem32`.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).
rtAddr - RT Address.
tr - Transmit (1) or Receive (0).
modeCode - mode code.
cdpNum - buffer (CDP) number.
pAddr - pointer to store the address.

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_BAD_INPUT - Invalid input parameter
ADT_FAILURE - Completed with error

1553 Bus Controller Functions

This section describes the Layer 1 API 1553 Bus Controller functions. These functions are defined in the file ADT_L1_1553_BC.c.

ADT_L1_1553_BC_AperiodicIsRunning

ADT_L0_UINT32 ADT_L1_1553_BC_AperiodicIsRunning
(ADT_L0_UINT32 *devID*,
ADT_L0_UINT32 *priority*,
ADT_L0_UINT32 * *plsRunning*)

This function determines if the BC is processing aperiodic messages.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).
priority - 0 for LOW priority or 1 for HIGH priority.
plsRunning - pointer to store the result, 0=NotRunning, 1=Running.

Returns:

ADT_SUCCESS - Completed without error
ADT_FAILURE - Completed with error

ADT_L1_1553_BC_AperiodicSend

ADT_L0_UINT32 ADT_L1_1553_BC_AperiodicSend (ADT_L0_UINT32 *devID*,
 ADT_L0_UINT32 *priority*,
 ADT_L0_UINT32 *msgnum*,
 ADT_L0_UINT32 *LPAMtime*)

This function injects BC Aperiodic messages into a running frame.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).
priority - 0 for LOW priority or 1 for HIGH priority.
msgnum - message number for the aperiodic message to send.
LPAMtime - low priority aperiodic message time (100ns LSB).
If this is ZERO, then the LPAM will be sent at end of frame no matter how much or little time is remaining in the frame.

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_BAD_INPUT - Invalid message number
ADT_ERR_NO_BCCB_TABLE - BCCB table pointer is zero (table not allocated)
ADT_FAILURE - Completed with error

Warnings & Notes:

- The aperiodic BCCB (or the last message in an aperiodic list of messages) must have **ADT_L1_1553_BC_NO_NEXT_MSG (0xFFFFFFFF)** as the **NextMsgNum** to indicate the end of the aperiodic list of messages.
- Before sending any aperiodic message, the application should check to see if the PE is currently processing aperiodic messages with the **ADT_L1_1553_BC_AperiodicIsRunning()** function.
- Subframe and Branch logic is not allowed in aperiodic chains.
- See section on **Aperiodic Messages** for detailed information on aperiodic message scheduling.

ADT_L1_1553_BC_CB_CDPAllocate

ADT_L0_UINT32 ADT_L1_1553_BC_CB_CDPAllocate (ADT_L0_UINT32 *devID*,
 ADT_L0_UINT32 *msgnum*,
 ADT_L0_UINT32 *numCDP*)

This function allocates memory for a BC Control Block and allocates/links CDPs.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).
msgnum - message number (zero-indexed) for which to allocate a BCCB.
numCDP - number of CDPs to allocate and link for the BCCB.

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_BAD_INPUT - Invalid message number
ADT_ERR_NO_BCCB_TABLE - BCCB table pointer is zero (table not allocated)
ADT_ERR_BCCB_ALREADY_ALLOCATED - A BCCB has already been allocated
 for *msgnum*
ADT_FAILURE - Completed with error

ADT_L1_1553_BC_CB_CDPFree

ADT_L0_UINT32 ADT_L1_1553_BC_CB_CDPFree (ADT_L0_UINT32 *devID*,
 ADT_L0_UINT32 *msgnum*)

This function frees memory for a BC Control Block and associated CDPs.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).
msgnum - message number (zero-indexed) of the BCCB to free.

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_BAD_INPUT - Invalid device number
ADT_ERR_NO_BCCB_TABLE - BCCB table pointer is zero (table not allocated)
ADT_ERR_BCCB_NOT_ALLOCATED - No BCCB has been allocated for *msgnum*
ADT_FAILURE - Completed with error

ADT_L1_1553_BC_CB_CDPRead

```
ADT_L0_UINT32 ADT_L1_1553_BC_CB_CDPRead (ADT_L0_UINT32 devID,  
                                         ADT_L0_UINT32 msgnum,  
                                         ADT_L0_UINT32 cdpNum,  
                                         ADT_L1_1553_CDP * pCdp)
```

This function reads a CDP for a BC Control Block.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).
msgnum - message (BC CB) number.
cdpNum - buffer (CDP) number.
pCdp - pointer to the CDP structure.

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_BAD_INPUT - Invalid message number
ADT_ERR_NO_BCCB_TABLE - BCCB table pointer is zero (table not allocated)
ADT_ERR_BCCB_NOT_ALLOCATED - No BCCB has been allocated for msgnum
ADT_FAILURE - Completed with error

ADT_L1_1553_BC_CB_CDPReadWords

```
ADT_L0_UINT32 ADT_L1_1553_BC_CB_CDPReadWords (  
    ADT_L0_UINT32 devID,  
    ADT_L0_UINT32 msgnum,  
    ADT_L0_UINT32 cdpNum,  
    ADT_L0_UINT32 wordOffset,  
    ADT_L0_UINT32 numOfWords,  
    ADT_L0_UINT32 *pWords)
```

This function reads consecutive words from a BCCB CDP. The user must reference a valid CDP wordOffset – see AltaCore manual or 1553 Quick Reference for CDP structure. This function can be useful to read only a small number of words versus the whole CDP – this can save many backplane cycles.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).
msgnum - message (BC CB) number.
cdpNum - buffer (CDP) number.
wordOffset - word offset in to the CDP.
numOfWords - number of words to read from the wordOffset of the CDP.
pWords - pointer to the UINT32 array of words.

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_BAD_INPUT Invalid message number, *cdpNum*, *wordOffset* or
 numOfWords
ADT_FAILURE - Completed with error

ADT_L1_1553_BC_CB_CDPWrite

```
ADT_L0_UINT32 ADT_L1_1553_BC_CB_CDPWrite (ADT_L0_UINT32 devID,  
                                         ADT_L0_UINT32 msgnum,  
                                         ADT_L0_UINT32 cdpNum,  
                                         ADT_L1_1553_CDP * pCdp)
```

This function writes a CDP for BC Control Block.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).
msgnum - message (BC CB) number.
cdpNum - buffer (CDP) number.
pCdp - pointer to the CDP structure.

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_BAD_INPUT - Invalid message number
ADT_ERR_NO_BCCB_TABLE - BCCB table pointer is zero (table not allocated)
ADT_ERR_BCCB_NOT_ALLOCATED - No BCCB has been allocated for msgnum
ADT_FAILURE - Completed with error

ADT_L1_1553_BC_CB_CDPWriteWords

```
ADT_L0_UINT32 ADT_L1_1553_BC_CB_CDPWriteWords (
    ADT_L0_UINT32 devID,
    ADT_L0_UINT32 msgnum,
    ADT_L0_UINT32 cdpNum,
    ADT_L0_UINT32 wordOffset,
    ADT_L0_UINT32 numOfWords,
    ADT_L0_UINT32 *pWords)
```

This function writes consecutive words to BCCB CDP. The user must reference a valid CDP wordOffset – see AltaCore manual or 1553 Quick Reference for CDP structure. This function can be useful to write only a small number of words verses the whole CDP – this can save many backplane cycles.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).
msgnum - message (BC CB) number.
cdpNum - buffer (CDP) number.
wordOffset - word offset in to the CDP.
numOfWords - number of words to write to the wordOffset of the CDP.
pWords - pointer to the UINT32 array of words.

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_BAD_INPUT - Invalid message number, *cdpNum*, *wordOffset* or
 numOfWords
ADT_FAILURE - Completed with error

ADT_L1_1553_BC_CB_Read

```
ADT_L0_UINT32 ADT_L1_1553_BC_CB_Read (ADT_L0_UINT32 devID,  
                                      ADT_L0_UINT32 msgnum,  
                                      ADT_L1_1553_BC_CB * bccb)
```

This function reads a BC Control Block.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).
msgnum - message number (zero-indexed) of the BCCB to read.
bccb - pointer to store the BCCB read.

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_BAD_INPUT - Invalid message number
ADT_ERR_NO_BCCB_TABLE - BCCB table pointer is zero (table not allocated)
ADT_ERR_BCCB_NOT_ALLOCATED - No BCCB has been allocated for *msgnum*
ADT_FAILURE - Completed with error

ADT_L1_1553_BC_CB_ReadWords

```
ADT_L0_UINT32 ADT_L1_1553_BC_CB_ReadWords (
    ADT_L0_UINT32 devID,
    ADT_L0_UINT32 msgnum,
    ADT_L0_UINT32 wordOffset,
    ADT_L0_UINT32 numOfWords,
    ADT_L0_UINT32 *pWords)
```

This function reads consecutive words from a BCCB. The user must reference a valid BCCB wordOffset – see AltaCore-1553 manual or 1553 Quick Reference for BCCB structure.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).
msgnum - message (BC CB) number.
wordOffset - word offset in to the BCCB.
numOfWords - number of words to read from the wordOffset of the BCCB.
pWords - pointer to the UINT32 array of words.

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_BAD_INPUT Invalid message number, wordOffset or numOfWords
ADT_FAILURE - Completed with error

ADT_L1_1553_BC_CB_Write

```
ADT_L0_UINT32 ADT_L1_1553_BC_CB_Write (ADT_L0_UINT32 devID,  
                                         ADT_L0_UINT32 msgnum,  
                                         ADT_L1_1553_BC_CB * bccb)
```

This function writes a BC Control Block.

**DO NOT SET BOTH ADT_L1_1553_BC_CB_CSR_STARTFRAME AND
ADT_L1_1553_BC_CB_CSR_SCHEDTIMING ON THE SAME BCCB! IF
ADT_L1_1553_BC_CB_CSR_STARTFRAME IS SET DO NOT SET
ADT_L1_1553_BC_CB_CSR_SCHEDTIMING! You cannot schedule from the
start of frame on the frame start message.**

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).
msgnum - message number (zero-indexed) of the BCCB to write.
bccb - pointer to the BCCB to write.

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_BAD_INPUT - Invalid message number
ADT_ERR_NO_BCCB_TABLE - BCCB table pointer is zero (table not allocated)
ADT_ERR_BCCB_NOT_ALLOCATED - No BCCB has been allocated for *msgnum*
ADT_FAILURE - Completed with error

ADT_L1_1553_BC_CB_WriteWords

```
ADT_L0_UINT32 ADT_L1_1553_BC_CB_WriteWords (
    ADT_L0_UINT32 devID,
    ADT_L0_UINT32 msgnum,
    ADT_L0_UINT32 wordOffset,
    ADT_L0_UINT32 numOfWords,
    ADT_L0_UINT32 *pWords)
```

This function writes consecutive words to BCCB. The user must reference a valid BCCB wordOffset – see AltaCore-1553 manual or 1553 Quick Reference for BCCB structure.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).
msgnum - message (BC CB) number.
wordOffset - word offset in to the BCCB.
numOfWords - number of words to write to the wordOffset of the BCCB.
pWords - pointer to the array of words.

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_BAD_INPUT - Invalid message number, wordOffset or numOfWords
ADT_FAILURE - Completed with error

ADT_L1_1553_BC_CB_SetAddressBranchValues

```
ADT_L0_UINT32 ADT_L1_1553_BC_CB_SetAddressBranchvalues (
    ADT_L0_UINT32 devID,
    ADT_L0_UINT32 msgNum,
    ADT_L0_UINT32 srcMsgNum,
    ADT_L0_UINT32 srcCdpNum,
    ADT_L0_UINT32 srcWordOffset,
    ADT_L0_UINT32 maskValue,
    ADT_L0_UINT32 compareValue,
    ADT_L0_UINT32 destMsgNum)
```

This function sets the registers for referenced BCCB *msgNum* for an “Address Branch Only” function (the user must also have the appropriate BCCB CSR bit set). The “Address Branch Only” function directs the PE look for a specific data value and make a BC branch decision to another BCCB. This can be used to make IF-ELSE decisions chains during any point in BCCB message execution. See BC18.c example to see how possible code can be setup.

Example: Force PE to look for a value of 0x00005555 from CDP Data Word 1 (which is word offset 0x11 or byte offset 0x44 of a CDP) of BCCB 5, CDP 0. If true, then jump to BCCB 10. The current BCCB message being defined is 6.

```
ADT_L1_1553_BC_CB_SetAddressBranchValue(DEVID, 6, 5, 0, 0x11, 0x0000FFFF, 0x00005555, 10);
```

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).
msgNum - message (BC CB) number where setup will occur (usually is the current BCCB number begin defined in your code)
srcMsgNum - BCCB number from which the data comparision will occur.
srcCdpNum - CDP number of the *srcMsgNum* from which the data comparison will occur.
srcWordOffset - CDP word offset (from *srcMsgNum* and *srcCdpNum*) from which the data comparison will occur.
maskValue - 32-bit logical AND value for masking desired bits on the *srcWordOffset*.
compareValue - data value expected after the *maskValue* is applied.
destMsgNum - destination BCCB number that BC execution will jump to if there is a positive value after the mask/compare operation.

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_BAD_INPUT - Invalid message number, wordOffset or numOfWords
ADT_FAILURE - Completed with error

ADT_L1_1553_BC_Close

ADT_L0_UINT32 ADT_L1_1553_BC_Close (ADT_L0_UINT32 *devID*)

This function frees all BC resources for the device.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_BAD_INPUT - Invalid device number
ADT_FAILURE - Completed with error

ADT_L1_1553_BC_GetFrameCount

ADT_L0_UINT32 ADT_L1_1553_BC_GetFrameCount (ADT_L0_UINT32 *devID*,
 ADT_L0_UINT32 * *pFrameCount*)

This function reads the BC total frame count.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).
pFrameCount - pointer to store the frame count.

Returns:

ADT_SUCCESS - Completed without error
ADT_FAILURE - Completed with error

ADT_L1_1553_BC_Init

ADT_L0_UINT32 ADT_L1_1553_BC_Init (ADT_L0_UINT32 *devID*,
 ADT_L0_UINT32 *max_num_msgs*,
 ADT_L0_UINT32 *minors_per_major*,
 ADT_L0_UINT32 *bcCsr*)

This function initializes BC settings for the device.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).
max_num_msgs - maximum number of BC control blocks to be used.
minors_per_major - number of minor frames per major frame.
bcCsr - initial value for the BC CSR.

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_BAD_INPUT - Invalid device number
ADT_FAILURE - Completed with error

ADT_L1_1553_BC_InjCmdWordError

ADT_L0_UINT32 ADT_L1_1553_BC_InjCmdWordError

(*ADT_L0_UINT32 devID,*
ADT_L0_UINT32 msgNum,
ADT_L0_UINT32 cmd1or2,
ADT_L0_UINT32 syncErr,
ADT_L0_UINT32 manchesterErr,
ADT_L0_UINT32 parityErr)

This function configures a BC message for errors on the command word. There are three parameters to select the error type to inject. These are syncErr, manchesterErr, and parityErr. These parameters should be used in one of the following ways: If syncErr, manchesterErr, and parityErr are all zero, no error will be injected. If syncErr is non-zero, the sync pattern will be inverted. If manchesterErr is non-zero, a manchester (zero-crossing) error will be injected on bit 3 of the word. If parityErr is non-zero, the parity bit will be inverted. If both manchesterErr and parityErr are non-zero, a manchester (zero-crossing) error will be injected on the parity bit.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).
msgNum - message number.
cmd1or2 - selects the CMD word (1 or 2).
syncErr - set to 1 for sync error or 0 for no sync error.
manchesterErr - set to 1 for manchester error or 0 for no manchester error.
parityErr - set to 1 for parity error or 0 for no parity error.

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_BAD_INPUT - Invalid message number
ADT_FAILURE - Completed with error

ADT_L1_1553_BC_IsRunning

ADT_L0_UINT32 ADT_L1_1553_BC_IsRunning (ADT_L0_UINT32 *devID*,
 ADT_L0_UINT32 * *pIsRunning*)

This function determines if the BC is running or stopped.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).
pIsRunning - pointer to store the result, 0=Not Running, 1=Running.

Returns:

ADT_SUCCESS - Completed without error
ADT_FAILURE - Completed with error

ADT_L1_1553_BC_ReadCmdWordError

ADT_L0_UINT32 ADT_L1_1553_BC_ReadCmdWordError

```
(ADT_L0_UINT32 devID,  
ADT_L0_UINT32 msgNum,  
ADT_L0_UINT32 cmd1or2,  
ADT_L0_UINT32 * pSyncErr,  
ADT_L0_UINT32 * pManchesterErr,  
ADT_L0_UINT32 * pParityErr)
```

This function reads the settings for a BC message for errors on the command word. There are three parameters that determine the error settings. These are syncErr, manchesterErr, and parityErr. These parameters should be used in one of the following ways: If syncErr, manchesterErr, and parityErr are all zero, no error will be injected. If syncErr is non-zero, the sync pattern will be inverted. If manchesterErr is non-zero, a manchester (zero-crossing) error will be injected on bit 3 of the word. If parityErr is non-zero, the parity bit will be inverted. If both manchesterErr and parityErr are non-zero, a manchester (zero-crossing) error will be injected on the parity bit.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).
msgNum - message number.
cmd1or2 - selects the CMD word (1 or 2).
pSyncErr - ptr to store sync error (1 for sync error or 0 for no sync error).
pManchesterErr - ptr to store manch error (1 for manch error or 0 for no manch error).
pParityErr - ptr to store parity error (1 for parity error or 0 for no parity error).

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_BAD_INPUT - Invalid message number
ADT_FAILURE - Completed with error

ADT_L1_1553_BC_Start

ADT_L0_UINT32 ADT_L1_1553_BC_Start (ADT_L0_UINT32 *devID*,
 ADT_L0_UINT32 *msgnum*)

This function starts BC operation for the device on the selected message number.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).
msgnum - message number to start the BC on.

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_BAD_INPUT - Invalid message number
ADT_ERR_NO_BCCB_TABLE - BCCB table pointer is zero (table not allocated)
ADT_FAILURE - Completed with error

ADT_L1_1553_BC_Stop

ADT_L0_UINT32 ADT_L1_1553_BC_Stop (ADT_L0_UINT32 *devID*)

This function stops BC operation for the device.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).

Returns:

ADT_SUCCESS - Completed without error
ADT_FAILURE - Completed with error

ADT_L1_1553_BC_CB_GetAddr

```
ADT_L0_UINT32 ADT_L1_1553_BC_CB_GetAddr(ADT_L0_UINT32 devID,  
                                         ADT_L0_UINT32 msgnum,  
                                         ADT_L0_UINT32 *pAddr)
```

This function gets the byte address of a BC Control Block (BCCB) – see AltaCore-1553 manual or 1553 Quick Reference for BCCB structure. This address can be used to directly read or write words in a BCCB using the functions `ADT_L1_ReadDeviceMem32` and `ADT_L1_WriteDeviceMem32`.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).
msgnum - message (BC CB) number.
pAddr - pointer to store the address.

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_BAD_INPUT - Invalid message number
ADT_FAILURE - Completed with error

ADT_L1_1553_BC_CB_CDP_GetAddr

```
ADT_L0_UINT32 ADT_L1_1553_BC_CB_GetAddr(ADT_L0_UINT32 devID,  
                                         ADT_L0_UINT32 msgnum,  
                                         ADT_L0_UINT32 cdpNum,  
                                         ADT_L0_UINT32 *pAddr)
```

This function gets the byte address of a BC CDP – see AltaCore-1553 manual or 1553 Quick Reference for the CDP structure. This address can be used to directly read or write words in a CDP using the functions `ADT_L1_ReadDeviceMem32` and `ADT_L1_WriteDeviceMem32`.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).
msgnum - message (BC CB) number.
cdpNum - buffer (CDP) number.
pAddr - pointer to store the address.

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_BAD_INPUT - Invalid message number or CDP number
ADT_FAILURE - Completed with error

1553 Signal Generator Functions

This section describes the Layer 1 API 1553 Signal Generator functions. These functions are defined in the file ADT_L1_1553_SG.c.

ADT_L1_1553_SG_AddVectors

```
ADT_L0_UINT32 ADT_L1_1553_SG_AddVectors (ADT_L0_UINT32 numVec,  
                                         ADT_L0_UINT32 vector2bit,  
                                         ADT_L0_UINT32 * pVectors,  
                                         ADT_L0_UINT32 sizeInWords,  
                                         ADT_L0_UINT32 * pNumVectors)
```

This function adds a sequence of vectors to a list of vectors.

Parameters:

numVec - number of 2-bit vectors to add (1 vector = 20ns).
vector2bit - 2-bit pattern to add for each vector (00=GND, 01=LOW, 10=HIGH, 11=GND).
pVectors - pointer to the array of 32-bit words containing vectors.
sizeInWords - max size (in words) of the vector array.
pNumVectors - pointer to a UINT32 containing the current vector count (in bits).

Returns:

ADT_SUCCESS - Completed without error
ADT_FAILURE - Completed with error

ADT_L1_1553_SG_Configure

```
ADT_L0_UINT32 ADT_L1_1553_SG_Configure (ADT_L0_UINT32 devID)
```

This function configures Signal Generator operation for the device by initializing the SG registers.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).

Returns:

ADT_SUCCESS - Completed without error
ADT_FAILURE - Completed with error

ADT_L1_1553_SG_CreateSGCB

```
ADT_L0_UINT32 ADT_L1_1553_SG_CreateSGCB (ADT_L0_UINT32 devID,  
                                         char bus,  
                                         ADT_L0_UINT32 timeHigh,  
                                         ADT_L0_UINT32 timeLow,  
                                         ADT_L0_UINT32 * pVectors,  
                                         ADT_L0_UINT32 numVectors)
```

This function allocates and writes a SGCB to the device.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).
bus - 1553 bus (A or B).
timeHigh - upper 32 bits of the time for this SGCB.
timeLow - lower 32 bits of the time for this SGCB.
pVectors - pointer to an array of 32-bit words containing vectors.
numVectors - number of vectors (in bits, may use only part of the last word).

Returns:

ADT_SUCCESS - Completed without error
ADT_FAILURE - Completed with error

ADT_L1_1553_SG_Free

```
ADT_L0_UINT32 ADT_L1_1553_SG_Free (ADT_L0_UINT32 devID)
```

This function frees all SG CBs for the device.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).

Returns:

ADT_SUCCESS - Completed without error
ADT_FAILURE - Completed with error

ADT_L1_1553_SG_Start

ADT_L0_UINT32 ADT_L1_1553_SG_Start (ADT_L0_UINT32 *devID*)

This function starts Signal Generator operation for the device.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).

Returns:

ADT_SUCCESS - Completed without error

ADT_FAILURE - Completed with error

ADT_L1_1553_SG_Stop

ADT_L0_UINT32 ADT_L1_1553_SG_Stop (ADT_L0_UINT32 *devID*)

This function stops Signal Generator operation for the device.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).

Returns:

ADT_SUCCESS - Completed without error

ADT_FAILURE - Completed with error

ADT_L1_1553_SG_IsRunning

ADT_L0_UINT32 ADT_L1_1553_SG_IsRunning (ADT_L0_UINT32 *devID*,
ADT_L0_UINT32 * *pIsRunning*)

This function determines if the Signal Generator is running or stopped.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).

pIsRunning - pointer to store the result, 0=Not Running, 1=Running.

Returns:

ADT_SUCCESS - Completed without error

ADT_FAILURE - Completed with error

ADT_L1_1553_SG_WordToVectors

ADT_L0_UINT32 ADT_L1_1553_SG_WordToVectors

(ADT_L0_UINT32 *m1553word*,
ADT_L0_UINT32 * *pVectors*,
ADT_L0_UINT32 *sizeInWords*,
ADT_L0_UINT32 * *pNumVectors*)

This function adds a 1553 word to a list of vectors.

Parameters:

m1553word - 16-bit 1553 word (bit 31 is SYNC (1=cmd sync, 0 = data sync), bit 30 is PARITY (1=parity error), bits 16-29 not used, set to zero).
pVectors - pointer to the array of 32-bit words containing vectors.
sizeInWords - max size (in words) of the vector array.
pNumVectors - pointer to a UINT32 containing the current vector count (in 2-bit 20ns vectors).

Returns:

ADT_SUCCESS - Completed without error
ADT_FAILURE - Completed with error - not enough words available in vectors array

1553 Playback Functions

This section describes the Layer 1 API 1553 Playback functions. These functions are defined in the file ADT_L1_1553_PB.c.

Note that playback is intended for “normal” messages without protocol errors – it is difficult to accurately playback messages with errors. Abnormal conditions, like broadcast RTBC messages for example (things like this occur in AS4111 RT validation testing), may be discarded as invalid CDP records and the message will be skipped by playback. **Messages with errors or abnormal conditions may play back inaccurately or may be skipped altogether.**

ADT_L1_1553_PB_Allocate

ADT_L0_UINT32 ADT_L1_1553_PB_Allocate (ADT_L0_UINT32 *devID*,
 ADT_L0_UINT32 *numPkts*)

This function allocates and links playback packets.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).
numPkts - number of playback packets to allocate.

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_BAD_INPUT - Invalid number of packets
ADT_FAILURE - Completed with error

ADT_L1_1553_PB_CDPWrite

ADT_L0_UINT32 ADT_L1_1553_PB_CDPWrite (ADT_L0_UINT32 *devID*,
 ADT_L1_1553_CDP * *pCdp*,
 ADT_L0_UINT32 *options*,
 ADT_L0_UINT32 *isFirstMsg*)

This function converts a CDP to a PB packet and writes it to the PB buffer.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).
pCdp - pointer to the CDP to write to the PB packet.
Options - packet control word options:

ADT_L1_1553_PBP_CONTROL_STOP	0x00000200
ADT_L1_1553_PBP_CONTROL_LED	0x00000100
ADT_L1_1553_PBP_CONTROL_TRGOUT	0x00000080
ADT_L1_1553_PBP_CONTROL_INT	0x00000040
ADT_L1_1553_API_PB_CDPWRITE_ATON	0x80000000

isFirstMsg - 1 for first message in playback or 0 for NOT first message. See the section on “1553 Playback Operation” for details on playback functions and options.

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_BUFFER_FULL - PB buffer is full
ADT_ERR_BAD_INPUT - Invalid CDP
ADT_FAILURE - Completed with error

ADT L1 1553 PB Free

ADT L0 UINT32 ADT L1 1553 PB Free (ADT L0 UINT32 devID)

This function frees all playback packets for the device.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).

Returns:

ADT_SUCCESS - Completed without error
ADT_FAILURE - Completed with error

ADT L1 1553 PB GetRtResponse

This function gets the playback RT response word.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).
pRtResp - pointer to store the RT response word.

Returns:

ADT_SUCCESS - Completed without error
ADT_FAILURE - Completed with error

ADT_L1_1553_PB_IsRunning

ADT_L0_UINT32 ADT_L1_1553_PB_IsRunning (ADT_L0_UINT32 *devID*,
 ADT_L0_UINT32 * *pIsRunning*)

This function determines if playback is running or stopped.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).
pIsRunning - pointer to store the result, 0=NotRunning, 1=Running.

Returns:

ADT_SUCCESS - Completed without error
ADT_FAILURE - Completed with error

ADT_L1_1553_PB_SetRtResponse

ADT_L0_UINT32 ADT_L1_1553_PB_SetRtResponse (ADT_L0_UINT32 *devID*,
 ADT_L0_UINT32 *rtResp*)

This function sets the playback RT response word.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).
rtResp - RT response word - each bit corresponds to an RT, if set then playback status word.

Returns:

ADT_SUCCESS - Completed without error
ADT_FAILURE - Completed with error

ADT_L1_1553_PB_Start

ADT_L0_UINT32 ADT_L1_1553_PB_Start (ADT_L0_UINT32 *devID*)

This function starts Playback operation for the device.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).

Returns:

ADT_SUCCESS - Completed without error
ADT_FAILURE - Completed with error

ADT_L1_1553_PB_Stop

ADT_L0_UINT32 ADT_L1_1553_PB_Stop (ADT_L0_UINT32 *devID*)

This function stops Playback operation for the device.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).

Returns:

ADT_SUCCESS - Completed without error
ADT_FAILURE - Completed with error

A429 General Functions

This section describes the Layer 1 API A429 General functions. These functions are defined in the file ADT_L1_A429_General.c.

ADT_L1_A429_InitDefault

ADT_L0_UINT32 ADT_L1_A429_InitDefault (ADT_L0_UINT32 *devID*,
 ADT_L0_UINT32 *numIQEntries*)

This function initializes an A429 device, allocates interrupt queue, and sets default configuration for ARINC 429 protocol. This function calls ADT_L1_InitDevice and ADT_L1_A429_InitDevice.

Also see the next function, ADT_L1_A429_InitDefault_ExtenedOptions. This function provides more options for startup of the device channel for resets and forcing API initialization, which may be useful for development and embedded applications.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Bank #).
numIQEntries - number of interrupt queue entries to allocate.

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_BAD_INPUT - Invalid number of IQ entries
ADT_ERR_BITFAIL - Failed Built-In Test
ADT_ERR_DEVICEINUSE - Device in use (see explanation in the Layer 1 API Operational Discussion under “Initializing and Closing the API”)
ADT_FAILURE - Completed with error

ADT_L1_A429_InitDefault_ExtendedOptions

```
ADT_L0_UINT32 ADT_L1_A429_InitDefault_ExtendedOptions  
    (ADT_L0_UINT32 devID,  
     ADT_L0_UINT32 numIQEntries,  
     ADT_L0_UINT32 startupOptions)
```

This function initializes an A429 device, allocates interrupt queue, and sets default configuration for ARINC 429 protocol. This function calls ADT_L1_InitDevice and ADT_L1_A429_InitDevice.

Additional startup options are provided to allow the user to specify to skip Force API initialization (need for previous soft application crashes), skip Memory Test (for fast startups), and perform a hard reset on the A429 bank (not the card – this clears all channel protocol engine registers). The ADT_L1_API_DEVICEINIT_NOKP option can be used to bypass loading the driver kernel plug-in but in most cases this option should NOT be set.

This function may be preferred over ADT_L1_A429_InitDefault where the user always wants the A429 bank to initialize regardless of previous application or hardware states.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Bank #).

numIQEntries - number of interrupt queue entries to allocate.

startupOptions - user selected startup options defined as follows:

```
#define ADT_L1_API_DEVICEINIT_FORCEINIT      0x00000001  
#define ADT_L1_API_DEVICEINIT_NOMEMTEST       0x00000002  
#define ADT_L1_API_DEVICEINIT_NOKP            0x00000004  
#define ADT_L1_API_DEVICEINIT_ROOTPERESET     0x80000000
```

The ADT_L1_API_DEVICEINIT_FORCEINIT option should ONLY be used in development and testing. This option is provided for cases where the device may not have been closed properly and is used to override the ADT_ERR_DEVICEINUSE error. This option should NOT be used as the normal initialization method for your application, because it bypasses protection against two applications using the same device.

The ADT_L1_API_DEVICEINIT_NOMEMTEST option skips the default memory test on initialization.

The ADT_L1_API_DEVICEINIT_ROOTPERESET option resets the protocol engine and firmware to power on state upon initialization.

The ADT_L1_API_DEVICEINIT_NOKP option can be used to bypass loading the driver kernel plug-in but in most cases this option should NOT be set. This option only applies to platforms that use the Jungo WinDriver software for the device driver (Windows, Linux, Solaris). If this option is used then the application cannot use hardware interrupts. The kernel plug-in is required for hardware interrupts.

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_BAD_INPUT - Invalid number of IQ entries
ADT_ERR_BAD_CHAN_NUM - Bad channel number or channel does not exist
ADT_ERR_BITFAIL - Failed Built-In Test
ADT_FAILURE - Completed with error

ADT_L1_A429_InitDevice

ADT_L0_UINT32 ADT_L1_A429_InitDevice (ADT_L0_UINT32 *devID*,
 ADT_L0_UINT32 *numIQEntries*)

This function initializes an A429 device and allocates the interrupt queue.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Bank #).
numIQEntries - number of interrupt queue entries to allocate.

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_BAD_INPUT - Invalid number of IQ entries
ADT_ERR_BITFAIL - Failed Built-In Test
ADT_FAILURE - Completed with error

ADT_L1_A429_GetConfig

ADT_L0_UINT32 ADT_L1_A429_GetConfig (ADT_L0_UINT32 *devID*,
 ADT_L0_UINT32 **pChanConfig*)

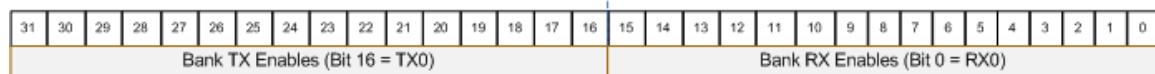
This function gets the TX-RX channel configuration for an A429 device.

This RX and TX configuration for the bank device of channels is located at Root PE Offset from ADT_L1.h:

```
#define ADT_L1_A429_PE_TXRX_CHANCONFIG      0x000C
```

Here is the Data Structure from the AltaCore ARINC manual:

Root PE Register 0x000C



*Note: The number of Tx and Rx Channels Varies Depending on Order Configuration – See Your Getting Started and Hardware Manual for Details on Configurations and Pin-Outs.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Bank #).
pChanConfig - pointer for the TX-RX channel selection.

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_UNSUPPORTED_CHANNELTYPE - Not an A429 device
ADT_FAILURE - Completed with error

ADT_L1_A429_GetPEInfo

ADT_L0_UINT32 ADT_L1_A429_GetPEInfo (ADT_L0_UINT32 *devID*,
 ADT_L0_UINT32 **peInfo*)

This function gets A429 PE ID and Version information. Also See:
ADT_L1_GetVersionInfo().

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).
peInfo - pointer to store the PE ID and Version.

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_UNSUPPORTED_CHANNELTYPE - Not an A429 device
ADT_FAILURE - Completed with error

ADT_L1_A429_TimeClear

ADT_L0_UINT32 ADT_L1_A429_TimeClear (ADT_L0_UINT32 *devID*)

This function clears the time tag value for the device.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Bank #).

Returns:

ADT_SUCCESS - Completed without error

ADT_ERR_TIMEOUT - Timeout waiting for CSR bit to clear

ADT_ERR_UNSUPPORTED_CHANNELTYPE - Not an A429 device

ADT_FAILURE - Completed with error

ADT_L1_A429_UseExtClk

ADT_L0_UINT32 ADT_L1_A429_UseExtClk (ADT_L0_UINT32 *devID*,

 ADT_L0_UINT32 *useExtClk*,

 ADT_L0_UINT32 *clkFreq*)

This function enables an A429 bank to use an external clock.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Bank #).

useExtClk - enables or disables external clock (1=enable, 0=disable).

clkFreq - must be 1, 5, or 10 (1MHz, 5MHz, or 10MHz).

Returns:

ADT_SUCCESS - Completed without error

ADT_ERR_BAD_INPUT - Invalid value for *clkFreq*

ADT_FAILURE - Completed with error

ADT_L1_A429_TimeGet

```
ADT_L0_UINT32 ADT_L1_A429_TimeGet (ADT_L0_UINT32 devID,  
                                     ADT_L0_UINT32 * pTimeHigh,  
                                     ADT_L0_UINT32 * pTimeLow)
```

This function gets the time tag value for the device.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Bank #).
pTimeHigh - pointer to store the upper 32-bits of the 64-bit time tag value.
pTimeLow - pointer to store the lower 32-bits of the 64-bit time tag value.

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_TIMEOUT - Timeout waiting for CSR bit to clear
ADT_ERR_UNSUPPORTED_CHANNELTYPE - Not an A429 device
ADT_FAILURE - Completed with error

ADT_L1_A429_TimeSet

```
ADT_L0_UINT32 ADT_L1_A429_TimeSet (ADT_L0_UINT32 devID,  
                                     ADT_L0_UINT32 timeHigh,  
                                     ADT_L0_UINT32 timeLow)
```

This function sets the time tag value for the device.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Bank #).
timeHigh - upper 32-bits of the 64-bit time tag value.
timeLow - lower 32-bits of the 64-bit time tag value.

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_UNSUPPORTED_CHANNELTYPE - Not an A429 device
ADT_FAILURE - Completed with error

ADT_L1_A429_IrigLatchedTimeGet

```
ADT_L0_UINT32 ADT_L1_A429_IrigLatchedTimeGet (ADT_L0_UINT32 devID,  
                                              ADT_L0_UINT32 * pIrigTimeHigh, ADT_L0_UINT32 * pIrigTimeLow,  
                                              ADT_L0_UINT32 * plntTimeHigh, ADT_L0_UINT32 * plntTimeLow,  
                                              ADT_L0_UINT32 * pDeltaTimeHigh, ADT_L0_UINT32 * pDeltaTimeLow)
```

If the device has locked onto a valid IRIG signal this function gets the latched IRIG and internal time values for the channel. If IRIG LOCK is not present, ADT_FAILURE is returned. These values are latched by the firmware on every 1 second IRIG sync. Use the ADT_L1_Global_CalibrateIrigDac function to calibrate to the IRIG signal.

The latched IRIG time is in IRIG BCD format, as shown below:

IRIG Time High – 0x000000C8																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved												BCD Years										BCD Days									

IRIG Time Low – 0x000000CC																																
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
Reserved												BCD Hours										BCD Minutes										BCD Seconds

NOTE:

IRIG Standard 200-98 – The IRIG-B time code does not include years.

IRIG Standard 200-04 – The IRIG-B time code can include years.

The latched internal time is in 64-bit binary format (not BCD), with a 20ns LSB. The delta time is the difference between the IRIG time (converted from BCD to binary) and the internal time. Add the delta time to internal time values (like the time stamp on a RXP) to convert them to binary IRIG time.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).
pIrigTimeHigh - pointer to store the upper 32-bits of the 64-bit IRIG BCD time (latched).
pIrigTimeLow - pointer to store the lower 32-bits of the 64-bit IRIG BCD time (latched).
plntTimeHigh - pointer to store the upper 32-bits of the 64-bit internal time (latched).
plntTimeLow - pointer to store the lower 32-bits of the 64-bit internal time (latched).
pDeltaTimeHigh - pointer to store the upper 32-bits of the 64-bit delta time.
pDeltaTimeLow - pointer to store the lower 32-bits of the 64-bit delta time.

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_TIMEOUT - Timeout waiting for CSR bit to clear
ADT_FAILURE - Completed with error

ADT_L1_A429_PBTimeGet

```
ADT_L0_UINT32 ADT_L1_A429_PBTimeGet (ADT_L0_UINT32 devID,  
                                     ADT_L0_UINT32 * pTimeHigh,  
                                     ADT_L0_UINT32 * pTimeLow)
```

This function gets the Playback time tag value for the device.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Bank #).
pTimeHigh - pointer to store the upper 32-bits of the 64-bit time tag value.
pTimeLow - pointer to store the lower 32-bits of the 64-bit time tag value.

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_TIMEOUT - Timeout waiting for CSR bit to clear
ADT_ERR_UNSUPPORTED_CHANNELTYPE - Not an A429 device
ADT_FAILURE - Completed with error

ADT_L1_A429_PBTimeSet

```
ADT_L0_UINT32 ADT_L1_A429_PBTimeSet (ADT_L0_UINT32 devID,  
                                      ADT_L0_UINT32 timeHigh,  
                                      ADT_L0_UINT32 timeLow)
```

This function sets the Playback time tag value for the device.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Bank #).
timeHigh - upper 32-bits of the 64-bit time tag value.
timeLow - lower 32-bits of the 64-bit time tag value.

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_UNSUPPORTED_CHANNELTYPE - Not an A429 device
ADT_FAILURE - Completed with error

ADT_L1_A429_SC_ArmTrigger

ADT_L0_UINT32 ADT_L1_A429_SC_ArmTrigger (ADT_L0_UINT32 *devID*,
 ADT_L0_UINT32 *rxChan*)

This function arms the Signal Capture for a trigger.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Bank #).
rxChan - which receive channel to use (zero for channel 1, non-zero for channel 2).

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_UNSUPPORTED_CHANNELTYPE - Not an A429 device
ADT_FAILURE - Completed with error

ADT_L1_A429_SC_ReadBuffer

ADT_L0_UINT32 ADT_L1_A429_SC_ReadBuffer (ADT_L0_UINT32 *devID*,
 ADT_L0_UINT32 *rxChan*,
 ADT_L0_UINT8 **buffer*)

This function reads the Signal Capture data buffer for the specified receive channel.

This reads 4096 unsigned 8-bit values.

Each value represents a voltage where **0 is -11.264 volts** and **255 is +11.264 volts**. Therefore the voltage range is 22.528 volts over 256 possible values, or 0.08835 volts per LSB.

Each of the 4096 values represents a time period of 1 microsecond. Therefore the entire buffer represents 4096 microseconds of time.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Bank #).
rxChan - which receive channel to use (zero for channel 1, non-zero for channel 2).
buffer - array of 4096 unsigned 8-bit values.

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_UNSUPPORTED_CHANNELTYPE - Not an A429 device
ADT_FAILURE - Completed with error

ADT_L1_A429_IntervalTimerGet

```
ADT_L0_UINT32 ADT_L1_A429_IntervalTimerGet(ADT_L0_UINT32 devID,  
                                         ADT_L0_UINT32 *pIntvlTmrReg);
```

This function retrieves the Root PE Interval Timer Register. The register is comprised of 24 MSB, 1 usec time value and 5 control and status bits in the 8 LSB. The following provides the ADT_L1.h definitions:

```
#define ADT_L1_A429_PE_INTVLTMR      0x004C  
#define     ADT_L1_A429_PE_INTVLTMR_STARTSTOP 0x00000001  
#define     ADT_L1_A429_PE_INTVLTMR_STARTONTRIG 0x00000002  
#define     ADT_L1_A429_PE_INTVLTMR_EXTTRIG   0x00000008  
#define     ADT_L1_A429_PE_INTVLTMR_SETINT    0x00000010  
#define     ADT_L1_A429_PE_INTVLTMR_TIMECOMP  0x00000080  
#define     ADT_L1_A429_PE_INTVLTMR_TIME24   0xFFFFFFF00  
#define     ADT_L1_A429_PE_INTVLTMR_TIMESHIFT 8
```

Also see the AltaCore-A429 manual for more detail definitions and actions of the register bits.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Bank #).
pIntvlTmrReg - pointer to the register value result.

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_BAD_INPUT - Not an A429 device
ADT_FAILURE - Completed with error

ADT_L1_A429_IntervalTimerSet

```
ADT_L0_UINT32 ADT_L1_A429_IntervalTimerGet(ADT_L0_UINT32 devID,
                                            ADT_L0_UINT32 intvltmrReg);
```

This function writes a user value to the Root PE Interval Timer Register. The register is comprised of 24 MSB, 1 usec time value and 5 control and status bits in the 8 LSB. The following provides the ADT_L1.h definitions:

```
#define ADT_L1_A429_PE_INTVLTMR          0x004C
#define      ADT_L1_A429_PE_INTVLTMR_STARTSTOP 0x00000001
#define      ADT_L1_A429_PE_INTVLTMR_STARTONTRIG 0x00000002
#define      ADT_L1_A429_PE_INTVLTMR_EXTTRIG   0x00000008
#define      ADT_L1_A429_PE_INTVLTMR_SETINT    0x00000010
#define      ADT_L1_A429_PE_INTVLTMR_TIMECOMP   0x00000080
#define      ADT_L1_A429_PE_INTVLTMR_TIME24    0xFFFFFFF00
#define      ADT_L1_A429_PE_INTVLTMR_TIMESHIFT  8
```

Also see the AltaCore-A429 manual for more detail definitions and actions of the register bits.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Bank #).
intvltmrReg - user value to write to the Interval Register.

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_BAD_INPUT - Not an A429 device
ADT_FAILURE - Completed with error

A429 Interrupt Functions

This section describes the Layer 1 API A429 Interrupt functions. These functions are defined in the file ADT_L1_A429_INT.c.

ADT_L1_A429_INT_CheckChannellntPending

ADT_L0_UINT32 ADT_L1_A429_INT_CheckChannellntPending
(ADT_L0_UINT32 *devID*,
ADT_L0_UINT32 * *pIsIntPending*)

This function checks to see if an interrupt is pending for a given A429 device.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Bank #).
pIsIntPending - pointer to store the result (0 = no int pending, 1 = int pending).

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_BAD_INPUT - Invalid channel number
ADT_ERR_UNSUPPORTED_CHANNELTYPE - Not an A429 device
ADT_FAILURE - Completed with error

ADT_L1_A429_INT_DisableInt

ADT_L0_UINT32 ADT_L1_A429_INT_DisableInt (ADT_L0_UINT32 *devID*)

This function disables interrupts for the device.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Bank #).

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_UNSUPPORTED_CHANNELTYPE - Not an A429 device
ADT_FAILURE - Completed with error

ADT_L1_A429_INT_EnableInt

ADT_L0_UINT32 ADT_L1_A429_INT_EnableInt (ADT_L0_UINT32 *devID*)

This function enables interrupts for the channel.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Bank #).

Returns:

ADT_SUCCESS - Completed without error

ADT_ERR_UNSUPPORTED_CHANNELTYPE - Not an A429 device

ADT_FAILURE - Completed with error

ADT_L1_A429_INT_GetIntSeqNum

ADT_L0_UINT32 ADT_L1_A429_INT_GetIntSeqNum (ADT_L0_UINT32 *devID*,
ADT_L0_UINT32 * *pSeqNum*)

This function reads the interrupt sequence number for the device.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Bank #).

pSeqNum - pointer to store the sequence number.

Returns:

ADT_SUCCESS - Completed without error

ADT_ERR_UNSUPPORTED_CHANNELTYPE - Not an A429 device

ADT_FAILURE - Completed with error

ADT_L1_A429_INT_IQ_ReadEntry

```
ADT_L0_UINT32 ADT_L1_A429_INT_IQ_ReadEntry (ADT_L0_UINT32 devID,  
                                              ADT_L0_UINT32 * pType,  
                                              ADT_L0_UINT32 * pInfo)
```

This function reads one new entry from the interrupt queue.

For more information on interrupt queue entries refer to the “Layer 1 API Operational Discussion” section, under “Interrupt Operation” in the section for “A429 Device Interrupt Functions”.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Bank #).
pType - pointer to store the interrupt type word.
pInfo - pointer to store the interrupt info word.

Returns:

ADT_SUCCESS - Completed without error
ADT_FAILURE - Completed with error
ADT_ERR_IQ_NO_NEW_ENTRY - No new entry in the queue

ADT_L1_A429_INT_IQ_ReadNewEntries

ADT_L0_UINT32 ADT_L1_A429_INT_IQ_ReadNewEntries
(ADT_L0_UINT32 *devID*,
ADT_L0_UINT32 *MaxNumEntries*,
ADT_L0_UINT32 * *pNumEntries*,
ADT_L0_UINT32 * *pType*,
ADT_L0_UINT32 * *pInfo*)

This function reads all new entries from the interrupt queue.

For more information on interrupt queue entries refer to the “Layer 1 API Operational Discussion” section, under “Interrupt Operation” in the section for “A429 Device Interrupt Functions”.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Bank #).
maxNumEntries - maximum number of entries to read (size of buffer).
pNumEntries - pointer to store the number of messages read.
pType - pointer to store the interrupt type words (array sized by maxNumEntries).
pInfo - pointer to store the interrupt info words (array sized by maxNumEntries).

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_BAD_INPUT - invalid pointer
ADT_FAILURE - Completed with error

ADT_L1_A429_INT_IQ_ReadRawEntry

This function reads one new raw entry from the interrupt queue. ADT_L1_A429_INT has the following structure:

```
typedef struct adt_l1_a429_int {
    ADT_L0_UINT32 NextPtr;
    ADT_L0_UINT32 Type_SeqNum;
    ADT_L0_UINT32 IntData;
} ADT_L1_A429_INT;
```

NextPtr contains a pointer to the next interrupt queue entry. Type_SeqNum is the “IQP Type and Sequence Number” word from the Interrupt Queue Packet, where the upper 16-bits provide the interrupt type and the lower 16-bits provide a sequence number. IntData is a pointer to the data structure that caused the interrupt.

For more information on interrupt queue entries refer to the “Layer 1 API Operational Discussion” section, under “Interrupt Operation” in the section for “A429 Device Interrupt Functions”.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Bank #).
int_buffer - pointer to the ADT_L1_A429_INT data buffer type where the interrupt info is stored.

Returns:

ADT_SUCCESS - Completed without error
ADT_FAILURE - Completed with error
ADT_ERR_IQ_NO_NEW_ENTRY - No new entry in the queue

ADT_L1_A429_INT_IQ_ReadNewRawEntries

```
ADT_L0_UINT32 ADT_L1_A429_INT_IQ_ReadNewEntries  
    (ADT_L0_UINT32 devID,  
     ADT_L0_UINT32 MaxNumEntries,  
     ADT_L0_UINT32 * pNumEntries,  
     ADT_L1_A429_INT * intbuffer)
```

This function reads all new entries from the interrupt queue into an array of type ADT_L1_A429_INT. ADT_L1_A429_INT has the following structure:

```
typedef struct adt_l1_a429_int {  
    ADT_L0_UINT32 NextPtr;  
    ADT_L0_UINT32 Type_SeqNum;  
    ADT_L0_UINT32 IntData;  
} ADT_L1_A429_INT;
```

NextPtr contains a pointer to the next interrupt queue entry. Type_SeqNum is the “IQP Type and Sequence Number” word from the Interrupt Queue Packet, where the upper 16-bits provide the interrupt type and the lower 16-bits provide a sequence number. IntData is a pointer to the data structure that caused the interrupt.

For more information on interrupt queue entries refer to the “Layer 1 API Operational Discussion” section, under “Interrupt Operation” in the section for “A429 Device Interrupt Functions”.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Bank #).
maxNumEntries - maximum number of entries to read (size of buffer).
pNumEntries - pointer to store the number of messages read.
intbuffer - pointer to store the raw interrupt data array (array sized by maxNumEntries).

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_BAD_INPUT - invalid pointer
ADT_FAILURE - Completed with error

ADT_L1_A429_INT_SetIntSeqNum

ADT_L0_UINT32 ADT_L1_A429_INT_SetIntSeqNum (ADT_L0_UINT32 *devID*,
ADT_L0_UINT32 *seqNum*)

This function writes the interrupt sequence number for the device.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Bank #).
seqNum - sequence number to write.

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_UNSUPPORTED_CHANNELTYPE - Not an A429 device
ADT_FAILURE - Completed with error

A429 Multichannel Receive Functions

This section describes the Layer 1 API A429 Multichannel Receive functions. These functions are defined in the file ADT_L1_A429_RX_MC.c.

ADT_L1_A429_RXMC_BufferCreate

ADT_L0_UINT32 ADT_L1_A429_RXMC_BufferCreate (ADT_L0_UINT32 *devID*,
 ADT_L0_UINT32 numRxP)

This function allocates the requested number of RxPs for the multichannel receive buffer.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Bank #).
numRxP - number of RxP buffers requested.

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_UNSUPPORTED_CHANNELTYPE - Not an A429 device
ADT_ERR_BAD_INPUT - Invalid number of RxP
ADT_FAILURE - Completed with error

ADT_L1_A429_RXMC_BufferFree

ADT_L0_UINT32 ADT_L1_A429_RXMC_BufferFree (ADT_L0_UINT32 *devID*)

This function frees all board memory used for the multichannel receive buffer.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Bank #).

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_UNSUPPORTED_CHANNELTYPE - Not an A429 device
ADT_FAILURE - Completed with error

ADT_L1_A429_RXMC_ReadNewRxPs

```
ADT_L0_UINT32 ADT_L1_A429_RXMC_ReadNewRxPs  
    (ADT_L0_UINT32 devID,  
     ADT_L0_UINT32 maxNumRxPs,  
     ADT_L0_UINT32 * pNumRxPs,  
     ADT_L1_A429_RXP * pRxPBuffer)
```

This function reads all new RxPs from the multichannel RxP buffer.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Bank #).
maxNumRxPs - maximum number of RxPs to read (size of buffer).
pNumRxPs - pointer to store the number of RxPs read.
pRxPBuffer - pointer to store the RxP records.

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_BAD_INPUT - invalid pointer
ADT_ERR_UNSUPPORTED_CHANNELTYPE - Not an A429 device
ADT_FAILURE - Completed with error

ADT_L1_A429_RXMC_ReadNewRxPsDMA

```
ADT_L0_UINT32 ADT_L1_A429_RXMC_ReadNewRxPs  
    (ADT_L0_UINT32 devID,  
     ADT_L0_UINT32 maxNumRxPs,  
     ADT_L0_UINT32 * pNumRxPs,  
     ADT_L1_A429_RXP * pRxPBuffer)
```

This function reads all new RxPs from the multichannel RxP buffer using DMA.

NOTE THAT ONLY SELECTED LAYER 0 SUPPORT DMA (currently Windows and Linux) AND ONLY SELECTED BOARD TYPES SUPPORT DMA.

PCI devices with PLX9056 – PMC, PC104P, PCI

NOT supported on PCCD products

PCIE devices with A429 PE version 0406 or later

This function is not multiple-application/thread safe. This function should only be used if one thread controls all channels on a given board that are using DMA.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Bank #).
maxNumRxPs - maximum number of RxPs to read (size of buffer).
pNumRxPs - pointer to store the number of RxPs read.
pRxPBuffer - pointer to store the RxP records.

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_BAD_INPUT - invalid pointer
ADT_ERR_UNSUPPORTED_CHANNELTYPE - Not an A429 device
ADT_FAILURE - Completed with error

ADT_L1_A429_RXMC_ReadRxP

ADT_L0_UINT32 ADT_L1_A429_RXMC_ReadRxP
(*ADT_L0_UINT32 devID,*
 ADT_L0_UINT32 RxP_index,
 *ADT_L1_A429_RXP * pRxP*)

This function reads a specific RxP from the multichannel RxP buffer.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Bank #).
RxP_index - index of the RxP to read.
pRxP - pointer to store the RxP record.

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_BAD_INPUT - invalid pointer
ADT_ERR_UNSUPPORTED_CHANNELTYPE - Not an A429 device
ADT_FAILURE - Completed with error

ADT_L1_A429_RXMC_WriteRxP

```
ADT_L0_UINT32 ADT_L1_A429_RXMC_WriteRxP  
          (ADT_L0_UINT32 devID,  
           ADT_L0_UINT32 RxP_index,  
           ADT_L1_A429_RXP * pRxP)
```

This function writes a specific RxP to the multichannel RxP buffer.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Bank #).
RxP_index - index of the RxP to write.
pRxP - pointer to the RxP to write.

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_BAD_INPUT - invalid pointer
ADT_ERR_UNSUPPORTED_CHANNELTYPE - Not an A429 device
ADT_FAILURE - Completed with error

A429 Receive Functions

This section describes the Layer 1 API A429 Receive functions. These functions are defined in the file ADT_L1_A429_RX.c.

ADT_L1_A429_RX_Channel_Init

```
ADT_L0_UINT32 ADT_L1_A429_RX_Channel_Init (ADT_L0_UINT32 devID,  
                                         ADT_L0_UINT32 RxChanNum,  
                                         ADT_L0_UINT32 BitRateHz,  
                                         ADT_L0_UINT32 numRxP,  
                                         ADT_L0_UINT32 includeInMcRx)
```

This function initializes data structures for the specified receive channel.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Bank #).
RxChanNum - receive channel number (0-15).
BitRateHz - bit rate in Hz (500-500000).
numRxP - number of receive packets to allocate for the channel.
includeInMcRx - indicates whether or not to include this channel in the multichannel receive buffer. If this value is non-zero, then this channel is included in the MCRX buffer. If this value is zero, then this channel is not included in the MCRX buffer.

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_BAD_INPUT - Invalid channel number or invalid bit rate
ADT_ERR_UNSUPPORTED_CHANNELTYPE - Not an A429 device
ADT_FAILURE - Completed with error

ADT_L1_A429_RX_Channel_Close

ADT_L0_UINT32 ADT_L1_A429_RX_Channel_Close (ADT_L0_UINT32 *devID*,
 ADT_L0_UINT32 *RxChanNum*)

This function clears data structures and frees memory used by the specified receive channel.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).
RxChanNum - receive channel number (0-15).

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_BAD_INPUT - Invalid channel number
ADT_ERR_UNSUPPORTED_CHANNELTYPE - Not an A429 device
ADT_FAILURE - Completed with error

ADT_L1_A429_RX_Channel_Start

ADT_L0_UINT32 ADT_L1_A429_RX_Channel_Start (ADT_L0_UINT32 *devID*,
 ADT_L0_UINT32 *RxChanNum*)

This function starts receive operation for the specified receive channel.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Bank #).
RxChanNum - receive channel number (0-15).

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_BAD_INPUT - Invalid channel number
ADT_ERR_UNSUPPORTED_CHANNELTYPE - Not an A429 device
ADT_FAILURE - Completed with error

ADT_L1_A429_RX_Channel_Stop

ADT_L0_UINT32 ADT_L1_A429_RX_Channel_Stop (ADT_L0_UINT32 *devID*,
ADT_L0_UINT32 *RxChanNum*)

This function stops receive operation for the specified receive channel.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Bank #).
RxChanNum - receive channel number (0-15).

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_BAD_INPUT - Invalid channel number
ADT_ERR_UNSUPPORTED_CHANNELTYPE - Not an A429 device
ADT_FAILURE - Completed with error

ADT_L1_A429_RX_Channel_ReadNewRxPs

ADT_L0_UINT32 ADT_L1_A429_RX_Channel_ReadNewRxPs (
 ADT_L0_UINT32 *devID*,
 ADT_L0_UINT32 *RxChanNum*,
 ADT_L0_UINT32 *maxNumRxPs*,
 ADT_L0_UINT32 **pNumRxPs*,
 ADT_L1_A429_RXP **pRxPBuffer*)

This function reads all new receive packets for the specified receive channel.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).
RxChanNum - receive channel number (0-15).
maxNumRxPs - maximum number of receive packets to read (size of RxP buffer).
pNumRxPs - pointer to store the number of receive packets read.
pRxPBuffer - pointer to the array of receive packets.

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_BAD_INPUT - Invalid channel number
ADT_ERR_UNSUPPORTED_CHANNELTYPE - Not an A429 device
ADT_ERR_BAD_CHAN_NUM - The RX channel has not been initialized
ADT_FAILURE - Completed with error

ADT_L1_A429_RX_Channel_ReadNewRxPsDMA

```
ADT_L0_UINT32 ADT_L1_A429_RX_Channel_ReadNewRxPs (  
    ADT_L0_UINT32 devID,  
    ADT_L0_UINT32 RxChanNum,  
    ADT_L0_UINT32 maxNumRxPs,  
    ADT_L0_UINT32 *pNumRxPs,  
    ADT_L1_A429_RXP *pRxPBuffer)
```

This function reads all new receive packets for the specified receive channel using DMA.

NOTE THAT ONLY SELECTED LAYER 0 SUPPORT DMA (currently Windows and Linux) AND ONLY SELECTED BOARD TYPES SUPPORT DMA.

PCI devices with PLX9056 – PMC, PC104P, PCI

NOT supported on PCCD products

PCIE devices with A429 PE version 0406 or later

This function is not multiple-application/thread safe. This function should only be used if one thread controls all channels on a given board that are using DMA.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).

RxChanNum - receive channel number (0-15).

maxNumRxPs - maximum number of receive packets to read (size of RxP buffer).

pNumRxPs - pointer to store the number of receive packets read.

pRxPBuffer - pointer to the array of receive packets.

Returns:

ADT_SUCCESS - Completed without error

ADT_ERR_BAD_INPUT - Invalid channel number

ADT_ERR_UNSUPPORTED_CHANNELTYPE - Not an A429 device

ADT_ERR_BAD_CHAN_NUM - The RX channel has not been initialized

ADT_FAILURE - Completed with error

ADT_L1_A429_RX_Channel_ReadRxP

```
ADT_L0_UINT32 ADT_L1_A429_RX_Channel_ReadRxP (
    ADT_L0_UINT32 devID,
    ADT_L0_UINT32 RxChanNum,
    ADT_L0_UINT32 RxP_index,
    ADT_L1_A429_RXP *pRxP)
```

This function reads a specific receive packet for the specified receive channel.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Bank #).
RxChanNum - receive channel number (0-15).
RxP_index - index (0-N) of the receive packet to read.
pRxP - pointer to store the receive packet.

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_BAD_INPUT - Invalid channel number
ADT_ERR_UNSUPPORTED_CHANNELTYPE - Not an A429 device
ADT_ERR_BAD_CHAN_NUM – The RX channel has not been initialized
ADT_FAILURE - Completed with error

ADT_L1_A429_RX_Channel_WriteRxP

```
ADT_L0_UINT32 ADT_L1_A429_RX_Channel_WriteRxP (
    ADT_L0_UINT32 devID,
    ADT_L0_UINT32 RxChanNum,
    ADT_L0_UINT32 RxP_index,
    ADT_L1_A429_RXP *pRxP)
```

This function writes a specific receive packet to the specified receive channel.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Bank #).
RxChanNum - receive channel number (0-15).
RxP_index - index (0-N) of the receive packet to write.
pRxP - pointer to the receive packet to write.

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_BAD_INPUT - Invalid channel number
ADT_ERR_UNSUPPORTED_CHANNELTYPE - Not an A429 device
ADT_ERR_BAD_CHAN_NUM – The RX channel has not been initialized
ADT_FAILURE - Completed with error

ADT_L1_A429_RX_Channel_CVTReadRxP

```
ADT_L0_UINT32 ADT_L1_A429_RX_Channel_ReadRxP (
    ADT_L0_UINT32 devID,
    ADT_L0_UINT32 RxChanNum,
    ADT_L0_UINT32 labelIndex,
    ADT_L1_A429_RXP *pRxP)
```

This function reads a specific receive packet for the specified receive channel in the Current Value Table that is label indexed. The user will need to reverse the labelIndex number prior to making this call.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Bank #).
RxChanNum - receive channel number (0-15).
labelIndex - raw (MSB first) label number (first 8 bits) of the word/label.
pRxP - pointer to store the receive packet.

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_BAD_INPUT - Invalid channel number
ADT_ERR_UNSUPPORTED_CHANNELTYPE - Not an A429 device
ADT_FAILURE - Completed with error

ADT_L1_A429_RX_Channel_CVTWriteRxP

```
ADT_L0_UINT32 ADT_L1_A429_RX_Channel_CVTWriteRxP (
    ADT_L0_UINT32 devID,
    ADT_L0_UINT32 RxChanNum,
    ADT_L0_UINT32 labelIndex,
    ADT_L1_A429_RXP *pRxP)
```

This function writes a specific receive packet to the specified receive channel channel in the Current Value Table that is label indexed.. The user will need to reverse the labelIndex number prior to making this call.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Bank #).
RxChanNum - receive channel number (0-15).
labelIndex - raw (MSB first) label number (first 8 bits) of the word/label.
pRxP - pointer to the receive packet to write.

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_BAD_INPUT - Invalid channel number
ADT_ERR_UNSUPPORTED_CHANNELTYPE - Not an A429 device
ADT_FAILURE - Completed with error

ADT_L1_A429_RX_Channel_SetConfig

```
ADT_L0_UINT32 ADT_L1_A429_RX_Channel_SetConfig (
    ADT_L0_UINT32 devID,
    ADT_L0_UINT32 RxChanNum,
    ADT_L0_UINT32 Setup1,
    ADT_L0_UINT32 Setup2)
```

This function sets the protocol settings for the specified receive channel. This function writes values to the Setup1 and Setup2 registers in the receive channel root registers. Refer to the A429 Protocol Engine manual for details on these registers.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Bank #).
RxChanNum - receive channel number (0-15).
Setup1 - value for the receive channel Setup1 register (see PE manual).
Setup2 - value for the receive channel Setup2 register (see PE manual).

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_BAD_INPUT - Invalid channel number
ADT_ERR_UNSUPPORTED_CHANNELTYPE - Not an A429 device
ADT_FAILURE - Completed with error

ADT_L1_A429_RX_Channel_GetConfig

```
ADT_L0_UINT32 ADT_L1_A429_RX_Channel_GetConfig (
    ADT_L0_UINT32 devID,
    ADT_L0_UINT32 RxChanNum,
    ADT_L0_UINT32 *pSetup1,
    ADT_L0_UINT32 *pSetup2)
```

This function gets the protocol settings for the specified receive channel. This function reads values from the Setup1 and Setup2 registers in the receive channel root registers. Refer to the A429 Protocol Engine manual for details on these registers.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Bank #).
RxChanNum - receive channel number (0-15).
pSetup1 - pointer for the receive channel Setup1 register (see PE manual).
pSetup2 - pointer for the receive channel Setup2 register (see PE manual).

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_BAD_INPUT - Invalid channel number
ADT_ERR_UNSUPPORTED_CHANNELTYPE - Not an A429 device
ADT_FAILURE - Completed with error

ADT_L1_A429_RX_Channel_SetMaskCompare

```
ADT_L0_UINT32 ADT_L1_A429_RX_Channel_SetMaskCompare (
    ADT_L0_UINT32 devID,
    ADT_L0_UINT32 RxChanNum,
    ADT_L0_UINT32 mask1,
    ADT_L0_UINT32 compare1,
    ADT_L0_UINT32 mask2,
    ADT_L0_UINT32 compare2)
```

This function sets the mask/compare interrupt settings for the specified receive channel. This function writes values to the Mask1, Compare1, Mask2, and Compare2 registers in the receive channel root registers. Every time the channel receives a label, it ANDs the value with the Mask value then compares to the Compare value – if the two are equal then an interrupt (and interrupt queue entry) is generated. There are two sets of Mask/Compare registers to allow the user to interrupt on two different conditions for the same receive channel. Refer to the A429 Protocol Engine manual for details on these registers.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Bank #).
RxChanNum - receive channel number (0-15).
mask1 - value for the receive channel Mask1 register (see PE manual).
compare1 - value for the receive channel Compare1 register (see PE manual).
mask2 - value for the receive channel Mask2 register (see PE manual).
compare2 - value for the receive channel Compare2 register (see PE manual).

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_BAD_INPUT - Invalid channel number
ADT_ERR_UNSUPPORTED_CHANNELTYPE - Not an A429 device
ADT_FAILURE - Completed with error

ADT_L1_A429_RX_Channel_GetMaskCompare

```
ADT_L0_UINT32 ADT_L1_A429_RX_Channel_GetMaskCompare (
    ADT_L0_UINT32 devID,
    ADT_L0_UINT32 RxChanNum,
    ADT_L0_UINT32 *pMask1,
    ADT_L0_UINT32 *pCompare1,
    ADT_L0_UINT32 *pMask2,
    ADT_L0_UINT32 *pCompare2)
```

This function gets the mask/compare interrupt settings for the specified receive channel. This function reads values from the Mask1, Compare1, Mask2, and Compare2 registers in the receive channel root registers. Every time the channel receives a label, it ANDs the value with the Mask value then compares to the Compare value – if the two are equal then an interrupt (and interrupt queue entry) is generated. There are two sets of Mask/Compare registers to allow the user to interrupt on two different conditions for the same receive channel. Refer to the A429 Protocol Engine manual for details on these registers.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Bank #).
RxChanNum - receive channel number (0-15).
pMask1 -pointer for the receive channel Mask1 register (see PE manual).
pCompare1 - pointer for the receive channel Compare1 register (see PE manual).
pMask2 - pointer for the receive channel Mask2 register (see PE manual).
pCompare2 - pointer for the receive channel Compare2 register (see PE manual).

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_BAD_INPUT - Invalid channel number
ADT_ERR_UNSUPPORTED_CHANNELTYPE - Not an A429 device
ADT_FAILURE - Completed with error

ADT_L1_A717_RX_Channel_SetConfig

```
ADT_L0_UINT32 ADT_L1_A717_RX_Channel_SetConfig (
    ADT_L0_UINT32 devID,
    ADT_L0_UINT32 RxChanNum,
    ADT_L0_UINT32 Csr,
    ADT_L0_UINT32 Sync1,
    ADT_L0_UINT32 Sync2,
    ADT_L0_UINT32 Sync3,
    ADT_L0_UINT32 Sync4)
```

This function sets the protocol settings for the specified receive channel. This function writes values to the 4 SubFrame registers in the receive channel root registers. Refer to the A429 Protocol Engine manual for details on these registers.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, *Bank #*).
RxChanNum - RX Channel number (0-15).
Csr - value for the A717 CSR register.
Sync1 - value for the A717 SubFrame #1 Sync Word register.
Sync2 - value for the A717 SubFrame #2 Sync Word register.
Sync3 - value for the A717 SubFrame #3 Sync Word register.
Sync4 - value for the A717 SubFrame #4 Sync Word register.

Returns:

- **ADT_SUCCESS** - Completed without error
- **ADT_ERR_BAD_INPUT** - Invalid Rx Channel number
- **ADT_ERR_UNSUPPORTED_CHANNELTYPE** - Not an A429 device
- **ADT_FAILURE** - Completed with error

ADT_L1_A717_RX_Channel_GetConfig

```
ADT_L0_UINT32 ADT_L1_A717_RX_Channel_SetConfig (
    ADT_L0_UINT32 devID,
    ADT_L0_UINT32 RxChanNum,
    ADT_L0_UINT32 pCsr,
    ADT_L0_UINT32 pSync1,
    ADT_L0_UINT32 pSync2,
    ADT_L0_UINT32 pSync3,
    ADT_L0_UINT32 pSync4)
```

This function gets the protocol settings for the specified receive channel. This function reads values from the SubFrame Sync Word registers in the receive channel root registers. Refer to the A429 Protocol Engine manual for details on these registers.

Parameters:

devID - device identifier (*Backplane, Board Type, Board #, Channel Type, Bank #*).
RxChanNum - RX Channel number (0-15).
pCsr - value for the A717 CSR register.
pSync1 - pointer to the A717 SubFrame #1 Sync Word register.
pSync2 - pointer to the A717 SubFrame #2 Sync Word register.
pSync3 - pointer to the A717 SubFrame #3 Sync Word register.
pSync4 - pointer to the A717 SubFrame #4 Sync Word register.

Returns:

- **ADT_SUCCESS** - Completed without error
- **ADT_ERR_BAD_INPUT** - Invalid Rx Channel number
- **ADT_ERR_UNSUPPORTED_CHANNELTYPE** - Not an A429 device
- **ADT_FAILURE** - Completed with error

A429 Transmit Functions

This section describes the Layer 1 API A429 Transmit functions. These functions are defined in the file ADT_L1_A429_TX.c.

ADT_L1_A429_TX_Channel_Init

```
ADT_L0_UINT32 ADT_L1_A429_TX_Channel_Init (ADT_L0_UINT32 devID,  
                                         ADT_L0_UINT32 TxChanNum,  
                                         ADT_L0_UINT32 BitRateHz,  
                                         ADT_L0_UINT32 numTXCB)
```

This function initializes data structures for the specified transmit channel.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Bank #).
TxChanNum - transmit channel number (0-15).
BitRateHz - bit rate in Hz (500-500000).
numTXCB - number of Transmit Control Blocks to allocate for the channel.

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_BAD_INPUT - Invalid channel number or invalid bit rate
ADT_ERR_UNSUPPORTED_CHANNELTYPE - Not an A429 device
ADT_FAILURE - Completed with error

ADT_L1_A429_TX_Channel_Close

ADT_L0_UINT32 ADT_L1_A429_TX_Channel_Close (ADT_L0_UINT32 *devID*,
ADT_L0_UINT32 *TxChanNum*)

This function clears data structures and frees memory used by the specified transmit channel.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Bank #).
TxChanNum - transmit channel number (0-15).

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_BAD_INPUT - Invalid channel number
ADT_ERR_UNSUPPORTED_CHANNELTYPE - Not an A429 device
ADT_FAILURE - Completed with error

ADT_L1_A429_TX_Channel_CB_TXPAllocate

ADT_L0_UINT32 ADT_L1_A429_TX_Channel_CB_TXPAllocate (

ADT_L0_UINT32 *devID*,
ADT_L0_UINT32 *TxChanNum*,
ADT_L0_UINT32 *msgnum*,
ADT_L0_UINT32 *numTXP*)

This function allocates memory for a transmit control block (TXCB) and its associated transmit packets (TXPs).

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Bank #).
TxChanNum - transmit channel number (0-15).
msgnum - message number (TXCB index) for which to allocate transmit packets.
numTXP - number of transmit packets to allocate for the TXCB.

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_BAD_INPUT - Invalid channel number
ADT_ERR_NO_TXCB_TABLE - TXCB table has not been allocated
ADT_ERR_TXCB_ALREADY_ALLOCATED - TXCB has already been allocated
ADT_ERR_UNSUPPORTED_CHANNELTYPE - Not an A429 device
ADT_FAILURE - Completed with error

ADT_L1_A429_TX_Channel_CB_TXPFree

```
ADT_L0_UINT32 ADT_L1_A429_TX_Channel_CB_TXPFree (
    ADT_L0_UINT32 devID,
    ADT_L0_UINT32 TxChanNum,
    ADT_L0_UINT32 msgnum)
```

This function frees memory for a transmit control block (TXCB) and its associated transmit packets (TXPs).

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Bank #).
TxChanNum - transmit channel number (0-15).
msgnum - message number (TXCB index) for which to allocate transmit packets.

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_BAD_INPUT - Invalid channel number
ADT_ERR_NO_TXCB_TABLE - TXCB table has not been allocated
ADT_ERR_TXCB_NOT_ALLOCATED - TXCB has not been allocated
ADT_ERR_UNSUPPORTED_CHANNELTYPE - Not an A429 device
ADT_FAILURE - Completed with error

ADT_L1_A429_TX_Channel_CB_Write

```
ADT_L0_UINT32 ADT_L1_A429_TX_Channel_CB_Write (
    ADT_L0_UINT32 devID,
    ADT_L0_UINT32 TxChanNum,
    ADT_L0_UINT32 msgnum,
    ADT_L1_A429_TXCB *txcb)
```

This function writes a transmit control block for the specified channel. This function only writes the following parts of the TX CB:

- Next TX CB number
- Control Word
- TX Period

If you need to change the total number of TXPs for the TXCB refer to the example program ADT_L1_A429_ex_tx7.c to see how to do this.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Bank #).
TxChanNum - transmit channel number (0-15).
msgnum - message number (TXCB index).
txcb - pointer to the transmit control block structure to write.

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_BAD_INPUT - Invalid channel number
ADT_ERR_NO_TXCB_TABLE - TXCB table has not been allocated
ADT_ERR_TXCB_NOT_ALLOCATED - TXCB has not been allocated
ADT_ERR_UNSUPPORTED_CHANNELTYPE - Not an A429 device
ADT_FAILURE - Completed with error

ADT_L1_A429_TX_Channel_CB_Read

```
ADT_L0_UINT32 ADT_L1_A429_TX_Channel_CB_Read (
    ADT_L0_UINT32 devID,
    ADT_L0_UINT32 TxChanNum,
    ADT_L0_UINT32 msgnum,
    ADT_L1_A429_TXCB *txcb)
```

This function reads a transmit control block for the specified channel.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Bank #).
TxChanNum - transmit channel number (0-15).
msgnum - message number (TXCB index).
txcb - pointer to the transmit control block structure.

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_BAD_INPUT - Invalid channel number
ADT_ERR_NO_TXCB_TABLE - TXCB table has not been allocated
ADT_ERR_TXCB_NOT_ALLOCATED - TXCB has not been allocated
ADT_ERR_UNSUPPORTED_CHANNELTYPE - Not an A429 device
ADT_FAILURE - Completed with error

ADT_L1_A429_TX_Channel_CB_TXPWrite

```
ADT_L0_UINT32 ADT_L1_A429_TX_Channel_CB_TXPWrite (  
                                ADT_L0_UINT32 devID,  
                                ADT_L0_UINT32 TxChanNum,  
                                ADT_L0_UINT32 msgnum,  
                                ADT_L0_UINT32 txpNum,  
                                ADT_L1_A429_TXP *pTxp)
```

This function writes a transmit packet (TXP) for the specified transmit control block. Only the LSB 6-bits (0-5) of the TXP Control Word, 32-bit Pre-TX Delay and 32-bit Data Word values are written. This function does not overwrite the TXP Reserved API Info Word.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Bank #).
TxChanNum - transmit channel number (0-15).
msgnum - message number (TXCB index).
txpNum - transmit packet number (TXP index).
pTxp - pointer to the transmit packet structure to write.

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_BAD_INPUT - Invalid channel number
ADT_ERR_NO_TXCB_TABLE - TXCB table has not been allocated
ADT_ERR_TXCB_NOT_ALLOCATED - TXCB has not been allocated
ADT_ERR_UNSUPPORTED_CHANNELTYPE - Not an A429 device
ADT_FAILURE - Completed with error

ADT_L1_A429_TX_Channel_CB_TXPRead

```
ADT_L0_UINT32 ADT_L1_A429_TX_Channel_CB_TXPRead (
    ADT_L0_UINT32 devID,
    ADT_L0_UINT32 TxChanNum,
    ADT_L0_UINT32 msgnum,
    ADT_L0_UINT32 txpNum,
    ADT_L1_A429_TXP *pTxp)
```

This function reads a transmit packet for the specified transmit control block.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Bank #).
TxChanNum - transmit channel number (0-15).
msgnum - message number (TXCB index).
txpNum - transmit packet number (TXP index).
pTxp - pointer to the transmit packet structure.

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_BAD_INPUT - Invalid channel number
ADT_ERR_NO_TXCB_TABLE - TXCB table has not been allocated
ADT_ERR_TXCB_NOT_ALLOCATED - TXCB has not been allocated
ADT_ERR_UNSUPPORTED_CHANNELTYPE - Not an A429 device
ADT_FAILURE - Completed with error

ADT_L1_A429_TX_Channel_Start

ADT_L0_UINT32 ADT_L1_A429_TX_Channel_Start (ADT_L0_UINT32 *devID*,
 ADT_L0_UINT32 *TxChanNum*,
 ADT_L0_UINT32 *msgnum*)

This function starts transmit operation for the specified transmit channel with the specified message number (TXCB index).

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Bank #).
TxChanNum - transmit channel number (0-15).
msgnum - message number (TXCB index).

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_BAD_INPUT - Invalid channel number
ADT_ERR_NO_TXCB_TABLE - TXCB table has not been allocated
ADT_ERR_UNSUPPORTED_CHANNELTYPE - Not an A429 device
ADT_FAILURE - Completed with error

ADT_L1_A429_TX_Channel_Stop

ADT_L0_UINT32 ADT_L1_A429_TX_Channel_Stop (ADT_L0_UINT32 *devID*,
 ADT_L0_UINT32 *TxChanNum*)

This function stops transmit operation for the specified transmit channel.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Bank #).
TxChanNum - transmit channel number (0-15).

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_BAD_INPUT - Invalid channel number
ADT_ERR_UNSUPPORTED_CHANNELTYPE - Not an A429 device
ADT_FAILURE - Completed with error

ADT_L1_A429_TX_Channel_IsRunning

```
ADT_L0_UINT32 ADT_L1_A429_TX_Channel_IsRunning (
    ADT_L0_UINT32 devID,
    ADT_L0_UINT32 TxChanNum,
    ADT_L0_UINT32 *pIsRunning)
```

This function determines if the specified transmit channel is running.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Bank #).
TxChanNum - transmit channel number (0-15).
pIsRunning - pointer to store the “is running state”, where zero indicates not running and non-zero indicates running.

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_BAD_INPUT - Invalid channel number
ADT_ERR_UNSUPPORTED_CHANNELTYPE - Not an A429 device
ADT_FAILURE - Completed with error

ADT_L1_A429_TX_Channel_AperiodicIsRunning

```
ADT_L0_UINT32 ADT_L1_A429_TX_Channel_AperiodicIsRunning (
    ADT_L0_UINT32 devID,
    ADT_L0_UINT32 TxChanNum,
    ADT_L0_UINT32 *pIsRunning)
```

This function determines if the specified transmit channel has an Aperiodic TXCB-TXP list posted for execution. This user should call this routine prior to posting an Aperiodic TXCB, and should not call the ADT_L1_A429_TX_Channel_AperiodicSend() function unless the parameter pIsRunning is zero. There are example programs that show how to use this function.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Bank #).
TxChanNum - transmit channel number (0-15).
pIsRunning - pointer to store the “is running state”, where zero indicates not running and non-zero indicates running.

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_BAD_INPUT - Invalid channel number
ADT_ERR_UNSUPPORTED_CHANNELTYPE - Not an A429 device
ADT_FAILURE - Completed with error

ADT_L1_A429_TX_Channel_SendLabel

```
ADT_L0_UINT32 ADT_L1_A429_TX_Channel_SendLabel (
    ADT_L0_UINT32 devID,
    ADT_L0_UINT32 TxChanNum,
    ADT_L0_UINT32 Label)
```

This function sends a Label (as aperiodic or one-shot). THIS FUNCTION DOES NOT RETURN UNTIL THE LABEL HAS BEEN SENT AND IT IS SAFE TO SEND ANOTHER LABEL ON THE CHANNEL.

THIS FUNCTION IS NOT THREAD SAFE BECAUSE IT ALLOCATES AND FREES MEMORY ON THE DEVICE. Do not use multiple threads to call this function for multiple TX channels on the same device.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Bank #).
TxChanNum - transmit channel number (0-15).
Label - 32-bit A429 Label word to send.

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_BAD_INPUT - Invalid channel number
ADT_ERR_UNSUPPORTED_CHANNELTYPE - Not an A429 device
ADT_ERR_TIMEOUT - Timeout waiting for APERIODIC TXP register to clear
ADT_FAILURE - Completed with error

ADT_L1_A429_TX_Channel_SendLabelBlock

```
ADT_L0_UINT32 ADT_L1_A429_TX_Channel_SendLabelBlock (
    ADT_L0_UINT32 devID,
    ADT_L0_UINT32 TxChanNum,
    ADT_L0_UINT32 numLabels,
    ADT_L0_UINT32 pLabels)
```

This function sends a block of Labels (as aperiodic or one-shot). THIS FUNCTION DOES NOT RETURN UNTIL THE LABELS HAVE BEEN SENT AND IT IS SAFE TO SEND ANOTHER BLOCK OF LABELS ON THE CHANNEL.

THIS FUNCTION IS NOT THREAD SAFE BECAUSE IT ALLOCATES AND FREES MEMORY ON THE DEVICE. Do not use multiple threads to call this function for multiple TX channels on the same device.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Bank #).
TxChanNum - transmit channel number (0-15).
numLabels - number of 32-bit A429 Label words to send (maximum 1000).
pLabels - pointer to the 32-bit A429 Label words to send.

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_BAD_INPUT - Invalid channel number, invalid number of labels, or null pointer
ADT_ERR_UNSUPPORTED_CHANNELTYPE - Not an A429 device
ADT_ERR_TIMEOUT - Timeout waiting for APERIODIC TXP register to clear
ADT_FAILURE - Completed with error

ADT_L1_A429_TX_Channel_AperiodicSend

```
ADT_L0_UINT32 ADT_L1_A429_TX_Channel_AperiodicSend (
    ADT_L0_UINT32 devID,
    ADT_L0_UINT32 TxChanNum,
    ADT_L0_UINT32 msgnum)
```

This function posts a single TXCB (with one or more TXPs) for execution. If there are TXCBs being processed, then the current TXCB will complete and then this posted TXCB will execute. You should call the

ADT_L1_A429_TX_Channel_AperiodicIsRunning prior to calling this function to verify that a previously posted Aperiodic is not executing. This function will overwrite any current posted Aperiodic TXCB and unknown results could occur if you do not ensure a current Aperiodic is not running. There are example programs that show how to use this function.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Bank #).
TxChanNum - transmit channel number (0-15).
msgnum - TXCB number to send.

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_BAD_INPUT - Invalid channel number
ADT_ERR_UNSUPPORTED_CHANNELTYPE - Not an A429 device
ADT_ERR_TIMEOUT - Timeout waiting for APERIODIC TXP register to clear
ADT_FAILURE - Completed with error

ADT_L1_A429_TX_Channel_SetConfig

```
ADT_L0_UINT32 ADT_L1_A429_TX_Channel_SetConfig (
    ADT_L0_UINT32 devID,
    ADT_L0_UINT32 TxChanNum,
    ADT_L0_UINT32 CSR1,
    ADT_L0_UINT32 CSR2)
```

This function sets the protocol settings for the specified transmit channel. This function writes values to the CSR1 and CSR2 registers in the transmit channel root registers. Refer to the A429 Protocol Engine manual for details on these registers.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Bank #).
TxChanNum - transmit channel number (0-15).
CSR1 - value for the transmit channel CSR1 register (see PE manual).
CSR2 - value for the transmit channel CSR2 register (see PE manual).

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_BAD_INPUT - Invalid channel number
ADT_ERR_UNSUPPORTED_CHANNELTYPE - Not an A429 device
ADT_FAILURE - Completed with error

ADT_L1_A429_TX_Channel_GetConfig

```
ADT_L0_UINT32 ADT_L1_A429_TX_Channel_GetConfig (
    ADT_L0_UINT32 devID,
    ADT_L0_UINT32 TxChanNum,
    ADT_L0_UINT32 *pCSR1,
    ADT_L0_UINT32 *pCSR2)
```

This function gets the protocol settings for the specified transmit channel. This function reads values from the CSR1 and CSR2 registers in the transmit channel root registers. Refer to the A429 Protocol Engine manual for details on these registers.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Bank #).
TxChanNum - transmit channel number (0-15).
pCSR1 - pointer for the transmit channel CSR1 register (see PE manual).
pCSR2 - pointer for the transmit channel CSR2 register (see PE manual).

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_BAD_INPUT - Invalid channel number
ADT_ERR_UNSUPPORTED_CHANNELTYPE - Not an A429 device
ADT_FAILURE - Completed with error

ADT_L1_A429_TX_ChannelGetTxpCount

```
ADT_L0_UINT32 ADT_L1_A429_TX_Channel_GetTxpCount (
    ADT_L0_UINT32 devID,
    ADT_L0_UINT32 TxChanNum,
    ADT_L0_UINT32 *txpCnt)
```

This function returns the txp count for an A429 Transmit Channel.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Bank #).
TxChanNum - transmit channel number (0-15).
txpCnt - pointer for the transmit channel txp count.

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_BAD_INPUT - Invalid channel number
ADT_ERR_UNSUPPORTED_CHANNELTYPE - Not an A429 device
ADT_FAILURE - Completed with error

ADT_L1_A429_TX_Channel_CB_GetAddr

```
ADT_L0_UINT32 ADT_L1_A429_TX_Channel_CB_GetAddr(
    ADT_L0_UINT32 devID,
    ADT_L0_UINT32 TxChanNum,
    ADT_L0_UINT32 msgnum,
    ADT_L0_UINT32 *pAddr)
```

This function gets the byte address of a TX Control Block (TXCB) – see AltaCore-A429 manual or A429 Quick Reference for the TXCB structure. This address can be used to directly read or write words in a TXCB using the functions `ADT_L1_ReadDeviceMem32` and `ADT_L1_WriteDeviceMem32`.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).
TxChanNum - TX Channel Number.
msgnum - message number.
pAddr - pointer to store the address.

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_BAD_INPUT - Invalid channel or message number
ADT_FAILURE - Completed with error

ADT_L1_A429_TX_Channel_CB_TXP_GetAddr

```
ADT_L0_UINT32 ADT_L1_A429_TX_Channel_CB_TXP_GetAddr(  
    ADT_L0_UINT32 devID,  
    ADT_L0_UINT32 TxChanNum,  
    ADT_L0_UINT32 msgnum,  
    ADT_L0_UINT32 txpNum,  
    ADT_L0_UINT32 *pAddr)
```

This function gets the byte address of a TXP – see AltaCore-A429 manual or A429 Quick Reference for the TXP structure. This address can be used to directly read or write words in a TXP using the functions `ADT_L1_ReadDeviceMem32` and `ADT_L1_WriteDeviceMem32`.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Channel #).
TxChanNum - TX Channel Number.
msgnum - message number.
txpNum - TXP number.
pAddr - pointer to store the address.

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_BAD_INPUT - Invalid channel or message number
ADT_FAILURE - Completed with error

A429 Signal Generator Functions

This section describes the Layer 1 API A429 Signal Generator functions. These functions are defined in the file ADT_L1_A429_SG.c.

ADT_L1_A429_SG_AddVectors

```
ADT_L0_UINT32 ADT_L1_A429_SG_AddVectors (ADT_L0_UINT32 numVec,  
                                         ADT_L0_UINT32 vector2bit,  
                                         ADT_L0_UINT32 * pVectors,  
                                         ADT_L0_UINT32 sizeInWords,  
                                         ADT_L0_UINT32 * pNumVectors)
```

This function adds a sequence of vectors to a list of vectors.

Parameters:

numVec - number of 2-bit vectors to add (1 vector = 100ns).
vector2bit - 2-bit pattern to add for each vector (00=GND, 01=LOW, 10=HIGH, 11=GND).
pVectors - pointer to the array of 32-bit words containing vectors.
sizeInWords - max size (in words) of the vector array.
pNumVectors - pointer to a UINT32 containing the current vector count (in bits).

Returns:

ADT_SUCCESS - Completed without error
ADT_FAILURE - Completed with error

ADT_L1_A429_SG_Configure

```
ADT_L0_UINT32 ADT_L1_A429_SG_Configure (ADT_L0_UINT32 devID,  
                                         ADT_L0_UINT32 txChannel,  
                                         ADT_L0_UINT32 slewRate)
```

This function configures Signal Generator operation for the device by initializing the SG registers.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Bank #).
txChannel - TX channel (0-15) to use for the signal generator.
slewRate - selects low-speed (0) or high-speed (1) slew rate for the transmit channel.

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_UNSUPPORTED_CHANNELTYPE - Not an A429 device
ADT_FAILURE - Completed with error

ADT_L1_A429_SG_CreateSGCB

```
ADT_L0_UINT32 ADT_L1_A429_SG_CreateSGCB (ADT_L0_UINT32 devID,  
                                         ADT_L0_UINT32 timeHigh,  
                                         ADT_L0_UINT32 timeLow,  
                                         ADT_L0_UINT32 * pVectors,  
                                         ADT_L0_UINT32 numVectors)
```

This function allocates and writes a SGCB to the device.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Bank #).
timeHigh - upper 32 bits of the time for this SGCB.
timeLow - lower 32 bits of the time for this SGCB.
pVectors - pointer to an array of 32-bit words containing vectors.
numVectors - number of vectors (in bits, may use only part of the last word).

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_UNSUPPORTED_CHANNELTYPE - Not an A429 device
ADT_FAILURE - Completed with error

ADT_L1_A429_SG_Free

```
ADT_L0_UINT32 ADT_L1_A429_SG_Free (ADT_L0_UINT32 devID)
```

This function frees all SG CBs for the device.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Bank #).

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_UNSUPPORTED_CHANNELTYPE - Not an A429 device
ADT_FAILURE - Completed with error

ADT L1 A429 SG Start

ADT_L0_UINT32 ADT_L1_A429_SG_Start (ADT_L0_UINT32 devID)

This function starts Signal Generator operation for the device.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Bank #).

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_UNSUPPORTED_CHANNELTYPE - Not an A429 device
ADT_FAILURE - Completed with error

APT L1 A429 SG Stop

ADT L0 UINT32 ADT L1 A429 SG Stop (ADT L0 UINT32 devID)

This function stops Signal Generator operation for the device.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Bank #).

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_UNSUPPORTED_CHANNELTYPE - Not an A429 device
ADT_FAILURE - Completed with error

ADT L1 A429 SG IsRunning

This function determines if the Signal Generator is running or stopped.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Bank #).
plsRunning - pointer to store the result, 0=Not Running, 1=Running.

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_UNSUPPORTED_CHANNELTYPE - Not an A429 device
ADT_FAILURE - Completed with error

ADT_L1_A429_SG_WordToVectors

ADT_L0_UINT32 ADT_L1_A429_SG_WordToVectors

(ADT_L0_UINT32 *a429word*,
ADT_L0_UINT32 *halfBitTime100ns*,
ADT_L0_UINT32 * *pVectors*,
ADT_L0_UINT32 *sizeInWords*,
ADT_L0_UINT32 * *pNumVectors*)

This function adds an A429 word to a list of vectors.

Parameters:

A429word is the 32-bit A429 word.
halfBitTime100ns - half-bit time (100ns LSB).
pVectors - pointer to the array of 32-bit words containing vectors.
sizeInWords - max size (in words) of the vector array.
pNumVectors - pointer to a UINT32 containing the current vector count (in 2-bit 20ns vectors).

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_UNSUPPORTED_CHANNELTYPE - Not an A429 device
ADT_FAILURE - Completed with error - not enough words available in vectors array

A429 Playback Functions

This section describes the Layer 1 API A429 Playback functions. These functions are defined in the file ADT_L1_A429_PB.c.

ADT_L1_A429_TX_Channel_PB_Init

```
ADT_L0_UINT32 ADT_L1_A429_TX_Channel_PB_Init (ADT_L0_UINT32 devID,  
                                              ADT_L0_UINT32 TxChanNum,  
                                              ADT_L0_UINT32 BitRateHz,  
                                              ADT_L0_UINT32 numPBCB)
```

This function initializes playback data structures for the specified transmit channel.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Bank #).
TxChanNum - transmit channel number (0-15).
BitRateHz - bit rate in Hz (500-500000).
numPBCB - number of Playback Control Blocks to allocate for the TX channel.

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_BAD_INPUT - Invalid channel number or invalid bit rate
ADT_ERR_UNSUPPORTED_CHANNELTYPE - Not an A429 device
ADT_FAILURE - Completed with error

ADT_L1_A429_TX_Channel_PB_Close

ADT_L0_UINT32 ADT_L1_A429_TX_Channel_PB_Close

(ADT_L0_UINT32 *devID*,
ADT_L0_UINT32 *TxChanNum*)

This function clears data structures and frees memory used by the specified transmit channel.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Bank #).
TxChanNum - transmit channel number (0-15).

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_BAD_INPUT - Invalid channel number
ADT_ERR_UNSUPPORTED_CHANNELTYPE - Not an A429 device
ADT_FAILURE - Completed with error

ADT_L1_A429_TX_Channel_PB_CB_PXPAllocate

ADT_L0_UINT32 ADT_L1_A429_TX_Channel_PB_CB_TXPAllocate (

ADT_L0_UINT32 *devID*,
ADT_L0_UINT32 *TxChanNum*,
ADT_L0_UINT32 *PBCBnum*,
ADT_L0_UINT32 *numPXP*)

This function allocates memory for a playback control block (PBCB) and its associated playback transmit packets (PXPs).

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Bank #).
TxChanNum - transmit channel number (0-15).
PBCBnum - PBCB number (PBCB index) for which to allocate transmit packets.
numPXP - number of playback transmit packets to allocate for the PBCB.

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_BAD_INPUT - Invalid channel number
ADT_ERR_NO_TXCB_TABLE - TXCB table has not been allocated
ADT_ERR_TXCB_ALREADY_ALLOCATED - TXCB has already been allocated
ADT_ERR_UNSUPPORTED_CHANNELTYPE - Not an A429 device
ADT_FAILURE - Completed with error

ADT_L1_A429_TX_Channel_PB_CB_PXPFree

```
ADT_L0_UINT32 ADT_L1_A429_TX_Channel_PB_CB_TXPFree (
    ADT_L0_UINT32 devID,
    ADT_L0_UINT32 TxChanNum,
    ADT_L0_UINT32 PBCBnum)
```

This function frees memory for a transmit control block (TXCB) and its associated transmit packets (TXPs).

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Bank #).
TxChanNum - transmit channel number (0-15).
PBCB - PBCB number (PBCB index) for which to free allocated PXP packets.

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_BAD_INPUT - Invalid channel number
ADT_ERR_NO_TXCB_TABLE - TXCB table has not been allocated
ADT_ERR_TXCB_NOT_ALLOCATED - TXCB has not been allocated
ADT_ERR_UNSUPPORTED_CHANNELTYPE - Not an A429 device
ADT_FAILURE - Completed with error

ADT_L1_A429_TX_Channel_PB_CB_Write

```
ADT_L0_UINT32 ADT_L1_A429_TX_Channel_PB_CB_Write (
    ADT_L0_UINT32 devID,
    ADT_L0_UINT32 TxChanNum,
    ADT_L0_UINT32 msgnum,
    ADT_L1_A429_TXCB *pbcb)
```

This function writes a transmit control block for the specified channel.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Bank #).
TxChanNum - transmit channel number (0-15).
msgnum - message number (TXCB index).
pbcb - pointer to the playback control block structure to write.

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_BAD_INPUT - Invalid channel number
ADT_ERR_NO_TXCB_TABLE - TXCB table has not been allocated
ADT_ERR_TXCB_NOT_ALLOCATED - TXCB has not been allocated
ADT_ERR_UNSUPPORTED_CHANNELTYPE - Not an A429 device
ADT_FAILURE - Completed with error

ADT_L1_A429_TX_Channel_PB_CB_Read

```
ADT_L0_UINT32 ADT_L1_A429_TX_Channel_PB_CB_Read (
    ADT_L0_UINT32 devID,
    ADT_L0_UINT32 TxChanNum,
    ADT_L0_UINT32 msgnum,
    ADT_L1_A429_TXCB *pbcb)
```

This function reads a transmit control block for the specified channel.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Bank #).
TxChanNum - transmit channel number (0-15).
msgnum - message number (TXCB index).
pbcb - pointer to the playback control block structure.

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_BAD_INPUT - Invalid channel number
ADT_ERR_NO_TXCB_TABLE - TXCB table has not been allocated
ADT_ERR_TXCB_NOT_ALLOCATED - TXCB has not been allocated
ADT_ERR_UNSUPPORTED_CHANNELTYPE - Not an A429 device
ADT_FAILURE - Completed with error

ADT_L1_A429_TX_Channel_PB_CB_PXPWrite

```
ADT_L0_UINT32 ADT_L1_A429_TX_Channel_PB_CB_PXPWrite (
    ADT_L0_UINT32 devID,
    ADT_L0_UINT32 TxChanNum,
    ADT_L0_UINT32 msgnum,
    ADT_L0_UINT32 ppxpNum,
    ADT_L1_A429_TXP *pPxp)
```

This function writes a transmit packet for the specified transmit control block.

NOTE: only the lower 6-bits of the TXP Control word are written – user cannot change the other bits in this word.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Bank #).
TxChanNum - transmit channel number (0-15).
msgnum - message number (TXCB index).
ppxpNum - transmit packet number (TXP index).
pPxp - pointer to the transmit packet structure to write.

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_BAD_INPUT - Invalid channel number
ADT_ERR_NO_TXCB_TABLE - TXCB table has not been allocated
ADT_ERR_TXCB_NOT_ALLOCATED - TXCB has not been allocated
ADT_ERR_UNSUPPORTED_CHANNELTYPE - Not an A429 device
ADT_FAILURE - Completed with error

ADT_L1_A429_TX_Channel_PB_CB_PXPRead

```
ADT_L0_UINT32 ADT_L1_A429_TX_Channel_PB_CB_PXPRead (
    ADT_L0_UINT32 devID,
    ADT_L0_UINT32 TxChanNum,
    ADT_L0_UINT32 msgnum,
    ADT_L0_UINT32 ppxpNum,
    ADT_L1_A429_TXP *pPxp)
```

This function reads a transmit packet for the specified transmit control block.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Bank #).
TxChanNum - transmit channel number (0-15).
msgnum - message number (PBCB index).
ppxpNum - transmit packet number (PXP index).
pPxp - pointer to the transmit packet structure.

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_BAD_INPUT - Invalid channel number
ADT_ERR_NO_TXCB_TABLE - TXCB table has not been allocated
ADT_ERR_TXCB_NOT_ALLOCATED - TXCB has not been allocated
ADT_ERR_UNSUPPORTED_CHANNELTYPE - Not an A429 device
ADT_FAILURE - Completed with error

ADT_L1_A429_TX_Channel_PB_Start

ADT_L0_UINT32 ADT_L1_A429_TX_Channel_PB_Start
(ADT_L0_UINT32 *devID*,
ADT_L0_UINT32 *TxChanNum*)

This function starts playback transmit operation for the specified transmit channel.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Bank #).
TxChanNum - transmit channel number (0-15).

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_BAD_INPUT - Invalid channel number
ADT_ERR_NO_TXCB_TABLE - TXCB table has not been allocated
ADT_ERR_UNSUPPORTED_CHANNELTYPE - Not an A429 device
ADT_FAILURE - Completed with error

ADT_L1_A429_TX_Channel_PB_Stop

ADT_L0_UINT32 ADT_L1_A429_TX_Channel_PB_Stop
(ADT_L0_UINT32 *devID*,
ADT_L0_UINT32 *TxChanNum*)

This function stops playback transmit operation for the specified transmit channel.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Bank #).
TxChanNum - transmit channel number (0-15).

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_BAD_INPUT - Invalid channel number
ADT_ERR_UNSUPPORTED_CHANNELTYPE - Not an A429 device
ADT_FAILURE - Completed with error

ADT_L1_A429_PB_Start

ADT_L0_UINT32 ADT_L1_A429_PB_Start (ADT_L0_UINT32 *devID*)

This function starts playback on all transmit channels for the device bank.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Bank #).

Returns:

ADT_SUCCESS - Completed without error

ADT_ERR_UNSUPPORTED_CHANNELTYPE - Not an A429 device

ADT_FAILURE - Completed with error

ADT_L1_A429_PB_Stop

ADT_L0_UINT32 ADT_L1_A429_PB_Stop (ADT_L0_UINT32 *devID*)

This function stops playback on all transmit channels for the device bank.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Bank #).

Returns:

ADT_SUCCESS - Completed without error

ADT_ERR_UNSUPPORTED_CHANNELTYPE - Not an A429 device

ADT_FAILURE - Completed with error

ADT_L1_A429_TX_Channel_PB_IsRunning

```
ADT_L0_UINT32 ADT_L1_A429_TX_Channel_PB_IsRunning (  
    ADT_L0_UINT32 devID,  
    ADT_L0_UINT32 TxChanNum,  
    ADT_L0_UINT32 *pIsRunning)
```

This function determines if the specified transmit channel is running.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Bank #).
TxChanNum - transmit channel number (0-15).
pIsRunning - pointer to store the “is running state”, where zero indicates not running and non-zero indicates running.

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_BAD_INPUT - Invalid channel number
ADT_ERR_UNSUPPORTED_CHANNELTYPE - Not an A429 device
ADT_FAILURE - Completed with error

ADT_L1_A429_TX_Channel_PB_SetConfig

```
ADT_L0_UINT32 ADT_L1_A429_TX_Channel_PB_SetConfig (  
    ADT_L0_UINT32 devID,  
    ADT_L0_UINT32 TxChanNum,  
    ADT_L0_UINT32 CSR1,  
    ADT_L0_UINT32 CSR2)
```

This function sets the protocol settings for the specified transmit channel. This function writes values to the CSR1 and CSR2 registers in the transmit channel root registers. Refer to the A429 Protocol Engine manual for details on these registers.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Bank #).
TxChanNum - transmit channel number (0-15).
CSR1 - value for the transmit channel CSR1 register (see PE manual).
CSR2 - value for the transmit channel CSR2 register (see PE manual).

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_BAD_INPUT - Invalid channel number
ADT_ERR_UNSUPPORTED_CHANNELTYPE - Not an A429 device
ADT_FAILURE - Completed with error

ADT_L1_A429_TX_Channel_PB_GetConfig

```
ADT_L0_UINT32 ADT_L1_A429_TX_Channel_PB_GetConfig (
    ADT_L0_UINT32 devID,
    ADT_L0_UINT32 TxChanNum,
    ADT_L0_UINT32 *pCSR1,
    ADT_L0_UINT32 *pCSR2)
```

This function gets the protocol settings for the specified transmit channel. This function reads values from the CSR1 and CSR2 registers in the transmit channel root registers. Refer to the A429 Protocol Engine manual for details on these registers.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Bank #).
TxChanNum - transmit channel number (0-15).
pCSR1 - pointer for the transmit channel CSR1 register (see PE manual).
pCSR2 - pointer for the transmit channel CSR2 register (see PE manual).

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_BAD_INPUT - Invalid channel number
ADT_ERR_UNSUPPORTED_CHANNELTYPE - Not an A429 device
ADT_FAILURE - Completed with error

ADT_L1_A429_TX_Channel_PB_RXPWrite

```
ADT_L0_UINT32 ADT_L1_A429_TX_Channel_PB_RXPWrite (  
                                ADT_L0_UINT32 devID,  
                                ADT_L0_UINT32 txChanNum,  
                                ADT_L0_UINT32 numRXPs,  
                                ADT_L1_A429_RXP *RxpBuffer,  
                                ADT_L0_UINT32 options,  
                                ADT_L0_UINT32 isFirstMessage)
```

This function converts a block of RXPs to PXP and writes them to an empty PBCB buffer.

Parameters:

devID - device identifier (Backplane, Board Type, Board #, Channel Type, Bank #).
txChanNum - transmit channel number (0-15).
numRXPs - number of RXPs pointed to by pRXP.
**RxpBuffer* - pointer to the RXP Buffer (probably and RXP array) to write to the PB TXP.
options are the packet control word options:

ADT_L1_A429_PBCB_CONTROL_STOPONPBCBCOMP	0x00000010
ADT_L1_A429_PBCB_CONTROL_INTONPBCBCOMP	0x00000100
ADT_L1_A429_PB_API_ATON	0x80000000

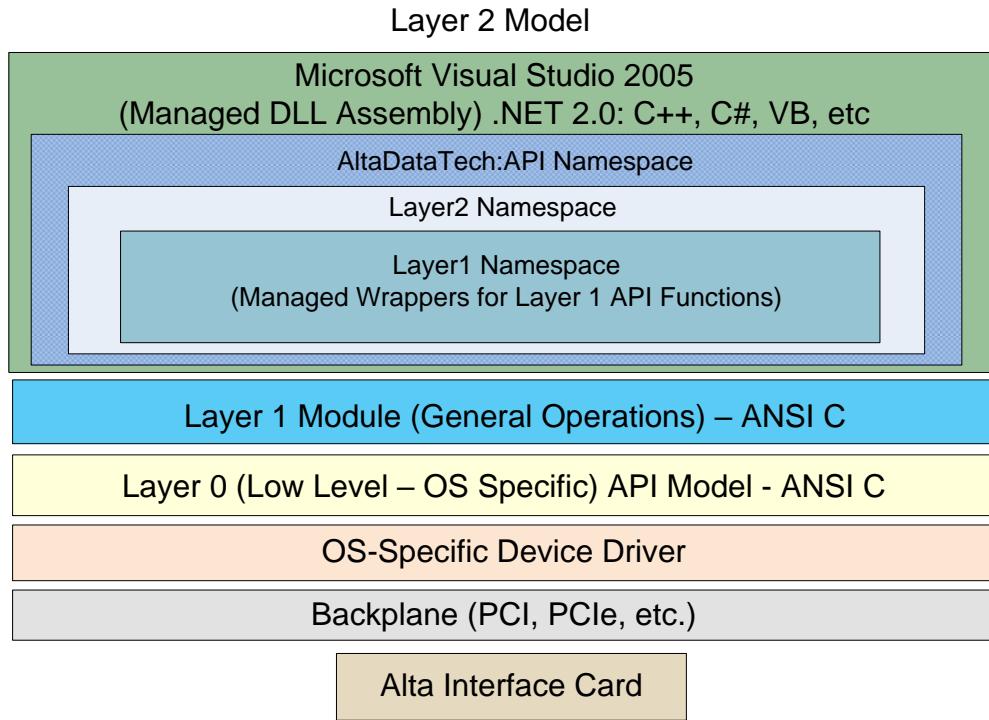
isFirstMessage - 1 for the first message in playback or 0 for NOT first message. See the section on “A429 Playback Operation” for details on playback functions and options.

Returns:

ADT_SUCCESS - Completed without error
ADT_ERR_BUFFER_FULL - Playback buffer is full (no Empty PBCBs)
ADT_ERR_PBCB_TOOMANYPXPS – numRXPs exceeds PBCB PXP buffer alloc
ADT_ERR_BAD_INPUT - Invalid channel number
ADT_ERR_UNSUPPORTED_CHANNELTYPE - Not an A429 device
ADT_FAILURE - Completed with error

The Layer 2 APIs

This section discusses the Layer 2 APIs.



The concept of the Layer 2 APIs is to encapsulate the basic ANSI “C” Layer 1 API in higher-level object-oriented code. There could be many different Layer 2 APIs for different higher-level languages (C++, ADA, C#, etc.).

Layer 2 API for Microsoft Windows .NET 2.0

Alta currently provides one module that would be considered a Layer 2 API – the ADT_L1_NET20 module, which is a Windows .NET 2.0 assembly. This is discussed further in Appendix B because it applies only to Microsoft Windows environments.

Appendix A – ENET-ONLY BSD API

This appendix discusses the ENET-ONLY BSD API, which uses a simplified Layer 0 API that uses Ethernet sockets for UDP communication with Alta ENET products. This API does not use any OS-level device driver and can be used on many platforms. This API provides full source code. **As the name implies, this is ONLY for Alta ENET devices and cannot be used with Alta PCI/PCIe devices.** The Layer 1 API is the same as for the standard API.

The ENET-ONLY API can be found on the Alta Software CD in the folder “Alta_BSD_ENET_API”. The information below is also in the provided “readme” files.

This API currently supports the following platforms:

- Windows
- Linux
- Solaris
- VxWorks

This API can be ported to just about any platform that supports Ethernet sockets.

The distribution files are configured for Linux/Solaris. If you change platforms you must edit two files:

1. ADT_L0.h - Edit the #includes appropriately for the platform.
Edit the #define for ADT_L0_CALL_CONV for the platform.
2. ADT_L0_BSD_ENET.c - Edit the #define for the platform
(WINDOWS, LINUX_SOLARIS, or VXWORKS).

FOR SOLARIS:

Go to the /Source folder, execute the following commands:

```
LD_LIBRARY_PATH=/usr/local/lib/sparcv9:/usr/sfw/lib/sparcv9
export LD_LIBRARY_PATH
PATH=/usr/local/bin:/usr/sfw/bin:$PATH
export PATH
```

You will need to know the IP address of your system (client).

You can get this with the following command:

```
# ifconfig -a
```

The ENET device default IP address is 192.168.0.128. Your system IP address must be on the same subnet (192.168.0.*). You can set a static IP address on your system as shown below:

```
# ifconfig eth0 192.168.0.1 netmask 255.255.255.0 up
```

This sets the system IP address for the eth0 interface to 192.168.0.1.

The /Source folder contains all the Alta Layer 0 and Layer 1 source files, as well as one of the example programs (ADT_L1_1553_ex_bcbm1_Timing_Test.c). The example program contains the "main" program.

You can use this example program or one of the others from the /Examples folder.
NOTE THAT YOU MUST EDIT THE EXAMPLE PROGRAM TO ENSURE THAT THE DEVID DEFINITION SPECIFIES THE APPROPRIATE ENET PRODUCT TYPE (ADT_PRODUCT_ENET1553, ADT_PRODUCT_ENET2_1553, ADT_PRODUCT_ENETA429).

Build the main program and API:

```
# gcc *.c -lssl -lsocket -lresolv -lrt
```

This will create the executable file "a.out". Run the program as follows:

```
# ./a.out
```

FOR LINUX:

Go to the /Source folder, execute the following commands:

```
export LD_LIBRARY_PATH=.:$LD_LIBRARY_PATH
```

You will need to know the IP address of your system (client).

You can get this with the following command:

```
# /sbin/ifconfig -a
```

The ENET device default IP address is 192.168.0.128. Your system IP address must be on the same subnet (192.168.0.*). You can set a static IP address on your system as shown below:

```
# /sbin/ifconfig eth0 192.168.0.1 netmask 255.255.255.0 up
```

This sets the system IP address for the eth0 interface to 192.168.0.1.

The /Source folder contains all the Alta Layer 0 and Layer 1 source files, as well as one of the example programs (ADT_L1_1553_ex_bcbm1_Timing_Test.c).

The example program contains the "main" program.

You can use this example program or one of the others from the /Examples folder.
NOTE THAT YOU MUST EDIT THE EXAMPLE PROGRAM TO ENSURE THAT THE DEVID DEFINITION SPECIFIES THE ENET-1553 PRODUCT TYPE
(ADT_PRODUCT_ENET1553, ADT_PRODUCT_ENET2_1553,
ADT_PRODUCT_ENETA429, etc.).

Build the main program and API:

```
# gcc *.c
```

This will create the executable file "a.out". Run the program as follows:

```
# ./a.out
```

This of course is a very simple approach, you could use a make file to build as desired.

FOR WINDOWS:

The ENET device default IP address is 192.168.0.128. Your system IP address must be on the same subnet (192.168.0.*). Use the Windows Control Panel to assign a static IP address to your NIC, for example 192.168.0.1. Connect an Ethernet cable from your NIC to the ENET device.

We will use MSVS 2008 to setup a project to build and run the code as an example. Create a new project for a Console Application, for Application Settings uncheck "precompiled header" and check "empty project", click Finish - now you have an empty project.

Copy all the source files (from the Source folder provided with the Alta ENET API) to your project.

In the solution explorer add the two header files under Header Files.

Add all the C files under Source Files. Go to Project Properties -> Configuration Properties -> Linker -> Input, for Additional Dependencies enter "ws2_32.lib", click OK.

Edit ADT_L0.h. Comment/uncomment the #includes - for Windows you want string.h, memory.h, and malloc.h. Edit the define for ADT_L0_CALL_CONV - for Windows you want __stdcall.

Edit ADT_L0_BSD_ENET.c. Edit to define WINDOWS for the platform.

Edit the example program (ADT_L1_1553_ex_bcbm1_Timing_Test.c) and edit the DEVID definition to match your ENET product type.

Now you can build the project (Build -> Rebuild Solution).

Run the program - you will be prompted for the computer IP (192.168.0.1) and ENET IP (192.168.0.128) and the program will display timing info. If all this works you are up and running!

FOR VXWORKS:

The ENET device default IP address is 192.168.0.128. Your system IP address must be on the same subnet (192.168.0.*).

Setting the IP address of a network interface is BSP-dependent. Refer to the BSP documentation for your specific SBC.

For this example, we describe using Wind River Workbench to create a Downloadable Kernel Module (DKM).

In Project Explorer, select New->VxWorks Downloadable Kernel Module Project. Enter a project name in the subsequent dialog. Click Next, then Next again (do not reference subprojects).

Choose Build Defaults per your environment and click Next.

Choose the Build Support and build command for your environment and click Next.

Choose the Build Specs for your BSP and click Next.

Choose/select Build Target Name and Build Tool (normally Linker) and click Next.

Select Indexer choices and click Finish.

Select Working Sets if desired and click OK.

The Framework has now been created for the DKM project. Now we'll add the source and header files.

In Project Explorer, right-click on the DKM just created and select Import.

Choose File System, then click Next.

Browse to the location where you downloaded and unzipped the Alta BSD-Only API.

Select all *.c and *.h files from the Source folder to import. Click Finish.

Edit ADT_L0.h. Comment/uncomment the #includes - for VxWorks you want string.h and stdlib.h. Edit the define for ADT_L0_CALL_CONV - for VxWorks you want nothing.

Edit ADT_L0_BSD_ENET.c. Edit to define VXWORKS for the platform.

Edit the example program (ADT_L1_1553_ex_bcbm1_Timing_Test.c) and edit the DEVID definition to match your ENET product type.

Now we can build the DKM project:

In Project Explorer, right-click on the DKM just created and select Build Project.

Select Generate Includes in the subsequent dialog. * (Steps with * are first time only steps.)

Click Next in the Analyze include directives dialog.*

In the subsequent dialog, "Resolve include directives", if there are no unresolved directives, click Next. *

If there are unresolved directives, resolve them before you click Next.*

Click Finish in the "Specify include search path scope" dialog.*

After successful build, modify startup to load the DKM just created.

Execute the main() found in ADT_L1_1553_ex_bcbm1_Timing_Test.c.

NOTE: Some VxWorks systems will not allow main() as a function entry point. Modify accordingly if this applies.

When you run the program - you will be prompted for the computer IP (192.168.0.xx) and ENET IP (192.168.0.128) and the program will display timing info. If all this works you are up and running!

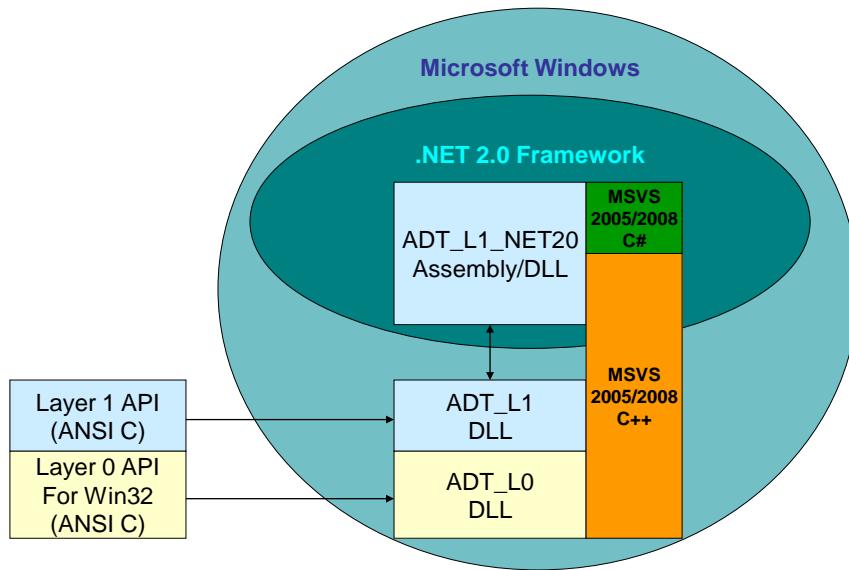
Appendix B – Microsoft Windows

This appendix discusses installation and operation under Microsoft Windows operating systems.

NOTE:

If you are only using ENET devices you can use the ENET-ONLY BSD API, discussed in [Appendix A](#). This API does not use a kernel-level driver and can be used on almost all platforms. You cannot use the ENET-ONLY BSD API with PCI/PCIe devices

Windows Module Architecture



Windows Device Drivers

Alta Data Technologies uses the **Jungo WinDriver** (version 12.2.1) driver development toolkit for Windows device drivers. These drivers will work for **Windows 7/8.1/10 (32-bit and 64-bit)**.

Windows System Requirements

Hardware:

- Computer with a 1.6 GHz or faster processor
- 1024 MB or more of RAM
- 3 GB or more of available hard-disk space
- 1024x768 or higher-resolution display with 256 colors, Direct-X 9 capable
- Keyboard and Microsoft Mouse or compatible pointing device

Architecture:

- 32-bit (x86) or 64-bit (x64)

Operating System:

- Microsoft® Windows 7™
- Microsoft® Windows Server™ 2008 R2
- Microsoft® Windows 8.1™
- Microsoft® Windows Server™ 2012 R2
- Microsoft® Windows 10™

Other:

- Microsoft® .NET Framework version 2.0 and 3.5
- Adobe® Reader (for PDF documentation)

Supported Alta Products

All Alta products are supported for Windows. This includes all PCI, PCI Express, and ENET products for MIL-STD-1553, WMUX, and A429.

Supported Compilers and Development Environments

Alta Data Technologies uses **Microsoft Visual Studio 2008 and 2010** for building and testing the API and example programs on Windows platforms. We do not provide specific support for any other development environments.

Windows Power Settings

If Windows goes into “sleep” mode to save power this will stop communication with the Alta device and will be a problem for any software using the device. Go to the Windows Control Panel, System and Security, Power Options. For your power plan, select “Change Plan Settings”. There are three settings – “Dim the display”, “Turn off the display”, and “Put the computer to sleep”. **Select “Never” for “Put the computer to sleep”.**

Installation of Alta Products

Updating to a different version of the Alta software/driver:

1. Shut down the system and remove all Alta cards from the system. This allows the driver to be uninstalled (cannot have any devices in the system using the driver).

*** Removing the Alta cards from the system is the preferred method, but if removing the Alta cards is not an option . . . ***

- a. Go to the Windows Device Manager.
 - b. Card(s) are listed under "Jungo". Right-click to uninstall the Alta card(s) here. You do not have to uninstall AltaDT.
2. Power up.
 3. If the cards were **not** removed from the system, check the Device Manager to verify that the card(s) are NOT seen under "Jungo".
 4. Go to the Windows Control Panel, select Uninstall a Program, and uninstall the old version of the Alta Software.
 5. **If you are removing a newer version to install an older version, go to C:\Windows\System32\drivers and remove the file AltaDT.sys.**
 6. Reboot.
 7. Install the new Alta Software. **You will need administrator privileges to install the software.**
 - a. If you are running **32-bit Windows 7/8.1/10**, open the **Win32** folder on the CD (or in the folder extracted from the downloaded .zip file) and run the Setup.exe program found here.
 - b. For **64-bit Windows 7/8.1**, open the **Win64\Win_7_8** folder on the CD (or in the folder extracted from the downloaded .zip file) and run the Setup.exe program found here.
 - c. For **64-bit Windows 10**, open the **Win64\Win_10** folder on the CD (or in the folder extracted from the downloaded .zip file) and run the Setup.exe program found here.

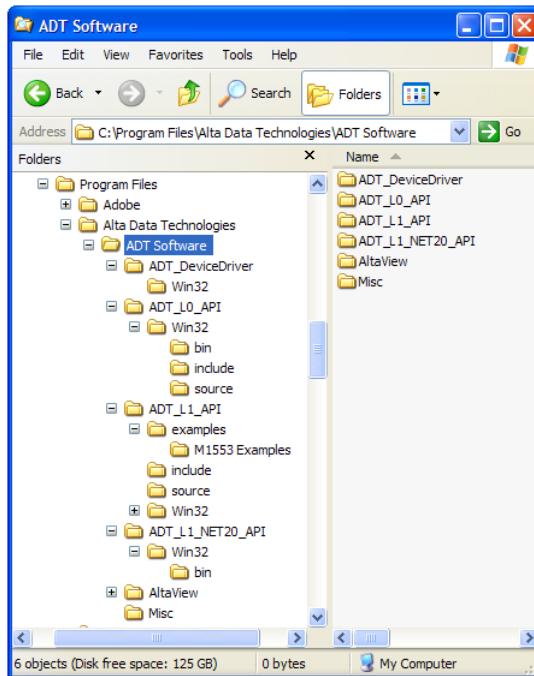
This will create the directory "C:\Program Files\Alta Data Technologies\Alta Software\" which will contain the directories for the Alta device drivers, APIs, and associated software.

8. Shut down the system and install the Alta card(s), previously removed.
9. Power up. Windows should detect the card(s) and perform plug and play configuration.
10. Check the Device Manager to verify that the card(s) are seen under "Jungo". If not, they are probably under "Other PCI Devices" - right-click and select Update Driver Software.
11. You can now use the AltaView software to open Alta devices and test the installation.

12. If you will be using the Alta API with software projects developed with a different version of the Alta software/driver, you will want to update your projects to use the newly installed versions of:
- ADT_L0.h
 - ADT_L0.lib
 - ADT_L0.dll
 - ADT_L1.h
 - ADT_L1.lib
 - ADT_L1.dll
 - For C#/.NET, update the ADT_L1_NET20 assembly as well.

First time installation of the Alta software, driver, and hardware on the system:

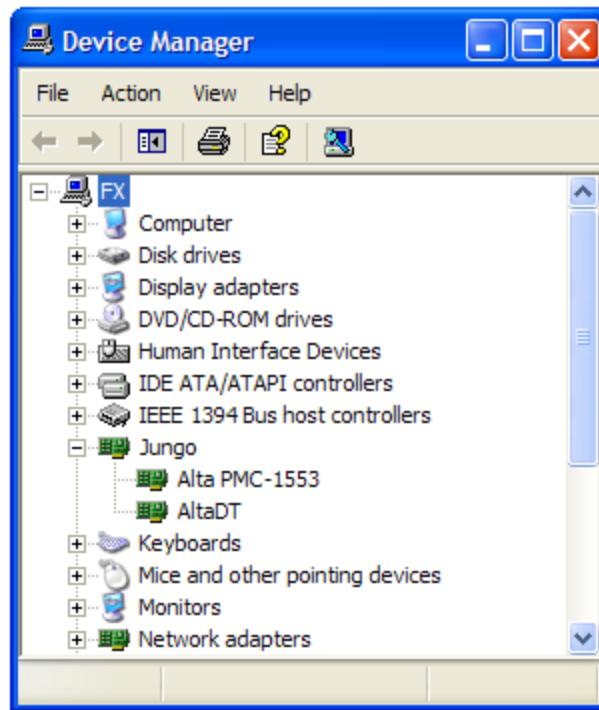
- First install the Alta Software installation package. You will need administrator privileges to install the software.
 - If you are running **32-bit Windows 7/8.1/10**, open the **Win32** folder on the CD and run the Setup.exe program found here.
 - For **64-bit Windows 7/8.1**, open the **Win64\Win_7_8** folder on the CD and run the Setup.exe program found here.
 - For **64-bit Windows 10**, open the **Win64\Win_10** folder on the CD and run the Setup.exe program found here.
 - This will create the directory “**C:\Program Files\Alta Data Technologies\Alta Software**” which will contain the directories for the Alta device drivers, APIs, and associated software.



The installer will also install the device driver for you. **For 7/8.1/10, make sure you have admin privileges (you may need to change roles to become administrator)!**

- Shut down the computer, install the board, turn on the computer.

3. The system should detect the new hardware. If the system asks for the inf file, it can be found in the “C:\Windows\inf” directory.
4. Installation is now complete. You can check the installation from the Windows Device Manager (go to the Windows Control Panel (Classic View), double-click the System icon, click the Hardware tab, click the Device Manager button). The Alta device will appear under “Jungo” as shown below:



You can also test the installation using the **AltaView** software that is installed as part of the Alta Software installation package – You should see the **AltaView & AltaRTVal** icons on the Windows desktop after the software installation.



AltaView is a Windows GUI bus analyzer application that controls the Alta boards. Refer to the **AltaView** Software User's Manual for more information. **AltaRTVal** is a Windows GUI SAE AS4111 5.2 Protocol RT Validation Test Application. Refer to the **AltaRTVal** Software User's Manual for more information.

To uninstall the Alta driver and hardware:

1. Turn off the computer and remove the hardware.
2. Turn on the computer.
3. Run the “Command Prompt” to run in console mode.
 - a. For **7/8.1/10**, right click to select “Run as Administrator”.
4. Change directory to the “C:\Program Files\Alta Data Technologies\Alta Software\ADT_DeviceDriver\WinXX” directory, where XX is 32 or 64.
5. Enter the following command: **wdreg –inf AltaDT_driver.inf uninstall**
6. You can uninstall the Alta Software package from the “Add/Remove Programs” option under the Windows Control Panel.

Anti-Virus Note

Some customers have noticed software functions, archiving and file access issues running anti-virus software. These programs can be very intrusive for file access and real-time applications. You may want to disable anti-virus software for real-time or critical file access applications.

Windows Layer 0 API Files

The Windows Layer 0 module consists of the following files:

- ADT_L0.h** – Top-level header file for Layer 0.
ADT_L0_Windows.c – Implementation of Layer 0 functions for Windows.
(Source code to interface with the Jungo WinDriver module is not distributed)

This is built into a Windows DLL file and LIB file. Higher level API layers use the following files:

- ADT_L0.h** – Top-level header file for Layer 0.
ADT_L0.lib – Windows library file for Layer 0.
ADT_L0.dll – Windows DLL file for Layer 0.

You will not normally write programs to use the Layer 0 API directly – typical applications will use the Layer 1 or higher level APIs.

Windows Layer 1 API Files

For Microsoft Windows environments, the Layer 1 API files are built into a Windows DLL file and LIB file. User applications and higher level API layers use the following files:

- ADT_L1.h** – Top-level header file for Layer 1.
ADT_L1.lib – Windows library file for Layer 1.
ADT_L1.dll – Windows DLL file for Layer 1.

You will also need the following Layer 0 files to run with the Layer 1 files:

- ADT_L0.h** – Top-level header file for Layer 0.
ADT_L0.dll – Windows DLL file for Layer 0.

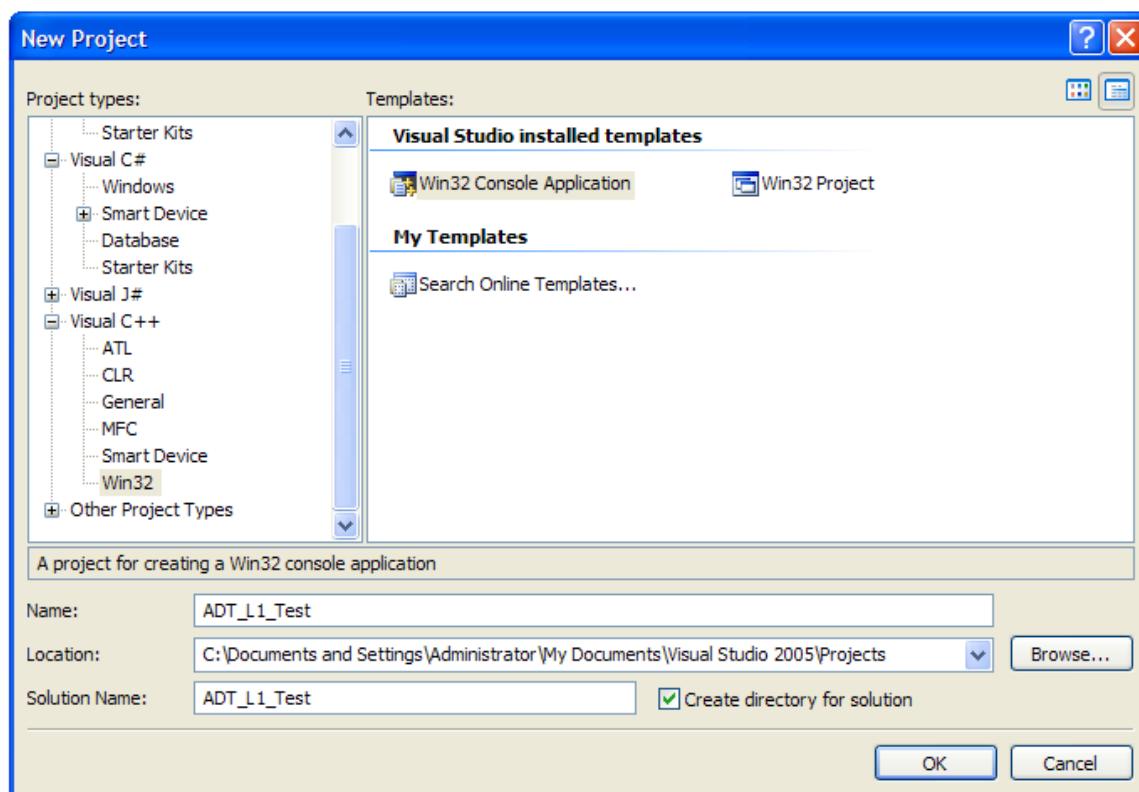
Layer 1 API Example Programs – **READ ME**

The Layer 1 API is distributed with example programs that can be used to test your installation and to help get started on developing your own programs. These examples can be found at “C:\Program Files\Alta Data Technologies\Alta Software\ADT_L1_API\examples”. Read-me text files are provided with the examples to describe what each example demonstrates. There is a key parameter called “Device ID – DEVID” that may need to be changed to match your card configuration. The readme file provide more information for this easy change. **USE THESE EXAMPLES TO GET STARTED QUICKLY – THERE ARE MANY EXAMPLES THAT WILL JUMP START MOST APPLICATIONS!!**

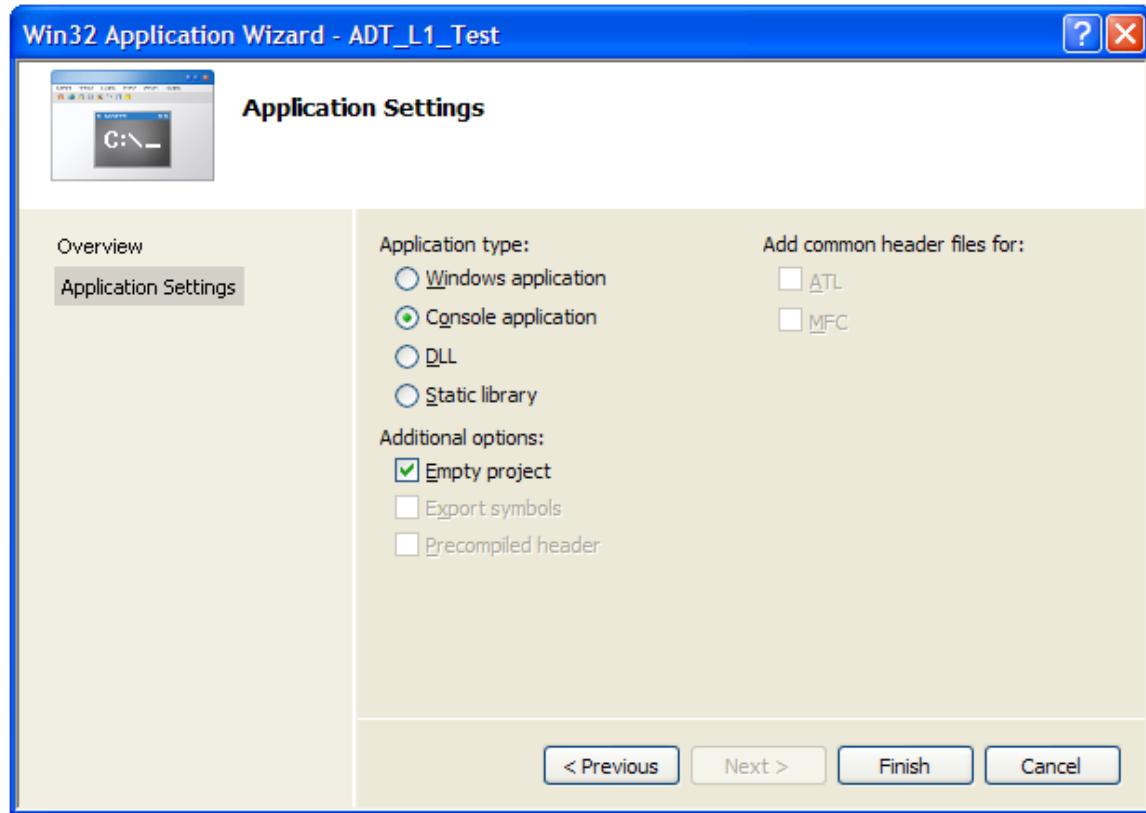
Using MSVS 2005/2008/2010/2012/2013/2015 C++ with the Layer 1 API

Microsoft Visual C++ can be used with the ANSI “C” Layer 1 API and examples. We will demonstrate setting up a project to build and run one of the Layer 1 example programs using Microsoft Visual Studio 2005 (Professional Edition).

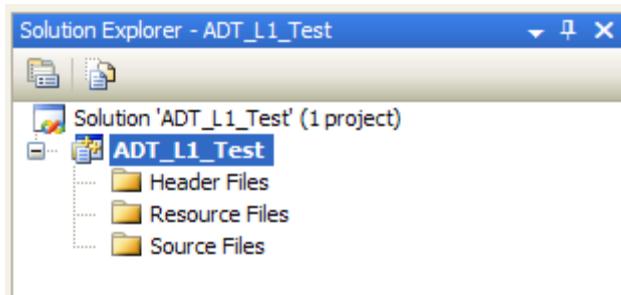
The Layer 1 API example programs will be built as Windows console applications. Create a MS Visual C++ Win32 console application project. Name it “ADT_L1_Test”.



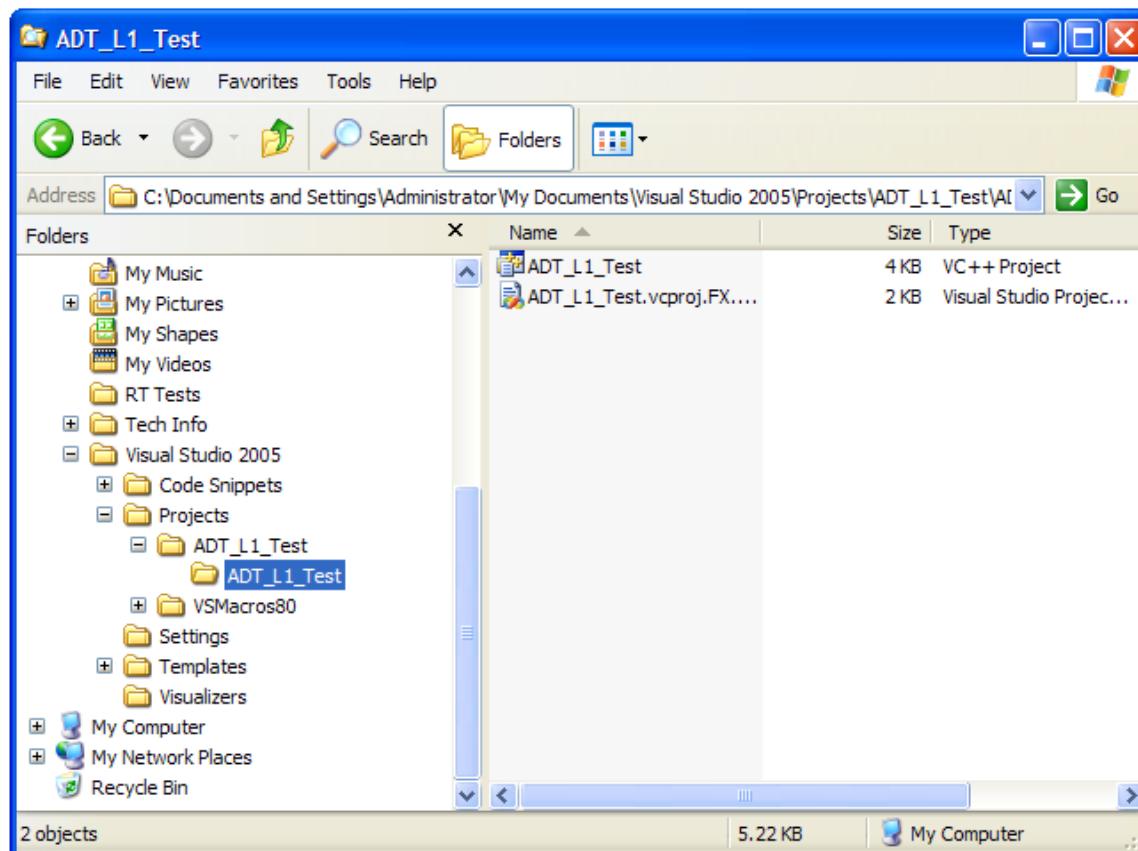
At the next window that pops up, select “Application Settings” (on the left side of the window). **Uncheck “Precompiled header” and check “Empty Project”**. Click “Finish”.



This will create an empty project.



This also creates our basic project directory tree.



Now we need to copy files from the Alta Software installation directories to our project directory. Copy the following API files and place them in the project directory:

For 32-bit Windows:

C:\Program Files\Alta Data Technologies\Alta Software\ADT_L0_API\Win32\include\ADT_L0.h
C:\Program Files\Alta Data Technologies\Alta Software\ADT_L0_API\Win32\bin\ADT_L0.dll
C:\Program Files\Alta Data Technologies\Alta Software\ADT_L1_API\include\ADT_L1.h
C:\Program Files\Alta Data Technologies\Alta Software\ADT_L1_API\Win32\bin\ADT_L1.lib
C:\Program Files\Alta Data Technologies\Alta Software\ADT_L1_API\Win32\bin\ADT_L1.dll

For 64-bit Windows:

C:\Program Files\Alta Data Technologies\Alta Software\ADT_L0_API\include\ADT_L0.h
C:\Program Files\Alta Data Technologies\Alta Software\ADT_L0_API\Win64\bin\ADT_L0.dll
C:\Program Files\Alta Data Technologies\Alta Software\ADT_L1_API\include\ADT_L1.h
C:\Program Files\Alta Data Technologies\Alta Software\ADT_L1_API\Win64\bin\ADT_L1.lib
C:\Program Files\Alta Data Technologies\Alta Software\ADT_L1_API\Win64\bin\ADT_L1.dll

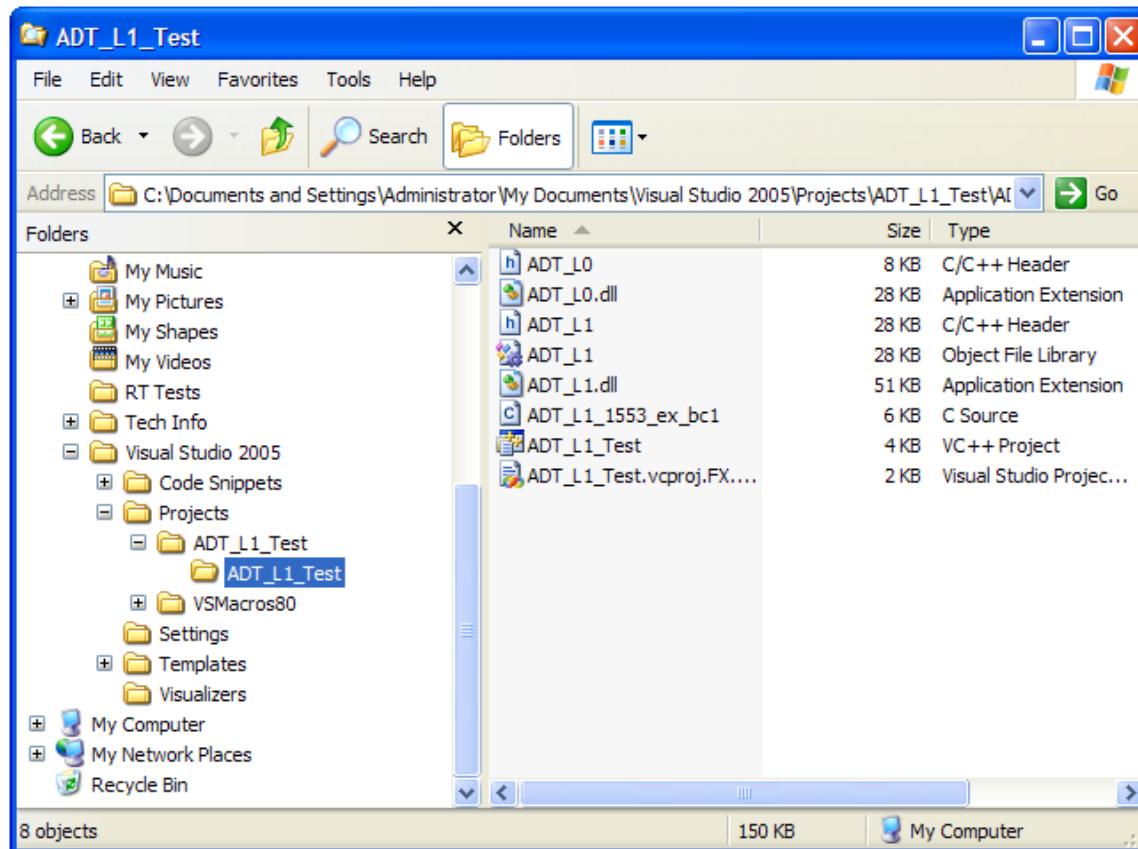
Note: If you are using the 64-bit Alta Software installation you will see “\Win64” rather than “\Win32”. Use the “\Win64” files if you are building a 64-bit application (x64 target platform).

As of version 2.6.0.0, the 64-bit installation provides lib/dll files for both “Win64” and “x86”. The “x86” files can be used to run 32-bit applications on 64-bit Windows – this allows applications developed on 32-bit Windows to run on 64-bit Windows in 32-bit “Windows On Windows” mode. The “Win64” files are used to run 64-bit applications on 64-bit Windows. See the application note “Creating-32bit-applications-on-64bit-Windows-systems.pdf” for details – this is available from the Knowledge Base on the Alta web site.

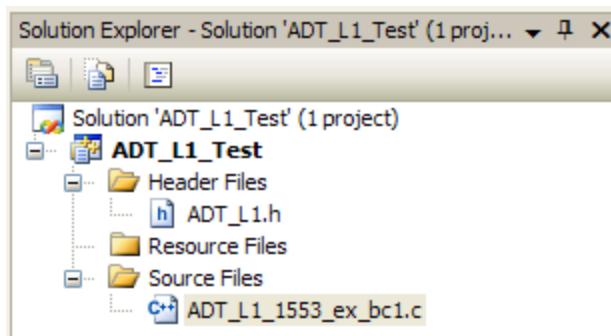
We also need one of the example programs:

```
C:\Program Files\Alta Data Technologies\Alta Software\ADT_L1_API\examples\  
M1553 Examples\ADT_L1_1553_ex_bc1.c
```

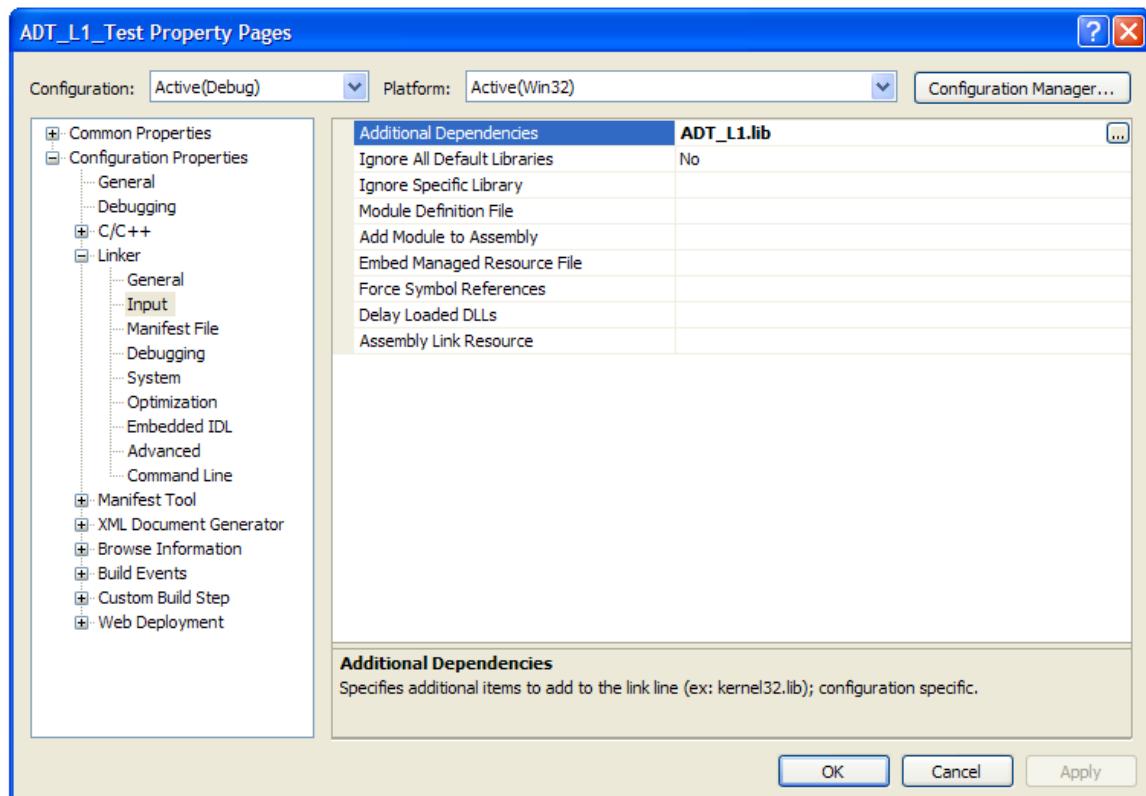
Now we should have the files we need in our project directory.



In the Microsoft Visual C++ environment we can add these files to the project. In the solution explorer, right-click on “Header Files”, select “Add Existing Item”, and add the ADT_L1.h file. Right-click on “Source Files”, select “Add Existing Item”, and add the example program. Now your solution explorer should look something like this.



We also have to configure the project to include the LIB file for the Layer 1 API. In the Solution Explorer, right-click on the project (ADT_L1_Test) then click on “Properties” to bring up the project property pages. Navigate the tree on the left to select “Input” (under Configuration Properties, Linker). In the “Additional Dependencies” field, type “ADT_L1.lib” as shown below. Click on the “OK” button.



Now your project setup is complete! **You should edit the example program and make sure it is using a device identifier that corresponds to an Alta device installed in your system.** If the DEVID is not correct you will see error 3 on initialization.

Note: If you are using the 64-bit Alta Software installation you will need to change the “Solution Platform” setting in your MSVS project from “Win32” to “x64” so that the project will build a 64-bit executable.

You can now build the example program.

You should copy the two DLL files (ADT_L0.dll and ADT_L1.dll) to the folder where the executable is generated (/ADT_L1_Test/Debug or /ADT_L1_Test/x64/Debug) to ensure that the EXE can find the DLLs at run time.

You can now use this project to build any of the example programs or to develop your own programs.

NOTE ON MSI INTERRUPTS:

PCI Express devices use Message Signalled Interrupts (MSI). If you have two (or more) independent applications controlling different channels of the same PCIe board, and two or more of these applications use interrupts, this will not work – will crash the system (because the last application to enable interrupts overwrites the setting made by the others).

You CAN use independent applications on different channels of the same PCIe board if no more than one of them uses interrupts.

You CAN use interrupts on multiple channels of the same PCIe board if all of the channels using interrupts are controlled by the same application.

You CANNOT use interrupts on multiple channels of the same PCIe board if the channels using interrupts are controlled by separate independent applications.

On Windows platforms there is a way to work around this problem (by changing the inf file for the device to use line-based interrupts rather than MSI interrupts). Contact Alta technical support for more information on this approach.

Windows .NET Support

For Microsoft Windows environments using the .NET 2.0 framework, the Layer 1 API is encapsulated in a .NET 2.0 assembly (managed DLL) that defines the namespace **AltaDataTech:API:Layer1**. This is provided in the file:

ADT_L1_NET20.dll - .NET 2.0 assembly containing Layer 1 functions.

Alta also provides assemblies for later .NET versions, like ADT_L1_NET40.dll for .NET 4.0 and ADT_L1_NET45.dll for .NET 4.5. MSVS 2010 defaults to .NET 4.0, MSVS 2012 defaults to .NET 4.5.

This is basically a set of .NET managed-code wrapper functions that encapsulate the unmanaged ADT_L1 Layer 1 API functions. This module can be used on Microsoft Windows platforms with Microsoft Visual Studio 2005 for C#, VB, C++ or any other language supporting .NET. For more information on the Microsoft .NET framework go to:

<http://msdn2.microsoft.com/en-us/netframework/default.aspx>

http://en.wikipedia.org/wiki/.NET_Framework

Note: The source code for the ADT_L1_NET20 assembly is not included in the software distribution. This source code is considered Alta proprietary and is not distributed without an appropriate non-disclosure agreement.

You will also need the following Layer 0 and Layer 1 files to run with the ADT_L1_NET20 assembly:

ADT_L0.dll – Windows DLL file for Layer 0.

ADT_L1.dll – Windows DLL file for Layer 1.

Layer 1 .NET API Example Programs – **READ ME**

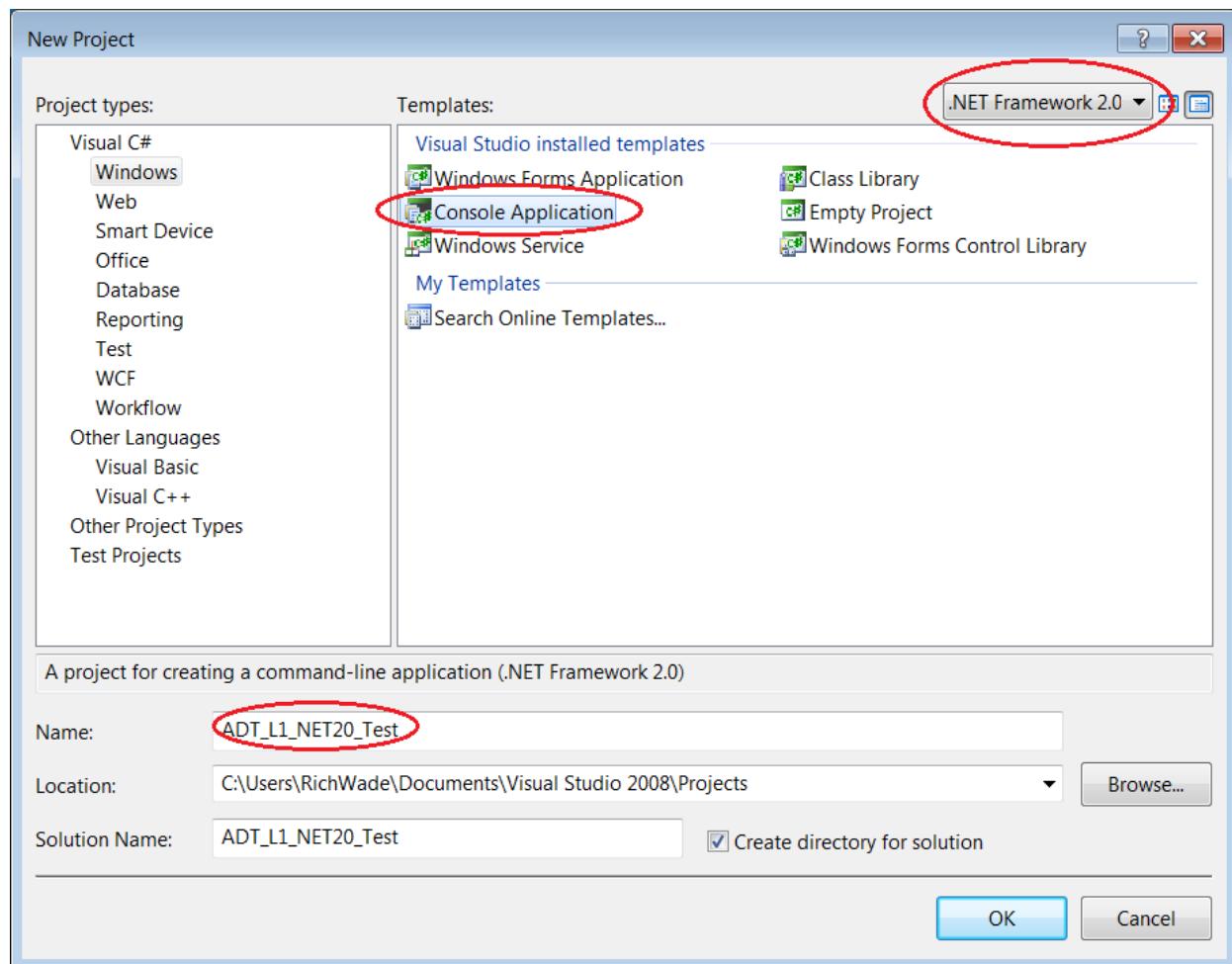
The Layer 1 .NET API is distributed with example programs that can be used to test your installation and to help get started on developing your own programs. These examples can be found at “C:\Program Files\Alta Data Technologies\Alta Software\ADT_L1_NET20_API\examples”. Read-me text files are provided with the examples to describe what each example demonstrates. There is a key parameter called “Device ID – DEVID” that may need to be changed to match your card configuration. The readme file provide more information for this easy change. **USE THESE EXAMPLES TO GET STARTED QUICKLY – THERE ARE MANY EXAMPLES THAT WILL JUMP START MOST APPLICATIONS!!**

Using MSVS 2005/2008 C# with the Layer 1 .NET 2.0 API

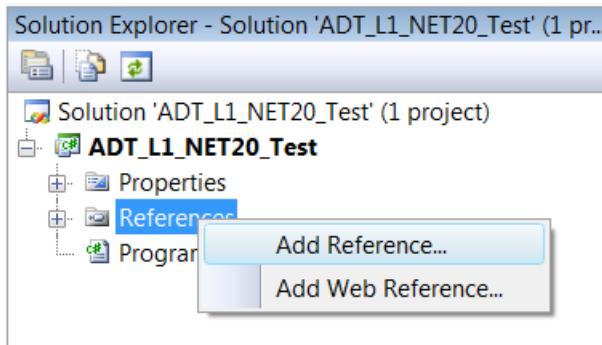
Microsoft Visual Studio 2005/2008 can be used to develop .NET 2.0 applications using the ADT_L1_NET20 assembly. We will demonstrate setting up a project to build and run a simple C# .NET 2.0 program using the Alta APIs (with Microsoft Visual Studio 2008 Professional Edition).

Create a Microsoft Visual C# Windows console application project. Name it “ADT_L1_NET20_Test”.

SELECT A TARGET FRAMEWORK OF .NET FRAMEWORK 2.0.



This will create our project with a template C# file called “program.cs”. We need to link our project to the ADT_L1_NET20 assembly so we can access the API from our program. In the solution explorer, right click on “References” and select “Add Reference...”.



This brings up the “Add Reference” dialog box. Select the “Browse” tab and go to the directory:

For 32-bit Windows:

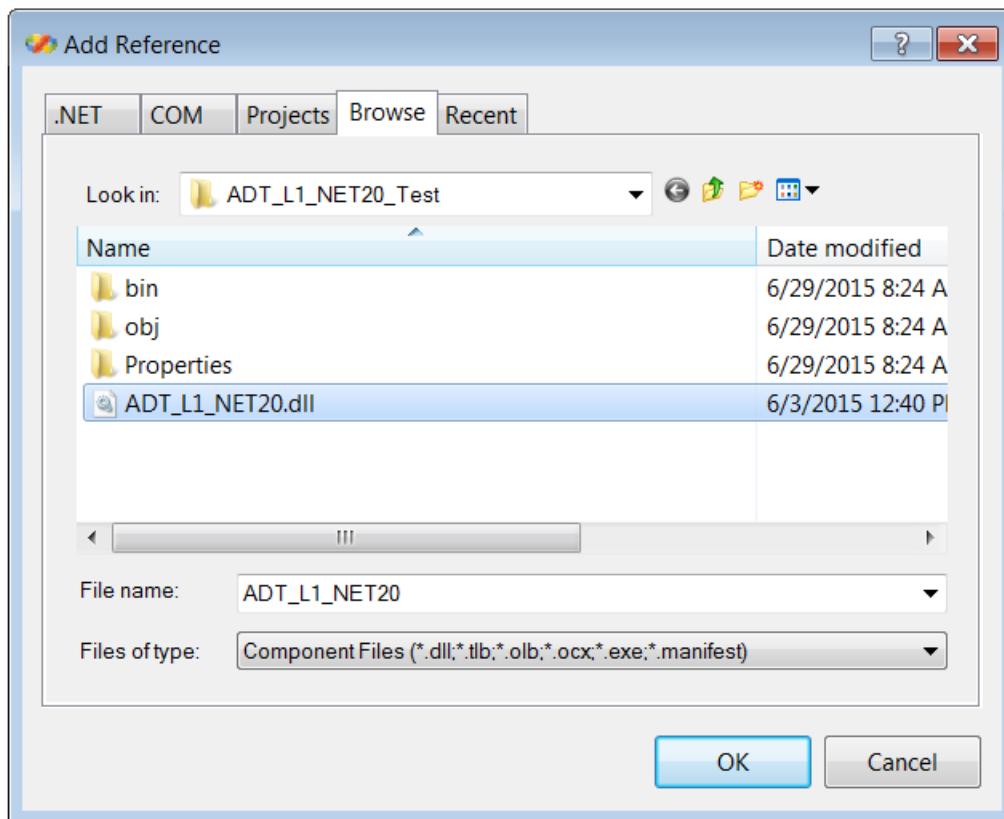
C:\Program Files\Alta Data Technologies\Alta Software\ADT_L1_NET20_API\Win32\bin

For 64-bit Windows:

C:\Program Files\Alta Data Technologies\Alta Software\ADT_L1_NET20_API\Win64\bin

Note that you can copy the ADT_L1_NET20.dll file to your project directory and reference it there.

Select the file “ADT_L1_NET20.dll” as shown below. Click the “OK” button.



We also need the DLL files for the Layer 0 and Layer 1 APIs. Copy these files into the \bin\debug directory for the project:

For 32-bit Windows:

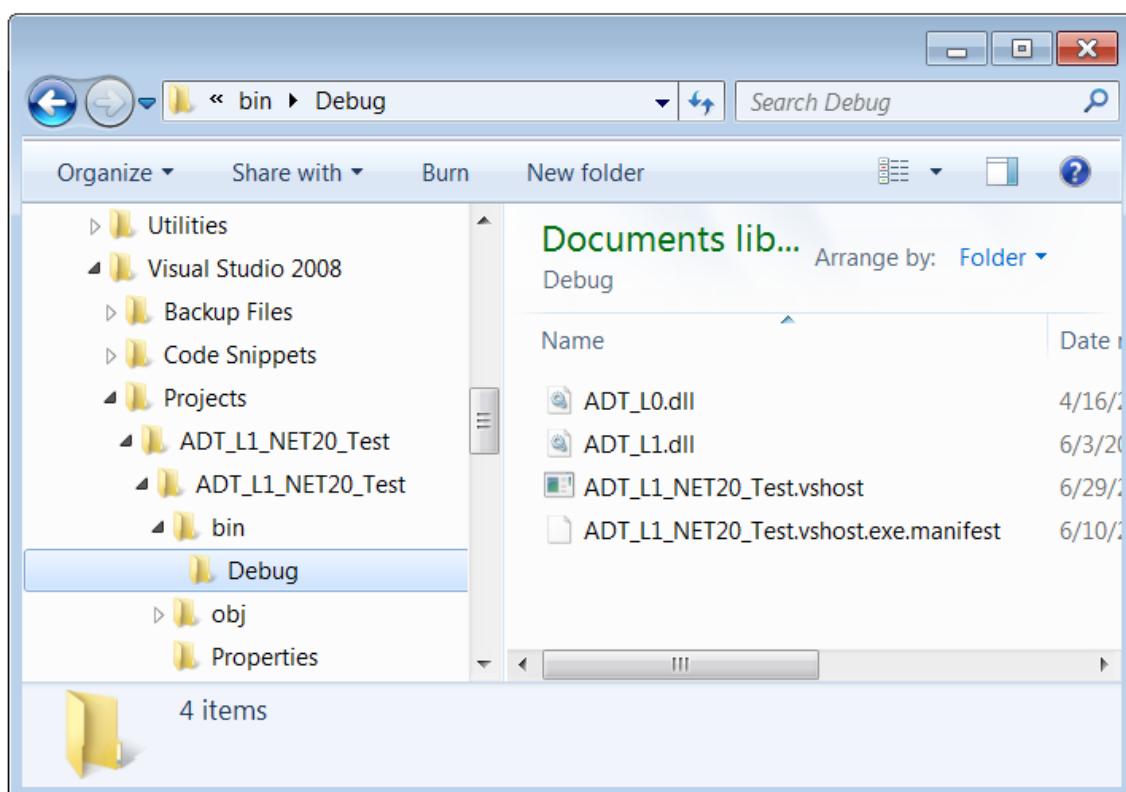
C:\Program Files\Alta Data Technologies\Alta Software\ADT_L0_API\Win32\bin\ADT_L0.dll
C:\Program Files\Alta Data Technologies\Alta Software\ADT_L1_API\Win32\bin\ADT_L1.dll

For 64-bit Windows:

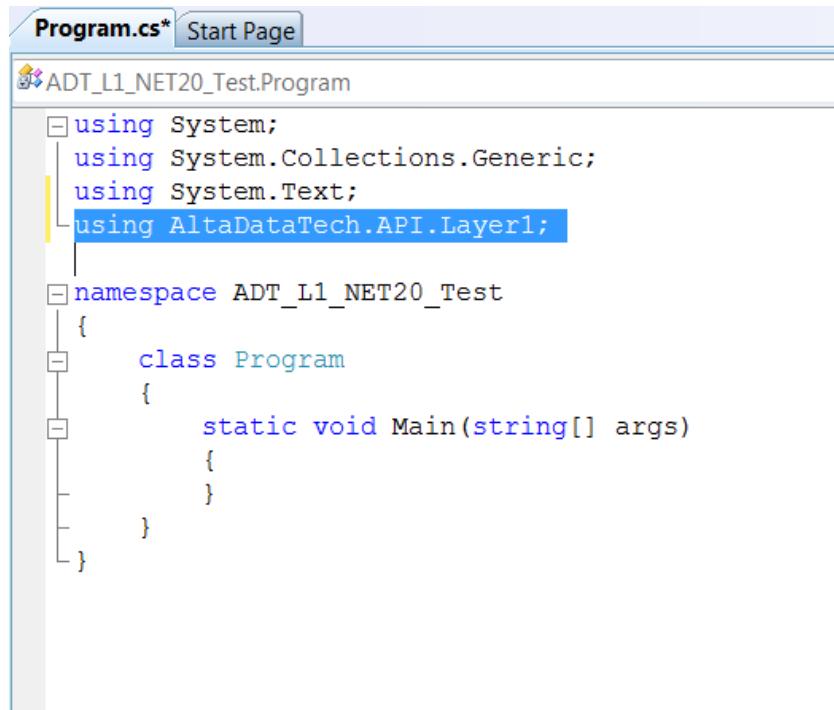
C:\Program Files\Alta Data Technologies\Alta Software\ADT_L0_API\Win64\bin\ADT_L0.dll
C:\Program Files\Alta Data Technologies\Alta Software\ADT_L1_API\Win64\bin\ADT_L1.dll

Note: When you use a release build of your application you will need these DLLs in the \bin\release directory for your project. If you distribute the final application, these DLLs and the ADT_L1_NET20 assembly should be in the same directory as the executable file for the application.

Now our project directory looks something like this:



Now let's modify the shell C# program to use the Alta API. First we will add a "using" statement to tell the program that we are using the Alta Layer 1 API assembly, as shown below:



The screenshot shows a C# code editor window titled "Program.cs*". The code is as follows:

```
using System;
using System.Collections.Generic;
using System.Text;
using AltaDataTech.API.Layer1;

namespace ADT_L1_NET20_Test
{
    class Program
    {
        static void Main(string[] args)
        {
        }
    }
}
```

The line "using AltaDataTech.API.Layer1;" is highlighted with a yellow background.

Modify your main program as follows:

```
using System;
using System.Collections.Generic;
using System.Text;
using AltaDataTech.API.Layer1;

namespace ADT_L1_NET20_Test
{
    class Program
    {

        static void Main(string[] args)
        {
            ADT_L1 api = new ADT_L1();          // The ADT_L1 class contains the Layer 1 API functions
            UInt32 status = ADT_L1.SUCCESS;

            // Make sure the DEVICE ID parameter is valid for your device.
            UInt32 devid = ADT_L1.DEVID_PRODUCT_ENETA429P |
                            ADT_L1.DEVID_BOARDNUM_01 |
                            ADT_L1.DEVID_CHANNELTYPE_A429 |
                            ADT_L1.DEVID_BANK_01;

            Console.WriteLine("ADT Layer 1 .NET 2.0 Test Program");

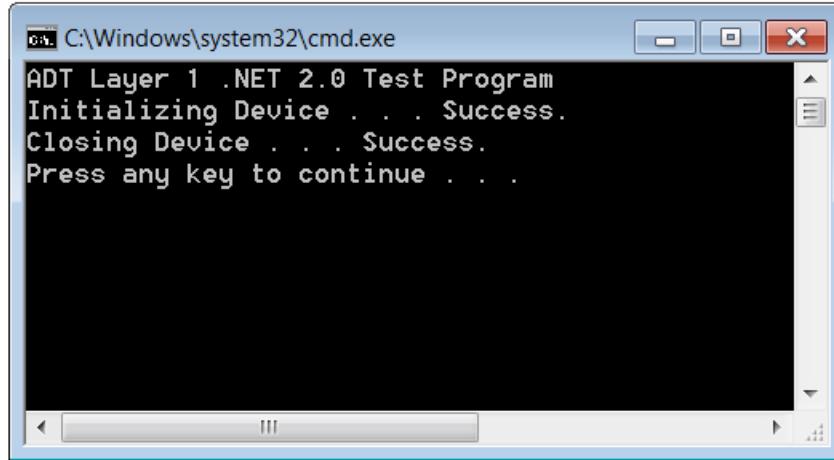
            /* For ENET devices, you |must first specify the IP addresses */
            if ((devid & 0xF0000000) == ADT_L1.DEVID_BACKPLANETYPE_ENET)
            {
                // Set Server IP to ENET IP, Client IP to local IP
                // This example uses Server at 192.168.0.128, Client at 192.168.0.2
                Console.Write("Setting IP Addresses . . . ");
                status = api.ENET_SetIpAddr(devid, 0x0A80080, 0x0A80002);
                if (status == ADT_L1.SUCCESS) Console.WriteLine("Success");
                else Console.WriteLine("FAILURE! Error = " + status.ToString());
            }

            // Use the API to initialize a device
            Console.Write("Initializing Device . . . ");
            status = api.InitDevice(devid, 0);
            if (status == ADT_L1.SUCCESS)
            {
                Console.WriteLine("Success");

                /* Close the device */
                Console.Write("Closing Device . . . ");
                status = api.CloseDevice(devid);
                if (status == ADT_L1.SUCCESS) Console.WriteLine("Success");
                else Console.WriteLine("FAILURE! Error = " + status.ToString());
            }
            else Console.WriteLine("FAILURE! Error = " + status.ToString());
        }
    }
}
```

This instantiates an ADT_L1 object called “api”. We use this object to call methods that correspond to Layer 1 API functions. In this simple example all we do is initialize a device then close it, but this shows how to access the API from C# using the ADT_L1_NET20 assembly.

As noted in the program comments, you should verify that you use a Device ID that is valid for an Alta board installed in your system. When you run the program you should see something like this:



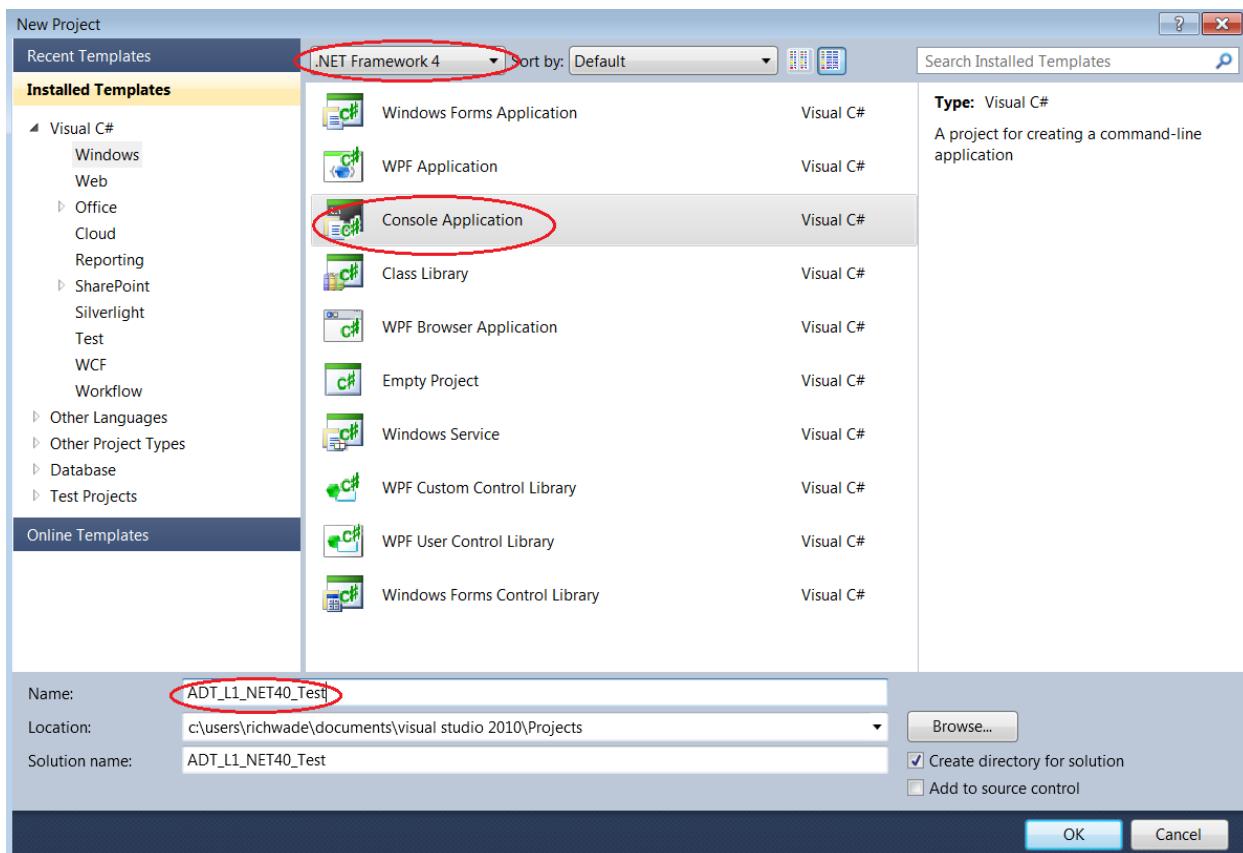
You should now be able to use the provided C# example programs and use the Alta API in your own programs written in C# (or any other .NET language).

Using MSVS 2010 C# with the Layer 1 .NET 4.0 API

Microsoft Visual Studio 2010 can be used to develop .NET 4.0 applications using the ADT_L1_NET40 assembly. We will demonstrate setting up a project to build and run a simple C# .NET 4.0 program using the Alta APIs (with Microsoft Visual Studio 2010 Professional Edition).

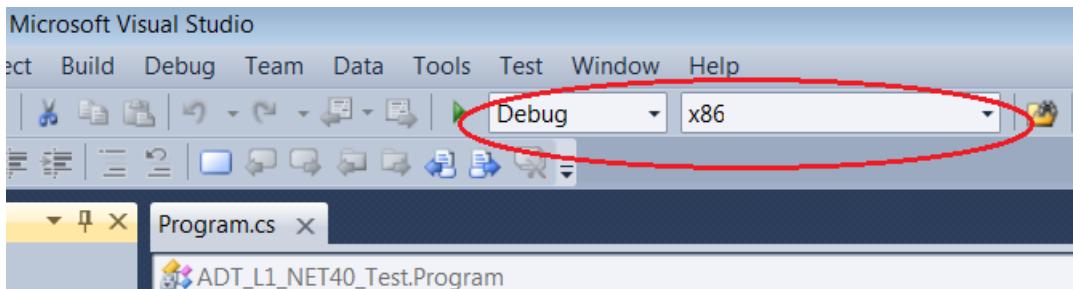
Create a Microsoft Visual C# Windows console application project. Name it "ADT_L1_NET40_Test".

SELECT A TARGET FRAMEWORK OF .NET FRAMEWORK 4.0.

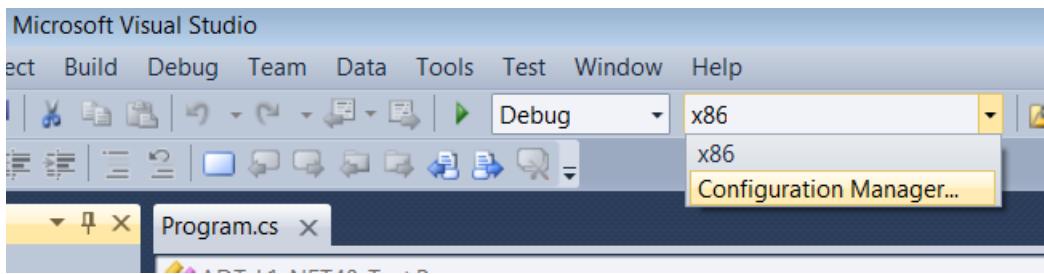


This will create our project with a template C# file called "program.cs".

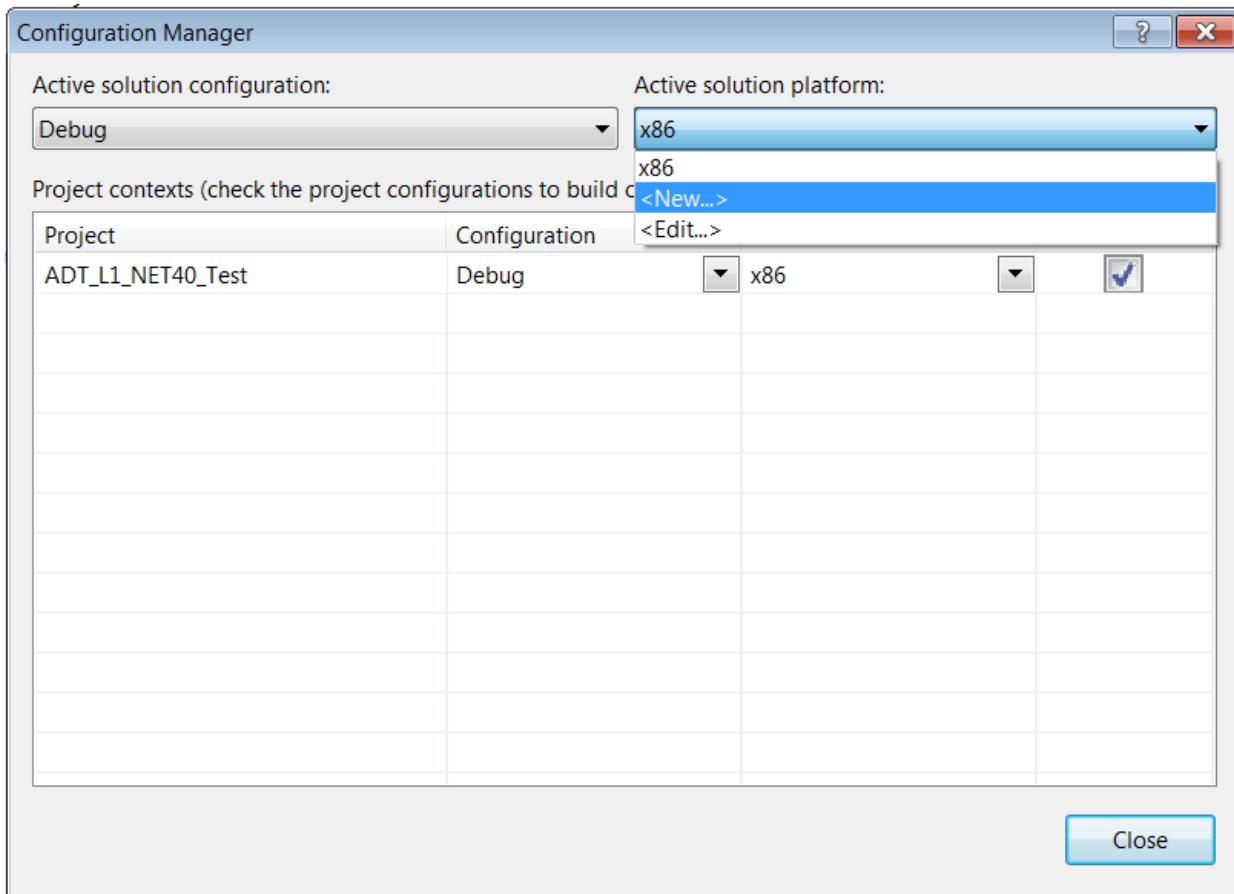
MSVS will default to an x86 Solution Platform.



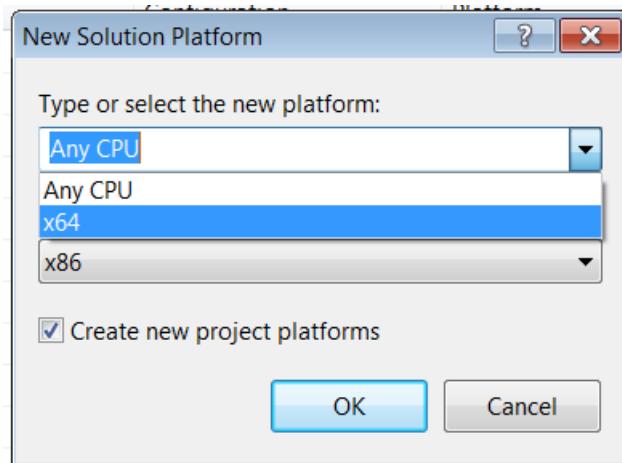
If you are using 64-bit Windows you need to change the Solution Platform setting to x64.



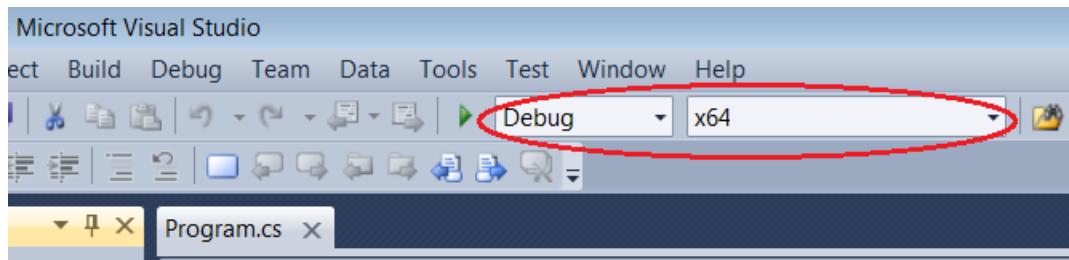
Select the "Configuration Manager".



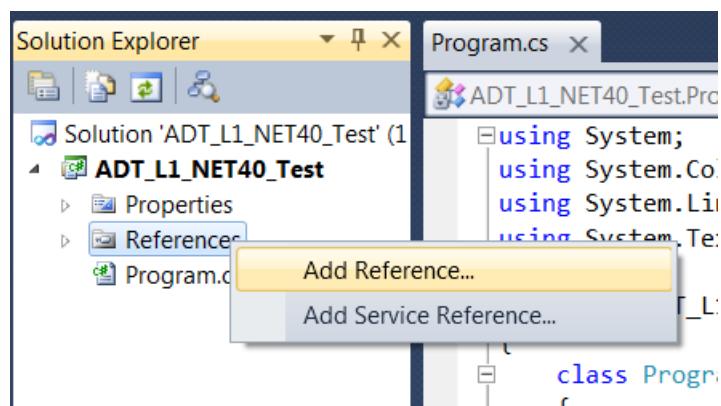
Under Active Solution Platform, select “<New...>.”



Select x64 and click OK. Close the Configuration Manager. Now we should have x64 as the Solution Platform.

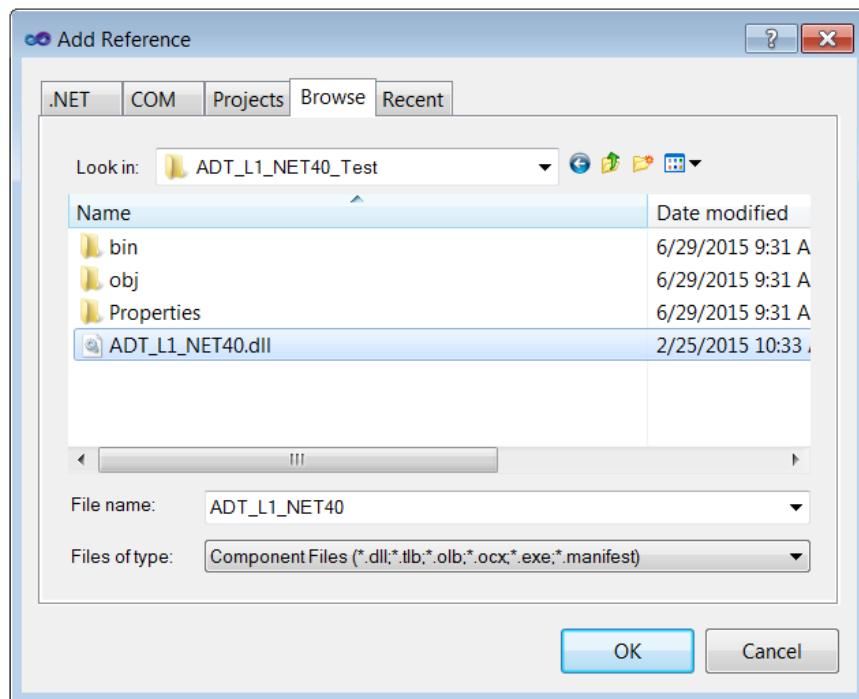


We need to link our project to the ADT_L1_NET40 assembly so we can access the API from our program. First copy the ADT_L1_NET40.dll file to your project directory. In the solution explorer, right click on “References” and select “Add Reference...”.



This brings up the “Add Reference” dialog box. Select the “Browse” tab and go to the project directory.

Select the file “ADT_L1_NET40.dll” as shown below. Click the “OK” button.



We also need the DLL files for the Layer 0 and Layer 1 APIs. Copy these files into the \bin\debug (\bin\x64\debug for 64-bit) directory for the project:

For 32-bit Windows:

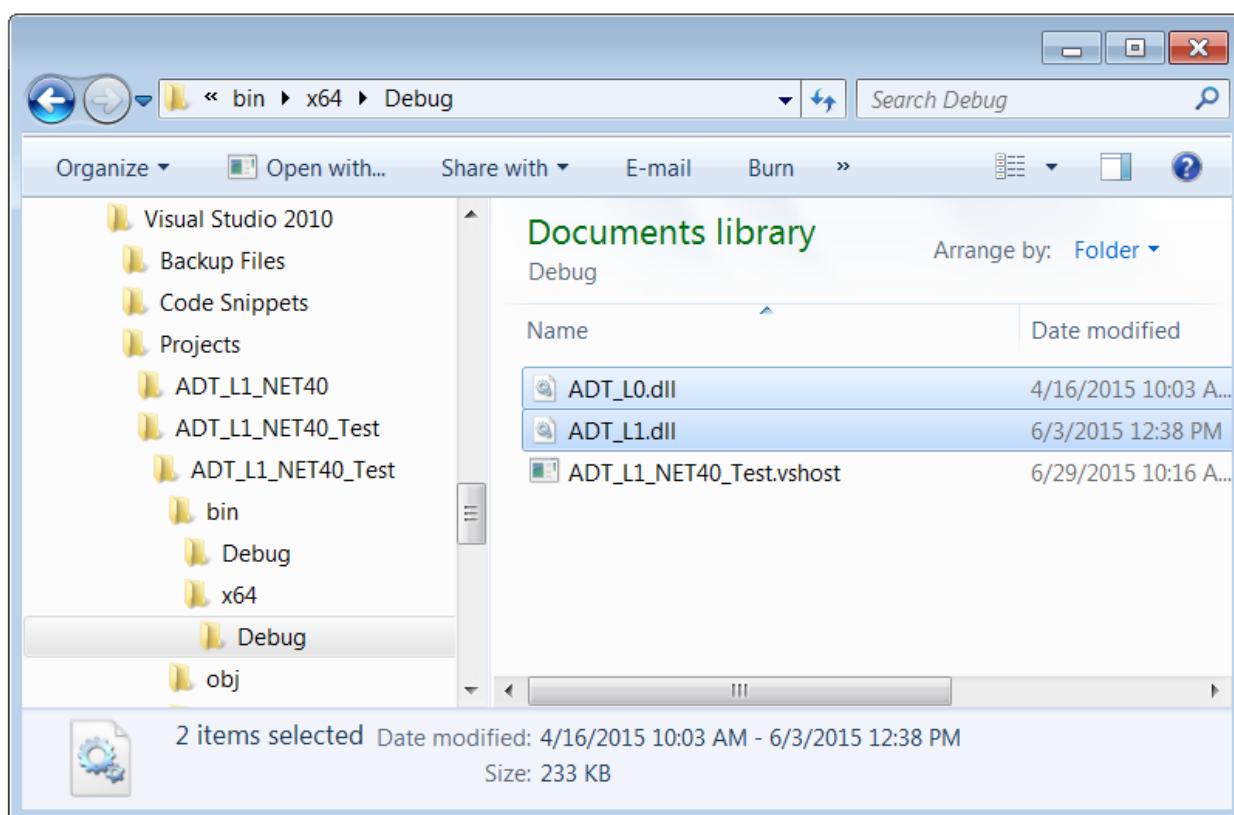
C:\Program Files\Alta Data Technologies\Alta Software\ADT_L0_API\Win32\bin\ADT_L0.dll
C:\Program Files\Alta Data Technologies\Alta Software\ADT_L1_API\Win32\bin\ADT_L1.dll

For 64-bit Windows:

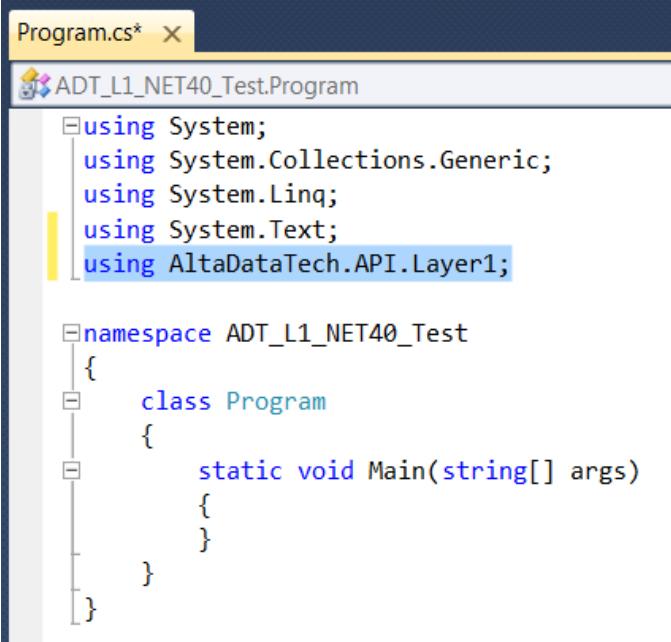
C:\Program Files\Alta Data Technologies\Alta Software\ADT_L0_API\Win64\bin\ADT_L0.dll
C:\Program Files\Alta Data Technologies\Alta Software\ADT_L1_API\Win64\bin\ADT_L1.dll

Note: When you use a release build of your application you will need these DLLs in the \bin\release directory for your project. If you distribute the final application, these DLLs and the ADT_L1_NET40 assembly should be in the same directory as the executable file for the application.

Now our project directory looks something like this:



Now let's modify the shell C# program to use the Alta API. First we will add a "using" statement to tell the program that we are using the Alta Layer 1 API assembly, as shown below:



```
Program.cs* X
ADT_L1_NET40_Test.Program
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using AltaDataTech.API.Layer1;

namespace ADT_L1_NET40_Test
{
    class Program
    {
        static void Main(string[] args)
        {
        }
    }
}
```

Modify your main program as follows:

```
using System;
using System.Collections.Generic;
using System.Text;
using AltaDataTech.API.Layer1;

namespace ADT_L1_NET40_Test
{
    class Program
    {

        static void Main(string[] args)
        {
            ADT_L1 api = new ADT_L1();          // The ADT_L1 class contains the Layer 1 API functions
            UInt32 status = ADT_L1.SUCCESS;

            // Make sure the DEVICE ID parameter is valid for your device.
            UInt32 devid = ADT_L1.DEVID_PRODUCT_ENETA429P |
                            ADT_L1.DEVID_BOARDNUM_01 |
                            ADT_L1.DEVID_CHANNELTYPE_A429 |
                            ADT_L1.DEVID_BANK_01;

            Console.WriteLine("ADT Layer 1 .NET 4.0 Test Program");

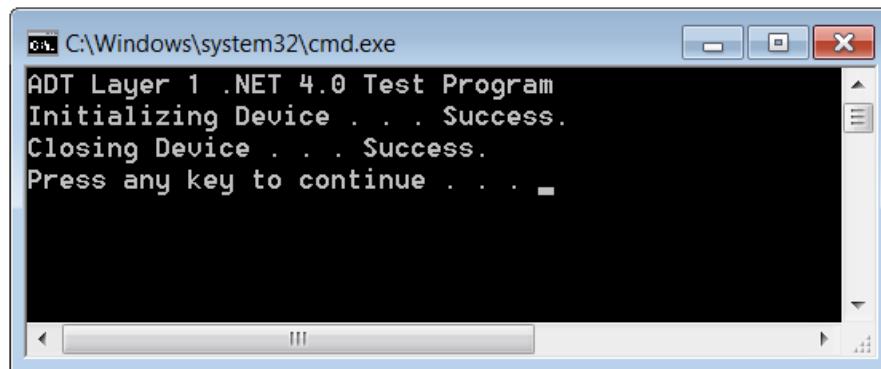
            /* For ENET devices, you must first specify the IP addresses */
            if ((devid & 0xF0000000) == ADT_L1.DEVID_BACKPLANETYPE_ENET)
            {
                // Set Server IP to ENET IP, Client IP to local IP
                // This example uses Server at 192.168.0.128, Client at 192.168.0.2
                Console.Write("Setting IP Addresses . . . ");
                status = api.ENESETIPAddr(devid, 0xC0A80080, 0xC0A80002);
                if (status == ADT_L1.SUCCESS) Console.WriteLine("Success");
                else Console.WriteLine("FAILURE! Error = " + status.ToString());
            }

            // Use the API to initialize a device
            Console.Write("Initializing Device . . . ");
            status = api.InitDevice(devid, 0);
            if (status == ADT_L1.SUCCESS)
            {
                Console.WriteLine("Success");

                /* Close the device */
                Console.Write("Closing Device . . . ");
                status = api.CloseDevice(devid);
                if (status == ADT_L1.SUCCESS) Console.WriteLine("Success");
                else Console.WriteLine("FAILURE! Error = " + status.ToString());
            }
            else Console.WriteLine("FAILURE! Error = " + status.ToString());
        }
    }
}
```

This instantiates an ADT_L1 object called “api”. We use this object to call methods that correspond to Layer 1 API functions. In this simple example all we do is initialize a device then close it, but this shows how to access the API from C# using the ADT_L1_NET40 assembly.

As noted in the program comments, you should verify that you use a Device ID that is valid for an Alta board installed in your system. When you run the program you should see something like this:



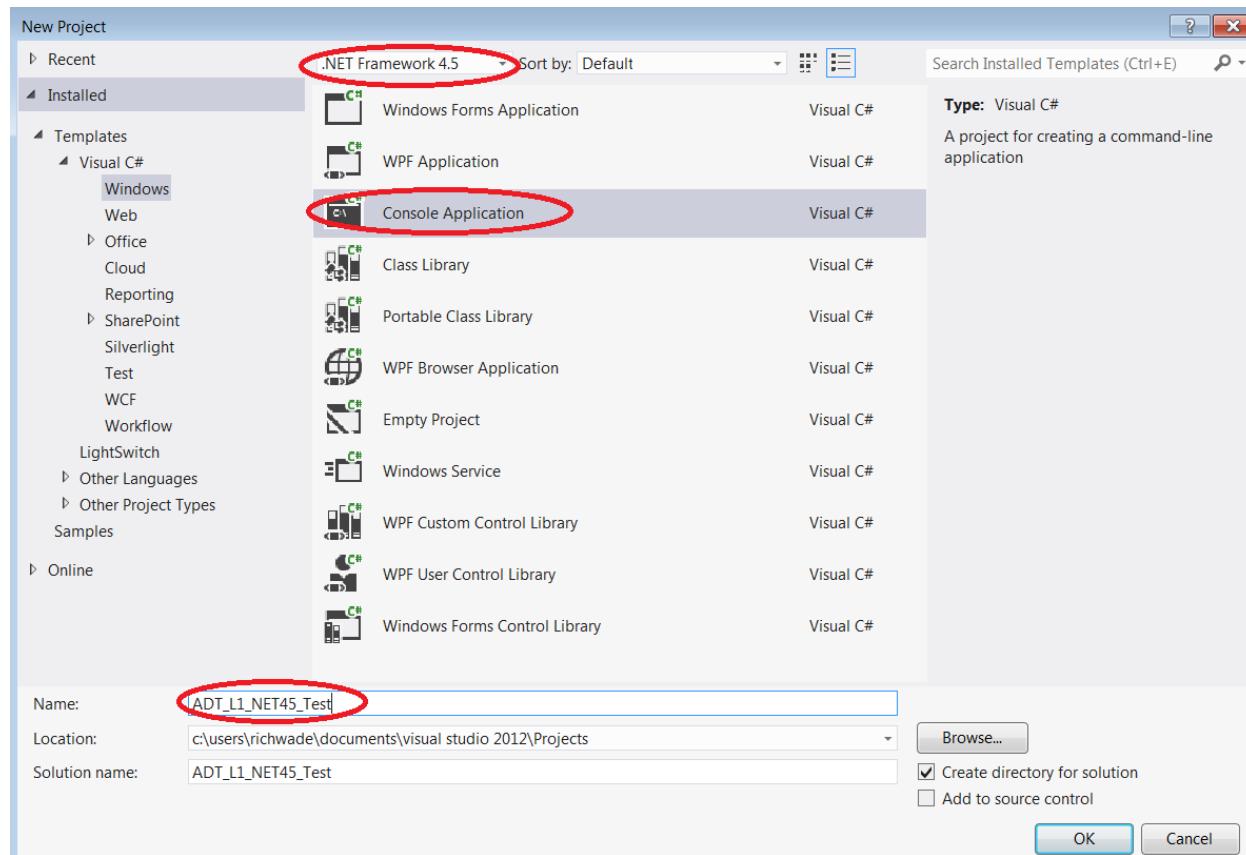
You should now be able to use the provided C# example programs and use the Alta API in your own programs written in C# (or any other .NET language).

Using MSVS 2012 C# with the Layer 1 .NET 4.5 API

Microsoft Visual Studio 2012 can be used to develop .NET 4.5 applications using the ADT_L1_NET45 assembly. We will demonstrate setting up a project to build and run a simple C# .NET 4.5 program using the Alta APIs (with Microsoft Visual Studio 2012 Professional Edition).

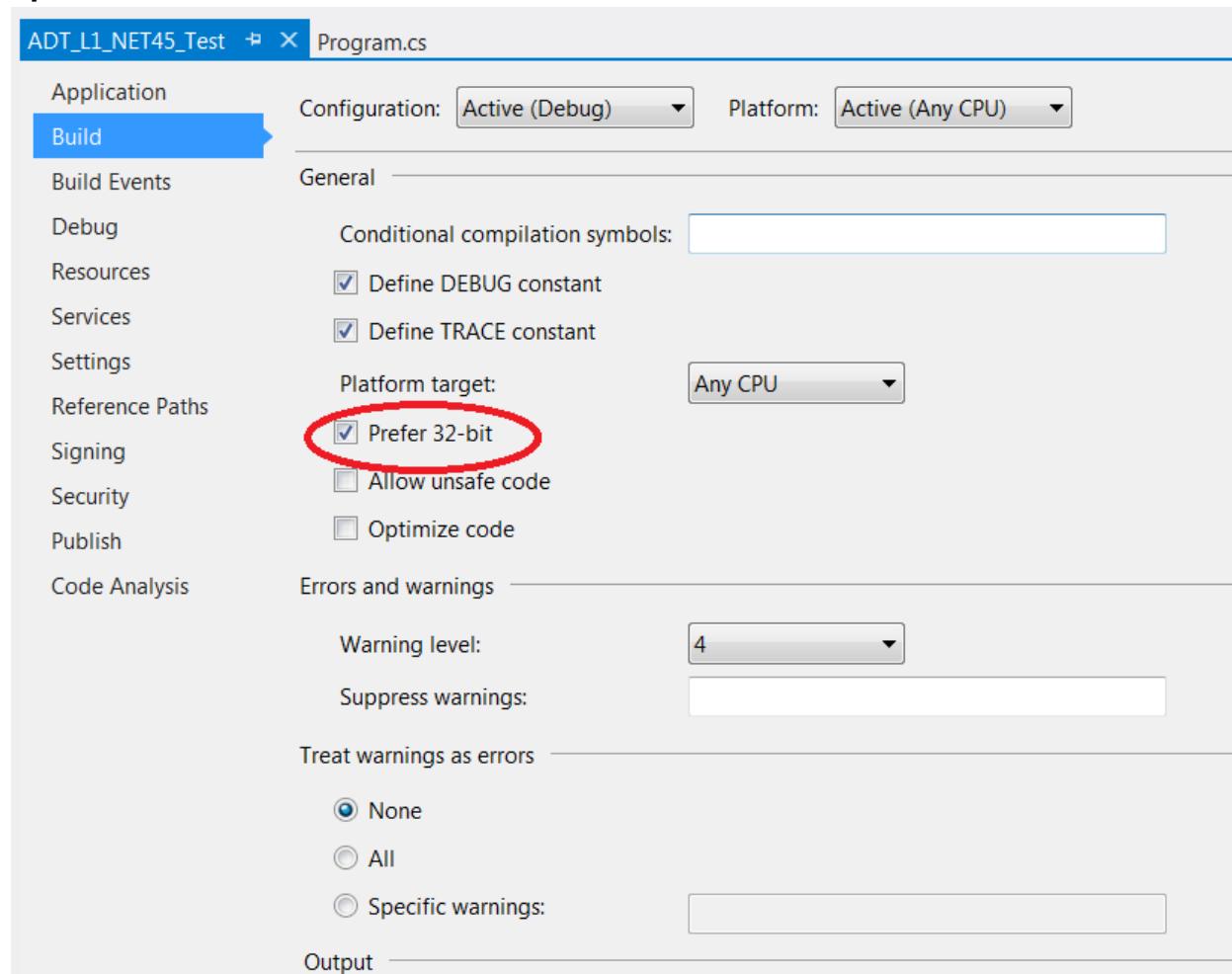
Create a Microsoft Visual C# Windows console application project. Name it “ADT_L1_NET45_Test”.

SELECT A TARGET FRAMEWORK OF .NET FRAMEWORK 4.5.

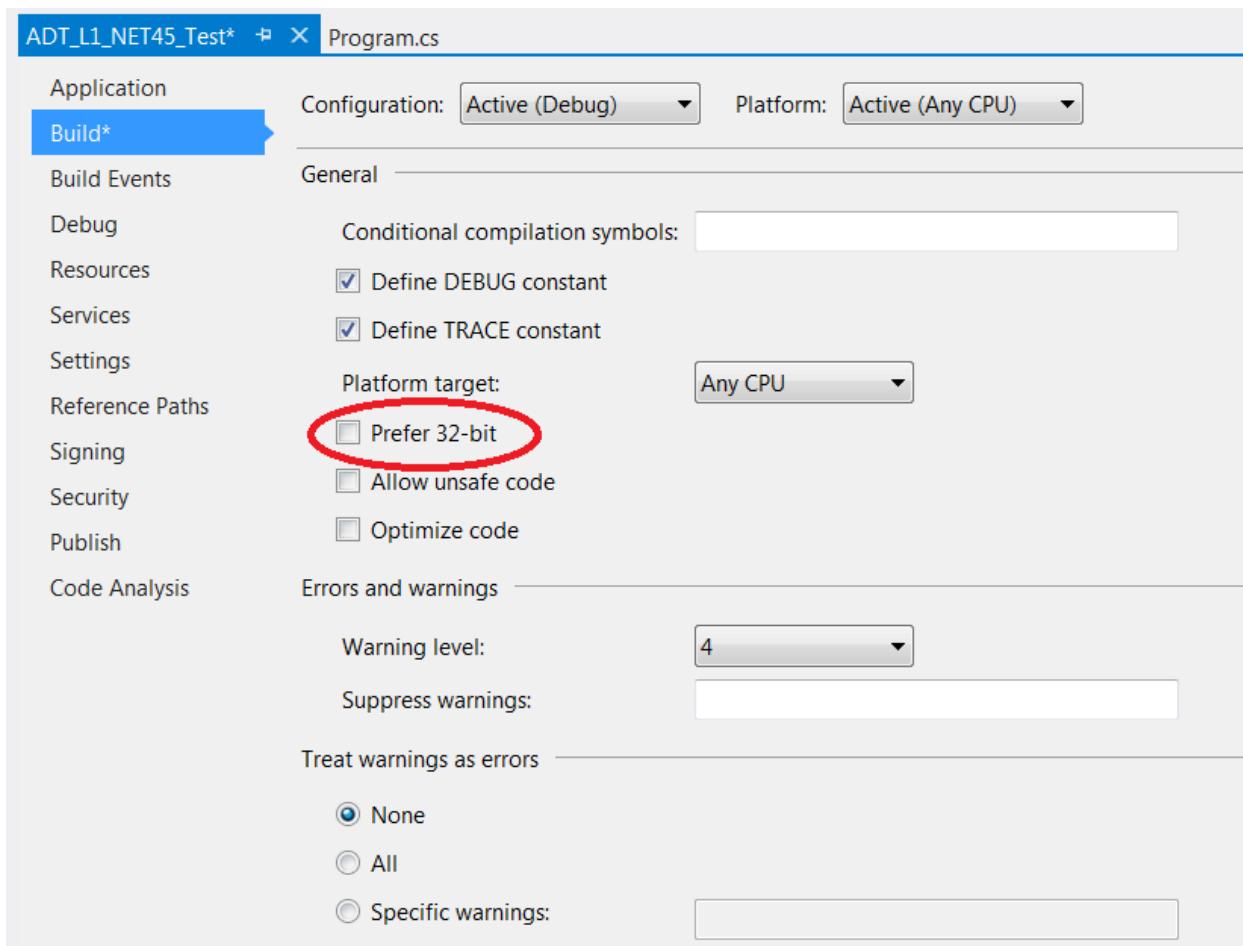


This will create our project with a template C# file called “program.cs”.

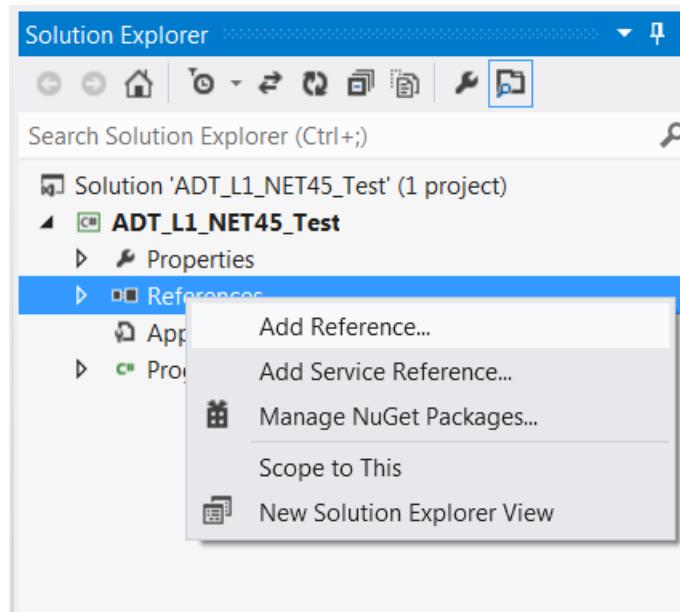
MSVS will default to an x86 32-bit build. Go to the project properties, “Build” options.



If you are using 64-bit Windows you need to use a 64-bit build. Uncheck “Prefer 32-bit”.

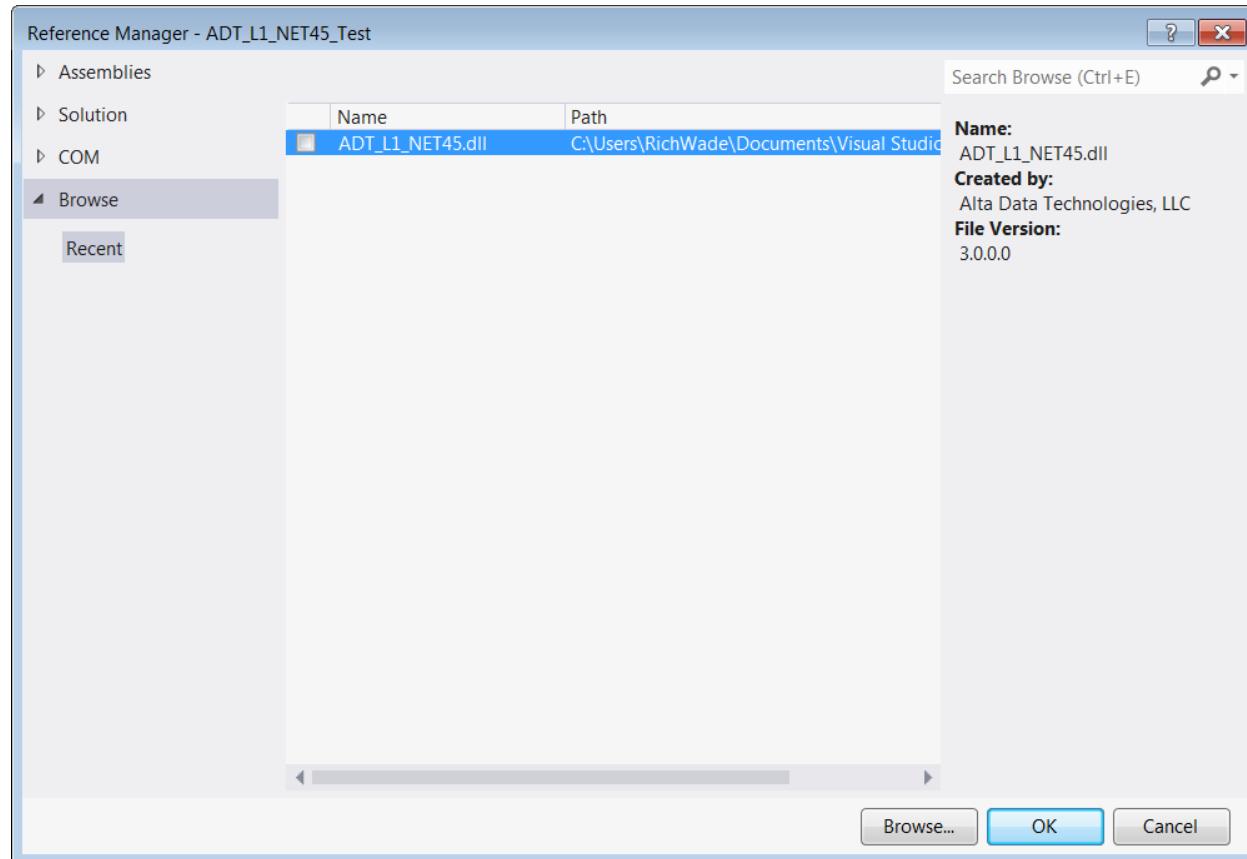


We need to link our project to the ADT_L1_NET45 assembly so we can access the API from our program. First copy the ADT_L1_NET45.dll file to your project directory. In the solution explorer, right click on “References” and select “Add Reference...”.



This brings up the “Add Reference” dialog box. Select the “Browse” tab and go to the project directory.

Select the file “ADT_L1_NET45.dll” as shown below. Click the “OK” button.



We also need the DLL files for the Layer 0 and Layer 1 APIs. Copy these files into the \bin\debug directory for the project:

For 32-bit Windows:

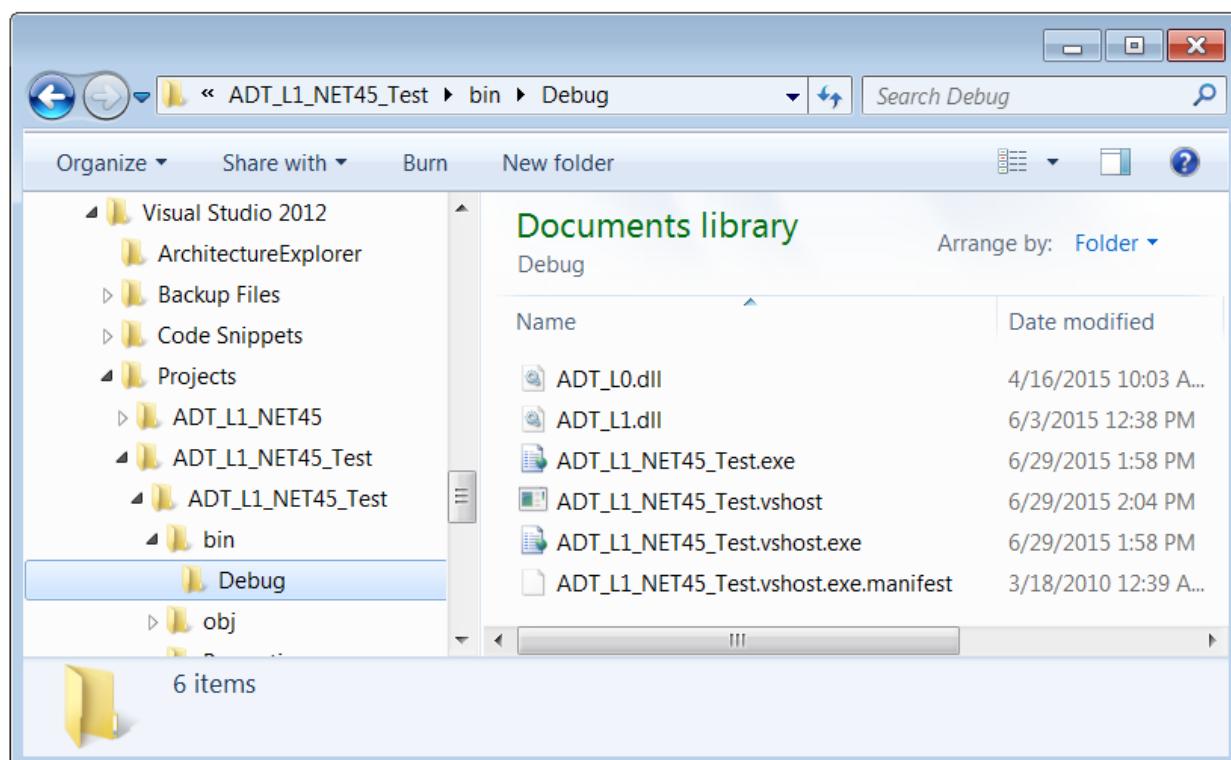
C:\Program Files\Alta Data Technologies\Alta Software\ADT_L0_API\Win32\bin\ADT_L0.dll
C:\Program Files\Alta Data Technologies\Alta Software\ADT_L1_API\Win32\bin\ADT_L1.dll

For 64-bit Windows:

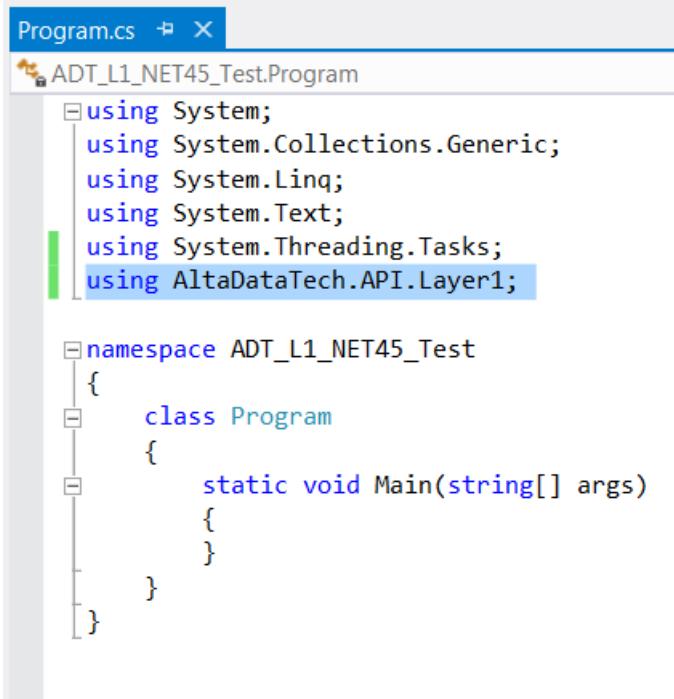
C:\Program Files\Alta Data Technologies\Alta Software\ADT_L0_API\Win64\bin\ADT_L0.dll
C:\Program Files\Alta Data Technologies\Alta Software\ADT_L1_API\Win64\bin\ADT_L1.dll

Note: When you use a release build of your application you will need these DLLs in the \bin\release directory for your project. If you distribute the final application, these DLLs and the ADT_L1_NET40 assembly should be in the same directory as the executable file for the application.

Now our project directory looks something like this:



Now let's modify the shell C# program to use the Alta API. First we will add a "using" statement to tell the program that we are using the Alta Layer 1 API assembly, as shown below:



```
Program.cs  ✘ X
ADT_L1_NET45_Test.Program
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using AltaDataTech.API.Layer1;

namespace ADT_L1_NET45_Test
{
    class Program
    {
        static void Main(string[] args)
        {
        }
    }
}
```

Modify your main program as follows:

```
using System;
using System.Collections.Generic;
using System.Text;
using AltaDataTech.API.Layer1;

namespace ADT_L1_NET45_Test
{
    class Program
    {

        static void Main(string[] args)
        {
            ADT_L1 api = new ADT_L1();          // The ADT_L1 class contains the Layer 1 API functions
            UInt32 status = ADT_L1.SUCCESS;

            // Make sure the DEVICE ID parameter is valid for your device.
            UInt32 devid = ADT_L1.DEVID_PRODUCT_ENETA429P |
                            ADT_L1.DEVID_BOARDDNUM_01 |
                            ADT_L1.DEVID_CHANNELTYPE_A429 |
                            ADT_L1.DEVID_BANK_01;

            Console.WriteLine("ADT Layer 1 .NET 4.5 Test Program");

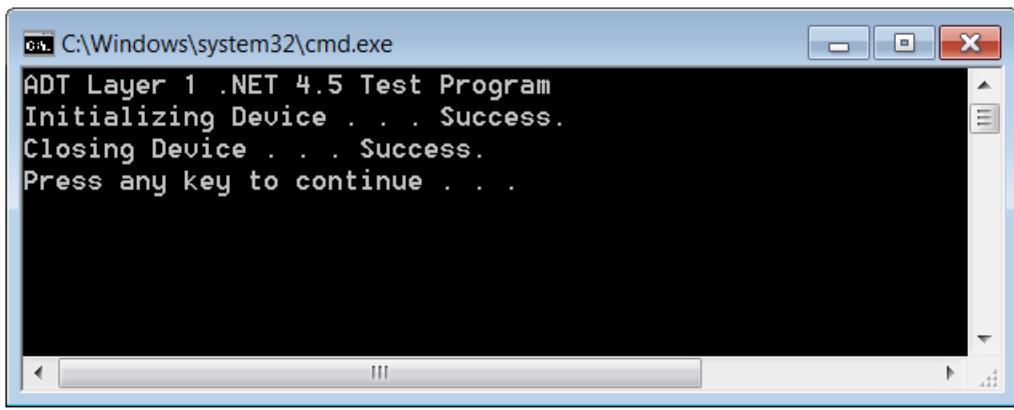
            /* For ENET devices, you must first specify the IP addresses */
            if ((devid & 0xF0000000) == ADT_L1.DEVID_BACKPLANETYPE_ENET)
            {
                // Set Server IP to ENET IP, Client IP to local IP
                // This example uses Server at 192.168.0.128, Client at 192.168.0.2
                Console.Write("Setting IP Addresses . . . ");
                status = api.ENE_SetIpAddr(devid, 0xC0A80080, 0xC0A80002);
                if (status == ADT_L1.SUCCESS) Console.WriteLine("Success");
                else Console.WriteLine("FAILURE! Error = " + status.ToString());
            }

            // Use the API to initialize a device
            Console.Write("Initializing Device . . . ");
            status = api.InitDevice(devid, 0);
            if (status == ADT_L1.SUCCESS)
            {
                Console.WriteLine("Success");

                /* Close the device */
                Console.Write("Closing Device . . . ");
                status = api.CloseDevice(devid);
                if (status == ADT_L1.SUCCESS) Console.WriteLine("Success");
                else Console.WriteLine("FAILURE! Error = " + status.ToString());
            }
            else Console.WriteLine("FAILURE! Error = " + status.ToString());
        }
    }
}
```

This instantiates an ADT_L1 object called “api”. We use this object to call methods that correspond to Layer 1 API functions. In this simple example all we do is initialize a device then close it, but this shows how to access the API from C# using the ADT_L1_NET45 assembly.

As noted in the program comments, you should verify that you use a Device ID that is valid for an Alta board installed in your system. When you run the program you should see something like this:



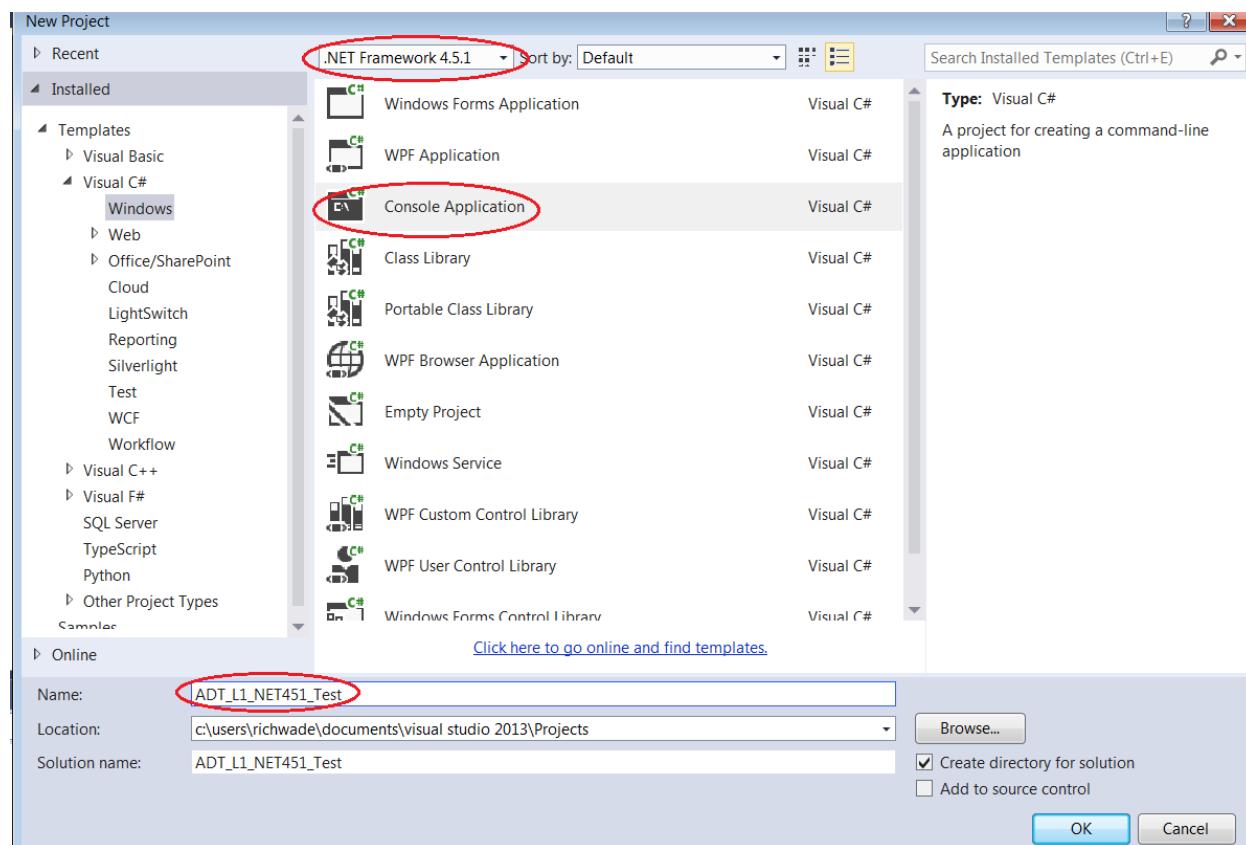
You should now be able to use the provided C# example programs and use the Alta API in your own programs written in C# (or any other .NET language).

Using MSVS 2013 C# with the Layer 1 .NET 4.51 API

Microsoft Visual Studio 2013 can be used to develop .NET 4.51 applications using the ADT_L1_NET451 assembly. We will demonstrate setting up a project to build and run a simple C# .NET 4.51 program using the Alta APIs (with Microsoft Visual Studio 2013 Professional Edition).

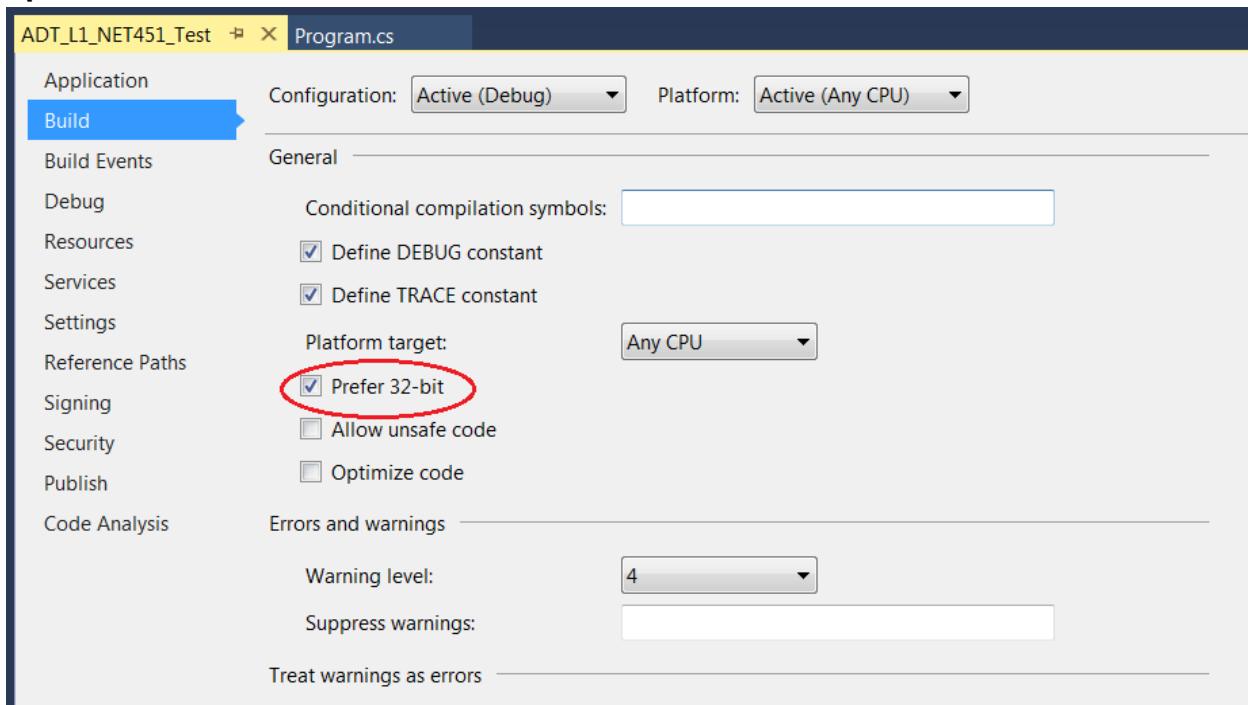
Create a Microsoft Visual C# Windows console application project. Name it “ADT_L1_NET451_Test”.

SELECT A TARGET FRAMEWORK OF .NET FRAMEWORK 4.51.

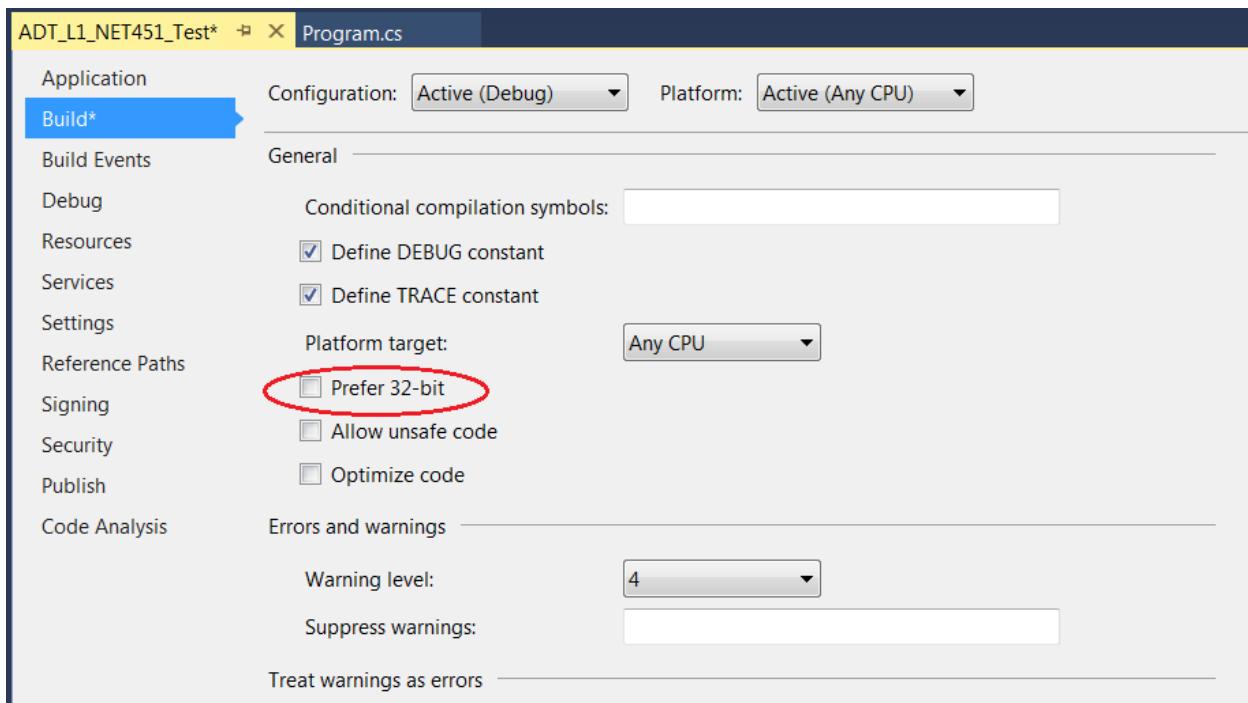


This will create our project with a template C# file called “program.cs”.

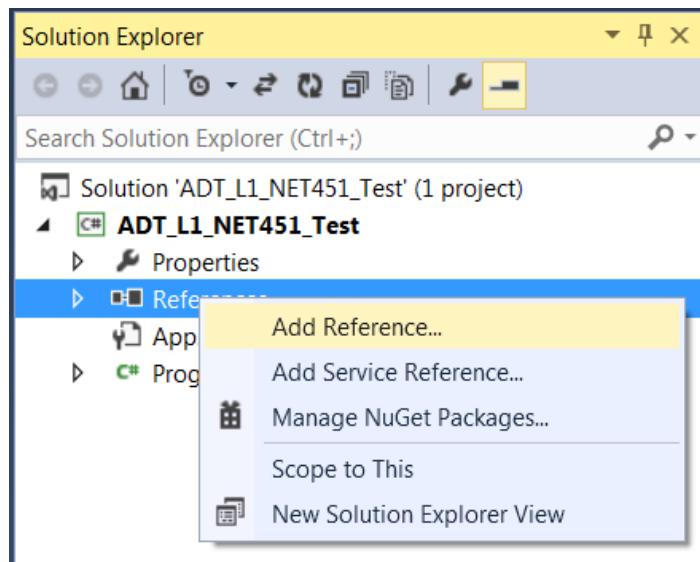
MSVS will default to an x86 32-bit build. Go to the project properties, “Build” options.



If you are using 64-bit Windows you need to use a 64-bit build. Uncheck “Prefer 32-bit”.

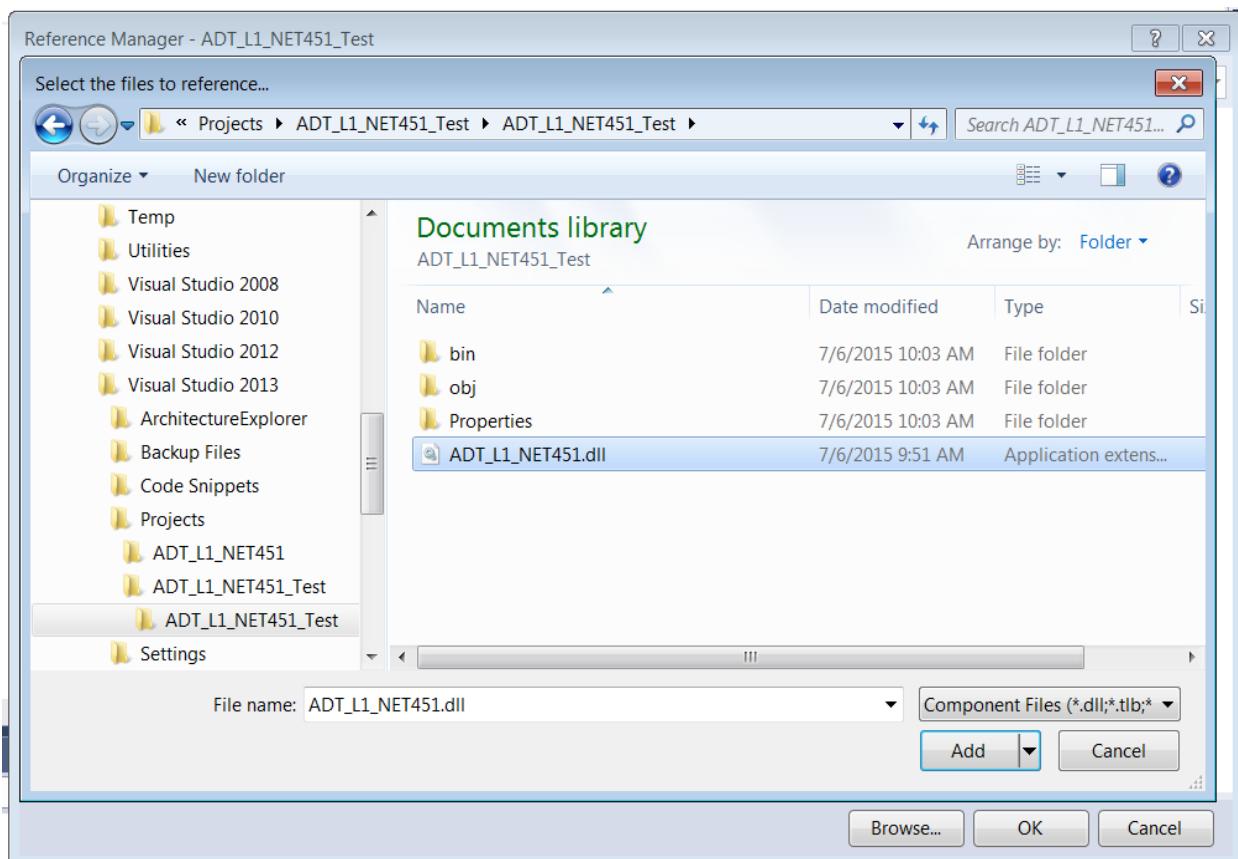


We need to link our project to the ADT_L1_NET451 assembly so we can access the API from our program. First copy the ADT_L1_NET451.dll file to your project directory. In the solution explorer, right click on “References” and select “Add Reference...”.



This brings up the “Add Reference” dialog box. Select the “Browse” button and go to the project directory.

Select the file “ADT_L1_NET451.dll” as shown below. Click the “OK” button.



We also need the DLL files for the Layer 0 and Layer 1 APIs. Copy these files into the \bin\debug directory for the project:

For 32-bit Windows:

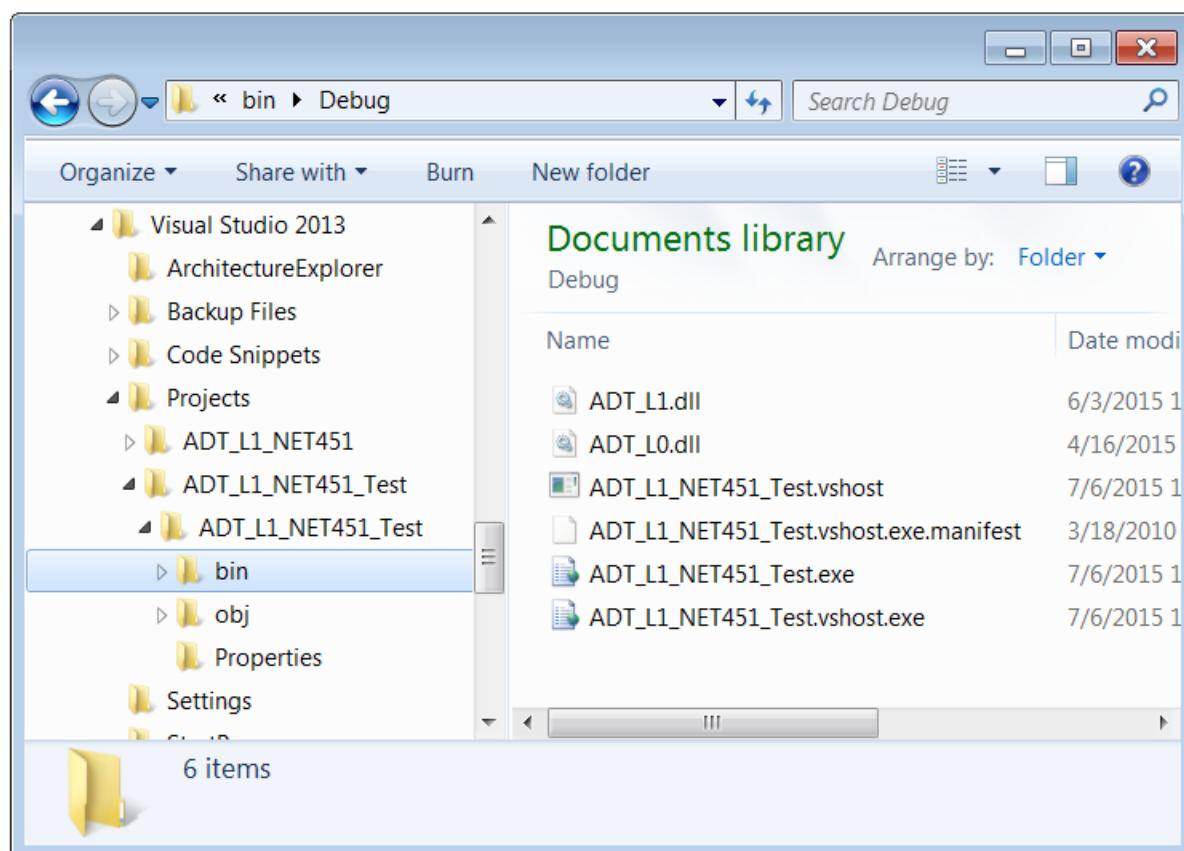
C:\Program Files\Alta Data Technologies\Alta Software\ADT_L0_API\Win32\bin\ADT_L0.dll
C:\Program Files\Alta Data Technologies\Alta Software\ADT_L1_API\Win32\bin\ADT_L1.dll

For 64-bit Windows:

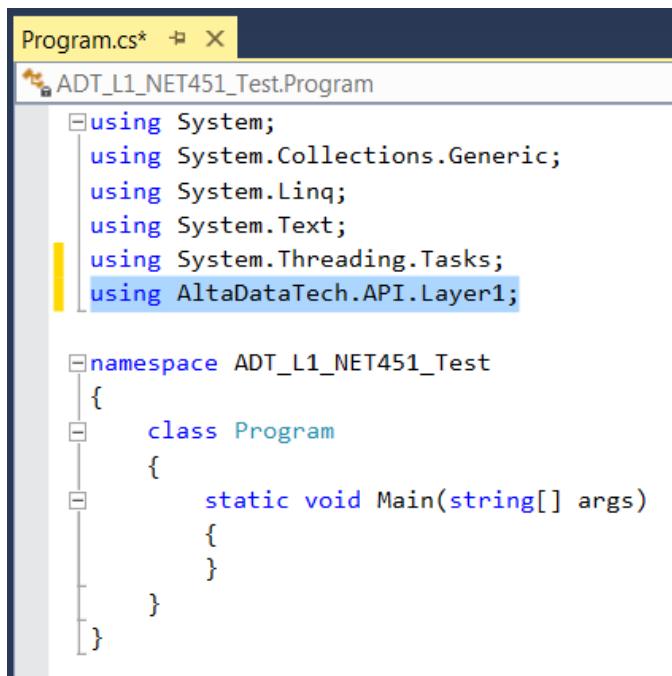
C:\Program Files\Alta Data Technologies\Alta Software\ADT_L0_API\Win64\bin\ADT_L0.dll
C:\Program Files\Alta Data Technologies\Alta Software\ADT_L1_API\Win64\bin\ADT_L1.dll

Note: When you use a release build of your application you will need these DLLs in the \bin\release directory for your project. If you distribute the final application, these DLLs and the ADT_L1_NET40 assembly should be in the same directory as the executable file for the application.

Now our project directory looks something like this:



Now let's modify the shell C# program to use the Alta API. First we will add a "using" statement to tell the program that we are using the Alta Layer 1 API assembly, as shown below:



```
Program.cs* X
ADT_L1_NET451_Test.Program
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using AltaDataTech.API.Layer1;

namespace ADT_L1_NET451_Test
{
    class Program
    {
        static void Main(string[] args)
        {
        }
    }
}
```

Modify your main program as follows:

```
using System;
using System.Collections.Generic;
using System.Text;
using AltaDataTech.API.Layer1;

namespace ADT_L1_NET451_Test
{
    class Program
    {

        static void Main(string[] args)
        {
            ADT_L1 api = new ADT_L1();          // The ADT_L1 class contains the Layer 1 API functions
            UInt32 status = ADT_L1.SUCCESS;

            // Make sure the DEVICE ID parameter is valid for your device.
            UInt32 devid = ADT_L1.DEVID_PRODUCT_ENETA429P |
                            ADT_L1.DEVID_BOARDDNUM_01 |
                            ADT_L1.DEVID_CHANNELTYPE_A429 |
                            ADT_L1.DEVID_BANK_01;

            Console.WriteLine("ADT Layer 1 .NET 4.51 Test Program");

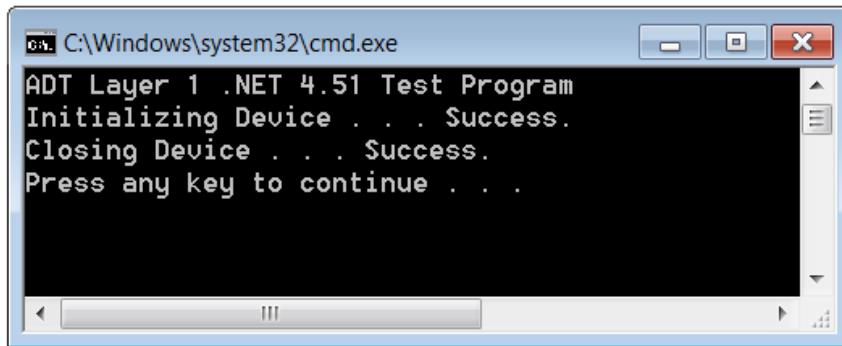
            /* For ENET devices, you must first specify the IP addresses */
            if ((devid & 0xF0000000) == ADT_L1.DEVID_BACKPLANETYPE_ENET)
            {
                // Set Server IP to ENET IP, Client IP to local IP
                // This example uses Server at 192.168.0.128, Client at 192.168.0.2
                Console.Write("Setting IP Addresses . . . ");
                status = api.ENE_SetIpAddr(devid, 0xC0A80080, 0xC0A80002);
                if (status == ADT_L1.SUCCESS) Console.WriteLine("Success");
                else Console.WriteLine("FAILURE! Error = " + status.ToString());
            }

            // Use the API to initialize a device
            Console.Write("Initializing Device . . . ");
            status = api.InitDevice(devid, 0);
            if (status == ADT_L1.SUCCESS)
            {
                Console.WriteLine("Success");

                /* Close the device */
                Console.Write("Closing Device . . . ");
                status = api.CloseDevice(devid);
                if (status == ADT_L1.SUCCESS) Console.WriteLine("Success");
                else Console.WriteLine("FAILURE! Error = " + status.ToString());
            }
            else Console.WriteLine("FAILURE! Error = " + status.ToString());
        }
    }
}
```

This instantiates an ADT_L1 object called “api”. We use this object to call methods that correspond to Layer 1 API functions. In this simple example all we do is initialize a device then close it, but this shows how to access the API from C# using the ADT_L1_NET451 assembly.

As noted in the program comments, you should verify that you use a Device ID that is valid for an Alta board installed in your system. When you run the program you should see something like this:

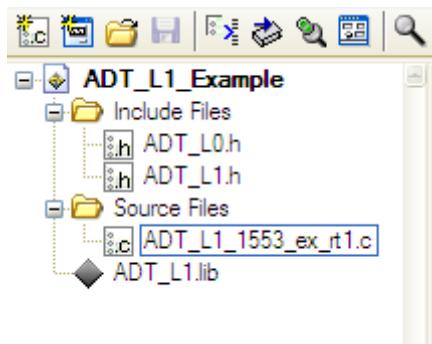


You should now be able to use the provided C# example programs and use the Alta API in your own programs written in C# (or any other .NET language).

Using LabWindows/CVI™ with the Layer 1 API

LabWindows/CVI is the National Instruments development environment for C/C++ programming. CVI can use the standard L0 and L1 Windows DLLs and setting this up is similar to setup for Microsoft Visual Studio.

Your project should start with one of the L1 example programs – we will use ADT_L1_1553_ex_rt1.c here. You will also need the L0 and L1 header files (ADT_L0.h and ADT_L1.h) and the L1 lib file (ADT_L1.lib). Here is what your project should look like:



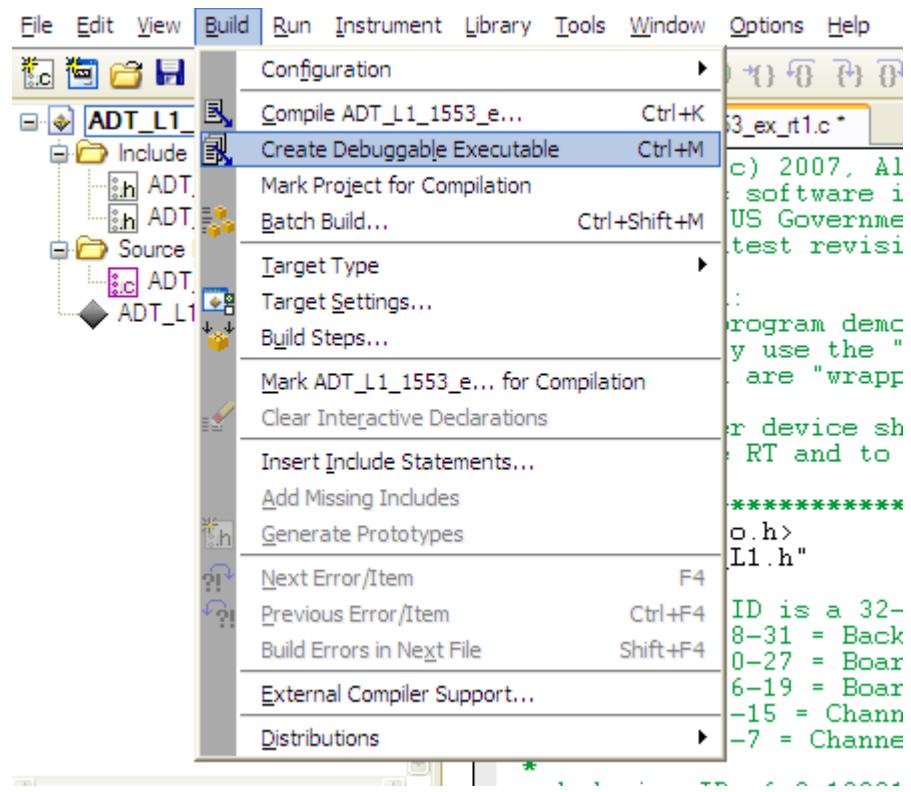
Edit the ADT_L0.h file. You will need to edit the #include lines to include the files needed when compiling in LabWindows/CVI. Here is what it should look like:

```
/* Need the following includes for functions that use memset.
   For Windows (MSVS) and RTX include string.h, memory.h, and malloc.h.
   For INTEGRITY or VxWorks 5.5 include string.h and stdlib.h.
   For NI-VISA LabVIEW Real-Time (built with LabWindows/CVI) include string.h and ansi_c.h
*/
#include <string.h>
/* #include <stdlib.h> */
/* #include <malloc.h> */
/* #include <memory.h> */
#include <ansi_c.h>
```

Edit the example program (ADT_L1_1553_ex_rt1.c). Verify that the #define for DEVID selects the board type that matches the Alta board you are using. The board type constants are defined in ADT_L1.h – see this file for a full list of the possible board type constants. This is what it looks like for a PCI-1553 board:

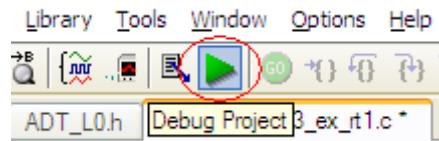
```
/* The DEVICE ID is a 32-bit value that identifies the following:
 *   bits 28-31 = Backplane Type (0 = Simulated, 1 = PCI)
 *   bits 20-27 = Board Type (0 = SIM-1553, 1 = TEST-1553, 2 = PMC-1553, 3 = PC104P-1553, 4 = PCI-1553)
 *   bits 16-19 = Board Number (0 to 15)
 *   bits 8-15 = Channel Type (0x10 = 1553)
 *   bits 0-7 = Channel Number (0 to 255)
 *
 * A device ID of 0x10201000 specifies the first 1553 channel of the
 * first ADT PMC-1553 board found.
 * A device ID of 0x10201001 specifies the second 1553 channel of the
 * first ADT PMC-1553 Board found.
 */
#define DEVID (ADT_PRODUCT_PCI1553 | ADT_DEVID_BOARDNUM_01 | ADT_DEVID_CHANNELTYPE_1553 | ADT_DEVID_CHANNELNUM_01)
```

Now you can build the project. Go to the “Build” menu and select “Create Debuggable Executable”.

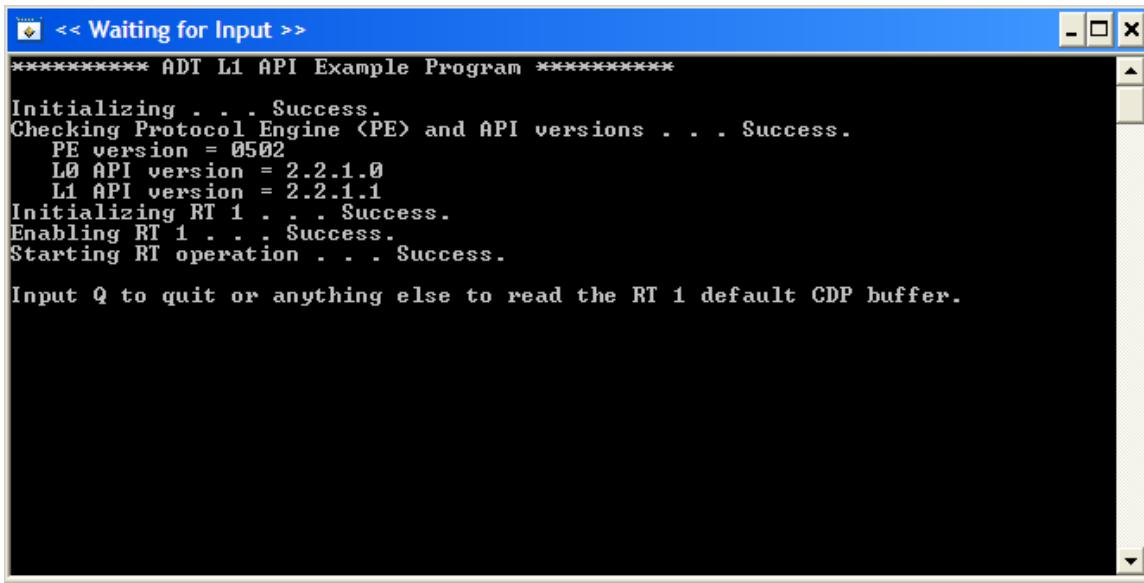


This will create an executable file. **This executable will need to be able to find the AltaAPI DLL files (ADT_L0.dll and ADT_L1.dll)** – you can copy these two DLL files into the same directory as your executable file (normally this will be in the folder “\My Documents\National Instruments\CVI”).

Now you can run the program by clicking on the green arrow icon in the LabWindows/CVI toolbar.



If you are using the ADT_L1_1553_ex_rt1.c example program, you should now see something like this:



```
<< Waiting for Input >>
*****
ADT L1 API Example Program *****

Initializing . . . Success.
Checking Protocol Engine (PE) and API versions . . . Success.
    PE version = 0502
    L0 API version = 2.2.1.0
    L1 API version = 2.2.1.1
Initializing RT 1 . . . Success.
Enabling RT 1 . . . Success.
Starting RT operation . . . Success.

Input Q to quit or anything else to read the RT 1 default CDP buffer.
```

In this example, RT1 is now running and ready for commands from the Bus Controller. You can use AltaView (on another 1553 channel) as BC and BM to send messages to the RT and verify the RT response.

Now you can replace the ADT_L1_1553_ex_rt1.c program in your project with any of the example programs for 1553 BC, RT, BM, etc. If you are using an A429 board you can use the A429 example programs. These example programs can provide a starting point for developing your own programs and applications using the AltaAPI.

Using LabVIEW™ and LabVIEW Real-Time™ with the Layer 1 API

There are several possible methods to use the Alta Layer 1 API with LabVIEW. **The recommended approach is to use the AltaAPI-LV software**, which is a LabVIEW library (lvlib) containing custom controls and a set of over 200 “wrapper” VIs corresponding to the AltaAPI functions. The AltaAPI-LV software uses the NI-VISA library in Layer 0 to communicate with hardware and therefore it can be used on both standard Windows systems and on LabVIEW Real-Time targets. The AltaAPI-LV software is supported for National Instruments LabVIEW™ version 8.6.1 or later.

Note that AltaAPI-LV is separate from the standard Windows version of the Alta API and uses NI-VISA rather than Jungo WinDriver at the low level (Layer 0). Therefore AltaAPI-LV may not support all the Alta products supported by the standard Windows driver. Contact Alta if you have questions on AltaAPI-LV support for a specific product.

Please refer to the AltaAPI-LV Users Manuals for more information.

Using LabVIEW™ with the Layer 1 .NET 2.0 API

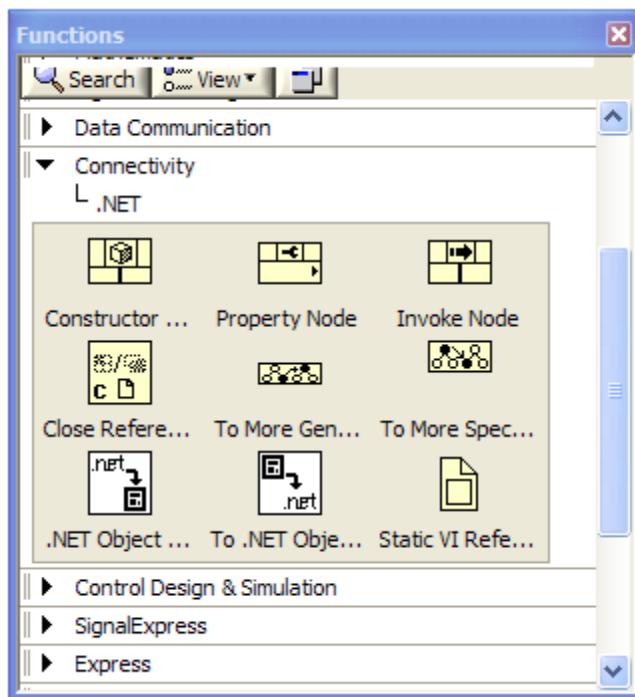
Another approach is to directly call the AltaAPI functions through either the Win32 DLL or the .NET assembly DLL. Note that this approach will not work with LabVIEW Real-Time targets. National Instruments LabVIEW™ version 8.5 or later integrates easily with the .NET framework and can import .NET assemblies to access the associated classes and methods. This makes it easy to import the Alta Layer 1 .NET 2.0 API.

WARNING – not all API functions use parameter types that are directly compatible with LabVIEW. Some API functions may not be usable in the LabVIEW environment because of this. LabVIEW applications that need these functions should use the AltaAPI-LV software rather than directly using the Alta Layer 1 .NET 2.0 API.

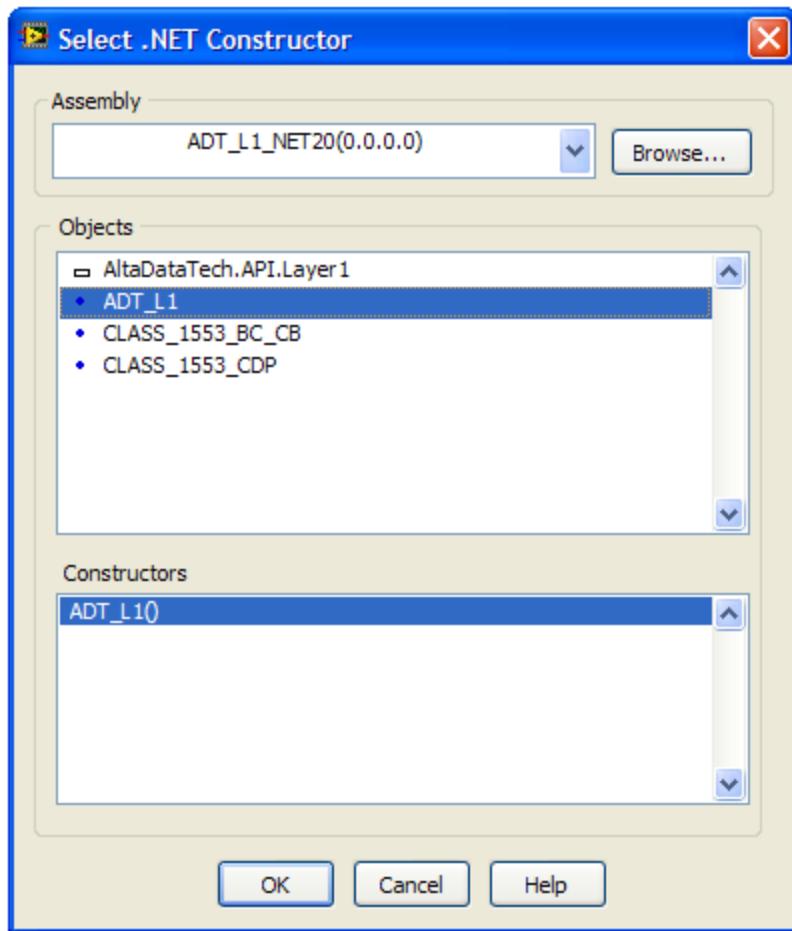
Place a copy of the following files where they can easily be referenced from your LabVIEW™ code:

ADT_L1_NET20.dll
ADT_L1.dll
ADT_L0.dll

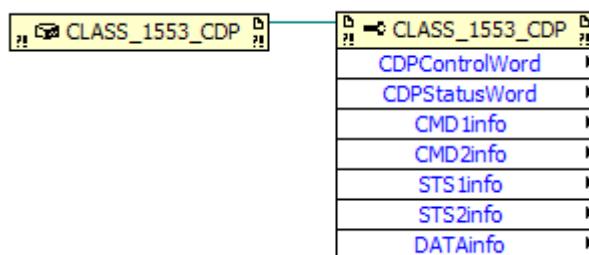
Right-click on your block diagram to bring up the functions window, then go to the .NET palette.



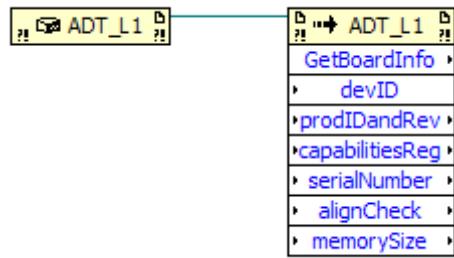
Select the **Constructor Node** and place it on your block diagram. A dialog-box will pop-up for you to select the .NET assembly to use. Browse to find the ADT_L1_NET20.dll assembly. Select the desired class within the assembly and click the OK button.



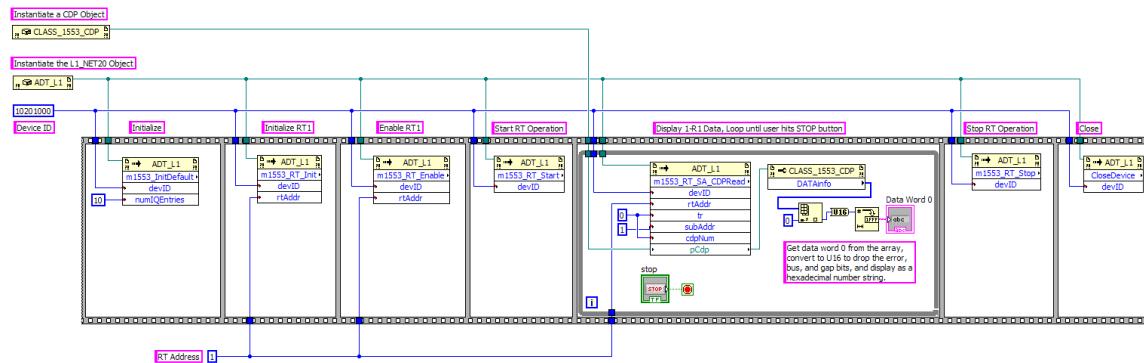
The **Property Node** can be used to access properties and public members of a class.



The **Invoke Node** can be used to call the methods of a class.



You can now build your LabVIEW™ application using the Alta Layer 1 .NET 2.0 classes and methods.



Layer 1 .NET 2.0 API Classes and Methods

The ADT_L1_NET20 assembly contains the following classes:

CLASS_1553_CDP

This class corresponds to the Layer 1 structure ADT_L1_1553_CDP. Instances of this class are passed as parameters for the ADT_L1 methods.

CLASS_1553_BC_CB

This class corresponds to the Layer 1 structure ADT_L1_1553_BC_CB. Instances of this class are passed as parameters for the ADT_L1 methods.

CLASS_A429_RXP

This class corresponds to the Layer 1 structure ADT_L1_A429_RXP. Instances of this class are passed as parameters for the ADT_L1 methods.

CLASS_A429_TXP

This class corresponds to the Layer 1 structure ADT_L1_A429_TXP. Instances of this class are passed as parameters for the ADT_L1 methods. This class can also be used for playback (PXPs).

ADT_L1

This class encapsulates some Layer 0 API functions and most of the Layer 1 API functions. The following table lists the methods in this class and the corresponding Layer 0 and Layer 1 API functions.

ADT_L1 Method	Layer 0 or Layer 1 API Function
MapMemory	ADT_L0_MapMemory
ReadMem32	ADT_L0_ReadMem32
WriteMem32	ADT_L0_WriteMem32
DevicePresent	ADT_L1_DevicePresent
DevicePresent_pcilInfo	ADT_L1_DevicePresent_pcilInfo
InitDevice	ADT_L1_InitDevice
CloseDevice	ADT_L1_CloseDevice
GetVersionInfo	ADT_L1_GetVersionInfo
GetBoardInfo	ADT_L1_GetBoardInfo
GetMemoryAvailable	ADT_L1_GetMemoryAvailable
ReadDeviceMem32	ADT_L1_ReadDeviceMem32
WriteDeviceMem32	ADT_L1_WriteDeviceMem32
ProgramBoardFlash	ADT_L1_ProgramBoardFlash
msSleep	ADT_L1_msSleep
ENET_SetIpAddr	ADT_L1_ENET_SetIpAddr
ENET_GetIpAddr	ADT_L1_ENET_GetIpAddr
ENET_ADCP_Reset	ADT_L1_ENET_ADCP_Reset
ENET_ADCP_GetStatistics	ADT_L1_ENET_ADCP_GetStatistics
ENET_ADCP_ClearStatistics	ADT_L1_ENET_ADCP_ClearStatistics
InitMemMgmt	ADT_L1_InitMemMgmt
CloseMemMgmt	ADT_L1_CloseMemMgmt
MemoryAlloc	ADT_L1_MemoryAlloc
MemoryFree	ADT_L1_MemoryFree
GetMemoryAvailable	ADT_L1_GetMemoryAvailable
Global_TimeClear	ADT_L1_Global_TimeClear
Global_I2C_ReadTemp	ADT_L1_Global_I2C_ReadTemp
Global_I2C_SetIrigDac	ADT_L1_Global_I2C_SetIrigDac
Global_I2C_SetVVdac	ADT_L1_Global_I2C_SetVVdac
Global_ConfigExtClk	ADT_L1_Global_ConfigExtClk
Global_CalibrateIrigDac	ADT_L1_Global_CalibrateIrigDac
Global_CalibrateIrigDacOptions	ADT_L1_Global_CalibrateIrigDacOptions
Global_ReadIrigTime	ADT_L1_Global_ReadIrigTime
BIT_MemoryTest	ADT_L1_BIT_MemoryTest
BIT_InitiatedBIT	ADT_L1_BIT_InitiatedBIT
BIT_PeriodicBIT	ADT_L1_BIT_PeriodicBIT
INT_HandlerAttach	ADT_L1_INT_HandlerAttach
INT_HandlerDetach	ADT_L1_INT_HandlerDetach

m1553_InitDefault	ADT_L1_1553_InitDefault
m1553_InitChannel	ADT_L1_1553_InitChannel
m1553_InitChannelLive	ADT_L1_1553_InitChannelLive
m1553_SetConfig	ADT_L1_1553_SetConfig
m1553_GetConfig	ADT_L1_1553_GetConfig
m1553_GetPEInfo	ADT_L1_1553_GetPEInfo
m1553_TimeSet	ADT_L1_1553_TimeSet
m1553_TimeGet	ADT_L1_1553_TimeGet
m1553_TimeClear	ADT_L1_1553_TimeClear
m1553_IrigLatchedTimeGet	ADT_L1_1553_IrigLatchedTimeGet
m1553_PBTimeSet	ADT_L1_1553_PBTimeSet
m1553_PBTimeGet	ADT_L1_1553_PBTimeGet
m1553_UseExtClk	ADT_L1_1553_UseExtClk
m1553_ForceTrgOut	ADT_L1_1553_ForceTrgOut
m1553_SC_ArmTrigger	ADT_L1_1553_SC_ArmTrigger
m1553_SC_ReadBuffer	ADT_L1_1553_SC_ReadBuffer
m1553_CDP_Calculate_1760_Checksum	ADT_L1_1553_CDP_Calculate_1760_Checksum
m1553_IntervalTimerGet	ADT_L1_1553_IntervalTimerGet
m1553_IntervalTimerSet	ADT_L1_1553_IntervalTimerSet
m1553_INT_EnableInt	ADT_L1_1553_INT_EnableInt
m1553_INT_DisableInt	ADT_L1_1553_INT_DisableInt
m1553_INT_SetIntSeqNum	ADT_L1_1553_INT_SetIntSeqNum
m1553_INT_GetIntSeqNum	ADT_L1_1553_INT_GetIntSeqNum
m1553_INT_CheckChannelIntPending	ADT_L1_1553_INT_CheckChannelIntPending
m1553_INT_IQ_ReadEntry	ADT_L1_1553_INT_IQ_ReadEntry
m1553_INT_IQ_ReadNewEntries	ADT_L1_1553_INT_IQ_ReadNewEntries
m1553_BC_Init	ADT_L1_1553_BC_Init
m1553_BC_Close	ADT_L1_1553_BC_Close
m1553_BC_CB_CDPAllocate	ADT_L1_1553_BC_CB_CDPAllocate
m1553_BC_CB_CDPFree	ADT_L1_1553_BC_CB_CDPFree
m1553_BC_CB_Write	ADT_L1_1553_BC_CB_Write
m1553_BC_CB_Read	ADT_L1_1553_BC_CB_Read
m1553_BC_CB_CDPWrite	ADT_L1_1553_BC_CB_CDPWrite
m1553_BC_CB_CDPWriteWords	ADT_L1_1553_BC_CB_CDPWriteWords
m1553_BC_CB_CDPRead	ADT_L1_1553_BC_CB_CDPRead
m1553_BC_CB_CDPReadWords	ADT_L1_1553_BC_CB_CDPReadWords
m1553_BC_Start	ADT_L1_1553_BC_Start
m1553_BC_Stop	ADT_L1_1553_BC_Stop
m1553_BC_AperiodicSend	ADT_L1_1553_BC_AperiodicSend
m1553_BC_IsRunning	ADT_L1_1553_BC_IsRunning
m1553_BC_AperiodicIsRunning	ADT_L1_1553_BC_AperiodicIsRunning
m1553_BC_GetFrameCount	ADT_L1_1553_BC_GetFrameCount
m1553_BC_InjCmdWordError	ADT_L1_1553_BC_InjCmdWordError
m1553_BC_ReadCmdWordError	ADT_L1_1553_BC_ReadCmdWordError
m1553_BC_CB_SetAddressBranchValues	ADT_L1_1553_BC_CB_SetAddressBranchValues
m1553_BC_CB_GetAddr	ADT_L1_1553_BC_CB_GetAddr
m1553_BC_CB_CDP_GetAddr	ADT_L1_1553_BC_CB_CDP_GetAddr
m1553_RT_Init	ADT_L1_1553_RT_Init
m1553_RT_Close	ADT_L1_1553_RT_Close

m1553_RT_Start	ADT_L1_1553_RT_Start
m1553_RT_Stop	ADT_L1_1553_RT_Stop
m1553_RT_Enable	ADT_L1_1553_RT_Enable
m1553_RT_Disable	ADT_L1_1553_RT_Disable
m1553_RT_Monitor	ADT_L1_1553_RT_Monitor
m1553_RT_SetRespTime	ADT_L1_1553_RT_SetRespTime
m1553_RT_GetRespTime	ADT_L1_1553_RT_GetRespTime
m1553_RT_InjStsWordError	ADT_L1_1553_RT_InjStsWordError
m1553_RT_ReadStsWordError	ADT_L1_1553_RT_ReadStsWordError
m1553_RT_SetOptions	ADT_L1_1553_RT_SetOptions
m1553_RT_GetOptions	ADT_L1_1553_RT_GetOptions
m1553_RT_SetSingleRTAddr	ADT_L1_1553_RT_SetSingleRTAddr
m1553_RT_GetSingleRTAddr	ADT_L1_1553_RT_GetSingleRTAddr
m1553_RT_GetExternalRTAddr	ADT_L1_1553_RT_GetExternalRTAddr
m1553_RT_GetLastCmd	ADT_L1_1553_RT_GetLastCmd
m1553_RT_StatusWrite	ADT_L1_1553_RT_StatusWrite
m1553_RT_StatusRead	ADT_L1_1553_RT_StatusRead
m1553_RT_SA_LegalizationWrite	ADT_L1_1553_RT_SA_LegalizationWrite
m1553_RT_SA_LegalizationRead	ADT_L1_1553_RT_SA_LegalizationRead
m1553_RT_MC_LegalizationWrite	ADT_L1_1553_RT_MC_LegalizationWrite
m1553_RT_MC_LegalizationRead	ADT_L1_1553_RT_MC_LegalizationRead
m1553_RT_SA_CDPAllocate	ADT_L1_1553_RT_SA_CDPAllocate
m1553_RT_SA_CDPFree	ADT_L1_1553_RT_SA_CDPFree
m1553_RT_SA_CDPWrite	ADT_L1_1553_RT_SA_CDPWrite
m1553_RT_SA_CDPWriteWords	ADT_L1_1553_RT_SA_CDPWriteWords
m1553_RT_SA_CDPRead	ADT_L1_1553_RT_SA_CDPRead
m1553_RT_SA_CDPReadWords	ADT_L1_1553_RT_SA_CDPReadWords
m1553_RT_MC_CDPAllocate	ADT_L1_1553_RT_MC_CDPAllocate
m1553_RT_MC_CDPFree	ADT_L1_1553_RT_MC_CDPFree
m1553_RT_MC_CDPWrite	ADT_L1_1553_RT_MC_CDPWrite
m1553_RT_SA_CDPWriteWords	ADT_L1_1553_RT_MC_CDPWriteWords
m1553_RT_MC_CDPRead	ADT_L1_1553_RT_MC_CDPRead
m1553_RT_SA_CDPReadWords	ADT_L1_1553_RT_MC_CDPReadWords
m1553_RT_SA_CDP_GetAddr	ADT_L1_1553_RT_SA_CDP_GetAddr
m1553_RT_MC_CDP_GetAddr	ADT_L1_1553_RT_MC_CDP_GetAddr
m1553_BM_Config	ADT_L1_1553_BM_Config
m1553_BM_Start	ADT_L1_1553_BM_Start
m1553_BM_Stop	ADT_L1_1553_BM_Stop
m1553_BM_Clear	ADT_L1_1553_BM_Clear
m1553_BM_FilterRead	ADT_L1_1553_BM_FilterRead
m1553_BM_FilterWrite	ADT_L1_1553_BM_FilterWrite
m1553_BM_BufferCreate	ADT_L1_1553_BM_BufferCreate
m1553_BM_BufferFree	ADT_L1_1553_BM_BufferFree
m1553_BM_ReadNewMsgs	ADT_L1_1553_BM_ReadNewMsgs
m1553_BM_ReadNewMsgsDMA	ADT_L1_1553_BM_ReadNewMsgsDMA
m1553_BM_CDPWrite	ADT_L1_1553_BM_CDPWrite
m1553_BM_CDPRead	ADT_L1_1553_BM_CDPRead
m1553_PB_SetRtResponse	ADT_L1_1553_PB_SetRtResponse
m1553_PB_GetRtResponse	ADT_L1_1553_PB_GetRtResponse

m1553_PB_Allocate	ADT_L1_1553_PB_Allocate
m1553_PB_Free	ADT_L1_1553_PB_Free
m1553_PB_CDPWrite	ADT_L1_1553_PB_CDPWrite
m1553_PB_Start	ADT_L1_1553_PB_Start
m1553_PB_Stop	ADT_L1_1553_PB_Stop
m1553_PB_IsRunning	ADT_L1_1553_PB_IsRunning
m1553_SG_Configure	ADT_L1_1553_SG_Configure
m1553_SG_Free	ADT_L1_1553_SG_Free
m1553_SG_CreateSGCB	ADT_L1_1553_SG_CreateSGCB
m1553_SG_Start	ADT_L1_1553_SG_Start
m1553_SG_Stop	ADT_L1_1553_SG_Stop
m1553_SG_IsRunning	ADT_L1_1553_SG_IsRunning
m1553_SG_WordToVectors	ADT_L1_1553_SG_WordToVectors
m1553_SG_AddVectors	ADT_L1_1553_SG_AddVectors
a429_InitDefault	ADT_L1_A429_InitDefault
a429_InitDevice	ADT_L1_A429_InitDevice
a429_GetConfig	ADT_L1_A429_GetConfig
a429_GetPEInfo	ADT_L1_A429_GetPEInfo
a429_TimeSet	ADT_L1_A429_TimeSet
a429_TimeGet	ADT_L1_A429_TimeGet
a429_PBTimeSet	ADT_L1_A429_PBTimeSet
a429_PBTimeGet	ADT_L1_A429_PBTimeGet
a429_TimeClear	ADT_L1_A429_TimeClear
a429_SC_ArmTrigger	ADT_L1_A429_SC_ArmTrigger
a429_SC_ReadBuffer	ADT_L1_A429_SC_ReadBuffer
a429_IntervalTimerGet	ADT_L1_A429_IntervalTimerGet
a429_IntervalTimerSet	ADT_L1_A429_IntervalTimerSet
a429_IrigLatchedTimeGet	ADT_L1_A429_IrigLatchedTimeGet
a429_INT_EnableInt	ADT_L1_A429_INT_EnableInt
a429_INT_DisableInt	ADT_L1_A429_INT_DisableInt
a429_INT_CheckDeviceIntPending	ADT_L1_A429_INT_CheckDeviceIntPending
a429_INT_IQ_ReadEntry	ADT_L1_A429_INT_IQ_ReadEntry
a429_INT_SetIntSeqNum	ADT_L1_A429_INT_SetIntSeqNum
a429_INT_GetIntSeqNum	ADT_L1_A429_INT_GetIntSeqNum
a429_RXMC_BufferCreate	ADT_L1_A429_RXMC_BufferCreate
a429_RXMC_BufferFree	ADT_L1_A429_RXMC_BufferFree
a429_RXMC_ReadNewRxPs	ADT_L1_A429_RXMC_ReadNewRxPs
a429_RXMC_ReadNewRxPsDMA	ADT_L1_A429_RXMC_ReadNewRxPsDMA
a429_RXMC_ReadRxP	ADT_L1_A429_RXMC_ReadRxP
a429_RX_Channel_Init	ADT_L1_A429_RX_Channel_Init
a429_RX_Channel_Close	ADT_L1_A429_RX_Channel_Close
a429_RX_Channel_Start	ADT_L1_A429_RX_Channel_Start
a429_RX_Channel_Stop	ADT_L1_A429_RX_Channel_Stop
a429_RX_Channel_ReadNewRxPs	ADT_L1_A429_RX_Channel_ReadNewRxPs
a429_RX_Channel_ReadNewRxPsDMA	ADT_L1_A429_RX_Channel_ReadNewRxPsDMA
a429_RX_Channel_ReadRxP	ADT_L1_A429_RX_Channel_ReadRxP

a429_RX_Channel_CVTReadRxP	ADT_L1_A429_RX_Channel_CVTReadRxP
a429_RX_Channel_WriteRxP	ADT_L1_A429_RX_Channel_WriteRxP
a429_RX_Channel_CVTWriteRxP	ADT_L1_A429_RX_Channel_CVTWriteRxP
a429_RX_Channel_SetConfig	ADT_L1_A429_RX_Channel_SetConfig
a429_RX_Channel_GetConfig	ADT_L1_A429_RX_Channel_GetConfig
a429_RX_Channel_SetMaskCompare	ADT_L1_A429_RX_Channel_SetMaskCompare
a429_RX_Channel_GetMaskCompare	ADT_L1_A429_RX_Channel_GetMaskCompare
a717_RX_Channel_SetConfig	ADT_L1_A717_RX_Channel_SetConfig
a717_RX_Channel_GetConfig	ADT_L1_A717_RX_Channel_GetConfig
a429_TX_Channel_Init	ADT_L1_A429_TX_Channel_Init
a429_TX_Channel_Close	ADT_L1_A429_TX_Channel_Close
a429_TX_Channel_CB_TXPAllocate	ADT_L1_A429_TX_Channel_CB_TXPAllocate
a429_TX_Channel_CB_TXPFree	ADT_L1_A429_TX_Channel_CB_TXPFree
a429_TX_Channel_CB_Write	ADT_L1_A429_TX_Channel_CB_Write
a429_TX_Channel_CB_Read	ADT_L1_A429_TX_Channel_CB_Read
a429_TX_Channel_CB_TXPWrite	ADT_L1_A429_TX_Channel_CB_TXPWrite
a429_TX_Channel_CB_TXPRead	ADT_L1_A429_TX_Channel_CB_TXPRead
a429_TX_Channel_Start	ADT_L1_A429_TX_Channel_Start
a429_TX_Channel_Stop	ADT_L1_A429_TX_Channel_Stop
a429_TX_Channel_IsRunning	ADT_L1_A429_TX_Channel_IsRunning
a429_TX_Channel_SendLabel	ADT_L1_A429_TX_Channel_SendLabel
a429_TX_Channel_SendLabelBlock	ADT_L1_A429_TX_Channel_SendLabelBlock
a429_TX_Channel_GetConfig	ADT_L1_A429_TX_Channel_GetConfig
a429_TX_Channel_SetConfig	ADT_L1_A429_TX_Channel_SetConfig
a429_TX_Channel_AperiodicSend	ADT_L1_A429_TX_Channel_AperiodicSend
a429_TX_Channel_AperiodicIsRunning	ADT_L1_A429_TX_Channel_AperiodicIsRunning
a429_TX_Channel_CB_GetAddr	ADT_L1_A429_TX_Channel_CB_GetAddr
a429_TX_Channel_CB_TXP_GetAddr	ADT_L1_A429_TX_Channel_CB_TXP_GetAddr
a429_SG_Configure	ADT_L1_A429_SG_Configure
a429_SG_Free	ADT_L1_A429_SG_Free
a429_SG_CreateSGCB	ADT_L1_A429_SG_CreateSGCB
a429_SG_Start	ADT_L1_A429_SG_Start
a429_SG_Stop	ADT_L1_A429_SG_Stop
a429_SG_IsRunning	ADT_L1_A429_SG_IsRunning
a429_SG_WordToVectors	ADT_L1_A429_SG_WordToVectors
a429_SG_AddVectors	ADT_L1_A429_SG_AddVectors
a429_TX_Channel_PB_Init	ADT_L1_A429_TX_Channel_PB_Init
a429_TX_Channel_PB_Close	ADT_L1_A429_TX_Channel_PB_Close
a429_TX_Channel_PB_CB_TXPAllocate	ADT_L1_A429_TX_Channel_PB_CB_TXPAllocate
a429_TX_Channel_PB_CB_TXPFree	ADT_L1_A429_TX_Channel_PB_CB_TXPFree
a429_TX_Channel_PB_CB_Write	ADT_L1_A429_TX_Channel_PB_CB_Write
a429_TX_Channel_PB_CB_Read	ADT_L1_A429_TX_Channel_PB_CB_Read
a429_TX_Channel_PB_CB_TXPWrite	ADT_L1_A429_TX_Channel_PB_CB_TXPWrite
a429_TX_Channel_PB_CB_TXPRead	ADT_L1_A429_TX_Channel_PB_CB_TXPRead
a429_TX_Channel_PB_Start	ADT_L1_A429_TX_Channel_PB_Start
a429_TX_Channel_PB_Stop	ADT_L1_A429_TX_Channel_PB_Stop
a429_TX_Channel_PB_IsRunning	ADT_L1_A429_TX_Channel_PB_IsRunning
a429_TX_Channel_PB_SetConfig	ADT_L1_A429_TX_Channel_PB_SetConfig

a429_TX_Channel_PB_GetConfig	ADT_L1_A429_TX_Channel_PB_GetConfig
a429_TX_Channel_PB_RXPWrite	ADT_L1_A429_TX_Channel_PB_RXPWrite

The ADT_L1 class also defines constants corresponding to the Layer 0 and Layer 1 constants. These are listed below:

ADT_L1 Constant	Layer 0 or Layer 1 Constant
DEVID_BACKPLANETYPE_SIMULATED	ADT_DEVID_BACKPLANETYPE_SIMULATED
DEVID_BACKPLANETYPE_PCI	ADT_DEVID_BACKPLANETYPE_PCI
DEVID_BACKPLANETYPE_ENET	ADT_DEVID_BACKPLANETYPE_ENET
DEVID_BOARDTYPE_SIM1553	ADT_DEVID_BOARDTYPE_SIM1553
DEVID_BOARDTYPE_TEST1553	ADT_DEVID_BOARDTYPE_TEST1553
DEVID_BOARDTYPE_PMC1553	ADT_DEVID_BOARDTYPE_PMC1553
DEVID_BOARDTYPE_PC104P1553	ADT_DEVID_BOARDTYPE_PC104P1553
DEVID_BOARDTYPE_PCI1553	ADT_DEVID_BOARDTYPE_PCI1553
DEVID_BOARDTYPE_PCCD1553	ADT_DEVID_BOARDTYPE_PCCD1553
DEVID_BOARDTYPE_PCI104E1553	ADT_DEVID_BOARDTYPE_PCI104E1553
DEVID_BOARDTYPE_XMC1553	ADT_DEVID_BOARDTYPE_XMC1553
DEVID_BOARDTYPE_XMCMW	ADT_DEVID_BOARDTYPE_XMCMW
DEVID_BOARDTYPE_ECD54_1553	ADT_DEVID_BOARDTYPE_ECD54_1553
DEVID_BOARDTYPE_PCIE4L1553	ADT_DEVID_BOARDTYPE_PCIE4L1553
DEVID_BOARDTYPE_PCIE1L1553	ADT_DEVID_BOARDTYPE_PCIE1L1553
DEVID_BOARDTYPE_MPCIE1553	ADT_DEVID_BOARDTYPE_MPCIE1553
DEVID_BOARDTYPE_SIMA429	ADT_DEVID_BOARDTYPE_SIMA429
DEVID_BOARDTYPE_TESTA429	ADT_DEVID_BOARDTYPE_TESTA429
DEVID_BOARDTYPE_PMCA429	ADT_DEVID_BOARDTYPE_PMCA429
DEVID_BOARDTYPE_PMCA429HD	ADT_DEVID_BOARDTYPE_PMCA429HD
DEVID_BOARDTYPE_PC104PA429	ADT_DEVID_BOARDTYPE_PC104PA429
DEVID_BOARDTYPE_PCIA429	ADT_DEVID_BOARDTYPE_PCIA429
DEVID_BOARDTYPE_PCCDA429	ADT_DEVID_BOARDTYPE_PCCDA429
DEVID_BOARDTYPE_PCI104EA429	ADT_DEVID_BOARDTYPE_PCI104EA429
DEVID_BOARDTYPE_XMCA429	ADT_DEVID_BOARDTYPE_XMCA429
DEVID_BOARDTYPE_ECD54_A429	ADT_DEVID_BOARDTYPE_ECD54_A429
DEVID_BOARDTYPE_PCIE4LA429	ADT_DEVID_BOARDTYPE_PCIE4LA429
DEVID_BOARDTYPE_PCIE1LA429	ADT_DEVID_BOARDTYPE_PCIE1LA429
DEVID_BOARDTYPE_MPCIEA429	ADT_DEVID_BOARDTYPE_MPCIEA429
DEVID_BOARDTYPE_PMCMA4	ADT_DEVID_BOARDTYPE_PMCMA4
DEVID_BOARDTYPE_PC104PMA4	ADT_DEVID_BOARDTYPE_PC104PMA4
DEVID_BOARDTYPE_XMCMA4	ADT_DEVID_BOARDTYPE_XMCMA4
DEVID_BOARDTYPE_TBOLTMA4	ADT_DEVID_BOARDTYPE_TBOLTMA4
DEVID_BOARDTYPE_ENET1553	ADT_DEVID_BOARDTYPE_ENET1553
DEVID_BOARDTYPE_PMCE1553	ADT_DEVID_BOARDTYPE_PMCE1553
DEVID_BOARDTYPE_ENETA429	ADT_DEVID_BOARDTYPE_ENETA429
DEVID_BOARDTYPE_ENETA429P	ADT_DEVID_BOARDTYPE_ENETA429P
DEVID_PRODUCT_SIM1553	ADT_PRODUCT_SIM1553
DEVID_PRODUCT_TEST1553	ADT_PRODUCT_TEST1553

DEVID_PRODUCT_PMC1553	ADT_PRODUCT_PMC1553
DEVID_PRODUCT_PC104P1553	ADT_PRODUCT_PC104P1553
DEVID_PRODUCT_PCI1553	ADT_PRODUCT_PCI1553
DEVID_PRODUCT_PCCD1553	ADT_PRODUCT_PCCD1553
DEVID_PRODUCT_PCI104E1553	ADT_PRODUCT_PCI104E1553
DEVID_PRODUCT_XMC1553	ADT_PRODUCT_XMC1553
DEVID_PRODUCT_XMCMW	ADT_PRODUCT_XMCMW
DEVID_PRODUCT_ECD54_1553	ADT_PRODUCT_ECD54_1553
DEVID_PRODUCT_PCIE4L1553	ADT_PRODUCT_PCIE4L1553
DEVID_PRODUCT_PCIE1L1553	ADT_PRODUCT_PCIE1L1553
DEVID_PRODUCT_MPCIE1553	ADT_PRODUCT_MPCIE1553
DEVID_PRODUCT_SIMA429	ADT_PRODUCT_SIMA429
DEVID_PRODUCT_TESTA429	ADT_PRODUCT_TESTA429
DEVID_PRODUCT_PMCA429	ADT_PRODUCT_PMCA429
DEVID_PRODUCT_PMCA429HD	ADT_PRODUCT_PMCA429HD
DEVID_PRODUCT_PC104PA429	ADT_PRODUCT_PC104PA429
DEVID_PRODUCT_PCIA429	ADT_PRODUCT_PCIA429
DEVID_PRODUCT_PCCDA429	ADT_PRODUCT_PCCDA429
DEVID_PRODUCT_PCI104EA429	ADT_PRODUCT_PCI104EA429
DEVID_PRODUCT_XMCA429	ADT_PRODUCT_XMCA429
DEVID_PRODUCT_ECD54_A429	ADT_PRODUCT_ECD54_A429
DEVID_PRODUCT_PCIE4LA429	ADT_PRODUCT_PCIE4LA429
DEVID_PRODUCT_PMCMA4	ADT_PRODUCT_PMCMA4
DEVID_PRODUCT_PC104PMA4	ADT_PRODUCT_PC104PMA4
DEVID_PRODUCT_XMCMA4	ADT_PRODUCT_XMCMA4
DEVID_PRODUCT_TBOLTMA4	ADT_PRODUCT_TBOLTMA4
DEVID_PRODUCT_ENET1553	ADT_PRODUCT_ENET1553
DEVID_PRODUCT_PMCE1553	ADT_PRODUCT_PMCE1553
DEVID_PRODUCT_ENETA429	ADT_PRODUCT_ENETA429
DEVID_PRODUCT_ENETA429P	ADT_PRODUCT_ENETA429P
DEVID_BOARDNUM_01	ADT_DEVID_BOARDNUM_01
DEVID_BOARDNUM_02	ADT_DEVID_BOARDNUM_02
DEVID_BOARDNUM_03	ADT_DEVID_BOARDNUM_03
DEVID_BOARDNUM_04	ADT_DEVID_BOARDNUM_04
DEVID_BOARDNUM_05	ADT_DEVID_BOARDNUM_05
DEVID_BOARDNUM_06	ADT_DEVID_BOARDNUM_06
DEVID_BOARDNUM_07	ADT_DEVID_BOARDNUM_07
DEVID_BOARDNUM_08	ADT_DEVID_BOARDNUM_08
DEVID_BOARDNUM_09	ADT_DEVID_BOARDNUM_09
DEVID_BOARDNUM_10	ADT_DEVID_BOARDNUM_10
DEVID_BOARDNUM_11	ADT_DEVID_BOARDNUM_11
DEVID_BOARDNUM_12	ADT_DEVID_BOARDNUM_12
DEVID_BOARDNUM_13	ADT_DEVID_BOARDNUM_13
DEVID_BOARDNUM_14	ADT_DEVID_BOARDNUM_14
DEVID_BOARDNUM_15	ADT_DEVID_BOARDNUM_15
DEVID_BOARDNUM_16	ADT_DEVID_BOARDNUM_16
DEVID_CHANNELTYPE_GLOBALS	ADT_DEVID_CHANNELTYPE_GLOBALS
DEVID_CHANNELTYPE_1553	ADT_DEVID_CHANNELTYPE_1553
DEVID_CHANNELTYPE_A429	ADT_DEVID_CHANNELTYPE_A429

DEVID_CHANNELNUM_01	ADT_DEVID_CHANNELNUM_01
DEVID_CHANNELNUM_02	ADT_DEVID_CHANNELNUM_02
DEVID_CHANNELNUM_03	ADT_DEVID_CHANNELNUM_03
DEVID_CHANNELNUM_04	ADT_DEVID_CHANNELNUM_04
DEVID_CHANNELNUM_05	ADT_DEVID_CHANNELNUM_05
DEVID_CHANNELNUM_06	ADT_DEVID_CHANNELNUM_06
DEVID_CHANNELNUM_07	ADT_DEVID_CHANNELNUM_07
DEVID_CHANNELNUM_08	ADT_DEVID_CHANNELNUM_08
DEVID_CHANNELNUM_09	ADT_DEVID_CHANNELNUM_09
DEVID_CHANNELNUM_10	ADT_DEVID_CHANNELNUM_10
DEVID_CHANNELNUM_11	ADT_DEVID_CHANNELNUM_11
DEVID_CHANNELNUM_12	ADT_DEVID_CHANNELNUM_12
DEVID_CHANNELNUM_13	ADT_DEVID_CHANNELNUM_13
DEVID_CHANNELNUM_14	ADT_DEVID_CHANNELNUM_14
DEVID_CHANNELNUM_15	ADT_DEVID_CHANNELNUM_15
DEVID_CHANNELNUM_16	ADT_DEVID_CHANNELNUM_16
DEVID_BANK_01	ADT_DEVID_BANK_01
DEVID_BANK_02	ADT_DEVID_BANK_02
DEVID_BANK_03	ADT_DEVID_BANK_03
DEVID_BANK_04	ADT_DEVID_BANK_04
SUCCESS	ADT_SUCCESS
FAILURE	ADT_FAILURE
ERR_MEM_MAP_SIZE	ADT_ERR_MEM_MAP_SIZE
ERR_NO_DEVICE	ADT_ERR_NO_DEVICE
ERR_CANT_OPEN_DEV	ADT_ERR_CANT_OPEN_DEV
ERR_DEV_NOT_INITED	ADT_ERR_DEV_NOT_INITED
ERR_DEV_ALREADY_OPEN	ADT_ERR_DEV_ALREADY_OPEN
ERR_UNSUPPORTED_BACKPLANE	ADT_ERR_UNSUPPORTED_BACKPLANE
ERR_UNSUPPORTED_PRODTYPE	ADT_ERR_UNSUPPORTED_PRODTYPE
ERR_UNSUPPORTED_CHANNELTYPE	ADT_ERR_UNSUPPORTED_CHANNELTYPE
ERR_CANT_OPEN_DRIVER	ADT_ERR_CANT_OPEN_DRIVER
ERR_CANT_SET_DRV_OPTIONS	ADT_ERR_CANT_SET_DRV_OPTIONS
ERR_CANT_GET_DEV_INFO	ADT_ERR_CANT_GET_DEV_INFO
ERR_INVALID_BOARD_NUM	ADT_ERR_INVALID_BOARD_NUM
ERR_INVALID_CHANNEL_NUM	ADT_ERR_INVALID_CHANNEL_NUM
ERR_DRIVER_READ_FAIL	ADT_ERR_DRIVER_READ_FAIL
ERR_DRIVER_WRITE_FAIL	ADT_ERR_DRIVER_WRITE_FAIL
ERR_DEVICE_CLOSE_FAIL	ADT_ERR_DEVICE_CLOSE_FAIL
ERR_DRIVER_CLOSE_FAIL	ADT_ERR_DRIVER_CLOSE_FAIL
ERR_KP_OPEN_FAIL	ADT_ERR_KP_OPEN_FAIL
ERR_ENET_NO_PORT_AVAILABLE	ADT_ERR_ENET_NO_PORT_AVAILABLE
ERR_ENET_READ_FAIL	ADT_ERR_ENET_READ_FAIL
ERR_ENET_WRITE_FAIL	ADT_ERR_ENET_WRITE_FAIL
ERR_ENET_NOTRUNNING	ADT_ERR_ENET_NOTRUNNING
ERR_ENET_INVALID_SIZE	ADT_ERR_ENET_INVALID_SIZE
ERR_ENET_SENDFAIL	ADT_ERR_ENET_SENDFAIL
ERR_ENET_SELECTFAIL	ADT_ERR_ENET_SELECTFAIL
ERR_ENET_SELECTTIMEOUT	ADT_ERR_ENET_SELECTTIMEOUT
ERR_ENET_BADSEQNUM	ADT_ERR_ENET_BADSEQNUM
ERR_ENET_SRVSTSFAIL	ADT_ERR_ENET_SRVSTSFAIL

ERR_ENET_BADPRODUCTID	ADT_ERR_ENET_BADPRODUCTID
ERR_BAD_INPUT	ADT_ERR_BAD_INPUT
ERR_MEM_TEST_FAIL	ADT_ERR_MEM_TEST_FAIL
ERR_MEM_MGT_NO_INIT	ADT_ERR_MEM_MGT_NO_INIT
ERR_MEM_MGT_INIT	ADT_ERR_MEM_MGT_INIT
ERR_MEM_MGT_NO_MEM	ADT_ERR_MEM_MGT_NO_MEM
ERR_BAD_DEV_TYPE	ADT_ERR_BAD_DEV_TYPE
ERR_RT_FT_UNDEF	ADT_ERR_RT_FT_UNDEF
ERR_RT_SA_UNDEF	ADT_ERR_RT_SA_UNDEF
ERR_RT_SA_CDP_UNDEF	ADT_ERR_RT_SA_CDP_UNDEF
ERR_IQ_NO_NEW_ENTRY	ADT_ERR_IQ_NO_NEW_ENTRY
ERR_NO_BCCB_TABLE	ADT_ERR_NO_BCCB_TABLE
ERR_BCCB_ALREADY_ALLOCATED	ADT_ERR_BCCB_ALREADY_ALLOCATED
ERR_BCCB_NOT_ALLOCATED	ADT_ERR_BCCB_NOT_ALLOCATED
ERR_PB_BUFFER_FULL	ADT_ERR_BUFFER_FULL
ERR_TIMEOUT	ADT_ERR_TIMEOUT
ERR_BAD_CHAN_NUM	ADT_ERR_BAD_CHAN_NUM
ERR_BITFAIL	ADT_ERR_BITFAIL
ERR_DEVICEINUSE	ADT_ERR_DEVICEINUSE
ERR_NO_TXCB_TABLE	ADT_ERR_NO_TXCB_TABLE
ERR_TXCB_ALREADY_ALLOCATED	ADT_ERR_TXCB_ALREADY_ALLOCATED
ERR_TXCB_NOT_ALLOCATED	ADT_ERR_TXCB_NOT_ALLOCATED
ERR_PBCB_TOOMANYPXPS	ADT_ERR_PBCB_TOOMANYPXPS
ERR_NORXCHCVT_ALLOCATED	ADT_ERR_NORXCHCVT_ALLOCATED
ERR_NO_DATA_AVAILABLE	ADT_ERR_NO_DATA_AVAILABLE
DEVICEINIT_FORCEINIT	ADT_L1_API_DEVICEINIT_FORCEINIT
DEVICEINIT_NOMEMTEST	ADT_L1_API_DEVICEINIT_NOMEMTEST
DEVICEINIT_NOKP	ADT_L1_API_DEVICEINIT_NOKP
DEVICEINIT_ROOTPERESET	ADT_L1_API_DEVICEINIT_ROOTPERESET
M1553_BC_CB_CSR_HALTONERROR <i>BC_CB_CSR_HALTONERROR</i>	ADT_L1_1553_BC_CB_CSR_HALTONERROR
M1553_BC_CB_CSR_BUSA <i>BC_CB_CSR_BUSA</i>	ADT_L1_1553_BC_CB_CSR_BUSA
M1553_BC_CB_CSR_BUSB <i>BC_CB_CSR_BUSB</i>	ADT_L1_1553_BC_CB_CSR_BUSB
M1553_BC_CB_CSR_STARTFRAME <i>BC_CB_CSR_STARTFRAME</i>	ADT_L1_1553_BC_CB_CSR_STARTFRAME
M1553_BC_CB_CSR_ENDFRAME <i>BC_CB_CSR_ENDFRAME</i>	ADT_L1_1553_BC_CB_CSR_ENDFRAME
M1553_BC_CB_CSR_SCHEDTIMING <i>BC_CB_CSR_SCHEDTIMING</i>	ADT_L1_1553_BC_CB_CSR_SCHEDTIMING
M1553_BC_CB_CSR_WAITFORTRG <i>BC_CB_CSR_WAITFORTRG</i>	ADT_L1_1553_BC_CB_CSR_WAITFORTRG
M1553_BC_CB_CSR_GENEXTTRG <i>BC_CB_CSR_GENEXTTRG</i>	ADT_L1_1553_BC_CB_CSR_GENEXTTRG
M1553_BC_CB_CSR_INTMSGCOMP <i>BC_CB_CSR_INTMSGCOMP</i>	ADT_L1_1553_BC_CB_CSR_INTMSGCOMP
M1553_BC_CB_CSR_ADDRBRANCH	ADT_L1_1553_BC_CB_CSR_ADDRBRANCH

<i>BC_CB_CSR_ADDRBRANCH</i>	
M1553_BC_CB_CSR_WAITFORTRG <i>BC_CB_CSR_WAITFORTRG</i>	ADT_L1_1553_BC_CB_CSR_WAITFORTRG
M1553_BC_CB_CSR_CDPBRANCHONLY <i>BC_CB_CSR_CDPBRANCHONLY</i>	ADT_L1_1553_BC_CB_CSR_CDPBRANCHONLY
M1553_BC_CB_CSR_DELAYONLY <i>BC_CB_CSR_DELAYONLY</i>	ADT_L1_1553_BC_CB_CSR_DELAYONLY
M1553_BC_CB_CSR_BRNCHONVALUE <i>BC_CB_CSR_BRNCHONVALUE</i>	ADT_L1_1553_BC_CB_CSR_BRNCHONVALUE
M1553_BC_CB_CSR_BRNCHRETURN <i>BC_CB_CSR_BRNCHRETURN</i>	ADT_L1_1553_BC_CB_CSR_BRNCHRETURN
M1553_BC_CB_CSR_TYPE_NOP <i>BC_CB_CSR_TYPE_NOP</i>	ADT_L1_1553_BC_CB_CSR_TYPE_NOP
M1553_BC_CB_CSR_TYPE_BCRT <i>BC_CB_CSR_TYPE_BCRT</i>	ADT_L1_1553_BC_CB_CSR_TYPE_BCRT
M1553_BC_CB_CSR_TYPE_RTBC <i>BC_CB_CSR_TYPE_RTBC</i>	ADT_L1_1553_BC_CB_CSR_TYPE_RTBC
M1553_BC_CB_CSR_TYPE_RTRT <i>BC_CB_CSR_TYPE_RTRT</i>	ADT_L1_1553_BC_CB_CSR_TYPE_RTRT
M1553_BC_CB_CSR_TYPE_MCDATA <i>BC_CB_CSR_TYPE_MCDATA</i>	ADT_L1_1553_BC_CB_CSR_TYPE_MCDATA
M1553_BC_CB_CSR_TYPE_MCNO DATA <i>BC_CB_CSR_TYPE_MCNO DATA</i>	ADT_L1_1553_BC_CB_CSR_TYPE_MCNO DATA
M1553_BC_NO_NEXT_MSG <i>BC_NO_NEXT_MSG</i>	ADT_L1_1553_BC_NO_NEXT_MSG
M1553_BC_CSR_RUN	ADT_L1_1553_BC_CSR_RUN
M1553_BC_CSR_STOPPED	ADT_L1_1553_BC_CSR_STOPPED
M1553_BC_CSR_FRMOFLOW	ADT_L1_1553_BC_CSR_FRMOFLOW
M1553_BC_CSR_STOPONOFLOW	ADT_L1_1553_BC_CSR_STOPONOFLOW
M1553_BC_CSR_EN_SUBFRAMES	ADT_L1_1553_BC_CSR_EN_SUBFRAMES
M1553_BC_CSR_INTONFRMOFLOW	ADT_L1_1553_BC_CSR_INTONFRMOFLOW
M1553_BC_CSR_INTONSTOP	ADT_L1_1553_BC_CSR_INTONSTOP
M1553_BC_CSR_INTONRETRYCMPLT	ADT_L1_1553_BC_CSR_INTONRETRYCMPLT
M1553_CDP_CONTROL_TXERRINJ	ADT_L1_1553_CDP_CONTROL_TXERRINJ
M1553_CDP_CONTROL_INTERR	ADT_L1_1553_CDP_CONTROL_INTERR
M1553_CDP_CONTROL_INTNOERR	ADT_L1_1553_CDP_CONTROL_INTNOERR
M1553_CDP_CONTROL_INTCMPTRUE	ADT_L1_1553_CDP_CONTROL_INTCMPTRUE
M1553_CDP_CONTROL_LEDONCMPLT	ADT_L1_1553_CDP_CONTROL_LEDONCMPLT
M1553_CDP_CONTROL_TRGOUTERR	ADT_L1_1553_CDP_CONTROL_TRGOUTERR
M1553_CDP_CONTROL_TRGOUTNOERR	ADT_L1_1553_CDP_CONTROL_TRGOUTNOERR
M1553_CDP_CONTROL_TRGOUTCMPTR	ADT_L1_1553_CDP_CONTROL_TRGOUTCMPTR
M1553_CDP_CONTROL_CDPCMP	ADT_L1_1553_CDP_CONTROL_CDPCMP
M1553_CDP_CONTROL_FRCDCWCNUM	ADT_L1_1553_CDP_CONTROL_FRCDCWCNUM
M1553_CDP_CONTROL_FRCDCWCON	ADT_L1_1553_CDP_CONTROL_FRCDCWCON
M1553_CDP_STATUS_MSGWC	ADT_L1_1553_CDP_STATUS_MSGWC
M1553_CDP_STATUS_BUSAB	ADT_L1_1553_CDP_STATUS_BUSAB
M1553_CDP_STATUS_TWOBUSERR	ADT_L1_1553_CDP_STATUS_TWOBUSERR

M1553_CDP_STATUS_STSWRNGADD	ADT_L1_1553_CDP_STATUS_STSWRNGADD
M1553_CDP_STATUS_NORESP	ADT_L1_1553_CDP_STATUS_NORESP
M1553_CDP_STATUS_WCERR	ADT_L1_1553_CDP_STATUS_WCERR
M1553_CDP_STATUS_PARERR	ADT_L1_1553_CDP_STATUS_PARERR
M1553_CDP_STATUS_BITERR	ADT_L1_1553_CDP_STATUS_BITERR
M1553_CDP_STATUS_SYNCERR	ADT_L1_1553_CDP_STATUS_SYNCERR
M1553_CDP_STATUS_NOERR	ADT_L1_1553_CDP_STATUS_NOERR
M1553_CDP_STATUS_CMPTTRUE	ADT_L1_1553_CDP_STATUS_CMPTTRUE
M1553_CDP_STATUS_SPURMSG	ADT_L1_1553_CDP_STATUS_SPURMSG
M1553_CDP_STATUS_BCRTMSG	ADT_L1_1553_CDP_STATUS_BCRTMSG
M1553_CDP_STATUS_RTBCMSG	ADT_L1_1553_CDP_STATUS_RTBCMSG
M1553_CDP_STATUS_RTRTMSG	ADT_L1_1553_CDP_STATUS_RTRTMSG
M1553_CDP_STATUS_MCMMSG	ADT_L1_1553_CDP_STATUS_MCMMSG
M1553_CDP_STATUS_BRDCSTMSG	ADT_L1_1553_CDP_STATUS_BRDCSTMSG
GLOBAL_CSR	ADT_L1_GLOBAL_CSR
GLOBAL_CSR_CLRTT	ADT_L1_GLOBAL_CSR_CLRTT
GLOBAL_CSR_SETTT	ADT_L1_GLOBAL_CSR_SETTT
GLOBAL_CSR_IRIG_LATCH	ADT_L1_GLOBAL_CSR_IRIG_LATCH
GLOBAL_CSR_IRIG_DETECT	ADT_L1_GLOBAL_CSR_IRIG_DETECT
GLOBAL_CSR_IRIG_LOCK	ADT_L1_GLOBAL_CSR_IRIG_LOCK
GLOBAL_CSR_FRCTRG	ADT_L1_GLOBAL_CSR_FRCTRG
GLOBAL_CSR_EC_SRCIN_485	ADT_L1_GLOBAL_CSR_EC_SRCIN_485
GLOBAL_CSR_EC_SRCIN_TTL	ADT_L1_GLOBAL_CSR_EC_SRCIN_TTL
GLOBAL_CSR_EC_SRCOUT_485	ADT_L1_GLOBAL_CSR_EC_SRCOUT_485
GLOBAL_CSR_EC_SRCOUT_TTL	ADT_L1_GLOBAL_CSR_EC_SRCOUT_TTL
GLOBAL_CSR_EC_FRQOUT_1MHZ	ADT_L1_GLOBAL_CSR_EC_FRQOUT_1MHZ
GLOBAL_CSR_EC_FRQOUT_5MHZ	ADT_L1_GLOBAL_CSR_EC_FRQOUT_5MHZ
GLOBAL_CSR_EC_FRQOUT_10MHZ	ADT_L1_GLOBAL_CSR_EC_FRQOUT_10MHZ
GLOBAL_CSR_LED1ON	ADT_L1_GLOBAL_CSR_LED1ON
GLOBAL_CSR_LED2ON	ADT_L1_GLOBAL_CSR_LED2ON
GLOBAL_INTPENDING	ADT_L1_GLOBAL_INTPENDING
GLOBAL_TRIGGERCSR	ADT_L1_GLOBAL_TRIGGERCSR
GLOBAL_SGLEDISCSTS	ADT_L1_GLOBAL_SGLEDISCSTS
GLOBAL_SGLEDISCCCTL	ADT_L1_GLOBAL_SGLEDISCCCTL
GLOBAL_DIFFDISCSTS	ADT_L1_GLOBAL_DIFFDISCSTS
GLOBAL_DIFFDISCCCTL	ADT_L1_GLOBAL_DIFFDISCCCTL
GLOBAL_I2CCTL	ADT_L1_GLOBAL_I2CCTL
GLOBAL_I2CSTS	ADT_L1_GLOBAL_I2CSTS
GLOBAL_I2C_TS_FPGA_ADDR	ADT_L1_GLOBAL_I2C_TS_FPGA_ADDR
GLOBAL_I2C_TS_XCVR_ADDR	ADT_L1_GLOBAL_I2C_TS_XCVR_ADDR
M1553_PE_ROOT_CSR PE_ROOT_CSR	ADT_L1_1553_PE_ROOT_CSR
M1553_PECSR_HWINTON PECSR_HWINTON	ADT_L1_1553_PECSR_HWINTON
M1553_PECSR_BCAST PECSR_BCAST	ADT_L1_1553_PECSR_BCAST
M1553_PECSR_INTBUS PECSR_INTBUS	ADT_L1_1553_PECSR_INTBUS
M1553_PECSR_MRT PECSR_MRT	ADT_L1_1553_PECSR_MRT
M1553_PECSR_SGON PECSR_SGON	ADT_L1_1553_PECSR_SGON
M1553_PECSR_CLKTRGON	ADT_L1_1553_PECSR_CLKTRGON

<i>PECSR_CLKTRGON</i>	
M1553_PECsr_FRCTRGIN <i>PECSR_FRCTRGIN</i>	ADT_L1_1553_PECsr_FRCTRGIN
M1553_PECsr_FRCTRGOUT <i>PECSR_FRCTRGOUT</i>	ADT_L1_1553_PECsr_FRCTRGOUT
M1553_PECsr_RDIRIGTM <i>PECSR_RDIRIGTM</i>	ADT_L1_1553_PECsr_RDIRIGTM
M1553_PECsr_ZEROTT <i>PECSR_ZEROTT</i>	ADT_L1_1553_PECsr_ZEROTT
M1553_PECsr_SETTT <i>PECSR_SETTT</i>	ADT_L1_1553_PECsr_SETTT
M1553_PECsr_READTT <i>PECSR_READTT</i>	ADT_L1_1553_PECsr_READTT
M1553_PECsr_USE_EC <i>PECSR_USE_EC</i>	ADT_L1_1553_PECsr_USE_EC
M1553_PECsr_FRQIN_1MHZ <i>PECSR_FRQIN_1MHZ</i>	ADT_L1_1553_PECsr_EC_FRQIN_1MHZ
M1553_PECsr_FRQIN_5MHZ <i>PECSR_FRQIN_5MHZ</i>	ADT_L1_1553_PECsr_EC_FRQIN_5MHZ
M1553_PECsr_FRQIN_10MHZ <i>PECSR_FRQIN_10MHZ</i>	ADT_L1_1553_PECsr_EC_FRQIN_10MHZ
M1553_PECsr_RS485_TRIG_IN <i>PECSR_RS485_TRIG_IN</i>	ADT_L1_1553_PECsr_RS485_TRIG_IN
M1553_PECsr_RS485_TRIG_OUT <i>PECSR_RS485_TRIG_OUT</i>	ADT_L1_1553_PECsr_RS485_TRIG_OUT
M1553_PECsr_BUS_B_ONLY <i>PECSR_BUS_B_ONLY</i>	ADT_L1_1553_PECsr_BUS_B_ONLY
M1553_PECsr_BUS_A_ONLY <i>PECSR_BUS_A_ONLY</i>	ADT_L1_1553_PECsr_BUS_A_ONLY
M1553_PECsr_BITFAIL_INT <i>PECSR_BITFAIL_INT</i>	ADT_L1_1553_PECsr_BITFAIL_INT
M1553_PECsr_RUNIBIT <i>PECSR_RUNIBIT</i>	ADT_L1_1553_PECsr_RUNIBIT
M1553_PECsr_CHAN_RESET <i>PECSR_CHAN_RESET</i>	ADT_L1_1553_PECsr_CHAN_RESET
M1553_IQP_TYPESEQ_SEQNUM	ADT_L1_1553_IQP_TYPESEQ_SEQNUM
M1553_IQP_TYPESEQ_SG	ADT_L1_1553_IQP_TYPESEQ_SG
M1553_IQP_TYPESEQ_PB	ADT_L1_1553_IQP_TYPESEQ_PB
M1553_IQP_TYPESEQ_BIT	ADT_L1_1553_IQP_TYPESEQ_BIT
M1553_IQP_TYPESEQ_INTVLTMR	ADT_L1_1553_IQP_TYPESEQ_INTVLTMR
M1553_IQP_TYPESEQ_BCCB	ADT_L1_1553_IQP_TYPESEQ_BCCB
M1553_IQP_TYPESEQ_BCFRMOVR	ADT_L1_1553_IQP_TYPESEQ_BCFRMOVR
M1553_IQP_TYPESEQ_BCSTOP	ADT_L1_1553_IQP_TYPESEQ_BCSTOP
M1553_IQP_TYPESEQ_BCRETRY	ADT_L1_1553_IQP_TYPESEQ_BCRETRY
M1553_IQP_TYPESEQ_BCCDP	ADT_L1_1553_IQP_TYPESEQ_BCCDP
M1553_IQP_TYPESEQ_RTCDP	ADT_L1_1553_IQP_TYPESEQ_RTCDP
M1553_IQP_TYPESEQ_BMCDP	ADT_L1_1553_IQP_TYPESEQ_BMCDP
M1553_BIT_ENCDECFAIL	ADT_L1_1553_BIT_ENCDECFAIL
M1553_BIT_MEMTESTFAIL	ADT_L1_1553_BIT_MEMTESTFAIL
M1553_BIT_PROCFAIL	ADT_L1_1553_BIT_PROCFAIL
M1553_BIT_TTAGFAIL	ADT_L1_1553_BIT_TTAGFAIL
M1553_BIT_TIMEOUTFAIL	ADT_L1_1553_BIT_TIMEOUTFAIL
M1553_BIT_LOOPFAIL	ADT_L1_1553_BIT_LOOPFAIL
M1553_BIT_POSTFAIL	ADT_L1_1553_BIT_POSTFAIL
M1553_BIT_PBITFAIL	ADT_L1_1553_BIT_PBITFAIL

M1553_BIT_IBITFAIL	ADT_L1_1553_BIT_IBITFAIL
M1553_BIT_POSTINPROG	ADT_L1_1553_BIT_POSTINPROG
M1553_BIT_PBITINPROG	ADT_L1_1553_BIT_PBITINPROG
M1553_BIT_IBITINPROG	ADT_L1_1553_BIT_IBITINPROG
A429_TXCB_NO_NEXT_TXCB	ADT_L1_A429_TXCB_NO_NEXT_TXCB
A429_BIT_MEMTESTFAIL	ADT_L1_A429_BIT_MEMTESTFAIL
A429_BIT_PROCFAIL	ADT_L1_A429_BIT_PROCFAIL
A429_BIT_TTAGFAIL	ADT_L1_A429_BIT_TTAGFAIL
A429_BIT_POSTFAIL	ADT_L1_A429_BIT_POSTFAIL
A429_BIT_PBITFAIL	ADT_L1_A429_BIT_PBITFAIL
A429_BIT_IBITFAIL	ADT_L1_A429_BIT_IBITFAIL
A429_BIT_POSTINPROG	ADT_L1_A429_BIT_POSTINPROG
A429_BIT_PBITINPROG	ADT_L1_A429_BIT_PBITINPROG
A429_BIT_IBITINPROG	ADT_L1_A429_BIT_IBITINPROG
A429_IQP_TYPESEQ_SEQNUM	ADT_L1_A429_IQP_TYPESEQ_SEQNUM
A429_IQP_TYPESEQ_CHNUM	ADT_L1_A429_IQP_TYPESEQ_CHNUM
A429_IQP_TYPESEQ_SG	ADT_L1_A429_IQP_TYPESEQ_SG
A429_IQP_TYPESEQ_CVTRXP	ADT_L1_A429_IQP_TYPESEQ_CVTRXP
A429_IQP_TYPESEQ_BIT	ADT_L1_A429_IQP_TYPESEQ_BIT
A429_IQP_TYPESEQ_TXPBCB	ADT_L1_A429_IQP_TYPESEQ_TXPBCB
A429_IQP_TYPESEQ_TXPBSTOP	ADT_L1_A429_IQP_TYPESEQ_TXPBSTOP
A429_IQP_TYPESEQ_TXPBXP	ADT_L1_A429_IQP_TYPESEQ_TXPBXP
A429_IQP_TYPESEQ_MCRXP	ADT_L1_A429_IQP_TYPESEQ_MCRXP
A429_IQP_TYPESEQ_CHRXP	ADT_L1_A429_IQP_TYPESEQ_CHRXP
A429_IQP_TYPESEQ_MSKRXP	ADT_L1_A429_IQP_TYPESEQ_MSKRXP
A429_TXCB_CONTROL_STOPONTXBCOMP	ADT_L1_A429_TXCB_CONTROL_STOPONTXBCOMP
A429_TXCB_CONTROL_INTONTXBCOMP	ADT_L1_A429_TXCB_CONTROL_INTONTXBCOMP
A429_TXP_CONTROL_DELAYONLY	ADT_L1_A429_TXP_CONTROL_DELAYONLY
A429_TXP_CONTROL_TRIGIN	ADT_L1_A429_TXP_CONTROL_TRIGIN
A429_TXP_CONTROL_TRIGOUT	ADT_L1_A429_TXP_CONTROL_TRIGOUT
A429_TXP_CONTROL_INTERRUPT	ADT_L1_A429_TXP_CONTROL_INTERRUPT
A429_TXP_CONTROL_PARITYODD	ADT_L1_A429_TXP_CONTROL_PARITYODD
A429_TXP_CONTROL_PARITYON	ADT_L1_A429_TXP_CONTROL_PARITYON
A429_RXP_CONTROL_TRGOUT	ADT_L1_A429_RXP_CONTROL_TRGOUT
A429_RXP_CONTROL_INTERRUPT	ADT_L1_A429_RXP_CONTROL_INTERRUPT
A429_RXP_CONTROL_DECERROR	ADT_L1_A429_RXP_CONTROL_DECERROR

Using AltaAPI with TenAsys INtime™

Alta has tested the AltaAPI with the TenAsys INtime operating system (real-time extension to Microsoft Windows) running ONLY in shared mode (regular Windows XP mode/partition). There are no changes required and follow the regular Windows XP install procedures discussed earlier in this section. As of this release, Alta has not developed or tested the AltaAPI in the Real-Time (RTOS) mode of INtime.

Using AltaAPI with IntervalZero RTX™

Alta has tested the AltaAPI with the IntervalZero RTX operating system (real-time extension to Microsoft Windows).

The following installation and build instructions apply to RTX 2009, which is no longer supported by IntervalZero. However, these instructions apply equally to RTX 2011 and RTX 2012. Use this as a guide for implementation on these versions of RTX.

Installation

First perform the normal Windows installation of the AltaAPI software and install the Alta board in the system. Verify proper operation by running AltaView and/or by building and running the AltaAPI example programs under Windows.

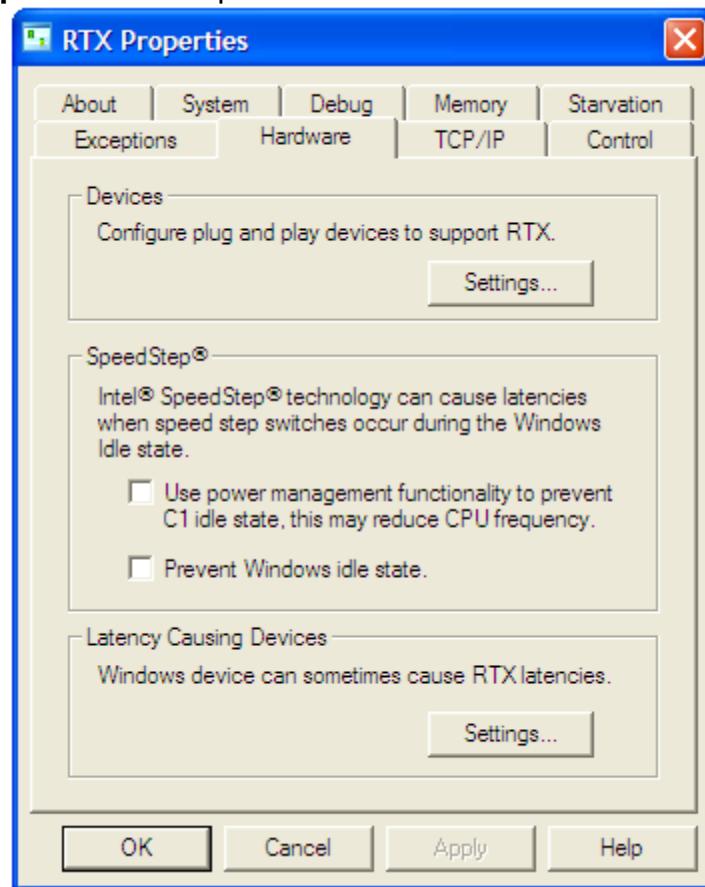
Configure the Alta Board as an RTX Device

You must convert the Alta hardware to an RTX Device before you can use it with RTX.
NOTE THAT YOU MUST HAVE THE ALTA BOARD INSTALLED IN THE SYSTEM
UNDER WINDOWS.

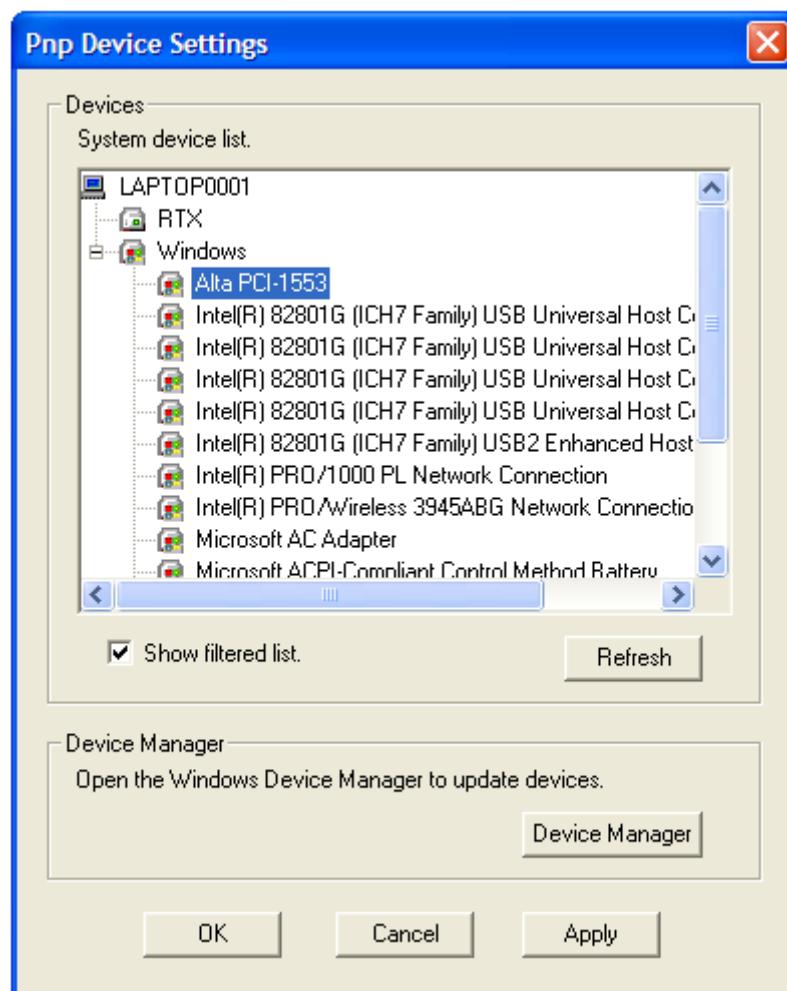
1. From the **Windows Start** menu, click **All Programs > IntervalZero > RTX 2009 > RTX Properties** to access the **RTX Properties** control panel.



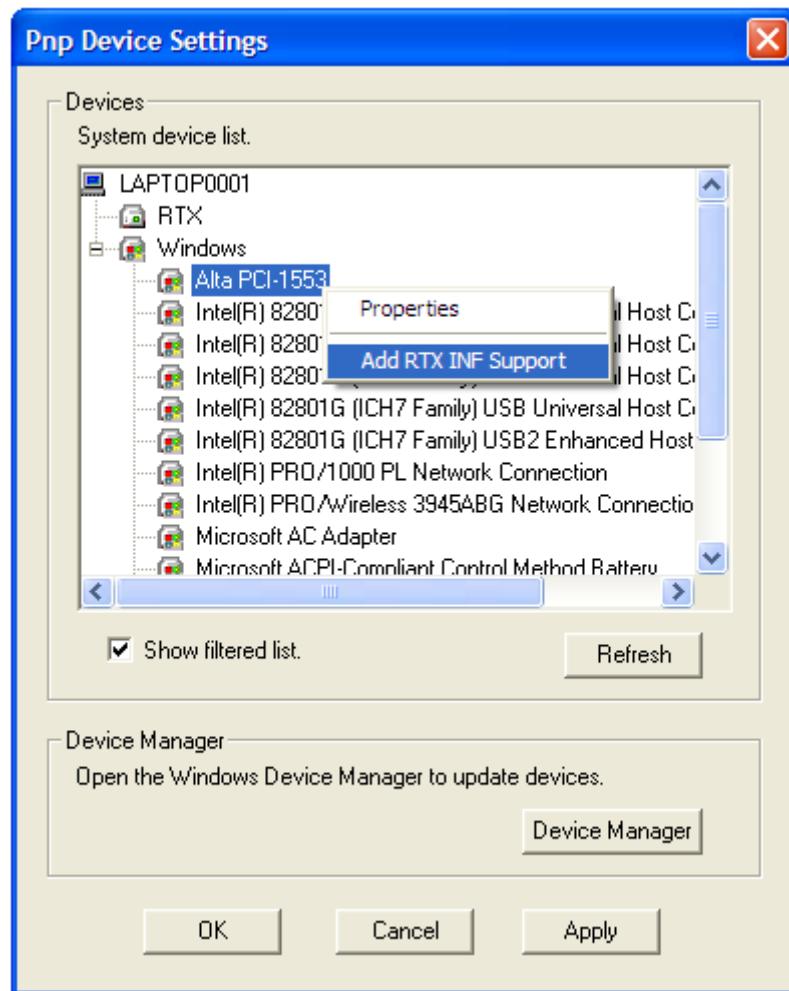
From the RTX Properties control panel select the **Hardware** tab.



2. In the **Devices** area, click on the **Settings** button to configure plug and play devices to support RTX. This shows the current plug and play devices on your system.

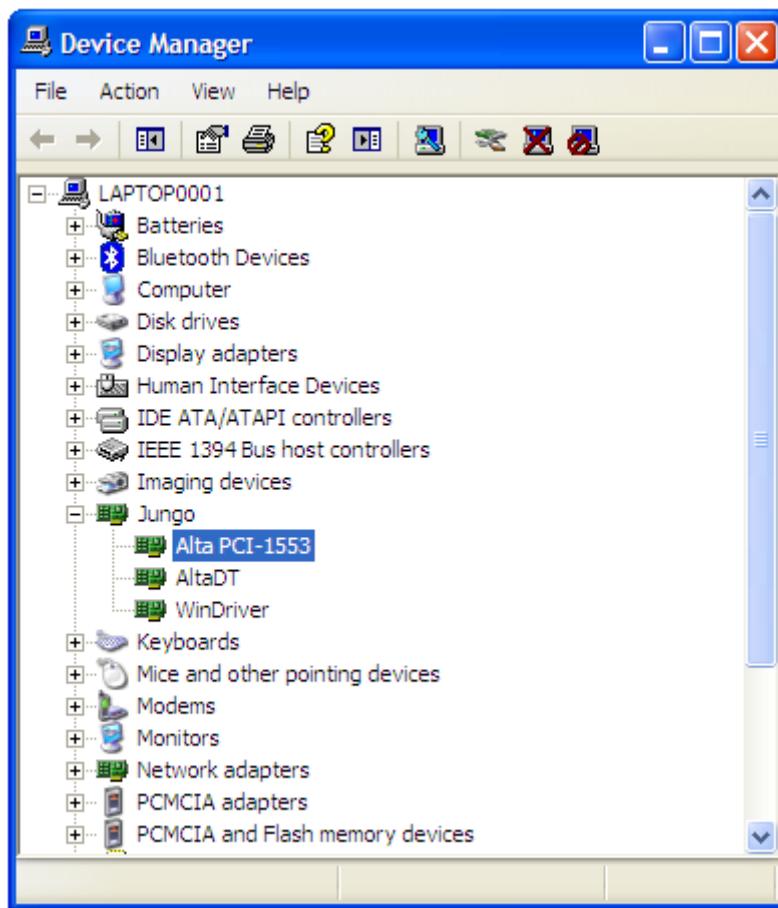


3. Right-click on the node for the Alta board you want to convert, then click **Add RTX INF Support**. This will edit the RtxPnp.inf file to support the selected device. In this example we are converting the Alta PCCD-1553.

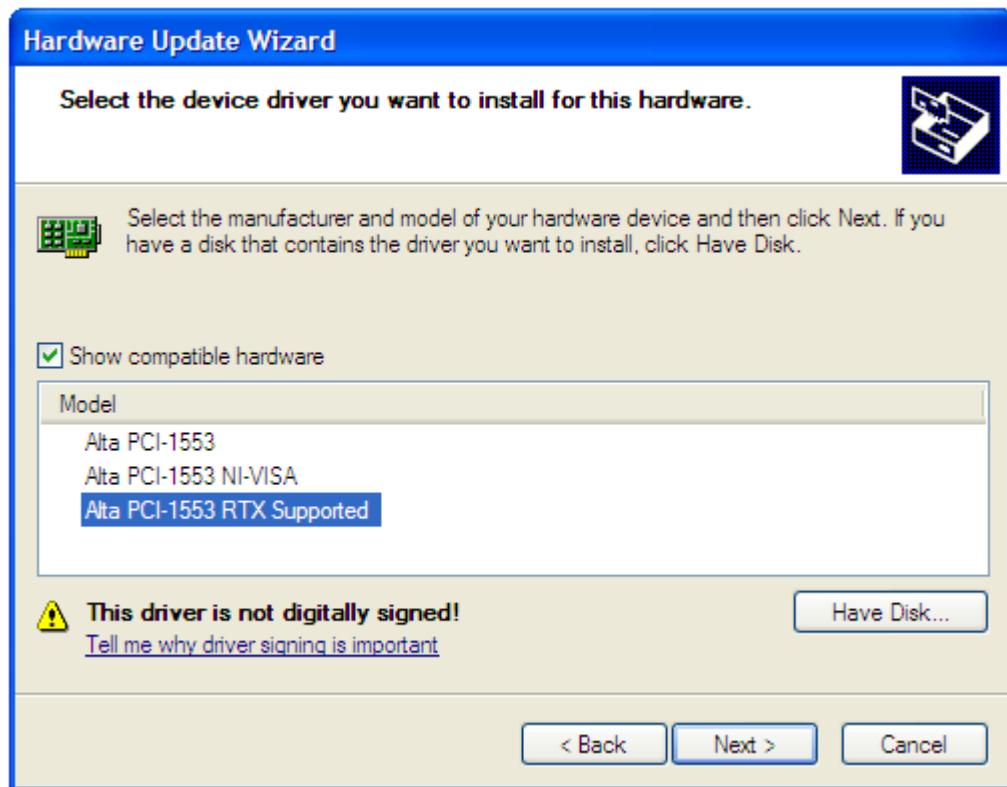


4. Click the **Apply** button.
5. Click the **Device Manager** button to open the Windows Device Manager.

6. Locate the device you just added RTX INF support for – note that Alta boards should show up under **Jungo**. Right-click on the device and select **Update Driver**.

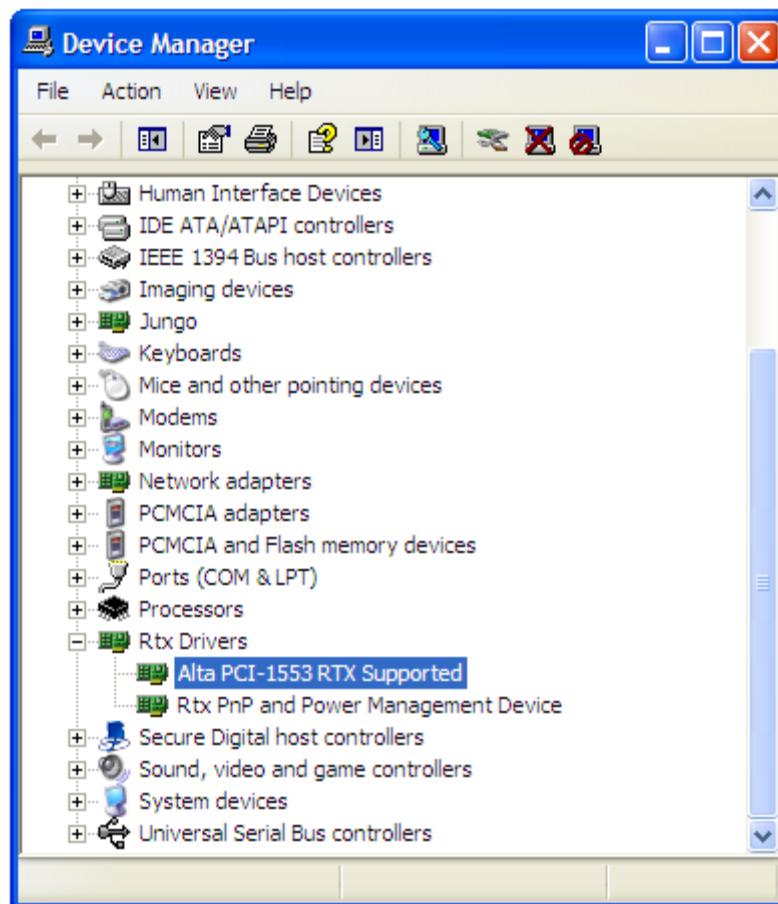


7. The **Found New Hardware Wizard** should now be displayed. If it asks if it can connect to Windows Update to Search for Software, select **No, not this time** and click **Next**. Select **Install from a list or specific location** and click **Next**. Choose the option **Don't search, I will choose the driver to install** and click **Next**.

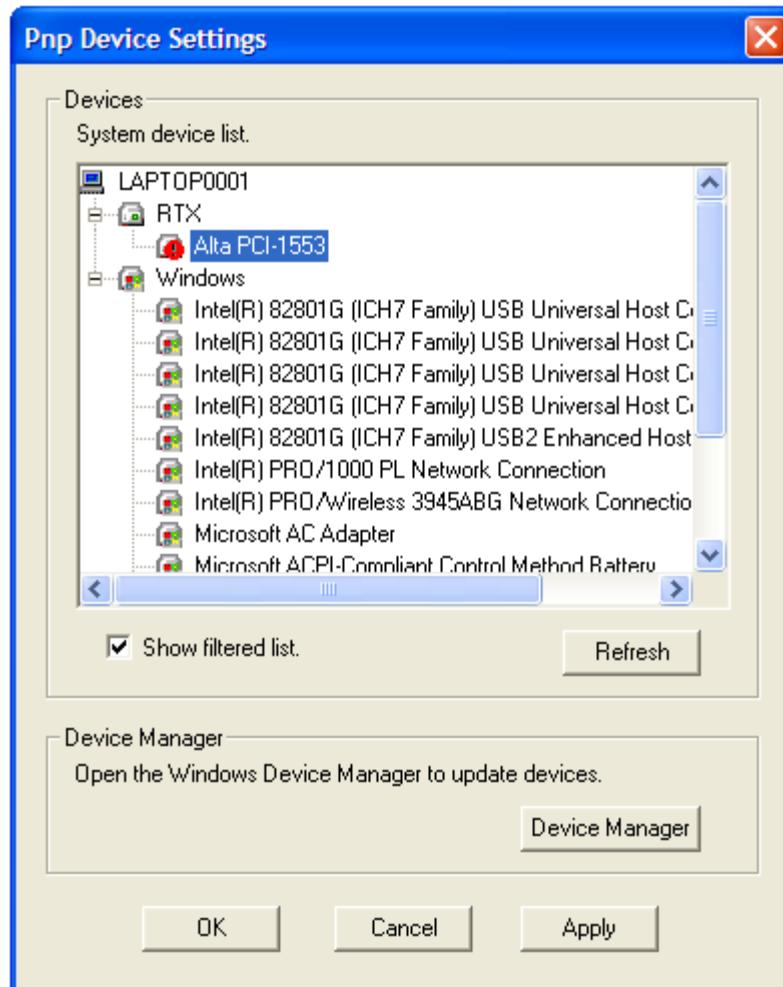


8. From the **Model List** select the device that is **RTX Supported** and click **Next**. When the wizard completes, click **Finish**. You may have to restart your computer for the changes to apply.

9. The device should now be displayed in the Windows Device Manager under **Rtx Drivers**.



10. Go back to the **RTX Properties** control panel. In the **PnP Device Settings** window, click the **Refresh** button. The device should now show up under **RTX** rather than **Windows**.



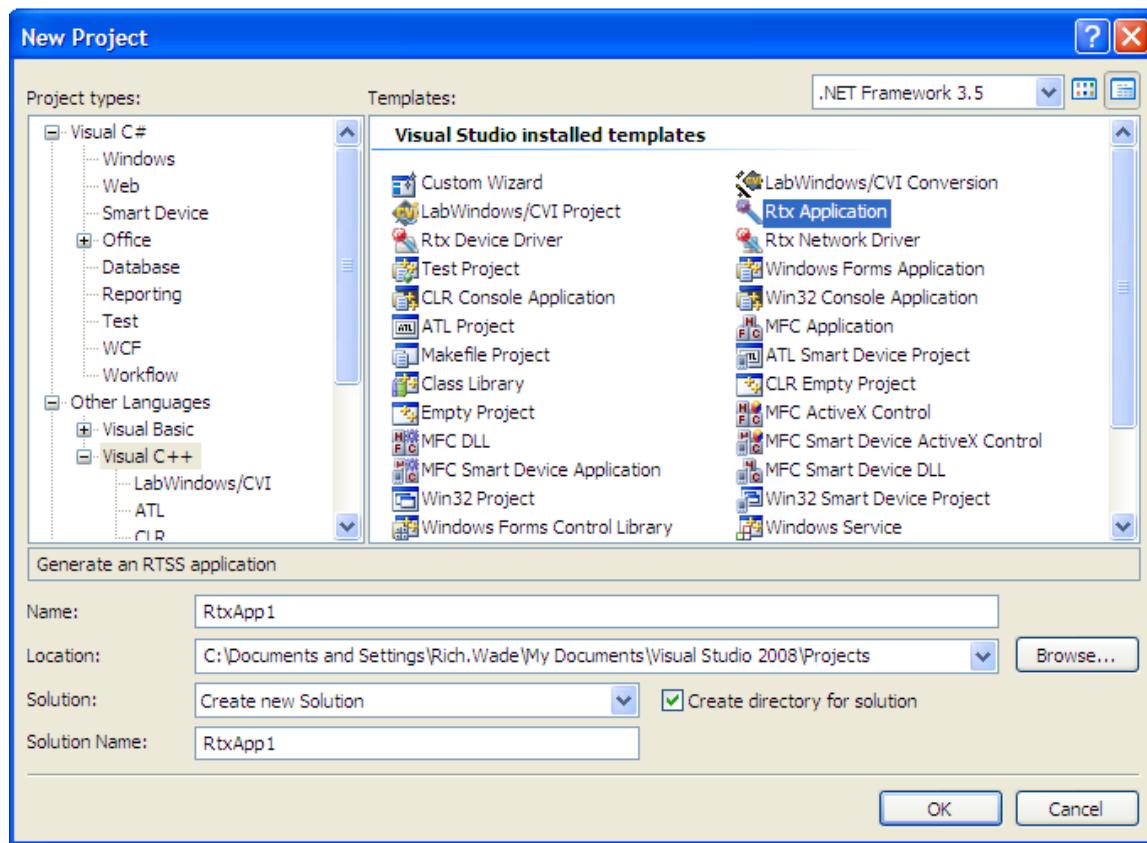
Configuration for Interrupts

At this point the Alta board is usable with RTX (either for Windows executables or for RTX rtss executables). However, the board may or may not be usable with hardware interrupts under RTX. **If your application will require interrupt support you will need to find a PCI slot that gives you an IRQ that is not shared with any other PCI devices.** This is explained in the IntervalZero white paper on “Working with Hardware Resource Limitations” – a PDF copy of this white paper is provided in the RTX folder on the Alta Software distribution CD. Note that Alta boards ONLY support Line-Based interrupts; Alta boards do not have the capability to support MSI interrupts.

Setup an RTX Application Project in MSVS

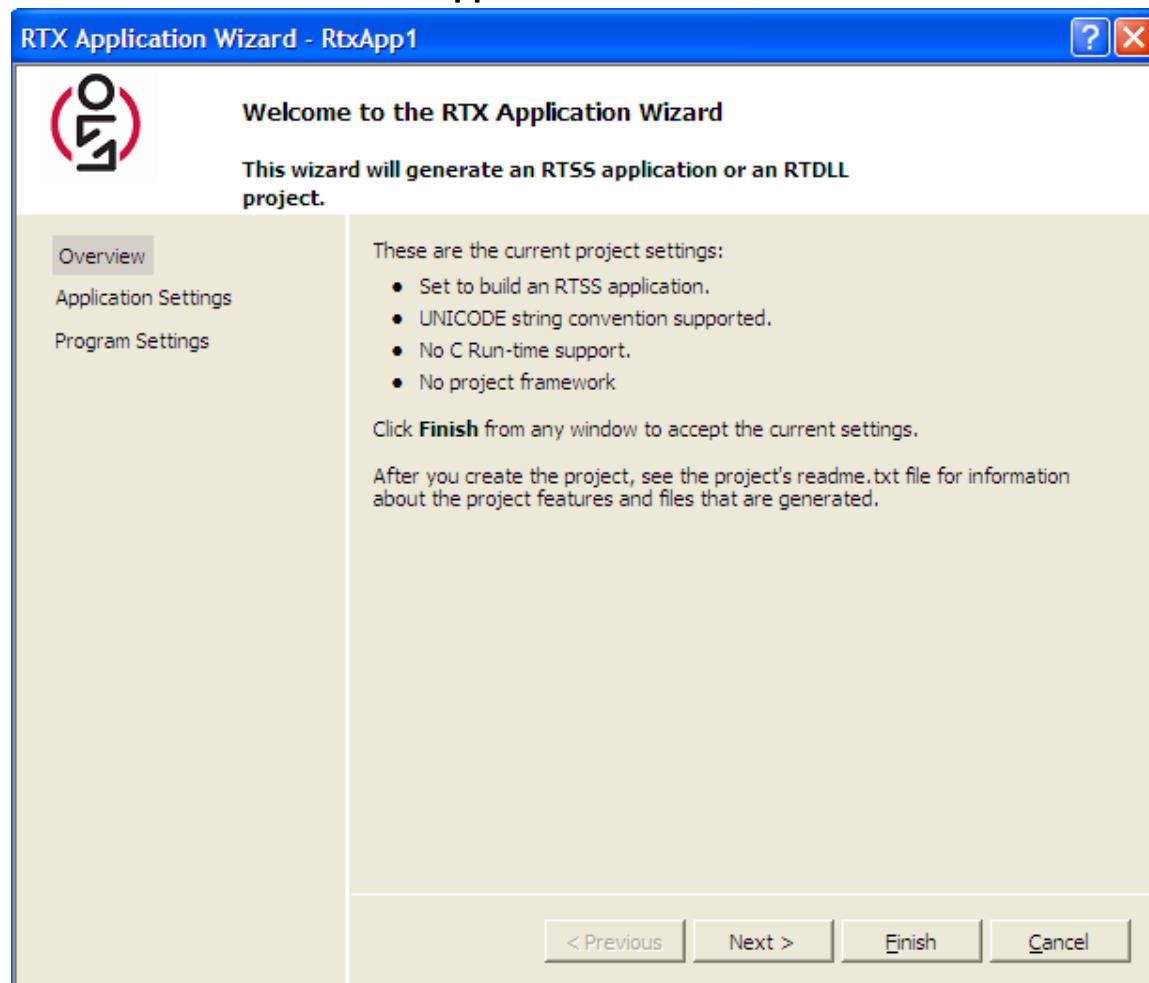
The RTX installation adds new project types to Microsoft Visual Studio to support building applications that will build either as Windows executables (exe files) or RTX executables (rtss files).

1. In MSVS, create a new project. Under **Project Types**, select **Visual C++**. From the Templates area, select **Rtx Application**. Enter the desired project name (or go with the default “RtxApp1”).

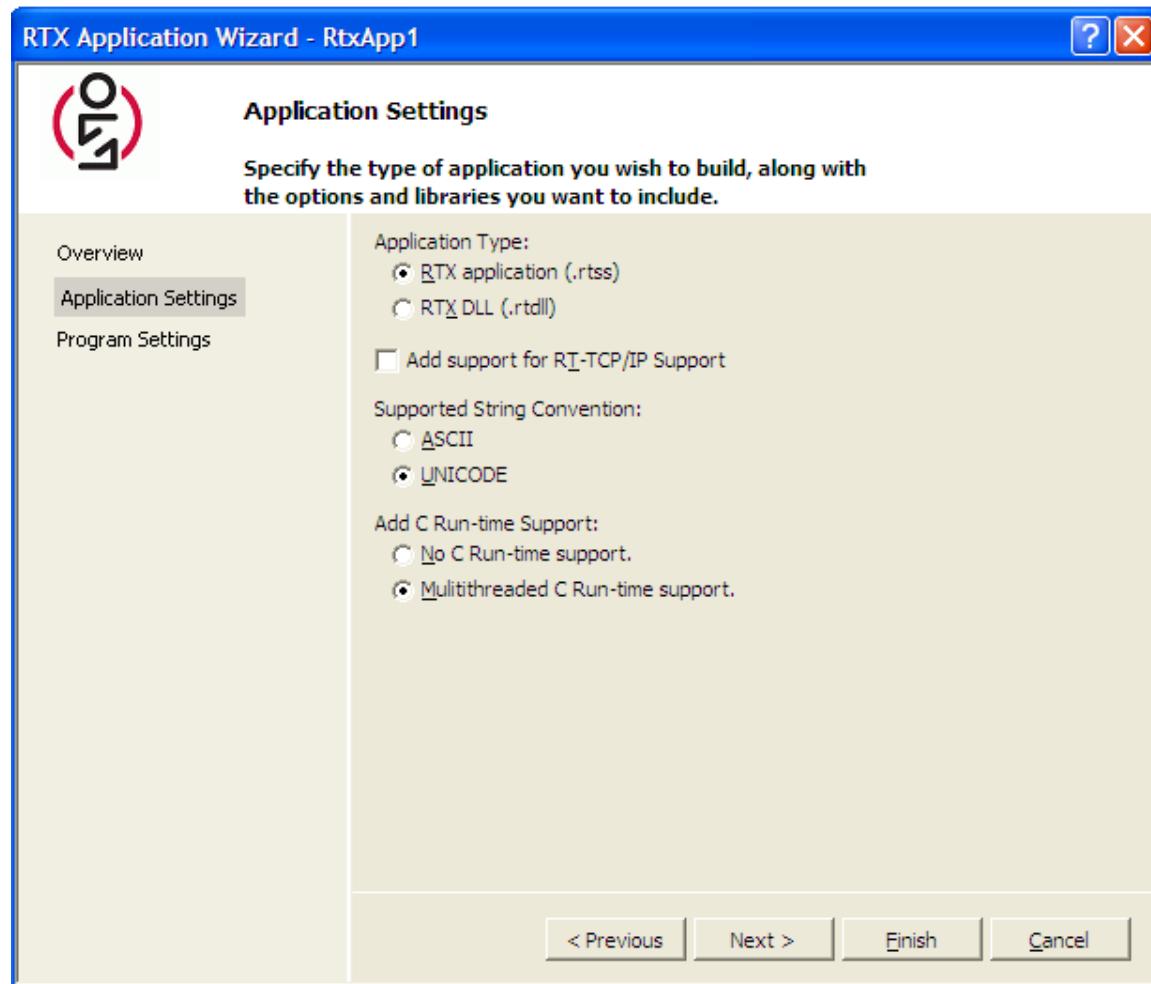


2. Click on the **OK** button.

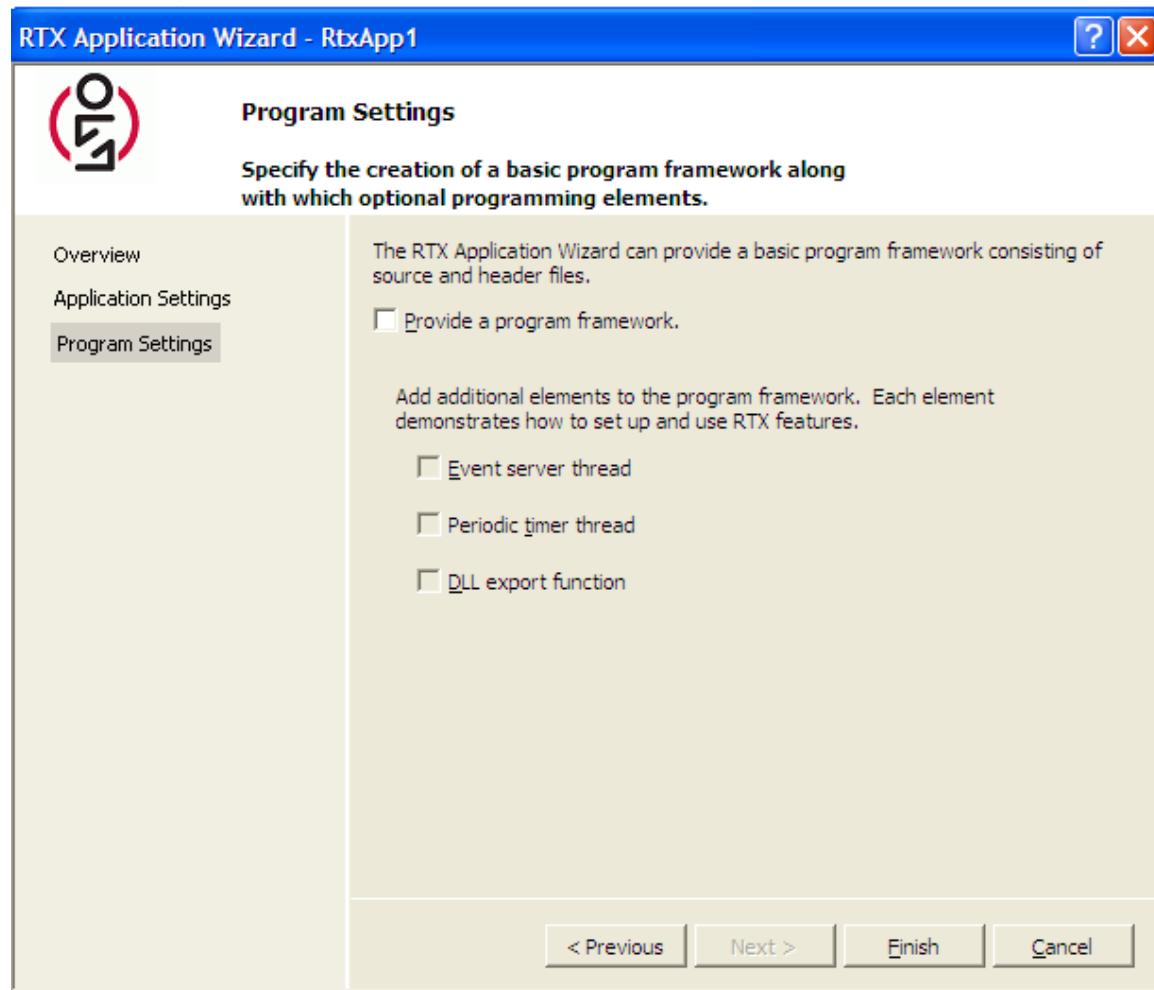
3. You will now see the **RTX Application Wizard**. Click the **Next** button.



4. You will now see the **Application Settings** window. Under **Add C Run-time support**, select **Multithreaded C Run-time support**. Click on the **Next** button.



5. You will now see the **Program Settings** window. You can use the defaults here, no change. Click on the **Finish** button.

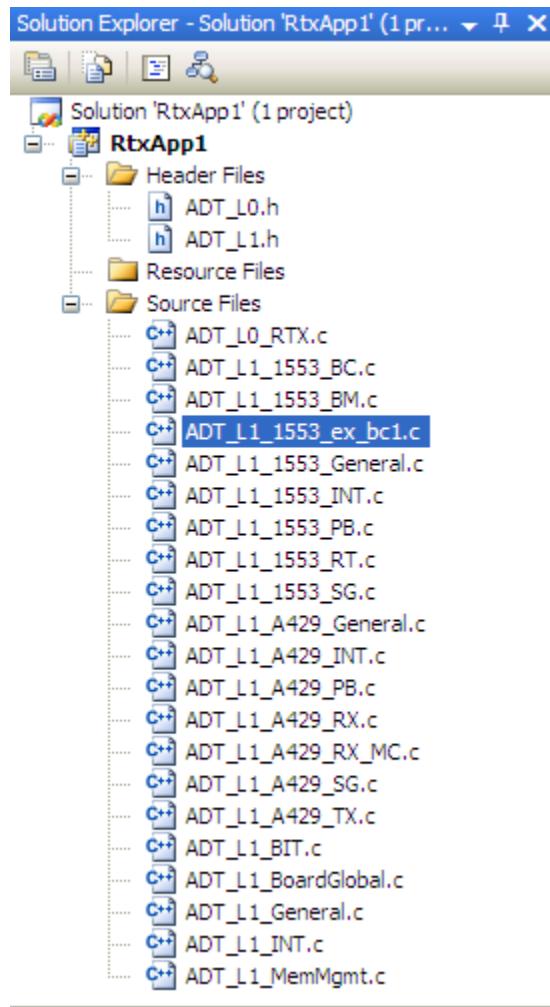


6. The wizard will create a folder for your project – in this example this folder is called “RtxApp1”. Now you need to copy the AltaAPI files to the project folder. The AltaAPI Installation CD provides a RTX folder. This folder contains a PDF white paper (on hardware resources – for interrupt configuration as mentioned earlier) and three folders for the AltaAPI Layer 0 (ADT_L0), the AltaAPI Layer 1 (ADT_L1), and example programs (ADT_L1_Examples), as shown below.



7. Copy all files from the ADT_L0 folder to your project folder – in our example this is (MSVS Project Path)\RtxApp1\RtxApp1.
8. Copy all files from the ADT_L1 folder to your project folder – in our example this is (MSVS Project Path)\RtxApp1\RtxApp1.
9. Copy ONE of the example programs to your project folder. The ADT_L1_Examples folder contains folders for A429 Examples and 1553 Examples. Let's assume you are installing a PCI-1553 board, so we will use one of the 1553 examples. Copy the **ADT_L1_1553_ex_bc1.c** file to your RTX application project folder (this is a very simple 1553 bus controller example that will send a single message on the bus).

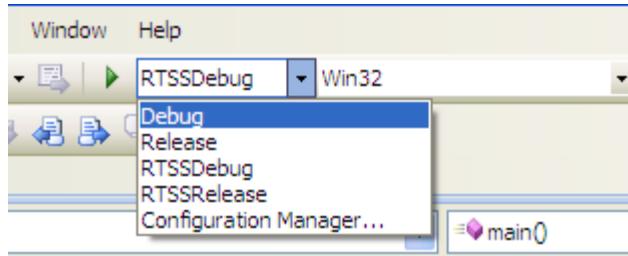
10. Add all these files to your project. The solution explorer should look something like this.



11. The file **ADT_L1_1553_ex_bc1.c** is your main program and all the other files are AltaAPI files. When you get to writing your own code you will replace the example program with your own main program. Before you build and run the program, edit the **ADT_L1_1553_ex_bc1.c** file and check the #define for DEVID. This tells the software what kind of Alta board you are using – if this is wrong you will get an error (error 3) when the program tries to initialize the device. The constants used to define the DEVID are defined in the ADT_L0.h file. The example below shows the correct setting for a PCI-1553 board.

```
/* The DEVICE ID is a 32-bit value that identifies the following:  
 * bits 28-31 = Backplane Type (0 = Simulated, 1 = PCI)  
 * bits 20-27 = Board Type (0 = SIM-1553, 1 = TEST-1553, 2 = PMC-1553, 3 = PC104P-1553, 4 = PCI-1553)  
 * bits 16-19 = Board Number (0 to 15)  
 * bits 8-15 = Channel Type (0x10 = 1553)  
 * bits 0-7 = Channel Number (0 to 255)  
 *  
 * A device ID of 0x10201000 specifies the first 1553 channel of the  
 * first ADT PMC-1553 board found.  
 * A device ID of 0x10201001 specifies the second 1553 channel of the  
 * first ADT PMC-1553 board found.  
 */  
#define DEVID (ADT_PRODUCT_PCI1553 | ADT_DEVID_BOARDNUM_01 | ADT_DEVID_CHANNELTYPE_1553 |ADT_DEVID_CHANNELNUM_01)
```

12. You can select Debug and Release configurations for both Windows and for RTX. First build a Win32 Debug configuration and test the program. If everything works in Windows you can then build a RTSS executable that you can run from the RTX Task Manager.



NOTE: Most of the example programs use simple I/O with printf and scanf statements. This will work fine when run on Windows, but when running as RTSS keyboard input with scanf is not supported by RTX. If you run the example programs as RTSS the scanf statements will not wait for user input, they will just be skipped.

RTX 2012 Execution Note

In RTX 2012, IntervalZero introduced a utility called “Stamp Tool”, which is used to “stamp” an RTSS application or RTDLL with a validated license string before execution of the RTSS application or use of the RTDLL.

If you do not properly stamp the application/DLL, you will receive this error message:
“**This application was not built with a valid license**”

The Stamp Tool utility is described in the RTX 2012 documentation and can be found in **%RTXDIR%\bin**.

Appendix C – Linux

This appendix discusses AltaAPI installation under the Linux operating system. Alta uses the Jungo WinDriver tools (version 12.2.1) for Linux device drivers.

NOTE:

If you are only using ENET devices you can use the ENET-ONLY BSD API, discussed in **Appendix A**. This API does not use a kernel-level driver and can be used on almost all platforms. You cannot use the ENET-ONLY BSD API with PCI/PCIe devices

Linux Distribution/Kernel Versions

The Alta Linux driver has been used successfully on RedHat, Concurrent RedHawk, CentOS, Debian and Ubuntu Linux distributions. **Alta does not currently support Linux on PowerPC.**

Alta supports standard kernels up through **version 4.6.2**. Standard LINUX kernels are available at <http://www.kernel.org/>.

Alta does not support special kernel builds, embedded Linux kernels, etc. We use the Jungo WinDriver package for Linux drivers and only support Linux systems compatible with the Jungo driver.

Supported Alta Products

All Alta products are supported for x86 Linux. This includes all PCI, PCI Express, and ENET products for MIL-STD-1553 and A429.

Supported C compilers

The AltaAPI was built with the GNU C compiler. No other compilers are specifically supported. **Alta does not support Linux cross-compiler environments (for example, BlueCat Linux from LynuxWorks).**

Prior to installation

Linux device driver modules are compiled with the kernel header files. To install a module into the kernel, the module must be built with the same **version.h** as the kernel.

Some distributions provide a compiled kernel without the file **version.h**. If so, you must install the Linux source code, build the kernel, and then build AltaDriver and the PlugIn module with the kernel **version.h**.

If the Linux source code is installed there will be “linux-<Kernel_Version>” (or a corresponding entry) in the **/usr/src** directory (**/usr/src/kernels** on RedHat/CentOS/Fedora). If the source code is not installed, either install the source code or reinstall Linux with the source code.

1. You can install the kernel source with the following:

- **Fedora (FC 22 and later)** Newer versions deprecate “yum”, which is being replaced with “dnf”. You can install the kernel source with the following:

```
$ su  
<enter root password>  
# dnf install "kernel-devel-uname-r == $(uname -r)"  
# exit
```

- **Fedora/RedHat/CentOS**

```
$ su  
<enter root password>  
# yum install kernel-headers kernel-devel  
# exit
```

- **Ubuntu**

```
$ sudo apt-get install linux-source
```

- **Debian**

```
$ su  
<enter root password>  
# apt-get install linux-source  
# exit
```

2. You must also create a symbolic link called **linux** to the kernel source directory in **/usr/src**.

- **For RedHat/CentOS/Fedora:**

```
[/usr/src]$ su  
<enter root password>  
[/usr/src]# ln -s kernels/<kernel_version> linux  
[/usr/src]# exit
```

- **For Ubuntu:**

```
[/usr/src]$ sudo ln -s linux-headers-<kernel_version> linux
```

- **For Debian:**

```
[/usr/src]$ su  
<enter root password>  
[/usr/src]# ln -s linux-headers-<kernel_version> linux  
[/usr/src]# exit
```

3. Debian linux needs a symbolic link to find “utsrelease.h”. The Jungo driver expects to find it at “/lib/modules/<kernel>/source/include/linux” (for Debian 7.8.0 amd64, <kernel> is “3.2.0-4-amd64”, replace with your kernel appropriately). However, Debian has this file at “/usr/src/linux-headers-3.2.0-4-amd64/include/generated”. We can create a symbolic link from where the driver expects the file to where the file really is.

- **For Debian:**

```
[/usr/src]$ su  
<enter root password>  
[/usr/src]# ln -s  
    /usr/src/linux-headers-3.2.0-4-amd64/include/generated/utsrelease.h  
    /lib/modules/3.2.0-4-amd64/source/include/linux/utsrelease.h  
[/usr/src]# exit
```

Installing AltaAPI

Shut down the system and install the Alta board. Turn on the system. Copy the file **ADT_Linux_x86_32(64).x.x.x.x.tar** for 32-bit (or 64 bit) (where .x.x.x.x is the current release version) from the **Linux** directory on the installation CD to an appropriate directory on your system. Extract the files with the following command:

```
[YourDirectory]$ tar -xvf ADT_Linux_x86_32(64).x.x.x.x.tar
```

This will create a directory called **ADT_API_x.x.x.x** containing the following:

- **ADT_Plugin** - contains kernel plug-in files.
- **AltaAPI_manpages** – contains man-page files for each of the AltaAPI Layer 1 functions.
- **altadriver_installation** - contains kernel driver files.
- **Examples** - contains the source files for 1553 and A429 example programs.
- **Source** - contains the **AltaAPI Layer 1** source files along with a makefile to build the Layer 1 shared object file.
- **Test** - contains the Layer 0 and Layer 1 shared object files, selected example programs, and a sample makefile that builds an example program into an executable file.
- **AltaAPI User's Manual**

Building and installing AltaDriver

1. Go to the driver installation directory:

```
[YourDirectory]$ cd ADT_API_x.x.x.x/altadriver_installation/redist
```

2. Run the configuration script. This will create the makefile needed to build and install the driver:

```
[redist]$ ./configure --disable-usb-support
```

3. Build the driver:

```
[redist]$ make
```

4. Install the driver (as super user):

```
[redist]$ su  
[redist]# make install
```

5. Change the protections on the /dev key node for the driver to allow users to access it:

```
[redist]# chmod 666 /dev/altadriver  
[redist]# exit
```

Caution: Since **/dev/altadriver** gives direct hardware access to user programs, it may compromise kernel stability on multi-user Linux systems. Access to this file should be restricted to trusted users.

Building and Installing the Kernel PlugIn

The Kernel Plug-in is **required** for using hardware interrupts from the Alta boards.

Note: If you do not install the kernel plug-in then you must use the “extended options” initialization function (ADT_L1_1553_InitDefault_ExtendedOptions for 1553 devices, ADT_L1_A429_InitDefault_ExtendedOptions for A429 devices) with the option ADT_L1_API_DEVICEINIT_NOKP.

Note: altadriver must be installed prior to installing the kernel PlugIn. Attempting to install altadriver after the kernel plug-in will result in a system error.

1. Go to the kernel plug-in installation directory:

```
[YourDirectory]$ cd ADT_API_x.x.x.x/ADT_PlugIn/configure
```

2. Run the configuration script. This will create the makefile needed to build and install the PlugIn:

```
[configure]$ ./configure
```

3. For 64-bit Linux only:

Edit the generated Makefile, find EXTRA_CFLAGS, add -DKERNEL_64BIT.

4. Build the PlugIn:

```
[configure]$ make
```

5. Install the PlugIn:

```
[configure]$ su  
[configure]# make install  
[configure]# exit
```

Building the AltaAPI Example Programs

A sample makefile is provided to help build your first API example program. This makefile can be copied and edited to create make files for other example programs or for your own test programs. Edit the MAKEFILE to use the correct example for your Alta device:

- 1553 - **ADT_L1_1553_ex_rt1.c**
- A429 - **ADT_L1_A429_ex_rxtx1.c**

NOTE: You **MUST** edit the example program to change the #define for DEVID to match your product type (PCI1553, PMC1553, PC104P1553, etc.).

Additional example programs are located in the **ADT_API_x.x.x.x/Examples** directory. There is a directory and associated readme for both 1553 and A429 Examples.

1. Go to the ADT_API_x.x.x.x/Test directory:

```
[YourDirectory]$ cd ADT_API_x.x.x.x/Test
```

2. Create a symbolic link to the AltaAPI Layer 0 library (x.x.x.x is the release version):

```
[Test]$ ln -sf libADT_L0_Linux_x86_32(64).so.x.x.x.x libADT_L0_Linux_x86_32(64).so
```

3. Create a symbolic link to the AltaAPI Layer 1 library (x.x.x.x is the release version):

```
[Test]$ ln -sf libADT_L1_Linux_x86_32(64).so.x.x.x.x libADT_L1_Linux_x86_32(64).so
```

Note: The ln -sf commands above with 32(64) refer to linking either the 32-bit L0/L1 shared object for 32-bit applications or the 64-bit L0/L1 shared object for 64-bit applications.

(The Layer 1 shared object file can be built from source in the /Source directory.)

4. Add this directory to the library load path so the compiler can find the shared library:

```
[Test]$ export LD_LIBRARY_PATH=.:$LD_LIBRARY_PATH
```

5. Build the API and example program:

NOTE: You **MUST** edit the example program to change the #define for DEVID to match your product type (PCI1553, PMC1553, PC104P1553, etc.).

```
[Test]$ make
```

Note: The **make** command above is applicable for 32-bit applications on 32-bit Linux or 64-bit applications on 64-bit Linux.

For creating 32-bit applications on 64-bit Linux, use **make -f make_32bit**

6. Run the example program:

```
[Test]$ ./test_rt1
```

You should see output similar to this:

```
***** ADT L1 API Example Program *****
```

```
Initializing . . . Success.  
Checking Protocol Engine (PE) and API versions . . . Success.  
    PE version = 0600  
    L0 API version = 3.0.2.0  
    L1 API version = 3.0.2.0  
Initializing RT1 . . . Success.  
Enabling RT1 . . . Success.  
Starting RT operation . . . Success.
```

```
Input Q to quit or anything else to read the RT 1 default CDP buffer.
```

You can use another device as a bus controller to send messages to RT 1 to verify that the board is working.

7. Error Code 3

If you see **error 3** on initialization, this indicates that the specified device could not be found in the system. The usual reason for this is failure to edit the example program to specify the appropriate product type in the Device ID (DEVID). As mentioned previously:

NOTE: You **MUST** edit the example program to change the #define for DEVID to match your product type (PCI1553, PMC1553, PC104P1553, etc.).

8. Error Code 19

If you see **error 19** on initialization, this indicates that the kernel plug-in is not installed. Refer to the section on “Building and Installing the Kernel PlugIn”.

Now you can try other example programs or write your own programs.

NOTE ON MSI INTERRUPTS:

PCI Express devices use Message Signalled Interrupts (MSI). If you have two (or more) independent applications controlling different channels of the same PCIe board, and two or more of these applications use interrupts, this will not work – the system will crash because the last application to enable interrupts overwrites the setting made by the others.

- You CAN use independent applications on different channels of the same PCIe board if no more than one of them uses interrupts.
- You CAN use interrupts on multiple channels of the same PCIe board if all of the channels using interrupts are controlled by the same application.
- You **CANNOT** use interrupts on multiple channels of the same PCIe board if the channels using interrupts are controlled by separate independent applications.

Symbol Setup

If you reboot the system the symbols created during the installation process will be lost. Below are suggestions for creating the Alta driver symbols:

1. LD_LIBRARY_PATH

- Add the following command to your **/etc/bashrc** file:

```
export LD_LIBRARY_PATH=<your directory>/ADT_API_x.x.x.x/Test:$LD_LIBRARY_PATH
```

Note: /etc/bashrc is one of many shell Config scripts used. This may differ on your system or shell.

- If you do not set the Alta library path during login type the following commands before attempting to build your application:

```
[YourDirectory]$ cd ADT_API_x.x.x.x/Test
[Test]$ export LD_LIBRARY_PATH=.:$LD_LIBRARY_PATH
```

2. Loading the Driver

- To automatically load the Alta driver during bootup, run the wdreg script from the **/etc/rc.d/rc.local** file:

```
<your directory>/ADT_API_x.x.x.x/altadriver_installation/redist wdreg altadriver
chmod 666 /dev/altadriver
```

- If you do not execute the wdreg command automatically on bootup, you can do it manually as follows:

```
[YourDirectory]$ cd ADT_API_x.x.x.x/altadriver_installation/redist
[redist]$ su
Password: (enter root password)
[redist]# ./wdreg altadriver
[redist]# chmod 666 /dev/altadriver
[redist]# exit
```

3. Loading the Kernel Plugin

- Add the following command to the **/etc/rc.d/rc.local** file where < . . . > is the kernel version for your system:

```
insmod [YourDirectory]/ADT_API_x.x.x.x/ADT_Plugin/configure/<LINUX . .
.>/kp_altadriver_module.ko
```

Note: If you do not install the kernel plugin automatically on bootup as above, you can install it manually as follows:

```
$ cd [YourDirectory]\ADT_API_x.x.x.x\ADT_Plugin\configure\<LINUX...>
$ su
Password: (enter root password)
# insmod kp_altadriver_module.ko
# exit
```

Uninstalling the Kernel PlugIn and Driver

You can uninstall the plug-in and driver as follows (as root/superuser):

First verify that the Alta modules are present.

```
# /sbin/lsmod | grep alta
```

You should see two Alta modules – **kp_altadriver_module** is the kernel plug-in and **altadriver** is the Alta driver.

Unload the kernel plug-in module.

```
# /sbin/rmmod kp_altadriver_module
```

Unload the Alta driver module.

```
# /sbin/modprobe -r altadriver
```

If you have added to your /etc/rc.d/rc.local file to load the Alta driver/plug-in on boot, edit this file to remove your additions.

Remove the WinDriver shared object file.

For 32-bit Linux, this is **/usr/lib/libwdapiXXXX.so**.

For 64-bit Linux, this is **/usr/lib64/libwdapiXXXX.so**.

Note that the “XXXX” is the Jungo WinDriver version, will be something like 1210 (for version 12.1.0).

Appendix D – VxWorks

This appendix discusses installation and operation under Wind River VxWorks operating systems.

NOTE:

If you are only using ENET devices you can use the ENET-ONLY BSD API, discussed in **Appendix A**. This API does not use a kernel-level driver and can be used on almost all platforms. You cannot use the ENET-ONLY BSD API with PCI/PCIe devices

Supported VxWorks Versions

Alta currently supports **VxWorks 5.X**, **VxWorks 6.x** and **VxWorks 7.x**. This has been tested with VxWorks 5.5 on an AMPRO Core Module 800 (Pentium M) with an Alta PC104P-1553 board. This has been tested with VxWorks 6.3 on a Mercury VPA-200 PPC SBC with an Alta PMC-1553 board and an Alta PMC-A429 board. This has been tested with VxWorks 6.5 with an Alta PMC-1553 board on a Curtis Wright 185. This has been tested with VxWorks 6.9 with an Alta PMC-1553, ENET-1553, ENET2-1553, ENET-A429, ENET-A429P, and PMC-MA4 on an Emerson MVME3100. This has been tested with VxWorks 7.0 with an Alta PMC-1553, XMC-1553, PMC-MA4 ad XMC-MA4 on an X-ES Xpedite 7570.

You may have to modify the Layer 0 API functions to be compatible with other VxWorks systems, platforms, and/or Board Support Packages.

See the file **VxWorks_readme.txt** in the ADT_L0 directory for information about memory map calls used on different systems. For most systems, the Alta memory mapping and interrupt handling can use standard VxWorks supplied PCI functions. Some CPUs (especially x86) seem to want their supplier-specific memory mapping calls. This file provides examples of our experience with different vendors.

Supported Alta Products

The PMC-1553, PC104P-1553, PMC-MA4, PMC-A429, XMC-1553 and the XMC-MA4 are the only Alta products we have tested VxWorks. Contact Alta if you need to use products not listed here.

Supported Development Environments

Alta uses the **Tornado 2.2** development environment to build for VxWorks 5.5 targets. The **Workbench** 3.3 development environment is used for VxWorks 6.x and Workbench 4.0 development environment is used for VxWorks 7.0.

For VxWorks 5.X and 6.X

Hardware Installation

Shut down the system and install the Alta board. You can use the **pciDeviceShow** command from the VxWorks console to verify that the board is seen by the system. The Alta PCI Vendor ID is **0xAD00**. Once you have located the Alta board, you can use the **pciHeaderShow** command to see the PCI configuration registers – you will need to know the BAR0 and BAR2 base addresses assigned to the board.

System Configuration

This does not apply to all systems. Adding space for mapped PCI/PCIe devices can be SBC and/or BSP specific.

Modify the **sysLib.c** file to add an entry to the **sysPhysMemDesc** table for the Alta board(s) being installed. Rebuild after modifying the file.

Each board requires 512 Bytes of PCI BAR0 address space and 8MB of PCI BAR2 address space. Note that the addresses and memory size must be aligned to the page size (4KB in the example below).

The physical PCI address should match the BAR0 and BAR2 values in the PCI configuration registers as shown by **pciHeaderShow**. In this example, the physical and virtual addresses are the same (VxWorks 5.5) – this can be different for other systems/BSPs.

```
{ /* Alta 1553 Board – 8 MB for BAR2 */
(void *) 0xFE000000, /* Physical PCI Address (4KB aligned) */
(void *) 0xFE000000, /* Virtual Address (4KB aligned) */
0x800000,           /* Size in bytes (4KB aligned) */
/* initial state mask */
VM_STATE_MASK_VALID | VM_STATE_MASK_WRITABLE | VM_STATE_MASK_CACHEABLE,
/* initial state */
VM_STATE_VALID | VM_STATE_WRITABLE | VM_STATE_CACHEABLE_NOT
},

{ /* Alta 1553 Board – 512 B for BAR0 */
(void *) 0xFE8FF000, /* Physical PCI Address (4KB aligned) */
(void *) 0xFE8FF000, /* Virtual Address (4KB aligned) */
0x1000,             /* Virtual Address (4KB aligned) */
/* initial state mask */
VM_STATE_MASK_VALID | VM_STATE_MASK_WRITABLE | VM_STATE_MASK_CACHEABLE,
/* initial state */
VM_STATE_VALID | VM_STATE_WRITABLE | VM_STATE_CACHEABLE_NOT
}
```

Software Installation

Copy the directory “**Alta VxWorks**” from the VxWorks directory on the installation CD to an appropriate directory on your system (where the Tornado/Workbench development environment will be used to build code for your VxWorks target). This directory contains the VxWorks Layer 0 API files, the Layer 1 API files, and the Layer 1 example programs.

Testing the Installation and L0 API

The Layer 0 API is the fundamental interface to the board for memory mapping and interrupt handling. The Layer 0 API consists of the following files:

ADT_L0.h
ADT_L0_VxWorks.c

NOTE: The ADT_L0_VxWorks.c file contains byte/word swapping macros that are used differently for Big-Endian (PPC) or Little-Endian (x86) systems. You need to modify this file to define these for your system. Define the constant X86_PLATFORM if you are using an x86 system. Comment out this #define statement if you are using a PPC system. You may also have to modify the ADT_L0_VxWorks.c file for the call to enable interrupts – make the appropriate call for your BSP.

A simple test program is provided to verify memory mapping of the board:

ADT_L0_test1.c

Use the Tornado/Workbench environment to build the ADT_L0_test1.c program. This program requires the ADT_L0_VxWorks.c file and the ADT_L0.h file.

When you run the ADT_L0_test1 program it will attempt to map memory to the board and will read and print several words from memory. The “Board-Level Align Check Word” value should be **0x12345678**. If you do not see this value then there is a memory mapping problem or there is a byte/word swapping problem. If you see swapped bytes/words the byte/word swapping macros in ADT_L0_VxWorks.c are not defined correctly for your system (see note above). If you see other unexpected values (or if the test program crashes or otherwise fails to run) then the problem may be incorrect physical or virtual memory settings.

Building and Testing the L1 API and Example Programs

After you have verified that the Layer 0 API is functioning properly you can then add the Layer 1 API and Layer 1 example programs.

The Layer 0 API consists of the following files:

ADT_L0.h
ADT_L0_VxWorks.c

The Layer 1 API consists of the following files:

ADT_L1.h
ADT_L1_General.c
ADT_L1_BoardGlobal.c
ADT_L1_MemMgmt.c
ADT_L1_BIT.c
ADT_L1_INT.c
ADT_L1_1553_General.c
ADT_L1_1553_INT.c
ADT_L1_1553_RT.c
ADT_L1_1553_BC.c
ADT_L1_1553_BM.c
ADT_L1_1553_PB.c
ADT_L1_1553_SG.c
ADT_L1_A429_General.c
ADT_L1_A429_INT.c
ADT_L1_A429_RX.c
ADT_L1_A429_TX.c
ADT_L1_A429_RX_MC.c
ADT_L1_A429_PB.c
ADT_L1_A429_SG.c

In addition to the API files listed above, you will need one of the Layer 1 example programs. A good initial test (for 1553 boards) is to start with a simple Remote Terminal program:

ADT_L1_1553_ex_rt1.c

A good initial test (for A429 boards) is to start with a simple transmit/receive program:

ADT_L1_A429_ex_rxtx1.c

Note that the “standard” example programs use “main()” for the main program entry point. VxWorks applications typically use a main function name other than “main()”, so when moving the standard example programs to VxWorks you should change “main()” to something else (like “test_rt1()”, for example).

You will need to edit the #define for the DEVID in the example program to match the type of Alta board you are using. For example, the following specifies the Alta PMC-1553 board type:

```
#define DEVID (ADT_PRODUCT_PMC1553 | ADT_DEVID_BOARDNUM_01 |  
ADT_DEVID_CHANNELTYPE_1553 |ADT_DEVID_CHANNELNUM_01)
```

The example program can now be compiled with the API functions to generate an executable program for the VxWorks system.

If you see **error 3** on initialization, this indicates that the specified device could not be found in the system. The usual reason for this is failure to edit the example program to specify the appropriate product type in the Device ID (DEVID). As mentioned above:

NOTE: You **MUST** edit the example program to change the #define for DEVID to match your product type (PCI1553, PMC1553, PC104P1553, etc.).

When you have the example programs running successfully on your system, you can then expand on the example code for your own applications.

For VxWorks 7.X

System Configuration

VxWorks 7.X utilizes the new VXBus Gen II interface, the devices resources will be reserved by the operating system once it has been identified by the driver.

Software Installation

Copy the directory “**L0 API VxWorks7**” from the VxWorks directory on the installation CD to an appropriate directory on your system (where the Tornado/Workbench development environment will be used to build code for your VxWorks target). This directory contains the VxWorks Layer 0 API files, the Layer 1 API files, and the Layer 1 example programs. The Alta driver consists of the following files:

```
AdtVxWorks7DevDriver.c  
AdtVxWorks7DevDriver.h
```

Should be copied into the VxWorks Image Project (VIP) for the particular SBC.

Testing the Installation and L0 API

The Layer 0 API is the fundamental interface to the board for memory mapping and interrupt handling. The Layer 0 API consists of the following files:

```
ADT_L0.h  
ADT_L0_VxWorks7.c
```

NOTE: The ADT_L0_VxWorks7.c file contains byte/word swapping macros that are used differently for Big-Endian (PPC) or Little-Endian (x86) systems. You need to modify this file to define these for your system. Define the constant X86_PLATFORM if you are using an x86 system. Comment out this #define statement if you are using a PPC system.

A simple test program is provided to verify memory mapping of the board:

ADT_L0_test1.c

Use the Tornado/Workbench environment to build the ADT_L0_test1.c program. This program requires the ADT_L0_VxWorks7.c file and the ADT_L0.h file.

When you run the ADT_L0_test1 program it will attempt to map memory to the board and will read and print several words from memory. The “Board-Level Align Check Word” value should be **0x12345678**. If you do not see this value then there is a memory mapping problem or there is a byte/word swapping problem. If you see swapped bytes/words the byte/word swapping macros in ADT_L0_VxWorks.c are not defined correctly for your system (see note above). If you see other unexpected values (or if the test program crashes or otherwise fails to run) then the problem may be incorrect physical or virtual memory settings.

Building and Testing the L1 API and Example Programs

After you have verified that the Layer 0 API is functioning properly you can then add the Layer 1 API and Layer 1 example programs.

The Layer 0 API consists of the following files:

ADT_L0.h
ADT_L0_VxWorks7.c

The Layer 1 API consists of the following files:

ADT_L1.h
ADT_L1_General.c
ADT_L1_BoardGlobal.c
ADT_L1_MemMgmt.c
ADT_L1_BIT.c
ADT_L1_INT.c
ADT_L1_1553_General.c
ADT_L1_1553_INT.c
ADT_L1_1553_RT.c
ADT_L1_1553_BC.c
ADT_L1_1553_BM.c
ADT_L1_1553_PB.c
ADT_L1_1553_SG.c
ADT_L1_A429_General.c
ADT_L1_A429_INT.c
ADT_L1_A429_RX.c
ADT_L1_A429_TX.c
ADT_L1_A429_RX_MC.c
ADT_L1_A429_PB.c
ADT_L1_A429_SG.c

In addition to the API files listed above, you will need one of the Layer 1 example programs. A good initial test (for 1553 boards) is to start with a simple Remote Terminal program:

ADT_L1_1553_ex_rt1.c

A good initial test (for A429 boards) is to start with a simple transmit/receive program:

ADT_L1_A429_ex_rxtx1.c

Note that the “standard” example programs use “main()” for the main program entry point. VxWorks applications typically use a main function name other than “main()”, so when moving the standard example programs to VxWorks you should change “main()” to something else (like “test_rt1()”, for example).

You will need to edit the #define for the DEVID in the example program to match the type of Alta board you are using. For example, the following specifies the Alta PMC-1553 board type:

```
#define DEVID (ADT_PRODUCT_PMC1553 | ADT_DEVID_BOARDNUM_01 |  
ADT_DEVID_CHANNELTYPE_1553 |ADT_DEVID_CHANNELNUM_01)
```

The example program can now be compiled with the API functions to generate an executable program for the VxWorks system.

If you see **error 3** on initialization, this indicates that the specified device could not be found in the system. The usual reason for this is failure to edit the example program to specify the appropriate product type in the Device ID (DEVID). As mentioned above:

NOTE: You **MUST** edit the example program to change the #define for DEVID to match your product type (PCI1553, PMC1553, PC104P1553, etc.).

When you have the example programs running successfully on your system, you can then expand on the example code for your own applications.

Appendix E – LynxOS

This appendix discusses installation and operation under LynuxWorks LynxOS operating systems.

NOTE:

If you are only using ENET devices you can use the ENET-ONLY BSD API, discussed in **Appendix A**. This API does not use a kernel-level driver and can be used on almost all platforms. You cannot use the ENET-ONLY BSD API with PCI/PCIe devices

Supported LynxOS Versions

Alta currently supports **LynxOS 4.0**. This has been tested on an ORION Technologies CPC7525 IBM PowerPC 750GX with an Alta PMC-1553 board. The driver currently only supports one board.

You may have to modify the driver or Layer 0 API functions to be compatible with other LynxOS systems.

Supported Alta Products

The PMC-1553 is the only Alta product supported for LynxOS.

Supported C Compilers

Alta uses the GNU C compiler (gcc). We do not provide any specific support for other compilers.

Hardware Installation

Shut down the system and install the Alta board. Boot LynxOS on your processor board, login to your system as root, on the shell prompt type:

```
user name: root  
bash-2.02# drm_stat
```

This will display information on all the PCI devices in the system. Look for the Alta Vendor ID (0xAD00) to identify any Alta boards. Make sure the BAR0 and BAR2 have valid (non-zero) virtual (Vaddr) and physical (Paddr) addresses for the board.

Software Installation

Copy the file “**Alta_LynxOS.tar**” from the LynxOs directory on the installation CD to an appropriate directory on your system. Extract this tar file – this will create a directory with the LynxOS driver, Layer 0 API files, the Layer 1 API files, and the Layer 1 example programs. This will contain five directories – alta_1553_examples, drivers.bsp, devices.bsp, dheaders, cfg.ppc

System Configuration

We use a static driver installation so that the driver is automatically installed when the system boots-up. There are four system directories where files need to be added and the driver and device files built into the kernel.

1. Change directory to the sys directory where LynxOS was installed. For example /opt/lynxos/4.0.0/ppc/sys
2. The following directories are BSP/platform specific and may be different on your system.
 - a. Change directory to /opt/lynxos/4.0.0/ppc/sys/drivers.xxx (for example, /drivers.oti_cpc7525)
 - b. Copy the alta_pmc_1553 directory from the Alta drivers.bsp directory into this directory (/drivers.xxx).
 - c. Change directory to /opt/lynxos/4.0.0/ppc/sys/devices.xxx.
 - d. Copy all files from the Alta devices.bsp directory.
 - e. Change directory to /opt/lynxos/4.0.0/ppc/sys/dheaders
 - f. Copy all files from the Alta dheaders directory.
 - g. Change directory to /opt/lynxos/4.0.0/ppc/sys/cfg.ppc
 - h. Copy all files from the Alta cfg.ppc directory.
3. Build the driver
 - a. Change directory to /opt/lynxos/4.0.0/ppc/sys
 - b. Setup the build environment. For example, type
bash-2.02# ./SETUP.bash
 - c. Change directory to /opt/lynxos/4.0.0/ppc/sys/drivers.xxx/alta_pmc_1553
 - d. Type “make” to build the driver.
 - e. Change directory to /opt/lynxos/4.0.0/ppc/sys/devices.xxx
 - f. Edit the Makefile in this directory and add the following: alta_info.x to the end of the line for DEVICE_FILES_all. Save the Makefile, type “make” to build the device tree.
4. Build the Kernel, statically link the driver, create /dev node
 - a. Change directory to opt/lynxos/4.0.0/ppc/sys/lynx.os (this should put you into the kernel BSP directory).
 - b. Edit the CONFIG.TBL file. At the bottom of the file, add
l:alta_pmc.cfg
 - c. Rebuild the LynxOS kernel.
 - d. Now the new kernel is ready to be loaded and the driver will be installed and ready to use after boot-up.
 - e. After booting the new kernel, login as root and at the shell prompt, type in “devices”. You should see the device name “alta_pmc”. Type in “drivers” and you should also see the driver name “alta_pmc”.

Building and Testing the L0 and L1 API and Example Programs

After you have verified that the Layer 0 API is functioning properly you can then add the Layer 1 API and Layer 1 example programs.

The Layer 0 API consists of the following files:

ADT_L0.h
ADT_L0_LynxOS.c

The Layer 1 API consists of the following files:

ADT_L1.h
ADT_L1_General.c
ADT_L1_BoardGlobal.c
ADT_L1_MemMgmt.c
ADT_L1_BIT.c
ADT_L1_INT.c
ADT_L1_1553_General.c
ADT_L1_1553_INT.c
ADT_L1_1553_RT.c
ADT_L1_1553_BC.c
ADT_L1_1553_BM.c
ADT_L1_1553_PB.c
ADT_L1_1553_SG.c

In addition to the API files listed above, you will need one of the Layer 1 example programs. A good initial test is to start with a simple Remote Terminal program:

ADT_L1_1553_ex_rt1.c

The example program can now be compiled with the API functions to generate an executable program for the LynxOS system. A sample makefile (make_rt1) is provided to build the ADT_L1_1553_ex_rt1.c program with the API. This generates an executable that will setup the Alta board as RT 1. Use another device (such as AltaView running on a different board/channel) as the BC and BM to communicate with the RT on the 1553 bus.

You can now try other example programs and develop your own applications for the Alta 1553 board.

Note on Interrupts

The LynxOS driver and L0 API currently implements interrupt handling for 1553 Channel 1 only. The driver provides an ioctl command (0x70) to wait for interrupt. The L0 API provides a corresponding function to wait for interrupt – this function does not return (blocks) until the interrupt is received. The LynxOS L0 API does not implement

the attach/detach interrupt handler functions and does not do a “call-back” to the users interrupt handler function. An interrupt-driven application would setup the board then go into a loop where it waits for interrupt, processes interrupts (call user interrupt handler function), and repeats.

Appendix F – GHS INTEGRITY

This appendix discusses installation and operation under Green Hills Software INTEGRITY operating systems.

NOTE:

If you are only using ENET devices you can use the ENET-ONLY BSD API, discussed in **Appendix A**. This API does not use a kernel-level driver and can be used on almost all platforms. You cannot use the ENET-ONLY BSD API with PCI/PCIe devices

Supported INTEGRITY Versions

Alta currently supports **INTEGRITY 5.0.10**. This has been tested on a SBS CK5 SBC with an Alta PMC-1553 board. The driver has also been tested with an Alta PMC-A429 board.

You may have to modify the Layer 0 API functions to be compatible with other INTEGRITY systems, platforms, and/or Board Support Packages.

Supported Alta Products

The PMC-1553 and the PMC-A429 are the only Alta products supported for INTEGRITY. Contact Alta if you need to use other products with this environment.

Supported Development Environments

Alta used the **MULTI** development environment (version 4.2.4) to build for INTEGRITY targets.

Hardware Installation

Shut down the system and install the Alta board. You may use the **pci busxxx** command at BIOS startup to verify proper hardware installation and find the bus that the device is installed on. The Alta PCI Vendor ID is **0xAD00**.

Kernel/Driver Configuration

Add the AltaDT_Driver.c to the Board Support Packages (BSP) and build the BSP library. If you must make any changes to the driver, rebuild after modifying the file.

Each Alta board requires 512 Bytes of PCI BAR0 address space and 8MB of PCI BAR2 address space. Note that the addresses and memory size must be aligned to the page size – if the page size is larger than 512 bytes then reserve the page size for BAR0.

Software Installation

Copy the directory “**GHS_INTEGRITY**” from the GHS_Integrity directory on the installation CD to an appropriate directory on your system (where the Multi development

environment will be used to build code for your INTEGRITY target). This directory contains the INTEGRITY Layer 0 API files, the Layer 1 API files, the Layer 1 example programs, and example MULTI projects.

Testing the Installation and L0 API

The Layer 0 API is the fundamental interface to the board for memory mapping and interrupt handling. In MULTI you can access the Layer 0 API test, which consists of the following INTEGRITY project located in:

..../GHS_Integrity/L0 API/ADT_Test_Projects/L0_ex_test1/default.gpj

When/default.gpj is running correctly INTEGRITY will attempt to map memory to the board and will read and print several words from memory. The “Board-Level Align Check Word” value should be **0x12345678**. If the numerals are out of order then there is a byte/word swapping problem. If you see other unexpected values (or if the test program crashes or otherwise fails to run) then the problem may be incorrect physical or virtual memory settings.

Building and Testing the L1 API and Example Programs

After you have verified that the Layer 0 API is functioning properly you can then add the Layer 1 API and Layer 1 example programs.

The Layer 0 API consists of the following files:

ADT_L0.h
ADT_L0_INTEGRITY.c

The Layer 1 API consists of the following files:

MIL-STD-1553
ADT_L1_1553_General.c
ADT_L1_1553_INT.c
ADT_L1_1553_BC.c
ADT_L1_1553_RT.c
ADT_L1_1553_BM.c
ADT_L1_1553_PB.c
ADT_L1_1553_SG.c

ARINC

ADT_L1_A429_General.c
ADT_L1_A429_INT.c
ADT_L1_A429_TX.c
ADT_L1_A429_RX.c
ADT_L1_A429_RX_MC.c
ADT_L1_A429_PB.c
ADT_L1_A429_SG.c

General

ADT_L1.h
ADT_L1_BoardGlobal.c
ADT_L1_BIT.c
ADT_L1_General.c
ADT_L1_INT.c
ADT_L1_MemMgmt.c

In addition to the API files listed above, you will need one of the Layer 1 example programs. For 1553, a good initial test is to start with a simple Remote Terminal program:

ADT_L1_1553_ex_rt1.c

For A429, a good initial test is to start with a simple RX/TX program:

ADT_L1_A429_ex_rxtx1.c

NOTE: ADT_L1_1553_ex_rt1.c, and ADT_L1_A429_ex_rxtx1.c are already built in to an example projects:

..../GHS_Integrity/L1 API/ADT_Test_Projects/M1553/L1_1553_ex_rt1/default.gpj
and

..../GHS_Integrity/L1 API/ADT_Test_Projects/A429/L1_A429_ex_rxtx1/default.gpj

If you are using ADT_L1_1553_ex_rt1.c to test a 1553 device, use another device (such as AltaView running on a different board/channel) as the BC and BM to communicate with the RT on the 1553 bus.

If you are using ADT_L1_A429_ex_rxtx1.c to test an A429 device, this program will send labels on TX channel 1 and receive them on RX channel 1 so it can talk to itself without an external connection to another device.

Once you have verified that your installation is working with one of these example programs you can then run other example programs and develop your own applications using Alta API.

NOTE: The current Alta driver for GHS INTEGRITY does not support hardware interrupts. Interrupt functionality can be used through software polled interrupts.

Appendix G – Solaris

Alta uses the Jungo WinDriver tools (version 9.01) for Solaris device drivers.

NOTE:

If you are only using ENET devices you can use the ENET-ONLY BSD API, discussed in **Appendix A**. This API does not use a kernel-level driver and can be used on almost all platforms. You cannot use the ENET-ONLY BSD API with PCI/PCIe devices

Supported Environments

Alta supports Solaris on **Sun SPARC 64-bit systems** and **Intel x86 32-bit systems**.

Alta does not support Solaris on Intel x86 64-bit systems. Alta does not at present provide specific support for other development environments. The Alta API was tested on the following platforms.

SPARC 64-bit Platform: **Sun SPARC Ultra 60** workstation (PCI only) running **Solaris 10 (Sun OS 5.10)** using the **GNU gcc** compiler version 3.4.6.

SPARC 64-bit Platform: **Sun SPARC Ultra 25** workstation (PCI and PCI Express) running **Solaris 10 (Sun OS 5.10)** using the **GNU gcc** compiler version 3.4.6. **The Sun M-series (M3000, M4000, M5000) systems ARE NOT COMPATIBLE WITH ALTA PRODUCTS AND ARE NOT SUPPORTED BY ALTA.**

Intel x86 32-bit Platform: **x86 32-bit system** (PCI and PCI Express) running **Solaris 10 (Sun OS 5.10)** using the **GNU gcc** compiler version 3.4.6.

This appendix contains information for installing and running on SPARC 64-bit systems followed by information for installing and running on Intel x86 32-bit systems.

Supported Alta Products

The following Alta products have been tested with Solaris:

PMC-1553, PCI-1553, PCIE1L-1553

Contact Alta if you need to use products not listed here.

Installation on SPARC 64-bit systems

The 64-bit driver only supports 64-bit applications. This driver does not support 32-bit applications running on 64-bit Solaris.

Shutdown the system and install the **Alta** board. Power the system up. Copy the file **ADT_Solaris_10_64.x.x.x.x.tar** (where x.x.x.x is the current release version) from the **Solaris_10** directory on the **Alta** installation CD to an appropriate directory on your system. Extract the source files with the following command :

```
[YourDirectory]# tar -xvf ADT_Solaris_10_64.x.x.x.x.tar
```

This will create a directory called **ADT_API_X.X.X.X** containing three subdirectories: **source**, **Driver**, and **examples**. The **Source** directory contains the Layer 1 **AltaAPI** source files along with sample make files to build them. It also contains the **ADT_L0_Sol64.so.x.x.x.x** Layer 0 shared library which provides access to the kernel level driver.

The **Driver** directory contains the kernel level driver files. Of interest to the user are the script files **install_altadriver** and **remove_altadriver** which are used to install and uninstall the kernel level driver module **altadriver**.

The **Examples** directory contains various M1553 and A429 API example programs. See the readme file in the examples folder for more information on the individual examples.

Installing the Driver on SPARC 64-bit systems

Installation of altadriver should be performed by the system administrator logged in as root, or with root privileges, since the altadriver installation process includes installation of the kernel driver module **altadriver**.

If not logged in as root enter superuser mode before running the driver installation script:

```
[YourDirectory/Driver]# su
```

Run the **install_altadriver** installation script, change group and user IDs, and set permissions as described below.

```
[YourDirectory/Driver]# exit
```

Go to the driver installation directory:

```
[YourDirectory]# cd /Driver/altadriver_installation
```

Install the WinDriver kernel level driver, **altadriver**, by running the **install_altadriver** installation script. For PCI devices, specify the vendor and device IDs of your PCI device in the installation command (where <vid> represents the device's vendor ID and <did> represents the device's device ID):

```
[YourDirectory/Driver/altadriver_installation]# ./install_altadriver <vid>,<did>
```

For example, Alta's Vendor ID is AD00 and the Alta PMC-1553 board device ID is 10 (Alta PCI-1553 device ID is 14, PCIE1L-1553 is 20). You can use the "prtconf" command to list the devices in the system. Alta boards should show up under one of the PCI nodes:

```
pci, instance #3  
    pciad00,14, instance #0 (driver not attached)
```

or

```
pci, instance #3  
    pcied00,20, instance #0 (driver not attached)
```

If your board shows up in the system as "pciad00,<did>" then use the following syntax with the ./install_altadriver command (in this example we use a device ID of 14 for a PCI-1553 board, if you are using a different board type change this to the appropriate device ID value):

```
[YourDirectory/Driver/altadriver_installation]# ./install_altadriver ad00,14
```

If your board shows up in the system as "pcied00,<did>" then use the following syntax with the ./install_altadriver command (in this example we use a device ID of 20 for a PCIE1L-1553 board, if you are using a different board type change this to the appropriate device ID value):

```
[YourDirectory/Driver/altadriver_installation]# ./install_altadriver exad00,20
```

You can now run the "prtconf" command again and the output for the board should no longer say "(driver not attached)".

```
pci, instance #3  
    pcied00,20, instance #0
```

Caution: Since **altadriver** gives direct hardware access to user programs, it may compromise kernel stability on multi-user Solaris systems. Please restrict access to trusted user. Change the user and group IDs and give read/write permissions of the device file, **altadriver**, depending on how you wish to allow users to access hardware through the device node.

Change user and group IDs as required

Set read/write permissions:

```
[YourDirectory/Driver/altadriver_installation]# chmod 666 /kernel/drv/sparcv9/altadriver
```

Note: The altadriver is generally located in the **/kernel/drv/sparcv9/** directory, but may be located in a different directory on some systems.

To uninstall the WinDriver kernel level driver **altadriver** run the uninstall script **remove_altadriver**. The kernel plugin must be uninstalled prior to uninstalling the driver:

```
[YourDirectory/Driver/altadriver_installation]# ./remove_altadriver
```

Installing the Kernel Plugin on SPARC 64-bit systems

Installation of the Kernel PlugIn should be performed by the system administrator logged in as root, or with root privileges.

Note: AltaDriver must be installed prior to installing the kernel PlugIn. Attempting to install AltaDriver after the PlugIn will result in a system error.

Go to the kernel plugin installation directory:

```
[YourDirectory]$ cd /Driver/kemode
```

Type the following commands to install the kernel plugin:

```
$ cp kp_altadrv.conf /kernel/drv  
$ cp kp_altadrv /kernel/drv/sparcv9  
$ ./wdreg kp_altadrv
```

Note: The kernel plugin is generally located in the **/kernel/drv/sparcv9** directory, but may be located in a different directory on some systems.

To **uninstall** the kernel plugin, type the following commands. The kernel plugin must be uninstalled prior to uninstalling the driver :

```
$ /usr/sbin/rem_drv kp_altadrv  
$ rm /kernel/drv/sparcv9/kp_altadrv  
$ rm /kernel/drv/kp_altadrv.conf
```

Note: The kernel plugin is generally located in the **/kernel/drv/sparcv9** directory, but may be located in a different directory on some systems.

Building and Testing the API and Example Programs on SPARC 64-bit Systems

You can either build in the provided **source** folder or create your own test directory for your working files.

If you create a test directory, copy all the relevant **A429** and/or **M1553** Layer 1 source files along with **ADT_L0.h** and **ADT_L1.h** to your working directory. A makefile (make_rt1) is provided for the **ADT_L1_1553_ex_rt1.c** example program. This make file can be copied and edited to create make files for other example programs or for your own test programs. In this example we will build the example program **ADT_L1_1553_ex_rt1.c** with the API.

NOTE: You should edit the **ADT_L1_1553_ex_rt1.c** example program to change the **#define DEVID** to match your product type (PMC1553, PCI1553, PCIE1L1553, etc.).

The **AltaAPI** Layer 0 module is provided as the shared library **ADT_L0_Sol64.so.x.x.x.x** where .x.x.x.x is the release version number. You can rename and copy this file to your test directory, or alternatively you can create a symbolic link to the file:

```
[Your_test_dir]$ ln -sf ADT_L0_Sol64.so.x.x.x.x ADT_L0_Sol64.so.
```

Set the library load path so the compiler can find the shared library:

```
[Your_test_dir]$ LD_LIBRARY_PATH=/usr/local/lib/sparcv9:/usr/sfw/lib/sparcv9:$LD_LIBRARY_PATH
```

Export the library load path:

```
[Your_test_dir]$ export LD_LIBRARY_PATH
```

You may need to set the PATH as well:

```
[Your_test_dir]$ PATH=/usr/local/bin:/usr/sfw/bin:$PATH
```

Export the path:

```
[Your_test_dir]$ export PATH
```

Note: If you reboot the system the changes to the paths are lost. One way to re-establish this load path on boot-up is to put these commands in the **.bashrc** file or appropriate startup script.

Edit the **ADT_L1_1553_ex_rt1.c** example program. Modify the DEVID definition to match your Alta board type.

Build the API and example program:

```
[ADT_L1_test]$ gmake -f make_rt1
```

If you get errors on gmake, you may have to change the CC and LD path from /usr/local/bin to /usr/sfw/bin.

Run the example program:

```
[ADT_L1_test]$ ./new_ex_rt1
```

If you get ERROR 3 on the device initialization, this indicates that you do not have the correct Device ID (DEVID) for your board type. Edit the example program and ensure that you have the correct DEVID.

If you get ERROR 19 on device initialization, this indicates that the kernel plug-in is not running. Go to the /Driver/kernmode folder and run the “./wdreg kp_altadrv” command again.

Note that the error codes 0-999 are defined in the file ADT_L0.h and the error codes 1000-1999 are defined in the file ADT_L1.h.

You are now ready to run other example programs or begin developing your own application. The example programs provide an excellent starting point for new applications. Feel free to modify the examples and sample makefile to fit your specific needs.

Installation on x86 32-bit Systems

Solaris x86 typically installs for 64-bit operation. The Jungo WinDriver tools only supports 32-bit operation on Solaris x86. You can see your kernel version with the following command:

```
# /usr/bin/isainfo -kv
```

You can switch to 32-bit mode with the following command:

```
# /usr/sbin/eeprom boot-file="kernel/unix"
```

After executing this command you must re-boot the system to load the 32-bit kernel.

Shutdown the system and install the **Alta** board. Power the system up. Copy the file **ADT_Solaris_10_x86_32.x.x.x.x.tar** (where x.x.x.x is the current release version) from the **Solaris_10** directory on the **Alta** installation CD to an appropriate directory on your system. Extract the source files with the following command :

```
[YourDirectory]# tar -xvf ADT_Solaris_10_x86_32.x.x.x.x.tar
```

This will create a directory called **ADT_API_X.X.X.X** containing three subdirectories: **source**, **Driver**, and **examples**. The **source** directory contains the Layer 1 **AltaAPI** source files along with sample make files to build them. It also contains the **ADT_L0_Solx86_32.so.x.x.x.x** Layer 0 shared library which provides access to the kernel level driver.

The **Driver** directory contains the kernel level driver files. Of interest to the user are the script files **install_altadriver** and **remove_altadriver** which are used to install and uninstall the kernel level driver module **altadriver**.

The **examples** folder contains various M1553 and A429 API example programs. See the readme file in the examples folder for more information on the individual examples.

Installing the Kernel Plugin on x86 32-bit Systems

Installation of the Kernel PlugIn should be performed by the system administrator logged in as root, or with root privileges.

Note: AltaDriver must be installed prior to installing the kernel PlugIn. Attempting to install AltaDriver after the PlugIn will result in a system error.

Go to the kernel plugin installation directory:

```
[YourDirectory]$ cd /Driver/kemode/solaris
```

Type the following commands to install the kernel plugin:

```
$ cp kp_altadrv.conf /kernel/drv  
$ cp kp_altadrv /kernel/drv  
$ ./wdreg kp_altadrv
```

Note: The kernel plugin is generally located in the **/kernel/drv/** directory, but may be located in a different directory on some systems.

To **uninstall** the kernel plugin, type the following commands. The kernel plugin must be uninstalled prior to uninstalling the driver:

```
$ /usr/sbin/rem_drv kp_altadrv  
$ rm /kernel/drv/kp_altadrv  
$ rm /kernel/drv/kp_altadrv.conf
```

Note: The kernel plugin is generally located in the **/kernel/drv/** directory, but may be located in a different directory on some systems.

Installing the Driver on x86 32-bit Systems

Installation of altadriver should be performed by the system administrator logged in as root, or with root privileges, since the altadriver installation process includes installation of the kernel driver module **altadriver**.

If not logged in as root enter superuser mode before running the driver installation script:

```
[YourDirectory/Driver]# su
```

Run the **install_altadriver** installation script, change group and user IDs, and set permissions as described below.

```
[YourDirectory/Driver]# exit
```

Go to the driver installation directory:

```
[YourDirectory]# cd /Driver/altadriver_installation
```

Install the WinDriver kernel level driver, **altadriver**, by running the **install_altadriver** installation script. For PCI devices, specify the vendor and device IDs of your PCI device in the installation command (where <vid> represents the device's vendor ID and <did> represents the device's device ID):

```
[YourDirectory/altadriver]# ./install_altadriver <vid>,<did>
```

For example, Alta's Vendor ID is AD00 and the Alta PMC-1553 board device ID is 10 (Alta PCI-1553 device ID is 14, PCIE1L-1553 is 20):

For a PMC-1553:

```
[YourDirectory/ altadriver]# ./install_altadriver ad00,10
```

For a PCI-1553:

```
[YourDirectory/ altadriver]# ./install_altadriver ad00,14
```

For a PCIE1L-1553:

```
[YourDirectory/ altadriver]# ./install_altadriver ad00,20
```

NOTE: Enter the command exactly as shown above.

Caution: Since **altadriver** gives direct hardware access to user programs, it may compromise kernel stability on multi-user Solaris systems. Please restrict access to trusted user. Change the user and group IDs and give read/write permissions of the device file, **altadriver**, depending on how you wish to allow users to access hardware through the device node.

Change user and group IDs as required

Set read/write permissions:

```
[YourDirectory/altadriver]# chmod 666 /kernel/drv/altadriver
```

Note: The altadriver is generally located in the **/kernel/drv/** directory, but may be located in a different directory on some systems.

To uninstall the WinDriver kernel level driver **altadriver** run the uninstall script **remove_altadriver**. The kernel plugin must be uninstalled prior to uninstalling the driver.

```
[YourDirectory/altadriver]# ./remove_altadriver
```

Building and Testing the API and Example Programs on x86 32-bit Systems

Note: On Intel x86 32-bit systems it is highly recommended to use "gmake" instead of the standard "make" delivered with Solaris. The example makefiles assume that

“gmake” is being used for compilation. If not, the user must edit the example makefile for their particular compiler. Alta used **GNU** compiler version 3.4.6 during API testing.

You can either build in the provided **source** folder or create your own test directory for your working files.

If you create a test directory, copy all the relevant **A429** and/or **M1553** Layer 1 source files along with **ADT_L0.h** and **ADT_L1.h** to your working directory. A makefile (make_rt1) is provided for the **ADT_L1_1553_ex_rt1.c** example program. This make file can be copied and edited to create make files for other example programs or for your own test programs. In this example we will build the example program **ADT_L1_1553_ex_rt1.c** with the API.

NOTE: You should edit the **ADT_L1_1553_ex_rt1.c** example program to change the **#define DEVID** to match your product type (PMC1553, PCI1553, PCIE1L1553, etc.).

The **AltaAPI** Layer 0 module is provided as the shared library **ADT_L0_Solx86_32.so.x.x.x.x** where .x.x.x.x is the release version number. You can rename and copy this file to your test directory, or alternatively you can create a symbolic link to the file:

```
[Your_test_dir]$ ln -sf ADT_L0_Solx86_32.so.x.x.x.x ADT_L0_Solx86_32.so.
```

Set your paths so the compiler can find the shared library:

```
[Your_test_dir]$ PATH=/usr/sfw/bin:$PATH  
[Your_test_dir]$ export PATH  
[Your_test_dir]$ LD_LIBRARY_PATH=/usr/sfw/lib:$LD_LIBRARY_PATH  
[Your_test_dir]$ export LD_LIBRARY_PATH
```

Note: If you reboot the system this change to the path is lost. One way to re-establish this load path on boot-up is to put the export command in the **.bashrc** file or the appropriate startup script.

Build the API and example program:

```
[ADT_L1_test]$ gmake -f make_rt1
```

Run the example program:

```
[ADT_L1_test]$ ./new_ex_rt1
```

If you get ERROR 3 on the device initialization, this indicates that you do not have the correct Device ID (DEVID) for your board type. Edit the example program and ensure that you have the correct DEVID.

If you get ERROR 19 on device initialization, this indicates that the kernel plug-in is not running. Go to the /Driver/kernmode/solaris folder and run the “./wdreg kp_altadrv” command again.

Note that the error codes 0-999 are defined in the file ADT_L0.h and the error codes 1000-1999 are defined in the file ADT_L1.h.

You are now ready to run other example programs or begin developing your own application. The example programs provide an excellent starting point for new applications. Feel free to modify the examples and sample makefile to fit your specific needs.

Manual Revision Information

Date	Rev	Description
11/14/08	E	Added Solaris Appendix F.
12/09/08	F	Image and Text Clean-up.
2/19/09	F1	Added following functions: -BC and RT CDPRead/WriteWords Functions. -ADT_L1_1553_INT_IQ_ReadRawEntry -ADT_L1_1553_INT_IQ_ReadNewRawEntries -ADT_L1_A429_INT_IQ_ReadRawEntry -ADT_L1_A429_INT_IQ_ReadNewRawEntries
3/11/09	F2	-Image and Text Clean-up. -Added reference to LabVIEW™ ADT_L1_NET20LV.dll (new dll for LabVIEW users) with new UINT Classes. -Added mask_value parameter to ADT_L1_1553_SC_ArmTrigger()
8/16/09	F3	-Changed Address -Added BCCB Read/WriteWords, IRIG Cal, Interval Timer -Updated Windows Install Appendix -Minor typo fixes
9/15/09	F4	-General formatting and clean-up. Fixed page numbering problem. -Updated LabVIEW information in the Windows Appendix A -Added information on IntervalZero RTX in the Windows App A -Corrected errors in the Linux Appendix B
10/11/09	F5	-Fixed a few minor problems in the Linux Appendix B -Added clarification of product ID, rev, and serial number for ADT_L1_GetBoardInfo. -Added clarification on Init_ExtendedOptions functions. -Added paragraph on 1553 Interval Timer in discussion of 1553 interrupt functions.
11/5/09	F6	-Added warning that PCCARD products are not supported for Windows 2000. -Added section on LabWindows/CVI in Appendix A. -Added clarification on supported Linux systems in Appendix B. -Updated VxWorks documentation in Appendix C.
11/13/09	F7	Major Rewrite of the Layer 1 1553 and ARINC Intro Sections.

		Added more overview and examples. Other areas are just minor reorganization and information changes.
2/9/10	G	Updates for v2.3.0.0 (Appendices A, B and F, Windows, Linux, Solaris).
3/8/10	G1	Corrected description of ADT_L1_A429_SG_AddVectors. Added warning about thread-safety for the function ADT_L1_1553_BM_ReadNewMsgsDMA.
5/31/10	G2	Added description to ADT_L1_A429_TX_Channel_SendLabel explaining that this function does NOT block until the Label has finished transmission. Added a section for Solaris x86 32-bit support. Added clarification on MSVS project setup in Appendix A.
8/3/10	G3	Added new ARINC TXP diagram Layer 1 API section. Added description to ADT_L1_A429_TX_Channel_TXPWrite function.
9/16/10	G4	Updates for v2.4.0.0. General cleanup and reformatting. Minor cleanup in discussion of Windows .NET support. Added information on 64-bit settings for creating a MSVS project to build the example programs. Updated Appendix B (Linux), now supports kernels up through 2.6.35 with WinDriver version 10.21.
11/3/10	G5	Added the functions ADT_L1_A429_IntervalTimerGet and ADT_L1_A429_IntervalTimerSet. Modified section on L1_NET20 in Appendix A – now supports hardware interrupts, added more constants, added .NET example programs.
2/18/11	G6	Added 1553 Branch “Address Only” discussion and ADT_L1_1553_BC_CB_SetAddressBranchValue function. Added new ARINC Aperiodic IsRunning and SendAperiodic functions. Added general 1553 Branching and NOP discussion. Added Examples for new 1553 branch and changed ARINC rtx4.c example for new ARINC aperiodic TXCB method. Clarified installation on 64-bit Windows in Appendix A. Added information on ARINC TXCB/TXP Allocation in Layer 1 API Section for ARINC TX. Updated the CDP data structure diagram. Added constants to support MPCIE-1553. Corrections to Appendix F (Solaris). Added warning in ADT_L1_Global_ReadIrigTime function about the firmware registers being 1 second off, software corrects for this. Added the functions ADT_L1_1553_IrigLatchedTimeGet and ADT_L1_A429_IrigLatchedTimeGet. Added a discussion on using IRIG time. Added a section on ENET Device Programming. Added a section on Supported Alta Products to each of the appendices for supported operating systems. Modified Appendix B (Linux) to account for changes where example programs are now built in the /Test folder (using the Layer 1 shared object) rather than in the /Source folder (using the Layer 1 source files).

5/9/11	G7	Minor clean-up of A429 programming discussion. Added “CSR” and “SW” to list of acronyms. Minor corrections to Appendix F, Solaris. Minor correction to discussion of 1553 BC retry operation. Added a note explaining that reading CDP buffers while the firmware is processing that buffer results in a CDP Status Word of 0xFFFFFFFF.
7/5/11	G8	Corrected A429 Signal Capture voltage information. Added note in Solaris appendix stating that Alta does not support the Sun M-series systems. Corrected third parameter in description of ADT_L1_A429_TX_Channel_AperiodicSend. Added ADT_L1_A429_TX_Channel_SendLabelBlock. Modified Appendix B (Linux) to add “lib” prefix to the shared object files for L0 and L1.
10/31/11	G9	Added constants for PCIE1L-A429, MPCIE-A429, PMCE-1553, and ENET-A429. Added clarification in description of the function ADT_L1_A429_TX_Channel_CB_Write.
1/5/12	H1	Added clarification to the description of the functions ADT_L1_1553_RT_MC_LegalizationRead and ADT_L1_1553_RT_MC_LegalizationWrite. Expanded on Intro ENET discussion and ADCP/APMP Overview.
3/7/12	H2	Added L0 error code ADT_ERR_ENET_BADPRODUCTID. Cleaned up notes on using the NOMEMTEST initialization option. In Appendix B (Linux), added note that the SGI UV systems are not supported. Corrections in Appendix F (Solaris). Added BookMark Discussion: ENET-1553 APMP CDP Format: PE + IRIG or Interval Timer Inserts.
3/15/12	H3	Appendix A (Windows) and Appendix B (Linux) – added note on potential problem when using MSI interrupts with two separate applications using interrupts on two channels of the same PCIe board. In Appendix B (Linux) – added note on Ubuntu Linux 11.10 64-bit.
3/26/12	H4	Cleaned up typo in one of the 1553 RT code examples.
9/17/12	H5	General cleanup of minor typos. Fixed typo in one of the A429 code examples. Corrected typos in function descriptions for ADT_L1_ENET_SetIpAddr and ADT_L1_ENET_GetIpAddr. Added the function ADT_L1_Error_to_String. Added clarification to the discussion of BIT operation. Updated VxWorks appendix. Updated GHS INTEGRITY appendix. Added clarification of the function ADT_L1_1553_RT_StatusWrite – cannot change RT Address or ME bit.

2/5/13	H6	Added clarification on the function ADT_L1_1553_RT_GetExternalRTAddr. Added discussion of ARINC banks and channels. Added clarification to discussion of ARINC TX programming (add faster labels to the TX list before slower labels). Added clarification on IRIG YEAR field. Updated Appendix B (Linux). Updated Appendix A (Windows) with note on running 32-bit applications on 64-bit Windows.
2/6/13	H7	Updated NAICS Code.
7/18/13	H8	Added explanation of RT handling of BROADCAST messages for Single-RT and Multiple-RT modes. Updated the ENET Protocols Section. Corrected parameter description of the RT SA/MC CDP ReadWords and WriteWords functions. Corrected parameter description for ADT_L1_GetBoardInfo. Added option bits for ADT_L1_1553_BM_Config. Updated discussion of ENET Path Delay.
11/7/13	H9	Minor correction to 1553 CDP diagram. Added ENET devices to list of tested products in VxWorks, Appendix C. Removed requirement in Solaris Appendix F to run wdreg after reboot for kernel plugin. Added constants for PC104P-MA4.
3/20/14	I1	Updated Appendix B (Linux). Added note on 1553 playback.
4/24/14	I2	Added “_pcilInfo” functions.
9/17/14	I3	Added thread-safety warning on A429 TX SendLabel and SendLabelBlock functions. Appendix D – added section on uninstalling the driver/plug-in (for Linux).
11/10/14	I4	Updated RTX section. Cleaned up minor grammatical errors.
2/25/15	I5	Updated Appendix B (Linux), for Jungo v11.7.0 driver. Updated discussion of ENET path delay. Updated section on IRIG time. Clarified int info word for TXPs. Updated discussion of Device ID to include Product Type. Added warning on SIM devices. Added note on Windows power settings. For ReadDeviceMem32 and WriteDeviceMem32 specified max count of 360 words for ENET devices.
5/14/15	I6	Updated Appendix A (Windows) with discussion on uninstalling and reinstalling to use a different software/driver version.

9/21/15	I7	<p>Updated Appendix A (Windows) for C# projects – MSVS 2010, 2012, 2013.</p> <p>Added new Appendix A to cover the ENET-ONLY BSD API and bumped all the other appendices up by one letter.</p> <p>Added new DMA functions to read A429 RXPs.</p>
6/30/16	I8	<p>Miscellaneous cleanup and clarification. Added “GetAddr” functions. Added note on 1553 BC Aperiodic messages/lists.</p> <p>Updated section on BIT operation. Added note on BC frame start vs schedule timing.</p>
9/28/16	I9	<p>Updated Appendix C (Linux) to provide guidelines for building 32-bit applications on 64-bit Linux. Added references to ENET-A429P where applicable.</p>
1/30/17	J1	<p>Minor return parameter additions related to UNSUPPORTED_BACKPLANE.</p>
9/14/17	J2	<p>Updated manual revision number to correlate with API version</p>
10/13/17	J3	<p>Updated manual revision number to correlate with API version.</p> <p>Changed descriptions of ADT_L1_1553_IrigLatchedTimeGet and ADT_L1_A429_IrigLatchedTimeGet to reflect implementation of checking for IRIG Lock before getting latched times.</p>
3/5/18	J4	<p>Updated manual revision number to correlate with API version</p>
8/27/18	J6	<p>Updated to reflect new products PMC-A429HD and TBOLT-MA4.</p>
12/2/18	J7	<p>Added info for missing ADT_L0_ReadMem32DMA().</p> <p>Added information on using non-contiguous CDP buffers for BCs and RTs. Updated Appendix D to include VxWorks 7.X information.</p>