



FireTrac Advanced 1394 I/O Solution

Operation and API Reference Manual

Doc #: DT-PRO130MAN2110E

Version 2.1.10

Project: firestackmil1394

Operating System: vxworks

Copyright(C) 2008-2020 by DapTechnology B.V. All rights reserved.

This text contains proprietary, confidential information of DapTechnology B.V., is distributed by under license from DapTechnology B.V., and may be used, copied and/or disclosed only pursuant to the terms of the FireStack End-User License Agreement (EULA).

This copyright notice must be retained as part of this text at all times.

Worldwide technical support and product information:

<http://www.daptechnology.com>
support@daptechnology.com

DapTechnology Headquarters
Beatrixstraat 4
7573AA Oldenzaal
The Netherlands
Phone: +31 541 532941

Dap USA Headquarters
708 West San Angelo Street
Gilbert, Arizona 85233
TollFree: (800) 863-0989
Phone: +1 (480) 422-1551
Fax: +1 (302) 439-3947

Table of Contents

Chapter 1. FireTrac Product Family	11
1.1 Network Simulation.....	12
1.2 Simulating AS5643 CCs and RNs.....	14
1.3 FireTrac4424bT.....	15
Software Requirements	15
Architecture	16
Dip Switches	18
Fall-back firmware	18
1.4 FireTrac3460bT.....	19
Software Requirements	19
Architecture	20
Firmware	21
Connector pinning	22
Jumpers	23
Fall-back firmware	25
1.5 Specifications.....	27
Chapter 2. Software Overview	28
2.1 VxWorks.....	28
Introduction	28
Supported Platforms	29
Device specific notes	30
Platform specific notes	30
Configuration Flags	31
Upgrading Firmware	32
License Management	33
Chapter 3. Definition of Terms	35
Chapter 4. Document Conventions	36
Chapter 5. Parameter Naming Conventions	37
Chapter 6. API Revision History	38
6.1 Changes in 2.1.x series.....	39
6.2 Changes in 2.0.x series.....	40
6.3 Changes in 1.0.x and 0.98.x series.....	41
Chapter 7. General Structures and Definitions	42
7.1 Type Definitions.....	42
Basic Types	42
Special Types	42
7.2 Structure Definitions.....	43
FXInt64	43
FXUInt64	43
FXAddress64	43
FXChannelMask	43

FXSetting	43
7.3 Constants.....	45
Speed Codes	45
Transaction Types	45

Chapter 8. Administrative Functions 46

8.1 Initialization.....	47
Functions	47
fxInitialize	47
fxTerminate.....	47
fxAddLicenseCertificate.....	47
8.2 Bus Initialization.....	49
Functions	49
fxGetNumberOfBuses.....	49
fxGetBusInfoList.....	49
fxCreateBusHandle.....	49
fxCloseBusHandle.....	50
fxGetEUI64.....	51
Structures	51
FXBusInfo.....	51
Settings	52
Features	52
Demo Mode.....	53
Stack Size.....	53
Thread Priorities.....	53
Byte Order.....	54
8.3 Memory Management.....	55
Functions	55
fxMemAlloc.....	55
fxMemFree.....	55
8.4 General.....	57
Functions	57
fxGetLibraryVersion.....	57
Structures	57
FXVersionInfo.....	57
8.5 Error Handling.....	58
Functions	58
fxGetErrorMessage.....	58
fxSetErrorCallback.....	58
fxGetErrorStatus.....	59
Constants	59
Error Codes.....	59
Type Definitions	59
FXErrorCallback.....	59

Chapter 9. AS5643 Protocol API Reference 61

9.1 AS5643 Frame Timing.....	62
Functions	62
fxMiiSetFrameTimingOptions.....	62
fxMiiSetStofCallback.....	63
fxMiiGetFrameOffsetTime.....	63
fxMiiGetStofTimestamp.....	64
Type Definitions	64
FXMiiStofCallback.....	64

Structures	65
FXMiiFrameTimingOptions	65
FXTimeStamp	66
Constants	66
Error Codes	66
Frame Synchronization Input Modes	67
Frame Synchronization Output Modes	67
Frame Control Flags	68
9.2 AS5643 Reception	69
Settings	69
Resource Usage	69
Functions	69
Channel Selections	69
fxMiiRcvEnableChannels	69
fxMiiRcvDisableChannels	70
fxMiiRcvGetEnabledChannels	70
Message Filters	71
fxMiiRcvAddFilterItem	71
fxMiiRcvRemoveFilterItem	72
fxMiiRcvClearMessageFilter	72
fxMiiRcvGetNumFilterItems	72
fxMiiRcvGetFilterItemList	73
Buffer Control	73
fxMiiRcvAddBuffer	75
fxMiiRcvLinkBuffers	76
fxMiiRcvRemoveBuffer	77
fxMiiRcvBufferStatus	77
Context Control	78
fxMiiRcvCreateContextHandle	78
fxMiiRcvCloseContextHandle	79
fxMiiRcvSetContextOptions	79
fxMiiRcvStartContext	80
fxMiiRcvStopContext	80
fxMiiRcvContextStatus	80
Type Definitions	81
FXMiiRcvContextHandle	81
FXMiiRcvCallback	81
Structures	81
FXMiiRcvFilterItem	81
FXMiiRcvBufferOptions	82
FXMiiRcvBufferStatus	82
FXMiiRcvEventOptions	83
FXMiiRcvContextOptions	83
FXMiiRcvContextStatus	84
Constants	84
Error Codes	84
Buffer Status Extended Staus Bits	84
Data Formats	85
Receive Packet Format	85
9.3 AS5643 Transmission	86
Settings	86
Resource Usage	86
Functions	86
Context Management	86
fxMiiTrmCreateContextHandle	86
fxMiiTrmCreateContextHandleExt	87
fxMiiTrmCloseContextHandle	88

Single Message Mode.....	88
fxMiTrmMessage.....	88
fxMiTrmSplitMessage.....	89
Streaming Messages Mode.....	90
fxMiTrmStrmWriteImmediate.....	90
fxMiTrmStrmWriteMessageList.....	91
fxMiTrmStrmWriteSplitMessageList.....	92
fxMiTrmStrmStart.....	93
fxMiTrmStrmStop.....	94
fxMiTrmStrmClear.....	94
fxMiTrmStrmGetStatus.....	95
Repeating Messages Mode.....	95
fxMiTrmCreateMessageHandle.....	96
fxMiTrmCreateMessageHandleExt.....	96
fxMiTrmCloseMessageHandle.....	97
fxMiTrmSetMessageData.....	98
fxMiTrmSetMessageSplitData.....	98
fxMiTrmSetMessageOptions.....	99
fxMiTrmStartMessage.....	100
fxMiTrmStopMessage.....	100
fxMiTrmGetMessageStatus.....	100
STOF Message Mode.....	101
fxMiTrmSetStofMessageOptions.....	101
fxMiTrmWriteStofMessage.....	101
fxMiTrmStartStofMessage.....	102
fxMiTrmStopStofMessage.....	102
Type Definitions	103
FXMiTrmContextHandle.....	103
FXMiTrmMessageHandle.....	103
FXMiTrmCallback.....	103
Structures	103
FXMiTrmContextOption.....	103
FXMiTrmMessageOption.....	104
FXBuffer.....	104
FXMiTrmMessage.....	104
FXMiTrmSplitMessage.....	105
FXMiTrmStrmStatus.....	105
FXMiTrmMessageStatus.....	105
FXMiStofMessage.....	106
Constants	107
Error Codes.....	107
Context Options.....	107
Context Modes.....	108
Jitter Ranges.....	108
Jitter Directions.....	109
Message Options.....	109
Jitter Modes.....	110
Auto VPC Modes.....	110
User Callback Event Code Bits.....	110
Data Formats	110
AS5643 Regeneration Format.....	110

Chapter 10. 1394 API Reference 112

10.1 Serial Bus Management.....	112
Settings	112
SBM Capabilities.....	112
Functions	112

fxSetBusResetCallback.....	112
fxGetBusGeneration.....	113
fxGetNumberOfNodesOnBus.....	113
fxGetLocalNodeId.....	114
fxGetMaxSpeedToNode.....	114
Type Definitions	115
FXBusResetCallback.....	115
10.2 Inbound Transactions.....	116
Functions	116
Memory Mapping Functions	116
fxMapLocalMemory.....	116
fxMapRequestHandler.....	117
fxClearMemoryMapping.....	118
Local Memory Access Functions.....	118
fxReadLocalMemory.....	118
fxWriteLocalMemory.....	119
fxLockLocalMemory.....	119
Type Definitions	120
FXRequestHandlerCallback.....	120
FXRequestNotificationCallback.....	121
Structures	121
FXTransactionData.....	121
FXMappingOptions.....	122
Constants	122
Error Codes.....	122
Response Codes.....	122
10.3 Outbound Transactions.....	124
Functions	124
fxReadTransaction.....	124
fxWriteTransaction.....	126
fxLockTransaction.....	127
fxClearTransaction.....	128
fxClearAllTransactions.....	128
fxGetTransactionStatus.....	128
fxGetNumTransactions.....	129
fxGetTransactionList.....	129
Type Definitions	130
FXTransactionCompleteCallback.....	130
Structures	130
FXTransactionOptions.....	130
FXTransactionInfo.....	131
Constants	131
Error Codes.....	131
Transaction Status.....	132
10.4 Isochronous Reception.....	133
Feature Inquiry Functions	135
fxIsoRcvGetNumberOfContexts.....	135
Reception Functions	136
Context Control.....	136
fxIsoRcvCreateContextHandle.....	136
fxIsoRcvCloseContextHandle.....	136
fxIsoRcvStartContext.....	137
fxIsoRcvStopContext.....	137
fxIsoRcvContextStatus.....	137
Buffer Control.....	138
fxIsoRcvBufferFillAddBuffer.....	140
fxIsoRcvPktPerBufferAddBuffer.....	141

fxIsoRcvDualBufferAddBuffer	141
fxIsoRcvLinkBuffers	142
fxIsoRcvRemoveBuffer	143
fxIsoRcvRemoveBuffers	143
fxIsoRcvBufferFillBufferStatus	144
fxIsoRcvPktPerBufferBufferStatus	144
fxIsoRcvDualBufferBufferStatus	145
Type Definitions	145
FXIsoRcvContextHandle	145
FXIsoRcvCallback	145
Structures	146
FXIsoRcvOption	146
FXIsoRcvBuffer	146
FXIsoRcvEventOptions	146
FXIsoRcvBufferStatus	147
FXIsoRcvBufferFillBufferStatus	147
FXIsoRcvPktPerBufferBufferStatus	147
FXIsoRcvDualBufferBufferStatus	148
FXIsoRcvContextStatus	148
Constants	149
Buffer Modes	149
Context Options	149
Buffer Options	151
Error Codes	151
Data Formats	151
Buffer-Fill mode Data Formats	152
With Header/Trailer	152
Without Header/Trailer	152
Packet-per-buffer mode and dual-buffer mode Data Formats	153
With Header/Trailer	153
Without Header/Trailer	153
10.5 Low-Level 1394	154
Settings	154
Resource Usage	154
Functions	154
Asynchronous Packet Reception Functions	154
fxAsyRcvWaitSingleRequest	154
fxAsyRcvWaitSingleResponse	155
fxAsyRcvSetPacketCallback	155
Single Packet Transmission Functions	156
fxAsyTrmWriteQuadletRequest	156
fxAsyTrmWriteBlockRequest	157
fxAsyTrmWriteResponse	157
fxAsyTrmReadQuadletRequest	158
fxAsyTrmReadBlockRequest	159
fxAsyTrmReadQuadletResponse	159
fxAsyTrmReadBlockResponse	160
fxAsyTrmLockRequest	161
fxAsyTrmStream	162
fxAsyTrmLockResponse	162
PHY Packets and registers	163
fxReadLocalPhyBaseReg	163
fxReadLocalPhyPageReg	164
fxWriteLocalPhyBaseReg	164
fxWriteLocalPhyPageReg	165
fxReadRemotePhyPageReg	165
fxReadRemotePhyBaseReg	166

fxPhyRemoteCommand.....	166
fxPhySetForceRoot.....	167
fxPhySetGapCount.....	168
fxPingRemoteNode.....	168
fxPhyPacketSetRcvCallback.....	169
fxPhyPacketTrmRaw.....	169
Topology Functions.....	170
fxGetSelfIdData.....	170
fxIssueBusReset.....	170
Type Definitions	171
FXAsyRcvPacketCallback.....	171
FXPhyPacketRcvCallback.....	171
Structures	172
FXAsyRcvPacket.....	172
Constants	172
Error Codes.....	172
PHY Remote Commands.....	172
PHY Confirmation Flags.....	173

Chapter 11. Time Input Device API Reference 174

11.1 Functions	174
fxGetNumberOfTimeInputs.....	174
fxGetTimeInputInfoList.....	174
fxCreateTimeInputHandle.....	175
fxCloseTimeInputHandle.....	175
fxSetTimeInputMode.....	176
fxSetTimeInputCurrentYear.....	176
fxSetTimeInputFreeRunningOffset.....	177
fxGetTimeInputStatus.....	177
fxSetTimeInputStatusCallback.....	177
fxSetTimeInputSecondCallback.....	178
11.2 Type Definitions	179
FXTimeInputHandle.....	179
FXTimeInputStatusCallback.....	179
FXTimeInputSecondCallback.....	179
11.3 Structures	180
FXTimeInputInfo.....	180
FXTimeInputStatus.....	181
11.4 Constants	182
Time Input Mode.....	182
Status Codes.....	183
Error Codes.....	183

Chapter 12. Examples 184

12.1 Inbound Transactions	185
Inbound Transaction Monitor.....	185
12.2 Outbound Transactions	187
Outbound Transaction Demo.....	187
12.3 Low-Level 1394	189
Low-level Demo.....	189
12.4 Isochronous Reception	191
Isochronous Reception Demo.....	191
12.5 Isochronous Transmission	193
Isochronous Transmission Demo.....	193

12.6 AS5643 Reception	195
Mil1394 Data Logger	195
Mil1394 Receive Demo	197
12.7 AS5643 Transmission	201
Mil1394Transmit Demo	201
12.8 AS5643 Cockpit Demo	208
AS5643 Control Computer Example	208
12.9 Extensions	213
External Timer	213

Chapter 13. FireStack Release Notes 215

Chapter 1. FireTrac Product Family

The FireTrac® product family complements DapTechnology's successful FireSpy® and AS5643 OHCI host adapter product lines. It clearly is the next generation SAE AS5643 data processing, simulation and testing solution.

DapTechnology has seen an increasing demand for more streamlined hardware systems for the processing of AS5643 (and generic 1394) data streams. Customers get increasingly involved in monitoring the actual data content rather than the 1394 layer. And for simulation purposes, they require advanced error insertion capabilities that can only be accomplished with non-off-the-shelf Link Layer implementations. IRIG time-stamping of monitored events on the bus is a typical requirement.

FireTrac® is the answer for this market need. It is designed to natively (not just as an add-on protocol) support AS5643. Besides the standard IEEE1394 features, FireTrac® has been architected to provide hardware level support for SAE AS5643 which reduces host processor burden, specifically for packet encapsulation, data extraction, receive/transmit STOF offsets, etc. As a key example, FireTrac® handles AS5643 transmission timing entirely in hardware therefore making it a lot more accurate. It is important to understand that FireTrac® is a dedicated and optimized solution for the processing of AS5643 type traffic. Support for this protocol is embedded in the hardware and not just in a software layer, as is provided with other solutions that rely on COTS OHCI chipsets.

In order to get the best out of the unique feature set of the FireTrac® card DapTechnology recommends using the hardware in combination with FireStack®, i.e. DapTechnology's home grown software stack. FireStack optimally supports the hardware and firmware layers embedded into FireTrac. As FireTrac's® host interface uses FireLink Extended (and not a standard OHCI Link Layer chip) functionality that has been tailored and optimized for the support of the AS5643 standard brings the combination of FireTrac® and FireStack® to an entirely new level.

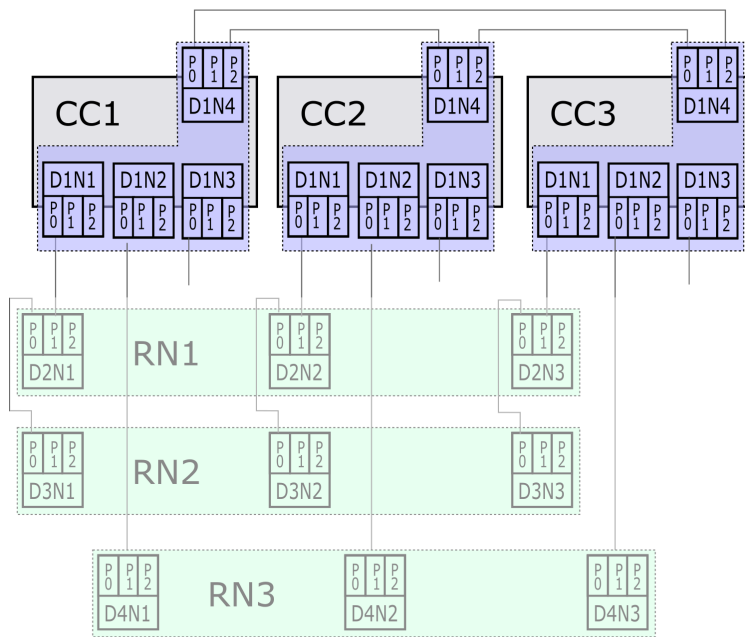
1.1. Network Simulation

The FireTrac® product family offers a large variety of how these I/O cards can be used to start, expand, and grow a simulation environment for AS5643 devices. While architecturally identical, the different channel numbers allow for a vast variety of configurations. Please note that the arguments presented below predominantly address triple redundant network systems, yet can relatively simply be adapted for system redundancies lower or higher than three (3).

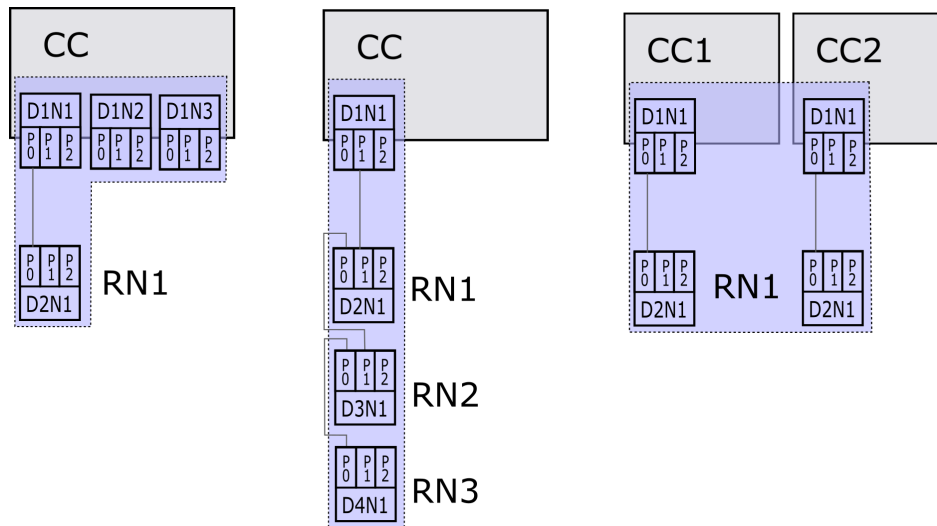
4-node FireTrac®:

The 4-node version(s) of FireTrac® is perfectly suited for use as a CC. As the picture below demonstrates the 4-node architecture is ideal to simulate a 3-branch CC including the interface for the CCDL. Using three FireTrac® cards (blue) a full and triple-redundant arrangement for control computers can be realized.

Paired with a triple redundant RN implementation using 3-node FireTrac® (displayed in green) one can easily build a minimal yet expandable instantiation of an AS5643 network consisting of CCs with CCDL and RNs.

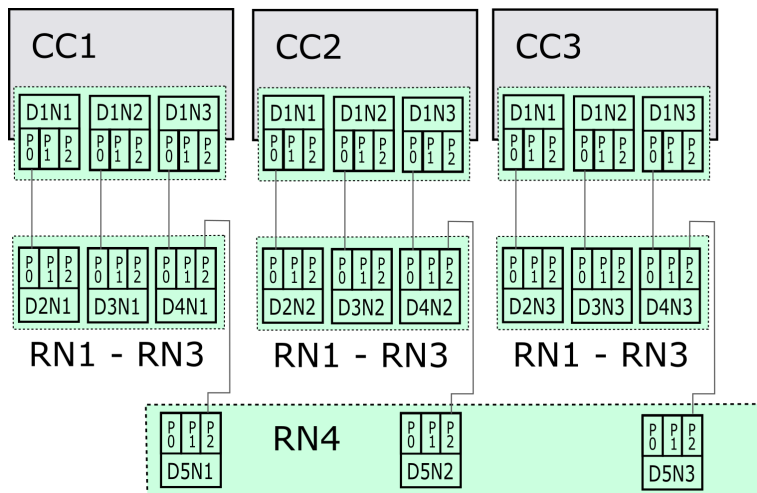


However, the above is only an implementation variant. A 4-node FireTrac® can also be used to simulate single branch bus behavior (left and middle) or mixed CC/RN situation in double redundancy (right).

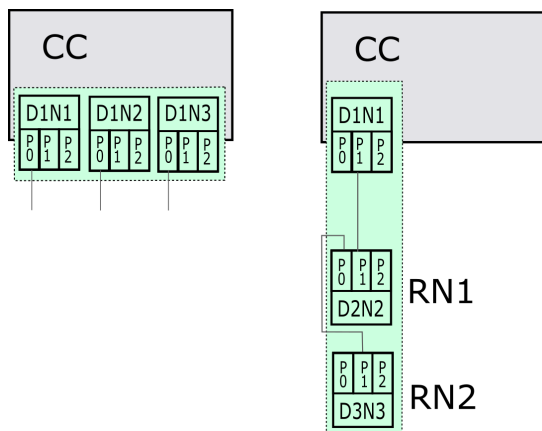


3-node FireTrac®:

The 3-node version(s) of FireTrac® is best suited to address triple-redundant and triple-branch architectures when no CCDL is needed. The picture below demonstrates how such a system can be arranged in different RN configurations by using just a few 3-node FireTrac® cards (green).



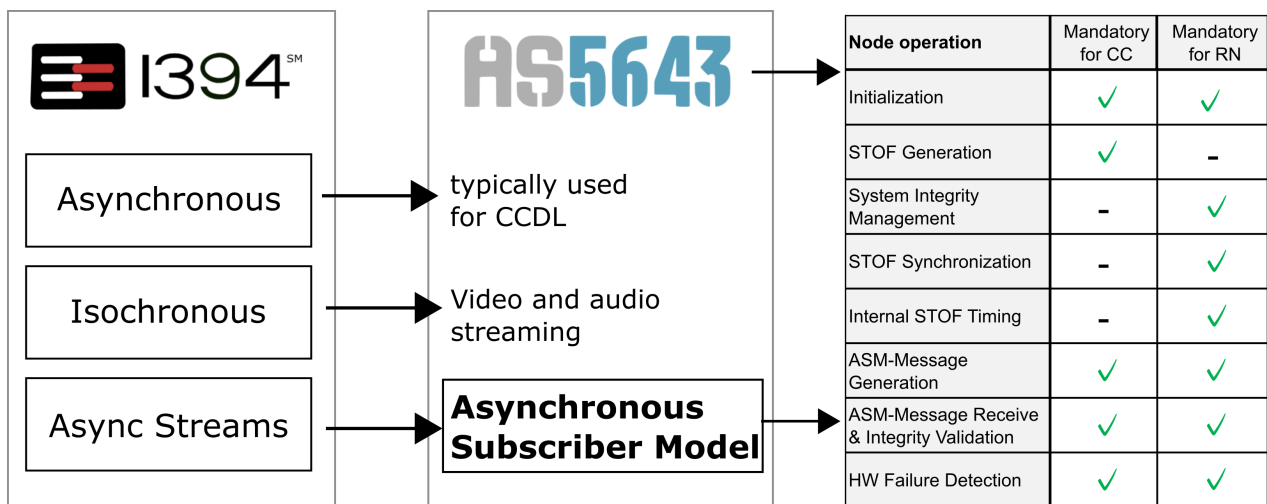
Other variants of how a single 3-node FireTrac® card can be used are depicted below. The left shows a CC devices simulating the triple bus interface (3 CC) whereas the right pictures demonstrates a possible single-branch usages model (CC + 2 RNs).



1.2. Simulating AS5643 CCs and RNs

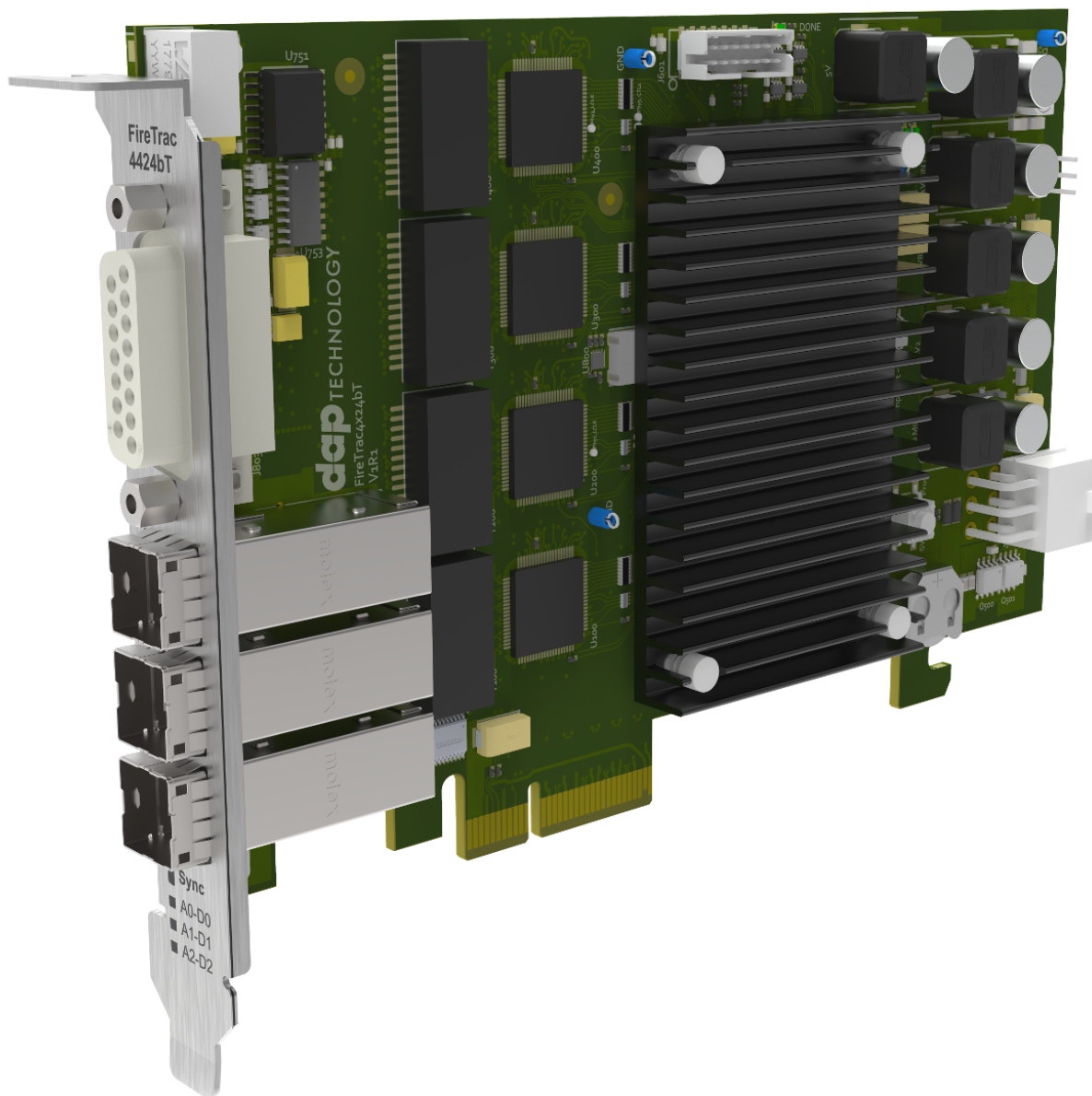
SAE AS5643 describes two types of device categories Control Computers (CC) and Remote Nodes (RN) whose functional definitions depend on the assigned tasks with respect to network communication and integrity verifications. Supporting both device types is an integral objective for FireTrac® and FireStack® and the FireStack API provides feature rich function calls for both categories in parallel. This way the implementer can focus on the AS5643 functional requirements for either CC and RNs. FireStack® doesn't put the device in a specific operational mode (either CC or RN) but all related functions and functionalities exist in parallel and – for a multi-node device can be used independently per node. Example applications are (or will be made) available to demonstrate both scenarios.

However, it is also important to understand that FireTrac® is not just a dedicated “AS5643 device”. It is also a fully compliant IEEE-1394 device. In particular this is important because next to the Asynchronous Streams (which are used for the Asynchronous Subscriber Model (ASM)) FireStack® also supports Asynchronous as well as Isochronous Messaging. Both are optional for the usage in AS5643. But due to its guaranteed quality-of-service the Asynchronous messaging seems to be the logical choice for Cross-Channel-Data-Link (CCDL) implementations. And video/audio streaming in conjunctions with ASM traffic is seen as a future growth path for future variants of AS5643.



1.3. FireTrac4424bT

The FireTrac4424bT implements a 4-bus AS5643 Interface Solution in x4 lane Gen2 PCI Express form factor.



1.3.1. Software Requirements

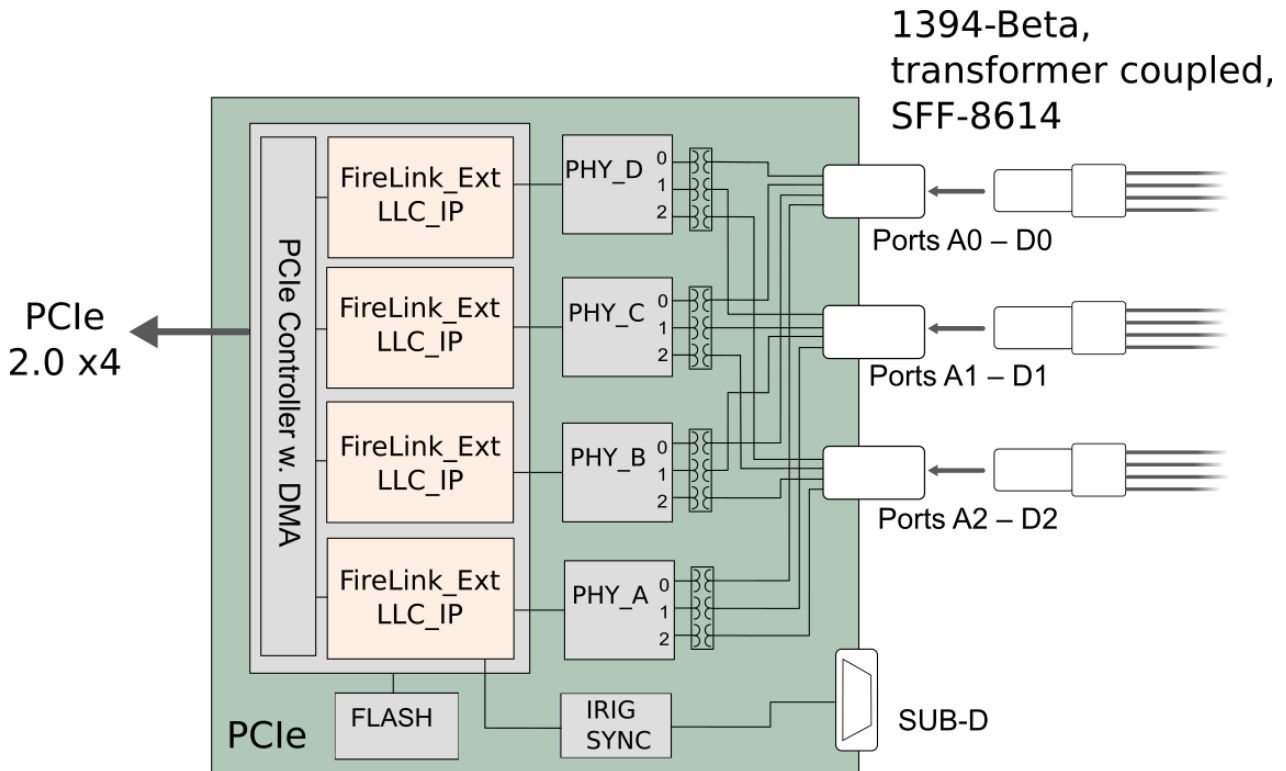
Important Version information

Depending on the FireTrac4424bT board version you may need to install a minimal software version. Please have a look at the following table to look up which software version is required based on the serial number of your card.

FireTrac Version	Software support	Serial number starts with
FireTrac4424bT V1R1	FireTrac software 2.1.3 and later	B-081-
FireTrac4424bT1 V1R1 (S100-S200)	FireTrac software 2.1.3 and later	B-082-

1.3.2. Architecture

The hardware consists of a PCI Express card with four (4) independent physical layer chipsets (transformer coupled) and off-board cable harnesses. Optionally, the harnesses can be connected to a 1U 19" rack mountable panel.



1394 Bus Interface

The 1394 Physical Layer interface consists of four (4) Physical Layer chips (TSB41BA3 from Texas Instruments), active transformers (Pulse) on all twelve (12) ports and three (3) SFF-8614 style off-board connectors. The four (4) instantiations of Link Layer Controller (LLC_IP) utilize DapTechnology's FireLink Extended LLC core that together with the AS5643 Extension module.

For this version of FireTrac® DapTechnology has opted to implement innovative off-board connectivity. Rather than reusing the SCSI2 connector (used on the FT3460bT) the FT4424bT uses SFF-8614 connectors.

SFF-8614 originates from the miniSAS HD interconnectivity technology and has proven its benefits with regards to signal integrity, data throughput, durability and usability.

Customers will benefit for the selected port-to-connector routing (A0-D0, A1-D1, A2-D2) as the need for cables can be adjusted to the specific usage scenario, therefore simplifying harnesses and reducing costs for unneeded cables.

For example, with just one (1) harness the '0"-ports on all 4 nodes (A-D) can be connected as leave nodes. Additional harnesses would only have to be added for daisy-chain or star connectivity.

IRIG Timing

IRIG Timing input is provided via an external signal input. IRIG B time stamping is attached to every single packet. The IRIG decoder decodes the following formats and generates a timestamp for each recorded packet:

- IRIG-B122 (IEEE1344)
- IRIG-B002 (IEEE1344) TTL
- IRIG-B002 (IEEE1344) RS422

External Frame Synchronization

Each 1394 Link has an individual Top Of Frame Input connector associated with it. An input signal can be used to synchronize the AS5643 timed packet transmission to an externally applied Top Of Frame input

signal. These connectors can also be configured as Sync output signal generator.

Direct Memory Access Transfer (DMA)

Data collected in the internal FIFOs (or DPRams) is transferred via DMA burst-block transfers. Bus mastering ensures that the PCI Express device transfers data without any CPU interaction. Optionally, interrupts can be enabled via register settings in order to notify the high-level application about new data availability.

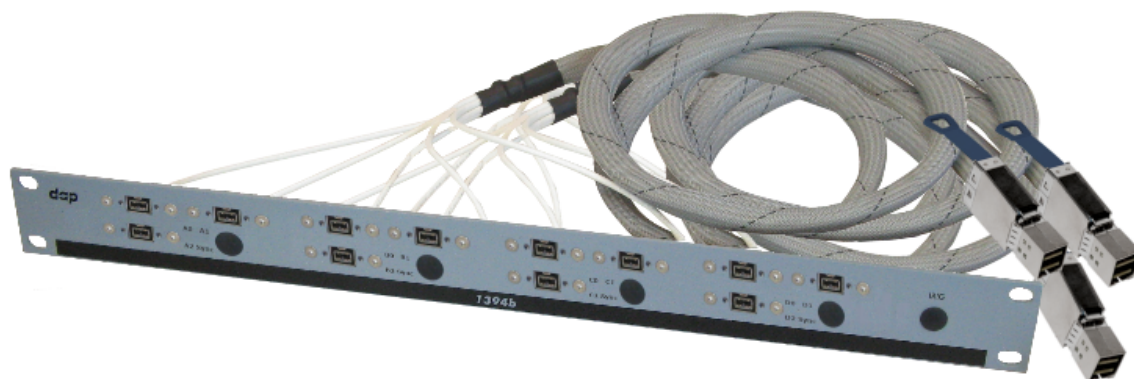
PCI Interface (DMA)

The PCI interface utilizes an IP core from PLDA and is PCI-X Specification 2.0a mode 1 compliant. The core is also PCI specification 3.0 compliant. 32-bit/64bit PCI-X & PCI master/target interface. Bus speed support up to 133MHz (currently only 100MHz supported). Actual implementation details can be found in the next section.

Cable/Harness

In order provide a wide variety of connectivity options – either to additional FireTracs or to other AS5643 equipment, DapTechnology offers a set of harnesses / interconnectivity options.

For example the 19"-breakout panel as depicted below offers an extremely flexible way to connecting bus devices via 1394b bilingual sockets. Such a "patch"-panel can also be configured with LEMO or 38999 sockets depending on the customer's interconnectivity preferences.



Beside the 19"-panels DapTechnology will offer also a series of "patch-cables". For examples, the SFF-to-SFF cable shown below can be used to effectively daisy-chain several FireTracs (ports A0-D0 to A1-D1) with just one single harness. Other variants include fan-out cables with Bilingual or 38999 connectivity. No termination variants are also available).



The selected connector/cable choices offer a large variety of options and cannot displayed in its entirety. Please check on the web for more variants and/or consult with our sales specialists regarding your specific needs, length and connector options. Cables for FireTrac4424bT cards need to be ordered separately. Please consult <http://www.daptechnology.com> for available cable configurations or contact

1.3.3. Dip Switches

SW601 - Flash selection for firmware boot

Location: Back side of the card lower-left quadrant

Positions:

- Main Flash: Start the normal firmware position
- Fallback Flash: Start the fallback firmware position

SW602

Location: Back side of the card top-left quadrant

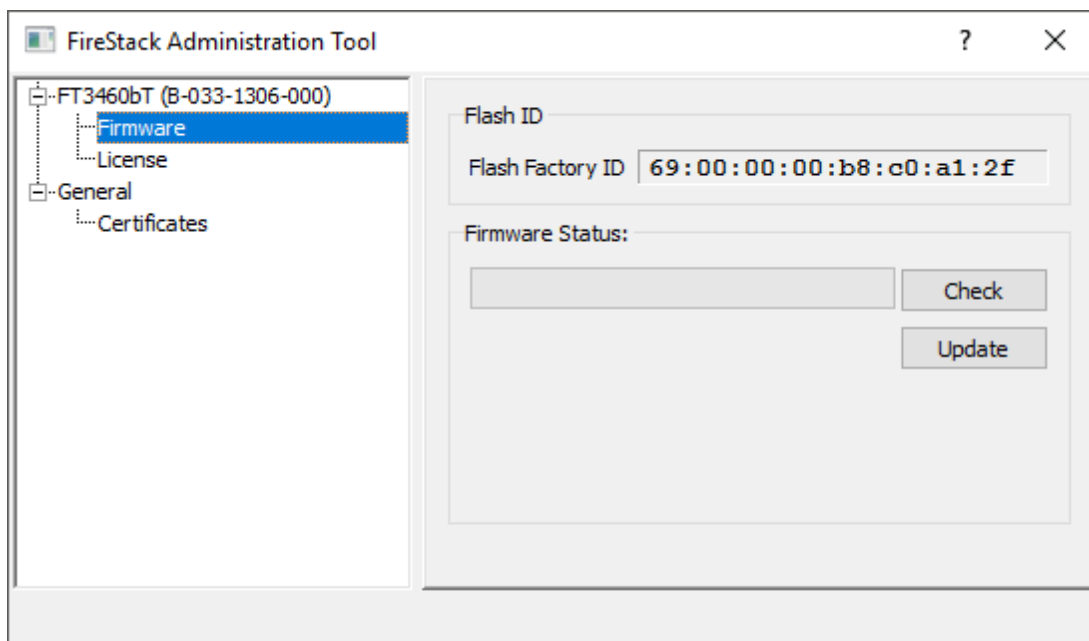
Positions:

- Not yet used

1.3.4. Fall-back firmware

If a firmware update was unsuccessful or cannot be updated, a fall-back version of the firmware can be used to recover the FireTrac4424bT device.

Turn off the host machine and remove the FireTrac4424bT device. On the FireTrac4424bT, place the dip switch SW601 in the Fallback Flash position. Plug the FireTrac4424bT back into the host machine and turn it on. The FireTrac4424bT will now install the fall-back firmware which is stored on the device itself. In Windows, open the Admintool, which is installed with the FireTrac4424bT software.



Select the Firmware tab, and click Update. This will install the latest version of the firmware. After the Admintool has completed updating the firmware, turn off the machine and place the dip switch SW601 back in the Main Flash position. After booting the machine the FireTrac firmware is up to date.

1.4. FireTrac3460bT

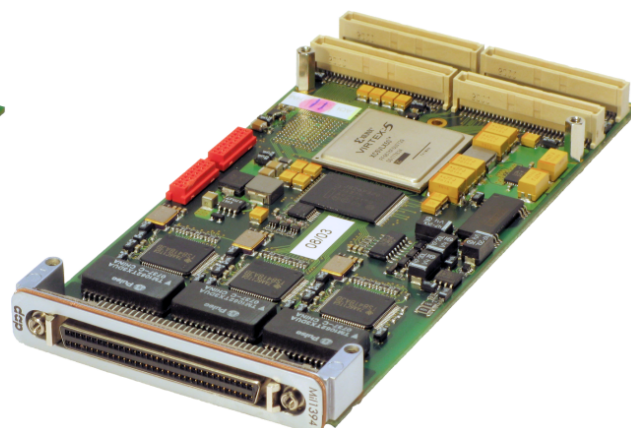
The FireTrac3460bT implements a triple-bus AS5643 Interface Solution in PMC form factor. DapTechnology is constantly trying to improve its products and as a result the FireTrac3460bT is currently at its 3rd board revision. Please refer to the pictures below for the different board versions.



FireTrac3460bT Version 3 (FT3460BT_V3R1)



FireTrac3460bT Version 2 (FT3460T_V2R1)
(replaced by V3)



FireTrac3460bT Version 1 (FT3460T_V1R1)
(replaced by V2)

1.4.1. Software Requirements

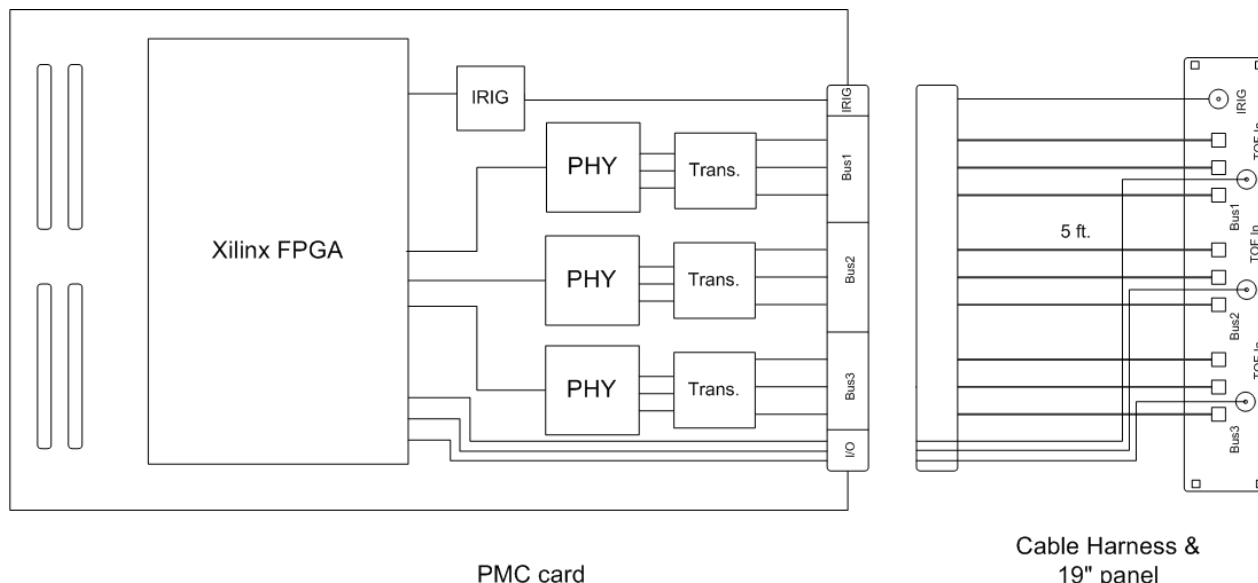
Important Version information

Depending on the FireTrac3460bT board version you may need to install a minimal software version. Please have a look at the following table to look up which software version is required based on the serial number of your card.

FireTrac Version	Software support	Serial number starts with
FireTrac3460bT V3 (S100-S200) manufactured after 01/01/2019	FireTrac software 2.0.10 and later	B-080-
FireTrac3460bT V3 manufactured after 01/01/2019	FireTrac software 2.0.10 and later	B-079-
FireTrac3460bT V3 (S100-S200)	FireTrac software 0.98.4 and later	B-036-
FireTrac3460bT V3	FireTrac software 0.98.4 and later	B-033-
FireTrac3460bT V2	FireTrac software 0.20 and later	B-025-
FireTrac3460bT V1	All software versions	B-024-

1.4.2. Architecture

The hardware consists of a PMC card with three independent physical layer chipsets (transformer coupled) and an off-board cable harness. Optionally, this harness can be connected to a 1U 19" rack mountable panel. PMC cards can be easily used on different form factor (PCI, PXI, cPCI, VME, ...). Based on future requirements, DapTechnology will also develop direct PCI or other adapter cards.



1394 Bus Interface

The 1394 Physical Layer interface consists of three Physical Layer chips (TSB41BA3 from Texas Instruments), active transformers (Pulse) on all nine ports and one SCSI-2 style off-board connector.

IRIG Timing

IRIG Timing input is provided via an external signal input. IRIG B time stamping is attached to every single packet.

External Frame Synchronization

Each 1394 Link has an individual Top Of Frame Input connector associated with it. An input signal can be used to synchronize the AS5643 timed packet transmission to an externally applied Top Of Frame input signal. On FireTrac V3 and later these connectors can also be configured as Sync output signal generator.

Direct Memory Access Transfer (DMA)

Data collected in the internal FIFOs (or DPRams) is transferred via DMA burst-block transfers. Bus mastering ensures that the PMC device transfers data without any CPU interaction. Optionally, interrupts can be enabled via register settings in order to notify the high-level application about new data availability.

On-Board Memory

Your FireTrac3460bT may or may not contain an on-board QDRII SRAM module of 9MB. Since this module is not used in any way, since 01/01/2019 it is no longer placed. *)

A tiny battery-backed configuration memory is used to store important fixed device-specific configuration data (**). This memory is written during the production process and is not used in any way to store additional/new

information after that.

An on-board FLASH chip is available that holds firmware to be loaded into the FPGA on system startup time. This memory is only used to store new firmware updates and license certificates.

All memory can be sanitized by the end-user except card-specific configuration data written during production of the card.

*) Only on FireTrac3460bT version 2 (FT3460T_V2R1) and 3 (FT3460T_V3R1) delivered before 01/01/2019. This memory is no longer placed.

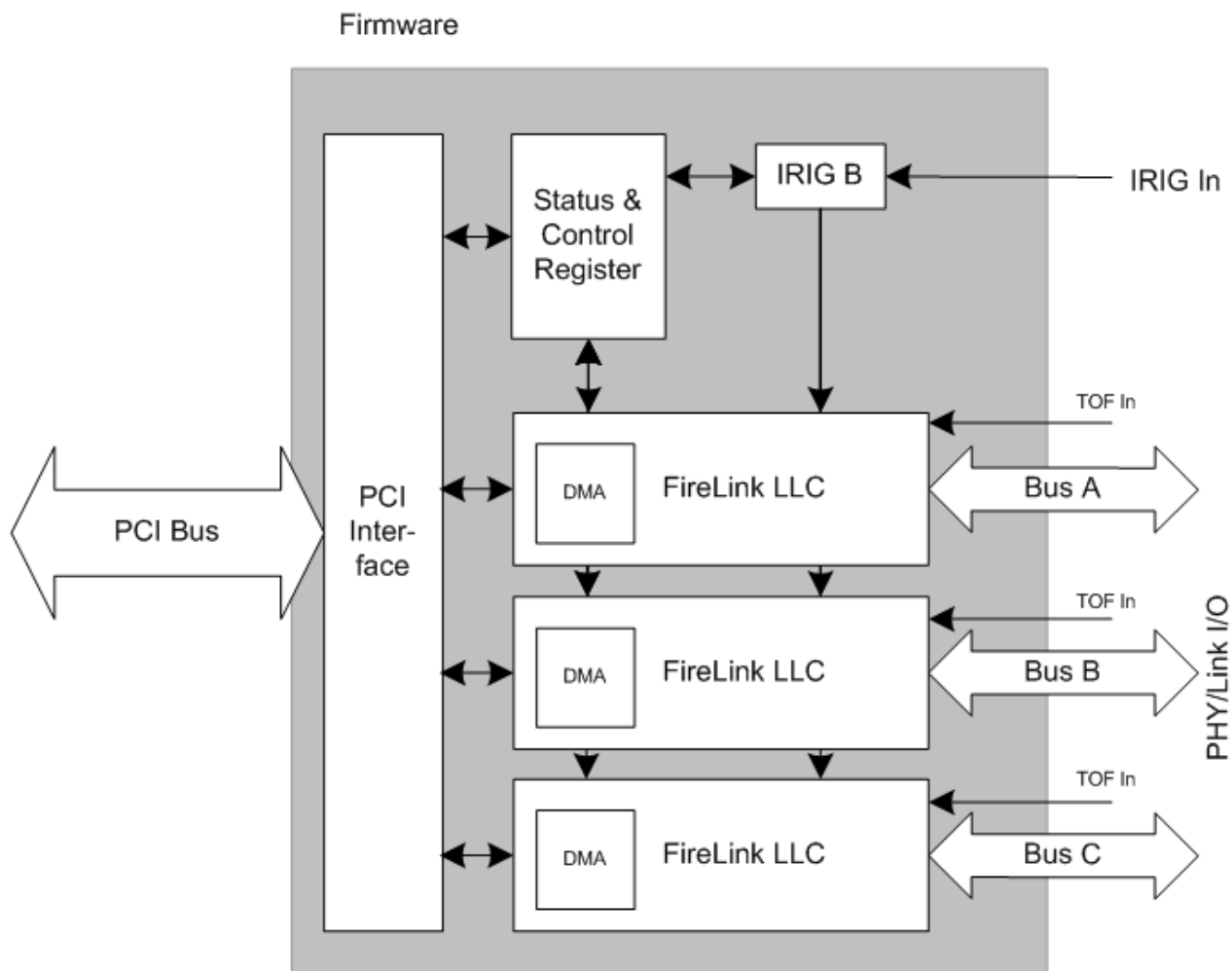
***) Only on FireTrac3460bT version 3 (FT3460T_V3R1)

Cable/Harness

Cables for FireTrac3460bT cards need to be ordered separately. Please consult <http://www.daptechnology.com> for available cable configurations or contact support@daptechnology.com.

1.4.3. Firmware

The following provides an overview of the FireTrac3460bT AS5643 FPGA firmware.



FireLink Extended Mil1394 Only

The three instantiations of Link Layer Controller (LLC) utilize DapTechnology’s FireLink Extended LLC core that together with the AS5643 Recording and Simulation Logic, the integrated IRIG controller and DMA engine form the centerpiece of the FireTrac3460bT AS5643 hardware and firmware.

IRIG Decoder Module

The IRIG decoder decodes the IRIG B122 and IEEE1344 standard and generates a timestamp for each recorded packet.

PCI Interface (DMA)

The PCI interface utilizes an IP core from PLDA and is PCI-X Specification 2.0a mode 1 compliant. The core is also PCI specification 3.0 compliant. 32-bit/64bit PCI-X & PCI master/target interface. Bus speed support up to 133MHz (currently only 100MHz supported). Actual implementation details can be found in the next section.

1.4.4. Connector pinning**68p SCSI II Connector Pinning**

Pin Number	Signal Name
42	PHY A TX0p
43	PHY A TX0n
8	PHY A RX0p
9	PHY A RX0n
39	PHY A TX1p
40	PHY A TX1n
5	PHY A RX1p
6	PHY A RX1n
36	PHY A TX2p
37	PHY A TX2n
2	PHY A RX2p
3	PHY A RX2n
54	PHY B TX0p
55	PHY B TX0n
20	PHY B RX0p
21	PHY B RX0n
51	PHY B TX1p
52	PHY B TX1n
17	PHY B RX1p
18	PHY B RX1n
48	PHY B TX2p
49	PHY B TX2n
14	PHY B RX2p
15	PHY B RX2n
66	PHY C TX0p
67	PHY C TX0n
32	PHY C RX0p
33	PHY C RX0n
63	PHY C TX1p
64	PHY C TX1n
29	PHY C RX1p
30	PHY C RX1n
60	PHY C TX2p
61	PHY C TX2n
26	PHY C RX2p
27	PHY C RX2n
23	SYNC A
24	SYNC B
57	SYNC C
58	SYNC GND
45	IRIG IN
46	IRIG GND
1, 4, 7, 10, 13, 16, 19, 22, 25, 28, 31, 34, 35, 38, 41, 44, 47, 50, 53, 56, 59, 62, 65, 68	EARTH

Sync A, B and C signal input and output* characteristics

Tabel Limiting values

Symbol	Parameter	Min	Max	Unit
V _{in}	Input voltage	-0.5	+12	V
I _o	Output current		±50	mA

Tabel Recommended operating conditions

Symbol	Parameter	Min	Max	Unit
V _{in}	Input voltage	0	5	V
V _{out}	Output voltage	0	5	V

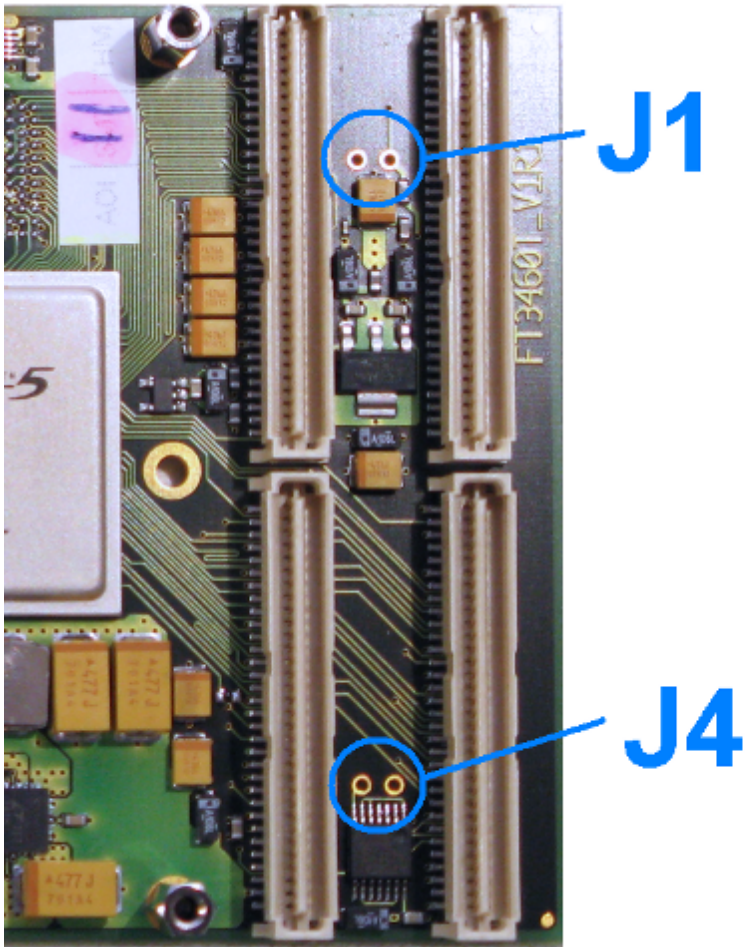
Tabel Static characteristics

		Conditions	Min	Max	Unit
V _{IH}	HIGH-level input voltage		2.0		V
V _{IL}	Low-level input voltage			0.8	V
V _{OH}	HIGH-level output voltage	I _o = 50µA	4.9		
V _{OL}	LOW-level output voltage	I _o = 50µA		0.1	

*) Output is only available on FireTrac V3 and later hardware revisions. Output impedance is 100 Ohm

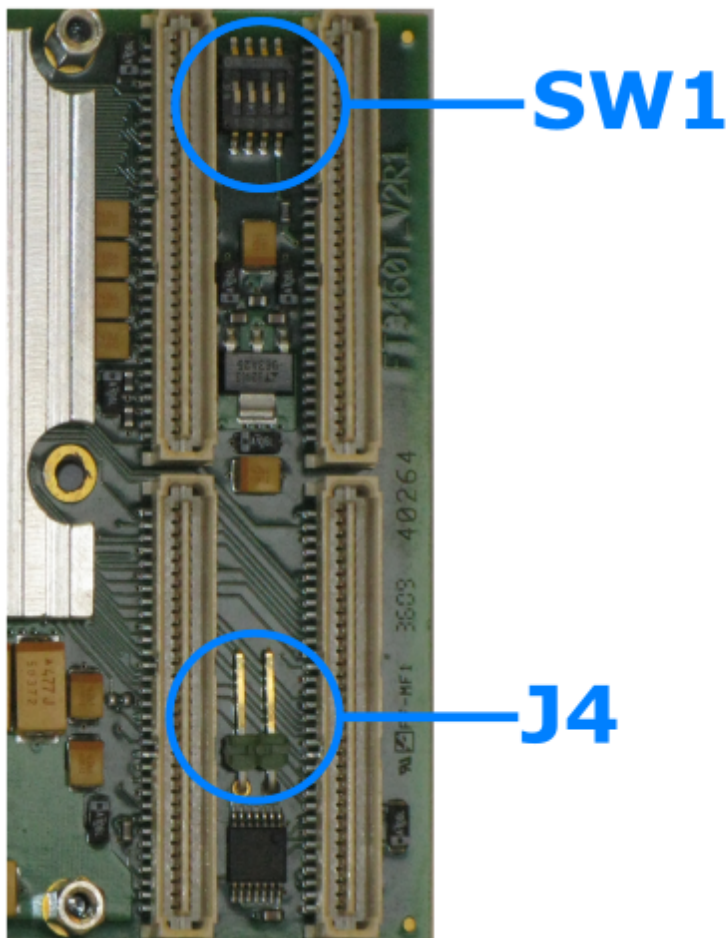
1.4.5. Jumpers

The FireTrac3460bT V1R1 has two jumpers, J1 and J4. Their positions are shown below. Actual jumpers are not shown.



J1, (FireTrac3460bT V1 only) when placed, protects the FLASH memory from being overwritten.
J4, when placed, forces the use of the fall-back firmware when a firmware update was unsuccessful.

Firetrac3460bT V1R2 and V2Rx have a switch block SW1 and jumper J4 as shown below:



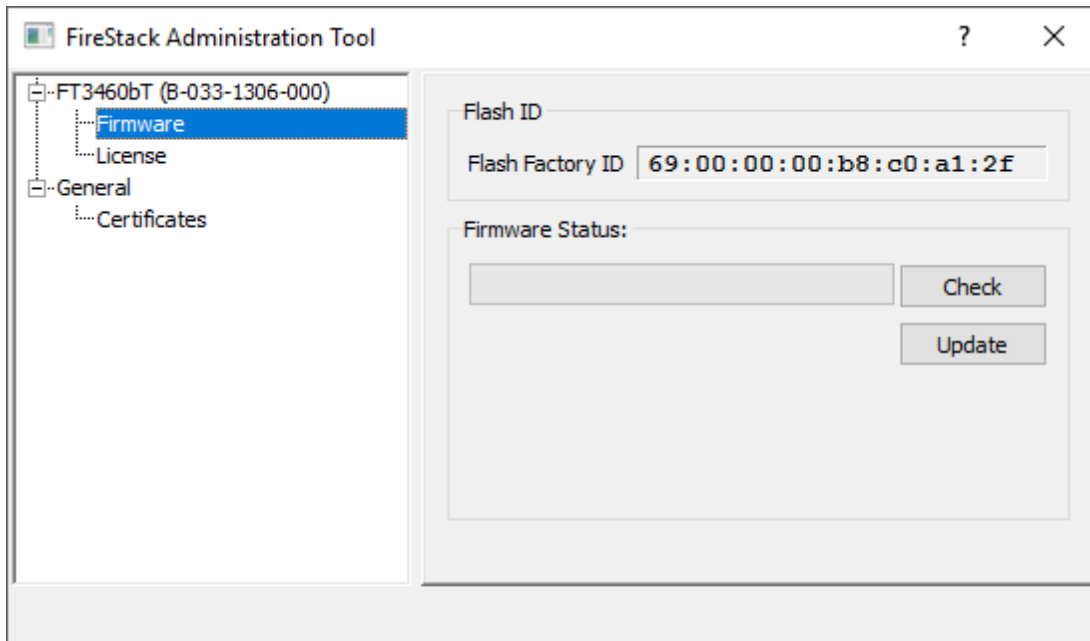
SW1 switch number 4, when enabled, protects the FLASH memory from being overwritten
J4, when placed, forces the use of the fall-back firmware when a firmware update was unsuccessful.

On FireTrac3460bT V3 a different Flash memory type is used for improved security of the firmware. This particular type does not allow hardware write protection. Although SW1 is still on the board its switch number 4 is not connected to the Flash memory but to the FPGA instead for future feature expansion.

1.4.6. Fall-back firmware

If a firmware update was unsuccessful or cannot be updated, a fall-back version of the firmware can be used to recover the FireTrac3460bT device.

Turn off the host machine and remove the FireTrac3460bT device. On the FireTrac3460bT, place the jumper on J4 as shown in [Jumpers](#). Plug the FireTrac3460bT back into the host machine and turn it on. The FireTrac3460bT will now install the fall-back firmware which is stored on the device itself. In Windows, open the Admintool, which is installed with the FireTrac3460bT software.



Select the Firmware tab, and click Update. This will install the latest version of the firmware. After the Admintool has completed updating the firmware, turn off the machine and remove the jumper from J4. After booting the machine the FireTrac firmware is up to date.

1.5. Specifications

FCC Class A Compliance

This equipment has been tested and found to comply with the limits for a Class A digital device, pursuant to part 15 of the FCC Rules. These limits are designed to provide reasonable protection against harmful interference when the equipment is operated in a commercial environment. This equipment generates, uses, and can radiate radio frequency energy and, if not installed and used in accordance with the instruction manual, may cause harmful interference to radio communications. Operation of this equipment in a residential area is likely to cause harmful interference in which case the user will be required to correct the interference at his own expense.

Modifications not expressly approved by the manufacturer could void the user's authority to operate the equipment under FCC rules.

PPC EABI memory limitations

For all platforms, the module has been built using the 'longcall' calling convention to enable the full 32bit address range.

PCI Configuration

Most Single Board Computers perform limited PCI configuration. In general, the BSP (Board Support Package) only configures PCI-PCI bridges and devices on the SBC itself. Some assume a fixed PCI topology, and use hard-coded PCI bus- and device numbers to address PCI devices. Many have limited or no support for PCI interrupt sharing.

This makes the DapTechnology PMC341bT a challenging device to support on vxWorks platforms. It contains a PCI-PCI bridge, with three OHCI link layer devices on the secondary PCI bus segment. It uses three PCI interrupt lines (`INT_A` ... `INT_C`) where most PCI adapters only use `INT_A`. This may result in interrupt sharing between the PMC341bT and an adapter in the other PMC site, or an on-board PCI device.

The FireStack software constructs the PCI topology starting from the PMC site(s), configures PCI-PCI bridges where necessary and routes interrupt lines. It also implements PCI interrupt sharing between the devices it supports, but interrupt sharing between a FireStack device and an on-board device (or third party device in another PCI site) ultimately depends on BSP support.

2.1.3. Device specific notes**FireTrac (FT3460bT)**

Make sure the PMC slot is operating no faster than 100MHz PCI-X mode. The FireStack software will check the PCI speed that the FireTrac is running at and if this exceeds 100MHz PCI-X mode it will print an error message similar to this when the FireStack software is loaded:

```
FireTrac in PMC2 set to 133MHz PCI-X mode, limit is 100MHz
PCI device configuration failed for PCI(1:7:0)
```

It may be possible to restrict the PCI-X speed of the PMC sites. Refer to the user manual of your single board computer and the next section of this manual.

2.1.4. Platform specific notes**VMetro M6000**

The VMetro M6000 defaults to 133MHz PCI-X mode. Refer to the section "PCI-X Capability Selection for PMC Slots" of Appendix A3, "DIP Switch Settings" in the VMetro Hardware Guide to configure the PMC slot(s). When the VMetro system is started, it prints a hardware inventory. Assuming a FireTrac card installed in PMC slot #1 (the rightmost slot), it can be recognized by this output:

```
[...]
PCI:  Segment                               Configuration
      PPC440SP PCI Bus#0 (Slot#1 + PCIe Bridge)  100 MHz PCI-X (1.0)
[...]
PCI:  Mezzanine                               Vendor      Device      Rev   Type
      Slot#1                                   0x194a     0x1202     03   PMC
      Slot#2                                   No device
[...]
```

The FireTrac V2 device has PCI vendor ID `0x194A` (DAP Technology) and device ID `0x1202`.

Motorola MV7100

On the Motorola MV7100, several PCI interrupt pins are routed to interrupt line #3:

- `INT_A` of the Tsi148 VMEbus controller
- `INT_C` of PMC site #1
- `INT_B` of PMC site #2

This means that the third OHCI bus of a PMC341bT installed in PMC #1 (or the second OHCI bus of a PMC341bT in PMC #2), shares an interrupt line with the VMEbus controller. If this OHCI device raises an interrupt, it will be delivered to the handlers of both FireStack and the VMEbus controller. The VME handler

was not written with interrupt sharing in mind and will issue the log message `VME interrupt with no cause`. If this happens it is best to comment out this log message (in `sysTempeVmeIntr.c`) and rebuild the `vxWorks` image.

RadStone PPC7D

By default PMC2 is configured for 133MHz PCI-X mode (this is the PMC site closest to the CPU). PMC1 runs at 100MHz PCI-X mode and is the recommended place to install a FireTrac card.

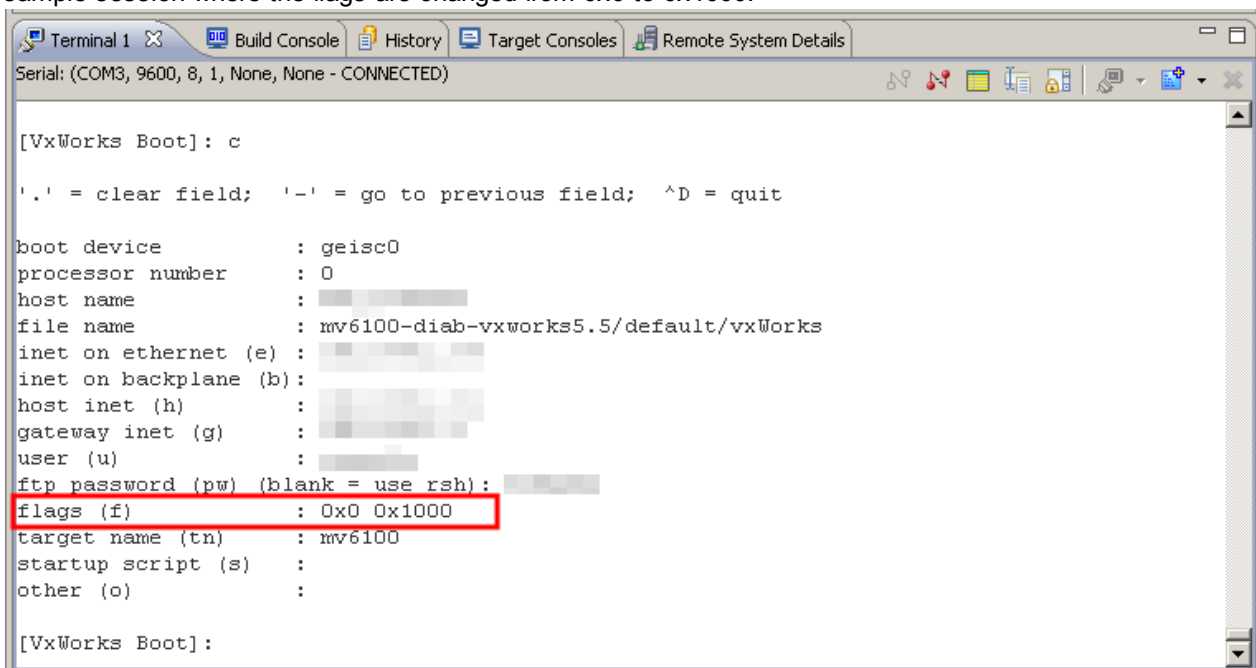
The default configuration of the RadStone PPC7D BSP limits available system memory to 32MB. With the default `vxWorks` image and the FireStack binary loaded, only 20MB RAM is left for DMA buffers. This is not enough to run the pre-compiled `Mil1394datalogger` example.

Refer to the 'Memory' section of the 'Radstone PowerX Platforms/Tornado BSP Manual' to find out how to make all installed memory available to `vxWorks`, or reduce the value of `BUFFERS_PER_CONTEXT` in the `datalogger.h` file of the `Mil1394datalogger` example to reduce the memory requirements of the example.

2.1.5. Configuration Flags

Under normal circumstances the FireStack software does not generate any output except in case of fatal errors. It is possible to get more verbose output from the FireStack software when it probes and configures supported hardware. This is implemented using the boot loader 'flags', specifically `SYSFLG_VENDOR_0` (0x1000).

To set boot loader flags, you have to interrupt the boot process and press 'c' to start configuration. Here's a sample session where the flags are changed from 0x0 to 0x1000:



```

Terminal 1 X Build Console History Target Consoles Remote System Details
Serial: (COM3, 9600, 8, 1, None, None - CONNECTED)

[VxWorks Boot]: c

'.' = clear field; '-' = go to previous field; ^D = quit

boot device      : geisc0
processor number : 0
host name        : 
file name        : mv6100-diab-vxworks5.5/default/vxWorks
inet on ethernet (e) : 
inet on backplane (b): 
host inet (h)    : 
gateway inet (g) : 
user (u)         : 
ftp password (pw) (blank = use rsh): 
flags (f)        : 0x0 0x1000
target name (tn) : mv6100
startup script (s) : 
other (o)        : 

[VxWorks Boot]:

```

Then, when the FireStack library is downloaded, the output is similar to this:

```

-> ld < firestackmil1394_0_98_7_ppc604_vxworks5.5/bin/firestackmil1394.out
DapTechnology firestackmil1394 version 0.98.7
Build date: Jun  6 2014, 14:43:11
Configured for vxworks version 5.5 [CPU=ppc604]
Supported single board computers:
  Motorola MVME6100
  GE Fanuc - PPC7D
Supported devices:
  194a:1201/0000:0000 FireTrac 3460bT (fallback firmware)
  194a:1202/0000:0000 FireTrac 3460bT
  194a:1203/0000:0000 FireTrac 3460bT (fallback firmware)

```

```

194a:1215/0000:0000 S1600 1394 Host Adapter Card
194a:1216/0000:0000 S1600 1394 Host Adapter Card (Fallback firmware)
104c:8025/194a:1001 PCI341bT OHCI Compliant Host Adapter
104c:8025/194a:1002 PMC341bT OHCI Compliant Host Adapter
104c:823f/194a:1003 FCG183PCIE OHCI Compliant Host Adapter
194a:1219/0000:0000 FireTrac 3460bT
194a:1220/0000:0000 FireTrac 3460bT (fallback firmware)
Probed Motorola MVME6100-0173 - MPC 7457
Configure Motorola MVME6100 [BSP 1.2/5]
Probed PMC341bT OHCI Compliant Host Adapter in PCI(3:4:0)
Probed PMC341bT OHCI Compliant Host Adapter in PCI(3:5:0)
Probed PMC341bT OHCI Compliant Host Adapter in PCI(3:6:0)
Probed FireTrac 3460bT in PCI(2:6:0)
Configure PCI(3:4:0): revision 1, PMC 1, offset 8, vector 0x00000050
Configure PCI(3:4:0): enable big endian operation
Configure PCI(3:5:0): revision 1, PMC 1, offset 9, vector 0x00000051
Configure PCI(3:5:0): enable big endian operation
Configure PCI(3:6:0): revision 1, PMC 1, offset 10, vector 0x00000052
Configure PCI(3:6:0): enable big endian operation
Configure PCI(2:6:0): revision 3, PMC 2, offset 6, vector 0x00000052
Configure PCI(2:6:0): enable big endian operation
value = 1039171808 = 0x3df080e0
->

```

The actual output varies with build configuration and hardware. In this case, a PMC341bT is installed in PMC site #1, and a FireTrac 3460bT in PMC site #2.

2.1.6. Upgrading Firmware

Firmware of the FireTrac card can be upgraded in the field using a special firmware upgrade utility, `fwupdater-firestackmil1394.out`. This is a kernel based application object module. It is important that the `firestackmil1394.out` module is **not** loaded during a firmware upgrade. The firmware upgrade utility can be loaded into the target system using the development tools on the host system or the kernel shell on the target.

This is an example kernel shell session of a firmware upgrade. We're upgrading the second of two FireTrac cards. The entry point is 'main':

```

-> ld < firestackmil1394_2_1_10_ppc604_vxworks5.5/bin/fwupdater-firestackmil1394.out
value = 1034304096 = 0x3da63a60
-> main
-----
--          DapTechnology Firmware Programmer
--          Version firestackmil1394-2.1.10-vxworks
-----

----- Options -----
[P] Print device list
[U] Update Firmware
[V] Verify Firmware
[T] Test All Available Firmware
[F] Flash License into Card
[S] Show Flash License in Card
[E] Erase Flash Data Section
[?] This Menu
[X] Exit

MMM
MM
MMMMM, .MM      MMMMM.MM  MMM.MMMMM~.
.MMMMMMMMMMMM  MMMMMMMMMMMM MMMMMMMMMMMM.
MMM.    MMMM MMMM    MMMM  MMM.    .MMM
MMM.                                     MMM.
MMM.    MMM  MMMM    MMM  MMMM.    .MMM.
MMMMMMMMMMMMM  NMMMMMMMMMMMMM MMMMMMMMMMMMM.
.MMMMMM MMM    ,MMMMM MMM  MM.MMMMMM,
MM
MMM

[0-1] Select other card
Found 2 programmers

Current Device: 0
>1
>u
Checking firmware in flash...
Firmware in flash is different.
Starting update...

```



```
Erasing Flash...
#####

Writing Firmware...
#####
Checking firmware in flash...

Verifying Firmware...
#####
Firmware in Flash is updated
Please power cycle system for changes to take effect
>
```

The whole procedure should complete in a couple of minutes. You will have to power cycle the target system before the new firmware takes effect, and the FireStack software will be able to use it.

2.1.7. License Management

A FireTrac card comes with a license certificate which enables features of the FireStack software. This license certificate can be registered at run-time with [fxAddLicenseCertificate](#), or permanently in flash memory on the FireTrac card. Licenses installed in flash memory are automatically registered when a 1394 bus is opened using [fxCreateBusHandle](#).

The license certificate looks like this:

```
--- START OF LICENSE CERTIFICATE ---

    kGuoVkYlsFq13aoB
    5JzabxdAGXIfhNnV
    FUGafv702OoxuBt3
    hyoyiZVgM0IhmzM8
    8tAz6d6ENObs+DVi
    SNlEXw==

--- END OF LICENSE CERTIFICATE ---
```

Normally, a FireTrac card is shipped with its license certificate in flash. The firmware upgrade utility, `fwupdater-firestackmil1394.out`, can be used to view or register this license certificate.

Assuming a FireTrac with no license certificate in flash memory, here's how to install it:

```
----- Options -----
[P] Print device list
[U] Update Firmware
[V] Verify Firmware
[T] Test All Available Firmware
[F] Flash License into Card
[S] Show Flash License in Card
[E] Erase Flash Data Section
[?] This Menu
[X] Exit

[0-1] Select other card
>s
No license in flash
>f
----- Flash License -----
With this feature it is possible to store a license key on the device
in internal flash memory. When FireStack is used to open a device
which contains a license key in flash memory, it will automatically use that,
eliminating the need to enter one manually.
A license key may be broken up into multiple lines of text input, an empty
line will finalize input.
```

```
-----  
kGuoVkYlsFq13aoB5JzabxdAGXIfhNnVFUgafv702OoxuBt3hyoyiZVgM0IhmzM88tAz6d6ENObs+DViSN1EXw==
```

```
Written License..  
>
```

Finally, type **s** to verify the license:

```
>s  
License: 'kGuoVkYlsFq13aoB5JzabxdAGXIfhNnVFUgafv702OoxuBt3hyoyiZVgM0IhmzM88tAz6d6ENObs+DViSN1EXw=='  
----- License Information -----  
Modules:  
    Low-Level API  
    Mil1394 Transmission  
  
Operating Systems:  
    VxWorks 5.5  
    VxWorks 6.2  
VersionMajor: 1
```

Chapter 3. Definition of Terms

Bus	A device can contain several PHYs, each one representing a node on a 1394 network. Each PHY of a device is referred to as a bus, even though multiple PHYs could be connected to the same 1394 network.
Node	Each PHY connected to a 1394 network is referred to as a node on the network.
Network	One or more interconnected 1394 PHYs form a 1394 network.

Chapter 4. Document Conventions

Each function description has its own numbered section. A function description can contain one or more of the following items:

Description

This item provides a general description of what the function does. In some cases when the use of a function is not trivial, an example is provided for a C-code development environment.

Parameters

This item contains a table with function parameter descriptions.

Return Codes

This item describes possible return values of the function.

Synopsis

This item contains the function prototype in C-style as it can be found in the API header file.

Chapter 5. Parameter Naming Conventions

The size of a function input data pointer is always called 'size' and placed after the data pointer in the function parameter list as an `uint32_t`:

```
void myFunc(const uint32_t* data, uint32_t size)
```

Size is always expressed in the number of items of the data pointer type, so in most of the cases the number of `uint32_t` values.

The size of allocated memory of a function output parameter pointer is always called `maxSize` and placed after the data pointer in the function parameter list as an `uint32_t`:

```
void myFunc(uint32_t* buffer, uint32_t maxSize)
```

`maxSize` is always expressed in the number of items of the data pointer type, so in most of the cases the number of `uint32_t` values.

A function input data pointer that is not of a struct type is called "data":

```
const uint32_t* data
```

A function output data pointer which is not of a struct type is called "buffer":

```
uint32_t* buffer
```

Bitmasks are always unsigned and their name indicates it is a bit mask:

```
uint32_t nodeMask
```

Chapter 6. API Revision History

This section contains an overview of changes to the API. This section does not reflect other changes like performance improvements and/or new functionality implemented without changing the API.

6.1. Changes in 2.1.x series

Release 2.1.3

General

- New macros have been defined for additional frame sync modes for the FireTrac4x24bT card:
 - [FX MIL SYNC SIGNAL D](#)
 - [FX MIL SYNC BUS 3](#)
 - [FX MIL SYNC OUT SIGNAL D](#)

6.2. Changes in 2.0.x series

Release 2.0.6

AS5643 Transmission Related

- New option for Repeating mode context: [FX_MIL_TRM_CONTEXT_OPT_FRAME_SKIP_COUNT](#)

Release 2.0.3

- New settings for resource usage of Asynchronous modules
 - [FX_SETTING_ID_ASYNC_MAX_TRM_QUEUE_LENGTH](#)
 - [FX_SETTING_ID_ASYNC_NUM_RCV_BUFFERS](#)
- New settings for resource usage of AS5643 modules
 - [FX_SETTING_ID_MIL_RCV_MAX_BUFFERS](#)
 - [FX_SETTING_ID_MIL_TRM_MAX_BUFFERS](#)

Release 2.0.1

General

- Structure [FXBusInfoStruct](#) has been extended with a deviceType field

AS5643 Related

- New [Input Modes](#) for Frame Synchronization [FX_MIL_SYNC_SIGNAL_A](#) to [FX_MIL_SYNC_SIGNAL_C](#) and [FX_MIL_SYNC_BUS_0](#) to [FX_MIL_SYNC_BUS_2](#)
- New [Output Modes](#) for Frame Synchronization [FX_MIL_SYNC_OUT_SIGNAL_A](#) to [FX_MIL_SYNC_OUT_SIGNAL_C](#)
- New function [fxMilTrmCreateContextHandleExt](#)
- New structure [FXMilTrmContextOption](#)
- New context options [FX_MIL_TRM_CONTEXT_OPT_MODE](#), [FX_MIL_TRM_CONTEXT_OPT_JITTER_RANGE](#) and [FX_MIL_TRM_CONTEXT_OPT_JITTER_DIRECTION](#)
- New message option [FX_MIL_TRM_OPT_JITTER_MODE](#)

Release 2.0.0

General

- New macros [FX_SPEED_TYPE_AUTO](#) and [FX_SPEED_TYPE_FIXED](#)
- New macros [FX_SPEED_MASK](#) and [FX_SPEED_TYPE_MASK](#)

Asynchronous Transactions

- Structure [FXTransactionOptions](#) has been extended with a speedMode field. Setting this to zero reverts to previous behavior.

Low Level

- New [fxIssueBusReset](#) function
- New [fxPhySetForceRoot](#) function
- New [fxPhySetGapCount](#) function

6.3. Changes in 1.0.x and 0.98.x series

General

- [fxGetEUI64\(\)](#) moved out of Low-Level module into Bus Initialization section
- `FX_EVENT_ISOTRM_BUFFER` macro removed
- New macros `FX_EVENT_ISOTRM_CONTEXT` and `FX_EVENT_ISOTRM_PACKET`
- New macros `FX_ERR_MODULE_NOT_AVAILABLE` and `FX_ERR_MODULE_NOT_LICENSED`
- New Setting type [FX_SETTING_ID_DEMO_MODE](#)

Low-Level Module

- New function [fxPingRemoteNode\(\)](#)

Serial Bus Management

- `FX_GENERAL_FEATURE_CONNECTION_MANAGEMENT` macro removed as it is not used
- `FX_SBM_BUS_MANAGER_CAPABLE` macro removed as it is not used
- New function [fxGetMaxSpeedToNode\(\)](#)

Outbound Transactions

- New macro `FX_OBD_TRANSACTION_MAXREACHED`
- New macro `FX_ERR_OBD_TRN_MAXIMUM_REACHED`

Isochronous Transmission

- All definitions in this module are new

Mil1394 Transmission

- [FXMilTrmCallback](#) has a new parameter `eventCodes`
- The `optionList` and `optionSize` parameters have been removed from [fxMilTrmStrmWriteImmediate](#). The options were ignored in the previous releases, and each packet in data must contain the offset and the speed option (see [Data Formats](#)).
- The macro `FX_MIL_CTRLFLAG_SKIPAFTERFRAMEEND` is no longer supported.
- New function [fxMilTrmSetMessageOptions\(\)](#)

Chapter 7. General Structures and Definitions

7.1. Type Definitions

7.1.1. Basic Types

The following basic types are defined to ensure that each user-defined variable will have the right number of bits when used as parameter for a function call.

<code>int32_t</code>	32 bits signed integer
<code>uint32_t</code>	32 bits unsigned integer
<code>float32_t</code>	32 bits single precision floating-point
<code>float64_t</code>	64 bits double precision floating-point

7.1.2. Special Types

<code>FXReturnCode</code>	Data type for holding error codes as returned by all functions. A human readable error message can be found for each error code with the function fxGetErrorMessage .
<code>FXBusHandle</code>	Handle to a bus opened by the fxCreateBusHandle function.

7.2. Structure Definitions

7.2.1. FXInt64

Description

This structure can be used to hold a 64 bit signed integer value.

Synopsis

```
typedef struct {
    uint32_t    highWord;
    uint32_t    lowWord;
} FXInt64;
```

7.2.2. FXUInt64

Description

This structure can be used to hold a 64 bit unsigned integer value.

Synopsis

```
typedef struct {
    uint32_t    highWord;
    uint32_t    lowWord;
} FXUInt64;
```

7.2.3. FXAddress64

Description

This structure can be used to hold a 1394 memory address.

Synopsis

```
typedef struct {
    uint32_t    highAddress;
    uint32_t    lowAddress;
} FXAddress64;
```

7.2.4. FXChannelMask

Description

This structure can be used to hold a 64 bit channel mask value.

Members

channelHi	bit31: Channel 63 bit0: Channel 32
channelLo	bit31: Channel 31 bit0: Channel 0

Synopsis

```
typedef struct {
    uint32_t    channelHi;
    uint32_t    channelLo;
} FXChannelMask;
```

7.2.5. FXSetting

Description

This structure can be used to hold a setting. An array of FXSetting can be specified when creating a handle to a device. Available settings depend on several aspects like which FireStack modules are included, operating system and link layer type. Each component that has settings will document available settings in the manual section of that component.

Members

settingId	Setting ID
value	Setting value.

Synopsis

```
typedef struct {  
    uint32_t    settingId;  
    int32_t     value;  
}FXSetting;
```

7.3. Constants

7.3.1. Speed Codes

The definitions below may be used for specifying packet transmission speed.

FX_SPEED_100	100 Mbps
FX_SPEED_200	200 Mbps
FX_SPEED_400	400 Mbps
FX_SPEED_800	800 Mbps
FX_SPEED_1600	1600 Mbps
FX_SPEED_3200	3200 Mbps

7.3.2. Transaction Types

The definitions below may be used for specifying transaction access types. The definitions can be logically ORed if necessary.

FX_TRANSACTION_READ_ACCESS	Read access
FX_TRANSACTION_WRITE_ACCESS	Write access
FX_TRANSACTION_LOCK_ACCESS	Lock access

Chapter 8. Administrative Functions

This section describes all administrative functions. Administrative functions can be used to open and close a device and get handles to the busses it is connected to. This section also contains descriptions of functions for retrieving information about a device or other piece of common API information.

8.1. Initialization

Before the user application calls any of the FireStack functions, the FireStack itself needs to be initialized with the function [fxInitialize\(\)](#). After the user closes the last device and just before the user-application exits, the FireStack needs to be terminated by making a function call to [fxTerminate\(\)](#).

8.1.1. Functions

8.1.1.1. *fxInitialize*

Description

This function needs to be called before any other functions are called and should only be called once per running application. This function sets up FireStack internal data structures and builds the list of supported devices. If this function is not called, the FireStack will simply not find any supported 1394 busses.

Return Codes

This function returns zero or greater upon success. A negative return value indicates an error. Please use the function [fxGetErrorMessage](#) to lookup descriptions corresponding to negative return values.

Synopsis

```
FXReturnCode fxInitialize(
    void
)
```

8.1.1.2. *fxTerminate*

Description

This function needs to be called right before the user-application exits and after the last FireStack handle has been closed.

Return Codes

This function returns zero or greater upon success. A negative return value indicates an error. Please use the function [fxGetErrorMessage](#) to lookup descriptions corresponding to negative return values.

Synopsis

```
FXReturnCode fxTerminate()
```

8.1.1.3. *fxAddLicenseCertificate*

Description

This function can be used to register license certificates by means of a zero terminated string. This must be done each session, otherwise the library enters a demo mode where everything is enabled for only a small amount of time (if this mode is available in your FireStack profile). This function must be called after [fxInitialize](#) and before [fxCreateBusHandle](#), otherwise the license will be ignored.

Please note that a license with an expiration date will automatically expire at midnight after that day, and terminate the FireStack. Any memory resources allocated with the FireStack should then not be used anymore. In case you register multiple license certificates for a device, their components will be combined. They will expire on the earliest expiration date (if any). If one of the licenses is invalid, [fxCreateBusHandle](#) will fail when trying to create a handle for the respective device(s).

Parameters

certificate	A zero-terminated string containing the license certificate. Space, tab and new-line characters will be ignored.
-------------	--

Return Codes

This function returns zero or greater upon success. A negative return value indicates an error. Please use the function [fxGetErrorMessage](#) to lookup descriptions corresponding to negative return values.

Synopsis

```
FXReturnCode fxAddLicenseCertificate(  
    const char* certificate  
)
```


8.2. Bus Initialization

Each device contains one or more 1394 PHY chips, referred to as busses in this document. Most of the functions work on a single PHY and need a bus handle as input to choose which 1394 bus to control.

The functions in this section can be used to acquire and release handles to the busses of a device.

8.2.1. Functions

8.2.1.1. *fxGetNumberOfBuses*

Description

This function will query for supported 1394 devices and build a list of [FXBusInfo](#) structures. A copy of the list can be retrieved by the user by calling [fxGetBusInfoList](#).

Parameters

numBuses	Returns the number of supported 1394 busses found
----------	---

Return Codes

This function returns zero or greater upon success. A negative return value indicates an error. Please use the function [fxGetErrorMessage](#) to lookup descriptions corresponding to negative return values.

Synopsis

```
FXReturnCode fxGetNumberOfBuses(
    uint32_t* numBuses
)
```

8.2.1.2. *fxGetBusInfoList*

Description

This function may be called after calling to [fxGetNumberOfBuses](#) to get an array of [FXBusInfo](#) structures. The user needs to take care of allocating an array and specifying its size when calling this function.

Parameters

list	User-allocated bus information list. The stack will copy its internal list into this one.
maxSize	The number of FXBusInfo structs that fit in the list
size	The actual number of FXBusInfo structs returned

Return Codes

This function returns zero or greater upon success. A negative return value indicates an error. Please use the function [fxGetErrorMessage](#) to lookup descriptions corresponding to negative return values.

Synopsis

```
FXReturnCode fxGetBusInfoList(
    FXBusInfo* list,
    uint32_t    maxSize,
    uint32_t*   size
)
```

8.2.1.3. *fxCreateBusHandle*

Description

This function creates and allocates a handle to the specified 1394 bus. The user has the option to either manually create an [FXBusInfo](#) structure with valid information or to use one of the [FXBusInfo](#) structs returned by [fxGetBusInfoList](#).

When opening a handle the user may choose to specify a list of settings by providing the function with an array of [FXSetting](#) structures. Available settings depend on various aspects like operating system used,

FireStack modules included in this specific FireStack release and higher-level protocols included in this specific FireStack release. Whenever a module offers user-configurable settings it will include them in the module's documentation.

In addition to the per module settings, the following general settings are available:

- [Setting for enabling/disabling features](#)
- [Setting for starting in Demo Mode](#)
- [Setting to force the byte order of 1394 packet data](#)

Parameters

info	User-specified FXBusInfo struct that corresponds to the 1394 bus to open.
settingList	An array of struct type FXSetting that allows providing settings when creating a handle or one can set this parameter to zero to leave out settings.
size	The number of settings in the provided settingList array.
busHandle	If successful, returns a handle to the opened bus.

Return Codes

This function returns zero or greater upon success. A negative return value indicates an error. Please use the function [fxGetErrorMessage](#) to lookup descriptions corresponding to negative return values.

```

FX_ERR_DEVICE_INIT_FAIL
FX_ERR_MEM_ALLOC_FAIL
FX_ERR_INVALID_PCI_REVISIONID
FX_ERR_NEEDS_FWUPDATE
FX_ERR_MODULE_NOT_AVAILABLE
FX_ERR_MODULE_NOT_LICENSED
FX_ERR_INVALID_PCI_SPEED

```

Synopsis

```

FXReturnCode fxCreateBusHandle(
    const FXBusInfo* info,
    FXSetting* settingList,
    size_t size,
    FXBusHandle* busHandle
)

```

8.2.1.4. fxCloseBusHandle

Description

This function frees resources used by the specified bus handle.

Parameters

busHandle	Reference handle to the bus to control. (see fxCreateBusHandle)
-----------	---

Return Codes

This function returns zero or greater upon success. A negative return value indicates an error. Please use the function [fxGetErrorMessage](#) to lookup descriptions corresponding to negative return values.

```

FX_ERR_INVALID_HANDLE
FX_ERR_DEVICE_CLOSE_FAIL

```

Synopsis

```

FXReturnCode fxCloseBusHandle(

```

```

        FXBusHandle    busHandle
    )

```

8.2.1.5. fxGetEUI64

Description

This function can be used to obtain a 64-bit extended unique identifier stored in the hardware.

Parameters

handle	Reference handle to the bus to control. (see fxCreateBusHandle)
EUI64	Pointer to FXUint64 structure to which the EUI-64 data is returned.

Return Codes

This function returns zero or greater upon success. A negative return value indicates an error. Please use the function [fxGetErrorMessage](#) to lookup descriptions corresponding to negative return values.

FX_ERR_INVALID_HANDLE

Synopsis

```

FXReturnCode fxGetEUI64(
    FXBusHandle    handle,
    FXUint64*     EUI64
)

```

8.2.2. Structures

8.2.2.1. FXBusInfo

Description

This structure defines data members that together provide sufficient information to identify a device. When used in combination with [fxCreateBusHandle](#) sufficient information needs to be filled in to identify the device to be opened, all other fields may be set to -1.

For example if a user just wants to open the first 1394 bus device found in the system it is sufficient to set deviceId to 0 and the remaining fields to -1 when calling [fxCreateBusHandle](#).

When a user wants to open a specific PCI physical location all PCI fields should be filled in correctly and all other fields need to be set to -1 when calling [fxCreateBusHandle](#).

Simplest way to use this structure to open a device is to let the FireStack fill in the fields. This can be done by using the function [fxGetBusInfoList](#).

Members

deviceId	Device identification number. Unique for each 1394 bus device connected to the system. This number is not related to physical configuration but picked by software.
pciBus	PCI bus number for PCI devices, -1 otherwise. Indicates the PCI bus number the 1394 bus device is connected to. Numbering is defined by the physical PCI bus topology.
pciDevice	PCI device number for PCI devices, -1 otherwise. Numbering is defined by the physical PCI bus topology.
pciFunction	PCI function for PCI devices, -1 otherwise. In case of a multi-function PCI device, this field holds the function index of the time input module.

pciRevision	PCI Revision for PCI devices, -1 otherwise. In case of FireTrac devices this represents the firmware version.
deviceType	The type of hardware device this bus is part of. Actual definitions can be found above this structure in firestack.h header file. They are in FX_DEVICE_TYPE_..... format.

Synopsis

```
typedef struct {
    /* Identification */
    int32_t          deviceId;

    /* PCI Physical location */
    int32_t          pciBus;
    int32_t          pciDevice;
    int32_t          pciFunction;

    /* Information */
    int32_t          pciRevision;

    /* Device Type */
    int32_t          deviceType;
} FXBusInfo;
```

8.2.3. Settings**8.2.3.1. Features**

The following setting can be used as settingId in an [FXSetting](#) structure passed to [fxCreateBusHandle\(\)](#) to control which features will be enabled for the bus that is being opened. By default all modules are enabled for which a valid license key is present.

FX_SETTING_ID_GEN_FEATURES

The following settings can be or-ed together to form the [FXSetting](#) value field:

FX_GENERAL_FEATURE_CONFIG_ROM

This feature will setup a configuration ROM for the local node and expose that onto the 1394 bus. If this feature is enabled other nodes on the bus can read the configuration ROM of the local node by means of Asynchronous Read Requests or Asynchronous Block Read Requests.

FX_GENERAL_FEATURE_INBOUND_TRANSACTIONS

Set this bit to enable the [Inbound Transactions](#) Module. Please be aware that a valid license is needed for this module.

FX_GENERAL_FEATURE_OUTBOUND_TRANSACTIONS

Set this bit to enable the [Outbound Transactions](#) Module. Please be aware that a valid license is needed for this module.

FX_GENERAL_FEATURE_ISO_RECEIVE

Set this bit to enable the [Isochronous Reception](#) Module. Please be aware that a valid license is needed for this module.

FX_GENERAL_FEATURE_FIRESTACK_MIL1394_RECEIVE

Set this bit to enable the [Mil1394 Reception](#) Module. Please be aware that a valid license is needed for this module.

FX_GENERAL_FEATURE_FIRESTACK_MIL1394_TRANSMIT

Set this bit to enable the [Mil1394 Transmission](#) Module. Please be aware that a valid license is needed for

this module.

FX_GENERAL_FEATURE_LOWLEVEL

Set this bit to enable the [Low Level](#) Module. Please be aware that a valid license is needed for this module.

FX_GENERAL_FEATURE_BUS_MANAGEMENT

Set this bit to enable the Serial Bus Management Module. Please be aware that a valid license is needed for this module.

The following features are currently reserved:

- **FX_GENERAL_FEATURE_ISO_TRANSMIT**
- **FX_GENERAL_FEATURE_CONNECTION_MANAGEMENT**

8.2.3.2. Demo Mode

The following setting can be used as settingId in an [FXSetting](#) structure passed to [fxCreateBusHandle\(\)](#).

FX_SETTING_ID_DEMO_MODE

Set the value to 1 to start in demo mode with all features enabled. This mode expires after 5 minutes. Set the value to 0 to start in normal mode. Please note that for this mode a valid license certificate is required.

8.2.3.3. Stack Size

This ID can be used as a setting in an [FXSetting](#) structure passed to [fxCreateBusHandle\(\)](#):

FX_SETTING_ID_STACKSIZE

Several functions, such as [fxMilSetStofCallback\(\)](#), [fxMilRcvSetContextOptions\(\)](#), allow the user to register one or more callback functions. These callbacks execute within the context of the User Event Callback thread, one of several threads spawned for each bus opened with [fxCreateBusHandle\(\)](#).

The `FX_SETTING_ID_STACKSIZE` setting allows the use to increase the default stack size of the User Event Callback thread. The argument specifies the stack size in bytes, and must be a decimal value larger than 8192.

This setting is specific to the vxWorks version of the FireStack software.

8.2.3.4. Thread Priorities

This ID can be used as a setting in an [FXSetting](#) structure passed to [fxCreateBusHandle\(\)](#):

FX_SETTING_ID_THRPRIO_BASE

The firestackmil1394 software spawns several threads when a device is opened with [fxCreateBusHandle\(\)](#): an interrupt servicing thread, an internal event handling thread, and a thread from which user callbacks are executed. These threads have carefully chosen priorities: the interrupt handling threads have the highest priority, followed by the event handling and user event callback threads. The interrupt thread of the first FireLink device has the highest priority level (20). This results in the following threads and priorities, with the mil1394datalogger example running from the target shell:

```
-> i
  NAME          ENTRY          TID      PRI   STATUS   PC          SP          ERRNO  DELAY
-----
tExcTask      excTask      3df fe3c0  0  PEND    1b3190 3df fe2a0    0      0
tLogTask      logTask      3df fb820  0  PEND    1b3190 3df fb710    0      0
tShell        shell        3df 816a0  1  DELAY   1aeb40 3df 813c0    0      1
tWdbTask      wdbTask      3df 83940  3  READY   1a9fbc 3df 83810    0      0
fst_isr0      0x3def5228  3df 7b2e0  20  PEND    1a9fbc 3df 7b210    0      0
fst_isr1      0x3def5228  3deaae40  21  PEND    1a9fbc 3deaad70    0      0
fst_isr2      0x3def5228  3cf54810  22  PEND    1a9fbc 3cf54740    0      0
fst_iev0      0x3def5620  3df61930  25  PEND    1a9fbc 3df61830    0      0
fst_iev1      0x3def5620  3d6e3c50  26  PEND    1a9fbc 3d6e3b50    0      0
fst_iev2      0x3def5620  3cf52590  27  PEND    1a9fbc 3cf52490    0      0
fst_uev0      0x3def594c  3df5f710  30  PEND    1a9fbc 3df5f620    0      0
```

```
fst_uev1    0x3def594c    3d6e1a30    31    PEND            1a9fbc    3d6e1940    0      0
fst_uev2    0x3def594c    3cf50290    32    PEND            1a9fbc    3cf501a0    0      0
tNetTask    netTask       3dfbd630    50    READY          1a9e08    3dfbd3f0    0      0
tPhyTask    0x118f10     3dfba090    150   PEND            1a9fbc    3dfb9fc0    0      0
value = 0 = 0x0
```

The value passed with the `FX_SETTING_ID_THRPRIO_BASE` option, when used with [fxCreateBusHandle\(\)](#), will be the priority of the interrupt servicing thread of the device being opened. The priorities of the internal event handling threads are derived from the priority of the interrupt servicing thread using the algorithm described above. When using the option `FX_SETTING_ID_THRPRIO_BASE`, it is recommended to assign a unique, sequential priority level to every device being opened.

This setting is specific to the vxWorks version of the firestackmil1394 software.

8.2.3.5. Byte Order

This ID can be used as a setting in an [FXSetting](#) structure passed to [fxCreateBusHandle\(\)](#):

`FX_SETTING_ID_BYTE_ORDER`

This setting is used to control byte swapping during host bus accesses on the data portion of a 1394 packet. Byte swapping, if necessary, is performed by the host adapter to avoid CPU overhead. These values are possible:

`FX_BYTE_ORDER_HOST_NATIVE`

The default action when this setting is not applied. 1394 packet data is in host-native byte order.

`FX_BYTE_ORDER_BIG_ENDIAN`

Enforce big endian packet data byte order, even on little endian systems.

`FX_BYTE_ORDER_LITTLE_ENDIAN`

Enforce big endian packet data byte order, even on little endian systems.

8.3. Memory Management

When the user sets up buffers and hands the memory pointer to the FireStack for any kind of DMA operation like packet reception, the user needs to use the following functions to allocate and free the memory buffers.

8.3.1. Functions

8.3.1.1. *fxMemAlloc*

Description

This function can be used to allocate memory that can later be handed to the FireStack for DMA operations. Each function that offers a zero-copy interface to the Link Layer makes use of DMA and can only use memory allocated by this function. Any function that hands a reception buffer to the FireStack for example needs to use this function to allocate the memory buffer. Best practice is to allocate all needed buffers when the user-application starts and free allocated memory right before the application exits. Especially in real-time environments it is better not to allocate and free a lot during program execution.

Parameters

busHandle	Reference handle to the bus to control. (see fxCreateBusHandle)
ptr	If allocation is successful this parameter will contain the pointer. Zero otherwise.
size	The requested buffer size in bytes

Return Codes

This function returns zero or greater upon success. A negative return value indicates an error. Please use the function [fxGetErrorMessage](#) to lookup descriptions corresponding to negative return values.

FX_ERR_INVALID_PARAMETER
FX_ERR_INVALID_HANDLE
FX_ERR_MEM_ALLOC_FAIL

Synopsis

```
FXReturnCode fxMemAlloc(
    FXBusHandle busHandle,
    void** ptr,
    uint32_t size
)
```

8.3.1.2. *fxMemFree*

Description

This function needs to be used to free memory allocated by [fxMemAlloc](#).

Parameters

busHandle	Reference handle to the bus to control. (see fxCreateBusHandle)
ptr	Memory pointer of the buffer to release.

Return Codes

This function returns zero or greater upon success. A negative return value indicates an error. Please use the function [fxGetErrorMessage](#) to lookup descriptions corresponding to negative return values.

FX_ERR_INVALID_PARAMETER
FX_ERR_INVALID_HANDLE

Synopsis

```
FXReturnCode fxMemFree(  
    FXBusHandle busHandle,  
    void* ptr  
)
```


8.4. General

8.4.1. Functions

8.4.1.1. *fxGetLibraryVersion*

Description

This function returns the version information of the library.

Parameters

version	Returns the version information structure for the API lib in use.
---------	---

Return Codes

This function returns zero or greater upon success. A negative return value indicates an error. Please use the function [fxGetErrorMessage](#) to lookup descriptions corresponding to negative return values.

FX_ERR_INVALID_PARAMETER

Synopsis

```
FXReturnCode fxGetLibraryVersion(
    FXVersionInfo* version
)
```

8.4.2. Structures

8.4.2.1. *FXVersionInfo*

Description

This structure holds the three sub version numbers that all together form the complete software version. This structure does not indicate whether it concerns a beta or alpha version.

Members

majorVersion	The major version number.
minorVersion	The minor version number.
patchVersion	The patch version number.

Synopsis

```
typedef struct {
    uint32_t    majorVersion;
    uint32_t    minorVersion;
    uint32_t    patchVersion;
} FXVersionInfo;
```

8.5. Error Handling

Most functions return a negative return value (FXReturnCode) in case an error occurred. These error conditions can be tracked by the user in three ways:

- By inspecting the return value of each function call
- By registering a single error callback function ([fxSetErrorCallback](#)) that will be called every time an error occurs.
- By checking the current error state after a couple of function calls by using the function [fxGetErrorStatus](#).

To translate an error code into a human readable text, please use the function [fxGetErrorMessage](#).

8.5.1. Functions

8.5.1.1. *fxGetErrorMessage*

Description

This function can be used to lookup the message text corresponding to the specified error code.

Parameters

error	The error code to lookup.
buffer	The error text will be written to this memory pointer, including a terminating zero.
maxSize	The maximum size of the data that will be written, including the terminating zero.

Return Codes

On success, this function returns the length of the actual message (including the terminating zero), which may be more than what could be placed in the buffer. To get the full message, you should pass a buffer of at least that size to this function. A negative return value indicates an error.

FX_ERR_INVALID_PARAMETER

Synopsis

```
FXReturnCode fxGetErrorMessage(
    FXReturnCode    error,
    char*           buffer,
    uint32_t        maxSize
)
```

8.5.1.2. *fxSetErrorCallback*

Description

This function can be used to register a single callback function that will be called every time a function is about to return an error code.

Parameters

callback	The user-defined function that will be called when a function returns an error. There is only one function that can be registered: Calling this function with another callback function will replace the current one, and with 0 will disable the callback.
----------	---

Return Codes

This function returns zero or greater upon success. A negative return value indicates an error. Please use the function [fxGetErrorMessage](#) to lookup descriptions corresponding to negative return values.

Synopsis

```
FXReturnCode fxSetErrorCallback(
```

```

        FXErrorCallback callback
    )

```

8.5.1.3. *fxGetErrorStatus*

Description

If an API function returns an error, the error code is stored as the current error status. If the current error status already contains an error then it will not be overwritten. Therefore the error status will always reflect the first error that was encountered.

Every time this function is called, the error status is cleared and the first error after clearing it will again be remembered.

Current error status is not used by the API itself and does not have to be cleared by the user. It is just a convenience function to determine what was the first error after it was cleared the last time.

Return Codes

Please use the function [fxGetErrorMessage](#) to lookup descriptions corresponding to negative return values.

Synopsis

```
FXReturnCode fxGetErrorStatus()
```

8.5.2. Constants

8.5.2.1. Error Codes

The following values may be returned by functions.

```

FX_ERR_GENERAL
FX_ERR_INTERNAL_ERROR
FX_ERR_NOT_IMPLEMENTED
FX_ERR_INVALID_HANDLE
FX_ERR_DEVICE_INIT_FAIL
FX_ERR_DEVICE_CLOSE_FAIL
FX_ERR_MEM_ALLOC_FAIL
FX_ERR_INVALID_PARAMETER
FX_ERR_INVALID_LOCAL_NODE_ID
FX_ERR_NEEDS_FWUPDATE
FX_ERR_INVALID_PCI_SPEED

```

```

FX_ERR_LICENSE_FIRESTACK_VERSION
FX_ERR_LICENSE_NOT_FOUND
FX_ERR_LICENSE_EXPIRED
FX_ERR_LICENSE_DEVICE
FX_ERR_LICENSE_OPERATING_SYSTEM
FX_ERR_LICENSE_CORRUPT
FX_ERR_LICENSE_MODULE
FX_ERR_LICENSE_SYSTEMDATE

```

8.5.3. Type Definitions

8.5.3.1. *FXErrorCallback*

Description

Users can define a function of this type and register it using the function [fxSetErrorCallback](#). The user defined function will then be called each time a function is about to return a negative error code.

Parameters

errorCode	Contains the error code that is returned by a function that caused an error.
-----------	--

Synopsis

```
typedef void (*FXErrorCallback) {  
    FXReturnCode    errorCode  
};
```

Chapter 9. AS5643 Protocol API Reference

The SAE-AS5643 protocol differs from other 1394 Protocols because of its timing requirements. 1394b supports asynchronous transactions/streams and isochronous streams. Isochronous streams offer (in a way) something close to timed transmission because of its timed cycles, however this would just offer a 125us resolution which is not accurate enough for the AS5643 frame timing. Therefore, if a standard 1394b Link Layer were to be used for the AS5643 protocol, Asynchronous Streams need to be used in combination with some kind of software-implemented AS5643 protocol timing.

Although AS5643 protocol timing could be implemented in software, it would be complicated to absolutely guarantee the required accuracy. At DapTechnology we strongly believe that the AS5643 protocol timing should be considered an extension to the 1394b required functionality of a Link Layer and we therefore have been using our own Analyzer Engine for years in the FireSpy series. We have also implemented the AS5643 protocol timing in our FireLink Extended as an add-on module. DapTechnology's FireLink Extended is easily capable of meeting the AS5643 frame timing requirements and eliminates the need for complicated interrupt schemes or real-time operating systems to use the AS5643 protocol.

The FireStack software library contains an AS5643 protocol module that can be used to control the AS5643 hardware of either a custom FireLink Extended enabled product or DapTechnology's FireTrac I/O card. This section describes how frame timing can be configured and used for both timed transmission and reception.

9.1. AS5643 Frame Timing

The AS5643 protocol introduces the concept of time frames separated by a Start Of Frame packet transmitted onto the bus by a control computer. Each node is supposed to listen to those STOF messages and transmit and receive their own messages only at a predefined time offset relative to the STOF.

FireStack is very flexible in the way it handles the timing of Start of Frames. Frame synchronization for AS5643 reception and transmission may be configured in either one of the following modes:

- Free Running or internal clock (based on a 1 micro second input signal)
- STOF packets on the bus (just any packet on a configurable channel)
- External Sync Input Signals A, B or C
- Internally synchronized to one of the other two FireTrac buses

Please refer to [Frame Synchronization Input Modes](#) for more information.

When frame timing is configured to synchronize (not FreeRunning) then synchronization works as follows depending on the moment in time when the synchronization signal arrives.

- When the synchronization signal arrives within the synchronization margin then the frame counter is incremented and the frame offset clock is reset to zero. Obviously the next expected sync pulse is exactly the frame length after this moment.
- When the synchronization signal arrives outside the synchronization margin, then the frame counter is not incremented but the frame clock is reset to zero basically making it a very long frame. Obviously the next expected synchronization pulse is exactly the frame length after this moment.
- When the synchronization signal does not arrive before the end of the synchronization margin or exactly when the sync margin expires, the frame counter will be incremented and the frame offset clock will be set equal to the synchronization margin. In effect this deals with a missed synchronization signal and uses internal clock to continue operation until the sync signal becomes available again.

Regardless of synchronization mode, the user always needs to specify the frame length being used. If synchronization mode is set to external signal or packet on the bus then a synchronization margin also needs to be specified that determines when the hardware is sensitive to the input signal. This value should be in accordance with the STOF packet accuracy.

On FireTrac V3 and later devices, FireStack is also able to configure one or more of the external sync pins (A, B and/or C) as sync output rather than input. Please refer to [Frame Synchronization Output Modes](#) for more information.

9.1.1. Functions

9.1.1.1. *fxMilSetFrameTimingOptions*

Description

This function can be used to configure the way a bus will handle its frame timing. For more information please see [FXMilFrameTimingOptions](#).

Parameters

busHandle	Reference handle to the bus to control. (see fxCreateBusHandle)
options	Pointer to user-allocated structure that contains the options to set for frame synchronization (see FXMilFrameTimingOptions)

Return Codes

This function returns zero or greater upon success. A negative return value indicates an error. Please use the function [fxGetErrorMessage](#) to lookup descriptions corresponding to negative return values.

FX_ERR_INVALID_HANDLE
FX_ERR_INVALID_PARAMETER

```

FX_ERR_MIL_INVALID_FRAME_SYNC_MODE
FX_ERR_MIL_INVALID_FRAME_SYNC_OUT_MODE
FX_ERR_MIL_INVALID_FRAME_LENGTH
FX_ERR_MIL_INVALID_FRAME_SYNC_MARGIN
FX_ERR_MIL_INVALID_FRAME_SYNC_CHANNEL
FX_ERR_MIL_INVALID_DMA_PRELOAD_TIME

```

Synopsis

```

FXReturnCode fxMilSetFrameTimingOptions(
    FXBusHandle          busHandle,
    const FXMilFrameTimingOptions* options
)

```

9.1.1.2. *fxMilSetStofCallback*

Description

The user can choose to be notified whenever a Start Of Frame (STOF) occurs. This can be done by registering a user-defined callback function. The callback function could for example read current frame number and STOF timestamp for correlating packets to frames by means of timestamps.

This function can be used to register a callback function that will be called when the FireStack detects the start of frame. Specifying 0 (zero) as a pointer of the function will disable calling the function.

Parameters

busHandle	Reference handle to the bus to control. (see fxCreateBusHandle)
callback	Callback function. (see FXMilStofCallback)
userData	Pointer to a user-specified data. The pointer will be carried to user callback functions. See also FXMilStofCallback .

Return Codes

This function returns zero or greater upon success. A negative return value indicates an error. Please use the function [fxGetErrorMessage](#) to lookup descriptions corresponding to negative return values.

FX_ERR_INVALID_HANDLE

Synopsis

```

FXReturnCode fxMilSetStofCallback(
    FXBusHandle          busHandle,
    FXMilStofCallback    callback,
    void*                userData
)

```

9.1.1.3. *fxMilGetFrameOffsetTime*

Description

This function can be used to retrieve the current frame number and the current time offset within the frame.

Parameters

busHandle	Reference handle to the bus to control. (see fxCreateBusHandle)
frameNumber	Pointer to user-allocated variable that will return the current frame number.
frameOffset	Pointer to user-allocated variable that will return the current time offset in micro seconds relative to start of frame.

Return Codes

This function returns zero or greater upon success. A negative return value indicates an error. Please use the

function [fxGetErrorMessage](#) to lookup descriptions corresponding to negative return values.

FX_ERR_INVALID_HANDLE

Synopsis

```
FXReturnCode fxMilGetFrameOffsetTime(
    FXBusHandle      busHandle,
    uint32_t*        frameNumber,
    uint32_t*        frameOffset
)
```

9.1.1.4. fxMilGetStofTimestamp

Description

Read information for the last STOF. Returns timestamp and Frame number.

Parameters

busHandle	Reference handle to the bus to control. (see fxCreateBusHandle)
frameNumber	Pointer to frame number.
timeStamp	Pointer to FXTimeStamp .

Return Codes

This function returns zero or greater upon success. A negative return value indicates an error. Please use the function [fxGetErrorMessage](#) to lookup descriptions corresponding to negative return values.

FX_ERR_INVALID_HANDLE FX_ERR_INVALID_PARAMETER

Synopsis

```
FXReturnCode fxMilGetStofTimestamp(
    FXBusHandle      busHandle,
    uint32_t*        frameNumber,
    FXTimeStamp\*    timeStamp
)
```

9.1.2. Type Definitions

9.1.2.1. FXMilStofCallback

Description

This function definition is used to specify a callback function that will be called when the FireStack detects the start of frame.

Parameters

handle	Reference handle to the bus to control. (see fxCreateBusHandle)
userData	Pointer to the data specified in fxMilSetStofCallback .

Synopsis

```
typedef void (*FXMilStofCallback) {
    FXBusHandle      handle,
    void*            userData
};
```


9.1.3. Structures

9.1.3.1. FXMilFrameTimingOptions

Description

This structure holds options for frame time synchronization and can be used with the function [fxMilSetFrameTimingOptions](#) to setup synchronization.

Members

syncMode	<p>One of the modes documented in Frame Synchronization Input Modes or-ed with one of the modes documented in Frame Synchronization Output Modes.</p> <p>default: (FX_MIL_SYNC_FREERUNNING FX_MIL_SYNC_OUT_NONE)</p>
frameLength	<p>The duration of a frame expressed in micro seconds.</p> <p>A negative value means that the frame length will not be changed.</p> <p>minimum: 3000, maximum: 30000, default: 12500</p>
syncMargin	<p>In case the syncMode is not set to FX_MIL_SYNC_FREERUNNING this value defines the margin in micro seconds before and after the frame length during which a synchronization event will result in incrementing the frame counter.</p> <p>When a synchronization event is detected before the frame offset time reaches $\text{frameLength} - \text{syncMargin}$, the synchronization event will result in clearing the frame offset time, but no new frame is started (frame counter is not incremented).</p> <p>When no synchronization events are detected and the frame offset reaches a value of $\text{frameLength} + \text{syncMargin}$, a new frame is automatically started (frame counter incremented) and the frame offset time is set to the value of the syncMargin (as if a sync was detected at the the time frameOffset was equal to the frameLength).</p> <p>A negative value means that the synchronization margin will not be changed.</p> <p>minimum: 10, maximum 3000, default: 100</p>
channel	<p>When syncMode is set to FX_MIL_SYNC_PACKET this value specifies the 1394 channel number to listen on for synchronization packets. Any received stream packet with the channel number specified here will result in the start of a next frame.</p> <p>A negative value means that the synchronization channel will not be changed.</p> <p>minimum: 0, maximum: 63, default: 31</p>
controlFlags	<p>Combination of one or more of the flags documented in Frame Control Flags.</p>
dmaPreloadTime	<p>This value is only intended for advanced users. It can be used to control the DMA preload time for packets to be transmitted at a specific frame offset time.</p> <p>Whenever a packet needs to be transmitted onto the bus it first will be copied from host memory to the transmission FiFo using the DMA controller. To ensure a packet is actually transmitted at exactly the moment in time it was scheduled at least the first part of the packet needs to be already in the FiFo. When copying a packet to the FiFo a large amount of time before it needs to be transmitted it is not possible anymore to insert packets in the FiFo that need to be transmitted before that packet. Therefore, this value needs to be chosen in such a way that it will be preloaded just in time to be transmitted on time and therefore still offering a lot of flexibility to edit the transmission queue for the current frame.</p> <p>You should set this time to a value equal to or greater than the time that the whole system needs to load the packet. If you set it too small, the packet may be transmitted</p>

on a time later than specified by the frame offset time for that packet. Most systems will have a DMA latency of at most a few micro seconds in normal situations. To allow for special cases (higher priority DMA transfers for instance) it is safe to set this value to a few milli seconds. If you however want to do some real time processing and you want to be able to change some values of the packet data just before it is transmitted, you could set this value as low as possible.

A negative value means that the DMA pre-load time will not be changed.

minimum: 10, maximum: 3000, default: 1000

Synopsis

```
typedef struct {
    uint32_t    syncMode;
    int32_t     frameLength;
    int32_t     syncMargin;
    int32_t     channel;
    uint32_t    controlFlags;
    int32_t     dmaPreloadTime;
} FXMilFrameTimingOptions;
```

9.1.3.2. FXTimeStamp

Description

This structure can be used to hold a timestamp value and corresponding time source status.

Members

seconds	Time stamp seconds value
subSeconds	Time stamp sub-seconds value. (resolution defined by FXTimeInputInfo structure)
statusCode	Time stamp status code: 0 = Invalid 1 = Free Running 2 = Synced (time output is synced to the IRIG input and becoming more accurate over time) 3 = Accurate (time output is synced to the IRIG input with maximum accuracy)

Synopsis

```
typedef struct {
    uint32_t    seconds;
    uint32_t    subSeconds;
    uint32_t    statusCode;
} FXTimeStamp;
```

9.1.4. Constants

9.1.4.1. Error Codes

The following values may be returned by AS5643 Frame Timing functions.

FX_ERR_MIL_INVALID_FRAME_SYNC_MODE
FX_ERR_MIL_INVALID_FRAME_LENGTH
FX_ERR_MIL_INVALID_FRAME_SYNC_MARGIN
FX_ERR_MIL_INVALID_FRAME_SYNC_CHANNEL
FX_ERR_MIL_INVALID_DMA_PRELOAD_TIME
FX_ERR_MIL_INVALID_FRAME_SYNC_OUT_MODE

9.1.4.2. Frame Synchronization Input Modes

The following frame synchronization input modes are available for use with the [FXMiiFrameTimingOptions](#) structure. Please note that the number of Sync Signals and Sync Buses depends on the hardware used.

FX_MIL_SYNC_FREERUNNING	When this mode is set the bus will not synchronize to an input event but will use an internal clock for frame timing. On FireTrac cards this clock originates from the time input device and therefore it may be synchronized to an IRIG time input or just be the built-in clock depending on the way the time input device is configured.
FX_MIL_SYNC_PACKET	When this mode is set the bus will listen on the specified channel for stream packets and whenever a matching packet arrives it is considered an synchronization input event.
FX_MIL_SYNC_SIGNAL	When this mode is set the bus will listen on an external sync input for synchronization events. The specific sync input pin that will be listened on is A for Node 0, B for node 1, C for node 2 and D for node 3.
FX_MIL_SYNC_SIGNAL_A	When this mode is set the bus will listen on an external sync input A for synchronization events.
FX_MIL_SYNC_SIGNAL_B	When this mode is set the bus will listen on an external sync input B for synchronization events.
FX_MIL_SYNC_SIGNAL_C	When this mode is set the bus will listen on an external sync input C for synchronization events.
FX_MIL_SYNC_SIGNAL_D	When this mode is set the bus will listen on an external sync input D for synchronization events.
FX_MIL_SYNC_BUS_0	When this mode is set the bus will synchronize to the frame timer of Bus 0. This mode can not be used on bus 0 itself.
FX_MIL_SYNC_BUS_1	When this mode is set the bus will synchronize to the frame timer of Bus 1. This mode can not be used on bus 1 itself.
FX_MIL_SYNC_BUS_2	When this mode is set the bus will synchronize to the frame timer of Bus 2. This mode can not be used on bus 2 itself.
FX_MIL_SYNC_BUS_3	When this mode is set the bus will synchronize to the frame timer of Bus 3. This mode can not be used on bus 3 itself.

9.1.4.3. Frame Synchronization Output Modes

The following frame synchronization output modes are available for use with the [FXMiiFrameTimingOptions](#) structure:

(Functionality only available on FireTrac V3 and later devices!)

FX_MIL_SYNC_OUT_NONE	When this mode is set the bus will not generate a sync pulse on any external sync pins.
FX_MIL_SYNC_OUT_SIGNAL_A	When this mode is set the bus will generate a sync pulse on external sync pin A. Please make sure that only one bus will control this pin. Please also make sure this pin is not set to input mode by any of the buses.
FX_MIL_SYNC_OUT_SIGNAL_B	When this mode is set the bus will generate a sync pulse on external sync pin B. Please make sure that only one bus will control this pin. Please also make sure this pin is not set to input mode by any of the buses.
FX_MIL_SYNC_OUT_SIGNAL_C	When this mode is set the bus will generate a sync pulse on external

	sync pin C. Please make sure that only one bus will control this pin. Please also make sure this pin is not set to input mode by any of the buses.
FX_MIL_SYNC_OUT_SIGNAL_D	When this mode is set the bus will generate a sync pulse on external sync pin D. Please make sure that only one bus will control this pin. Please also make sure this pin is not set to input mode by any of the buses.

9.1.4.4. Frame Control Flags

The following frame control flags are available:

FX_MIL_CTRLFLAG_SKIPAFTERFRAMEEND	This flag is unsupported and will result in the function call to return an error.
-----------------------------------	---

9.2. AS5643 Reception

AS5643 reception provides a filtering mechanism that filters incoming packets based on their channel number and/or message ID. This section forms the description of all functions needed for receiving AS5643 stream messages.

Basically the following steps need to be taken for initialization and startup of AS5643 reception:

1. Set the channel(s) you are interested in to AS5643 mode. This will disable ISO reception on the specified channels and hand all packets to the AS5643 reception mechanism. Please refer to the [Channel Selections](#) section for more details.
2. Insert filter items into the filter table to include messages for reception. Messages can be filtered on channel number or message ID or a combination of both. Each filter item is associated with a context handle which determines the buffer locations to store the packets in. Each packet received can only be sent to one of the reception contexts. Please refer to the [Message Filters](#) section for more details.
3. The user may use up-to 8 reception contexts, each having its own settings and buffer list. Therefore, the user first needs to setup buffer lists for received packet storage. Each reception context can be associated with one buffer list. For more details please refer to the [Buffer Control](#) section.
4. Use the [Context Control](#) functions to start/stop a context for reception into a specific buffer list and retrieve current context status information.
5. Depending on settings used, each time a buffer is full or a packet is received, a user-defined callback function is called. In case a buffer is full the user can remove it from the buffer list and use its contents for its own purpose. In case of a received packet callback, the user can read from the corresponding buffer but should not remove the buffer from the list until the hardware is done writing to it.

Each time a Start Of Frame (STOF) is detected, a user-defined callback function will be called. Please refer to the [Frame Timing Section](#).

9.2.1. Settings

9.2.1.1. Resource Usage

The following settings can be used as settingId in an [FXSetting](#) instance passed to [fxCreateBusHandle\(\)](#) to control resource usage by the AS5643 Reception module for the bus that is being opened.

FX_SETTING_ID_MIL_RCV_MAX_BUFFERS

This setting determines the maximum number of buffers that may be appended in total to AS5643 Reception contexts at any given point in time.

Default: 10000

9.2.2. Functions

9.2.2.1. Channel Selections

AS5643 channel selection functions can be used to choose between ISO mode (default) and AS5643 mode reception. By default all channels will be handled by isochronous reception mechanisms unless these functions are used to set them to AS5643 mode. If a channel is set to AS5643 mode then message filtering and context settings will determine if and how messages will be received.

9.2.2.1.1 fxMilRcvEnableChannels

Description

This function can be used to set the AS5643 mode flag for multiple channels at the same time.

Parameters

busHandle	Reference handle to the bus to control. (see fxCreateBusHandle)
channelMask	Pointer to FXChannelMask . All channel bits that are 1 will enable the corresponding channel for AS5643 reception. All channel bits that are 0 are left unchanged.

Return Codes

This function returns zero or greater upon success. A negative return value indicates an error. Please use the function [fxGetErrorMessage](#) to lookup descriptions corresponding to negative return values.

FX_ERR_INVALID_HANDLE
FX_ERR_INVALID_PARAMETER
FX_ERR_MIL_RCV_INTERNAL_ERROR

Synopsis

```
FXReturnCode fxMilRcvEnableChannels(
    FXBusHandle          busHandle,
    const FXChannelMask* channelMask
)
```

9.2.2.1.2 fxMilRcvDisableChannels

Description

This function can be used to clear the AS5643 mode flag for multiple channels at the same time.

Parameters

busHandle	Reference handle to the bus to control. (see fxCreateBusHandle)
channelMask	Pointer to FXChannelMask . All channel bits that are 1 will disable the corresponding channel for AS5643 reception. All channel bits that are 0 are left unchanged.

Return Codes

This function returns zero or greater upon success. A negative return value indicates an error. Please use the function [fxGetErrorMessage](#) to lookup descriptions corresponding to negative return values.

FX_ERR_INVALID_HANDLE
FX_ERR_INVALID_PARAMETER
FX_ERR_MIL_RCV_INTERNAL_ERROR

Synopsis

```
FXReturnCode fxMilRcvDisableChannels(
    FXBusHandle          busHandle,
    const FXChannelMask* channelMask
)
```

9.2.2.1.3 fxMilRcvGetEnabledChannels

Description

Returns the status of the current channel mask. Bit: 0 - ISO, 1 - AS5643.

Parameters

busHandle	Reference handle to the bus to control. (see fxCreateBusHandle)
channelMask	Pointer to FXChannelMask . The current channel mask value will be copied to this.

Return Codes

This function returns zero or greater upon success. A negative return value indicates an error. Please use the function [fxGetErrorMessage](#) to lookup descriptions corresponding to negative return values.

FX_ERR_INVALID_HANDLE
FX_ERR_INVALID_PARAMETER

FX_ERR_MIL_RCV_INTERNAL_ERROR

Synopsis

```
FXReturnCode fxMilRcvGetEnabledChannels(
    FXBusHandle          busHandle,
    FXChannelMask*      channelMask
)
```

9.2.2.2. Message Filters

If a channel is selected for AS5643 mode then all incoming packets will be run against a comprehensive message filter system. Messages can be filtered on channel number or AS5643 message ID or a combination of both.

Message filter functions can be used to add and remove filter items from the message filter system. A filter item may specify one of the following:

1. a specific MessageID on a specific channel
2. a specific MessageID on any channel
3. any MessageID on a specific channel

Note that one "a specific MessageID on any channel" filter item actually occupies 63 entries (channel 0 to 62). There are total of 4096 filter entries available.

Packets received can only be sent to one of the available reception contexts. Therefore, when adding new filter items to the message filter system it is not allowed to add an item that would potentially cause a packet to be matched with more than one filter item. As an example the following two filter items may not coexist in the filter system:

- message ID = 1, channel = 2, context handle1
- message ID = 1, any channel, context handle2

The problem with those two items is that a packet with message ID = 1 on channel 2 would be matched by both items which is not allowed. Therefore, in this case the user should just leave out the first item.

Exception in the rule of the combination of filter items is that when "any MessageID on a specific channel" item is added, instead of returning an error if one or more "a specific MessageID with the same channel" items already exist, the message filter system enables the new "any MessageID" item, and all packets with the specified channel will be sent to the specified context. Removing "a specific messageID" item(s) is not required before adding the new "any MessageID" item.

Example - assume that the message filter system already has the following filter items:

- message ID = 10, channel = 4, context handle1
- message ID = 20, channel = 4, context handle2

and then add the following new item:

- any message ID, channel = 4, context handle3

will result in that all packets with channel 4 will be sent to the context handle3.

Each filter item needs to specify a reception context in which a matching packet will be stored. Please refer to the [Context Control](#) section for more information.

9.2.2.2.1 fxMilRcvAddFilterItem

Description

Adds the filter item to the filter table. An error will be returned if the item causes a double match situation.

Parameters

item	Pointer to FXMilRcvFilterItem .
------	---

Return Codes

This function returns zero or greater upon success. A negative return value indicates an error. Please use the function [fxGetErrorMessage](#) to lookup descriptions corresponding to negative return values.

```

FX_ERR_INVALID_HANDLE
FX_ERR_MIL_RCV_INVALID_FILTER_ATTRIBUTES
FX_ERR_MIL_RCV_INVALID_CONTEXT_HANDLE
FX_ERR_MIL_RCV_FILTER_DOUBLE_MATCH
FX_ERR_MIL_RCV_INSUFFICIENT_FILTER_TABLE_SPACE

```

Synopsis

```

FXReturnCode fxMilRcvAddFilterItem(
    const FXMilRcvFilterItem*    item
)

```

9.2.2.2.2 fxMilRcvRemoveFilterItem

Description

Removes the filter item from the filter table. Note that the contextID field of the `FXMilRcvFilterItem` will not be considered when this function searches for the existing filter item.

Parameters

item	Pointer to FXMilRcvFilterItem .
------	---

Return Codes

This function returns zero or greater upon success. A negative return value indicates an error. Please use the function [fxGetErrorMessage](#) to lookup descriptions corresponding to negative return values.

```

FX_ERR_INVALID_HANDLE
FX_ERR_MIL_RCV_INVALID_CONTEXT_HANDLE
FX_ERR_MIL_RCV_INVALID_FILTER_ATTRIBUTES
FX_ERR_MIL_RCV_FILTER_ITEM_NOT_FOUND

```

Synopsis

```

FXReturnCode fxMilRcvRemoveFilterItem(
    const FXMilRcvFilterItem*    item
)

```

9.2.2.2.3 fxMilRcvClearMessageFilter

Description

Clears the message filter table.

Parameters

busHandle	Reference handle to the bus to control. (see fxCreateBusHandle)
-----------	---

Return Codes

This function returns zero or greater upon success. A negative return value indicates an error. Please use the function [fxGetErrorMessage](#) to lookup descriptions corresponding to negative return values.

```

FX_ERR_INVALID_HANDLE

```

Synopsis

```

FXReturnCode fxMilRcvClearMessageFilter(
    FXBusHandle    busHandle
)

```

9.2.2.2.4 fxMilRcvGetNumFilterItems

Description

This function will query for filter items. A copy of the list can be retrieved by the user by calling [fxMilRcvGetFilterItemList](#).

Parameters

busHandle	Reference handle to the bus to control. (see fxCreateBusHandle)
numFilterItems	Returns the number of filter items.

Return Codes

This function returns zero or greater upon success. A negative return value indicates an error. Please use the function [fxGetErrorMessage](#) to lookup descriptions corresponding to negative return values.

FX_ERR_INVALID_HANDLE
FX_ERR_INVALID_PARAMETER

Synopsis

```
FXReturnCode fxMilRcvGetNumFilterItems(
    FXBusHandle          busHandle,
    uint32_t*           numFilterItems
)
```

9.2.2.2.5 fxMilRcvGetFilterItemList**Description**

This function may be called after calling to [fxMilRcvGetNumFilterItems](#) to get an array of [FXMilRcvFilterItem](#) structures.

The user needs to take care of allocating an array and specifying its size when calling this function.

Parameters

busHandle	Reference handle to the bus to control. (see fxCreateBusHandle)
list	User-allocated filter item information list. The stack will copy its internal list into this one.
maxSize	The number of FXMilRcvFilterItem structures that fit in the list.
size	The actual number of FXMilRcvFilterItem structures returned.

Return Codes

This function returns zero or greater upon success. A negative return value indicates an error. Please use the function [fxGetErrorMessage](#) to lookup descriptions corresponding to negative return values.

FX_ERR_INVALID_HANDLE
FX_ERR_INVALID_PARAMETER

Synopsis

```
FXReturnCode fxMilRcvGetFilterItemList(
    FXBusHandle          busHandle,
    FXMilRcvFilterItem* list,
    uint32_t             maxSize,
    uint32_t*           size
)
```

9.2.2.3. Buffer Control

In order to start a context for reception, the user first needs to setup the necessary buffers for packet storage. Buffer Control functions can be used to register memory buffers with the AS5643 reception module. The user needs to take care of allocating memory blocks that can be used as reception buffer. As long as a piece of memory is registered as reception buffer, the user may not free it or write to it. The user may read from it at

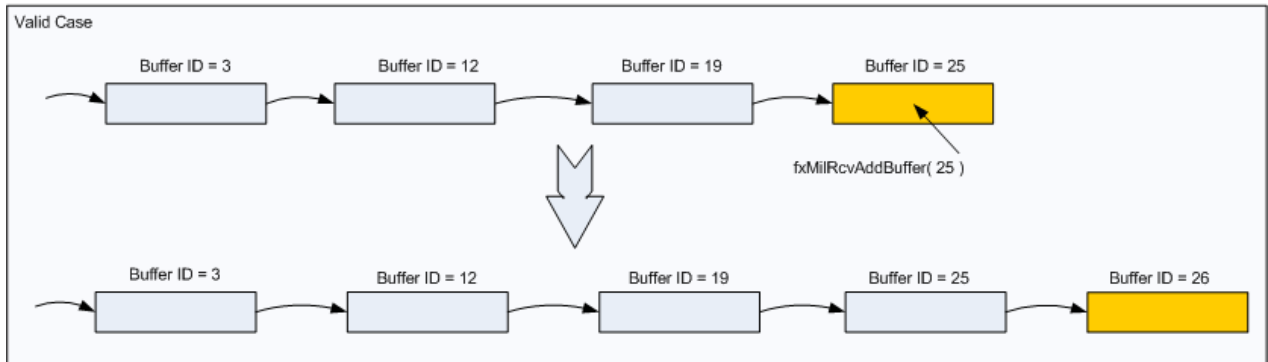
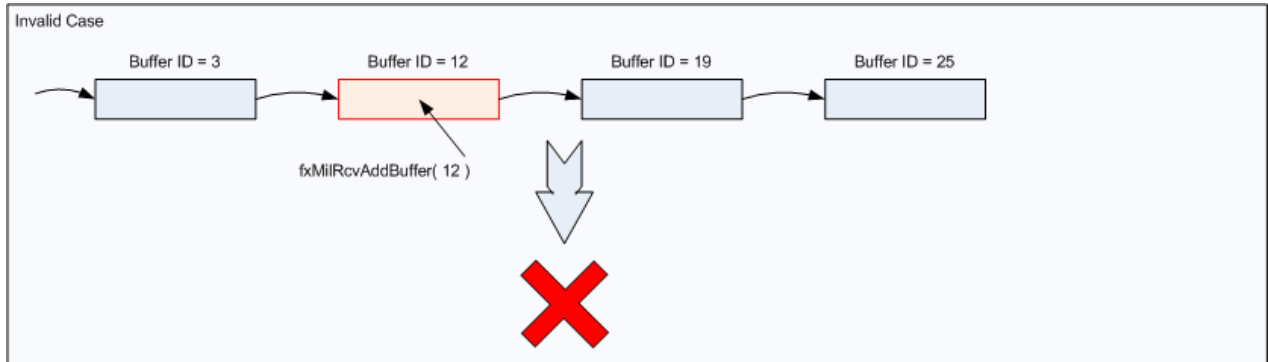
all times. After removing a buffer from the reception list, the user may write and/or free memory again.

Buffers need to be setup such that they form a list. It is not allowed to link buffers in a loop. Having buffers in a loop fashion would yield unspecified results.

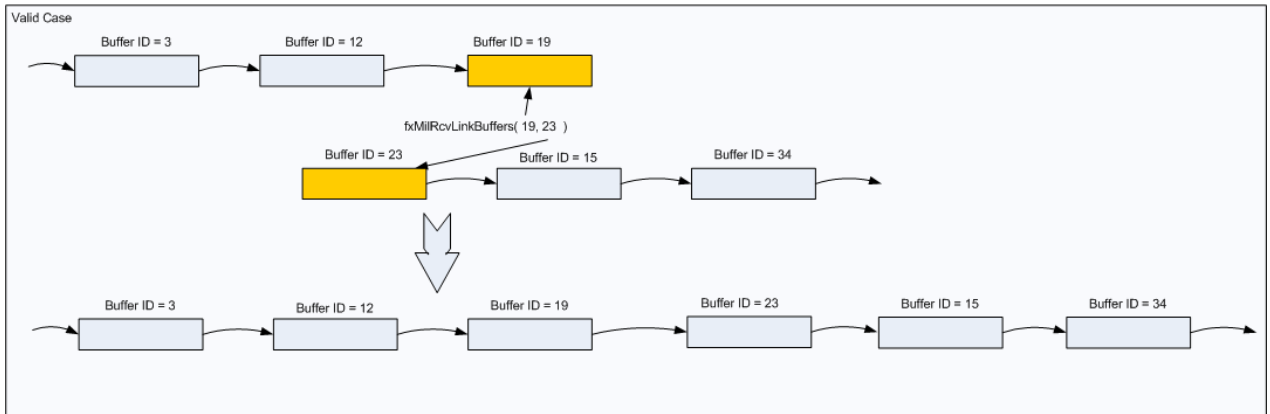
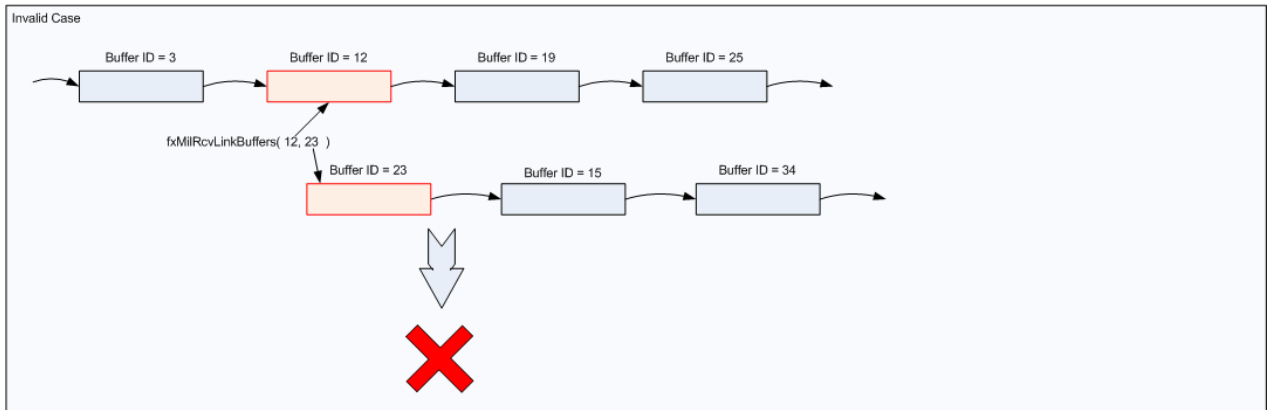
For each buffer the user may set buffer options. However, please note that options need to be the same for all buffers in the same list.

Multiple lists can be setup in memory and when starting a context for reception, a specific buffer can be used as starting point for storing the packets. Please also note that adding the same buffer to more than one list yields unspecified results.

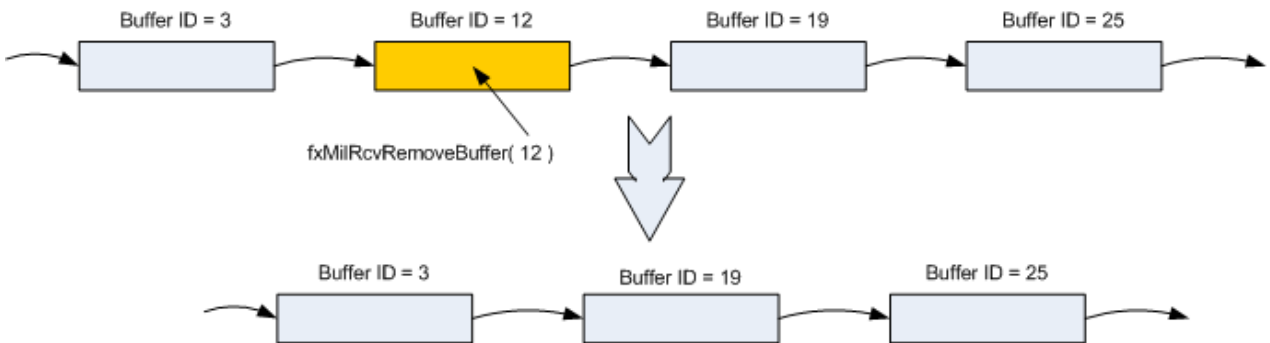
Please refer to [Receive Packet Format](#) for a detailed specification of the received data format.



Add a new buffer to existing list - previous buffer must be the last one of a list



Link two lists of buffers - "From" must be the last one of a list, and "To" must be the first one of another list



Remove a buffer from the list

9.2.2.3.1 fxMiiRcvAddBuffer

Description

This function can be used to setup buffer lists that can be used by contexts to receive packets in. If `prevBufferID` is set, `prevBuffer`'s descriptor will point forward to the new buffer's descriptor. If `prevBufferID` is set (forming a list) then the options must be the same as for previous buffer. Any memory location used as buffer needs to be allocated with the [fxMemAlloc](#).

Parameters

<code>busHandle</code>	Reference handle to the bus to control. (see fxCreateBusHandle)
<code>prevBufferID</code>	-1 specifies first buffer in list.
<code>buffer</code>	User-specified buffer address. The buffer needs to be allocated with fxMemAlloc ().

size	Buffer size in quadlets. Each buffer must be at least big enough to hold one maximum-sized packet according to 1394 spec. Maximum size that can be specified is 16383 quadlets (65532 bytes); Otherwise, FX_ERR_MIL_RCV_BUFFER_TOO_LARGE error will be returned.
options	Pointer to FXMilRcvBufferOptions .
newBufferID	Filled with a new, valid bufferID if no error is reported.

Return Codes

This function returns zero or greater upon success. A negative return value indicates an error. Please use the function [fxGetErrorMessage](#) to lookup descriptions corresponding to negative return values.

FX_ERR_INVALID_HANDLE
FX_ERR_MIL_RCV_BUFFER_NOT_FOUND
FX_ERR_MIL_RCV_BUFFER_OPTIONS_NO_MATCH
FX_ERR_MIL_RCV_BUFFER_ALLOCATION_FAIL
FX_ERR_MIL_RCV_BUFFER_NOT_LAST_OF_LIST
FX_ERR_MIL_RCV_BUFFER_TOO_LARGE

Synopsis

```
FXReturnCode fxMilRcvAddBuffer(
    FXBusHandle          busHandle,
    int32_t              prevBufferID,
    uint32_t*            buffer,
    uint32_t              size,
    const FXMilRcvBufferOptions* options,
    uint32_t*            newBufferID
)
```

9.2.2.3.2 fxMilRcvLinkBuffers

Description

This function can be used to link two buffer lists. It is not allowed to link a buffer list in a loop fashion.

Parameters

busHandle	Reference handle to the bus to control. (see fxCreateBusHandle)
fromBufferID	toBufferID below will be appended to this bufferID.
toBufferID	This bufferID will be appended to fromBufferID above.

Return Codes

This function returns zero or greater upon success. A negative return value indicates an error. Please use the function [fxGetErrorMessage](#) to lookup descriptions corresponding to negative return values.

FX_ERR_INVALID_HANDLE
FX_ERR_MIL_RCV_BUFFER_NOT_FOUND
FX_ERR_MIL_RCV_BUFFER_OPTIONS_NO_MATCH
FX_ERR_MIL_RCV_BUFFER_NOT_FIRST_OF_LIST
FX_ERR_MIL_RCV_BUFFER_NOT_LAST_OF_LIST
FX_ERR_MIL_RCV_BUFFER_LINK_SINGLE
FX_ERR_MIL_RCV_INTERNAL_ERROR

Synopsis

```
FXReturnCode fxMilRcvLinkBuffers(
    FXBusHandle          busHandle,
    uint32_t              fromBufferID,
```

```

        uint32_t          toBufferID
    )

```

9.2.2.3.3 fxMilRcvRemoveBuffer

Description

This function can be used to remove a buffer from a buffer list and reclaim ownership of its memory location.

Parameters

busHandle	Reference handle to the bus to control. (see fxCreateBusHandle)
bufferID	Buffer to remove from the list

Return Codes

This function returns zero or greater upon success. A negative return value indicates an error. Please use the function [fxGetErrorMessage](#) to lookup descriptions corresponding to negative return values.

FX_ERR_INVALID_HANDLE
FX_ERR_MIL_RCV_BUFFER_NOT_FOUND
FX_ERR_MIL_RCV_INTERNAL_ERROR

Synopsis

```

FXReturnCode fxMilRcvRemoveBuffer(
    FXBusHandle      busHandle,
    uint32_t         bufferID
)

```

9.2.2.3.4 fxMilRcvBufferStatus

Description

This function returns the status of the specified buffer.

Parameters

contextHandle	Context Handle the buffer corresponds to.
bufferID	The ID of buffer.
status	Pointer to FXMilRcvBufferStatus .
updateReadStatus	If set to TRUE, the FireStack will update the curReadOffset of the specified buffer ID for the next function call and it will update the curReadBufferID of the specified context. If set to FALSE it will not update anything.

Return Codes

This function returns zero or greater upon success. A negative return value indicates an error. Please use the function [fxGetErrorMessage](#) to lookup descriptions corresponding to negative return values.

FX_ERR_INVALID_HANDLE
FX_ERR_INVALID_PARAMETER
FX_ERR_MIL_RCV_INVALID_CONTEXT_HANDLE
FX_ERR_MIL_RCV_BUFFER_NOT_FOUND

Synopsis

```

FXReturnCode fxMilRcvBufferStatus(
    FXMilRcvContextHandle contextHandle,
    uint32_t             bufferID,
    FXMilRcvBufferStatus* status,
    bool_t               updateReadStatus
)

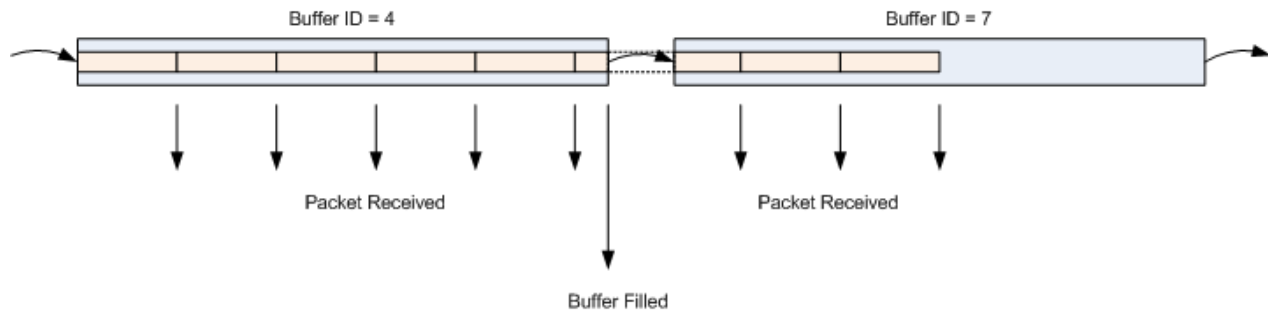
```

)

9.2.2.4. Context Control

The user may use up to 8 reception contexts. Each reception context can be controlled individually and the message filter system determines which filters will deliver packets in which context. One important aspect of a context's behavior is the way it notifies the user when new data is available.

The user may register zero to two callback functions ([fxMilRcvSetContextOptions](#)). One that will be called if new packets are available and the other one will be called if one or more buffers have been filled. The next diagram shows two buffers that have been filled with some packets. The diagram indicates at which moments in time the user callback functions would be called assuming that the callback functions would return immediately (before the next event comes in).



If no callback function is set the user is supposed to periodically call `fxMilRcvContextStatus` to retrieve the buffer ID the hardware is currently writing to. This information can then be used to process all packets up-to the write offset within the buffer the hardware is currently writing to, which can be retrieved by using `fxMilRcvBufferStatus`. Without a callback function set, status always reflects the actual hardware status. Although this is a supported way of operation, it is recommended to work through the callback mechanisms described below rather than continuously polling hardware status.

If one callback function is set then the user is supposed to request current status information from within that callback function by calling `fxMilRcvContextStatus` and `fxMilRcvBufferStatus` setting the `updateReadStatus` flag to true. After the user callback function returns it will only be called again if more data is available beyond the last `writeOffset` returned by `fxMilRcvBufferStatus`.

If at least one callback function is set then both the context and the buffer status functions will reflect the hardware status at the moment right before the callback function was called. The status reported to the user will not change during user callback execution.

If the user callback function returns without calling [fxMilRcvContextStatus](#) and/or without calling the function [fxMilRcvBufferStatus](#) for the current write buffer ID as returned by [fxMilRcvContextStatus](#), it will be called again after returning because in that case the internally stored last position returned to the user still did not catch up to the current write position.

If two callback functions are set then at least one of the two callback functions shall behave as described for a single callback function. The second callback function in that case does not have to call `fxMilRcvContextStatus` or `fxMilRcvBufferStatus`.

9.2.2.4.1 fxMilRcvCreateContextHandle

Description

This function can be used to acquire a receive context handle. The receive context handle is required when functions and data structures that have an access to a receive context.

Parameters

<code>busHandle</code>	Reference handle to the bus to control. (see fxCreateBusHandle)
------------------------	---

contextHandle	Pointer to FXMilRcvContextHandle .
---------------	--

Return Codes

This function returns zero or greater upon success. A negative return value indicates an error. Please use the function [fxGetErrorMessage](#) to lookup descriptions corresponding to negative return values.

FX_ERR_INVALID_HANDLE
FX_ERR_MIL_RCV_NO_AVAILABLE_CONTEXT

Synopsis

```
FXReturnCode fxMilRcvCreateContextHandle(
    FXBusHandle          busHandle,
    FXMilRcvContextHandle* contextHandle
)
```

9.2.2.4.2 fxMilRcvCloseContextHandle

Description

This function frees resources used by the specified receive context handle.

Parameters

contextHandle	Handle created by fxMilRcvCreateContextHandle .
---------------	---

Return Codes

This function returns zero or greater upon success. A negative return value indicates an error. Please use the function [fxGetErrorMessage](#) to lookup descriptions corresponding to negative return values.

FX_ERR_MIL_RCV_INVALID_CONTEXT_HANDLE
FX_ERR_MIL_RCV_CONTEXT_ALREADY_CLOSED

Synopsis

```
FXReturnCode fxMilRcvCloseContextHandle(
    FXMilRcvContextHandle contextHandle
)
```

9.2.2.4.3 fxMilRcvSetContextOptions

Description

This function sets options for a receive context. These options define the behavior of a context during reception.

Parameters

contextHandle	Context Handle that a user wants to set options to.
options	Pointer to FXMilRcvContextOptions .

Return Codes

This function returns zero or greater upon success. A negative return value indicates an error. Please use the function [fxGetErrorMessage](#) to lookup descriptions corresponding to negative return values.

FX_ERR_INVALID_HANDLE
FX_ERR_INVALID_PARAMETER
FX_ERR_MIL_RCV_INVALID_CONTEXT_HANDLE

Synopsis

```
FXReturnCode fxMilRcvSetContextOptions(
    FXMilRcvContextHandle contextHandle,
    FXMilRcvContextOptions* options
)
```

9.2.2.4.4 fxMilRcvStartContext

Description

This function starts data reception with specific context and buffer.

Parameters

contextHandle	Context Handle that a user wants to start data reception with.
bufferID	Buffer ID that a user wants to receive data.

Return Codes

This function returns zero or greater upon success. A negative return value indicates an error. Please use the function [fxGetErrorMessage](#) to lookup descriptions corresponding to negative return values.

FX_ERR_INVALID_HANDLE
FX_ERR_MIL_RCV_INVALID_CONTEXT_HANDLE
FX_ERR_MIL_RCV_BUFFER_NOT_FOUND
FX_ERR_MIL_RCV_INTERNAL_ERROR

Synopsis

```
FXReturnCode fxMilRcvStartContext(
    FXMilRcvContextHandle    contextHandle,
    uint32_t                bufferID
)
```

9.2.2.4.5 fxMilRcvStopContext

Description

This function stops data reception on the requested context after completion of a packet currently in progress.

Parameters

contextHandle	Context Handle that a user wants to stop data reception.
---------------	--

Return Codes

This function returns zero or greater upon success. A negative return value indicates an error. Please use the function [fxGetErrorMessage](#) to lookup descriptions corresponding to negative return values.

FX_ERR_INVALID_HANDLE
FX_ERR_MIL_RCV_INVALID_CONTEXT_HANDLE
FX_ERR_MIL_RCV_INTERNAL_ERROR

Synopsis

```
FXReturnCode fxMilRcvStopContext(
    FXMilRcvContextHandle    contextHandle
)
```

9.2.2.4.6 fxMilRcvContextStatus

Description

This function obtains the status of specific context.

Parameters

contextHandle	Context Handle that a user wants to get status for.
status	Pointer to FXMilRcvContextStatus .

Return Codes

This function returns zero or greater upon success. A negative return value indicates an error. Please use the function [fxGetErrorMessage](#) to lookup descriptions corresponding to negative return values.


```

FX_ERR_INVALID_HANDLE
FX_ERR_INVALID_PARAMETER
FX_ERR_MIL_RCV_INVALID_CONTEXT_HANDLE
FX_ERR_MIL_RCV_INTERNAL_ERROR

```

Synopsis

```

FXReturnCode fxMilRcvContextStatus(
    FXMilRcvContextHandle          contextHandle,
    FXMilRcvContextStatus\*       status
)

```

9.2.3. Type Definitions**9.2.3.1. FXMilRcvContextHandle****Description**

Handle to a receive context created by [fxMilRcvCreateContextHandle](#) function.

Synopsis

```

typedef uint32_t FXMilRcvContextHandle;

```

9.2.3.2. FXMilRcvCallback**Description**

This function definition is used to specify a callback function that can be used as

- buffer full callback
- packet received callback

as specified in the [context options](#).

Parameters

handle	Reference handle to the bus to control. (see fxCreateBusHandle)
userData	Pointer to the data specified in FXMilRcvEventOptions .
contextHandle	Context Handle that is associated with the callback occurrence.

Synopsis

```

typedef void (*FXMilRcvCallback) {
    FXBusHandle          handle,
    void*                userData,
    FXMilRcvContextHandle contextHandle
};

```

9.2.4. Structures**9.2.4.1. FXMilRcvFilterItem****Description**

This structure defines data members that will be used for the message filtering. A received packet with matching channel number and messageID will be copied to the specified context on reception.

Members

anyMessage	If true, FireStack will not check messageID for filtering, and messageID member will have no effect.
messageID	MessageID of AS5643 data.
anyChannel	If true, FireStack will not check channel number for filtering, and channelNumber member will have no effect.
channelNumber	Channel number of a stream packet.

contextHandle	Matched packet will be copied to data buffer associated with this contextHandle.
---------------	--

Synopsis

```
typedef struct {
    bool_t          anyMessage;
    uint32_t        messageID;
    bool_t          anyChannel;
    uint32_t        channelNumber;
    FXMilRcvContextHandle contextHandle;
} FXMilRcvFilterItem;
```

9.2.4.2. FXMilRcvBufferOptions**Description**

This structure defines options used for when a new receive buffer is created.

Members

bufferMode	Currently, only buffer-fill (0) mode is supported. Buffer-fill mode means that packets are stored back to back in the buffer until the end of the buffer is reached. Packets that don't fit in a buffer will straddle across buffers in this mode. All buffers in the same list must have this setting identical.
callBackEnabled	If set to 1 (one), a callback function will be called when the buffer is filled with packets and/or when a packet is received. (Depending on context options)

Synopsis

```
typedef struct {
    uint32_t    bufferMode;
    uint32_t    callBackEnabled;
} FXMilRcvBufferOptions;
```

9.2.4.3. FXMilRcvBufferStatus**Description**

This structure defines data fields indicating the current status of the data buffer.

Members

statusCode	Current status of the buffer: = 1: Running - Buffer is currently being or will be written to. = 0: Stopped - Buffer is filled. < 0: Error - An error has occurred while the buffer was updated. Holds an error code (TBD)
extendedStatusFlag	Bit flag of extra status information. Currently only one bit is defined. See Buffer Status Extended Status Bits for more information.
bufferSize	Same as the size parameter in fxMilRcvAddBuffer() .
bufferPtr	Pointer to the first quadlet of the user-provided buffer memory.
curWriteOffset	Quadlet offset within buffer memory the Link Layer will write the next data to. This field is updated after a packet has been received and/or after a buffer has been completely

	filled.
curReadOffset	Quadlet offset within buffer memory the user should continue reading received data. Whenever the user makes a function call to fxMilRcvBufferStatus function, the current value will be copied into the user-provided structure. After that the value of curWriteOffset will be copied to the internally maintained curReadOffset for the next buffer status request.
nextBufferID	The ID of the buffer that will be used after this one is filled. A value of 0 means this buffer is the last one in the list.

Synopsis

```
typedef struct {
    int32_t      statusCode;
    uint32_t     extendedStatusFlag;
    uint32_t     bufferSize;
    uint32_t*    bufferPtr;
    uint32_t     curWriteOffset;
    uint32_t     curReadOffset;
    uint32_t     nextBufferID;
} FXMilRcvBufferStatus;
```

9.2.4.4. FXMilRcvEventOptions**Description**

This structure defines options for a receive event.

Members

callback	Specify callback function pointer or zero to clear. (see FXMilRcvCallback)
userData	Pointer to a user-specified data. The pointer will be carried to user callback functions. (see FXMilRcvCallback)

Synopsis

```
typedef struct {
    FXMilRcvCallback callback,
    void*             userData
} FXMilRcvEventOptions;
```

9.2.4.5. FXMilRcvContextOptions**Description**

This structure defines options for a receive context.

Members

bufferFullOptions	Specify options for buffer-full event. (see FXMilRcvEventOptions)
packetRcvOptions	Specify options for packet-receive event. (see FXMilRcvEventOptions)

Synopsis

```
typedef struct {
    FXMilRcvEventOptions    bufferFullOptions,
    FXMilRcvEventOptions    packetRcvOptions
} FXMilRcvContextOptions;
```

9.2.4.6. FXMilRcvContextStatus

Description

This structure defines data members used for status inquiry for a receive context.

Members

statusCode	Current status of the context: = 1: Running - Context is currently actively receiving packets = 0: Stopped - Context has not yet started, stopped by fxMilRcvStopContext , or has reached the end of receive buffer < 0: Error - Context has encountered an error. Holds an error code (TBD)
curWriteBufferID	This field contains the buffer ID that is currently being written to. This basically means that any packet in progress, or the next packet if no packet is in progress, will be written to this buffer ID. This field will be updated each time the hardware has completely filled a buffer.
curReadBufferID	This field contains the buffer ID the user is supposed to continue processing data. This field is automatically updated by the FireStack whenever fxMilRcvBufferStatus is called.

Synopsis

```
typedef struct {
    int32_t    statusCode;
    uint32_t   curWriteBufferID;
    uint32_t   curReadBufferID;
} FXMilRcvContextStatus;
```

9.2.5. Constants

9.2.5.1. Error Codes

The following values may be returned by AS5643 Protocol functions.

```
FX_ERR_MIL_RCV_FILTER_ITEM_NOT_FOUND
FX_ERR_MIL_RCV_INVALID_FILTER_ATTRIBUTES
FX_ERR_MIL_RCV_INVALID_CONTEXT_HANDLE
FX_ERR_MIL_RCV_FILTER_DOUBLE_MATCH
FX_ERR_MIL_RCV_BUFFER_OPTIONS_NO_MATCH
FX_ERR_MIL_RCV_BUFFER_NOT_FOUND
FX_ERR_MIL_RCV_BUFFER_ALLOCATION_FAIL
FX_ERR_MIL_RCV_BUFFER_NOT_FIRST_OF_LIST
FX_ERR_MIL_RCV_BUFFER_NOT_LAST_OF_LIST
FX_ERR_MIL_RCV_BUFFER_LINK_SINGLE
FX_ERR_MIL_RCV_INTERNAL_ERROR
FX_ERR_MIL_RCV_INSUFFICIENT_FILTER_TABLE_SPACE
FX_ERR_MIL_RCV_BUFFER_TOO_LARGE
FX_ERR_MIL_RCV_INVALID_BUFFER_ADDRESS
FX_ERR_MIL_RCV_NO_AVAILABLE_CONTEXT
FX_ERR_MIL_RCV_CONTEXT_ALREADY_CLOSED
```

9.2.5.2. Buffer Status Extended Staus Bits

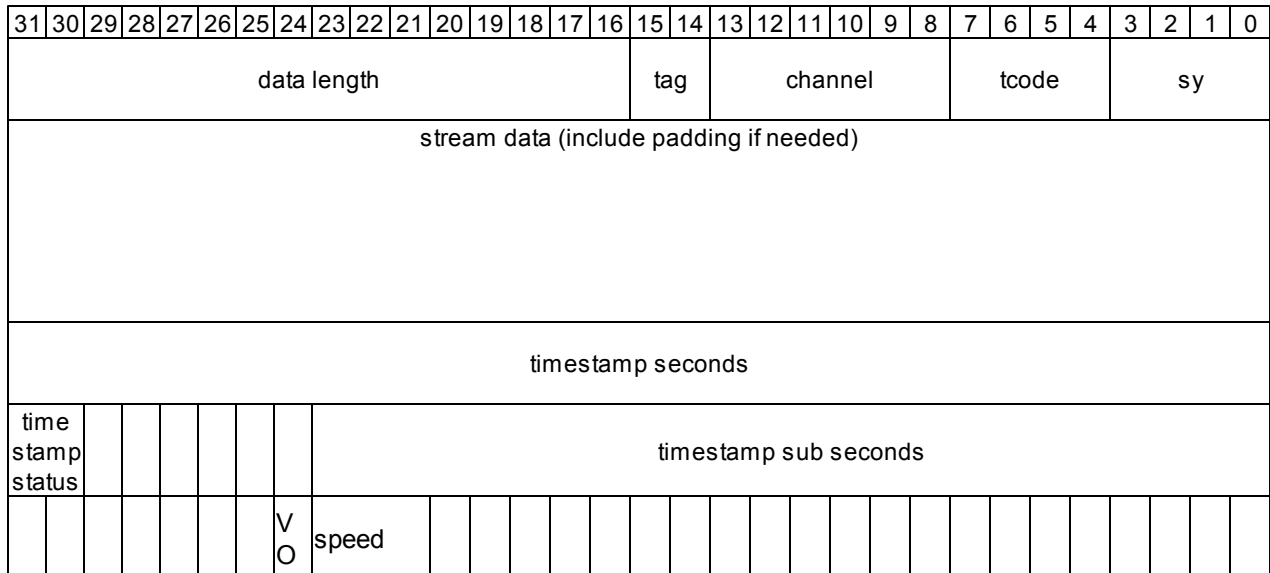
Name	Value	Description
------	-------	-------------

FX_MIL_RCV_BUFFER_EXT_STATUS_CNTX_STOPPED	0x00010000	If the associated context has stopped when this buffer is added, this bit set to 1.
---	------------	---

9.2.6. Data Formats

9.2.6.1. Receive Packet Format

The format of a received packet.



field	bits	description
data length	16	Number of bytes of stream data in this packet.
tag	2	The data format of this stream data.
channel	6	The channel number this packet is associated with.
tcode	4	The transaction code (should always be 0xA)
sy	4	Synchronization control field.
stream data		The data received with this packet. The last quadlet will be padded with zeroes, if necessary.
timestamp seconds	32	Time stamp seconds. Offset depends on time source
timestamp sub seconds	24	Time stamp sub-second value. Resolution of 1 micro second.
timestamp status	2	Status of the time source: 0 = Invalid 1 = Free Running 2 = Synced (time output is synchronized to the IRIG input and becoming more accurate over time) 3 = Accurate (time output is synchronized to the IRIG input with maximum accuracy)
speed*	3	Speed Code: 0 = 100Mb/s 1 = 200Mb/s 2 = 400Mb/s 3 = 800Mb/s
VO	1	VPC OK - indicates the status of the VPC (Vertical Parity Check): 0 = VPC is not presented in the received packet or is invalid 1 = VPC is presented in the received packet and is valid

* speed field will likely move to bits 31,30,29 in the future

9.3. AS5643 Transmission

FireStack AS5643 Transmission module can be used to control devices that support AS5643 timed transmission in hardware like DapTechnology's FireTrac and FireLink Extended. FireTrac offers very accurate transmission timing without software intervention enabling this functionality without the need for a Real-Time operating system. Accurate transmission timing can not only be used for transmitting packets at the correct scheduled transmission time, it also enables adding semi random jitter to the transmission timing in a very controlled manner. Please refer to [Context Options](#) for more information.

FireStack AS5643 Transmission module is split-up in 8 individual transmission contexts. In order to transmit any AS5643 messages the user first needs to acquire a context handle and choose the mode the context will operate in. Several context modes have been defined, each having its own function interface that is well chosen to control that specific mode as simple as possible.

The following transmission modes are available:

Streaming Messages	This mode allows the user to write large or small sets of messages to the FireStack and have them transmitted automatically on the specified frame offset times. The provided data needs to contain so called frame separator elements to indicate the following message needs to be transmitted in the next frame. (please refer to Streaming Messages Mode for more information)
Repeating Messages	This mode allows the user to setup a message that will automatically be transmitted each frame by the FireStack. The user will have a pointer to the actual data of the message and is allowed to manipulate the data at any point in time without having to worry about its timed transmission. Very useful for AS5643 status messages. (please refer to Repeating Messages Mode for more information)
Single Message	This mode allows the user to simply transmit a message as soon as possible but exactly at the specified frame offset time. Several messages may be handed to the FireStack for immediate transmission and the FireStack will then take care of the actual moment of transmission. (please refer to Single Message Mode for more information)
STOF Message	This mode allows the user to control transmission of STOF messages. (please refer to STOF Message Mode for more information)

9.3.1. Settings

9.3.1.1. Resource Usage

The following settings can be used as settingId in an [FXSetting](#) instance passed to [fxCreateBusHandle\(\)](#) to control resource usage by the AS5643 Reception module for the bus that is being opened.

FX_SETTING_ID_MIL_TRM_MAX_BUFFERS

This setting determines the maximum number of packets that may reside in total in all of the AS5643 Transmission contexts combined at any given point in time.

Default: 10000

9.3.2. Functions

9.3.2.1. Context Management

9.3.2.1.1 fxMilTrmCreateContextHandle

Description

In order for a user to transmit messages he needs to acquire a handle to a AS5643 transmission context and set it in a specific mode. This function can be used to acquire a context handle and choose a mode to operate in. **Please note that only one context with the FX_MIL_TRM_MODE_STOF mode can be created per one bus handle.**

For more advanced control over the way a context behaves, please use the function `fxMilTrmCreateContextHandle()` instead. This function allows, for example, applying semi random jitter to the transmission times of messages to be transmitted.

Parameters

<code>busHandle</code>	Reference handle to the bus to control. (see fxCreateBusHandle)
<code>mode</code>	Controls the transmission mode the context will be opened in. (see Context Modes)
<code>contextHandle</code>	Pointer to user-allocated variable that will return a handle to a context upon success.

Return Codes

This function returns zero or greater upon success. A negative return value indicates an error. Please use the function [fxGetErrorMessage](#) to look up descriptions corresponding to negative return values.

FX_ERR_INVALID_HANDLE
FX_ERR_MIL_TRM_NO_AVAILABLE_CONTEXT
FX_ERR_MIL_TRM_INVALID_CONTEXT_MODE
FX_ERR_MIL_TRM_INTERNAL_ERROR
FX_ERR_LICENSE_MODULE

Synopsis

```
FXReturnCode fxMilTrmCreateContextHandle(
    FXBusHandle          busHandle,
    uint32_t           mode,
    FXMilTrmContextHandle\* contextHandle
)
```

9.3.2.1.2 fxMilTrmCreateContextHandleExt

Description

In order for a user to transmit messages he needs to acquire a handle to a AS5643 transmission context and set it in a specific mode. This function can be used to acquire a context handle and choose a mode to operate in. **Please note that only one context with the `FX_MIL_TRM_MODE_STOF` mode can be created per one bus handle.**

Parameters

<code>busHandle</code>	Reference handle to the bus to control. (see fxCreateBusHandle)
<code>optionList</code>	Pointer to a user defined list of FXMilTrmContextOption elements that together form the options for the context to be created. For available options and their default values, please refer to Context Options . Options that are not specified will just use default values.
<code>listSize</code>	Specifies the number of items in the optionList.
<code>contextHandle</code>	Pointer to user-allocated variable that will return a handle to a context upon success.

Return Codes

This function returns zero or greater upon success. A negative return value indicates an error. Please use the function [fxGetErrorMessage](#) to look up descriptions corresponding to negative return values.

FX_ERR_INVALID_HANDLE
FX_ERR_MIL_TRM_NO_AVAILABLE_CONTEXT
FX_ERR_MIL_TRM_INVALID_CONTEXT_MODE
FX_ERR_MIL_TRM_INTERNAL_ERROR
FX_ERR_LICENSE_MODULE

Synopsis

```

FXReturnCode fxMilTrmCreateContextHandle(
    FXBusHandle          busHandle,
    FXMilTrmContextOption\* optionList,
    size_t             listSize,
    FXMilTrmContextHandle\* contextHandle
)

```

9.3.2.1.3 fxMilTrmCloseContextHandle

Description

This function can be used to release an AS5643 transmission context. Please make sure to release corresponding resources like message handles. This function immediately stops the context if it is running.

Parameters

handle	Reference handle to the context to control. (see fxMilTrmCreateContextHandle)
--------	---

Return Codes

This function returns zero or greater upon success. A negative return value indicates an error. Please use the function [fxGetErrorMessage](#) to look up descriptions corresponding to negative return values.

FX_ERR_MIL_TRM_INVALID_CONTEXT_HANDLE
FX_ERR_MIL_TRM_INTERNAL_ERROR
FX_ERR_LICENSE_MODULE

Synopsis

```

FXReturnCode fxMilTrmCloseContextHandle(
    FXMilTrmContextHandle handle
)

```

9.3.2.2. Single Message Mode

Sometimes it is useful to have a simple interface to just send out a message once as soon as possible but taking into account its frame offset time. This can be done by claiming a context in single message mode and then use one of the following functions.

9.3.2.2.1 fxMilTrmMessage

Description

This function can be used to schedule a message for transmission at a specific transmit offset time but as soon as possible. The function will just hand the message to the FireStack and then returns right away. The FireStack will then take care of actually transmitting it in one of the next frames at its frame offset time.

Parameters

handle	Reference handle to the context to control. The context handle must be created with FX_MIL_TRM_MODE_SINGLE as context mode. (see fxMilTrmCreateContextHandle)
optionList	Pointer to a user defined list of FXMilTrmMessageOption elements that together form the options for the message to be transmitted. For available options and their default values, please refer to Message Options . Options that are not specified will just use default values.
listSize	Specifies the number of items in the optionList.
data	Pointer to a DMA-capable buffer allocated by the user. The buffer will be used as message data including the VPC field but excluding the data CRC. (see fxMemAlloc for allocating DMA-capable buffers)
dataSize	Size of data in bytes.
callback	Specifies a user-defined callback function that will be called after specified message has been transmitted or if an error occurred. A value of zero indicates that no callback

	is needed.
userdata	This is a convenience feature that allows caller to specify arbitrary user data that fits in a variable of type void*. FireStack will not touch what is provided here. The exact value provided will be handed back to the user when the callback function is called.

Return Codes

This function returns zero or greater upon success. A negative return value indicates an error. Please use the function [fxGetErrorMessage](#) to look up descriptions corresponding to negative return values.

```

FX_ERR_FIRESTACK_DEMO_TIMEOUT
FX_ERR_FIRESTACK_NOT_INITIALIZED
FX_ERR_INVALID_PARAMETER
FX_ERR_INVALID_ADDRESS
FX_ERR_MIL_TRM_INTERNAL_ERROR
FX_ERR_MIL_TRM_INVALID_BUFFER_ADDRESS
FX_ERR_MIL_TRM_INVALID_CONTEXT_HANDLE
FX_ERR_MIL_TRM_INVALID_CONTEXT_MODE
FX_ERR_LICENSE_DEVICE
FX_ERR_LICENSE_MODULE
FX_ERR_MIL_TRM_OUT_OF_INTERNAL_RESOURCE
FX_ERR_MODULE_NOT_ENABLED

```

Synopsis

```

FXReturnCode fxMilTrmMessage(
    FXMilTrmContextHandle          handle,
    const FXMilTrmMessageOption\* optionList,
    size_t                          listSize,
    const void*                      data,
    size_t                          dataSize,
    FXMilTrmCallback              callback,
    void*                            userData
)

```

9.3.2.2.2 fxMilTrmSplitMessage

Description

This function does the same as [fxMilTrmMessage](#) except that the message data is specified by a list of data buffers rather than just one. Even though hardware DMA will be used to copy the data from the buffer locations into the transmission FIFO, this function is less efficient than [fxMilTrmMessage](#). Please be aware of possible performance degradation when using this function.

Parameters

handle	Reference handle to the context to control. The context handle must be created with FX_MIL_TRM_MODE_SINGLE as context mode. (see fxMilTrmCreateContextHandle)
optionList	Pointer to a user defined list of FXMilTrmMessageOption elements that together form the options for the message to be transmitted. For available options and their default values, please refer to Message Options . Options that are not specified will just use default values.
listSize	Specifies the number of items in the optionList.
bufferList	Ordered list of FXBuffer elements that together form the message data including VPC but excluding CRC. If less than 5 buffers are needed then just set the size of the last buffer(s) to zero.
callback	Specifies a user-defined callback function that will be called after specified message has been transmitted or if an error occurred. A value of zero indicates that no callback is needed.
userdata	This is a convenience feature that allows caller to specify arbitrary user data that fits in

	a variable of type void*. FireStack will not touch what is provided here. The exact value provided will be handed back to the user when the callback function is called.
--	--

Return Codes

This function returns zero or greater upon success. A negative return value indicates an error. Please use the function [fxGetErrorMessage](#) to look up descriptions corresponding to negative return values.

```

FX_ERR_FIRESTACK_DEMO_TIMEOUT
FX_ERR_FIRESTACK_NOT_INITIALIZED
FX_ERR_INVALID_PARAMETER
FX_ERR_INVALID_ADDRESS
FX_ERR_LICENSE_DEVICE
FX_ERR_LICENSE_MODULE
FX_ERR_MIL_TRM_INTERNAL_ERROR
FX_ERR_MIL_TRM_INVALID_CONTEXT_HANDLE
FX_ERR_MIL_TRM_INVALID_CONTEXT_MODE
FX_ERR_MIL_TRM_OUT_OF_INTERNAL_RESOURCE
FX_ERR_MODULE_NOT_ENABLED

```

Synopsis

```

FXReturnCode fxMilTrmSplitMessage(
    FXMilTrmContextHandle    handle,
    const FXMilTrmMessageOption* optionList,
    size_t                   listSize,
    FXBuffer                 bufferList[5],
    FXMilTrmCallback         callback,
    void*                    userdata
)

```

9.3.2.3. Streaming Messages Mode

In streaming mode the user can append large (or small) numbers of messages to a context. The user is required to supply the messages ordered on transmit offset time and needs to insert frame separator elements whenever the following frame should start.

Message data needs to be placed in DMA-capable memory buffers allocated with the FireStack function [fxMemAlloc](#) just like the reception buffers. FireStack will offer a zero-copy transmit API for messages in such buffers. FireStack will use hardware DMA to place the data buffers directly in the transmit FiFo without software intervention.

A flexible interface is available for appending the messages. Whenever a set of messages is appended options may be specified that apply to the complete set and overruling options may be specified per individual message.

The user may choose from several different ways of adding messages to the streaming mode context. It is allowed to mix the three functions for the same context. Each way of adding messages allows the user to register a callback function that will be called upon completion of the messages provided.

9.3.2.3.1 fxMilTrmStrmWriteImmediate

Description

After a context handle has been created in [FX_MIL_TRM_MODE_STREAMING](#) mode this function can be used to append messages to the transmission queue. This function may be called before the context is started or when it is already running. Using this function a set of messages can be appended at once, optionally including frame separator items to prepare a complete message stream ahead of time. This function may be mixed with function calls to other [fxMilTrmStreamWrite..](#) functions.

This function should be used when all messages to be transmitted are in one large data buffer including frame separator elements.

Parameters

handle	Reference handle to the context to control. The context handle must be created with FX_MIL_TRM_MODE_STREAMING as context mode. (see fxMilTrmCreateContextHandle)
data	Data points to one large buffer containing all the messages back-to-back optionally with a list of options per message. Data format may be one of the formats defined in section Data Formats and can be selected by specifying the data format option in the optionList. Data must point to DMA-capable memory. All messages will use the options resulting from defaults and optionList unless if the message itself contains overriding options. (see fxMemAlloc for allocating DMA-capable buffers)
dataSize	Size of data in bytes.
callback	Specifies a user-defined callback function that will be called after all specified messages have been transmitted or if an error occurred. A value of zero indicates that no callback is needed.
userData	This is a convenience feature that allows caller to specify arbitrary user data that fits in a variable of type void*. FireStack will not touch what is provided here. The exact value provided will be handed back to the user when the callback function is called.

Return Codes

This function returns zero or greater upon success. A negative return value indicates an error. Please use the function [fxGetErrorMessage](#) to look up descriptions corresponding to negative return values.

FX_ERR_INVALID_PARAMETER
FX_ERR_INVALID_ADDRESS
FX_ERR_MIL_TRM_INVALID_BUFFER_ADDRESS
FX_ERR_MIL_TRM_INVALID_CONTEXT_HANDLE
FX_ERR_MIL_TRM_INVALID_CONTEXT_MODE
FX_ERR_MIL_TRM_STRM_DATA_FORMAT_ERROR
FX_ERR_MIL_TRM_INTERNAL_ERROR
FX_ERR_LICENSE_MODULE

Synopsis

```

FXReturnCode fxMilTrmStrmWriteImmediate(
    FXMilTrmContextHandle    handle,
    void*                    data,
    size_t                   dataSize,
    FXMilTrmCallback        callback,
    void*                    userData
)

```

9.3.2.3.2 [fxMilTrmStrmWriteMessageList](#)**Description**

After a context handle has been created in [FX_MIL_TRM_MODE_STREAMING](#) mode this function can be used to append messages to the transmission queue. This function may be called before the context is started or when it is already running. Using this function a set of messages can be appended at once, optionally including frame separator items to prepare a complete message stream ahead of time. This function may be mixed with function calls to other [fxMilTrmStreamWrite..](#) functions.

This function should be used when the data for each message is located in its own single buffer. A list containing [FXMilTrmMessage](#) items needs to be created by the user, each item pointing to a data buffer and containing message options.

Parameters

handle	Reference handle to the context to control. The context handle must be created with FX_MIL_TRM_MODE_STREAMING as context mode.
--------	--

	(see fxMilTrmCreateContextHandle)
optionList	Pointer to a user defined list of FXMilTrmMessageOption elements that together form the options for all messages to be transmitted. For available options and their default values, please refer to Message Options . Options that are not specified will just use default values. Options specified per message will override what is specified here.
optionListSize	Specifies the number of items in the optionList.
messageList	List of FXMilTrmMessage items to append to the transmission queue.
messageListSize	Number of items in messageList.
callback	Specifies a user-defined callback function that will be called after all specified messages have been transmitted or if an error occurred. A value of zero indicates that no callback is needed.
userData	This is a convenience feature that allows caller to specify arbitrary user data that fits in a variable of type void*. FireStack will not touch what is provided here. The exact value provided will be handed back to the user when the callback function is called.

Return Codes

This function returns zero or greater upon success. A negative return value indicates an error. Please use the function [fxGetErrorMessage](#) to look up descriptions corresponding to negative return values.

FX_ERR_INVALID_PARAMETER
FX_ERR_INVALID_ADDRESS
FX_ERR_MIL_TRM_INVALID_BUFFER_ADDRESS
FX_ERR_MIL_TRM_INVALID_CONTEXT_HANDLE
FX_ERR_MIL_TRM_INVALID_CONTEXT_MODE
FX_ERR_MIL_TRM_STRM_DATA_FORMAT_ERROR
FX_ERR_MIL_TRM_OUT_OF_INTERNAL_RESOURCE
FX_ERR_MIL_TRM_INTERNAL_ERROR
FX_ERR_LICENSE_MODULE

Synopsis

```

FXReturnCode fxMilTrmStrmWriteMessageList(
    FXMilTrmContextHandle          handle,
    FXMilTrmMessageOption\*       optionList,
    size_t                        optionListSize,
    FXMilTrmMessage\*            messageList,
    size_t                        messageListSize,
    FXMilTrmCallback            callback,
    void*                          userData
)

```

9.3.2.3.3 fxMilTrmStrmWriteSplitMessageList

Description

After a context handle has been created in [FX_MIL_TRM_MODE_STREAMING](#) mode this function can be used to append messages to the transmission queue. This function may be called before the context is started or when it is already running. Using this function a set of messages can be appended at once, optionally including frame separator items to prepare a complete message stream ahead of time. This function may be mixed with function calls to other [fxMilTrmStreamWrite..](#) functions.

This function should be used when the data for each message is located in several buffers. A list containing [FXMilTrmSplitMessage](#) items needs to be created by the user, each item pointing to all buffers that make up the message and each item specifying message options.

Parameters

handle	Reference handle to the context to control. The context handle must be created with FX_MIL_TRM_MODE_STREAMING as context mode. (see fxMilTrmCreateContextHandle)
--------	--

optionList	Pointer to a user defined list of FXMilTrmMessageOption elements that together form the options for all messages to be transmitted. For available options and their default values, please refer to Message Options . Options that are not specified will just use default values. Options specified per message will override what is specified here.
optionListSize	Specifies the number of items in the optionList.
messageList	List of FXMilTrmSplitMessage items to append to the transmission queue.
messageListSize	Number of items in messageList.
callback	Specifies a user-defined callback function that will be called after all specified messages have been transmitted or if an error occurred. A value of zero indicates that no callback is needed.
userData	This is a convenience feature that allows caller to specify arbitrary user data that fits in a variable of type void*. FireStack will not touch what is provided here. The exact value provided will be handed back to the user when the callback function is called.

Return Codes

This function returns zero or greater upon success. A negative return value indicates an error. Please use the function [fxGetErrorMessage](#) to look up descriptions corresponding to negative return values.

FX_ERR_INVALID_PARAMETER
FX_ERR_INVALID_ADDRESS
FX_ERR_MIL_TRM_INVALID_BUFFER_ADDRESS
FX_ERR_MIL_TRM_INVALID_CONTEXT_HANDLE
FX_ERR_MIL_TRM_INVALID_CONTEXT_MODE
FX_ERR_MIL_TRM_STRM_DATA_FORMAT_ERROR
FX_ERR_MIL_TRM_OUT_OF_INTERNAL_RESOURCE
FX_ERR_MIL_TRM_INTERNAL_ERROR
FX_ERR_LICENSE_MODULE

Synopsis

```

FXReturnCode fxMilTrmStrmWriteSplitMessageList (
    FXMilTrmContextHandle          handle,
    FXMilTrmMessageOption*      optionList,
    size_t                          optionListSize,
    FXMilTrmSplitMessage*      messageList,
    size_t                          messageListSize,
    FXMilTrmCallback           callback,
    void*                            userData
)

```

9.3.2.3.4 fxMilTrmStrmStart

Description

This function can be used to start transmission of queued messages. Please make sure to first write at least some messages to the queue. After the context has been started the user is supposed to stay ahead of transmission with appending new messages. If transmission reaches the end of the queue, the context is stopped.

Parameters

handle	Reference handle to the context to control. The context handle must be created with FX_MIL_TRM_MODE_STREAMING as context mode. (see fxMilTrmCreateContextHandle)
frameNumber	Reserved for future use.

Return Codes

This function returns zero or greater upon success. A negative return value indicates an error. Please use the function [fxGetErrorMessage](#) to look up descriptions corresponding to negative return values.

```

FX_ERR_MIL_TRM_INVALID_CONTEXT_MODE
FX_ERR_MIL_TRM_INVALID_CONTEXT_HANDLE
FX_ERR_MIL_TRM_INTERNAL_ERROR
FX_ERR_MIL_TRM_STRM_EMPTY
FX_ERR_MIL_TRM_CONTEXT_ALREADY_STARTED
FX_ERR_LICENSE_MODULE

```

Synopsis

```

FXReturnCode fxMilTrmStrmStart(
    FXMilTrmContextHandle    handle,
    int32_t                  frameNumber
)

```

9.3.2.3.5 fxMilTrmStrmStop

Description

This function can be used to stop transmission of queued messages. The context will remember at what message it stopped and a function call to [fxMilTrmStrmStart](#) will resume transmission where it stopped.

Parameters

handle	Reference handle to the context to control. The context handle must be created with FX_MIL_TRM_MODE_STREAMING as context mode. (see fxMilTrmCreateContextHandle)
--------	---

Return Codes

This function returns zero or greater upon success. A negative return value indicates an error. Please use the function [fxGetErrorMessage](#) to lookup descriptions corresponding to negative return values.

```

FX_ERR_MIL_TRM_INVALID_CONTEXT_MODE
FX_ERR_MIL_TRM_INVALID_CONTEXT_HANDLE
FX_ERR_MIL_TRM_INTERNAL_ERROR
FX_ERR_MIL_TRM_CONTEXT_ALREADY_STOPPED
FX_ERR_MIL_TRM_OUT_OF_INTERNAL_RESOURCE

```

Synopsis

```

FXReturnCode fxMilTrmStrmStop(
    FXMilTrmContextHandle    handle
)

```

9.3.2.3.6 fxMilTrmStrmClear

Description

This function will clear the transmit queue. This function also stops the context if it is running before clearing the queue. After this function is called, the user becomes the owner of all buffers the stream had. They may now safely be freed.

Parameters

handle	Reference handle to the context to control. The context handle must be created with FX_MIL_TRM_MODE_STREAMING as context mode. (see fxMilTrmCreateContextHandle)
--------	---

Return Codes

This function returns zero or greater upon success. A negative return value indicates an error. Please use the function [fxGetErrorMessage](#) to look up descriptions corresponding to negative return values.

```

FX_ERR_MIL_TRM_INVALID_CONTEXT_MODE
FX_ERR_MIL_TRM_INVALID_CONTEXT_HANDLE
FX_ERR_MIL_TRM_INTERNAL_ERROR

```

Synopsis

```
FXReturnCode fxMilTrmStrmClear(
    FXMilTrmContextHandle    handle
)

```

9.3.2.3.7 fxMilTrmStrmGetStatus

Description

This function returns the current status of the specified stream context.

Parameters

handle	Reference handle to the context to control. The context handle must be created with FX_MIL_TRM_MODE_STREAMING as context mode. (see fxMilTrmCreateContextHandle)
status	Pointer to variable of FXMilTrmStrmStatus .

Return Codes

This function returns zero or greater upon success. A negative return value indicates an error. Please use the function [fxGetErrorMessage](#) to look up descriptions corresponding to negative return values.

FX_ERR_MIL_TRM_INVALID_CONTEXT_MODE
FX_ERR_MIL_TRM_INVALID_CONTEXT_HANDLE
FX_ERR_INVALID_PARAMETER

Synopsis

```
FXReturnCode fxMilTrmStrmGetStatus(
    FXMilTrmContextHandle    handle,
    FXMilTrmStrmStatus*     status
)

```

9.3.2.4. Repeating Messages Mode

It is very common for an AS5643 remote node to be required to periodically output status information messages. This could for example be vehicle information like fuel levels, current speed or any other sensor information. FireStack offers a very simple interface for outputting a status message every frame at its transmit offset time without software intervention.

Each status message needs to be setup with some options like transmit offset time, speed, channel number and auto VPC calculation. Then it needs to have a pointer to DMA-capable memory buffer for its message data section. After the message is started it will transmit a 1394 packet each frame and it will use the provided buffer pointer over and over again as message data.

As the user also knows about the buffer pointer he can manipulate that message data at any point in time and whenever the transmit offset time of this message occurs the FireStack will just send out the data that happens to be set at that moment in time.

So let's assume we have a simple status message that contains one signal representing the current speed. The user creates the message, sets its options, writes an initial value of 100Miles/hour to the buffer pointer and initializes the heartbeat value to one. Now the FireStack will start transmitting this message every frame at its frame offset time. If the user doesn't write to the heartbeat field then the receiving node will know that it is receiving the same speed value each frame over and over again and will just ignore all messages after the first one received.

Now the sensor detects that the speed has changed to 150miles/hour. The user application first writes the new speed to the message buffer and then increments the heartbeat. The next time after these write actions the FireTrac will arrive at the frame offset time for this message it will automatically transmit the updated data.

Please be aware that the two write actions the the message buffer are non-atomic, meaning that potentially the message can be sent with the new speed value but the old heartbeat value. For most AS5643 implementations this should not be a problem as the message will be considered stale data as the heartbeat did not change since the last time the message was received by the other node.

In repeating message mode the user can create and free message handles. A message handle allows a user to control that message throughout its lifetime. Messages created this way will automatically be maintained by FireStack. Functions for setting up a repeating message are defined below.

After the user has supplied FireStack with data pointer he is still allowed to write to those memory locations but memory should not be freed until the user has reclaimed ownership of the buffers. Please be aware that writing several quadlets to a memory buffer is not an atomic action and the message could be transmitted in between two quadlet writes. For atomic data updates the complete buffer needs to be replaced by another one. This can be done by using the functions for setting message data.

Optionally a user can specify a frame skip count. By default, each message will be transmitted at every frame, but specifying a number when a context is created, a message transmission can skip N frame(s). Refer to [Context Options](#) section for how to use this option and its valid data range.

9.3.2.4.1 fxMilTrmCreateMessageHandle

Description

In repeating message mode the user can create and free message handles. A message handle allows a user to control the message throughout its lifetime. Messages created this way will automatically be maintained by FireStack.

Parameters

contextHandle	Reference handle to the context to control. The context handle must be created with FX_MIL_TRM_MODE_REPEATING as context mode. (see fxMilTrmCreateContextHandle)
optionList	Pointer to a user defined list of FXMilTrmMessageOption elements that together form the options for the message to be transmitted. For available options and their default values, please refer to Message Options . Options that are not specified will just use default values.
listSize	Specifies the number of items in the optionList.
messageHandle	Pointer to user allocated variable of type FXMilTrmMessageHandle . Will return a handle to the newly created message on success.

Return Codes

This function returns zero or greater upon success. A negative return value indicates an error. Please use the function [fxGetErrorMessage](#) to look up descriptions corresponding to negative return values.

FX_ERR_INVALID_PARAMETER
FX_ERR_MIL_TRM_INVALID_CONTEXT_HANDLE
FX_ERR_MIL_TRM_INVALID_CONTEXT_MODE
FX_ERR_MIL_TRM_OUT_OF_INTERNAL_RESOURCE
FX_ERR_MIL_TRM_INTERNAL_ERROR
FX_ERR_LICENSE_MODULE

Synopsis

```
FXReturnCode fxMilTrmCreateMessageHandle(
    FXMilTrmContextHandle          contextHandle,
    FXMilTrmMessageOption*       optionList,
    size_t                          listSize,
    FXMilTrmMessageHandle*       messageHandle
)
```

9.3.2.4.2 fxMilTrmCreateMessageHandleExt

Description

In repeating message mode the user can create and free message handles. A message handle allows a user to control the message throughout its lifetime. Messages created this way will automatically be maintained by FireStack. This function allows users to add a callback function that is called upon a completion of every

transmission of the message.

Parameters

contextHandle	Reference handle to the context to control. The context handle must be created with FX_MIL_TRM_MODE_REPEATING as context mode. (see fxMilTrmCreateContextHandle)
optionList	Pointer to a user defined list of FXMilTrmMessageOption elements that together form the options for the message to be transmitted. For available options and their default values, please refer to Message Options . Options that are not specified will just use default values.
listSize	Specifies the number of items in the optionList.
callback	Specifies a user-defined callback function that will be called at every time specified message has been transmitted or if an error occurred. A value of zero indicates that no callback is needed.
userData	This is a convenience feature that allows caller to specify arbitrary user data that fits in a variable of type void*. FireStack will not touch what is provided here. The exact value provided will be handed back to the user when the callback function is called.
messageHandle	Pointer to user allocated variable of type FXMilTrmMessageHandle . Will return a handle to the newly created message on success.

Return Codes

This function returns zero or greater upon success. A negative return value indicates an error. Please use the function [fxGetErrorMessage](#) to look up descriptions corresponding to negative return values.

FX_ERR_INVALID_PARAMETER
FX_ERR_MIL_TRM_INVALID_CONTEXT_HANDLE
FX_ERR_MIL_TRM_INVALID_CONTEXT_MODE
FX_ERR_MIL_TRM_OUT_OF_INTERNAL_RESOURCE
FX_ERR_MIL_TRM_INTERNAL_ERROR
FX_ERR_LICENSE_MODULE
FX_ERR_FIRESTACK_DEMO_TIMEOUT
FX_ERR_LICENSE_EXPIRED

Synopsis

```

FXReturnCode fxMilTrmCreateMessageHandleExt (
    FXMilTrmContextHandle           contextHandle,
    FXMilTrmMessageOption*         optionList,
    size_t                          listSize,
    FXMilTrmCallback                callback,
    void*                            userData,
    FXMilTrmMessageHandle*         messageHandle
)

```

9.3.2.4.3 fxMilTrmCloseMessageHandle

Description

This function can be used to release a handle to a message. After this function is called the user becomes the owner of the buffer(s) the message pointed to. They may now safely be freed.

Parameters

handle	Reference handle to the message to control. (see fxMilTrmCreateMessageHandle)
--------	--

Return Codes

This function returns zero or greater upon success. A negative return value indicates an error. Please use the function [fxGetErrorMessage](#) to lookup descriptions corresponding to negative return values.

FX_ERR_MIL_TRM_INVALID_MESSAGE_HANDLE FX_ERR_MIL_TRM_INTERNAL_ERROR

Synopsis

```
FXReturnCode fxMilTrmCloseMessageHandle(
    FXMilTrmMessageHandle handle
)
```

9.3.2.4.4 fxMilTrmSetMessageData

Description

This function will either just set the message data pointer(s) or replace the existing one(s). If the existing one is replaced then after the function call the user is allowed to take ownership of the previous buffer. In a replacing case, the message data must have been set by the same function, otherwise, an error code (FX_ERR_MIL_TRM_RPT_MESSAGE_DATA_MISMATCH) will be returned.

Parameters

handle	Reference handle to the message to control. (see fxMilTrmCreateMessageHandle)
data	Pointer to a DMA-capable buffer allocated by the user. The buffer will be used as message data including the VPC field but excluding the data CRC. (see fxMemAlloc for allocating DMA-capable buffers)
size	Size of data in bytes.
frameNumber	Reserved for future use.

Return Codes

This function returns zero or greater upon success. A negative return value indicates an error. Please use the function [fxGetErrorMessage](#) to look up descriptions corresponding to negative return values.

FX_ERR_INVALID_PARAMETER
FX_ERR_INVALID_ADDRESS
FX_ERR_MIL_TRM_INVALID_MESSAGE_HANDLE
FX_ERR_MIL_TRM_OUT_OF_INTERNAL_RESOURCE
FX_ERR_MIL_TRM_INTERNAL_ERROR
FX_ERR_LICENSE_MODULE
FX_ERR_MIL_TRM_MSG_EXCEED_MAX_PAYLOAD_SIZE
FX_ERR_MIL_TRM_RPT_MESSAGE_DATA_MISMATCH

Synopsis

```
FXReturnCode fxMilTrmSetMessageData(
    FXMilTrmMessageHandle handle,
    void* data,
    size_t size,
    int32_t frameNumber
)
```

9.3.2.4.5 fxMilTrmSetMessageSplitData

Description

This function will either just set the message data pointer(s) or replace the existing one(s). If the existing one is replaced then after the function call the user is allowed to take ownership of the previous buffer. In a replacing case, the message data must have been set by the same function with the same number of split size, otherwise, an error code (FX_ERR_MIL_TRM_RPT_MESSAGE_DATA_MISMATCH) will be returned.

Parameters

handle	Reference handle to the message to control. (see fxMilTrmCreateMessageHandle)
bufferList	Orderd list of FXBuffer elements that together form the message data including VPC but

	excluding CRC. If less than 5 buffers are needed then just set the size of the last buffer(s) to zero.
frameNumber	Reserved for future use.

Return Codes

This function returns zero or greater upon success. A negative return value indicates an error. Please use the function [fxGetErrorMessage](#) to look up descriptions corresponding to negative return values.

FX_ERR_INVALID_PARAMETER
FX_ERR_INVALID_ADDRESS
FX_ERR_MIL_TRM_INVALID_MESSAGE_HANDLE
FX_ERR_MIL_TRM_OUT_OF_INTERNAL_RESOURCE
FX_ERR_LICENSE_MODULE
FX_ERR_MIL_TRM_INTERNAL_ERROR
FX_ERR_MIL_TRM_MSG_EXCEED_MAX_PAYLOAD_SIZE
FX_ERR_MIL_TRM_RPT_MESSAGE_DATA_MISMATCH

Synopsis

```
FXReturnCode fxMilTrmSetMessageSplitData(
    FXMilTrmMessageHandle    handle,
    FXBuffer                  bufferList[5],
    int32_t                  frameNumber
)
```

9.3.2.4.6 fxMilTrmSetMessageOptions

Description

This function can be used to modify the options of a repeating message.

Parameters

handle	Reference handle to the message to control. (see fxMilTrmCreateMessageHandle)
optionList	Pointer to a user defined list of FXMilTrmMessageOption elements that together form the options for the message to be transmitted. For available options and their default values, please refer to Message Options . Please note that the offset time option cannot be changed by this function.
listSize	Specifies the number of items in the optionList.
frameNumber	Reserved for future use.

Return Codes

This function returns zero or greater upon success. A negative return value indicates an error. Please use the function [fxGetErrorMessage](#) to look up descriptions corresponding to negative return values.

FX_ERR_MIL_TRM_INVALID_MESSAGE_HANDLE
FX_ERR_MIL_TRM_INTERNAL_ERROR
FX_ERR_MIL_TRM_MSG_DATA_NOT_SET
FX_ERR_LICENSE_MODULE
FX_ERR_FIRESTACK_DEMO_TIMEOUT
FX_ERR_LICENSE_EXPIRED

Synopsis

```
FXReturnCode fxMilTrmSetMessageOptions(
    FXMilTrmMessageHandle    handle,
    FXMilTrmMessageOption\*  optionList,
    size_t                  listSize,
    int32_t                  frameNumber
)
```

9.3.2.4.7 fxMilTrmStartMessage

Description

This function can be used to start transmission of a repeating messages. Please make sure to first set the message data. Whenever a repeating message is in the started state it will be transmitted in every frame at its frame offset time until it is stopped.

Parameters

handle	Reference handle to the message to control. (see fxMilTrmCreateMessageHandle)
frameNumber	Reserved for future use.

Return Codes

This function returns zero or greater upon success. A negative return value indicates an error. Please use the function [fxGetErrorMessage](#) to look up descriptions corresponding to negative return values.

FX_ERR_MIL_TRM_INVALID_MESSAGE_HANDLE
FX_ERR_MIL_TRM_INTERNAL_ERROR
FX_ERR_LICENSE_MODULE
FX_ERR_MIL_TRM_MSG_DATA_NOT_SET

Synopsis

```
FXReturnCode fxMilTrmStartMessage(  
    FXMilTrmMessageHandle    handle,  
    int32_t                  frameNumber  
)
```

9.3.2.4.8 fxMilTrmStopMessage

Description

Set a repeating message in the stopped state, preventing it from being transmitted in each frame.

Parameters

handle	Reference handle to the message to control. (see fxMilTrmCreateMessageHandle)
frameNumber	Reserved for future use.

Return Codes

This function returns zero or greater upon success. A negative return value indicates an error. Please use the function [fxGetErrorMessage](#) to look up descriptions corresponding to negative return values.

FX_ERR_MIL_TRM_INVALID_MESSAGE_HANDLE
FX_ERR_LICENSE_MODULE
FX_ERR_MIL_TRM_MSG_DATA_NOT_SET
FX_ERR_MIL_TRM_INTERNAL_ERROR

Synopsis

```
FXReturnCode fxMilTrmStopMessage(  
    FXMilTrmMessageHandle    handle,  
    int32_t                  frameNumber  
)
```

9.3.2.4.9 fxMilTrmGetMessageStatus

Description

This function returns the current status of the specified message.

Parameters

handle	Reference handle to the message to control.
--------	---

	(see fxMilTrmCreateMessageHandle)
status	Pointer to variable of FXMilTrmMessageStatus .

Return Codes

This function returns zero or greater upon success. A negative return value indicates an error. Please use the function [fxGetErrorMessage](#) to look up descriptions corresponding to negative return values.

FX_ERR_INVALID_PARAMETER
FX_ERR_MIL_TRM_INVALID_MESSAGE_HANDLE

Synopsis

```
FXReturnCode fxMilTrmGetMessageStatus(
    FXMilTrmMessageHandle    handle,
    FXMilTrmMessageStatus\*   status
)
```

9.3.2.5. STOF Message Mode

STOF mode will allow the user to set a complete context in a mode where it can transmit STOF messages. As this context has a very specific task we can also define functions that are really specific to STOF messages. It should also be possible to do atomic data updates and maybe even scheduled data updates and start/stop.

9.3.2.5.1 fxMilTrmSetStofMessageOptions**Description**

This function can be used to specify options for the STOF message.

Parameters

handle	Reference handle to the context to control. The context handle must be created with FX_MIL_TRM_MODE_STOF as context mode. (see fxMilTrmCreateContextHandle)
optionList	User defined list of FXMilTrmMessageOption elements that together form the options for the STOF message. For available options please refer to STOF Message Options. Options that are not specified will just use default values.
listSize	Specifies the number of items in the optionList.

Return Codes

This function returns zero or greater upon success. A negative return value indicates an error. Please use the function [fxGetErrorMessage](#) to look up descriptions corresponding to negative return values.

FX_ERR_MIL_TRM_INVALID_CONTEXT_HANDLE
FX_ERR_MIL_TRM_INVALID_CONTEXT_MODE

Synopsis

```
FXReturnCode fxMilTrmSetStofMessageOptions(
    FXMilTrmContextHandle    handle,
    FXMilTrmMessageOption\*   optionList,
    size_t                   listSize
)
```

9.3.2.5.2 fxMilTrmWriteStofMessage**Description**

This function will update the STOF message contents with the specified values. The data is copied into internal buffers by the FireStack and the user remains owner of the provided STOF message data.

Parameters

handle	Reference handle to the context to control.
--------	---

	(see fxMilTrmCreateContextHandle)
stofMessage	User provided pointer to a FXMilStofMessage structure to be used as new STOF message contents. Stack will copy the data such that the user remains the owner of the specified data pointer.
frameNumber	Reserved for future use.

Return Codes

This function returns zero or greater upon success. A negative return value indicates an error. Please use the function [fxGetErrorMessage](#) to look up descriptions corresponding to negative return values.

FX_ERR_MIL_TRM_INVALID_CONTEXT_HANDLE
FX_ERR_MIL_TRM_INVALID_CONTEXT_MODE
FX_ERR_MIL_TRM_OUT_OF_INTERNAL_RESOURCE
FX_ERR_MIL_TRM_INTERNAL_ERROR
FX_ERR_LICENSE_MODULE

Synopsis

```
FXReturnCode fxMilTrmWriteStofMessage (
    FXMilTrmContextHandle    handle,
    const FXMilStofMessage*  stofMessage,
    int32_t                  frameNumber
)
```

9.3.2.5.3 fxMilTrmStartStofMessage

Description

This function can be used to start transmission of a STOF packet. Please first set the options for the STOF packet in write some data to it before starting.

Parameters

handle	Reference handle to the context to control. (see fxMilTrmCreateContextHandle)
frameNumber	Reserved for future use.

Return Codes

This function returns zero or greater upon success. A negative return value indicates an error. Please use the function [fxGetErrorMessage](#) to look up descriptions corresponding to negative return values.

FX_ERR_MIL_TRM_INVALID_CONTEXT_HANDLE
FX_ERR_MIL_TRM_INVALID_CONTEXT_MODE
FX_ERR_MIL_TRM_INTERNAL_ERROR
FX_ERR_LICENSE_MODULE

Synopsis

```
FXReturnCode fxMilTrmStartStofMessage (
    FXMilTrmContextHandle    handle,
    int32_t                  frameNumber
)
```

9.3.2.5.4 fxMilTrmStopStofMessage

Description

This function can be used to stop transmission of the STOF packet.

Parameters

handle	Reference handle to the context to control. (see fxMilTrmCreateContextHandle)
frameNumber	Reserved for future use.

Return Codes

This function returns zero or greater upon success. A negative return value indicates an error. Please use the function [fxGetErrorMessage](#) to look up descriptions corresponding to negative return values.

```
FX_ERR_MIL_TRM_INVALID_CONTEXT_HANDLE
FX_ERR_MIL_TRM_INVALID_CONTEXT_MODE
FX_ERR_MIL_TRM_INTERNAL_ERROR
FX_ERR_LICENSE_MODULE
```

Synopsis

```
FXReturnCode fxMilTrmStopStofMessage(
    FXMilTrmContextHandle    handle,
    int32_t                  frameNumber
)
```

9.3.3. Type Definitions**9.3.3.1. FXMilTrmContextHandle**

```
typedef uint32_t FXMilTrmContextHandle;
```

9.3.3.2. FXMilTrmMessageHandle

```
typedef uint32_t FXMilTrmMessageHandle;
```

9.3.3.3. FXMilTrmCallback**Description**

This type definition forms the prototype for functions that can be registered as AS5643 Transmission callback function. It will return information like a handle to the bus that triggered the callback, the user-registered user data and a handle to the context that triggered the callback.

Parameters

handle	Reference handle to the bus that triggered the callback. (see fxCreateBusHandle)
userData	User-provided data pointer that was specified when the callback was registered.
contextHandle	Reference handle to the context that triggered the callback. (see fxMilTrmCreateContextHandle)
eventCodes	Event codes from the transfer status. See User Callback Event Code Bits for more details.

Synopsis

```
typedef void (*FXMilTrmCallback) (
    FXBusHandle    handle,
    void*          userData,
    FXMilTrmContextHandle contextHandle,
    uint32_t       eventCodes
)
```

9.3.4. Structures**9.3.4.1. FXMilTrmContextOption****Description**

This structure can be used to specify options when opening an AS5643 Transmission Context.

Members

optionId	ID value of the option to set. For options please refer to Context Options .
----------	--

value	Value to set for the specified option.
-------	--

Synopsis

```
typedef struct {
    uint32_t    optionId;
    uint32_t    value;
} FXMilTrmContextOption;
```

9.3.4.2. FXMilTrmMessageOption**Description**

This structure can be used to specify an option of AS5643 transmission. Available options depend on where these items are used and will be listed in the documentation of the functions using this data type.

Members

optionId	ID value of the option to set. For available options, please refer to Message Options .
value	Value to set for the specified option.

Synopsis

```
typedef struct {
    uint32_t    optionId;
    uint32_t    value;
} FXMilTrmMessageOption;
```

9.3.4.3. FXBuffer**Description**

This structure can be used to specify the location and size of a DMA-capable buffer.

Members

data	Pointer to a DMA-capable buffer allocated by the user. (see fxMemAlloc for allocating DMA-capable buffers)
size	Size of data in bytes.

Synopsis

```
typedef struct {
    void*       data;
    size_t      size;
} FXBuffer;
```

9.3.4.4. FXMilTrmMessage**Description**

This structure can be used to represent an AS5643 message when its data is located in a single buffer.

Members

optionList	Pointer to a user defined list of FXMilTrmMessageOption elements that together form the options for the message to be transmitted. For available options and their default values, please refer to Message Options . Options that are not specified will just use default values.
listSize	Specifies the number of items in the optionList.
data	Pointer to a DMA-capable buffer allocated by the user. The buffer will be used as message data including the VPC field but excluding the data CRC. (see fxMemAlloc for allocating DMA-capable buffers)
dataSize	Size of data in bytes.

Synopsis


```
typedef struct {
    FXMilTrmMessageOption*  optionList;
    size_t                  listSize;
    void*                   data;
    size_t                  dataSize;
} FXMilTrmMessage;
```

9.3.4.5. FXMilTrmSplitMessage

Description

This structure can be used to represent an AS5643 message when its data is located in more than one data buffer.

Members

optionList	Pointer to a user defined list of FXMilTrmMessageOption elements that together form the options for the message to be transmitted. For available options and their default values, please refer to Message Options . Options that are not specified will just use default values.
listSize	Specifies the number of items in the optionList.
bufferList	Orderd list of FXBuffer elements that together form the message data including VPC but excluding CRC. If less than 5 buffers are needed then just set the size of the last buffer(s) to zero.

Synopsis

```
typedef struct {
    FXMilTrmMessageOption*  optionList;
    size_t                  listSize;
    FXBuffer                 bufferList[5];
} FXMilTrmSplitMessage;
```

9.3.4.6. FXMilTrmStrmStatus

Description

This structure is used when [fxMilTrmStrmGetStatus](#) function is called.

Members

status	1: Context has started by calling fxMilTrmStrmStart function. Context may be in idling if the end of transmit queue has been reached. 0: Context has not started or has been stopped by calling fxMilTrmStrmStop function. The context can be started/resumed by calling fxMilTrmStrmStart function.
--------	---

Synopsis

```
typedef struct {
    uint32_t                status;
} FXMilTrmStrmStatus;
```

9.3.4.7. FXMilTrmMessageStatus

Description

This structure is used when [fxMilTrmGetMessageStatus](#) function is called.

Members

status	1: Message is active and is being transmitted repeatedly. 0: Message transmission has stopped.
contextHandle	Context handle to which the message belongs.

Synopsis

```
typedef struct {
```

```

        uint32_t          status;
        FXMilTrmContextHandle contextHandle;
    } FXMilTrmMessageStatus;

```

9.3.4.8. FXMilStofMessage

Description

This structure defines complete contents of an AS5643 STOF message as defined in the SAE-AS5643 specification.

Members

ccBranchStatus	The CC Status Word (Long Packed Boolean) shall indicate the failure state of each of the CCs.
networkBusMode	<p>The Network Bus Mode Word (Long Packed Boolean) shall indicate the current mode of operation for the CC Bus as defined below. One of the bits shall be set to one to indicate the current mode. These bits are mutually exclusive. If multiple bits are set, the Remote Nodes shall disregard this erroneous setting and continue to operate in the previous mode of operation. Any mode word setting other than those shown below shall be considered illegal and invalid. Use of the user defined bit assignments to accommodate different organizations should be defined in the network profile slash sheet for the target application.</p> <p>Network Bus Mode Bits:</p> <ul style="list-style-type: none"> • LSB • Bit 31: Start-up/Initialization • Bit 30: Normal • Bit 29: CC-In-Test (STOF messages may or may not be present depending upon which tests are being performed) • Bit 28: Reserved (Shall be initialized to zero) • Bit 27: Program Upload • Bit 26: In Test Mode – Read Only • Bit 25: In Test Mode – Read/Write • Bits 24 – 16: Reserved (Shall be initialized to zero (0)) • Bits 15 – 0: User Defined (as defined in the network profile slash sheet for the target application) Protocol Functions 78 Copyright 2008, DapTechnology, Version 11/28/2008 • MSB
vehicleState	The Vehicle State Word (Long Packed Boolean) in the STOF message is used to indicate the current state of the vehicle as defined in the network profile slash sheet for the target application.
vehicleTime	The Vehicle Time shall be a 32-bit Unsigned Long Integer denoting time as determined by the supported system. Format and use of Vehicle Time shall be defined in the network profile slash sheet for the target application.
quadlet4	User Defined
quadlet5	User Defined
quadlet6	User Defined
quadlet7	User Defined
quadlet8	User Defined
verticalParityCheck	Vertical Parity Check field. Will only be used if automatic VPC calculation is disabled.

Synopsis

```

typedef struct {
    uint32_t          ccBranchStatus;
    uint32_t          networkBusMode;
}

```

```

uint32_t    vehicleState;
uint32_t    vehicleTime;
uint32_t    quadlet4;
uint32_t    quadlet5;
uint32_t    quadlet6;
uint32_t    quadlet7;
uint32_t    quadlet8;
uint32_t    verticalParityCheck;
} FXMilStofMessage;

```

9.3.5. Constants

9.3.5.1. Error Codes

The following values may be returned by AS5643 Transmit functions.

FX_ERR_MIL_TRM_INVALID_CONTEXT_MODE
FX_ERR_MIL_TRM_INVALID_CONTEXT_HANDLE
FX_ERR_MIL_TRM_NO_AVAILABLE_CONTEXT
FX_ERR_MIL_TRM_INTERNAL_ERROR
FX_ERR_MIL_TRM_STRM_DATA_FORMAT_ERROR
FX_ERR_MIL_TRM_INVALID_MESSAGE_HANDLE
FX_ERR_MIL_TRM_OUT_OF_INTERNAL_RESOURCE
FX_ERR_MIL_TRM_INVALID_BUFFER_ADDRESS
FX_ERR_MIL_TRM_MSG_DATA_NOT_SET
FX_ERR_MIL_TRM_MSG_EXCEED_MAX_PAYLOAD_SIZE
FX_ERR_MIL_TRM_STRM_EMPTY
FX_ERR_MIL_TRM_RPT_MESSAGE_DATA_MISMATCH
FX_ERR_MIL_TRM_CONTEXT_ALREADY_STARTED
FX_ERR_MIL_TRM_CONTEXT_ALREADY_STOPPED
FX_ERR_MIL_TRM_STOF_CONTEXT_ALREADY_EXISTS

9.3.5.2. Context Options

The following message options are available:

FX_MIL_TRM_CONTEXT_OPT_MODE	This option shall be used to specify the transmission mode for the context to be opened. Please refer to Context Modes for more information. Default: This option shall not be left out.
FX_MIL_TRM_CONTEXT_OPT_JITTER_RANGE	This option can be used to set the range of the jitter on the transmission timing that is optionally applied to each packet transmitted by this context. For available jitter ranges, please refer to Jitter Ranges . Default: FX_MIL_TRM_JITTER_RANGE_0
FX_MIL_TRM_CONTEXT_OPT_JITTER_DIRECTION	This option can be used to set the range of the jitter on the transmission timing that is optionally applied to each packet transmitted by this context. For available jitter ranges, please refer to Jitter Directions . Default: FX_MIL_TRM_JITTER_RANGE_0
FX_MIL_TRM_CONTEXT_OPT_FRAME_SKIP_COUNT	For Repeating Messages Mode Only. This option specifies a number of frames to skip when a message gets transmitted. For example, specifying 0 (zero) makes a

	message gets transmitted at every frame while specifying 2 (two) makes a message gets transmitted at every 3 frames. Valid range of value is from 0 (zero) to 10 (inclusive). Default: 0
--	--

9.3.5.3. Context Modes

The following transmission modes are available:

FX_MIL_TRM_MODE_STREAMING	This mode allows the user to write large or small sets of messages to the FireStack and have them transmitted automatically on the specified frame offset times. The provided data needs to contain so called frame separator elements to indicate the following message needs to be transmitted in the next frame. (please refer to Streaming Messages Mode for more information)
FX_MIL_TRM_MODE_REPEATING	This mode allows to user to setup a message that will automatically be transmitted each frame by the FireStack. The user will have a pointer to the actual data of the message and is allowed to manipulate the data at any point in time without having to worry about its timed transmission. Very useful for AS5643 status messages. (please refer to Repeating Messages Mode for more information)
FX_MIL_TRM_MODE_SINGLE	This mode allows the user to simply transmit a message as soon as possible but exactly at the specified frame offset time. Several messages may be handed to the FireStack for immediate transmission and the FireStack will then take care of the actual moment of transmission. (please refer to Single Message Mode for more information)
FX_MIL_TRM_MODE_STOF	This mode allows the user to control transmission of STOF messages. (please refer to STOF Message Mode for more information)

9.3.5.4. Jitter Ranges

The following jitter ranges are available:

FX_MIL_TRM_JITTER_RANGE_0	This jitter range setting will result in a fixed jitter value of 0 micro seconds (no jitter).
FX_MIL_TRM_JITTER_RANGE_1	This jitter range setting will result in a fixed jitter value of 1 micro second. The jitter value could be only positive or only negative or either way depending on the jitter direction setting.
FX_MIL_TRM_JITTER_RANGE_3	This jitter range setting will result in a random jitter value for each packet that has its jitter option enabled with a value between 1 and 3 micro seconds. The jitter value could be only positive or only negative or either way depending on the jitter direction setting.
FX_MIL_TRM_JITTER_RANGE_7	This jitter range setting will result in a random jitter value for each packet that has its jitter option enabled with a value between 1 and 7 micro seconds. The jitter value could be only positive or only negative or either way depending on the jitter direction setting.
FX_MIL_TRM_JITTER_RANGE_15	This jitter range setting will result in a random jitter value for each packet that has its jitter option enabled with a value between 1 and 15 micro seconds. The jitter value could be only positive or only negative or either way depending on the jitter direction setting.
FX_MIL_TRM_JITTER_RANGE_31	This jitter range setting will result in a random jitter value for each packet that has its jitter option enabled with a value between 1 and

	31 micro seconds. The jitter value could be only positive or only negative or either way depending on the jitter direction setting.
FX_MIL_TRM_JITTER_RANGE_63	This jitter range setting will result in a random jitter value for each packet that has its jitter option enabled with a value between 1 and 63 micro seconds. The jitter value could be only positive or only negative or either way depending on the jitter direction setting.
FX_MIL_TRM_JITTER_RANGE_127	This jitter range setting will result in a random jitter value for each packet that has its jitter option enabled with a value between 1 and 127 micro seconds. The jitter value could be only positive or only negative or either way depending on the jitter direction setting.

9.3.5.5. Jitter Directions

The following jitter directions are available:

FX_MIL_TRM_JITTER_DIRECTION_B OTH	This jitter direction setting will result in randomly subtracting or adding a jitter value within the range defined by the jitter range setting.
FX_MIL_TRM_JITTER_DIRECTION_P OS	This jitter direction setting will result in only adding a jitter value within the range defined by the jitter range setting.
FX_MIL_TRM_JITTER_DIRECTION_N EG	This jitter direction setting will result in only subtracting a jitter value within the range defined by the jitter range setting.

9.3.5.6. Message Options

The following message options are available:

FX_MIL_TRM_OPT_SPEED	This option can be used to set the transmission speed of the message. Supported options are (depending on hardware capabilities): <ul style="list-style-type: none"> • FX_SPEED_100 • FX_SPEED_200 • FX_SPEED_400 • FX_SPEED_800 • FX_SPEED_1600 • FX_SPEED_3200 Default: FX_SPEED_400
FX_MIL_TRM_OPT_CHANNEL	This option can be used to set the 1394 channel the message needs to be transmitted on. Value needs to be positive and smaller than 64. Default: 0
FX_MIL_TRM_OPT_OFFSETTIME	This option can be used to set the AS5643 transmit offset time relative to start of frame. Values need to be positive and smaller than the frame length minus the sync margin. Please refer to AS5643 Frame Timing for more information about which value to use here. Default: 0
FX_MIL_TRM_OPT_JITTER_MODE	This option can be used to apply semi random jitter to the transmission time. Please refer to Jitter Modes for more information.
FX_MIL_TRM_OPT_AUTOVPC	This option can be used to specify whether the hardware needs to automatically populate the VPC field. See supported options in Auto VPC Modes . Default: FX_MIL_TRM_AUTOVPC_ENABLE_VPC (see Auto VPC Modes)

FX_MIL_TRM_OPT_ISFRAMESEPARATOR	<p>This option can be used to specify that a message is actually just an indication for start of the next frame. This option only makes sense for messages transmitted in Streaming Message Mode. If this option is enabled then no message is transmitted but hardware just waits for the start of frame. Supported options are:</p> <ul style="list-style-type: none"> • 0 - disabled meaning it is a regular message • 1 - enabled, meaning it is not a message but a wait start of frame item <p>Default: 0</p>
---------------------------------	---

9.3.5.7. Jitter Modes

The following jitter modes are available for the FX_MIL_TRM_OPT_JITTER_MODE option defined in [Message Options](#).

FX_MIL_TRM_JITTER_DISABLE	Jitter will not be applied to the transmission time of the message.
FX_MIL_TRM_JITTER_ENABLE	Jitter will be applied to the transmission time of the packet. The semi random jitter value will be determined based on the jitter settings of the context this message is part of.

9.3.5.8. Auto VPC Modes

The following Auto VPC modes are available for FX_MIL_TRM_OPT_AUTOVPC option defined in [Message Options](#).

FX_MIL_TRM_AUTOVPC_DISABLE_VPC	No VPC insertion
FX_MIL_TRM_AUTOVPC_ENABLE_VPC	Calculate correct VPC and insert as last data quadlet
FX_MIL_TRM_AUTOVPC_INCORRECT_VPC	Calculate incorrect (inverted) VPC and insert as last data quadlet

9.3.5.9. User Callback Event Code Bits

Definitions for possible event codes used in the [User Callback Function](#). For the Streaming mode, user callbacks can be enabled/disabled at any packet; Transmit events will be logical OR'ed between callbacks.

Define	Value	Description
FX_MIL_TRM_EVENT_TRANSMIT_COMPLETE	0x00000001	At least one of packet transmissions have been completed without any error.
FX_MIL_TRM_EVENT_TRANSMIT_ERROR	0x00010000	At least one of packet transmissions have been completed with an error.
FX_MIL_TRM_EVENT_NOT_TRANSMIT	0x00100000	At least one of packet transmissions were skipped.

9.3.6. Data Formats

9.3.6.1. AS5643 Regeneration Format

This section describes one of the data formats used in [fxMilTrmStrmWriteImmediate\(\)](#).

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
Item Type																															
Data (1 ~ N quadlets)																															

Item Type (32 bits)

1: Frame Separator

3: Stream Packet

Item Type 1 - Frame Separator

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																															
Reserved																															

Item Type 3 - Stream Packet

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
3																															
Offset																															
Speed																															
Payload Size																															
Header																															
Payload Data																															

Offset: (32 bits)

Offset from the start of a frame (microseconds)

Speed: (32 bits)

Speed code 1: 100 Mbps
 2: 200 Mbps
 3: 400 Mbps
 4: 800 Mbps

Payload Size: (32 bits)

Data Payload size in quadlets excluding the header

Header: (32 bits)

Header part of IEEE1394 stream packet

*Payload Data: (32 bits) * Payload Size*

Data Payload part of IEEE1394 stream packet

Chapter 10. 1394 API Reference

10.1. Serial Bus Management

The serial bus management module combines several tasks related to serial bus management. This module can be turned on or off by setting the corresponding [Feature](#). The exact functionality can be controlled by using the [SBM Capabilities](#) setting when opening the bus using [fxCreateBusHandle\(\)](#).

10.1.1. Settings

10.1.1.1. SBM Capabilities

The following setting can be used as settingId in an [FXSetting](#) structure passed to [fxCreateBusHandle\(\)](#) to control which SBM capabilities will be enabled for the bus that is being opened. Please make sure to also enable the [Serial Bus Management](#) module itself.

FX_SETTING_ID_SBM_CAPABILITIES

The following settings can be or-ed together to form the [FXSetting](#) value field:

FX_SBM_TRANSACTION_CAPABLE

Perform the duties of a Transaction capable node

FX_SBM_ISOCHRONOUS_CAPABLE

Perform the duties of an Isochronous capable node

FX_SBM_CYCLE_MASTER_CAPABLE

Perform the duties of a Cycle Master capable node

FX_SBM_IRM_CAPABLE

Perform the duties of an IRM capable node

The following features are currently reserved:

FX_SBM_BUS_MANAGER_CAPABLE

10.1.2. Functions

10.1.2.1. fxSetBusResetCallback

Description

This function can be used to register a single callback function that will be called each time a bus reset event occurs.

Parameters

busHandle	Reference handle to the bus to control. (see fxCreateBusHandle)
callback	The user-defined function that will be called when a bus reset event occurs. There is only one function that can be registered: Calling this function with another callback function will replace the current one, and with 0 will disable the callback.
userData	This is a convenience feature that allows caller to specify arbitrary user data that fits in a variable of type void*. FireStack will not touch what is provided here. The exact value provided will be handed back to the user when the callback

function is called.

Return Codes

This function returns zero or greater upon success. A negative return value indicates an error. Please use the function [fxGetErrorMessage](#) to lookup descriptions corresponding to negative return values.

FX_ERR_INVALID_HANDLE**Synopsis**

```
FXReturnCode fxSetBusResetCallback(
    FXBusHandle          busHandle,
    FXBusResetCallback callback,
    void*                userData
)
```

10.1.2.2. fxGetBusGeneration**Description**

This function can be used to get the current bus generation number. The bus generation number increments each time a bus reset occurs on the bus. All functionality that makes use of node IDs uses the bus generation number to determine if the node IDs used are still valid. This is very important as node IDs are dynamic and may change every time a bus reset occurs.

It is recommended that the user application calls this function once each time a bus reset occurs and then remembers the value until the next bus reset. Many functions take this value as an input and if the user passes an old generation number those functions will fail with a return code indicating that a bus reset occurred.

Whenever a bus reset occurs it is up to the user application to call this function and determine the new node IDs.

Parameters

busHandle	Reference handle to the bus to control. (see fxCreateBusHandle)
generation	Will return the current bus generation number.

Return Codes

This function returns zero or greater upon success. A negative return value indicates an error. Please use the function [fxGetErrorMessage](#) to lookup descriptions corresponding to negative return values.

FX_ERR_INVALID_HANDLE**FX_ERR_INVALID_PARAMETER****FX_ERR_INVALID_LOCAL_NODE_ID****Synopsis**

```
FXReturnCode fxGetBusGeneration(
    FXBusHandle handle,
    uint32_t* generation
)
```

10.1.2.3. fxGetNumberOfNodesOnBus**Description**

This function can be used to get the number of nodes on the bus. As the topology may change after every bus reset, a bus generation number needs to be passed to this function to ensure user application is aware of the latest bus reset that occurred on the bus. The bus generation can be determined by making a function call to [fxGetBusGeneration\(\)](#).

Parameters

busHandle	Reference handle to the bus to control. (see fxCreateBusHandle)
generation	Set this value to the current bus generation number. (see fxGetBusGeneration)
numNodes	User allocated buffer that will return the number of nodes on the bus.

Return Codes

This function returns zero or greater upon success. A negative return value indicates an error. Please use the function [fxGetErrorMessage](#) to lookup descriptions corresponding to negative return values.

FX_ERR_INVALID_HANDLE
FX_ERR_INVALID_PARAMETER

Synopsis

```
FXReturnCode fxGetNumberOfNodesOnBus(
    FXBusHandle handle,
    uint32_t generation,
    uint32_t* numNodes
)
```

10.1.2.4. fxGetLocalNodeId**Description**

This function can be used to get the Physical ID of the local Node on a bus we are connected to. As node IDs may change after every bus reset, a bus generation number needs to be passed to this function to ensure user application is aware of the latest bus reset that occurred on the bus. The bus generation can be determined by making a function call to [fxGetBusGeneration](#)().

Parameters

busHandle	Reference handle to the bus to control. (see fxCreateBusHandle)
generation	Set this value to the current bus generation number. (see fxGetBusGeneration)
nodeId	Will return the Physical ID of the local Node.

Return Codes

This function returns zero or greater upon success. A negative return value indicates an error. Please use the function [fxGetErrorMessage](#) to lookup descriptions corresponding to negative return values.

FX_ERR_INVALID_HANDLE
FX_ERR_INVALID_PARAMETER
FX_ERR_INVALID_LOCAL_NODE_ID

Synopsis

```
FXReturnCode fxGetLocalNodeId(
    FXBusHandle handle,
    uint32_t generation,
    int32_t* nodeId
)
```

10.1.2.5. fxGetMaxSpeedToNode**Description**

This function can be used to get the highest possible packet speed support by all nodes in between the local node and the specified node ID. As node IDs may change after every bus reset, a bus generation number

needs to be passed to this function to ensure user application is aware of the latest bus reset that occurred on the bus. The bus generation can be determined by making a function call to [fxGetBusGeneration\(\)](#).

Parameters

busHandle	Reference handle to the bus to control. (see fxCreateBusHandle)
generation	Set this value to the current bus generation number. (see fxGetBusGeneration)
nodeId	Will return the Physical ID of the local Node.
pathSpeed	User-allocated buffer that will be used to write the maximum speed to.

Return Codes

This function returns zero or greater upon success. A negative return value indicates an error. Please use the function [fxGetErrorMessage](#) to lookup descriptions corresponding to negative return values.

FX_ERR_FIRESTACK_NOT_INITIALIZED
FX_ERR_INVALID_HANDLE
FX_ERR_INVALID_PARAMETER
FX_ERR_BUS_RESET_DETECT
FX_ERR_GENERAL

Synopsis

```
FXReturnCode fxGetMaxSpeedToNode(
    FXBusHandle busHandle,
    uint32_t    generation,
    uint32_t    nodeId,
    uint32_t*   pathSpeed
)
```

10.1.3. Type Definitions

10.1.3.1. FXBusResetCallback

Description

Users can define a function of this type and register it using the function [fxSetBusResetCallback](#). The user defined function will then be called each time a bus reset event occurs.

Parameters

busHandle	Reference handle to the bus to control. (see fxCreateBusHandle)
userData	User-provided data pointer that was specified when the callback was registered.

Synopsis

```
typedef void (*FXBusResetCallback) {
    FXBusHandle    busHandle,
    void*          userData
};
```

10.2. Inbound Transactions

Inbound Transactions are defined in two separated methods: Map Local Memory and Transaction Handler.

Map Local Memory

The user can "map" a chunk of local memory to a specific address space which is defined by the IEEE1394 standard. When the stack receives an asynchronous request packet from a remote device and finds that the address and size of the packet fit one of mapped memory regions, it will automatically perform the requested operation (read, write or lock), if permitted, and will send a response packet back to the requester. The user will be notified by the notification callback function when the transaction completes.

Transaction Handler

Similar to the Map Local Memory above except that a user will "register" a specific address space instead of "mapping" local memory. When the stack receives a request packet, which has the same criteria mentioned above, it will call the user-specified handler callback function. The user can then perform any operation within the callback function. Upon returning from the handler callback, the FireStack may transmit a response packet. The notification callback function will be called after completion of the response process.

10.2.1. Functions

10.2.1.1. Memory Mapping Functions

10.2.1.1.1 fxMapLocalMemory

Description

This function creates and allocates a handle to the specified memory mapping information. All incoming packets with the address within the specified range will be automatically acknowledged and responded by the stack.

Parameters

busHandle	Reference handle to the bus to control. (see fxCreateBusHandle)
options	Pointer to FXMappingOptions structure that specifies memory regions, size, etc.
localMemory	The start address of the DMA-capable buffer which will be mapped to the IEEE1394 memory space specified by the region parameter above. (See fxMemAlloc for allocating DMA-capable buffers)
mappingHandle	Filled with a new, valid mapping handle if no error is reported.

Return Codes

This function returns zero or greater upon success. A negative return value indicates an error. Please use the function [fxGetErrorMessage](#) to lookup descriptions corresponding to negative return values.

```

FX_ERR_INVALID_HANDLE
FX_ERR_INVALID_PARAMETER
FX_ERR_INVALID_ADDRESS
FX_ERR_IBD_TRN_MAX_MAPPINGS_REACHED
FX_ERR_IBD_TRN_MAPPING_OVERLAPPING
FX_ERR_INTERNAL_ERROR
FX_ERR_LICENSE_MODULE
FX_ERR_LICENSE_EXPIRED
FX_ERR_FIRESTACK_DEMO_TIMEOUT
FX_ERR_INVALID_DATA_ADDRESS
FX_ERR_INVALID_DATA_SIZE
FX_ERR_IBD_TRN_INVLIAD_LOCAL_MEM_ADDRESS

```

Synopsis

```

FXReturnCode fxMapLocalMemory(
    FXBusHandle          busHandle,
    const FXMappingOptions* options,
    void*                localMemory,
    FXMappingHandle*    mappingHandle
)

```

10.2.1.1.2 fxMapRequestHandler

Description

This function creates and allocates a handle to the specified memory mapping information. The user-specified handler will be called when a matching incoming request is received. After returning from the handler callback, the FireStack may transmit a response packet. See [FXTransactionData](#), [FXRequestHandlerCallback](#) and [FXRequestNotificationCallback](#) for additional information.

Parameters

busHandle	Reference handle to the bus to control. (see fxCreateBusHandle)
options	Pointer to FXMappingOptions structure.
handlerCallback	User-specified callback function that will be called upon a reception of a request packet with address and size are within the memory region. See also FXRequestHandlerCallback .
handlerUserData	Pointer to a user-specified data. The pointer will be carried to the user callback function specified above. See also FXRequestHandlerCallback .
mappingHandle	Filled with a new, valid mapping handle if no error is reported.

Return Codes

This function returns zero or greater upon success. A negative return value indicates an error. Please use the function [fxGetErrorMessage](#) to lookup descriptions corresponding to negative return values.

```

FX_ERR_INVALID_HANDLE
FX_ERR_INVALID_PARAMETER
FX_ERR_INVALID_ADDRESS
FX_ERR_IBD_TRN_MAX_MAPPINGS_REACHED
FX_ERR_IBD_TRN_MAPPING_OVERLAPPING
FX_ERR_INTERNAL_ERROR
FX_ERR_LICENSE_MODULE
FX_ERR_LICENSE_EXPIRED
FX_ERR_FIRESTACK_DEMO_TIMEOUT
FX_ERR_INVALID_DATA_ADDRESS
FX_ERR_INVALID_DATA_SIZE

```

Synopsis

```

FXReturnCode fxMapRequestHandler(
    FXBusHandle          busHandle,
    const FXMemoryOptions* options,
    FXRequestHandlerCallback handlerCallback,
    void*                handlerUserData,
    FXMappingHandle*    mappingHandle
)

```

10.2.1.1.3 `fxClearMemoryMapping`**Description**

Clears the specified memory mapping region information and deletes the entry.

Parameters

mappingHandle	Reference handle to the mapping data. (see fxMapLocalMemory and fxMapRequestHandler)
---------------	---

Return Codes

This function returns zero or greater upon success. A negative return value indicates an error. Please use the function [fxGetErrorMessage](#) to lookup descriptions corresponding to negative return values.

FX_ERR_IBD_TRN_INVALID_MAPPING_HANDLE
FX_ERR_IBD_TRN_INTERNAL_ERROR

Synopsis

```
FXReturnCode fxClearMemoryMapping(
    FXMappingHandle    mappingHandle
)
```

10.2.1.2. Local Memory Access Functions10.2.1.2.1 `fxReadLocalMemory`**Description**

This function can be used to read from local IEEE-1394 memory space. Data returned is the same as would be returned by a remote node performing a read transaction on the local node.

Parameters

busHandle	Reference handle to the bus to control. (see fxCreateBusHandle)
address	The start address of the IEEE1394 memory space from which this transaction intends to read.
buffer	User-specified buffer address. Data will be copied to this buffer.
size	Pointer a variable that specifies the maximum size of the desired receive data size. The buffer specified above must have enough storage space. An actual data size of the read operation will be returned to the specified variable.

Return Codes

This function returns zero or greater upon success. A negative return value indicates an error. Please use the function [fxGetErrorMessage](#) to look up descriptions corresponding to negative return values.

FX_ERR_INVALID_HANDLE
FX_ERR_INVALID_PARAMETER
FX_ERR_INVALID_ADDRESS
FX_ERR_OBD_TRN_RCV_RESP_ADDRESS
FX_ERR_OBD_TRN_RCV_RESP_CONFLICT
FX_ERR_OBD_TRN_RCV_RESP_DATA
FX_ERR_OBD_TRN_RCV_RESP_TYPE

Synopsis

```
FXReturnCode fxReadLocalMemory(
    FXBusHandle        handle,
    const FXAddress64* address,
```

```

        void*          buffer,
        size_t*       size
    )

```

10.2.1.2.2 fxWriteLocalMemory

Description

This function can be used to write to local IEEE-1394 memory space. The result is the same as when a remote node performs a write transaction on the local node.

Parameters

busHandle	Reference handle to the bus to control. (see fxCreateBusHandle)
address	The start address of the IEEE1394 memory space from which this transaction intends to read.
data	User-specified data. Data will be copied to destination address.
size	Specify the maximum size of the desired receive data size. The buffer specified above must have enough storage space.

Return Codes

This function returns zero or greater upon success. A negative return value indicates an error. Please use the function [fxGetErrorMessage](#) to look up descriptions corresponding to negative return values.

```

FX_ERR_INVALID_HANDLE
FX_ERR_INVALID_PARAMETER
FX_ERR_INVALID_ADDRESS
FX_ERR_OBD_TRN_RCV_RESP_ADDRESS
FX_ERR_OBD_TRN_RCV_RESP_CONFLICT
FX_ERR_OBD_TRN_RCV_RESP_DATA
FX_ERR_OBD_TRN_RCV_RESP_TYPE

```

Synopsis

```

FXReturnCode fxWriteLocalMemory(
    FXBusHandle          handle,
    const FXAddress64*   address,
    void*                data,
    size_t                size
)

```

10.2.1.2.3 fxLockLocalMemory

Description

This function can be used to perform a lock operation on local IEEE-1394 memory space. The result is the same as when a remote node performs a lock transaction on the local node.

Parameters

handle	Reference handle to the bus to control. (see fxCreateBusHandle)
address	The start address of the IEEE1394 memory space from which this transaction intends to read.
lockOperation	Lock operation to perform on destination address
buffer	User-specified data.
size	Pointer a variable that specifies the maximum size of the desired receive data size. The buffer specified above must

	have enough storage space. An actual data size of the lock operation will be returned to the specified variable.
--	--

Return Codes

This function returns zero or greater upon success. A negative return value indicates an error. Please use the function [fxGetErrorMessage](#) to look up descriptions corresponding to negative return values.

```

FX_ERR_INVALID_HANDLE
FX_ERR_INVALID_PARAMETER
FX_ERR_INVALID_ADDRESS
FX_ERR_OBD_TRN_RCV_RESP_ADDRESS
FX_ERR_OBD_TRN_RCV_RESP_CONFLICT
FX_ERR_OBD_TRN_RCV_RESP_DATA
FX_ERR_OBD_TRN_RCV_RESP_TYPE

```

Synopsis

```

FXReturnCode fxLockLocalMemory(
    FXBusHandle          handle,
    const FXAddress64*   address,
    uint32_t             lockOperation,
    void*                buffer,
    size_t*              size
)

```

10.2.2. Type Definitions**10.2.2.1. FXRequestHandlerCallback****Description**

This function definition is used to specify a callback function that will be called when the stack receives an incoming request packet with address and size which are within a memory region created by [fxMapRequestHandler](#).

A return value of this user function is very important and must be one of [Response Codes](#).

If the transaction response needs to carry data from this callback function then a valid data pointer allocated with [fxMemAlloc\(\)](#) needs to be set in the [FXTransactionData](#) structure as well as the corresponding size. The user application shall not free that data until the response has been completed as indicated by calling [FXRequestNotificationCallback](#).

Parameters

busHandle	Reference handle to the bus to control. (see fxCreateBusHandle)
userData	Pointer to the data specified in fxMapRequestHandler .
mappingHandle	Reference handle to the mapping data. (see fxMapRequestHandler)
transactionData	Pointer to FXTransactionData . The stack will fill this structure with the packet data it has received. The callback implementation needs to set the data member for read and lock transactions.

Synopsis

```

typedef int32_t (*FXRequestHandlerCallback) (
    FXBusHandle          busHandle,
    void*                userData,
    FXMappingHandle      mappingHandle,
    FXTransactionData*   transactionData
);

```


10.2.2.2. FXRequestNotificationCallback

Description

This function definition is used to specify a callback function that will be called when an inbound transaction completes.

Parameters

busHandle	Reference handle to the bus to control. (see fxCreateBusHandle)
userData	Pointer to the data specified in FXMappingOptions .
mappingHandle	Reference handle to the mapping data. (see fxMapTransactionHandler)
result	For the mappingHandle created by fxMapLocalMemory , this will be a response code the stack has automatically sent to; for the mapping created by fxMapRequestHandler , this will be the value of which FXRequestHandlerCallback returns or FX_IBD_TRANSACTION_BUS_RESET if bus reset events are detected before a response packet is transmitted.

Synopsis

```
typedef int32_t (*FXRequestNotificationCallback) (
    FXBusHandle          busHandle,
    void*                userData,
    FXMappingHandle      mappingHandle,
    int32_t              result
);
```

10.2.3. Structures

10.2.3.1. FXTransactionData

Description

This structure defines data members that will be used to hold a request packet that has been received by the stack. The structure is also used to carry data with user-allocated memory buffer if applicable.

Members

generation	The current bus generation number. (see fxGetBusGeneration)
sourceID	Node ID of the IEEE1394 device who has initiated sending a request
transactionLabel	Automatically assigned by the FireStack. Each transaction has a unique label.
accessType	Indicates transaction mode (read, write or lock). For definition, refer to Transaction Types .
localAddress	48-bit address of memory space specified in IEEE1394 standard
data	When the handler callback (FXRequestHandlerCallback) is called, this member points to the stack internal receive buffer that contains received packet data. Before returning from the handler callback for read and lock transactions, allocate DMA-capable memory buffer and assign the data variable and set the corresponding size. The DMA memory must not be freed until the response process is complete. This is signaled by the notification event (FXRequestNotificationCallback).
size	Received data size in bytes. Also used to specify the size of DMA-capable memory allocated by the user.
lockOperation	Indicates the type of the lock operation (compare_swap,

	mask_swap, etc) if the accessType is the lock transaction.
--	--

Synopsis

```
typedef struct {
    uint32_t          sourceID;
    uint32_t          accessType;
    FXAddress64      localAddress;
    void*            data;
    size_t           size;
    uint32_t         lockOperation;
} FXTransactionData;
```

10.2.3.2. FXMappingOptions**Description**

This structure defines the components of a memory region along with permitted access mode and a user-specified callback function which will be called when the stack completes the transaction.

Members

startAddress	48-bit address of memory space specified in IEEE1394 standard
size	Memory size in bytes
accessTypeMask	Specify access mode (read, write or lock) that are allowed for the memory region. For definition, refer to Transaction Types .
callback	User-specified callback function that will be called upon a completion of a transaction with address and size are within the memory region.
userData	Pointer to a user-specified data. The pointer will be carried to user callback functions. See also FXRequestNotificationCallback .

Synopsis

```
typedef struct {
    FXAddress64      startAddress;
    size_t           size;
    uint32_t         accessTypeMask;
    FXRequestNotificationCallback callback;
    void*            userData;
} FXMappingOptions;
```

10.2.4. Constants**10.2.4.1. Error Codes**

The following values may be returned by Inbound Transactions functions.

```
FX_ERR_IBD_TRN_INVALID_MAPPING_HANDLE
FX_ERR_IBD_TRN_MAX_MAPPINGS_REACHED
FX_ERR_IBD_TRN_MAPPING_OVERLAPPING
FX_ERR_IBD_TRN_INTERNAL_ERROR
```

10.2.4.2. Response Codes

A return value of [FXRequestHandlerCallback](#) must be one of the following codes except `FX_IBD_TRANSACTION_BUS_RESET`. The return value will inform the FireStack what to do when it attempts transmitting a response packet. Returning `FX_IBD_TRANSACTION_ABORT` will result in not transmitting the response packet. The return value will be copied to the 'result' parameter of [FXRequestNotificationCallback](#) after the response process is completed except that if bus reset events are detected during the process, the result parameter will be `FX_IBD_TRANSACTION_BUS_RESET` (the response packet was NOT transmitted).

FX_IBD_TRANSACTION_ABORT	-2
FX_IBD_TRANSACTION_BUS_RESET	-1
FX_IBD_TRANSACTION_NORMAL_COMPLETE	0
FX_IBD_TRANSACTION_DATA_ERROR	1
FX_IBD_TRANSACTION_TYPE_ERROR	2
FX_IBD_TRANSACTION_ADDRESS_ERROR	3
FX_IBD_TRANSACTION_CONFLICT_ERROR	4

10.3. Outbound Transactions

This module can be used to perform memory transactions on remote nodes. Available memory transactions are read, write and lock.

For memory read and write transactions a block request will be used when more than 4 bytes are requested and a quadlet request is performed when exactly 4 bytes are requested. It is up to the user to determine if the target node supports block transactions or not.

When a memory transaction is initiated FireStack will determine the maximum speed to the destination node by performing the needed PHY remote accesses. FireStack will remember and reuse already determined speeds until the next bus reset.

As each bus reset potentially changes the node ID assignments FireStack maintains a generation number that increments each time a bus reset occurs. All information related to node IDs and topology is only valid for the duration of a single bus generation. Therefore, before the user can make use of any of the functions for retrieving topology information or functions referring nodeIDs the user must request current bus generation from FireStack by calling `fxGetBusGeneration()`. It is recommended to register a bus reset callback and use that as a trigger to update the generation number and topology information.

Outbound Transactions can be used in the following ways with respect to result indication:

- **Blocking Mode:** After transmitting the specified request packet, the FireStack Transaction functions will not return unless it received a response packet/code or waited for the specified amount of time (see [FXTransactionOptions](#)). Blocking mode is entered when no transaction handle parameter is provided when calling one of the transaction functions. The callback function member in [FXTransactionOptions](#) will be ignored in this mode.
- **Non-Blocking Mode with callback:** After transmitting the specified request packet, the FireStack Transaction functions will return immediately. The result of the transaction will be returned in the callback function which will be called by the FireStack upon a completion of the transaction. This mode is entered when both a transaction handle parameter and a callback parameter are passed when calling one of the transaction functions. The user does not have to clear the transaction.
- **Non-blocking mode without callback:** After transmitting the specified request packet, the FireStack Transaction functions will return immediately. The status of the transaction needs to be polled by the user. Once the status indicates the transaction completed, potentially with an error, the user is required to clear the transaction.

If certain requirements are met FireStack will perform Outbound Transactions in zero-copy mode otherwise stack will copy data to non-DMA memory.

- Zero-copy read transactions will only be performed in non-blocking mode with callback if the user specifies a zero pointer as buffer for the received data. FireStack will hand a pointer to the actual receive buffer to the user when calling the registered callback. The user is free to use this buffer during callback execution. As soon as the callback returns FireStack will reuse the buffer for reception of new packets. When the read transaction is performed in any other way FireStack will copy the received data to the specified memory location.
- Write transactions are always performed in a zero-copy fashion. Therefore, the user needs to pass a DMA-capable memory buffer as data. The user should not free or reuse the memory buffer during the duration of the transaction. As soon as the transaction finishes the user retrieves control of the memory buffer.
- Lock transactions are never performed in a zero-copy fashion as the involved data size is relatively small.

10.3.1. Functions

10.3.1.1. *fxReadTransaction*

Description

This function sends a read request packet to the specified destination node and waits to receive a response packet.

Parameters

busHandle	Reference handle to the bus to control. (see fxCreateBusHandle)
generation	Set this value to the current bus generation number. (see fxGetBusGeneration)
destNode	Node ID of destination device.
offset	The start address of the IEEE1394 memory space of the destination node from which this transaction intends to read.
buffer	User-specified buffer address. If set, data on a response packet will be copied to this buffer. For non-blocking mode buffer may be set to zero to operate in zero-copy mode. This case a pointer directly into the reception buffer will be passed to the user upon transaction completion.
size	Specify the maximum size of the desired receive data size. The buffer specified above must have enough storage space.
options	Pointer to FXTransactionOptions .
transactionHandle	Filled with a new, valid transaction handle if no error is reported. For block mode, specify zero as pointer to a handle.

Return Codes

This function returns zero or greater upon success. A negative return value indicates an error. Please use the function [fxGetErrorMessage](#) to look up descriptions corresponding to negative return values.

FX_ERR_INVALID_HANDLE
FX_ERR_INVALID_PARAMETER
FX_ERR_INVALID_ADDRESS
FX_ERR_OBD_TRN_OPERATIONTIMEOUT
FX_ERR_INVALID_LOCAL_NODE_ID
FX_ERR_BUS_RESET_DETECT
FX_ERR_INTERNAL_ERROR
FX_ERR_LICENSE_MODULE
FX_ERR_LICENSE_EXPIRED
FX_ERR_FIRESTACK_DEMO_TIMEOUT
FX_ERR_OBD_TRN_RCV_RESP_CONFLICT
FX_ERR_OBD_TRN_RCV_RESP_DATA
FX_ERR_OBD_TRN_RCV_RESP_TYPE
FX_ERR_OBD_TRN_RCV_RESP_ADDRESS
FX_ERR_OBD_TRN_MISSING_ACK
FX_ERR_OBD_TRN_RETRY_LIMIT_EXCEED

Synopsis

```

FXReturnCode fxReadTransaction(
    FXBusHandle          busHandle,
    uint32_t             generation,
    uint32_t             destNode,
    const FXAddress64*   offset,
    void*                buffer,
    size_t               size,
    const FXTransactionOptions* options,
    FXTransactionHandle* transactionHandle
)

```

10.3.1.2. fxWriteTransaction

Description

This function sends a write request packet to the specified destination node and waits to receive a response packet.

Parameters

busHandle	Reference handle to the bus to control. (see fxCreateBusHandle)
generation	Set this value to the current bus generation number. (see fxGetBusGeneration)
destNode	Node ID of destination device.
offset	The start address of the IEEE1394 memory space of the destination node to which this transaction intends to write.
buffer	User-specified buffer address. The buffer needs to be allocated with fxMemAlloc() .
size	Specify the size of data this function writes.
options	Pointer to FXTransactionOptions
transactionHandle	Filled with a new, valid transaction handle if no error is reported. For block mode, specify zero as pointer to a handle.

Return Codes

This function returns zero or greater upon success. A negative return value indicates an error. Please use the function [fxGetErrorMessage](#) to look up descriptions corresponding to negative return values.

```

FX_ERR_INVALID_HANDLE
FX_ERR_INVALID_ADDRESS
FX_ERR_OBD_TRN_OPERATIONTIMEOUT
FX_ERR_INVALID_LOCAL_NODE_ID
FX_ERR_BUS_RESET_DETECT
FX_ERR_INTERNAL_ERROR
FX_ERR_LICENSE_MODULE
FX_ERR_LICENSE_EXPIRED
FX_ERR_FIRESTACK_DEMO_TIMEOUT
FX_ERR_OBD_TRN_INVALID_BUFFER_ADDRESS
FX_ERR_OBD_TRN_RCV_RESP_CONFLICT
FX_ERR_OBD_TRN_RCV_RESP_DATA
FX_ERR_OBD_TRN_RCV_RESP_TYPE
FX_ERR_OBD_TRN_RCV_RESP_ADDRESS
FX_ERR_OBD_TRN_MISSING_ACK
FX_ERR_OBD_TRN_RETRY_LIMIT_EXCEED

```

Synopsis

```

FXReturnCode fxWriteTransaction(
    FXBusHandle          busHandle,
    uint32_t            generation,
    uint32_t            destNode,
    const FXAddress64*  offset,
    void*               buffer,
    size_t              size,
    const FXTransactionOptions* options,
    FXTransactionHandle* transactionHandle
)

```

10.3.1.3. *fxLockTransaction*

Description

This function sends a lock request packet to the specified destination node and waits to receive a response packet.

Parameters

busHandle	Reference handle to the bus to control. (see fxCreateBusHandle)
generation	Set this value to the current bus generation number. (see fxGetBusGeneration)
destNode	Node ID of destination device.
offset	The start address of the IEEE1394 memory space of the destination node.
operation	Type of lock operation.
reqBuffer	User-specified buffer address. Data to be sent with lock operation. The buffer needs to be allocated with fxMemAlloc() .
reqSize	Size of data.
respBuffer	User-specified buffer address. Pointer to data buffer to which result of lock operation is copied.
respSize	Pointer to size of the respBuffer.
options	Pointer to FXTransactionOptions
transactionHandle	Filled with a new, valid transaction handle if no error is reported. For block mode, specify zero as pointer to a handle.

Return Codes

This function returns zero or greater upon success. A negative return value indicates an error. Please use the function [fxGetErrorMessage](#) to look up descriptions corresponding to negative return values.

```

FX_ERR_INVALID_HANDLE
FX_ERR_INVALID_PARAMETER
FX_ERR_INVALID_ADDRESS
FX_ERR_OBD_TRN_OPERATIONTIMEOUT
FX_ERR_INVALID_LOCAL_NODE_ID
FX_ERR_BUS_RESET_DETECT
FX_ERR_INTERNAL_ERROR
FX_ERR_LICENSE_MODULE
FX_ERR_LICENSE_EXPIRED
FX_ERR_FIRESTACK_DEMO_TIMEOUT
FX_ERR_OBD_TRN_INVALID_BUFFER_ADDRESS
FX_ERR_OBD_TRN_RCV_RESP_CONFLICT
FX_ERR_OBD_TRN_RCV_RESP_DATA
FX_ERR_OBD_TRN_RCV_RESP_TYPE
FX_ERR_OBD_TRN_RCV_RESP_ADDRESS
FX_ERR_OBD_TRN_MISSING_ACK
FX_ERR_OBD_TRN_RETRY_LIMIT_EXCEED

```

Synopsis

```

FXReturnCode fxLockTransaction(
    FXBusHandle          busHandle,
    uint32_t             generation,
    uint32_t             destNode,

```

```

    const FXAddress64*      offset,
    uint32_t                operation,
    void*                   reqBuffer,
    size_t                  reqSize,
    void*                   respBuffer,
    size_t                  respSize,
    const FXTransactionOptions* options,
    FXTransactionHandle*    transactionHandle
)

```

10.3.1.4. *fxClearTransaction*

Description

This function cancels and deletes the transaction.

Parameters

transactionHandle	Reference handle to the outbound transaction data. (see fxReadTransaction , fxWriteTransaction , fxLockTransaction)
-------------------	--

Return Codes

This function returns zero or greater upon success. A negative return value indicates an error. Please use the function [fxGetErrorMessage](#) to look up descriptions corresponding to negative return values.

FX_ERR_OBD_TRN_INVALID_TRANSACTION_HANDLE

Synopsis

```

FXReturnCode fxClearTransaction(
    FXTransactionHandle    transactionHandle
)

```

10.3.1.5. *fxClearAllTransactions*

Description

This function cancels and deletes all existing transactions.

Parameters

busHandle	Reference handle to the bus to control. (see fxCreateBusHandle)
-----------	--

Return Codes

This function returns zero or greater upon success. A negative return value indicates an error. Please use the function [fxGetErrorMessage](#) to look up descriptions corresponding to negative return values.

FX_ERR_INVALID_HANDLE

Synopsis

```

FXReturnCode fxClearAllTransactions(
    FXBusHandle    busHandle
)

```

10.3.1.6. *fxGetTransactionStatus*

Description

This function returns the status and other attributes of the transaction.

Parameters

transactionHandle	Reference handle to the outbound transaction data. (see fxReadTransaction , fxWriteTransaction , fxLockTransaction)
-------------------	--

info	See FXTransactionInfo .
------	---

Return Codes

This function returns zero or greater upon success. A negative return value indicates an error. Please use the function [fxGetErrorMessage](#) to look up descriptions corresponding to negative return values.

FX_ERR_OBD_TRN_INVALID_TRANSACTION_HANDLE**Synopsis**

```
FXReturnCode fxGetTransactionStatus(
    FXTransactionHandle      transactionHandle,
    FXTransactionInfo*      info
)
```

10.3.1.7. fxGetNumTransactions**Description**

This function will query for active transactions. A list of the transactions can be retrieved by calling [FxGetTransactionList](#).

Parameters

busHandle	Reference handle to the bus to control. (see fxCreateBusHandle)
numTransactions	Returns the number of transactions.

Return Codes

This function returns zero or greater upon success. A negative return value indicates an error. Please use the function [fxGetErrorMessage](#) to look up descriptions corresponding to negative return values.

FX_ERR_INVALID_HANDLE
FX_ERR_LICENSE_MODULE
FX_ERR_LICENSE_EXPIRED
FX_ERR_FIRESTACK_DEMO_TIMEOUT

Synopsis

```
FXReturnCode fxGetNumTransactions(
    FXBusHandle      busHandle,
    uint32_t*        numTransactions
)
```

10.3.1.8. fxGetTransactionList**Description**

This function may be called after calling to [FxGetNumTransactions](#) to get an array of [FXTransactionInfo](#) structures.

The user needs to take care of allocating an array and specifying its size when calling this function.

Parameters

busHandle	Reference handle to the bus to control. (see fxCreateBusHandle)
list	Returns the number of transactions.
maxSize	The number of FXTransactionInfo structures that fit in the list.
size	The actual number of FXTransactionInfo structures returned.

Return Codes

This function returns zero or greater upon success. A negative return value indicates an error. Please use the

function [fxGetErrorMessage](#) to look up descriptions corresponding to negative return values.

```

FX_ERR_INVALID_HANDLE
FX_ERR_LICENSE_MODULE
FX_ERR_LICENSE_EXPIRED
FX_ERR_FIRESTACK_DEMO_TIMEOUT

```

Synopsis

```

FXReturnCode fxGetTransactionList(
    FXBusHandle          busHandle,
    FXTransactionInfo*   list,
    uint32_t             maxSize,
    uint32_t*           size
)

```

10.3.2. Type Definitions

10.3.2.1. FXTransactionCompleteCallback

Description

This type definition defines the function prototype for transaction completion indication. If a read transaction was performed in zero-copy mode, buffer points directly to the data of the received packet and shall only be referred to during the callback. In any other case the buffer member is a memory pointer provided by user when initiating the transaction.

Parameters

busHandle	Reference handle to the bus to control. (see fxCreateBusHandle)
userData	Pointer to the data specified in FXTransactionOptions .
transactionHandle	Reference handle to the outbound transaction data. (see fxReadTransaction , fxWriteTransaction , fxLockTransaction)
result	Error status, response code, etc.
size	Size of the received data buffer or the max size specified whichever smaller.
buffer	Buffer pointer either to the buffer specified by the user or to the FireStack internal data buffer. (see fxReadTransaction)

Synopsis

```

typedef int32_t (*FXTransactionCompleteCallback) (
    FXBusHandle          busHandle,
    void*               userData,
    FXTransactionHandle  transactionHandle,
    uint32_t             result,
    size_t              size,
    void*               buffer
);

```

10.3.3. Structures

10.3.3.1. FXTransactionOptions

Description

This structure defines options for an outbound transaction.

Members

timeout	The FireStack will wait for the specified time (in
---------	--

	milliseconds) until the destination sends a response packet.
callback	Callback function (see FXTransactionCompleteCallback). In blocking mode, this pointer will not be used.
userData	Pointer to a user-specified data. The pointer will be carried to user callback functions. See also FXTransactionCompleteCallback .
speedMode	Selects the speed mode for the outbound transaction. Value shall either be FX_SPEED_TYPE_AUTO or FX_SPEED_TYPE_FIXED or-ed together with one of the speed macros FX_SPEED_100 to FX_SPEED_3200.

Synopsis

```
typedef struct {
    uint32_t                timeout;
    FXTransactionCompleteCallback callback;
    void*                  userData;
    uint32_t                speedMode;
} FXTransactionOptions;
```

10.3.3.2. FXTransactionInfo**Description**

This structure represents the current status of an outbound transaction.

Members

transactionHandle	Reference handle to the outbound transaction data. (see fxReadTransaction , fxWriteTransaction , fxLockTransaction)
nodeId	Node ID to which the request was sent.
accessMode	See Transaction Types .
status	See Transaction Status .

Synopsis

```
typedef struct {
    FXTransactionHandle transactionHandle;
    uint32_t            nodeId;
    uint32_t            accessMode;
    uint32_t            status;
} FXTransactionInfo;
```

10.3.4. Constants**10.3.4.1. Error Codes**

The following values may be returned by Outbound Transactions functions.

```
FX_ERR_OBD_TRN_INVALID_TRANSACTION_HANDLE
FX_ERR_OBD_TRN_OPERATIONTIMEOUT
FX_ERR_OBD_TRN_INVALID_BUFFER_ADDRESS
FX_ERR_OBD_TRN_RCV_RESP_CONFLICT
FX_ERR_OBD_TRN_RCV_RESP_DATA
FX_ERR_OBD_TRN_RCV_RESP_TYPE
FX_ERR_OBD_TRN_RCV_RESP_ADDRESS
FX_ERR_OBD_TRN_MISSING_ACK
FX_ERR_OBD_TRN_RETRY_LIMIT_EXCEED
```

10.3.4.2. Transaction Status

FX_OBD_TRANSACTION_INACTIVE	0
FX_OBD_TRANSACTION_WAITING	1
FX_OBD_TRANSACTION_SUCCESS	2
FX_OBD_TRANSACTION_TIMEOUT	4
FX_OBD_TRANSACTION_BUSRESET	8
FX_OBD_TRANSACTION_RESPONSE_CONFLICT_ERR	20
FX_OBD_TRANSACTION_RESPONSE_DATA_ERR	21
FX_OBD_TRANSACTION_RESPONSE_TYPE_ERR	22
FX_OBD_TRANSACTION_RESPONSE_ADDRESS_ERR	23
FX_OBD_TRANSACTION_MISSING_ACK_ERR	24
FX_OBD_TRANSACTION_RETRY_LIMIT_EXCEED_ERR	25

10.4. Isochronous Reception

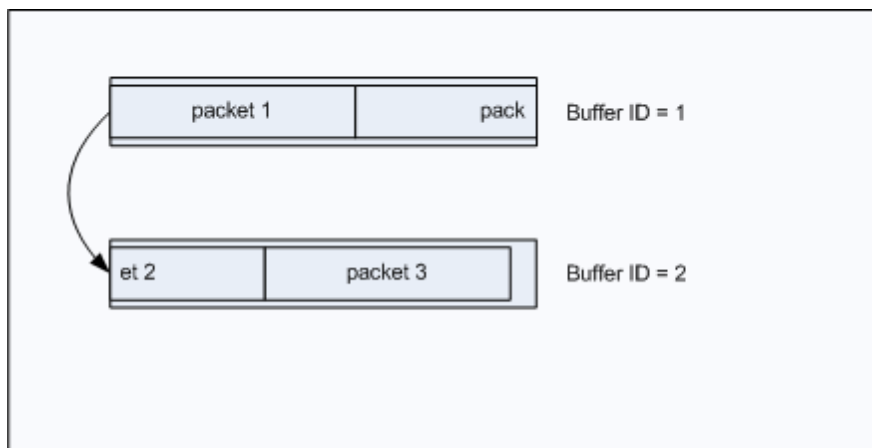
This section forms the description of all functions and data structures needed for receiving isochronous packets.

Followings should be considered and decided when a new context is created by [fxIsoRcvCreateContextHandle](#) function.

1. Buffer mode (see below)
2. A channel number the new context will look for the reception. Multi-channel reception may be possible if available; packets with more than one channel number can be received in one context program.
3. Choice of storing header/trailer. (see [Context Options](#) and [Data Formats](#))
4. Other filtering options: tag values, etc. (see [Context Options](#))
5. Running options - a context can wait for certain conditions before it starts receiving packets: sync value.

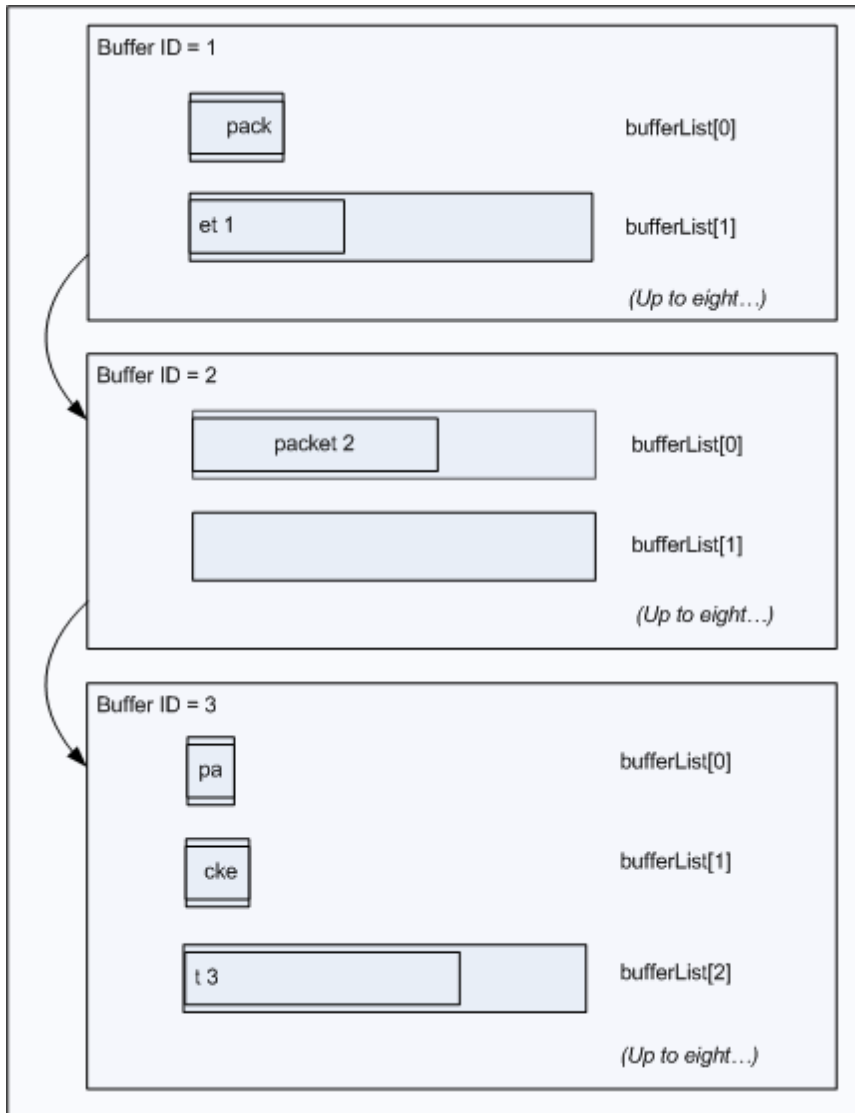
The Isochronous Reception module supports three different buffer modes: Buffer-Fill mode, Packet-per-Buffer mode, and Dual-Buffer mode. A receive context can be created by selecting one of the buffer modes, and each buffer mode has a dedicated adding-buffers function. When a new context is created, a user must choose a buffer mode in which the context will run. The buffer mode, other context options, and event options must be selected at the time of creation of a context; a user must create a new context if a context with different options is needed.

- **Buffer-Fill mode:** In this mode, all received packets are concatenated into a contiguous stream of data and fill each buffer completely. Packets may straddle multiple buffers in this mode.



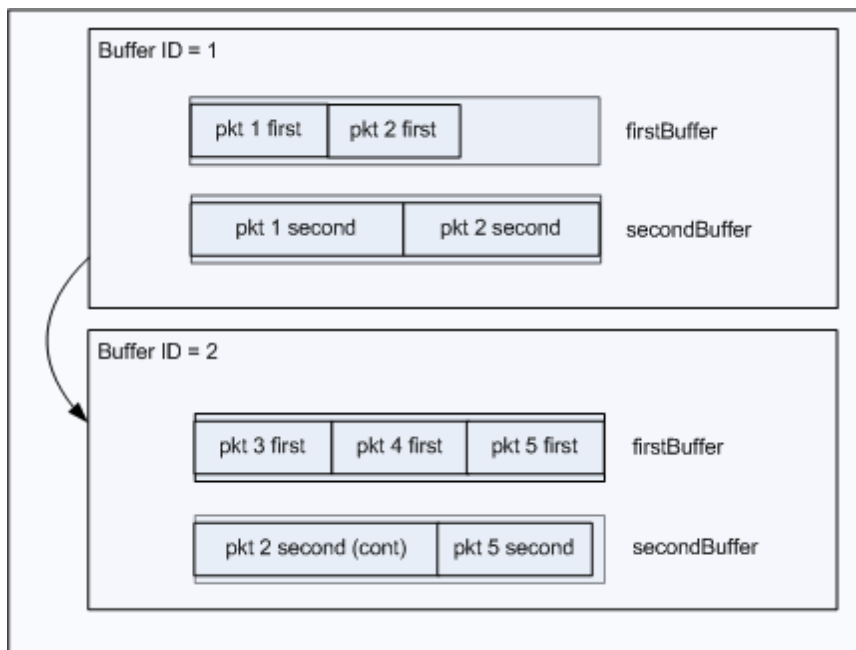
Buffer Fill Mode illustration

- **Packet-per-Buffer mode:** Each received packet is stored in the buffer block added to the context by calling the add-buffer(s) function for this mode. Any leftover data are discarded, and packets never straddle into another buffer block which is added to the same context by calling another add-buffer(s) function. Each buffer block consists of 1 to 8 separate buffers.



Packet-per-buffer Mode illustration

- Dual-Buffer mode:** When an isochronous receive context is in dual-buffer mode, all received packets are viewed as containing a first portion of the payload followed by a second portion. The dual-buffer mode operations are similar to buffer-fill mode, but provide two separate series of buffers to stream isochronous packet data: firstBuffer series and secondBuffer series. In this buffer mode, a buffer will be retired when either the firstBuffer or secondBuffer has been filled by packet data. FirstBuffer data will not span a buffer. A user must set up first data buffers in multiples of firstEachSize (including header and trailer quadlets if the store-header option is enabled). The illustration below shows a sequence of varying length. The first buffer is retired after packet 2 second data payload has spanned the second buffer, and the second buffer is retired after packet 5 first data completely fills the first data buffer. Note that the context may receive packets with empty second portions (i.e., only first data payload), and this is illustrated in the packet 3 and 4 below.



Dual-Buffer Mode illustration

10.4.1. Feature Inquiry Functions

10.4.1.1. *fxIsoRcvGetNumberOfContexts*

Description

This function returns the number of the receive contexts the stack supports.

Parameters

busHandle	Reference handle to the bus to control. (see fxCreateBusHandle)
pNumber	Address of integer value to which number of the receive contexts will be returned.

Return Codes

This function returns zero or greater upon success. A negative return value indicates an error. Please use the function [fxGetErrorMessage](#) to lookup descriptions corresponding to negative return values.

FX_ERR_INVALID_HANDLE
FX_ERR_INVALID_PARAMETER

Synopsis

```
FXReturnCode fxIsoRcvGetNumberOfContexts(
    FXBusHandle          busHandle,
    uint32_t*            pNumber
)
```

10.4.2. Reception Functions

10.4.2.1. Context Control

10.4.2.1.1 fxIsoRcvCreateContextHandle

Description

This function create an isochronous receive context with specified buffer mode, event options, and context options.

Parameters

busHandle	Reference handle to the bus to control. (see fxCreateBusHandle)
bufferMode	Specify one of buffer modes. (see Buffer Modes)
pEventOptions	Pointer to FXIsoRcvEventOptions .
pContextOptions	Pointer to FXIsoRcvOption . For available options please refer to Context Options . Options that not specified will just use default values.
optionSize	Specifies number of items in contextOptions.
pContextHandle	A pointer of handle to which a new receive context handle will be stored.

Return Codes

This function returns zero or greater upon success. A negative return value indicates an error. Please use the function [fxGetErrorMessage](#) to lookup descriptions corresponding to negative return values.

FX_ERR_INVALID_HANDLE
FX_ERR_INVALID_PARAMETER
FX_ERR_ISO_RCV_FEATURE_NOT_SUPPORTED
FX_ERR_ISO_RCV_MULTICHANNEL_BUFFER_MISMATCH
FX_ERR_ISO_RCV_MULTICHANNEL_NOT_AVAILABLE
FX_ERR_ISO_RCV_NO_AVAILABLE_CONTEXT

Synopsis

```
FXReturnCode fxIsoRcvCreateContextHandle(
    FXBusHandle          busHandle,
    uint32_t             bufferMode,
    FXIsoRcvEventOptions* pEventOptions,
    FXIsoRcvOption*      pContextOptions,
    size_t               optionSize,
    FXIsoRcvContextHandle* pContextHandle
)
```

10.4.2.1.2 fxIsoRcvCloseContextHandle

Description

This function closes specified context and frees its resources.

Parameters

contextHandle	Reference handle to the context to control. (see fxIsoRcvCreateContextHandle)
---------------	--

Return Codes

This function returns zero or greater upon success. A negative return value indicates an error. Please use the function [fxGetErrorMessage](#) to lookup descriptions corresponding to negative return values.

FX_ERR_ISO_RCV_INVALID_CONTEXT_HANDLE

Synopsis


```
FXReturnCode fxIsoRcvCloseContextHandle(
    FXIsoRcvContextHandle contextHandle
)
```

10.4.2.1.3 fxIsoRcvStartContext

Description

This function starts data reception with specific context.

Parameters

contextHandle	Reference handle to the context to control. (see fxIsoRcvCreateContextHandle)
bufferID	Buffer ID at which the context will start reception.

Return Codes

This function returns zero or greater upon success. A negative return value indicates an error. Please use the function [fxGetErrorMessage](#) to lookup descriptions corresponding to negative return values.

FX_ERR_ISO_RCV_INVALID_CONTEXT_HANDLE
FX_ERR_ISO_RCV_BUFFER_NOT_FOUND
FX_ERR_ISO_RCV_CONTEXT_MISMATCH
FX_ERR_INTERNAL_ERROR

Synopsis

```
FXReturnCode fxIsoRcvStartContext(
    FXIsoRcvContextHandle contextHandle,
    uint32_t bufferID
)
```

10.4.2.1.4 fxIsoRcvStopContext

Description

This function stops data reception with specific context.

Parameters

contextHandle	Reference handle to the context to control. (see fxIsoRcvCreateContextHandle)
---------------	--

Return Codes

This function returns zero or greater upon success. A negative return value indicates an error. Please use the function [fxGetErrorMessage](#) to lookup descriptions corresponding to negative return values.

FX_ERR_ISO_RCV_INVALID_CONTEXT_HANDLE

Synopsis

```
FXReturnCode fxIsoRcvStopContext(
    FXIsoRcvContextHandle contextHandle
)
```

10.4.2.1.5 fxIsoRcvContextStatus

Description

This function returns status of the specified context.

Parameters

contextHandle	Reference handle to the context to control. (see fxIsoRcvCreateContextHandle)
status	Pointer to FXIsoRcvContextStatus .

Return Codes

This function returns zero or greater upon success. A negative return value indicates an error. Please use the function [fxGetErrorMessage](#) to lookup descriptions corresponding to negative return values.

FX_ERR_ISO_RCV_INVALID_CONTEXT_HANDLE
FX_ERR_INVALID_PARAMETER

Synopsis

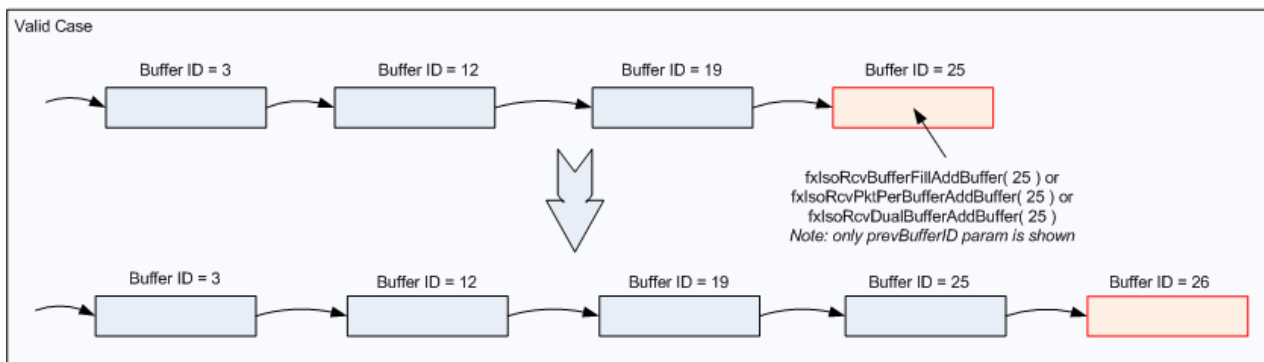
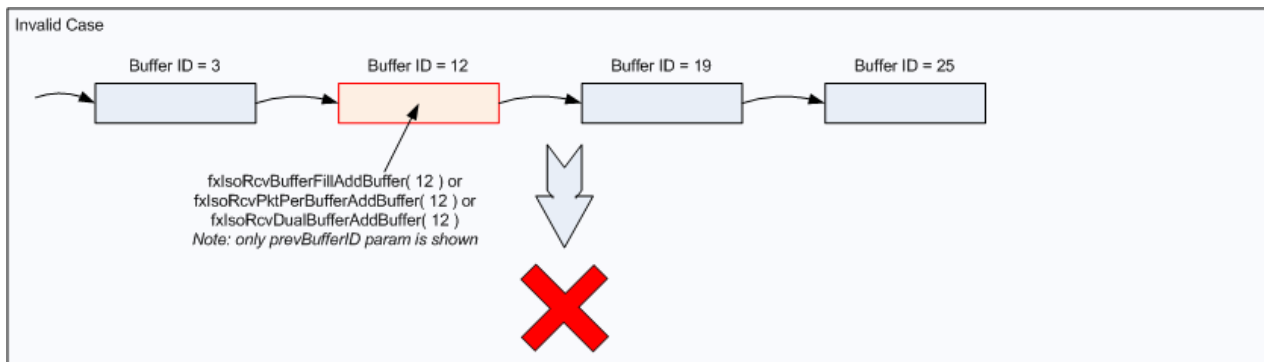
```
FXReturnCode fxIsoRcvContextStatus (
    FXIsoRcvContextHandle contextHandle,
    FXIsoRcvContextStatus* status
)
```

10.4.2.2. Buffer Control

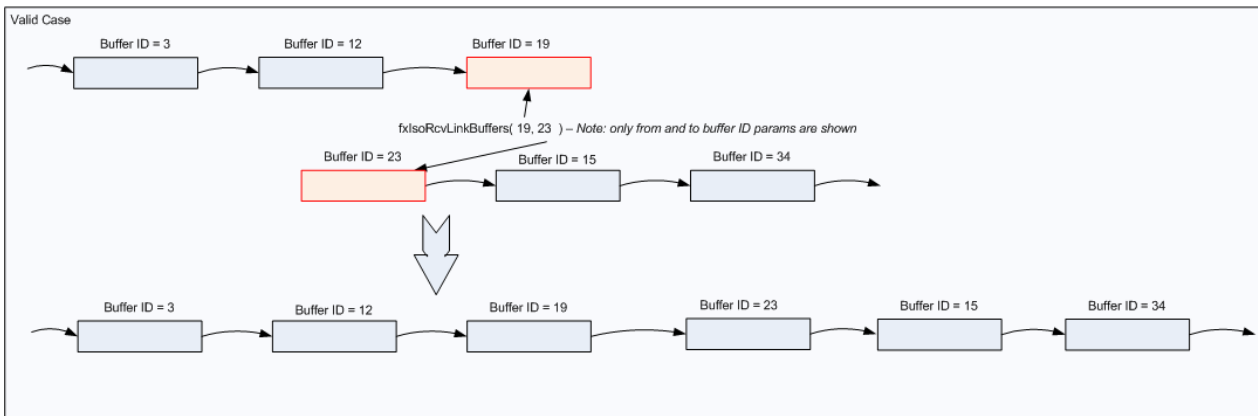
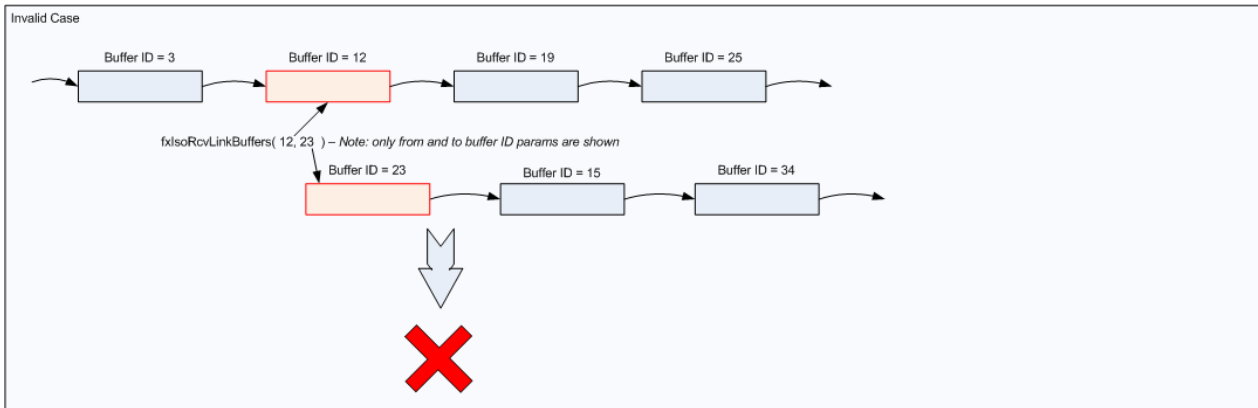
In order to start a context for reception, the user first needs to setup the necessary buffers for packet storage. Buffer Control functions can be used to register memory buffers with the Mil1394 reception module. The user needs to take care of allocating memory blocks that can be used as reception buffer. As long as a piece of memory is registered as reception buffer, the user may not free it or write to it. The user may read from it at all times. After removing a buffer from the reception list, the user may write and/or free memory again. Buffers need to be setup such that they form a list. It is not allowed to link buffers in a loop. Having buffers in a loop fashion would yield unspecified results.

Multiple lists can be setup in memory and when starting a context for reception, a specific buffer can be used as starting point for storing the packets. Please also note that adding the same buffer to more than one list yields unspecified results.

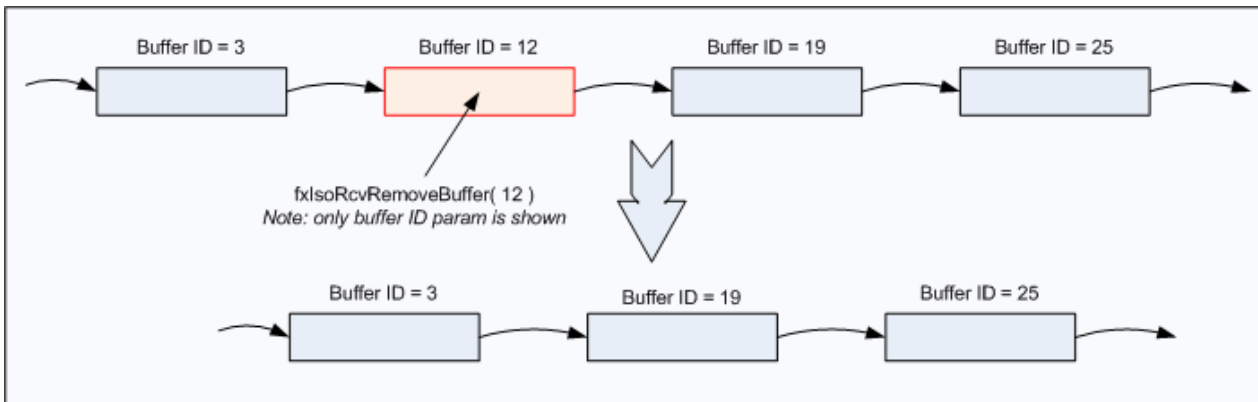
Please refer to [Data Formats](#) for a detailed specification of the received data format.



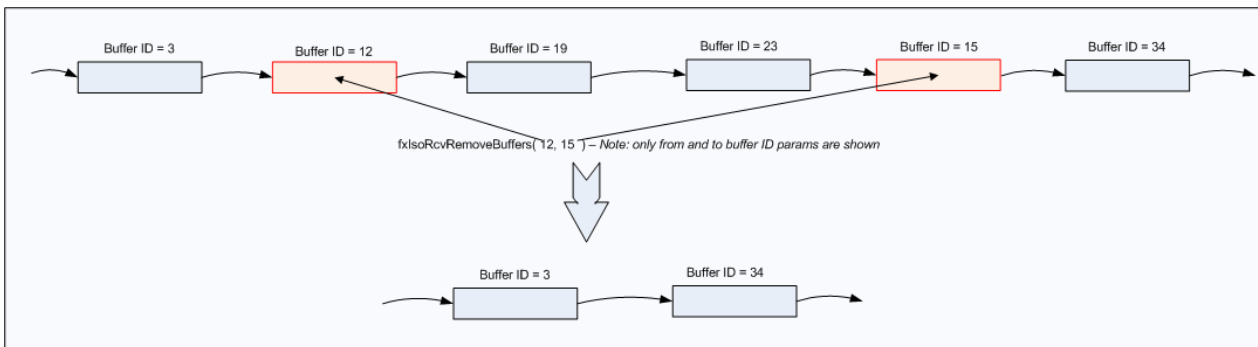
Add a new buffer to existing list - previous buffer must be the last one of the list



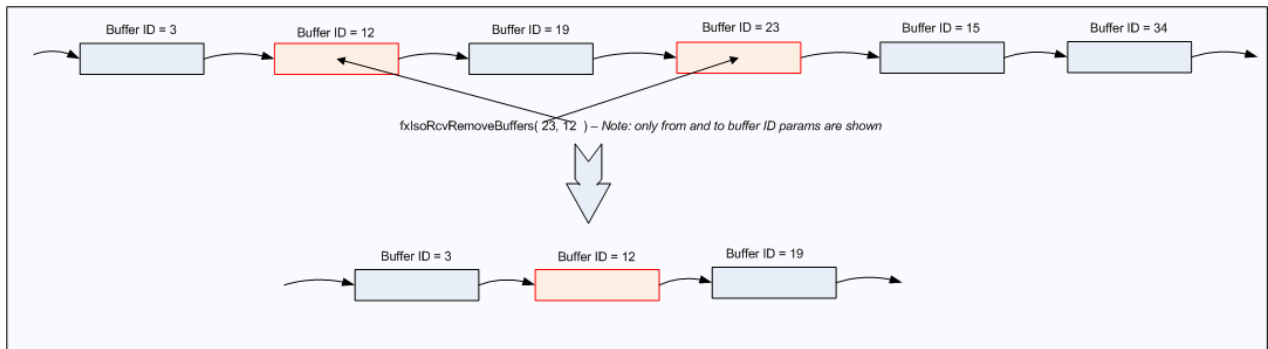
Link two lists of buffers - "From" must be the last one of one list, and "To" must be the first one of another list



Remove a buffer from the list



Remove buffers from the list



Remove buffers from the list - toBuffer is NOT linked after fromBuffer - remove from fromBuffer till the end of buffer list

10.4.2.2.1 fxIsoRcvBufferFillAddBuffer

Description

This function adds a new receive buffer to the specified context that is created in the [Buffer-Fill mode](#). If prevBufferID is set, prevBuffer's forward link will point to the new buffer.

Parameters

contextHandle	Reference handle to the context to control. Must be created by fxIsoRcvCreateContextHandle with FX_ISO_RCV_CNTX_MODE_BUFF_FILL buffer mode.
prevBufferID	0 (zero) specifies first buffer in list.
bufferOptions	User defined list of FXIsoRcvOption elements that together form the options for the context. For available options please refer to Buffer Options . Options that not specified will just use default values.
optionsSize	Specifies the number of items in the bufferOptions.
buffer	Pointer to a DMA-capable buffer allocated by the user. (see fxMemAlloc for allocating DMA-capable buffers)
size	Size of buffer in bytes.
newBufferID	Filled with a new, valid bufferID if no error is reported.

Return Codes

This function returns zero or greater upon success. A negative return value indicates an error. Please use the function [fxGetErrorMessage](#) to lookup descriptions corresponding to negative return values.

FX_ERR_INVALID_HANDLE
FX_ERR_INVALID_PARAMETER
FX_ERR_ISO_RCV_INVALID_CONTEXT_HANDLE
FX_ERR_ISO_RCV_BUFFER_TYPE_MISMATCH
FX_ERR_ISO_RCV_BUFFER_NOT_FOUND
FX_ERR_ISO_RCV_CONTEXT_MISMATCH
FX_ERR_ISO_RCV_NO_AVAILABLE_BUFFER
FX_ERR_ISO_RCV_BUFFER_NOT_LAST_OF_LIST

Synopsis

```

FXReturnCode fxIsoRcvBufferFillAddBuffer(
    FXIsoRcvContextHandle contextHandle,
    uint32_t prevBufferID,
    FXIsoRcvOption\* bufferOptions,
    size_t optionsSize,
    void* buffer,
    size_t size,

```

```

        uint32_t*          newBufferID
    )

```

10.4.2.2.2 fxIsoRcvPktPerBufferAddBuffer

Description

This function adds a new receive buffer to the specified context that is created in the [Packet-per-Buffer mode](#). If prevBufferID is set, prevBuffer's forward link will point to the new buffer.

Parameters

contextHandle	Reference handle to the context to control. Must be created by fxIsoRcvCreateContextHandle with FX_ISO_RCV_CNTX_MODE_PKT_PER_BUFF buffer mode.
prevBufferID	0 (zero) specifies first buffer in list.
bufferOptions	User defined list of FXIsoRcvOption elements that together form the options for the context. For available options please refer to Buffer Options . Options that not specified will just use default values.
optionsSize	Specifies the number of items in the bufferOptions.
bufferList	User defined list (up to 8) of FXIsoRcvBuffer .
newBufferID	Filled with a new, valid bufferID if no error is reported.

Return Codes

This function returns zero or greater upon success. A negative return value indicates an error. Please use the function [fxGetErrorMessage](#) to lookup descriptions corresponding to negative return values.

```

FX_ERR_INVALID_HANDLE
FX_ERR_INVALID_PARAMETER
FX_ERR_ISO_RCV_INVALID_CONTEXT_HANDLE
FX_ERR_ISO_RCV_BUFFER_TYPE_MISMATCH
FX_ERR_ISO_RCV_BUFFER_NOT_FOUND
FX_ERR_ISO_RCV_CONTEXT_MISMATCH
FX_ERR_ISO_RCV_NO_AVAILABLE_BUFFER
FX_ERR_ISO_RCV_BUFFER_NOT_LAST_OF_LIST

```

Synopsis

```

FXReturnCode fxIsoRcvPktPerBufferAddBuffer(
    FXIsoRcvContextHandle    contextHandle,
    uint32_t                prevBufferID,
    FXIsoRcvOption*         bufferOptions,
    size_t                  optionsSize,
    FXIsoRcvBuffer          bufferList[8],
    uint32_t*               newBufferID
)

```

10.4.2.2.3 fxIsoRcvDualBufferAddBuffer

Description

This function adds a new receive buffer to the specified context that is created in the [Dual-Buffer mode](#). If prevBufferID is set, prevBuffer's forward link will point to the new buffer.

Parameters

contextHandle	Reference handle to the context to control. Must be created by fxIsoRcvCreateContextHandle with FX_ISO_RCV_CNTX_MODE_DUAL_BUFF buffer mode.
prevBufferID	0 (zero) specifies first buffer in list.

bufferOptions	User defined list of FXIsoRcvOption elements that together form the options for the context. For available options please refer to Buffer Options . Options that not specified will just use default values.
optionsSize	Specifies the number of items in the bufferOptions.
numFirst	Specifies the number of the beginning of packets to be stored in the first buffer.
firstEachSize	Specifies the maximum size of each packet fragment the first buffer will store. Must be multiple of 4 and at least 8 bytes.
firstBuffer	Pointer to a DMA-capable buffer allocated by the user. Must be large enough to store (numFirst * firstEachSize) bytes of data. (see fxMemAlloc for allocating DMA-capable buffers)
secondBuffer	Pointer to a DMA-capable buffer allocated by the user. Remaining portion of packets (after storing its first portion to the first buffer) will be stored to this buffer in buffer-fill fashion. (see fxMemAlloc for allocating DMA-capable buffers)
secondSize	Size of secondBuffer in bytes.
newBufferID	Filled with a new, valid bufferID if no error is reported.

Return Codes

This function returns zero or greater upon success. A negative return value indicates an error. Please use the function [fxGetErrorMessage](#) to lookup descriptions corresponding to negative return values.

FX_ERR_INVALID_HANDLE
FX_ERR_INVALID_PARAMETER
FX_ERR_ISO_RCV_INVALID_CONTEXT_HANDLE
FX_ERR_ISO_RCV_BUFFER_TYPE_MISMATCH
FX_ERR_ISO_RCV_BUFFER_NOT_FOUND
FX_ERR_ISO_RCV_CONTEXT_MISMATCH
FX_ERR_ISO_RCV_NO_AVAILABLE_BUFFER
FX_ERR_ISO_RCV_BUFFER_NOT_LAST_OF_LIST

Synopsis

```

FXReturnCode fxIsoRcvDualBufferAddBuffer(
    FXIsoRcvContextHandle    contextHandle,
    uint32_t                prevBufferID,
    FXIsoRcvOption*         bufferOptions,
    size_t                  optionsSize,
    size_t                  numFirst,
    size_t                  firstEachSize,
    void*                   firstBuffer,
    void*                   secondBuffer,
    size_t                  secondSize,
    uint32_t*               newBufferID
)

```

10.4.2.2.4 fxIsoRcvLinkBuffers

Description

This function links two buffer lists together. Two buffers must belong to the same context.

Parameters

contextHandle	Reference handle to the context to control. (see fxIsoRcvCreateContextHandle)
---------------	--

fromBufferID	toBufferID below will be appended to this bufferID. Must be the last buffer of a buffer list.
toBufferID	This bufferID will be appended to fromBufferID above. Must be the first buffer of a buffer list.

Return Codes

This function returns zero or greater upon success. A negative return value indicates an error. Please use the function [fxGetErrorMessage](#) to lookup descriptions corresponding to negative return values.

```

FX_ERR_ISO_RCV_INVALID_CONTEXT_HANDLE
FX_ERR_ISO_RCV_BUFFER_NOT_FOUND
FX_ERR_ISO_RCV_CONTEXT_MISMATCH
FX_ERR_ISO_RCV_BUFFER_NOT_FIRST_OF_LIST
FX_ERR_ISO_RCV_BUFFER_NOT_LAST_OF_LIST
FX_ERR_INTERNAL_ERROR

```

Synopsis

```

FXReturnCode fxIsoRcvLinkBuffers(
    FXIsoRcvContextHandle    contextHandle,
    uint32_t                fromBufferID,
    uint32_t                toBufferID
)

```

10.4.2.2.5 fxIsoRcvRemoveBuffer

Description

This function deletes the specified receive buffer from the receive context.

Parameters

contextHandle	Reference handle to the context to control. (see fxIsoRcvCreateContextHandle)
bufferID	Buffer to remove from the list.

Return Codes

This function returns zero or greater upon success. A negative return value indicates an error. Please use the function [fxGetErrorMessage](#) to lookup descriptions corresponding to negative return values.

```

FX_ERR_ISO_RCV_INVALID_CONTEXT_HANDLE
FX_ERR_ISO_RCV_CONTEXT_MISMATCH
FX_ERR_INTERNAL_ERROR

```

Synopsis

```

FXReturnCode fxIsoRcvRemoveBuffer(
    FXIsoRcvContextHandle    contextHandle,
    uint32_t                bufferID
)

```

10.4.2.2.6 fxIsoRcvRemoveBuffers

Description

This function deletes the specified receive buffers from the receive context.

Parameters

contextHandle	Reference handle to the context to control. (see fxIsoRcvCreateContextHandle)
fromBufferID	First buffer to remove from the list.
toBufferID	Last buffer to remove from the list.

Return Codes

This function returns zero or greater upon success. A negative return value indicates an error. Please use the function [fxGetErrorMessage](#) to lookup descriptions corresponding to negative return values.

FX_ERR_ISO_RCV_INVALID_CONTEXT_HANDLE
FX_ERR_ISO_RCV_CONTEXT_MISMATCH
FX_ERR_INTERNAL_ERROR

Synopsis

```
FXReturnCode fxIsoRcvRemoveBuffers(
    FXIsoRcvContextHandle    contextHandle,
    uint32_t                fromBufferID,
    uint32_t                toBufferID
)
```

10.4.2.2.7 fxIsoRcvBufferFillBufferStatus

Description

This function returns the status of the specified buffer created by [fxIsoRcvBufferFillAddBuffer](#).

Parameters

contextHandle	Reference handle to the context to control. (see fxIsoRcvCreateContextHandle)
bufferID	ID of the receive buffer.
pStatus	Pointer to FXIsoRcvBufferFillBufferStatus .

Return Codes

This function returns zero or greater upon success. A negative return value indicates an error. Please use the function [fxGetErrorMessage](#) to lookup descriptions corresponding to negative return values.

FX_ERR_ISO_RCV_INVALID_CONTEXT_HANDLE
FX_ERR_INTERNAL_ERROR
FX_ERR_ISO_RCV_BUFFER_NOT_FOUND
FX_ERR_ISO_RCV_CONTEXT_MISMATCH
FX_ERR_INVALID_PARAMETER

Synopsis

```
FXReturnCode fxIsoRcvBufferFillBufferStatus(
    FXIsoRcvContextHandle    contextHandle,
    uint32_t                bufferID,
    FXIsoRcvBufferFillBufferStatus\* pStatus
)
```

10.4.2.2.8 fxIsoRcvPktPerBufferBufferStatus

Description

This function returns the status of the specified buffer created by [fxIsoRcvPktPerBufferAddBuffer](#).

Parameters

contextHandle	Reference handle to the context to control. (see fxIsoRcvCreateContextHandle)
bufferID	ID of the receive buffer.
pStatus	Pointer to FXIsoRcvPktPerBufferBufferStatus .

Return Codes

This function returns zero or greater upon success. A negative return value indicates an error. Please use the

function [fxGetErrorMessage](#) to lookup descriptions corresponding to negative return values.

```

FX_ERR_ISO_RCV_INVALID_CONTEXT_HANDLE
FX_ERR_INTERNAL_ERROR
FX_ERR_ISO_RCV_BUFFER_NOT_FOUND
FX_ERR_ISO_RCV_CONTEXT_MISMATCH
FX_ERR_INVALID_PARAMETER

```

Synopsis

```

FXReturnCode fxIsoRcvPktPerBufferBufferStatus(
    FXIsoRcvContextHandle           contextHandle,
    uint32_t                          bufferID,
    FXIsoRcvPktPerBufferBufferStatus\* pStatus
)

```

10.4.2.2.9 fxIsoRcvDualBufferBufferStatus

Description

This function returns the status of the specified buffer created by [fxIsoRcvDualBufferAddBuffer](#).

Parameters

contextHandle	Reference handle to the context to control. (see fxIsoRcvCreateContextHandle)
bufferID	ID of the receive buffer.
pStatus	Pointer to FXIsoRcvDualBufferBufferStatus .

Return Codes

This function returns zero or greater upon success. A negative return value indicates an error. Please use the function [fxGetErrorMessage](#) to lookup descriptions corresponding to negative return values.

```

FX_ERR_ISO_RCV_INVALID_CONTEXT_HANDLE
FX_ERR_INTERNAL_ERROR
FX_ERR_ISO_RCV_BUFFER_NOT_FOUND
FX_ERR_ISO_RCV_CONTEXT_MISMATCH
FX_ERR_INVALID_PARAMETER

```

Synopsis

```

FXReturnCode fxIsoRcvDualBufferBufferStatus(
    FXIsoRcvContextHandle           contextHandle,
    uint32_t                          bufferID,
    FXIsoRcvDualBufferBufferStatus\* pStatus
)

```

10.4.3. Type Definitions

10.4.3.1. FXIsoRcvContextHandle

Description

Handle to an isochronous receive context created by [fxIsoRcvCreateContextHandle](#) function.

Synopsis

```
typedef uint32_t FxIsoRcvContextHandle;
```

10.4.3.2. FXIsoRcvCallback

Description

This function definition is used to specify a callback function that can be used as

- buffer full callback

Parameters

handle	Reference handle to the bus to control. (see fxCreateBusHandle)
userData	Pointer data specified in FXIsoRcvEventOptions .
contextHandle	Context handle that is associated with the callback occurrence.

Synopsis

```
typedef void (*FXIsoRcvCallback) {
    FXBusHandle          handle,
    void*                userData,
    FXIsoRcvContextHandle contextHandle
};
```

10.4.4. Structures**10.4.4.1. FXIsoRcvOption****Description**

This structure defines isochronous context and buffer options.

Members

optionId	ID value of the option to set.
value	Value to set for the specified option.

Synopsis

```
typedef struct {
    uint32_t    optionId;
    uint32_t    value;
} FXIsoRcvOption;
```

10.4.4.2. FXIsoRcvBuffer**Description**

This structure can be used to specify the location and size of a DMA-capable buffer.

Members

data	Pointer to a DMA-capable buffer allocated by the user. (see fxMemAlloc for allocating DMA-capable buffers)
size	Size of data in bytes.

Synopsis

```
typedef struct {
    void*    data;
    size_t   size;
} FXIsoRcvBuffer;
```

10.4.4.3. FXIsoRcvEventOptions**Description**

This structure defines data members that will be used for a receive event.

Members

callback	Specify callback function pointer or zero to clear. (see FXIsoRcvCallback)
userData	Pointer data of a user-specific data. The data will be carried to user callback functions. (see FXIsoRcvCallback)

Synopsis

```
typedef struct {
    FXIsoRcvCallback callback;
    void* userData;
} FXIsoRcvEventOptions;
```

10.4.4.4. FXIsoRcvBufferStatus**Description**

This common structure defines data fields indicating the current status of the receive buffer.

Members

statusCode	Current status of the buffer: = 1: Running - Buffer is currently being or will be written to. = 0: Stopped - Buffer is filled.
prevBufferID	Buffer ID of this buffer is linked from. 0 (zero) if this buffer is the first one of the buffer list.
nextBufferID	Buffer ID of this buffer is linked to. 0 (zero) if this buffer is the last one of the buffer list.

Synopsis

```
typedef struct {
    int32_t statusCode;
    uint32_t prevBufferID;
    uint32_t nextBufferID;
} FXIsoRcvBufferStatus;
```

10.4.4.5. FXIsoRcvBufferFillBufferStatus**Description**

This structure defines data fields indicating the current status of the receive buffer for the [Buffer-Fill mode](#).

Members

status	See FXIsoRcvBufferStatus .
buffer	One entry of FXIsoRcvBuffer . It is the same buffer that is specified with fxIsoRcvBufferFillAddBuffer .
writeOffset	Byte offset within buffer memory the stack will write the next data to. This field is updated after a packet has been received and/or after a buffer has been completely filled.

Synopsis

```
typedef struct {
    FXIsoRcvBufferStatus status;
    FXIsoRcvBuffer buffer;
    uint32_t writeOffset;
} FXIsoRcvBufferFillBufferStatus;
```

10.4.4.6. FXIsoRcvPktPerBufferBufferStatus**Description**

This structure defines data fields indicating the current status of the receive buffer for the [Packet-per-Buffer mode](#).

Members

status	See FXIsoRcvBufferStatus .
--------	--

bufferList	Arrays of FXIsoRcvBuffer . It is the same buffer array that is specified with fxIsoRcvPktPerBufferAddBuffer .
writeOffset	Byte offset within buffer memory. Indicates either zero (no packet has been written) or the size of a complete packet received.

Synopsis

```
typedef struct {
    FXIsoRcvBufferStatus    status;
    FXIsoRcvBuffer         bufferList[8];
    uint32_t                writeOffset;
} FXIsoRcvPktPerBufferBufferStatus;
```

10.4.4.7. FXIsoRcvDualBufferBufferStatus**Description**

This structure defines data fields indicating the current status of the receive buffer for the [Dual-Buffer mode](#).

Members

status	See FXIsoRcvBufferStatus .
firstBuffer	One entry of FXIsoRcvBuffer . It is the same buffer that is specified as "firstBuffer" with fxIsoRcvDualBufferAddBuffer .
secondBuffer	One entry of FXIsoRcvBuffer . It is the same buffer that is specified as "secondBuffer" with fxIsoRcvDualBufferAddBuffer .
firstWriteOffset	Byte offset within buffer memory the stack will write the next data to. This field is updated after a packet has been received and/or after a buffer has been completely filled.
secondWriteOffset	Same as firstWriteOffset, but only valid if the buffer is created by fxIsoRcvDualBufferAddBuffer ; otherwise zero.

Synopsis

```
typedef struct {
    FXIsoRcvBufferStatus    status;
    FXIsoRcvBuffer         firstBuffer;
    FXIsoRcvBuffer         secondBuffer;
    uint32_t                firstWriteOffset;
    uint32_t                secondWriteOffset;
} FXIsoRcvDualBufferBufferStatus;
```

10.4.4.8. FXIsoRcvContextStatus**Description**

This structure defines data members used for status inquiry for a receive context.

Members

statusCode	Current status of the context: = 1: Running - Context is currently actively receiving packets. = 0: Stopped - Context has not yet started, stopped by FxIsoRcvStopContext() , or has reached the end of receive buffer list.
curWriteBufferID	This field contains the buffer ID that is currently being written to. This basically means that any packet in

	progress, or the next packet if no packet is in progress, will be written to this buffer ID. This field will be updated each time the hardware has completely filled a buffer.
--	--

Synopsis

```
typedef struct {
    int32_t      statusCode;
    uint32_t     curWriteBufferID;
} FXIsoRcvContextStatus;
```

10.4.5. Constants**10.4.5.1. Buffer Modes**

Define	Value	Description
FX_ISO_RCV_CNTX_MODE_BUFF_FILL	1	All received packets are concatenated into a contiguous stream of data and fill each buffer completely. Packets may straddle multiple buffers in this mode.
FX_ISO_RCV_CNTX_MODE_PKT_PER_BUFF	2	Each received packet is stored in the buffer block added to the context by calling the add-buffer(s) function for this mode. Any leftover data are discarded, and packets never straddle into another buffer block which is added to the same context by calling another add-buffer(s) function. Each buffer block consists of 1 to 8 separate buffers
FX_ISO_RCV_CNTX_MODE_DUAL_BUFF	3	When an isochronous receive context is in dual-buffer mode, all received packets are viewed as containing a first portion of the payload followed by a second portion. The dual-buffer mode operations are similar to buffer-fill mode, but provide two separate series of buffers to stream isochronous packet data: firstBuffer series and secondBuffer series.

10.4.5.2. Context Options

Define	Value	Description
FX_ISO_RCV_OPT_CNTX_STORE_HEADER	12011	If enabled (value 1), received isochronous packets will include the complete 4-byte isochronous packet header. The end of the packet will be marked with a speed code and an event code in the first doublet, and 1 16-bit timeStamp indicating the time of the most recently received (or sent) cycle start packet. If disabled (value 0), the packet header is stripped off of received isochronous packets. (see Data Formats) 1: Store isochronous header, 0: Not store - default: 1
FX_ISO_RCV_OPT_CNTX_TAG1_SYNC_FILTER	12012	If enabled and FX_ISO_RCV_OPT_CNTX_TAG1_EN is enabled, then packets with tag value 1 will only be accepted into the corresponding

Define	Value	Description
		context if the two most-significant bits of the packet's sync field are 0 (zero). Packets with tag values other than 1 will be filtered according to the FX_ISO_RCV_OPT_CNTX_TAG0_EN0, FX_ISO_RCV_OPT_CNTX_TAG0_EN2, FX_ISO_RCV_OPT_CNTX_TAG0_EN3 settings with no additional restrictions. 1: Enable, 0: Disable - default: 0
FX_ISO_RCV_OPT_CNTX_SYNC_VALUE	12013	This value will be compared to the sync field of each isochronous packet for the corresponding context if a receiving buffer has wait_for_sync option (see Buffer Options) enabled. Valid range: 0 to 15 - default: 0
FX_ISO_RCV_OPT_CNTX_CHANNEL_NUM	12014	Isochronous channel number for which the corresponding context will accept packets. Valid range: 0 to 63 - default: 0
FX_ISO_RCV_OPT_CNTX_MULTI_CHANNEL_EN_HI	12015	Enable the multi-channel receive on the corresponding context if this value or the value of FX_ISO_RCV_OPT_CNTX_MULTI_CHANNEL_EN_LO is non-zero. Valid range: 0 to 0xFFFFFFFF (channel 63-32). The value of 0x10000000 indicates enabling iso channel 63. Default: 0
FX_ISO_RCV_OPT_CNTX_MULTI_CHANNEL_EN_LO	12016	Enable the multi-channel receive on the corresponding context if this value or the value of FX_ISO_RCV_OPT_CNTX_MULTI_CHANNEL_EN_HI is non-zero. Valid range: 0 to 0xFFFFFFFF (channel 31-0). The value of 0x10000000 indicates enabling iso channel 31. Default: 0
FX_ISO_RCV_OPT_CNTX_CYCLE_MATCH	12017	Any 15-bit values (0 to 0x7FFF) enable the cycle match feature that keeps the corresponding context from running until the value matches to the low order two bits of cycleSeconds and the 13-bit cycleCount field in the cycle start packet. Disabled if value is greater than 0x7FFF - default: 0x8000 (disabled)
FX_ISO_RCV_OPT_CNTX_TAG0_EN	12018	If enabled, the corresponding context will match on isochronous receive packet with a tag field of 0. 1: Enable, 0: Disable - default: 1
FX_ISO_RCV_OPT_CNTX_TAG1_EN	12019	If enabled, the corresponding context will match on isochronous receive packet with a tag field of 1. 1: Enable, 0: Disable - default: 1
FX_ISO_RCV_OPT_CNTX_TAG2_EN	12020	If enabled, the corresponding context will match on isochronous receive packet with a tag field of 2. 1: Enable, 0: Disable - default: 1
FX_ISO_RCV_OPT_CNTX_TAG3_EN	12021	If enabled, the corresponding context will

Define	Value	Description
		match on isochronous receive packet with a tag field of 3. 1: Enable, 0: Disable - default: 1

10.4.5.3. Buffer Options

Define	Value	Description
FX_ISO_RCV_OPT_BUFF_WAIT_SYNC	12111	1: Wait for sync value to match 0: not - default: 0 (see Context Options)
FX_ISO_RCV_OPT_BUFF_ENABLE_CALLBACK	12112	1: Callback will be called upon completion of buffer 0: not - default: 0

10.4.5.4. Error Codes

Define	Description
FX_ERR_ISO_RCV_INVALID_CONTEXT_HANDLE	Specified context handle is invalid
FX_ERR_ISO_RCV_BUFFER_NOT_FOUND	Specified buffer ID does not exist
FX_ERR_ISO_RCV_FEATURE_NOT_SUPPORTED	Specified feature is not supported
FX_ERR_ISO_RCV_BUFFER_TOO_LARGE	Specified receive buffer is too large
FX_ERR_ISO_RCV_NO_AVAILABLE_CONTEXT	All contexts have been used
FX_ERR_ISO_RCV_BUFFER_TYPE_MISMATCH	Specified context has a wrong buffer mode
FX_ERR_ISO_RCV_CONTEXT_MISMATCH	Data buffer must belong to the same context
FX_ERR_ISO_RCV_NO_AVAILABLE_BUFFER	All resource for buffer control data have been used
FX_ERR_ISO_RCV_BUFFER_NOT_LAST_OF_LIST	Specified buffer is not the last one
FX_ERR_ISO_RCV_BUFFER_NOT_FIRST_OF_LIST	Specified buffer is not the first one
FX_ERR_ISO_RCV_MULTICHANNEL_BUFFER_MISMATCH	Multi-channel option must be enabled in Buffer-Fill mode and with Store Isoch-header option
FX_ERR_ISO_RCV_MULTICHANNEL_NOT_AVAILABLE	Multi-channel feature is currently used by the other context

10.4.6. Data Formats

There are four formats for isochronous receive packets depending on the settings of the store-header context option (see [Context Options](#)) and the buffer mode (see [Buffer Modes](#)). If the store-header option is disabled, then only the isochronous data without any padding, header quadlet or timestamp quadlet is put in the buffer.

Isochronous receive fields

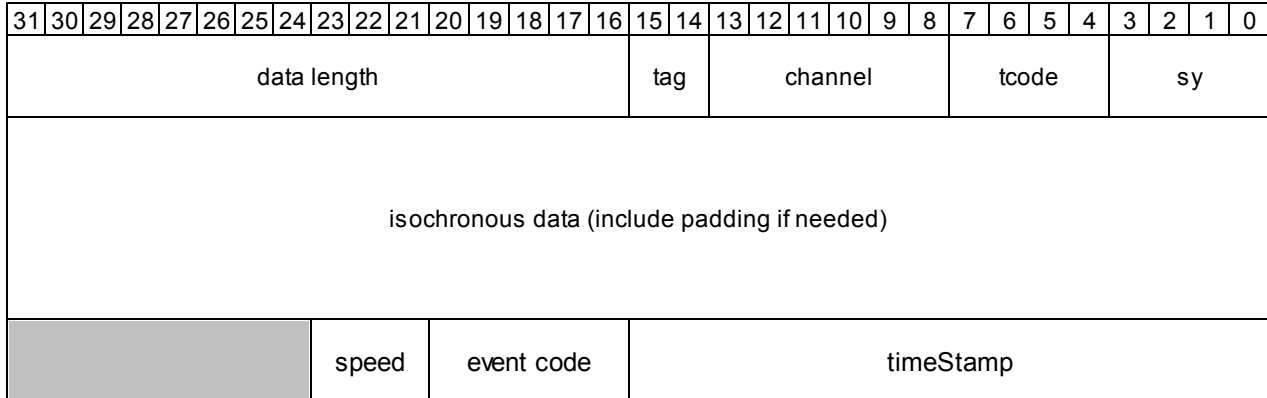
field	bits	description
data length	16	Number of bytes of isochronous data in this packet.
tag	2	The data format of this isochronous data.
channel	6	The channel number this packet is associated with.
tcode	4	The transaction code (should always be 0xA)
sy	4	Synchronization control field.
isochronous data		The data received with this buffer. The last quadlet will be padded with zeroes, if necessary.
padding		If the data length mode 4 is not zero, then zero-value bytes have been added onto the end of the packet to guarantee that a whole number of quadlets was sent. In three formats, the pad bytes are stripped off the packet.

field	bits	description
speed	3	Speed Code: 0 = 100Mb/s 1 = 200Mb/s 2 = 400Mb/s 3 = 800Mb/s
event code	5	0x02 = Long packet. The received data length was greater than the buffer's data length. 0x11 = Complete. 0x1D = In the Packet-per-Buffer mode, this event code indicates that a data field CRC or data length error was detected.
timeStamp	16	The time at which this packet was received, specified by the three low order bits of cycleSeconds, and the 13-bits of cycleCount from the most recently received (or sent) cycle start packet.

10.4.6.1. Buffer-Fill mode Data Formats

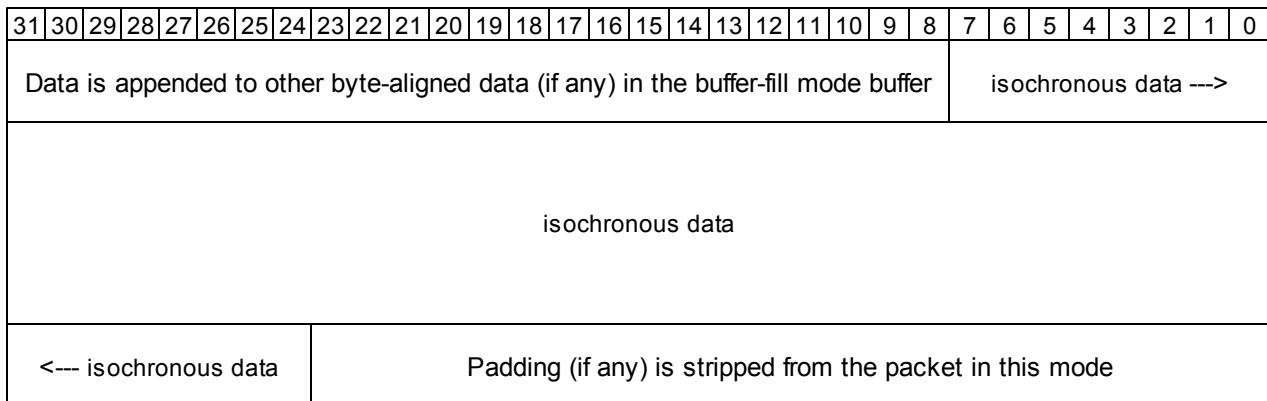
10.4.6.1.1 With Header/Trailer

The format of an isochronous receive packet with a context in the [Buffer-Fill mode](#) and the store-header option (see [Context Options](#)) **enabled**.



10.4.6.1.2 Without Header/Trailer

The format of an isochronous receive packet with a context in the [Buffer-Fill mode](#) and the store-header option (see [Context Options](#)) **disabled**.



10.4.6.2. Packet-per-buffer mode and dual-buffer mode Data Formats

10.4.6.2.1 With Header/Trailer

The format of an isochronous receive packet with a context in the [Packet-per-Buffer mode](#) or the [Dual-Buffer mode](#); and the store-header option (see [Context Options](#)) **enabled**.

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
																timeStamp															
data length																tag	channel				tcode				sy						
If headers & data are in the same buffer, then the data will be quadlet aligned. If headers are in a separate buffer from the data, then the data buffer may be byte aligned.																								isochronous data --->							
isochronous data																															
<--- isochronous data												Padding (if any) is stripped from the packet in this mode.																			

10.4.6.2.2 Without Header/Trailer

The format of an isochronous receive packet with a context in the [Packet-per-Buffer mode](#) or the [Dual-Buffer mode](#); and the store-header option (see [Context Options](#)) **disabled**.

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
Buffers with data only (no headers), like this, may be byte aligned																								isochronous data --->							
isochronous data																															
<--- isochronous data												Padding (if any) is stripped from the packet in this mode																			

10.5. Low-Level 1394

10.5.1. Settings

10.5.1.1. Resource Usage

The following settings can be used as settingId in an [FXSetting](#) instance passed to [fxCreateBusHandle\(\)](#) to control resource usage by the Low-Level module for the bus that is being opened.

`FX_SETTING_ID_ASYNC_MAX_TRM_QUEUE_LENGTH`

This setting determines the maximum number of Asynchronous packets that may reside in the transmit queue at any given point in time. As soon as a packet needs to be transmitted while the queue is full an error would be returned.

Minimum:4, Maximum 10000, Default: 5120

`FX_SETTING_ID_ASYNC_NUM_RCV_BUFFERS`

This setting determines how many buffers of the maximum buffer size supported by the Link Layer will be used to form the complete Asynchronous Reception buffer list. As long as buffer space is available, no Asynchronous packet will be missed by the hardware. As soon as software can not keep up with processing the received packets, the buffers will start filling up.

Minimum: 4, Maximum: 100, Default: 30

10.5.2. Functions

10.5.2.1. Asynchronous Packet Reception Functions

10.5.2.1.1 fxAsyRcvWaitSingleRequest

Description

This function can be used to receive a request packet which is sent to the local node.

Note that this is a blocking function, only one `fsAsyRcvWaitSingleRequest` or `fxAsyRcvWaitSingleResponse` may be active per bus at any time.

Parameters

handle	Reference handle to the bus to control. (see fxCreateBusHandle)
data	Pointer to data receive buffer which must be at least large enough to the 'size' parameter below.
size	Pointer to size data of the receive buffer in bytes. This function will update its value with the amount of written bytes.
speed	Pointer to data to which the speed of the received packet is set.
ackCode	Pointer to data to which the acknowledge code of the received packet is set.
timeout	This function returns with the timeout error if no packet is received within the specified amount of time (milliseconds).

Return Codes

This function returns zero or greater upon success. A negative return value indicates an error. Please use the function [fxGetErrorMessage](#) to lookup descriptions corresponding to negative return values.

`FX_ERR_INVALID_HANDLE`
`FX_ERR_LOW_LVL_RECEIVE_TIMEOUT`
`FX_ERR_INVALID_PARAMETER`

Synopsis

```

FXReturnCode fxAsyRcvWaitSingleRequest(
    FXBusHandle      handle,
    uint8_t*         data,
    uint32_t*        size,
    uint32_t*        speed,
    uint32_t*        ackCode,
    uint32_t         timeout
)

```

10.5.2.1.2 fxAsyRcvWaitSingleResponse

Description

This function can be used to receive a response packet which is sent to the local node.

Note that this is a blocking function, only one fsAsyRcvWaitSingleResponse or fxAsyRcvWaitSingleRequest may be active per bus at any time.

Parameters

handle	Reference handle to the bus to control. (see fxCreateBusHandle)
data	Pointer to data receive buffer which must be at least large enough to the 'size' parameter below.
size	Pointer to size data of the receive buffer in bytes. This function will update its value with the amount of written bytes.
speed	Pointer to data to which the speed of the received packet is set.
ackCode	Pointer to data to which the acknowledge code of the received packet is set.
timeout	This function returns with the timeout error if no packet is received within the specified amount of time (milliseconds).

Return Codes

This function returns zero or greater upon success. A negative return value indicates an error. Please use the function [fxGetErrorMessage](#) to lookup descriptions corresponding to negative return values.

FX_ERR_INVALID_HANDLE
FX_ERR_LOW_LVL_RECEIVE_TIMEOUT
FX_ERR_INVALID_PARAMETER

Synopsis

```

FXReturnCode fxAsyRcvWaitSingleResponse(
    FXBusHandle      handle,
    uint8_t*         data,
    uint32_t*        size,
    uint32_t*        speed,
    uint32_t*        ackCode,
    uint32_t         timeout
)

```

10.5.2.1.3 fxAsyRcvSetPacketCallback

Description

This function can be used to register a callback that will be called for each Asynchronous packet that is received by the Link Layer controller.

Parameters

handle	Reference handle to the bus to control. (see fxCreateBusHandle)
callback	Function pointer of the user-defined function that needs to be called when an Asynchronous Packet is received. (see FXAsyRcvPacketCallback)
userData	Pointer to a user-specified data. The pointer will be carried to the user callback function specified above.

Return Codes

This function returns zero or greater upon success. A negative return value indicates an error. Please use the function [fxGetErrorMessage](#) to lookup descriptions corresponding to negative return values.

FX_ERR_INVALID_HANDLE
FX_ERR_LOW_LVL_RECEIVE_TIMEOUT
FX_ERR_INVALID_PARAMETER

Synopsis

```
FXReturnCode fxAsyRcvSetPacketCallback(
    FXBusHandle      handle,
    FXAsyRcvPacketCallback  callback,
    void*            userData
);
```

10.5.2.2. Single Packet Transmission Functions

10.5.2.2.1 fxAsyTrmWriteQuadletRequest

Description

This function can be used to transmit a single Write Quadlet Request packet.

Parameters

handle	Reference handle to the bus to control. (see fxCreateBusHandle)
speed	Transmission speed. See Speed Codes for a valid value.
nodeID	Destination node ID to which a new packet is sent.
transactionLabel	Transaction Label field in the header.
retryCode	Retry Code field in the header.
offset	Pointer to FXAddress64 structure.
data	Quadlet data to be sent.

Return Codes

This function returns zero or greater upon success. A negative return value indicates an error. Please use the function [fxGetErrorMessage](#) to lookup descriptions corresponding to negative return values.

FX_ERR_INVALID_HANDLE
FX_ERR_INVALID_ADDRESS
FX_ERR_INVALID_LOCAL_NODE_ID
FX_ERR_INVALID_PARAMETER
FX_ERR_BUS_RESET_DETECT
FX_ERR_INTERNAL_ERROR

Synopsis

```
FXReturnCode fxAsyTrmWriteQuadletRequest(
```

```

    FXBusHandle        handle,
    uint32_t           speed,
    uint32_t           nodeID,
    uint32_t           transactionLabel,
    uint32_t           retryCode,
    const FXAddress64* offset,
    uint32_t           data
)

```

10.5.2.2.2 fxAsyTrmWriteBlockRequest

Description

This function can be used to transmit a single Write Block Request packet.

Parameters

handle	Reference handle to the bus to control. (see fxCreateBusHandle)
speed	Transmission speed. See Speed Codes for a valid value.
nodeID	Destination node ID to which a new packet is sent.
transactionLabel	Transaction Label field in the header.
retryCode	Retry Code field in the header.
offset	Pointer to FXAddress64 structure.
data	Pointer to a DMA-capable buffer of data to be sent. (See fxMemAlloc for allocating DMA-capable buffers)
data_byte_size	Data size in bytes of 'data' parameter above.

Return Codes

This function returns zero or greater upon success. A negative return value indicates an error. Please use the function [fxGetErrorMessage](#) to lookup descriptions corresponding to negative return values.

```

FX_ERR_INVALID_HANDLE
FX_ERR_INVALID_ADDRESS
FX_ERR_INVALID_LOCAL_NODE_ID
FX_ERR_INVALID_PARAMETER
FX_ERR_BUS_RESET_DETECT
FX_ERR_INTERNAL_ERROR

```

Synopsis

```

FXReturnCode fxAsyTrmWriteBlockRequest(
    FXBusHandle        handle,
    uint32_t           speed,
    uint32_t           nodeID,
    uint32_t           transactionLabel,
    uint32_t           retryCode,
    const FXAddress64* offset,
    uint8_t*           data,
    uint32_t           data_byte_size
)

```

10.5.2.2.3 fxAsyTrmWriteResponse

Description

This function can be used to transmit a single Write Response packet.

Parameters

handle	Reference handle to the bus to control. (see fxCreateBusHandle)
speed	Transmission speed. See Speed Codes for a valid value.
nodeID	Destination node ID to which a new packet is sent.
transactionLabel	Transaction Label field in the header.
retryCode	Retry Code field in the header.
responseCode	Response Code field in the header.

Return Codes

This function returns zero or greater upon success. A negative return value indicates an error. Please use the function [fxGetErrorMessage](#) to lookup descriptions corresponding to negative return values.

FX_ERR_INVALID_HANDLE
FX_ERR_INVALID_LOCAL_NODE_ID
FX_ERR_INVALID_PARAMETER
FX_ERR_BUS_RESET_DETECT
FX_ERR_INTERNAL_ERROR

Synopsis

```
FXReturnCode fxAsyTrmWriteResponse(
    FXBusHandle      handle,
    uint32_t         speed,
    uint32_t         nodeID,
    uint32_t         transactionLabel,
    uint32_t         retryCode,
    uint32_t         responseCode
)
```

10.5.2.2.4 fxAsyTrmReadQuadletRequest

Description

This function can be used to transmit a single Read Quadlet Request packet.

Parameters

handle	Reference handle to the bus to control. (see fxCreateBusHandle)
speed	Transmission speed. See Speed Codes for a valid value.
nodeID	Destination node ID to which a new packet is sent.
transactionLabel	Transaction Label field in the header.
retryCode	Retry Code field in the header.
offset	Pointer to FXAddress64 structure.

Return Codes

This function returns zero or greater upon success. A negative return value indicates an error. Please use the function [fxGetErrorMessage](#) to lookup descriptions corresponding to negative return values.

FX_ERR_INVALID_HANDLE
FX_ERR_INVALID_ADDRESS
FX_ERR_INVALID_LOCAL_NODE_ID
FX_ERR_INVALID_PARAMETER
FX_ERR_BUS_RESET_DETECT
FX_ERR_INTERNAL_ERROR

Synopsis

```
FXReturnCode fxAsyTrmReadQuadletRequest(
    FXBusHandle          handle,
    uint32_t             speed,
    uint32_t             nodeID,
    uint32_t             transactionLabel,
    uint32_t             retryCode,
    const FXAddress64*   offset
)
```

10.5.2.2.5 fxAsyTrmReadBlockRequest

Description

This function can be used to transmit a single Read Block Request packet.

Parameters

handle	Reference handle to the bus to control. (see fxCreateBusHandle)
speed	Transmission speed. See Speed Codes for a valid value.
nodeID	Destination node ID to which a new packet is sent.
transactionLabel	Transaction Label field in the header.
retryCode	Retry Code field in the header.
offset	Pointer to FXAddress64 structure.
req_byte_size	Data Length field in the header.

Return Codes

This function returns zero or greater upon success. A negative return value indicates an error. Please use the function [fxGetErrorMessage](#) to lookup descriptions corresponding to negative return values.

```
FX_ERR_INVALID_HANDLE
FX_ERR_INVALID_ADDRESS
FX_ERR_INVALID_LOCAL_NODE_ID
FX_ERR_INVALID_PARAMETER
FX_ERR_BUS_RESET_DETECT
FX_ERR_INTERNAL_ERROR
```

Synopsis

```
FXReturnCode fxAsyTrmReadBlockRequest(
    FXBusHandle          handle,
    uint32_t             speed,
    uint32_t             nodeID,
    uint32_t             transactionLabel,
    uint32_t             retryCode,
    const FXAddress64*   offset,
    uint32_t             req_byte_size
)
```

10.5.2.2.6 fxAsyTrmReadQuadletResponse

Description

This function can be used to transmit a single Read Quadlet Response packet.

Parameters

handle	Reference handle to the bus to control.
--------	---

	(see fxCreateBusHandle)
speed	Transmission speed. See Speed Codes for a valid value.
nodeID	Destination node ID to which a new packet is sent.
transactionLabel	Transaction Label field in the header.
retryCode	Retry Code field in the header.
responseCode	Response Code field in the header.
data	Quadlet data to be sent.

Return Codes

This function returns zero or greater upon success. A negative return value indicates an error. Please use the function [fxGetErrorMessage](#) to lookup descriptions corresponding to negative return values.

FX_ERR_INVALID_HANDLE
FX_ERR_INVALID_LOCAL_NODE_ID
FX_ERR_INVALID_PARAMETER
FX_ERR_BUS_RESET_DETECT
FX_ERR_INTERNAL_ERROR

Synopsis

```
FXReturnCode fxAsyTrmReadQuadletResponse(
    FXBusHandle      handle,
    uint32_t         speed,
    uint32_t         nodeID,
    uint32_t         transactionLabel,
    uint32_t         retryCode,
    uint32_t         responseCode,
    uint32_t         data
)
```

10.5.2.2.7 fxAsyTrmReadBlockResponse

Description

This function can be used to transmit a single Read Block Response packet.

Parameters

handle	Reference handle to the bus to control. (see fxCreateBusHandle)
speed	Transmission speed. See Speed Codes for a valid value.
nodeID	Destination node ID to which a new packet is sent.
transactionLabel	Transaction Label field in the header.
retryCode	Retry Code field in the header.
responseCode	Response Code field in the header.
data	Pointer to a DMA-capable buffer of data to be sent. (See fxMemAlloc for allocating DMA-capable buffers)
data_byte_size	Data size in bytes of 'data' parameter above.

Return Codes

This function returns zero or greater upon success. A negative return value indicates an error. Please use the function [fxGetErrorMessage](#) to lookup descriptions corresponding to negative return values.

FX_ERR_INVALID_HANDLE

FX_ERR_INVALID_LOCAL_NODE_ID
FX_ERR_INVALID_PARAMETER
FX_ERR_BUS_RESET_DETECT
FX_ERR_INTERNAL_ERROR

Synopsis

```
FXReturnCode fxAsyTrmReadBlockResponse(
    FXBusHandle      handle,
    uint32_t         speed,
    uint32_t         nodeID,
    uint32_t         transactionLabel,
    uint32_t         retryCode,
    uint32_t         responseCode,
    uint8_t*         data,
    uint32_t         data_byte_size
)
```

10.5.2.2.8 fxAsyTrmLockRequest

Description

This function can be used to transmit a single Lock Request packet.

Parameters

handle	Reference handle to the bus to control. (see fxCreateBusHandle)
speed	Transmission speed. See Speed Codes for a valid value.
nodeID	Destination node ID to which a new packet is sent.
transactionLabel	Transaction Label field in the header.
retryCode	Retry Code field in the header.
offset	Pointer to FXAddress64 structure.
extendedTcode	Extended tCode field in the header.
data	Pointer to a DMA-capable buffer of data to be sent. (See fxMemAlloc for allocating DMA-capable buffers)
data_byte_size	Data size in bytes of 'data' parameter above.

Return Codes

This function returns zero or greater upon success. A negative return value indicates an error. Please use the function [fxGetErrorMessage](#) to lookup descriptions corresponding to negative return values.

FX_ERR_INVALID_HANDLE
FX_ERR_INVALID_ADDRESS
FX_ERR_INVALID_LOCAL_NODE_ID
FX_ERR_INVALID_PARAMETER
FX_ERR_BUS_RESET_DETECT
FX_ERR_INTERNAL_ERROR

Synopsis

```
FXReturnCode fxAsyTrmLockRequest(
    FXBusHandle      handle,
    uint32_t         speed,
    uint32_t         nodeID,
    uint32_t         transactionLabel,
    uint32_t         retryCode,
    const FXAddress64* offset,
```

```

        uint32_t          extendedTcode,
        uint8_t*         data,
        uint32_t         data_byte_size
    )

```

10.5.2.2.9 fxAsyTrmStream

Description

This function can be used to transmit a single Asynchronous Stream packet.

Parameters

handle	Reference handle to the bus to control. (see fxCreateBusHandle)
speed	Transmission speed. See Speed Codes for a valid value.
tag	Tag field in the header.
channelNum	Channel field in the header.
sy	Sy field in the header.
data	Pointer to a DMA-capable buffer of data to be sent. (See fxMemAlloc for allocating DMA-capable buffers)
data_byte_size	Data size in bytes of 'data' parameter above.

Return Codes

This function returns zero or greater upon success. A negative return value indicates an error. Please use the function [fxGetErrorMessage](#) to lookup descriptions corresponding to negative return values.

```

FX_ERR_INVALID_HANDLE
FX_ERR_INVALID_LOCAL_NODE_ID
FX_ERR_INVALID_PARAMETER
FX_ERR_BUS_RESET_DETECT
FX_ERR_INTERNAL_ERROR

```

Synopsis

```

FXReturnCode fxAsyTrmStream(
    FXBusHandle    handle,
    uint32_t       speed,
    uint32_t       tag,
    uint32_t       channelNum,
    uint32_t       sy,
    uint8_t*       data,
    uint32_t       data_byte_size
)

```

10.5.2.2.10 fxAsyTrmLockResponse

Description

This function can be used to transmit a single Lock Response packet.

Parameters

handle	Reference handle to the bus to control. (see fxCreateBusHandle)
speed	Transmission speed. See Speed Codes for a valid value.
nodeID	Destination node ID to which a new packet is sent.
transactionLabel	Transaction Label field in the header.

retryCode	Retry Code field in the header.
responseCode	Response Code field in the header.
extendedTcode	Extended tCode field in the header.
data	Pointer to a DMA-capable buffer of data to be sent. (See fxMemAlloc for allocating DMA-capable buffers)
data_byte_size	Data size in bytes of 'data' parameter above.

Return Codes

This function returns zero or greater upon success. A negative return value indicates an error. Please use the function [fxGetErrorMessage](#) to lookup descriptions corresponding to negative return values.

FX_ERR_INVALID_HANDLE
FX_ERR_INVALID_LOCAL_NODE_ID
FX_ERR_INVALID_PARAMETER
FX_ERR_BUS_RESET_DETECT
FX_ERR_INTERNAL_ERROR

Synopsis

```
FXReturnCode fxAsyTrmLockResponse(
    FXBusHandle    handle,
    uint32_t       speed,
    uint32_t       nodeID,
    uint32_t       transactionLabel,
    uint32_t       retryCode,
    uint32_t       responseCode,
    uint32_t       extendedTcode,
    uint8_t*       data,
    uint32_t       data_byte_size
)
```

10.5.2.3. PHY Packets and registers

10.5.2.3.1 fxReadLocalPhyBaseReg

Description

This function can be used to read a local PHY register value.

Parameters

handle	Reference handle to the bus to control. (see fxCreateBusHandle)
offset	Register offset.
value	Pointer to a buffer to which a register value is returned.

Return Codes

This function returns zero or greater upon success. A negative return value indicates an error. Please use the function [fxGetErrorMessage](#) to lookup descriptions corresponding to negative return values.

FX_ERR_INVALID_HANDLE
FX_ERR_INVALID_PARAMETER
FX_ERR_LOW_LVL_PHY_REG_ACCESS_TIMEOUT
FX_ERR_INTERNAL_ERROR

Synopsis

```
FXReturnCode fxReadLocalPhyRegister(
    FXBusHandle    handle,
    uint8_t        offset,
```

```

        uint8_t*      value
    )

```

10.5.2.3.2 fxReadLocalPhyPageReg

Description

This function can be used to read local phy register value.

Parameters

handle	Reference handle to the bus to control. (see fxCreateBusHandle)
port	PHY port to select.
page	Page number to select.
offset	Register offset within selected page. Please note that the first byte in the page registers has offset 8.
value	Pointer to a buffer to which a register value is returned.

Return Codes

This function returns zero or greater upon success. A negative return value indicates an error. Please use the function [fxGetErrorMessage](#) to lookup descriptions corresponding to negative return values.

```

FX_ERR_INVALID_HANDLE
FX_ERR_INVALID_PARAMETER
FX_ERR_LOW_LVL_PHY_REG_ACCESS_TIMEOUT
FX_ERR_INTERNAL_ERROR

```

Synopsis

```

FXReturnCode fxReadLocalPhyPageReg(
    FXBusHandle      handle,
    uint8_t          port,
    uint8_t          page,
    uint8_t          offset,
    uint8_t*         value
)

```

10.5.2.3.3 fxWriteLocalPhyBaseReg

Description

This function can be used to write data to local PHY register.

Parameters

handle	Reference handle to the bus to control. (see fxCreateBusHandle)
offset	Register offset.
value	Data to be written to the register.

Return Codes

This function returns zero or greater upon success. A negative return value indicates an error. Please use the function [fxGetErrorMessage](#) to lookup descriptions corresponding to negative return values.

```

FX_ERR_INVALID_HANDLE
FX_ERR_INVALID_PARAMETER
FX_ERR_LOW_LVL_PHY_REG_ACCESS_TIMEOUT
FX_ERR_INTERNAL_ERROR

```

Synopsis

```

FXReturnCode fxWriteLocalPhyRegister(
    FXBusHandle    handle,
    uint8_t        offset,
    uint8_t        value
)

```

10.5.2.3.4 fxWriteLocalPhyPageReg

Description

This function can be used to write data to local phy register.

Parameters

handle	Reference handle to the bus to control. (see fxCreateBusHandle)
port	PHY port to select.
page	Page to select.
offset	Register offset within selected page. Please note that the first byte in the page registers has offset 8.
value	Data to be written to the register.

Return Codes

This function returns zero or greater upon success. A negative return value indicates an error. Please use the function [fxGetErrorMessage](#) to lookup descriptions corresponding to negative return values.

```

FX_ERR_INVALID_HANDLE
FX_ERR_INVALID_PARAMETER
FX_ERR_LOW_LVL_PHY_REG_ACCESS_TIMEOUT
FX_ERR_INTERNAL_ERROR

```

Synopsis

```

FXReturnCode fxWriteLocalPhyRegister(
    FXBusHandle    handle,
    uint8_t        port,
    uint8_t        page,
    uint8_t        offset,
    uint8_t        value
)

```

10.5.2.3.5 fxReadRemotePhyPageReg

Description

This function can be used to read a page register from remote node.

Parameters

busHandle	Reference handle to the bus to control. (see fxCreateBusHandle)
generation	Set this value to the current bus generation number. (see fxGetBusGeneration)
nodeId	Node ID of destination device.
port	Specify port number to read page register of.
page	Specify page number to read.
reg	Specify register index to read. The register offset will be calculated as $0x8 + \text{reg}$.
val	Specify a buffer to retrieve the value in.

Return Codes

This function returns zero or greater upon success. A negative return value indicates an error. Please use the function [fxGetErrorMessage](#) to lookup descriptions corresponding to negative return values.

FX_ERR_NORMAL
FX_ERR_INVALID_HANDLE
FX_ERR_LOW_LVL_RECEIVE_TIMEOUT

Synopsis

```
FXReturnCode fxReadRemotePhyPageReg(
    FXBusHandle      busHandle,
    uint32_t         generation,
    uint32_t         nodeId,
    uint8_t          port,
    uint8_t          page,
    uint16_t         reg,
    uint8_t*         val
)
```

10.5.2.3.6 fxReadRemotePhyBaseReg

Description

This function can be used to read a base register from remote node.

Parameters

busHandle	Reference handle to the bus to control. (see fxCreateBusHandle)
generation	Set this value to the current bus generation number. (see fxGetBusGeneration)
nodeId	Node ID of destination device.
reg	Specify register index to read.
val	Specify a buffer to retrieve the value in.

Return Codes

This function returns zero or greater upon success. A negative return value indicates an error. Please use the function [fxGetErrorMessage](#) to lookup descriptions corresponding to negative return values.

FX_ERR_NORMAL
FX_ERR_INVALID_HANDLE
FX_ERR_LOW_LVL_RECEIVE_TIMEOUT

Synopsis

```
FXReturnCode fxReadRemotePhyBaseReg(
    FXBusHandle      busHandle,
    uint32_t         generation,
    uint32_t         nodeId,
    uint16_t         reg,
    uint8_t*         val
)
```

10.5.2.3.7 fxPhyRemoteCommand

Description

This function can be used to perform a PHY Remote Command on a remote node.

Parameters

busHandle	Reference handle to the bus to control. (see fxCreateBusHandle)
generation	Set this value to the current bus generation number. (see fxGetBusGeneration)
nodeId	Node ID of destination device.
port	Specify the port number to select.
command	Specify the command to perform (see PHY Remote Commands)
confirmationFlags	Specify one or more PHY Confirmation Flags

Return Codes

This function returns zero or greater upon success. A negative return value indicates an error. Please use the function [fxGetErrorMessage](#) to lookup descriptions corresponding to negative return values.

FX_ERR_NORMAL

...

Synopsis

```
FXReturnCode fxPhyRemoteCommand(
    FXBusHandle      busHandle,
    uint32_t         generation,
    uint32_t         nodeId,
    uint8_t          port,
    uint8_t          command,
    uint8_t*         confirmationFlags
);
```

10.5.2.3.8 fxPhySetForceRoot

Description

This function can be used to transmit a PHY Configuration packet to set the specified nodeId as root. This function will not issue a bus reset.

Parameters

busHandle	Reference handle to the bus to control. (see fxCreateBusHandle)
generation	Set this value to the current bus generation number. (see fxGetBusGeneration)
nodeId	Node ID to set as root.

Return Codes

This function returns zero or greater upon success. A negative return value indicates an error. Please use the function [fxGetErrorMessage](#) to lookup descriptions corresponding to negative return values.

FX_ERR_NORMAL
FX_ERR_INVALID_HANDLE
FX_ERR_INVALID_PARAMETER
FX_ERR_LOW_LVL_RECEIVE_TIMEOUT
FX_ERR_BUS_RESET_DETECT
FX_ERR_LOW_LVL_PHY_MAX_OUTSTANDING

Synopsis

```
FXReturnCode fxPhySetForceRoot(
    FXBusHandle      busHandle,
    uint32_t         generation,
```

```

        uint32_t      nodeId
    )

```

10.5.2.3.9 fxPhySetGapCount

Description

This function can be used to transmit a PHY Configuration packet to set the gap count to the specified value. This function will not issue a bus reset.

Parameters

busHandle	Reference handle to the bus to control. (see fxCreateBusHandle)
generation	Set this value to the current bus generation number. (see fxGetBusGeneration)
gapCount	Gap count value to set as part of the PHY configuration packet.

Return Codes

This function returns zero or greater upon success. A negative return value indicates an error. Please use the function [fxGetErrorMessage](#) to lookup descriptions corresponding to negative return values.

```

FX_ERR_NORMAL
FX_ERR_INVALID_HANDLE
FX_ERR_INVALID_PARAMETER
FX_ERR_LOW_LVL_RECEIVE_TIMEOUT
FX_ERR_BUS_RESET_DETECT
FX_ERR_LOW_LVL_PHY_MAX_OUTSTANDING

```

Synopsis

```

FXReturnCode fxPhySetForceRoot(
    FXBusHandle      busHandle,
    uint32_t          generation,
    uint32_t          gapCount
)

```

10.5.2.3.10 fxPingRemoteNode

Description

This function can be used to perform a PHY Ping operation to a remote node.

Parameters

busHandle	Reference handle to the bus to control. (see fxCreateBusHandle)
generation	Set this value to the current bus generation number. (see fxGetBusGeneration)
nodeId	Node ID of destination device.
time	User allocated buffer that will be filled with the ping time measured according to 1394 spec.

Return Codes

This function returns zero or greater upon success. A negative return value indicates an error. Please use the function [fxGetErrorMessage](#) to lookup descriptions corresponding to negative return values.

```

FX_ERR_NORMAL
FX_ERR_INVALID_HANDLE
FX_ERR_INVALID_PARAMETER
FX_ERR_LOW_LVL_RECEIVE_TIMEOUT
FX_ERR_BUS_RESET_DETECT

```


FX_ERR_LOW_LVL_PHY_MAX_OUTSTANDING FX_ERR_NOT_IMPLEMENTED

Synopsis

```
FXReturnCode fxPingRemoteNode(
    FXBusHandle      busHandle,
    uint32_t         generation,
    uint32_t         nodeId,
    uint32_t*        time
)
```

10.5.2.3.11 fxPhyPacketSetRcvCallback

Description

This function can be used to register a user-defined callback function that is called for each PHY packet received. Please note that Self ID packets received during the Self Identification phase will not be received by this function.

Parameters

busHandle	Reference handle to the bus to control. (see fxCreateBusHandle)
callback	Function pointer of the user-defined function that will be called for each PHY packet received. (see FXPhyPacketRcvCallback)
userData	User provided data that will be passed to the user callback each time it is called.

Return Codes

This function returns zero or greater upon success. A negative return value indicates an error. Please use the function [fxGetErrorMessage](#) to lookup descriptions corresponding to negative return values.

FX_ERR_NORMAL

...

Synopsis

```
FXReturnCode fxPhyPacketSetRcvCallback(
    FXBusHandle      busHandle,
    FXPhyPacketRcvCallback callback,
    void*            userData
);
```

10.5.2.3.12 fxPhyPacketTmRaw

Description

This function can be used to transmit a PHY packet in RAW form.

Parameters

busHandle	Reference handle to the bus to control. (see fxCreateBusHandle)
phyPacket[2]	Contents of the PHY packet to transmit.
speedCode	Speed Code to use for the PHY packet transmission.

Return Codes

This function returns zero or greater upon success. A negative return value indicates an error. Please use the function [fxGetErrorMessage](#) to lookup descriptions corresponding to negative return values.

FX_ERR_NORMAL

...

Synopsis

```

FXReturnCode fxPhyPacketTrmRaw(
    FXBusHandle    busHandle,
    uint32_t       phyPacket[2],
    uint32_t       speedCode
);

```

10.5.2.4. Topology Functions**10.5.2.4.1 fxGetSelfIdData****Description**

This function can be used to obtain the raw selfID data for all nodes on the bus. Data returned contains all the Self ID packets as received on the bus back to back. The theoretical maximum size equals 63 devices times 4 packets times 2 quadlets.

Parameters

handle	Reference handle to the bus to control. (see fxCreateBusHandle)
generation	Set this value to the current bus generation number. (see fxGetBusGeneration)
maxSize	Maximum number of quadlets the API should write to buffer
size	Return the actual number of quadlets written to buffer.
buffer	User allocated buffer that can hold at least maxSize quadlets.

Return Codes

This function returns zero or greater upon success. A negative return value indicates an error. Please use the function [fxGetErrorMessage](#) to lookup descriptions corresponding to negative return values.

```

FX_ERR_NORMAL
FX_ERR_INVALID_HANDLE
FX_ERR_LOW_LVL_RECEIVE_TIMEOUT
FX_ERR_INVALID_PARAMETER
FX_ERR_LICENSE_MODULE
FX_ERR_LICENSE_EXPIRED
FX_ERR_FIRESTACK_DEMO_TIMEOUT

```

Synopsis

```

FXReturnCode fxGetSelfIdData(
    FXBusHandle handle,
    uint32_t generation,
    size_t maxSize,
    size_t* size,
    uint32_t* buffer
)

```

10.5.2.4.2 fxIssueBusReset**Description**

This function can be used to issue a bus reset. The bus reset is generated by writing to the local PHY registers and returns immediately after the register write actions completed.

Parameters

handle	Reference handle to the bus to control. (see fxCreateBusHandle)
--------	---

shortBusReset	This parameter can be used as follows: 0: Issue a short bus reset by writing the ISBR PHY register other values: Issue a long bus reset by writing the IBR PHY register
---------------	---

Return Codes

This function returns zero or greater upon success. A negative return value indicates an error. Please use the function [fxGetErrorMessage](#) to lookup descriptions corresponding to negative return values.

FX_ERR_NORMAL
FX_ERR_INVALID_HANDLE
FX_ERR_LICENSE_MODULE
FX_ERR_LICENSE_EXPIRED
FX_ERR_FIRESTACK_DEMO_TIMEOUT

Synopsis

```
FXReturnCode fxIssueBusReset(
    FXBusHandle busHandle,
    uint32_t shortBusReset
)
```

10.5.3. Type Definitions**10.5.3.1. FXAsyRcvPacketCallback****Description**

This function definition is used to specify a callback function that will be called when the stack receives an incoming asynchronous packet.

Parameters

busHandle	Reference handle to the bus to control. (see fxCreateBusHandle)
userData	Pointer to the data specified in fxAsyRcvSetPacketCallback .
packet	Data structure that holds the packet details. (see FXAsyRcvPacket)

Synopsis

```
typedef bool_t (*FXAsyRcvPacketCallback) (
    FXBusHandle busHandle,
    void* userData,
    FXAsyRcvPacket* packet
);
```

10.5.3.2. FXPhyPacketRcvCallback**Description**

This function definition is used to specify a callback function that will be called when the stack receives an incoming PHY packet.

Parameters

busHandle	Reference handle to the bus to control. (see fxCreateBusHandle)
phyPacket	Data quadlet of the PHY packet received. The second quadlet of the original packet is the inverse of the first quadlet by definition and therefore is not communicated to the user.
speedCode	Speed code of the PHY packet received
userData	Data provided by user when this callback function was registered.

Synopsis

```
typedef void (*FXPhyPacketRcvCallback) (
    FXBusHandle busHandle,
    uint32_t phyPacket,
    uint32_t speedCode,
    void* userData
);
```

10.5.4. Structures**10.5.4.1. FXAsyRcvPacket****Description**

This structure defines details of a received Asynchronous Packet.

Members

generation	Bus generation number that corresponds to the packet
speed	Speed code of the packet (see Speed Codes)
ackCode	The IEEE-1394 ack code associated with the packet.
header	Data pointer to memory location holding the packet header contents. This pointer is only valid during execution of the callback.
headerSize	Size of the header expressed in bytes.
data	Data pointer to memory location holding the packet data contents. This pointer is only valid during execution of the callback.
dataSize	Size of the data expressed in bytes.

Synopsis

```
struct FXAsyRcvPacket
{
    uint32_t    generation;
    uint32_t    speed;
    uint32_t    ackCode;
    void*       header;
    size_t      headerSize;
    void*       data;
    size_t      dataSize;
};
```

10.5.5. Constants**10.5.5.1. Error Codes**

The following values may be returned by functions.

```
FX_ERR_LOW_LVL_RECEIVE_TIMEOUT
FX_ERR_LOW_LVL_PHY_REG_ACCESS_TIMEOUT
FX_ERR_LOW_LVL_PHY_MAX_OUTSTANDING
FX_ERR_LOW_LVL_PHY_FAILED
```

10.5.5.2. PHY Remote Commands

```
FX_PHY_CMD_NOP                0
FX_PHY_CMD_TX_DISABLE_NOTIFY  1
FX_PHY_CMD_INIT_SUSPEND       2
```

FX_PHY_CMD_PORT_CLEAR_PORT	4
FX_PHY_CMD_PORT_ENABLE	5
FX_PHY_CMD_PORT_RESUME	6
FX_PHY_CMDEXT_NOP	7
FX_PHY_CMDEXT_PORT_INIT_STANDBY	8
FX_PHY_CMDEXT_PORT_RESTORE	9

10.5.5.3. PHY Confirmation Flags

FX_PHY_REMOTECONF_BIT_OK	0x01
FX_PHY_REMOTECONF_BIT_DISABLED	0x02
FX_PHY_REMOTECONF_BIT_BIAS	0x04
FX_PHY_REMOTECONF_BIT_CONNECTED	0x08
FX_PHY_REMOTECONF_BIT_FAULT	0x10
FX_PHY_REMOTECONF_BIT_STANDBYFAULT	0x20

Chapter 11. Time Input Device API Reference

11.1. Functions

The External Time Input functions allow the user to control any supported time input device. For example a FireTrac card may contain one time input even though it contains three busses. All three buses will then be physically connected to this time input and therefore the time input is controlled as a separate device from the FireStack API.

The [FXTimeInputInfo](#) structure contains the required PCI information to determine which time input device is physically located on the same PCI device as any busses the user is controlling.

The following steps should be carried out to setup a time input device for time synchronization:

1. Create a time input device handle using [fxCreateTimeInputHandle](#)
2. Register a user-defined callback function that will be called whenever the status changes. This can be done with the function [fxSetTimeInputStatusCallback](#).
3. Set the [time input mode](#) and enable external time input by using the function [fxSetTimeInputMode](#).
4. If the time has been synchronized or if synchronization failed, the user callback status function will be called. If synchronization was successful, the user can get current time information by using the function [fxGetTimeInputStatus](#).

11.1.1. fxGetNumberOfTimeInputs

Description

This function can be used to determine how many time input devices are available. Typically the user will call [fxGetTimeInputInfoList](#) after this function to get more information per time input device.

Parameters

numTimeInputs	Returns the number of time inputs
---------------	-----------------------------------

Return Codes

This function returns zero or greater upon success. A negative return value indicates an error. Please use the function [fxGetErrorMessage](#) to lookup descriptions corresponding to negative return values.

Synopsis

```
FXReturnCode fxGetNumberOfTimeInputs(
    uint32_t* numTimeInputs
)
```

11.1.2. fxGetTimeInputInfoList

Description

This function can be used to fill a user-allocated list of [FXTimeInputInfo](#) structures.

Parameters

list	User-allocated list that will be filled with time input information. For details, please refer to FXTimeInputInfo .
maxSize	The maximum number of FXTimeInputInfo items the FireStack is allowed to return.
size	The actual number of FXTimeInputInfo items the FireStack has returned.

Return Codes

This function returns zero or greater upon success. A negative return value indicates an error. Please use the function [fxGetErrorMessage](#) to lookup descriptions corresponding to negative return values.

FX_ERR_INVALID_PARAMETER**Synopsis**

```
FXReturnCode fxGetTimeInputInfoList(
    FXTimeInputInfo* list,
    uint32_t maxSize,
    uint32_t* size
)
```

11.1.3. fxCreateTimeInputHandle**Description**

This function can be used to open a time input device and get a handle to it. The user has the option to either manually create an [FXTimeInputInfo](#) structure with valid information or to use one of the [FXTimeInputInfo](#) structs returned by [fxGetTimeInputInfoList](#).

When opening a handle the user may choose to specify a list of settings by providing the function with an array of [FXSetting](#) structures. Available settings depend on various aspects like operating system used, FireStack modules included in this specific FireStack release and higher-level protocols included in this specific FireStack release. Whenever a module offers user-configurable settings it will include them in the module's documentation.

Most developers will want to start using this function without specifying any settings until they learn about a specific setting that they may find useful somewhere else in this manual.

Parameters

info	FXTimeInputInfo structure of the time input device to open. Missing fields will be filled in on successful open.
settingList	An array of struct type FXSetting that allows providing settings when creating a handle or one can set this parameter to zero to leave out settings.
size	The number of settings in the provided settingList array.
handle	The returned handle to the time input device.

Return Codes

This function returns zero or greater upon success. A negative return value indicates an error. Please use the function [fxGetErrorMessage](#) to lookup descriptions corresponding to negative return values.

FX_ERR_INVALID_PARAMETER
FX_ERR_DEVICE_INIT_FAIL
FX_ERR_NEEDS_FWUPDATE

Synopsis

```
FXReturnCode fxCreateTimeInputHandle(
    FXTimeInputInfo* info,
    FXSetting* settingList,
    size_t size,
    FXTimeInputHandle* handle
)
```

11.1.4. fxCloseTimeInputHandle**Description**

This function can be used to close a time input device.

Parameters

handle	The handle of the time input device to close.
--------	---

Return Codes

This function returns zero or greater upon success. A negative return value indicates an error. Please use the function [fxGetErrorMessage](#) to lookup descriptions corresponding to negative return values.

FX_ERR_INVALID_PARAMETER
FX_ERR_DEVICE_CLOSE_FAIL

Synopsis

```
FXReturnCode fxCloseTimeInputHandle(
    FXTimeInputHandle handle
)
```

11.1.5. fxSetTimeInputMode**Description**

Select the desired [mode](#) for the time input device.

Parameters

handle	Handle to the time input device to control.
mode	The desired input time mode .

Return Codes

This function returns zero or greater upon success. A negative return value indicates an error. Please use the function [fxGetErrorMessage](#) to lookup descriptions corresponding to negative return values.

FX_ERR_FIRESTACK_NOT_INITIALIZED
FX_ERR_INVALID_TIMEMODE
FX_ERR_INVALID_PARAMETER
FX_ERR_TIMEMODE_NOT_SUPPORTED

Synopsis

```
FXReturnCode fxSetTimeInputMode(
    FXTimeInputHandle handle,
    uint32_t mode
);
```

11.1.6. fxSetTimeInputCurrentYear**Description**

Set the current year. This can be useful for time modes that do not provide this information.

Parameters

handle	Handle to the time input device to control.
year	The current year.

Return Codes

This function returns zero or greater upon success. A negative return value indicates an error. Please use the function [fxGetErrorMessage](#) to lookup descriptions corresponding to negative return values.

FX_ERR_INVALID_PARAMETER
FX_ERR_INVALID_TIMEMODE

Synopsis

```
FXReturnCode fxSetTimeInputCurrentYear(
    FXTimeInputHandle handle,
    uint32_t year
);
```

11.1.7. fxSetTimeInputFreeRunningOffset**Description**

Set the current time offset in free running mode.

Parameters

handle	Handle to the time input device to control.
seconds	Starting offset for free running time, such as the number of seconds since 01/01/1970.

Return Codes

This function returns zero or greater upon success. A negative return value indicates an error. Please use the function [fxGetErrorMessage](#) to lookup descriptions corresponding to negative return values.

FX_ERR_INVALID_PARAMETER
FX_ERR_INVALID_TIMEMODE

Synopsis

```
FXReturnCode fxSetTimeInputFreeRunningOffset(
    FXTimeInputHandle handle,
    uint32_t seconds
);
```

11.1.8. fxGetTimeInputStatus**Description**

This function can be used to get the current status of a time input device.

Parameters

handle	Handle to the time input device to get the status from.
status	Returns the status. For more details, refer to FXTimeInputStatus .

Return Codes

This function returns zero or greater upon success. A negative return value indicates an error. Please use the function [fxGetErrorMessage](#) to lookup descriptions corresponding to negative return values.

FX_ERR_INVALID_PARAMETER
FX_ERR_INVALID_TIMEVALUE

Synopsis

```
FXReturnCode fxGetTimeInputStatus(
    FXTimeInputHandle handle,
    FXTimeInputStatus* status
)
```

11.1.9. fxSetTimeInputStatusCallback**Description**

Set a function to be called each time the status of the time input module changes.

Parameters

handle	Handle to the time input device to set the callback for.
callback	Pointer to user-defined callback function.
userData	Pointer to a user-specified data. The pointer will be carried to user callback functions. See also FXTimeInputStatusCallback .

Return Codes

This function returns zero or greater upon success. A negative return value indicates an error. Please use the function [fxGetErrorMessage](#) to lookup descriptions corresponding to negative return values.

FX_ERR_INVALID_PARAMETER**Synopsis**

```
FXReturnCode fxSetTimeInputStatusCallback(
    FXTimeInputHandle          handle,
    FXTimeInputStatusCallback callback,
    void*                    userData
)
```

11.1.10. fxSetTimeInputSecondCallback**Description**

Set a function to be called each time the seconds counter is incremented.

Parameters

handle	Handle to the time input device to set the callback for.
callback	Pointer to user-defined callback function.
userData	Pointer to a user-specified data. The pointer will be carried to user callback functions. See also FXTimeInputSecondsCallback .

Return Codes

This function returns zero or greater upon success. A negative return value indicates an error. Please use the function [fxGetErrorMessage](#) to lookup descriptions corresponding to negative return values.

FX_ERR_INVALID_PARAMETER**Synopsis**

```
FXReturnCode fxSetTimeInputStatusCallback(
    FXTimeInputHandle          handle,
    FXTimeInputSecondCallback callback,
    void*                    userData
)
```

11.2. Type Definitions

11.2.1. FXTimeInputHandle

FXTimeInputHandle	Reference handle to the time input device to control. (see fxCreateTimeInputHandle)
-------------------	---

11.2.2. FXTimeInputStatusCallback

Description

This type definition is used to specify a callback function that can be registered as status change callback function for a time input device by using the function [fxSetTimeInputStatusCallback](#).

Parameters

handle	Reference handle to the bus to control. (see fxCreateBusHandle)
userData	Pointer to the data specified in FxSetTimeInputStatusCallback .
statusCode	The new status of the time input device.

Synopsis

```
typedef void (*FXTimeInputStatusCallback) (
    FXTimeInputHandle handle,
    void*          userData,
    uint32_t       statusCode
);
```

11.2.3. FXTimeInputSecondCallback

Description

This type definition is used to specify a callback function that can be registered as a seconds counter increment callback function for a time input device by using the function [fxSetTimeInputSecondCallback](#).

Parameters

handle	Reference handle to the bus to control. (see fxCreateBusHandle)
userData	Pointer to the data specified in FxSetTimeInputSecondCallback .

Synopsis

```
typedef void (*FXTimeInputSecondCallback) (
    FXTimeInputHandle handle,
    void*          userData
);
```

11.3. Structures

11.3.1. FXTimeInputInfo

Description

This structure defines data members that together provide sufficient information to identify a device. When used in combination with [fxCreateTimeInputHandle](#) sufficient information needs to be filled in to identify the device to be opened, all other fields may be set to -1.

For example if a user just wants to open the first time input device found in the system it is sufficient to set `deviceId` to 0 and the remaining fields to -1 when calling [fxCreateTimeInputHandle](#).

When a user wants to open a specific PCI physical location all PCI fields should be filled in correctly and all other fields need to be set to -1 when calling [fxCreateTimeInputHandle](#).

Simplest way to use this structure to open a device is to let the FireStack fill in the fields. This can be done by using the function [fxGetTimeInputInfoList](#).

Members

<code>deviceId</code>	Device identification number. Unique for each time input device connected to the system. This number is not related to physical configuration but picked by software.
<code>pciBus</code>	PCI bus number for PCI devices, -1 otherwise. Indicates the PCI bus number the time input device is connected to. Numbering is defined by the physical PCI bus topology.
<code>pciDevice</code>	PCI device number for PCI devices, -1 otherwise. Numbering is defined by the physical PCI bus topology.
<code>pciFunction</code>	PCI function for PCI devices, -1 otherwise. In case of a multi-function PCI device, this field holds the function index of the time input module.
<code>pciRevision</code>	PCI Revision for PCI devices, -1 otherwise. In case of FireTrac devices this represents the firmware version.
<code>subSecondResolution</code>	Time stamps are split up in a number of seconds and a number of sub-second increments. This field defines the resolution of the sub-second increments in nano seconds. (1000 means sub-seconds expressed in micro seconds)

Synopsis

```
typedef struct {
    /* Identification */
    int32_t      deviceId;

    /* PCI Physical location */
    int32_t      pciBus;
    int32_t      pciDevice;
    int32_t      pciFunction;

    /* Information */
    int32_t      pciRevision;
}
```

```

    /* Device Capability Information */
    uint32_t      subSecondResolution;
} FXTimeInputInfo;

```

11.3.2. FXTimeInputStatus

Description

This structure defines data members that reflect the current status of a time input device.

Members

statusCode	Current status of the device
mode	The input time mode used
seconds	Number of seconds that have passed since 01/01/1970 assuming the input time uses UTC.
subSeconds	Number of sub-second intervals that have passed since the last seconds increment. For the resolution of sub-seconds, please see FXTimeInputInfo structure.

Synopsis

```

typedef struct {
    uint32_t      statusCode;
    uint32_t      mode;
    uint32_t      seconds;
    uint32_t      subSeconds;
} FXTimeInputStatus;

```

11.4. Constants

11.4.1. Time Input Mode

The Time Input Modes supported depend on the particular hardware model used. In case [fxSetTimeInputMode\(\)](#) is called with an unsupported mode it will indicate this by returning `FX_ERR_TIMEMODE_NOT_SUPPORTED`.

The following table shows the different modes that are available to the user. Depending on which mode is chosen one or more options from the mode options table may be OR-ed together with the selected mode.

<code>FX_TIMEINPUTMODE_FREERUNNING</code>	Free running mode uses the internal clock without any synchronization to an external source. This mode allows the user to set the current time to a specific value by using the function fxSetTimeInputFreeRunningOffset() .
<code>FX_TIMEINPUTMODE_IRIG</code>	Synchronize to an IRIG-B time source without IEEE-1344 extensions. This mode allows the user to specify the current year by using the function fxSetTimeInputCurrentYear() . By default this mode selects the modulated IRIG-B122 format which can be overridden by using either <code>FX_TIMEINPUTMODE_OPTION_TTL</code> or <code>FX_TIMEINPUTMODE_OPTION_RS422</code> .
<code>FX_TIMEINPUTMODE_IRIG_1344</code>	Synchronize to an IRIG-B time source with IEEE-1344 extensions. Using this mode the year will be used from the time source. By default this mode selects the modulated IRIG-B122 format which can be overridden by using either <code>FX_TIMEINPUTMODE_OPTION_TTL</code> or <code>FX_TIMEINPUTMODE_OPTION_RS422</code> .

The following mode options can be OR-ed with each other and one of the modes above to define the complete mode to be used:

<code>FX_TIMEINPUTMODE_OPTION_TTL</code>	This option changes the IRIG format from B-122 to B-002 TTL. By default, this option selects non-inverted active-high signaling which can be overridden by setting the <code>FX_TIMEINPUTMODE_OPTION_INVERTED</code> option. This option is only allowed in combination with <code>FX_TIMEINPUTMODE_OPTION_IRIG</code> or <code>FX_TIMEINPUTMODE_OPTION_IRIG_1344</code> .
<code>FX_TIMEINPUTMODE_OPTION_RS422</code>	This option changes the IRIG format from B-122 to B-002 RS422. By default, this option selects non-inverted active-high signaling which can be overridden by setting the <code>FX_TIMEINPUTMODE_OPTION_INVERTED</code> option. This option is only allowed in combination with <code>FX_TIMEINPUTMODE_OPTION_IRIG</code> or <code>FX_TIMEINPUTMODE_OPTION_IRIG_1344</code> .
<code>FX_TIMEINPUTMODE_OPTION_INVERTED</code>	This option changes the signaling from non-inverted active high to inverted active-low. This option is only allowed in combination with <code>FX_TIMEINPUTMODE_OPTION_TTL</code> or <code>FX_TIMEINPUTMODE_OPTION_RS422</code> .

Only the following combinations of options are allowed:

Free Running	FX_TIMEINPUTMODE_FREERUNNING
IRIG-B122	FX_TIMEINPUTMODE_IRIG
IRIG-B122-1344	FX_TIMEINPUTMODE_IRIG_1344
IRIG-B002-TTL	FX_TIMEINPUTMODE_IRIG FX_TIMEINPUTMODE_OPTION_TTL
IRIG-B002-1344-TTL	FX_TIMEINPUTMODE_IRIG_1344 FX_TIMEINPUTMODE_OPTION_TTL
IRIG-B002-TTL Inverted Polarity	FX_TIMEINPUTMODE_IRIG FX_TIMEINPUTMODE_OPTION_TTL FX_TIMEINPUTMODE_OPTION_INVERTED
IRIG-B002-1344-TTL Inverted Polarity	FX_TIMEINPUTMODE_IRIG_1344 FX_TIMEINPUTMODE_OPTION_TTL FX_TIMEINPUTMODE_OPTION_INVERTED
IRIG-B002-RS422	FX_TIMEINPUTMODE_IRIG FX_TIMEINPUTMODE_OPTION_RS422
IRIG-B002-1344-RS422	FX_TIMEINPUTMODE_IRIG_1344 FX_TIMEINPUTMODE_OPTION_RS422
IRIG-B002-RS422 Inverted Polarity	FX_TIMEINPUTMODE_IRIG FX_TIMEINPUTMODE_OPTION_RS422 FX_TIMEINPUTMODE_OPTION_INVERTED
IRIG-B002-1344-RS422 Inverted Polarity	FX_TIMEINPUTMODE_IRIG_1344 FX_TIMEINPUTMODE_OPTION_RS422 FX_TIMEINPUTMODE_OPTION_INVERTED

For backwards compatibility, the following alternative mode definitions are also supported:

FX_TIMEINPUTMODE_IRIG_B122	Exactly the same as FX_TIMEINPUTMODE_IRIG
FX_TIMEINPUTMODE_IRIG_B122_1344	Exactly the same as FX_TIMEINPUTMODE_IRIG_1344

11.4.2. Status Codes

FX_TIMEINPUTSTATUS_INVALID	No time input source signal has been detected
FX_TIMEINPUTSTATUS_FREERUNNING	Current time is free running and not synced to an IRIG source
FX_TIMEINPUTSTATUS_SYNCED	Current time is synchronized to an IRIG source
FX_TIMEINPUTSTATUS_SYNCEDACCURATE	Current time is synchronized to a stable IRIG source

11.4.3. Error Codes

The following values may be returned by Time Input Control functions.

FX_ERR_INVALID_TIMEVALUE
FX_ERR_INVALID_TIMEMODE
FX_ERR_INVALID_TIMEFORMAT
FX_ERR_TIMEMODE_NOT_SUPPORTED

Chapter 12. Examples

The software package comes with several examples to illustrate how each module can be used to perform certain tasks on the 1394 bus. This section provides a description of each of the examples. Please be aware that to be able to run an example one needs a valid license for the module that is demonstrated by the example. It is also possible to run the examples in demo mode for a maximum of 15 minutes. In this case please make sure that no license certificate is present in the flash memory of the board.

For each example a compiled binary is provided that can be run as-is. It is also possible to build the examples from source. Since it is not practical to provide a project file for each possible development environment used by our customers, we decided to provide a configuration file that can be used by CMake. CMake is freely available from the following website (<http://www.cmake.org>) and many operating systems include it in their package managers.

12.1. Inbound Transactions

12.1.1. Inbound Transaction Monitor

The Inbound Transaction Monitor demonstrates functions defined in section [Inbound Transactions](#).

When the demo application starts, the console should display a welcome message and a menu. With help of this menu you can give simple commands to evaluate the functionality of the Inbound Transaction API function set. The initial display should look like the following:

```

-----
--          Welcome to FireLinkExtended Inbound Transaction Monitor          --
-----
This example demonstrates the two methods of handling Inbound Transactions:
Map Local Memory and Transaction Handler.
You are supposed to use another device to perform the read/write/lock requests

Map Local Memory is demonstrated by letting you enter a StartAddress and Size.
The address is entered in HexaDecimal and the size in decimal. For this example
0xFFFF00000000 is added to the StartAddress, so entering a value of F0000400
results in a start address of 0xFFFFF0000400

Bus opened

-----
--          FireLinkExtended Inbound Transaction Monitor Menu              --
-----
?                                help (this menu)
--
-- Region commands
ml <address> <size>                map local memory region
mt                                map transaction
tm <n|e|d|t|a|c>                  transaction mode - n: Normal,
                                e: No response, d: Data error, t: Type Error
                                a: Address error, c: Conflict Error
--
-- Generic commands
--
qu                                quit demo
vb                                verbose on
>

```

Map Local Memory Region

The command

```
ml <address> <size>
```

creates a new memory mapped address range that is exposed onto the 1394 bus. FireStack autonomously transmits a response packet when it receives a request packet with matching address. If a write request packet with the valid address is received, the FireStack will update the local memory with the data payload of the received packet. Receiving a read request packet will transmit a response packet with data stored in the local memory.

Map Local Memory is demonstrated by letting you enter a start <address> and <size>. The address is entered in hexadecimal and the size in decimal. For this example 0xFFFF00000000 is added to the start address, so entering a value of F0000400 results in a start address of 0xFFFFF0000400

Map Transaction command

The command

```
mt
```

creates a new handler mapped address range from 0x00FFF000020 to 0x00FFF00009F (inclusive) with write access permission on the 1394 bus. This command is very similar to the Map Local Memory Region command above except that it does not map the local memory and does not transmit a response packet. Instead, this command calls an event callback function upon a reception of a valid request packet.

Transaction Mode

The command

```
tm <n|e|d|t|a|c>
```

can be used to set the transaction mode. The transaction mode determines the response code that will be transmitted upon a request as part of the response packet. The following response codes can be set as parameter for this command:

- e: No response
- d: Data error
- t: Type Error
- a: Address error
- c: Conflict Error

Other commands

Use '?' (help) command to display the menu. Use the 'qu' (quit) command to terminate the demo application. Use the 'vb' command to enable more verbose output when inbound requests are received that match a mapped address range.

12.2. Outbound Transactions

12.2.1. Outbound Transaction Demo

The Outbound Transaction Demo demonstrates functions defined in section [Outbound Transactions](#).

When the demo application starts, the console should display a welcome message and a menu. With help of this menu you can give simple commands to evaluate the functionality of the FireStack Outbound Transaction API function set. The initial display should look like the following:

```

-----
-- Welcome to FireLinkExtended Outbound Transaction Demo!!!
-----

Bus opened

-----
-- FireLinkExtended Outbound Transaction Demo Menu
-----
?                               help (this menu)
--
-- Request commands
--
rd <dest node> [size=4]          read  request (blocking)
wr <dest node> [size=4]          write request (blocking)
rdn <dest node> [size=4]        read  request (non-blocking)
--
-- Generic commands
--
qu                               quit demo
>

```

Read Request Command

The command

```
rd <dest node> [size]
```

can be used to send a read request packet for address 0xFFFFF0000400 to <dest node> and wait for a response. If a response packet is not received within 500ms, the command will display a timeout error message. The <dest node> needs to be specified without the bus number and should be between 0 and 63. The optional [size] parameter determines the number of bytes to read and needs to be a multiple of 4.

Write Request Command

The command

```
wr <dest node> [size]
```

can be used to send a write request packet for address 0xFFFFF0000800 to <dest node> and wait for a response. If a response packet is not received within 2s, the command will display a timeout error message. The <dest node> needs to be specified without the bus number and should be between 0 and 63. The optional [size] parameter determines the number of bytes to write and needs to be a multiple of 4.

Non-blocking Read Request Command

The command

```
rdn <dest node> [size]
```

can be used to send a read request packet for address 0xFFFFF0000400 to <dest node> and will return

immediately. The callback function will be called to indicate a successful reception of a response packet or a timeout error if no response is received within 2 seconds. The <dest node> needs to be specified without the bus number and should be between 0 and 63. The optional [size] parameter determines the number of bytes to read and needs to be a multiple of 4.

Other commands

Use '?' (help) command to display the menu. Use the 'qu' (quit) command to terminate the demo application.

12.3. Low-Level 1394

12.3.1. Low-level Demo

The Low-Level Demo demonstrates some functions defined in section [Low-level 1394](#).

When the demo application starts the console displays a welcome message and a menu. With help of this menu you can give simple commands to evaluate the functionality of the FireStack Low Level API function set. The initial display should look like the following:

```

-----
-- Welcome to FireLinkExtended Low Level Demo!!!
-----
Bus opened

-----
-- FireLinkExtended Low Level Demo Menu
-----
?                help (this menu)
--
-- Receive commands
--
srq <timeout>    single req  packet receive
srs <timeout>    single resp packet receive
--
-- Send commands
--
sp                send async packets
--
-- Phy Register commands
--
rp <reg>         read local phy register
wp <reg> <value> write local phy register
--
-- Generic commands
--
qu                quit demo
>

```

Receive a single request packet

The command

```
srq <timeout>
```

can be used to receive a single request packet that is sent to the local node. The command will return with a timeout error if no packet is received within <timeout> ms.

Receive a single response packet

The command

```
srs <timeout>
```

can be used to receive a single response packet that is sent to the local node. The command will return with a timeout error if no packet is received within <timeout> ms.

Send asynchronous packets

The command

```
sp
```

can be used to transmit several kinds of asynchronous packets. When issued it will transmit one Write Quadlet Request, one Write Block Request, one Write Response, one Read Quadlet Request, one Read Block Request, one Read Quadlet Response, one Read Block Response, one Lock Request, one Asynchronous Stream, and one Lock Response packet.

Read local PHY register

The command

```
rp <reg>
```

can be used to read a local PHY register. When issued it will read register offset <reg> and display the contents of the local PHY register.

Write local PHY register

The command

```
wp <reg> <value>
```

can be used to write <value> to local register offset <reg>.

Other commands

Use '?' (help) command to display the menu. Use the 'qu' (quit) command to terminate the demo application.

12.4. Isochronous Reception

12.4.1. Isochronous Reception Demo

The Isochronous Reception Demo demonstrates some functions defined in section [Isochronous Reception](#).

When the demo application starts the console displays a welcome message and a menu. With help of this menu you can give simple commands to evaluate the functionality of the FireStack Isochronous Reception API function set. The initial display should look like the following:

```

-----
---  Welcome to Isochronous Reception Demo      ---
-----

*** 3 Devices found
0: deviceId: 0, pciBus: 6, pciDevice: 6, pciFunc: 0, pciRev: 1
1: deviceId: 1, pciBus: 6, pciDevice: 5, pciFunc: 0, pciRev: 1
2: deviceId: 2, pciBus: 6, pciDevice: 4, pciFunc: 0, pciRev: 1

-----
---      FireStack Isochronous Reception Demo Menu      ---
-----
?                help (this menu)

--- Bus Commands ---

ob <device num>      open bus
cb                  close bus

--- Context Commands ---

crc <F | P | D> [M | C]  create context
                      <buff-Fill | Pkt-per-buff | Dual-buff>
                      [multi-channel | cycle-match]
clc <contextID>       close context
gc <contextID>        start context
hc <contextID>        stop context
cs <contextID>        context status

--- Generic Commands ---

qu                  quit application

>

```

Open a bus

The command

```
ob <device num>
```

can be used to open a bus on the specified device. The device number should be one of the first numbers displayed at the beginning of the welcome message.

Close a bus

The command

```
cb
```

can be used to close an open bus that has been opened by the "ob" command. This application can open one bus at any given time.

Create a context

The command

```
crc <F | P | D> [M | C]
```

can be used to create a context which is required in order to set up receive buffers. A user can choose one of three buffer modes: Buffer-Fill, Packet-per-Buffer, or Dual-Buffer mode. Optionally, a context can be created with the multi-channel and/or the cycle-match options. Please refer to the [Isochronous Reception](#) section for details. This command returns a number that is assigned to a newly created context. The number will be needed for other commands described below.

Close a context

The command

```
clc <contextID>
```

can be used to close a context that is created by the "crc" command and frees all resources associated with the context.

Start a context

The command

```
gc <contextID>
```

can be used to start isochronous reception on the specified context.

Stop a context

The command

```
hc <contextID>
```

can be used to stop isochronous reception on the specified context.

Display context status

The command

```
cs <contextID>
```

can be used to display the current status of the specified context. Please refer to the [Isochronous Reception](#) section for details.

Other commands

Use '?' (help) command to display the menu. Use the 'qu' (quit) command to terminate the demo application.

12.5. Isochronous Transmission

12.5.1. Isochronous Transmission Demo

The Isochronous Transmission Demo demonstrates some functions defined in section Isochronous Transmission.

When the demo application starts the console displays a welcome message and a menu. With help of this menu you can give simple commands to evaluate the functionality of the FireStack Isochronous Transmission API function set. The initial display should look like the following:

```

-----
--- Welcome to Isochronous Transmit Demo ---
-----

*** 3 Devices found
0: deviceId: 0, pciBus: 5, pciDevice: 4, pciFunc: 0, pciRev: 10
1: deviceId: 1, pciBus: 5, pciDevice: 4, pciFunc: 1, pciRev: 10
2: deviceId: 2, pciBus: 5, pciDevice: 4, pciFunc: 2, pciRev: 10

-----
--- FireStack Isochronous Transmit Demo Menu ---
-----
?                               help (this menu)

--- Bus Commands ---

ob <device num>                 open bus
cb                               close bus

--- Context Commands ---

crc <channel> <speed> [L]       create context
                                [L: loop]
clc <contextID>                 close context
gc <contextID> [C]             start context
                                [C: cycle-match]
hc <contextID>                 stop context
qc <contextID>                 get context status

--- Generic Commands ---

qu                               quit application

>

```

Open a bus

The command

```
ob <device num>
```

can be used to open a bus on the specified device. The device number should be one of the first numbers displayed at the beginning of the welcome message.

Close a bus

The command

```
cb
```

can be used to close an open bus that has been opened by the "ob" command. This application can open

one bus at any given time.

Create a context

The command

```
crc <channel> <speed> [L]
```

can be used to create a Isochronous Transmission context which is required in order to transmit data. A user can choose a channel number (0 to 63) and speed (0: 100 Mbps, 1: 200 Mbps, 2: 400 Mbps). A loop option can also be selected by adding a letter 'L' at the end of the command line. This command will set up a list of packets and skips (no packet transmission) for 1600 cycles; every 4th cycle has a skip (no transmission), every 9th cycle has a header-only packet, all other cycles have packets being transmitted with its data payload size of 1024 bytes, and the last cycle with a callback option. Please refer to the Isochronous Transmission section for details. This command returns a number that is assigned to a newly created context. The number will be needed for other commands described below.

Close a context

The command

```
clc <contextID>
```

can be used to close a context that is created by the "crc" command and frees all resources associated with the context.

Start a context

The command

```
gc <contextID>
```

can be used to start isochronous transmission on the specified context. Optionally, a letter 'C' at the end of the command line will ask you to enter a cycle number at which the transmission will start.

Stop a context

The command

```
hc <contextID>
```

can be used to stop isochronous transmission on the specified context.

Query context status

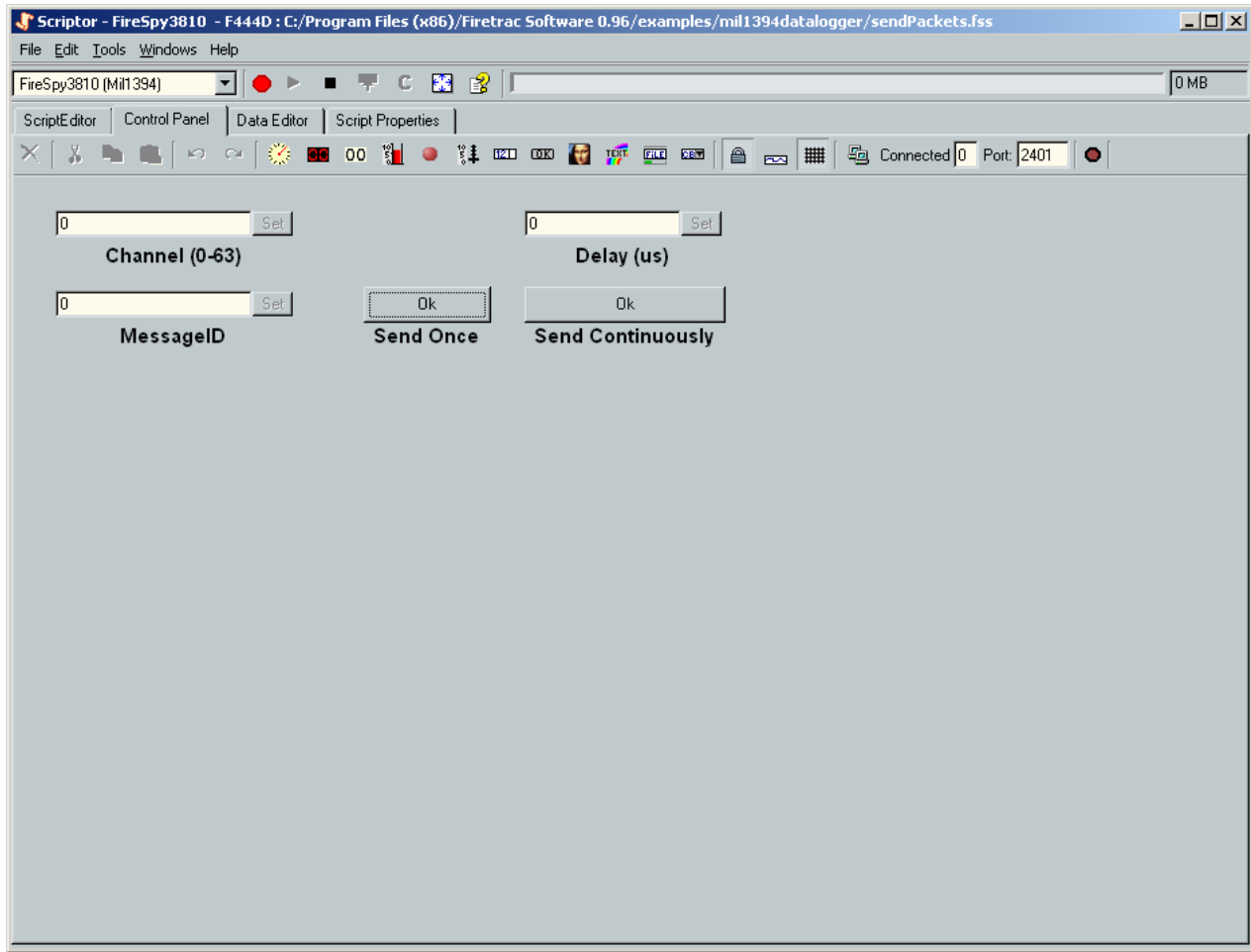
The command

```
qc <contextID> [C]
```

can be used to display the current status of the specified context. Please refer to the Isochronous Transmission section for details.

Other commands

Use '?' (help) command to display the menu. Use the 'qu' (quit) command to terminate the demo application.



When pressing the Send Once button a single packet will be transmitted that should be filtered-in by the Mil1394 Data Logger and after pressing the button a couple of times the console output should look something like the following:

```

Bus[0]: Opened
Bus[0]: Channelmask:11111000000000000000000000000001000000000000000000000000000000000000000000
Bus[0]: anyChan: 0, channel: 31, anyMessID: 1, messID: 4294967295, context: 2561
Bus[0]: anyChan: 0, channel: 0, anyMessID: 1, messID: 4294967295, context: 2561
Bus[0]: anyChan: 0, channel: 1, anyMessID: 0, messID: 11, context: 2561
Bus[0]: anyChan: 0, channel: 2, anyMessID: 0, messID: 12, context: 2561
Bus[0]: anyChan: 0, channel: 3, anyMessID: 0, messID: 13, context: 2561
Bus[0]: anyChan: 0, channel: 4, anyMessID: 0, messID: 14, context: 2561
Bus[0] Context[0]: Options: bufferCB: 1, packet CB: 1
Bus[1]: Opened
Bus[1]: Channelmask:11111000000000000000000000000001000000000000000000000000000000000000000000
Bus[1]: anyChan: 0, channel: 31, anyMessID: 1, messID: 4294967295, context: 5377
Bus[1]: anyChan: 0, channel: 0, anyMessID: 1, messID: 4294967295, context: 5377
Bus[1]: anyChan: 0, channel: 1, anyMessID: 0, messID: 11, context: 5377
Bus[1]: anyChan: 0, channel: 2, anyMessID: 0, messID: 12, context: 5377
Bus[1]: anyChan: 0, channel: 3, anyMessID: 0, messID: 13, context: 5377
Bus[1]: anyChan: 0, channel: 4, anyMessID: 0, messID: 14, context: 5377
Bus[1] Context[0]: Options: bufferCB: 1, packet CB: 1
Bus[2]: Opened
Bus[2]: Channelmask:11111000000000000000000000000001000000000000000000000000000000000000000000
Bus[2]: anyChan: 0, channel: 31, anyMessID: 1, messID: 4294967295, context: 8193
Bus[2]: anyChan: 0, channel: 0, anyMessID: 1, messID: 4294967295, context: 8193
Bus[2]: anyChan: 0, channel: 1, anyMessID: 0, messID: 11, context: 8193
Bus[2]: anyChan: 0, channel: 2, anyMessID: 0, messID: 12, context: 8193
Bus[2]: anyChan: 0, channel: 3, anyMessID: 0, messID: 13, context: 8193
Bus[2]: anyChan: 0, channel: 4, anyMessID: 0, messID: 14, context: 8193
Bus[2] Context[0]: Options: bufferCB: 1, packet CB: 1
Bus[0] Context[00000000]: Started Reception
Bus[1] Context[00000000]: Started Reception
Bus[2] Context[00000000]: Started Reception
Bus[0] Cntxt[0] Channel[0]
    [0x00000000] [0x12345678] [0x00000001] [0x01000100]
    [0x00000001] [0x00000000] [0x00000000] [0x00000000]
Bus[0] Cntxt[0] Channel[0]
    [0x00000000] [0x12345678] [0x00000001] [0x01000100]
    [0x00000001] [0x00000001] [0x00000000] [0x00000000]
Bus[0] Cntxt[0] Channel[0]
    [0x00000000] [0x12345678] [0x00000001] [0x01000100]
    [0x00000001] [0x00000002] [0x00000000] [0x00000000]
Bus[0] Cntxt[0] Channel[0]
    [0x00000000] [0x12345678] [0x00000001] [0x01000100]
    [0x00000001] [0x00000003] [0x00000000] [0x00000000]

```

The example will exit when pressing a key.

12.6.2. Mil1394 Receive Demo

The Mil1394 Reception Demo demonstrates some functions defined in section [AS5643 Reception](#).

When the demo application starts the console displays a welcome message and a menu. With help of this menu you can give simple commands to evaluate the functionality of the FireTrac AS5643 Reception API function set. The initial display should look like the following:

```

-----
-- Welcome to FireLinkExtended Mill394 Receive Demo!!!
-----

Bus opened
Channelmask:11111000000000000000000000001000000000000000000000000000000000000000
Number of filters: 6

channel  messageID  context
    0          any    2817
    1          11    3073
    2          12    3329
    3          13    2817
    4          14    3073
   31         any    2561

Context[0]: Started Reception
Context[1]: Started Reception
Context[2]: Started Reception
Context[3]: Started Reception

-----
-- FireLinkExtended Mill394 Receive Demo Menu
-----
?                               help (this menu)
--
-- Filter commands
--
af <channel> <messageId> <context>  add filter (for channel and messageId: 'a' = any)
rf <channel> <messageId>            remove filter (channel and messageId: 'a' = any)
lf                                  list filters
--
-- Generic commands
--
qu                                  quit demo
>

```

Add filter item

The command

```
af <channel> <messageId> <context>
```

can be used to add an entry to the current filter table. Matching packets with the same <channel> and <messageId> will then be received into the specified <context>. The example only supports 4 contexts.

Remove filter item

The command

```
rf <channel> <messageId>
```

can be used to remove a previously added filter entry from the current filter table. Please specify the <channel> and <messageId> exactly as when the original item was added.

List filter items

The command

```
lf
```

can be used to show the current contents of the filter table.

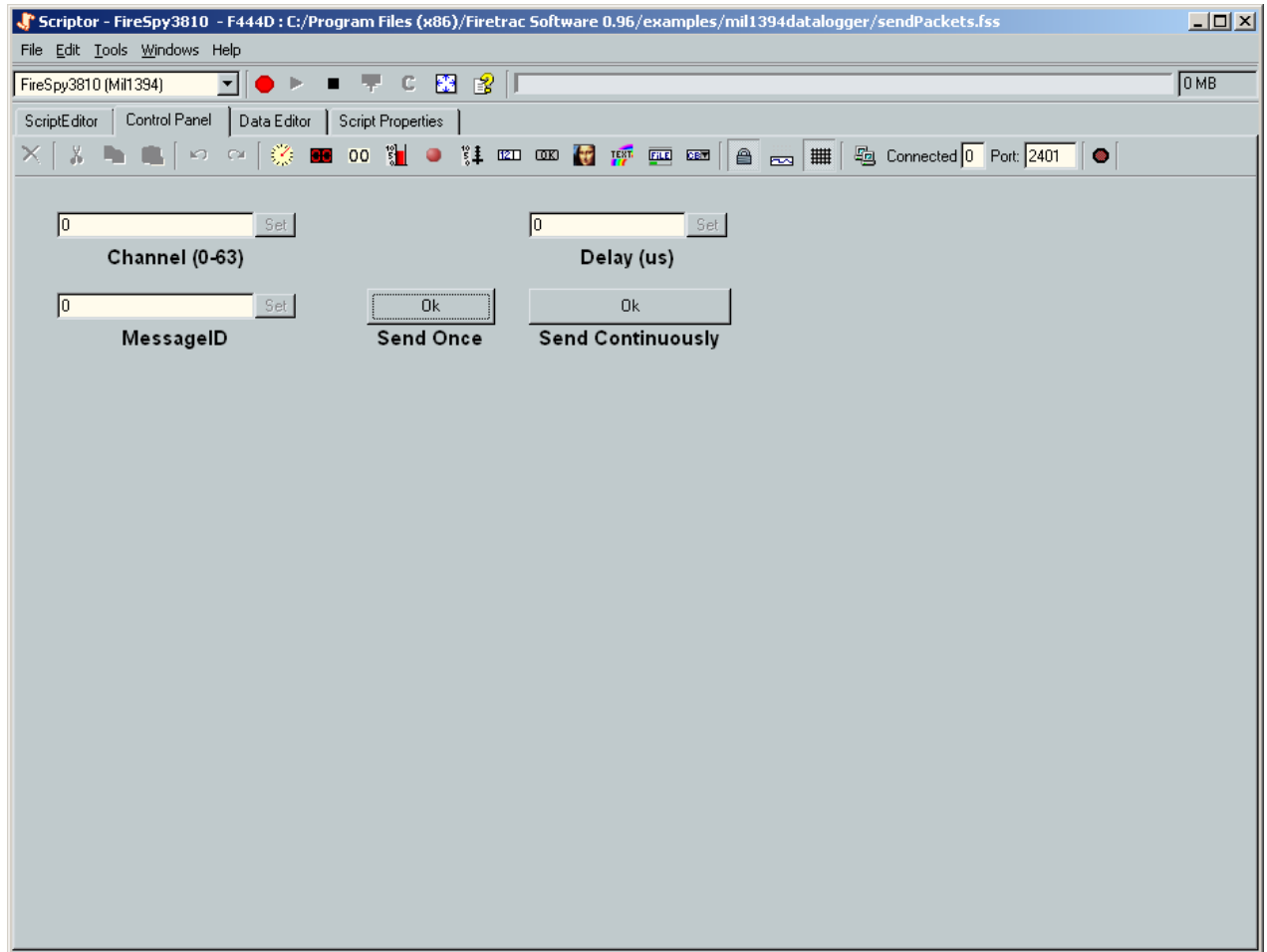
Other commands

Use '?' (help) command to display the menu. Use the 'qu' (quit) command to terminate the demo application.

Receiving messages

Another device connected to the 1394 bus can be used to transmit packets that match the current filter table entries.

In the folder where the source code of the example is located one can also find a FireSpy script that can be used to transmit matching packets. When the script is started the control panel should look like the following:



When pressing the Send Once button a single packet will be transmitted that should be filtered-in by the Mil1394 Data Logger and after pressing the button a couple of times the console output should look something like the following:

```
-----
-- Welcome to FireLinkExtended Mill394 Receive Demo!!!
-----

Bus opened
Channelmask:11111000000000000000000000000001000000000000000000000000000000
Number of filters: 6

channel  messageID  context
    0         any    2817
    1         11    3073
    2         12    3329
    3         13    2817
    4         14    3073
   31         any    2561

Context[0]: Started Reception
Context[1]: Started Reception
Context[2]: Started Reception
Context[3]: Started Reception

-----
-- FireLinkExtended Mill394 Receive Demo Menu
-----
?                               help (this menu)
--
-- Filter commands
--
af <channel> <messageId> <context>  add filter (for channel and messageId: 'a' = any)
rf <channel> <messageId>             remove filter (channel and messageId: 'a' = any)
lf                                   list filters
--
-- Generic commands
--
qu                                   quit demo

>Context[1] Channel[0]
   [0x00000000] [0x12345678] [0x00000001] [0x01000100]
   [0x00000001] [0x00000000] [0x00000000] [0x00000000]
Context[1] Channel[0]
   [0x00000000] [0x12345678] [0x00000001] [0x01000100]
   [0x00000001] [0x00000001] [0x00000000] [0x00000000]
Context[1] Channel[0]
   [0x00000000] [0x12345678] [0x00000001] [0x01000100]
   [0x00000001] [0x00000002] [0x00000000] [0x00000000]
```

12.7. AS5643 Transmission

12.7.1. Mil1394Transmit Demo

The Mil1394 Transmission Demo demonstrates some functions defined in section [AS5643 Transmission](#). Most functions do not offer many options to customize their execution. Main reason is that we wanted to offer simple and straightforward source code to show how this API works.

When the demo application starts the console displays a welcome message and a menu. With help of this menu you can give simple commands to evaluate the functionality of the FireTrac AS5643 Transmission API function set. The initial display should look like the following:

```

-----
--- Welcome to Mill1394 Transmit Demo ---
-----

*** 3 Devices found
0: deviceId: 0, pciBus: 6, pciDevice: 4, pciFunc: 0, pciRev: 5, devType: 4
1: deviceId: 1, pciBus: 6, pciDevice: 4, pciFunc: 1, pciRev: 5, devType: 4
2: deviceId: 2, pciBus: 6, pciDevice: 4, pciFunc: 2, pciRev: 5, devType: 4

-----
--- Mill1394 Transmit Demo Menu ---
-----
?                               help (this menu)

--- Bus Commands ---

ob <device num>                 open bus
cb                               close bus

--- Mill1394 Trm Single Mode ---

sms                             transmit single message
spms                            transmit single split message

--- Mill1394 Trm Stream Mode ---

rg <filename>                   read regen file as stream
sml                             stream message list
spml                            stream split message list
gst                             start stream
hst                             stop stream
cst                             clear stream
stst                            display stream status

--- Mill1394 Trm Repeating Message Mode ---

cms <msgsize> <ch> <offset> <jitter> create message
gms <msgnum>                   start message created with cms
hms <msgnum>                   stop message created with cms
wms <msgnum>                   close message created with cms
oms <msgnum> [c val(0-63)] [s val(0-2)] [v val(0-1)]
                               change message options
stms <msgnum>                  display message status created with cms
lms                             list messages

--- Mill1394 Trm STOF Message Mode ---

gf                               start STOF message
hf                               stop STOF message
wf                               write STOF message
of [c val(0-63)] [s val(0-2)] [v val(0-1)]
                               set STOF options

--- Mill1394 Trm Frame Sync Mode ---

fmf <extOut val(0-3)> <frameLength> <controlFlag>
                               frame syncMode FreeRunning
fmp <extOut val(0-3)> <frameLength> <controlFlag> <channel> <syncmargin>
                               frame syncMode Packet
fms <extIn val(1-3)> <extOut val(0-3)> <frameLength> <controlFlag> <syncmargin>
                               frame syncMode External Signal
fmb <bus val(0-2)> <extOut val(0-3)> <frameLength> <controlFlag> <syncmargin>
                               frame syncMode other bus

--- Generic commands ---

sts                             get STOF Timestamp
qu                              quit demo

```

Opening and closing a bus

Open Bus

The command

```
ob <device num>
```

can be used to open one of the buses found and listed at the top of the console output.

Close Bus

The command

```
cb
```

can be used to close the currently opened bus.

Single Message Mode

Commands below will create one context in single message mode if the context does not exist. This mode allows to transmit a single message at a time at a specific offset time.

Transmit a single message

The command

```
sms
```

can be used to transmit a single message on channel 15 at frame offset time 2500us and at speed S200.

Transmit a single message with split buffers

The command

```
spms
```

can be used to transmit a single message on channel 16 at frame offset time 5500 and at speed S100. The major difference compared to the "sms" command above is that the message originates from multiple buffers and is automatically combined by the DMA engine of the Link Layer.

Streaming Messages Mode

Commands below will create one context in streaming messages mode if the context does not exist. This mode can be used to manage a queue of packets (stream) that can be prepared ahead of time. This is very useful to for example stream messages from an input file at specific offset times.

Read FireSpy regeneration file as stream

The command

```
rg <filename>
```

can be used to enqueue packets from a file in [Regeneration format](#) to the streaming mode context. This format can be generated by using the export functionality of the FireSpy Recorder.

Stream message list

The command

```
sml
```

can be used to enqueue 4 messages to the streaming mode context to be transmitted.

Stream message list with split buffers

The command

```
spml
```

can be used to enqueue 4 messages to the streaming mode context to be transmitted. The major difference compared to the "sml" command above is that each message originates from several memory buffers and is automatically combined by the DMA engine of the Link Layer.

Start stream

The command

```
gst
```

can be used to start the context opened in streaming mode. The context will then transmit any packets already in the queue at their requested offset times.

Stop stream

The command

```
hst
```

can be used to stop the context opened in streaming mode.

Clear stream

The command

```
cst
```

can be used to clear any messages currently in the transmission queue of the context opened in streaming mode.

Display stream context status

The command

```
stst
```

can be used to display status of the context opened in streaming mode.

Repeating Message Mode

Commands below will create one context in repeating message mode if the context does not exist. This mode can be used to setup packets that need to be repeatedly transmitted each new frame at the requested frame offset time. Contents will remain the same in this example.

Create message

The command

```
cms <msgsize> <ch> <offset>
```

can be used to setup a message object that can be controlled individually. It will be setup with <msgsize> number of bytes to be transmitted on channel <ch> at frame offset time <offset>. The message will be created in stopped state. A message ID is returned that can be used by the other functions to control the message.

Start message

The command

```
gms <msgnum>
```

can be used to start transmission of a message already created by specifying its <msgnum>. The message will be transmitted once every frame at the specified frame offset time.

Stop message

The command

```
hms <msgnum>
```

can be used to stop transmission of a message already created and currently in transmission mode.

Close message

The command

```
wms <msgnum>
```

can be used to release any resources associated with an already created message by specifying its <msgnum>.

Change message options

The command

```
oms <msgnum> [c val(0-63)] [s val(0-2)] [v val(0-1)]
```

can be used to change options of an already created message by specifying its <msgnum>.

c: Specify channel. Valid range of values is from 0 to 63.

s: Specify transmission speed. Please refer to [Speed Codes](#) for definitions of speed codes.

v: Specify Auto VPC mode. 0: Disable, 1: Enable

Display message status

The command

```
stms
```

can be used to display status of a repeating message.

List messages

The command

```
lms
```

can be used to get a list of messages that have been created so far.

STOF Messages Mode

Commands below will create one context in STOF message mode if the context does not exist. This mode can be used to control the transmission of STOF packets from the local node.

Start STOF transmission

The command

```
gf
```

can be used to start transmission of STOF packets by the local node.

Stop STOF transmission

The command

```
hf
```

can be used to make the local node stop transmitting STOF packets.

Write STOF message contents

The command

```
wf
```

can be used to set the contents of the STOF message. At startup the contents are undefined so it is recommended to call this function before starting STOF message transmission.

Set STOF message options

The command

```
of [c val(0-63)] [s val(0-2)] [v val(0-1)]
```

can be used to set the options of the STOF message.

c: Specify channel. Valid range of values is from 0 to 63.

s: Specify transmission speed. Please refer to [Speed Codes](#) for definitions of speed codes.

v: Specify Auto VPC mode. 0: Disable, 1: Enable

Frame Synchronisation Options

FireStack is very flexible in the way it generates the start of frame signal locally. Start of frame can be based on an internal clock, can be synchronized to STOF packets on the bus and it can be synchronized to an external signal.

Freerunning mode

The command

```
fmf <extOut> <frameLength> <controlFlag>
```

can be used to disable frame synchronization and use an internal clock to time the frames of duration <frameLength>us and with [controlFlags](#) set to <controlFlags>. On FireTrac V3 devices only: <extOut> Can

be used to enable one of the External Sync Outpus: 0:None, 1:A, 2:B, 3:C

Synchronize to STOF packets

The command

```
fmp <extOut> <frameLength> <controlFlag> <channel> <syncmargin>
```

can be used to enable frame synchronization based on STOF packets received. You have to specify the expected frame length <frameLength>, <channel> the sync packets are to be expected and the <syncMargin> as well as <controlFlag>. Please refer to [this](#) section for a description of the options. On FireTrac V3 devices only: <extOut> Can be used to enable one of the External Sync Outpus: 0:None, 1:A, 2:B, 3:C

Synchronize to external signal

The command

```
fms <extIn> <extOut> <frameLength> <controlFlag> <syncmargin>
```

can be used to enable frame synchronization based on an external signal received on one of the external sync pins, 0:A, 1:B or 2:C. You have to specify the expected frame length <frameLength> and the <syncMargin> as well as <controlFlag>. Please refer to [this](#) section for a description of the options. On FireTrac V3 devices only: <extOut> Can be used to enable one of the External Sync Outpus: 0:None, 1:A, 2:B, 3:C

Synchronize internally to one of the other buses

The command

```
fmb <bus> <extOut> <frameLength> <controlFlag> <syncmargin>
```

can be used to enable frame synchronization to one of the other buses of the FireTrac device. Specify <bus> 0 for Bus A, 1 for bus B and 2 for bus C. You have to specify the expected frame length <frameLength> and the <syncMargin> as well as <controlFlag>. Please refer to [this](#) section for a description of the options. On FireTrac V3 devices only: <extOut> Can be used to enable one of the External Sync Outpus: 0:None, 1:A, 2:B, 3:C

Misc

Display STOF Time Stamp

The command

```
sts
```

can be used to display STOF time stamp information such as frame number, seconds, and sub-seconds data.

Other commands

Use '?' (help) command to display the menu. Use the 'qu' (quit) command to terminate the demo application.

12.8. AS5643 Cockpit Demo

12.8.1. AS5643 Control Computer Example

Introduction

This document describes the DapTechnology AS5643 Control Computer example. It implements the duties of an AS5643 Control Computer as described in Aerospace Standard AS5643 Rev. B, and serves to demonstrate how to implement an AS5643 CC using the DapTechnology FireStack.

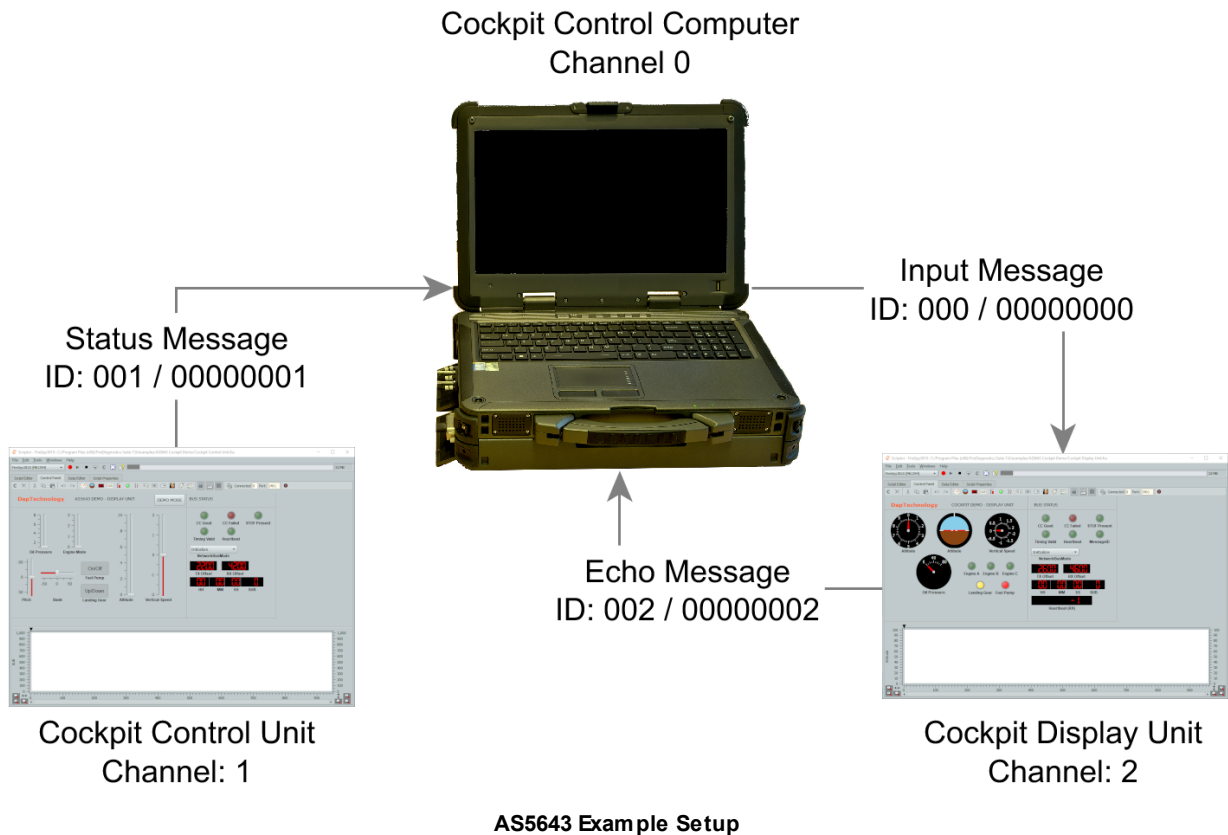
Getting Started

To get started with the AS5643 Control Computer example, you need at least a FireTrac card. A FireSpy analyzer is highly recommended.

The AS5643 demonstration setup includes:

- The Control Computer
- A Control Unit
- A Display Unit

These units are connected with IEEE1394 FireWire. The Control Unit has some controls to set parameters like altitude, pitch and more. It sends these to the Control Computer which processes the data and sends data to the Display Unit which visualizes this information using dials and lights.



The FireDiagnostics Suite includes a Control Unit and Display Unit example. In the absence of a FireSpy, the Control Computer example can simulate a missing Control Node and/or Display Node. It can transmit the messages the missing nodes would have sent, and emulate their behavior. The Control Computer, designed to be used in embedded systems, doesn't do visualization, though. Simulation is optional: it is controlled with `CC_SIMULATE_UNITS`, defined in `ControlComputer.h`.

Usage

When the AS5643 CC is started it will print some messages during initialization, and present a menu like this:

```

Initialize FireStack...
Open AS5643 Bus...
Configure AS5643 Bus...
Initialize CC Receiver...
Initialize Display Unit...
Initialize Control Unit...
New CC Network Bus Mode: Initializing
Response from Control Unit.
Response from Display Unit.
New CC Network Bus Mode: Normal
Control Unit comes online
Display Unit comes online

-----
-- AS5643 Control Computer Example Menu
-----
?          help (this menu)
q          quit
s          status of the AS5643 network
p          print current value of various sensors
m          monitor value one sensor
>

```

Option 's' will print out the current status of the CC, the AS5643 Bus and the Remote Nodes.

Option 'p' will show the sensor data as defined in our Slash Sheet: Vertical Speed, Altitude, Bank, Pitch, ... or show a message if no valid message data has been received yet.

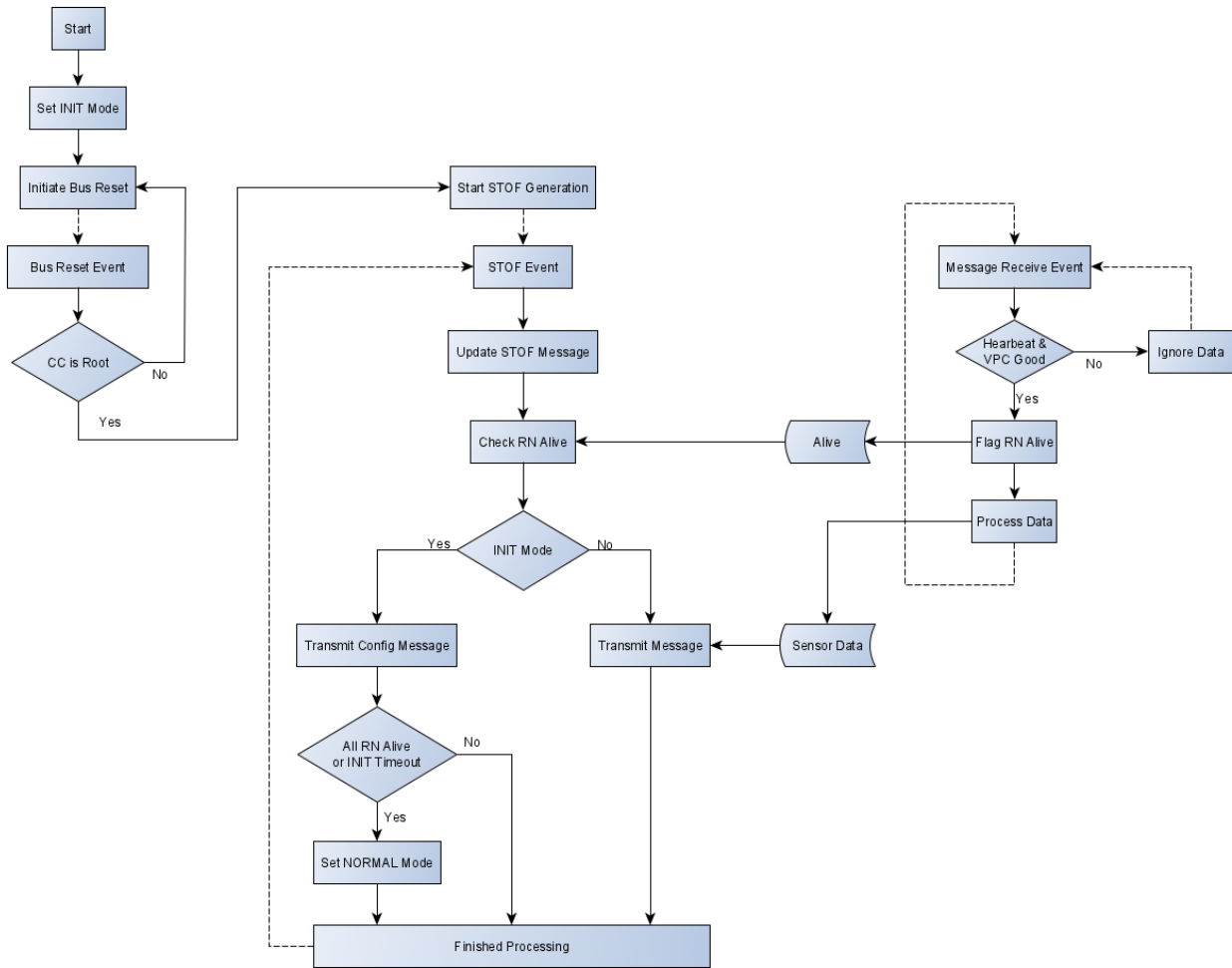
Option 'm' allows you to select one sensor and it will continuously refresh the current value of that sensor.

Application Design

In order to perform the duties of a CC as defined by the AS5643 specifications, the CC needs to perform a number of steps:

- It needs to become the root node.
- It needs to listen for messages transmitted by the Remote Nodes as defined in the Slash Sheet document.
- Once it is the root node, it starts transmitting STOF packets.
- The CC transmits configuration packets to all Remote Nodes.
- It processes the data from packets received from Remote Nodes.
- When all Remote Nodes have sent valid responses the initialization phase is finished and the CC transitions to normal mode.
- The CC continues to receive messages from Remote Nodes, process the data and transmit data to Remote Nodes.
- The CC continues to monitor the bus configuration to make sure it can still perform its duties and the status of the Remote Nodes.

The diagram below visualizes the events, states and actions of the CC:



AS5643 CC State Diagram

The implementation heavily uses callbacks to process events. We shall now look at this in more detail, and explain the advantages of using the DapTechnology FireStack.

Becoming the root node

As defined in AS5643 sec. 3.3.1, the CC needs to be the root node. The CC does this by transmitting a PHY Config Packet to set the root-holdoff bit in each PHY's Base Register 1, to ensure it becomes the root node. It then issues a bus reset.

FireStack allows us to register a callback for the Bus Reset event. In this callback we check that we are (still) the root node. When the CC first becomes the root node, this event is used as a trigger to start generating STOF packets, the next step in the initialization sequence.

STOF Generation

The CC transmits a STOF packet at the start of each frame. The format of the STOF packet is defined in AS5643 sec. 3.3.1.1.1. The DapTechnology FireStack makes STOF management easy with specialized functions for this task. Again, we register a callback to be called at the beginning of each frame. We use this to update the STOF message transmitted at the beginning of the next frame.

Bus Configuration

During initialization the CC transmits configuration messages to the Remote Nodes. This starts when the CC starts STOF generation. The configuration messages can be used to update the STOF Transmit and Receive Offsets of the Remote Nodes.

When a Remote Node has seen three consecutive frames with correct STOF timing it starts to send packets back to the CC.

The whole process can be seen in this in this view from the FireSpy Recorder:

Number	Packet/event	Node	Size	Source	Destination	Time (incr.)
0	R (bus reset 1)	a				
1	PhySelfID0	a	8			167.552 us
2	PhySelfID0	a	8			1.790 us
3	PhySelfID0	a	8			2.126 us
4	PhyConf	a	8			14.428487 ms
5	R (bus reset 2)	a				44.230 us
6	PhySelfID0	a	8			167.511 us
7	PhySelfID0	a	8			1.790 us
8	PhySelfID0	a	8			2.146 us
9	Streaming	a	52		ch 31	24.528127 ms
10	Streaming	a	84		ch 1	4.199931 ms
11	Streaming	a	92		ch 2	400.004 us
12	Streaming	a	52		ch 31	7.900889 ms
13	Streaming	a	84		ch 1	4.199961 ms
14	Streaming	a	92		ch 2	399.994 us
15	Streaming	a	52		ch 31	7.900889 ms
16	Streaming	a	84		ch 0	2.202189 ms
17	Streaming	a	92		ch 0	399.404 us
18	Streaming	a	92		ch 2	1.998342 ms
19	Streaming	a	52		ch 31	7.900869 ms
20	Streaming	a	84		ch 0	2.202016 ms
21	Streaming	a	92		ch 0	399.455 us
22	Streaming	a	92		ch 2	1.998484 ms
23	Streaming	a	52		ch 31	7.900889 ms
24	Streaming	a	84		ch 0	2.201833 ms
25	Streaming	a	92		ch 0	399.475 us
26	Streaming	a	92		ch 2	1.998627 ms

AS5643 Network Initialization Sequence

The first bus reset happens when the CC node opening the bus. Each of the three nodes (the CC and two Remote Nodes) sends a self-ID packet. The CC proceeds to become root by transmitting a PHY Configuration packet and issuing a bus reset.

The first two frames after the CC becomes root contain a STOF message (channel 31) and two configuration messages (channel 0 and 1). After the third STOF message, the Remote Nodes reply back to the CC on channel 0.

From there on, the CC stops sending (configuration) messages to the Control Node, and only sends messages to update the Display Node status.

Receiving AS5643 Messages using a Message Filter

FireTrac provides a filtering mechanism that filters incoming packets in hardware based on their channel number and/or message ID. We use this feature to subscribe to the messages of our Remote Nodes as defined in the Slash Sheet. A callback function will be called for every packet received that matches the message filter.

The validity of every message received is verified using the embedded Vertical Parity Check, described in AS5643 sec. 3.2.4.1.9.4. Only messages with correct VPC will be processed.

The CC keeps statistics of messages received from Remote Nodes. If no message has been received from a Remote Node for a number of frames (defined in the Slash Sheet), the node will be marked as offline.

Transmission of AS5643 Messages

The DapTechnology FireStack offers different transmission modes. For this example, we use the Repeating Messages Mode with callback. This mode fits the task at hand: transmitting messages *every frame* to the Remote Nodes. We just need to update the DMA buffers with the packet data.

The CC uses a double buffering for the DMA buffers with packet data. Shortly before transmitting the packet at its defined STOF offset, FireTrac transfers the packet data from the DMA memory to its transmit FIFOs. When done, the transmit complete callback is called and we swap the active buffer.

This leaves us the entire frame, minus the DMA Preload Time to safely manipulate the contents of the DMA buffers without the risk of transmitting partial or corrupted packets. Refer to the `FXMiiFrameTimingOptions` section in the FireStack API Reference Manual for a detailed description of the DMA Preload Time.

However, in this implementation we use the STOF callback to construct messages to be transmitted later in the frame. This works because our Slash Sheet defines STOF offsets sufficiently late in the frame. For very short STOF offsets, transmission data would have to be prepared before the start of frame, because otherwise packets would be transmitted in the next frame.

Disclaimer

This example serves to demonstrate the AS5643 features of the DapTechnology FireStack. DapTechnology wants stress that this is **NOT A TEMPLATE FOR AN ACTUAL FLIGHT COMPUTER**.

The example is licensed under these terms, which appear in every source file:

```
Copyright (c) 2019, 2020 Dap Technology B.V. All rights reserved.
```

```
Dap Technology B.V.
```

```
DAP TECHNOLOGY IS PROVIDING THIS DESIGN, CODE, OR INFORMATION "AS IS" AS  
A COURTESY TO YOU. BY PROVIDING THIS DESIGN, CODE, OR INFORMATION AS  
ONE POSSIBLE IMPLEMENTATION OF THIS FEATURE, APPLICATION OR STANDARD,  
DAP TECHNOLOGY IS MAKING NO REPRESENTATION THAT THIS IMPLEMENTATION IS  
FREE FROM ANY CLAIMS OF INFRINGEMENT, AND YOU ARE RESPONSIBLE FOR  
OBTAINING ANY RIGHTS YOU MAY REQUIRE FOR YOUR IMPLEMENTATION.  
DAP TECHNOLOGY EXPRESSLY DISCLAIMS ANY WARRANTY WHATSOEVER WITH RESPECT  
TO THE ADEQUACY OF THE IMPLEMENTATION, INCLUDING BUT NOT LIMITED TO ANY  
WARRANTIES OR REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE FROM  
CLAIMS OF INFRINGEMENT, IMPLIED WARRANTIES OF MERCHANTABILITY AND  
FITNESS FOR A PARTICULAR PURPOSE.
```

12.9. Extensions

12.9.1. External Timer

The External Timer example demonstrates some functions defined in section [External Timer](#).

When the demo application starts the console displays a welcome message and a menu. With help of this menu you can give simple commands to evaluate the functionality of the FireTrac External Timer API function set. The initial display should look like the following:

```

-----
-- Welcome to External Timer Demo!!!
-----
Time inputs found: 1
Just opening the first one.

Device Id: 3
PCI bus: 8
PCI device: 4
PCI function: 3
Sub-second resolution: 1000 nano seconds

-----
-- External Timer Demo Menu
-----
?                               help (this menu)
--
-- Configuration commands
--
sm <mode>                       set mode: 0 - free running, 1 - IRIG_B122, 2 - IRIG_B122
sy <year>                       set year
so <offset>                     set free running offset
--
-- Status commands
--
pt <0/1>                       enable print time each second. 0 - disable, 1 - enable
st                               get current status
--
-- Generic commands
--
qu                               quit demo
>

```

The external timer example controls the time input device as can be found on a FireTrac card. In order for this demo to show anything else but free running clock one needs to connect the IRIG input connector to an IRIG-B122 or IRIG-B122-1344 signal source.

Set Mode

The command

```
sm <mode>
```

can be used to select the time device mode. Valid modes are:

- 0 - Free running, an internal clock will be used and command "so" can be used to set its current time in seconds since 1970.
- 1 - Sync to an IRIG-B122 source currently connected to the IRIG input connector. Current year can be set using command "sy"
- 2 - Sync to an IRIG-B122-1344 source currently connected to the IRIG input connector

Set Year

The command

`sy <year>`

can be used in IRIG-B122 mode to set the <year> to use as current year.

Set free running offset

The command

`so <offset>`

can be used in Free Running mode to set the current time in seconds since 1-1-1970.

Print Time Each second

The command

`pt <0/1>`

can be used to print the current time each second. This does not work in free running mode. Set to 1 to enable, 0 to disable.

Get current status

The command

`st`

can be used to print the current time device status to the console.

Other commands

Use '?' (help) command to display the menu. Use the 'qu' (quit) command to terminate the demo application.

Chapter 13. FireStack Release Notes

FireStack version 2.1.10

Admintool

- Fixed a problem that prevented the user from changing the License Certificate location (19701)

Async Transactions

- OHCI Cards: Fixed a problem with responding to block reads from local Configuration ROM (19318)

Mil1394 General

- AS5643 Cockpit Demo: Control Computer Example (15236)

OSAL - Windows

- Support for the new FireAdapter3465bT (19301)

FireStack version 2.1.9

Firmware

- Integrate xFT4424bT version 6

OSAL - Linux

- Fixed permissions for users not in UNIX group 'fwupdater', which were causing licensing problems (19104)
- Support Linux kernel 5.4 (18889)

OSAL - Windows

- Fixed checking firmware version by fxCreateBusHandle function (19084)

xFT4424bT version 6

Top-Level

- Fixed a problem with sync output pins B, C and D (19018)

FireStack version 2.1.8

Mil1394 General

- Fixed a problem that prevented FX_MIL_SYNC_OUT_SIGNAL_D from working as expected (19017)

Mil1394 Transmission

- Fixed a problem that made it impossible to stop and then restart a streaming mode Mil1394 Transmission context (18951)

OSAL - Linux

- Support for SUSE SLES 15 (18903)

FireStack version 2.1.7

Firmware

- Integrate xFT3460bT_V1 version 21
- Integrate xFT3460bT_V2 version 21
- Integrate xFT3460bT_V3 version 13
- Integrate xFT4424bT version 5

xFT3460bT V1 version 21

Firmware

- Integrate FireLink Extended version v1_05_e

xFT3460bT V2 version 21

Firmware

- Integrate FireLink Extended version v1_05_e

xFT3460bT V3 version 13

Firmware

- Integrate FireLink Extended version v1_05_e

xFT4424bT version 5

Firmware

- Integrate FireLink Extended version v1_05_e

FireLink Extended version v1_05_e

Mil1394

- Fixed a problem that would cause the first word of a DMA descriptor block to become corrupted in case an internal FIFO transitions into a wait state when it is full. As a consequence the context may not issue a transmission completion interrupt or perform an incorrect branch operation. (18779)

FireStack version 2.1.6

Mil1394 General

- FireTrac4424bT: Fixed a problem with the following sync input modes: FX_MIL_SYNC_SIGNAL_D, FX_MIL_SYNC_BUS_0, FX_MIL_SYNC_BUS_1, FX_MIL_SYNC_BUS_2, FX_MIL_SYNC_BUS_3 and FX_MIL_SYNC_PACKET on any bus and FX_MIL_SYNC_SIGNAL on the 4th bus. (18730)

FireStack version 2.1.5

Admintool

- Admintool no longer requires administrator rights (18508)
- Fixed incorrect topology right after opening bus (18698)

Firmware

- Integrate xFT3460bT_V3 version 12
- Integrate xFT4424bT version 4

Low-Level API

- Improved error checking on DMA buffer usage (18696)

Manual

- Release notes now include release notes for firmware (18522)

OSAL - LabView (RT)

- Fixed asynchronous transmit functions with data block to correctly use a DMA buffer (18694)
- Improved error checking on DMA buffer usage (18695)
- Installer signed using new SHA256 certificate (18641)
- Support for FireTrac4x24bT on LabVIEW (RT) (17733)

OSAL - QNX

- Support for FireTrac4x24bT on QNX (18256)

OSAL - Windows

- Install correct VCRedist version (18580)
- Support for FireTrac4x24bT on Windows (17238)
- Windows drivers for Windows versions prior to 10 no longer signed using insecure SHA1 certificate (18684)

Time Input Device

- FireTrac4424bT: Support for IRIG-B002 (IEEE1344) TTL and RS422 (18494)

xFT4424bT version 4

IRIG-B

- Support for DCLS mode IRIG-B, both TTL and RS422 (17437)

xFT3460bT_V3 version 12

Firmware

- Support for FireTrac3460bT1 delivered after 01/01/2019 (18519)

FireStack version 2.1.4

Async Transactions

- Fixed a problem with responding to block reads from local Configuration ROM (18340)

Firmware

- Integrate xFT4424bT version 3

LAL - OHCI

- Add data payload byteswap option to FireStack 2.1 release branch (18484)

OSAL - Linux

- Fixed memory mapped register I/O on FireTrac3460bT V1,V2 on 32bit platforms (18331)

OSAL - VxWorks

- Support the new FireTrac4424bT on VxWorks (18323)

xFT4424bT version 3

PCIe Endpoint

- Fixed the PCI_CFG_HCI_CTL register for big-endian systems (18324)

FireStack version 2.1.3

Firmware

- Integrate xFT4424bT version 1

Low-Level API

- fxAsyRcWWaitSingleRequest now returns the correct size for ReadBlockRequest (17552)

OSAL - Linux

- Firetrac examples no longer fail when the terminal is set for unicode input (17113)
- Firetrac firmware updates no longer require root privileges. Users in the group fwupdater can run the fwupdater utility and update the firmware. The group fwupdater is created when the package is installed, but no users are added to this group. (18051)
- Now supports Linux kernel versions up to 5.1, with (optionally) the Real-Time Linux extensions (17843)
- Support for the new FireTrac4424bT on Linux (17239)

xFT4424bT version 1

Firmware

- Integrate FireLink Extended version v1_05_d

Top-Level

- Initial firmware release for FireTrac4424bT (17327)

FireLink Extended version v1_05_d

Mil1394

- Support for 4 Sync Inputs/Outputs (was 3) (17791)