FireTrac Advanced 1394 I/O Solution

Operation and API Reference Manual

Doc #: DT-PRO120MAN218E

Version 2.1.8

Project: firetrac

Operating System: win32

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 FireTrac Personality Profiles 28
2.2 Windows 30
Software Installation 30
Manual driver (un)installation 33
Administration Tool 35
Updating Firmware 36
Registering Licenses 38
Driver parameter settings 40
Notes on Borland Builder 41

Chapter 3. Definition of Terms 43

Chapter 4. Document Conventions 44

Chapter 5. Parameter Naming Conventions 45

Chapter 6. API Revision History 46

6.1 Changes in 2.1.x series 47
6.2 Changes in 2.0.x series 48
6.3 Changes in 1.0.x and 0.98.x series 49

Chapter 7. General Structures and Definitions 50

7.1 Type Definitions 50
Basic Types 50
Special Types 50
7.2 Structure Definitions 51
FXInt64 51
FXUint64 51

FXAddress64 51
FXChannelMask 51
FXSetting 51
7.3 Constants 53
Speed Codes 53
Transaction Types 53
Chapter 8. Administrative Functions 54
8.1 Initialization 55
Functions 55
fxInitialize 55
fxTerminate 55
8.2 Bus Initialization 56
Functions 56
fxGetNumberOfBuses 56
fxGetBusInfoList 56
fxCreateBusHandle 56
fxCloseBusHandle 57
fxGetEUI64 58
Structures 58
FXBusInfo 58
Settings 59
Features 59
Demo Mode 60
Byte Order 60
8.3 Memory Management 61
Functions 61
fxMemAlloc 61
fxMemFree 61
8.4 General 63
Functions 63
fxGetLibraryVersion 63
Structures 63
FXVersionInfo 63
8.5 Error Handling 64
Functions 64
fxGetErrorMessage 64
fxSetErrorCallback 64
fxGetErrorStatus 65
Constants 65
Error Codes 65
Type Definitions 65
FXErrorCallback 65
Chapter 9. AS5643 Protocol API Reference 66
9.1 AS5643 Frame Timing 67
Functions 67
fxMilSetFrameTimingOptions 67
fxMilSetStofCallback 68
fxMilGetFrameOffsetTime 68
fxMilGetStofTimestamp 69
Type Definitions 69
FXMilStofCallback 69
Structures 70

FXMilFrameTimingOptions 70
FXTimeStamp 71
Constants 71
Error Codes 71
Frame Synchronization Input Modes 72
Frame Synchronization Output Modes 72
Frame Control Flags 73
9.2 AS5643 Reception 74
Settings 74
Resource Usage 74
Functions 74
Channel Selections 74
fxMilRcvEnableChannels 74
fxMilRcvDisableChannels 75
fxMilRcvGetEnabledChannels 75
Message Filters 76
fxMilRcvAddFilterItem 76
fxMilRcvRemoveFilterItem 77
fxMilRcvClearMessageFilter 77
fxMilRcvGetNumFilterItems 77
fxMilRcvGetFilterItemList 78
Buffer Control 78
fxMilRcvAddBuffer 80
fxMilRcvLinkBuffers 81
fxMilRcvRemoveBuffer 82
fxMilRcvBufferStatus 82
Context Control 83
fxMilRcvCreateContextHandle 83
fxMilRcvCloseContextHandle 84
fxMilRcvSetContextOptions 84
fxMilRcvStartContext 85
fxMilRcvStopContext 85
fxMilRcvContextStatus 85
Type Definitions 86
FXMilRcvContextHandle 86
FXMilRcvCallback 86
Structures 86
FXMilRcvFilterItem 86
FXMilRcvBufferOptions 87
FXMilRcvBufferStatus 87
FXMilRcvEventOptions 88
FXMilRcvContextOptions 88
FXMilRcvContextStatus 89
Constants 89
Error Codes 89
Buffer Status Extended Staus Bits 89
Data Formats 90
Receive Packet Format 90
9.3 AS5643 Transmission 91
Settings 91
Resource Usage 91
Functions 91
Context Management 91
fxMilTrmCreateContextHandle 91
fxMilTrmCreateContextHandleExt 92
fxMilTrmCloseContextHandle 93
Single Message Mode 93

fxMilTrmMessage ... 93
fxMilTrmSplitMessage ... 94
Streaming Messages Mode ... 95
fxMilTrmStrmWriteImmediate ... 95
fxMilTrmStrmWriteMessageList ... 96
fxMilTrmStrmWriteSplitMessageList ... 97
fxMilTrmStrmStart ... 98
fxMilTrmStrmStop ... 99
fxMilTrmStrmClear ... 99
fxMilTrmStrmGetStatus ... 100
Repeating Messages Mode ... 100
fxMilTrmCreateMessageHandle ... 101
fxMilTrmCreateMessageHandleExt ... 101
fxMilTrmCloseMessageHandle ... 102
fxMilTrmSetMessageData ... 103
fxMilTrmSetMessageSplitData ... 103
fxMilTrmSetMessageOptions ... 104
fxMilTrmStartMessage ... 105
fxMilTrmStopMessage ... 105
fxMilTrmGetMessageStatus ... 105
STOF Message Mode ... 106
fxMilTrmSetStofMessageOptions ... 106
fxMilTrmWriteStofMessage ... 106
fxMilTrmStartStofMessage ... 107
fxMilTrmStopStofMessage ... 107
Type Definitions ... 108
FXMilTrmContextHandle ... 108
FXMilTrmMessageHandle ... 108
FXMilTrmCallback ... 108
Structures ... 108
FXMilTrmContextOption ... 108
FXMilTrmMessageOption ... 109
FXBuffer ... 109
FXMilTrmMessage ... 109
FXMilTrmSplitMessage ... 110
FXMilTrmStrmStatus ... 110
FXMilTrmMessageStatus ... 110
FXMilStofMessage ... 111
Constants ... 112
Error Codes ... 112
Context Options ... 112
Context Modes ... 113
Jitter Ranges ... 113
Jitter Directions ... 114
Message Options ... 114
Jitter Modes ... 115
Auto VPC Modes ... 115
User Callback Event Code Bits ... 115
Data Formats ... 115
AS5643 Regeneration Format ... 115

Chapter 10. 1394 API Reference 117

10.1 Serial Bus Management ... 117
Settings ... 117
SBM Capabilities ... 117
Functions ... 117
fxSetBusResetCallback ... 117

fxGetBusGeneration 118
fxGetNumberOfNodesOnBus 118
fxGetLocalNodeId 119
fxGetMaxSpeedToNode 119
Type Definitions 120
FXBusResetCallback 120
10.2 Inbound Transactions 121
Functions 121
Memory Mapping Functions 121
fxMapLocalMemory 121
fxMapRequestHandler 122
fxClearMemoryMapping 123
Local Memory Access Functions 123
fxReadLocalMemory 123
fxWriteLocalMemory 124
fxLockLocalMemory 124
Type Definitions 125
FXRequestHandlerCallback 125
FXRequestNotificationCallback 126
Structures 126
FXTransactionData 126
FXMappingOptions 127
Constants 127
Error Codes 127
Response Codes 127
10.3 Outbound Transactions 129
Functions 129
fxReadTransaction 129
fxWriteTransaction 131
fxLockTransaction 132
fxClearTransaction 133
fxClearAllTransactions 133
fxGetTransactionStatus 133
fxGetNumTransactions 134
fxGetTransactionList 134
Type Definitions 135
FXTransactionCompleteCallback 135
Structures 135
FXTransactionOptions 135
FXTransactionInfo 136
Constants 136
Error Codes 136
Transaction Status 137
10.4 Low-Level 1394 138
Settings 138
Resource Usage 138
Functions 138
Asynchronous Packet Reception Functions 138
fxAsyRcvWaitSingleRequest 138
fxAsyRcvWaitSingleResponse 139
fxAsyRcvSetPacketCallback 139
Single Packet Transmission Functions 140
fxAsyTrmWriteQuadletRequest 140
fxAsyTrmWriteBlockRequest 141
fxAsyTrmWriteResponse 141
fxAsyTrmReadQuadletRequest 142
fxAsyTrmReadBlockRequest 143

fxAsyTrmReadQuadletResponse ... 143
fxAsyTrmReadBlockResponse ... 144
fxAsyTrmLockRequest ... 145
fxAsyTrmStream ... 146
fxAsyTrmLockResponse ... 146
PHY Packets and registers ... 147
fxReadLocalPhyBaseReg ... 147
fxReadLocalPhyPageReg ... 148
fxWriteLocalPhyBaseReg ... 148
fxWriteLocalPhyPageReg ... 149
fxReadRemotePhyPageReg ... 149
fxReadRemotePhyBaseReg ... 150
fxPhyRemoteCommand ... 150
fxPhySetForceRoot ... 151
fxPhySetGapCount ... 152
fxPingRemoteNode ... 152
fxPhyPacketSetRcvCallback ... 153
fxPhyPacketTrmRaw ... 153
Topology Functions ... 154
fxGetSelfIdData ... 154
fxIssueBusReset ... 154
Type Definitions ... 155
FXAsyRcvPacketCallback ... 155
FXPhyPacketRcvCallback ... 155
Structures ... 156
FXAsyRcvPacket ... 156
Constants ... 156
Error Codes ... 156
PHY Remote Commands ... 156
PHY Confirmation Flags ... 157

Chapter 11. Time Input Device API Reference 158

11.1 Functions ... 158
fxGetNumberOfTimeInputs ... 158
fxGetTimeInputInfoList ... 158
fxCreateTimeInputHandle ... 159
fxCloseTimeInputHandle ... 159
fxSetTimeInputMode ... 160
fxSetTimeInputCurrentYear ... 160
fxSetTimeInputFreeRunningOffset ... 161
fxGetTimeInputStatus ... 161
fxSetTimeInputStatusCallback ... 161
fxSetTimeInputSecondCallback ... 162
11.2 Type Definitions ... 163
FXTimeInputHandle ... 163
FXTimeInputStatusCallback ... 163
FXTimeInputSecondCallback ... 163
11.3 Structures ... 164
FXTimeInputInfo ... 164
FXTimeInputStatus ... 165
11.4 Constants ... 166
Time Input Mode ... 166
Status Codes ... 167
Error Codes ... 167

Chapter 12. Examples 168

12.1 Inbound Transactions 169
Inbound Transaction Monitor 169
12.2 Outbound Transactions 171
Outbound Transaction Demo 171
12.3 Low-Level 1394 173
Low-level Demo 173
12.4 AS5643 Reception 175
Mil1394 Data Logger 175
Mil1394 Receive Demo 177
12.5 AS5643 Transmission 181
Mil1394Transmit Demo 181
12.6 Extensions 188
External Timer 188
Chapter 13. FireStack Release Notes 190

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 IEE1394 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.

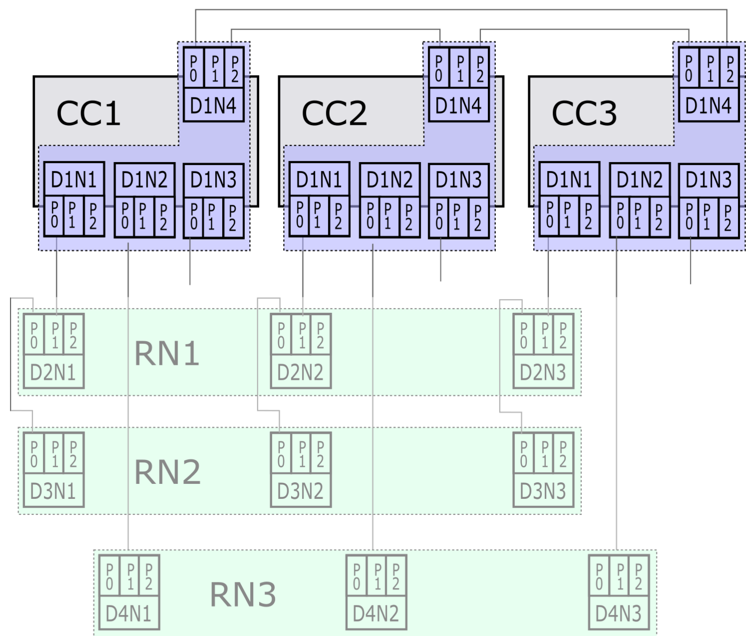
1.1. Network Simulation

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).

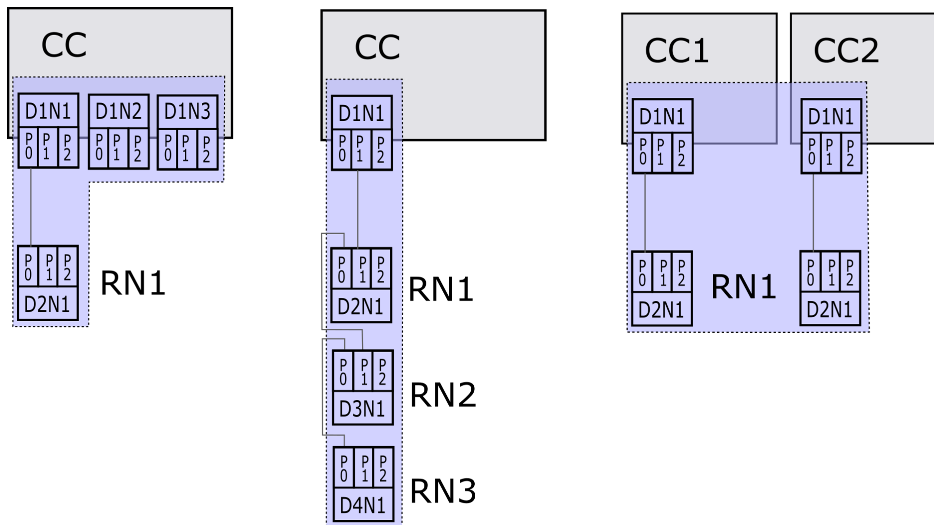
4-node FireTrac®:

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.



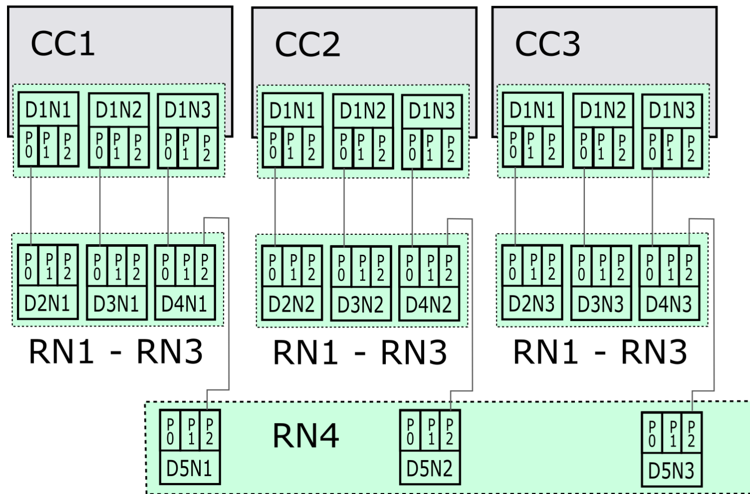
However, the above is only an implementation variant. A 4-node FireTrac® can also be used to simulate single branch bus behavior (left and middle) or mixed CC/RN situation in double redundancy (right).



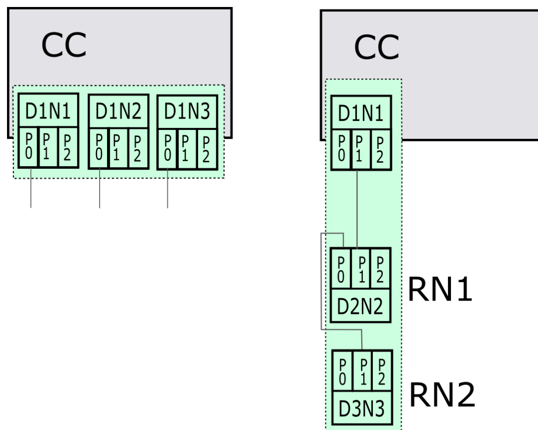
Copyright(C) 2008-2020, DapTechnology, Version 2/26/2020

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).



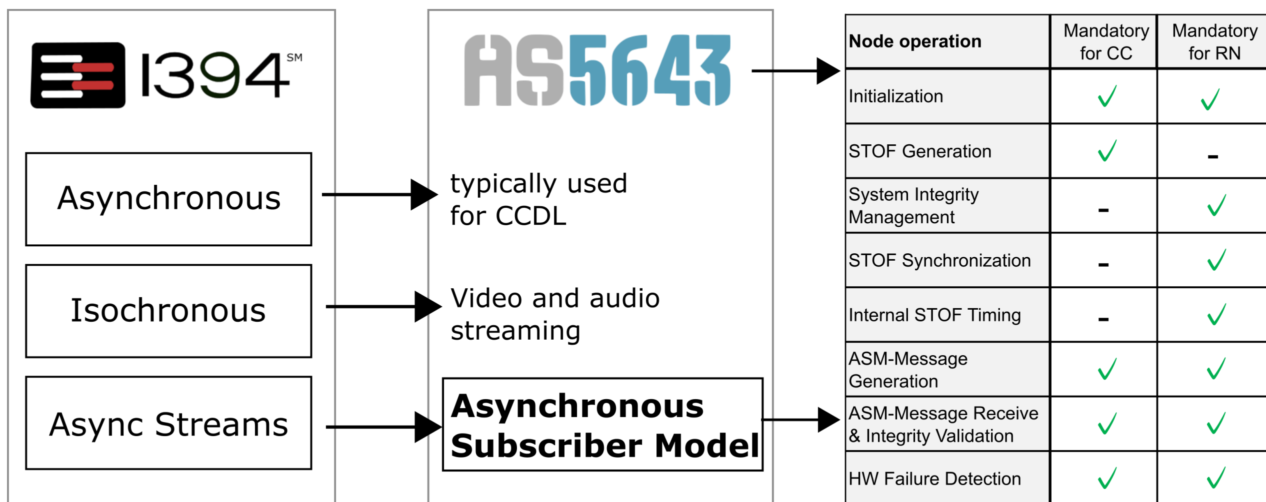
Other variants of how a single 3-node FireTrac® card can be used are depicted below. The left shows a CC devices simulating the triple bus interface (3 CC) whereas the right pictures demonstrates a possible single-branch usages model (CC + 2 RNs).



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 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.

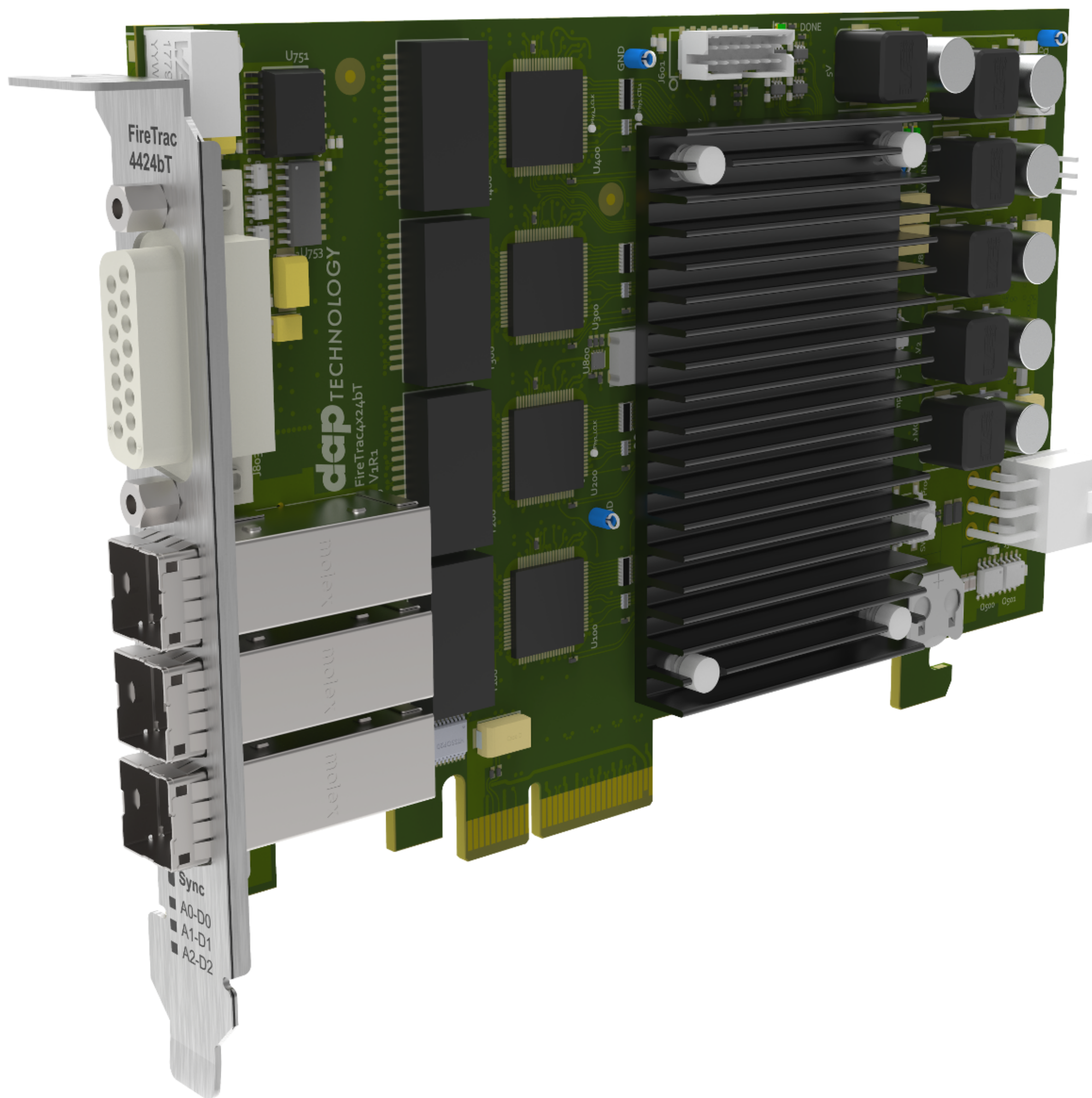


Node operation	Mandatory for CC	Mandatory for RN
Initialization	√	√
STOF Generation	√	-
System Integrity Management	-	√
STOF Synchronization	-	√
Internal STOF Timing	-	√
ASM-Message Generation	√	√
ASM-Message Receive & Integrity Validation	√	√
HW Failure Detection	√	√

 Copyright(C) 2008-2020, DapTechnology, Version 2/26/2020

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

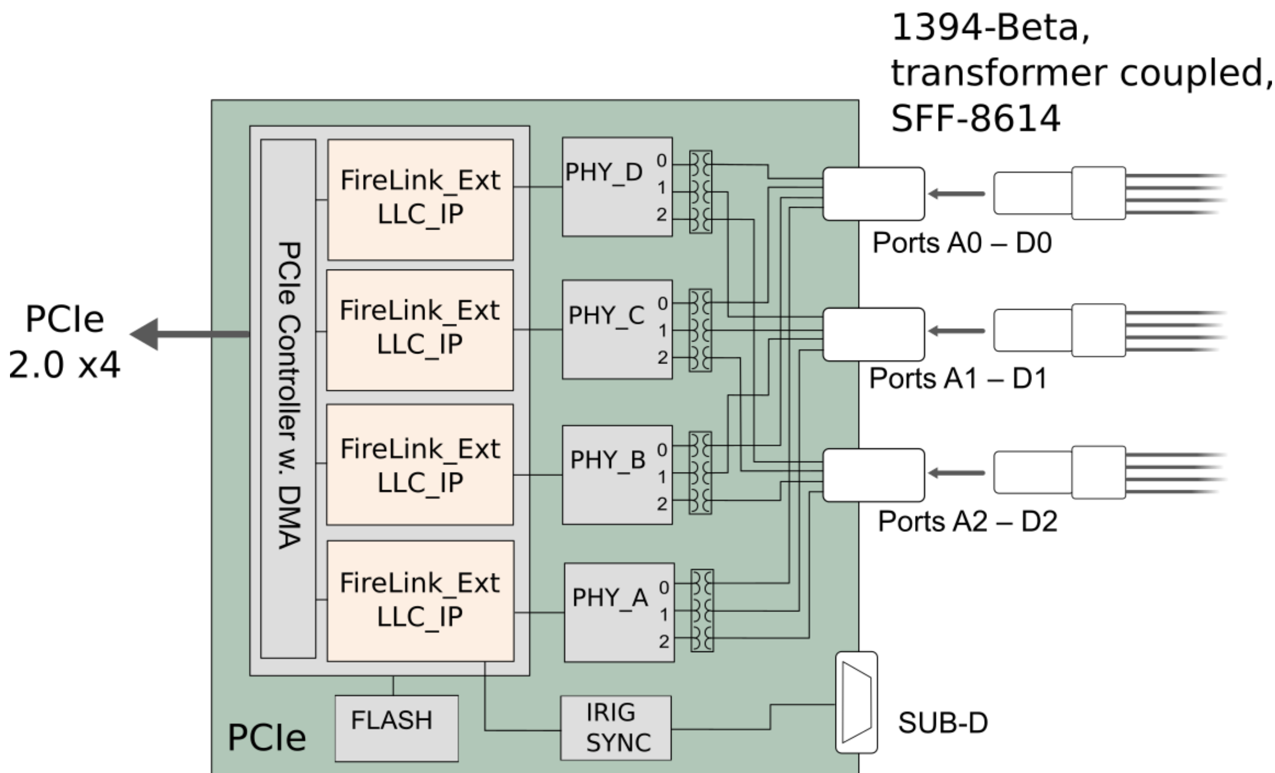
Depending on the FireTrac4424bT board version you may need to install a minimal software version. Please have a look at the following table to look up which software version is required based on the serial number of your card.

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-

Copyright(C) 2008-2020, DapTechnology, Version 2/26/2020

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

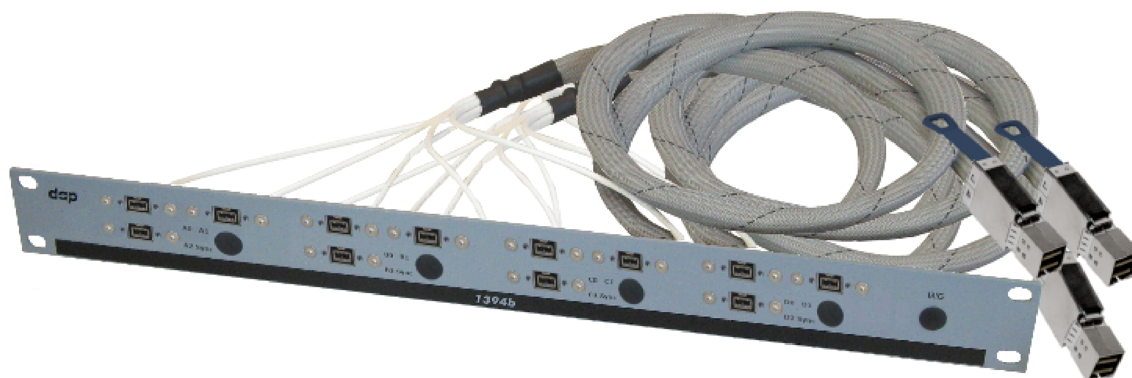
Copyright(C) 2008-2020, DapTechnology, Version 2/26/2020

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

support@daptechnology.com.

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

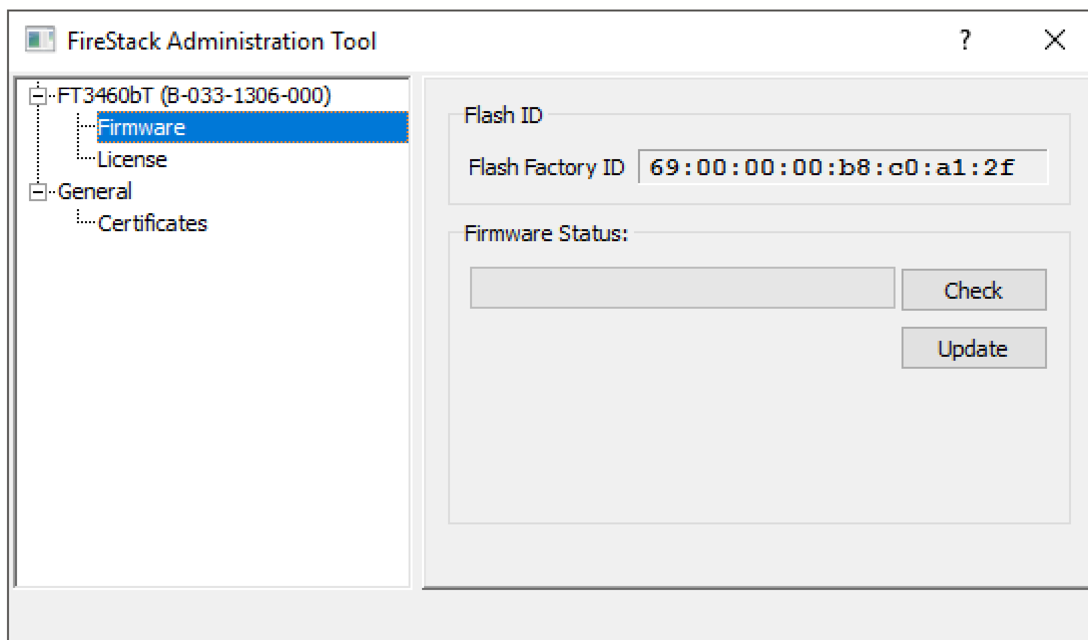
SW602
Location: Back side of the card top-left quadrant
Positions:

- 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.

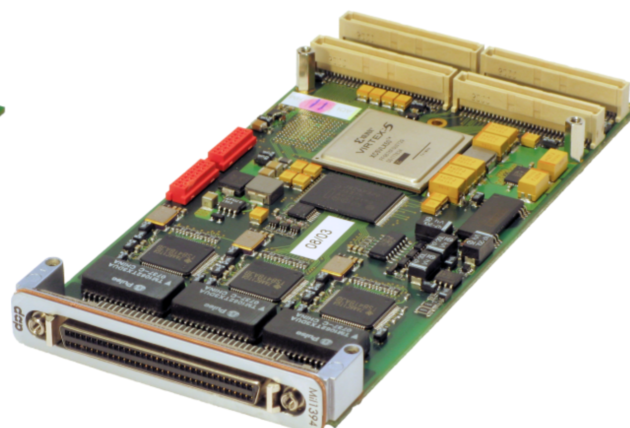
Copyright(C) 2008-2020, DapTechnology, Version 2/26/2020

1.4. FireTrac3460bT

The FireTrac3460bT implementes 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

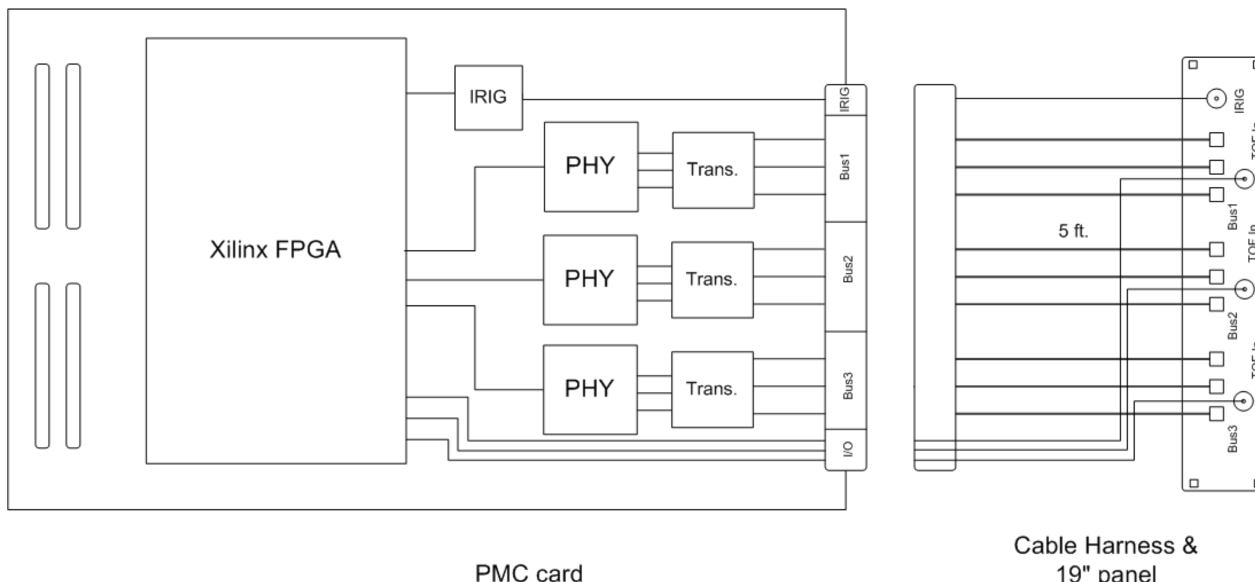
Important Version information

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

Copyright(C) 2008-2020, DapTechnology, Version 2/26/2020

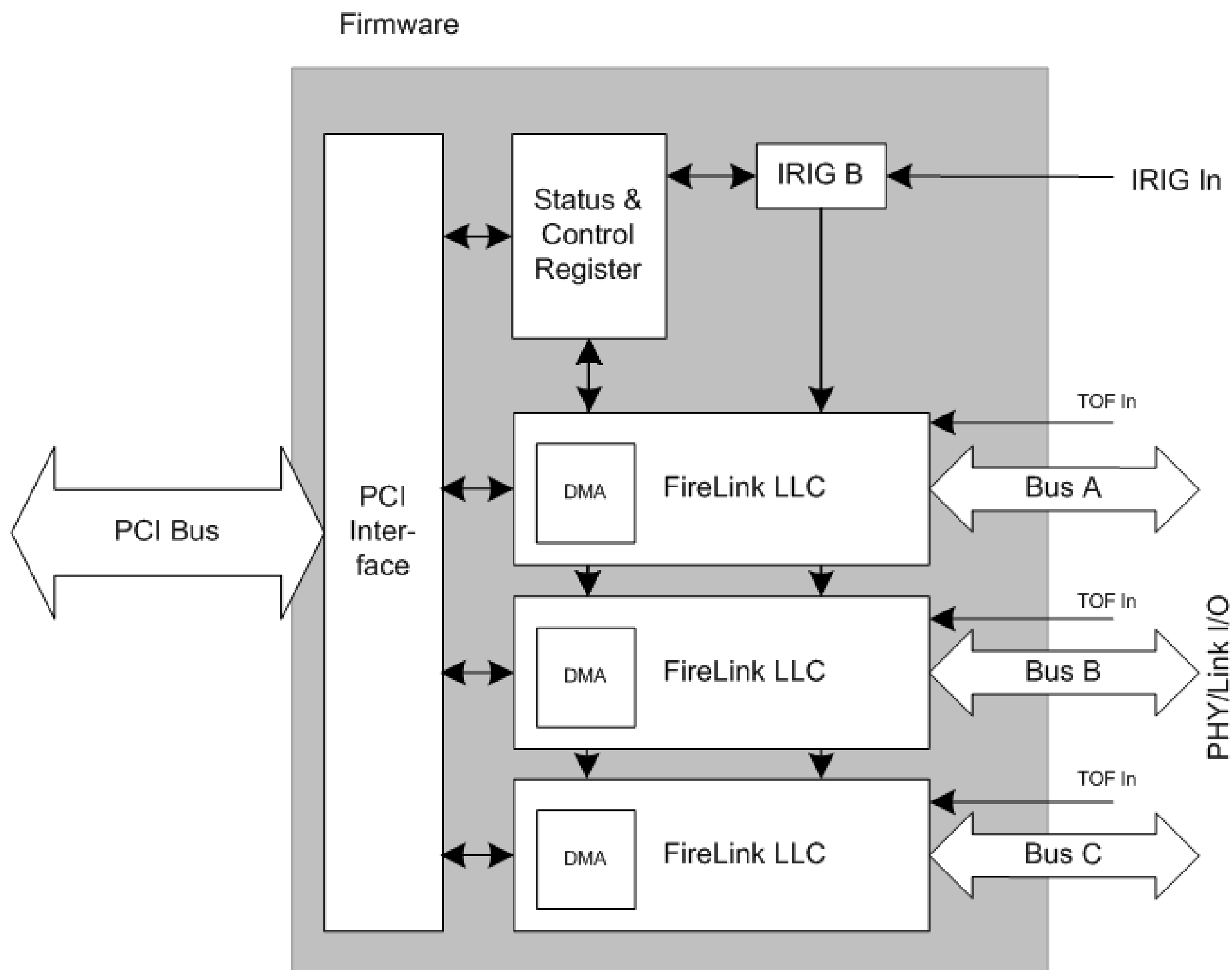
information after that.
An on-board FLASH chip is available that holds firmware to be loaded into the FPGA on system startup time. This memory is only used to store new firmware updates and license certificates.
All memory can be sanitized by the end-user except card-specific configuration data written 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

Copyright(C) 2008-2020, DapTechnology, Version 2/26/2020

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

Tabel Limiting values

Symbol	Parameter	Min	Max	Unit
Vin	Input voltage	-0.5	+12	V
Io	Output current		±50	mA

Tabel Recommended operating conditions

Symbol	Parameter	Min	Max	Unit
Vin	Input voltage	0	5	V
Vout	Output voltage	0	5	V

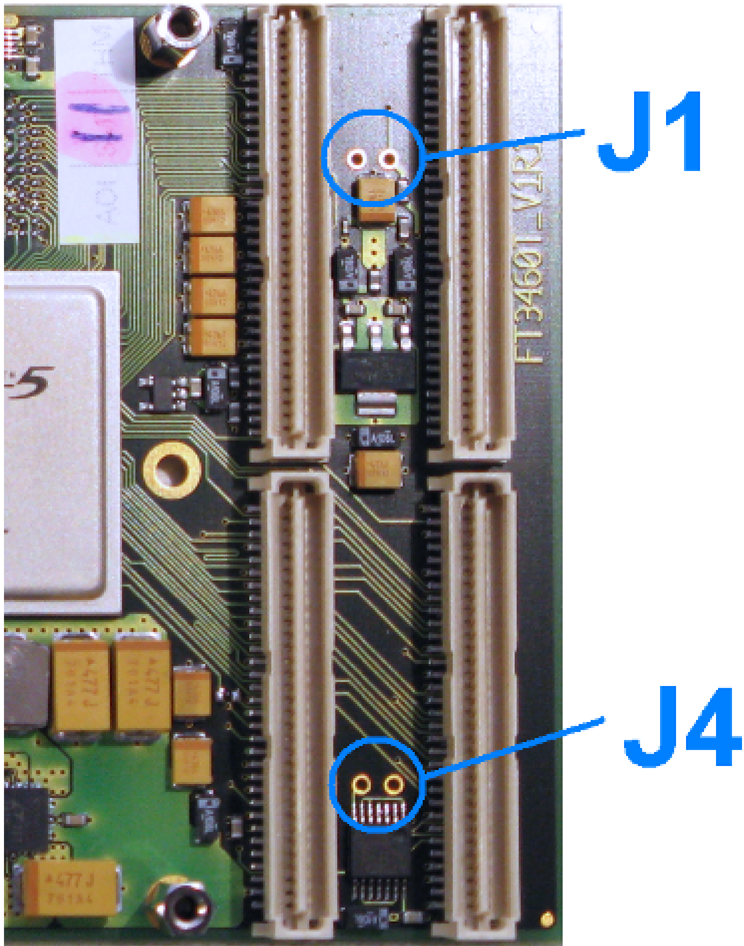
Tabel Static characteristics

		Conditions	Min	Max	Unit
VIH	HIGH-level input voltage		2.0		V
VIL	Low-level input voltage			0.8	V
VOH	HIGH-level output voltage	Io = 50µA	4.9		
VIL	LOW-level output voltage	Io = 50µA		0.1	

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

1.4.5. Jumpers

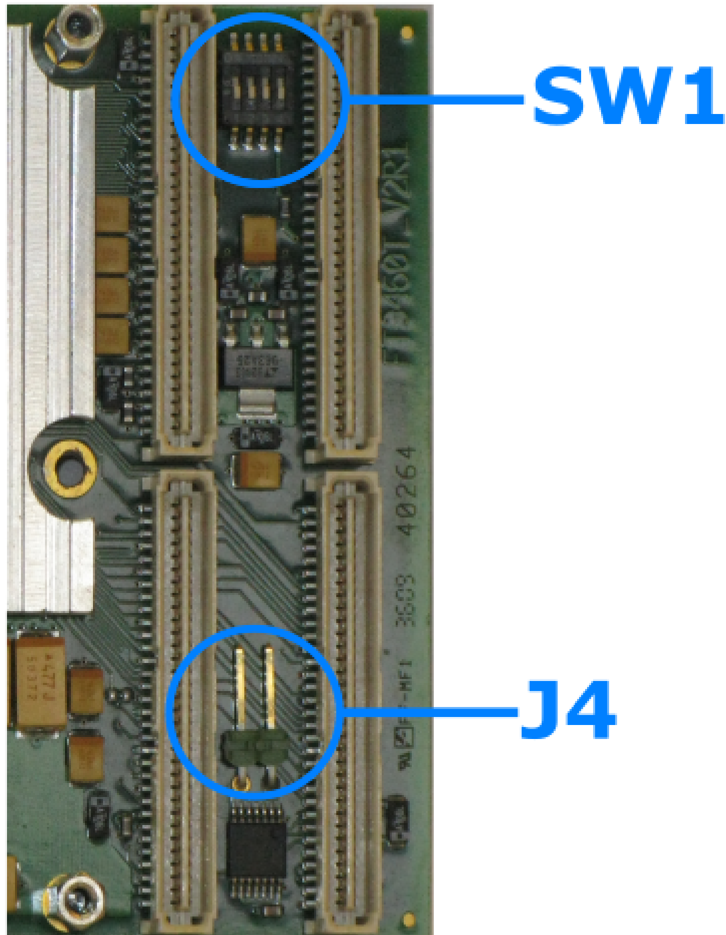
The FireTrac3460bT V1R1 has two jumpers, J1 and J4. Their positions are shown below. Actual jumpers are not shown.



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:

 Copyright(C) 2008-2020, DapTechnology, Version 2/26/2020



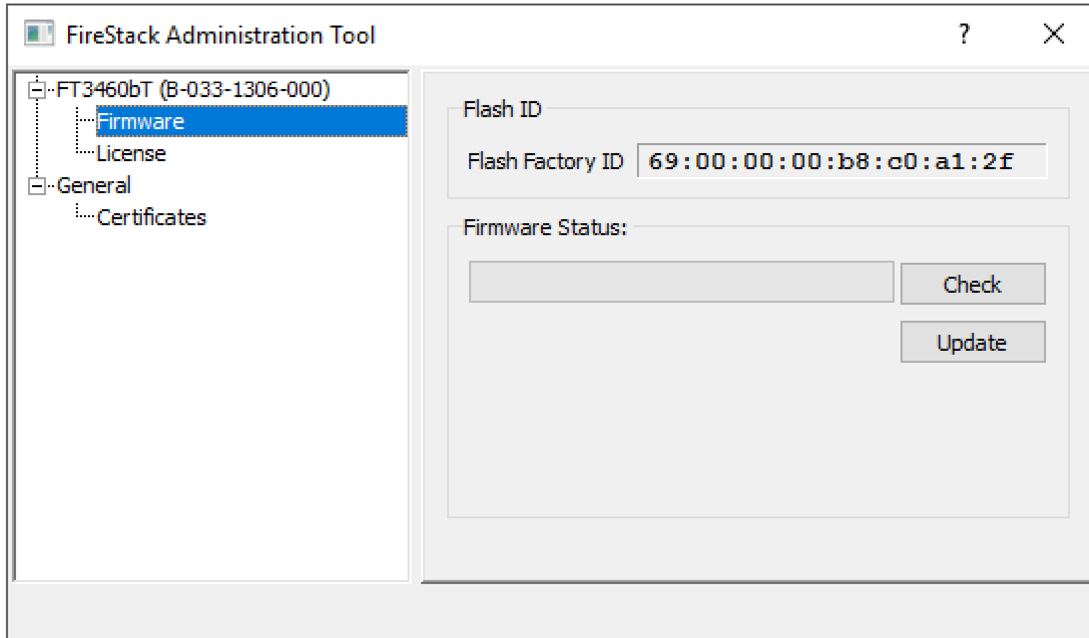
SW1 switch number 4, when enabled, protects the FLASH memory from being overwritten
J4, when placed, forces the use of the fall-back firmware when a firmware update was unsuccessful.

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.

Copyright(C) 2008-2020, DapTechnology, Version 2/26/2020

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.

Chapter 2. Software Overview

2.1. FireTrac Personality Profiles

FireTrac® provides a very universal approach to dealing with AS5643 type of traffic. It is uniquely suited for SAE AS5643 level Data Extraction and Analysis (RX) but also Data Generation and Device Simulation (TX).

With the FireTrac® platform DapTechnology has enabled dedicated implementations needed in the A&D environment. Such application-centric solutions are focused on very specific tasks and, due to their very clearly defined requirements, DapTechnology is able to even further optimize the firmware and software architecture.

Examples for such "customized" solutions are:
- Data Recorders (single/multi-channel, with/without Stealth, ...)
- System Simulators (multi-streams generator, error injection, ...)
- System Monitors (Health, Data Extraction, ...)

FireTrac® uses a combination of 1394 and AS5643 support in hardware, as well as a robust, well-designed 1394 software stack (FireStack®) running on a host computer. Interfacing to this stack can be done via an API that offers C/C++ interfaces on multiple platforms and a LabVIEW interface on Windows. Currently, DapTechnology supports Windows, VxWorks, Linux, QNX, IRIX and LabVIEW(RT) platforms. Support for other operating systems can be developed on demand.

An essential feature of FireTrac® is its adaptable software and firmware layers. In order to be usable in embedded designs having limited resources, great emphasis was given to keep the footprint and host system resource utilization fairly small. And, in order to optimize and streamline performance, the individual features and functions of FireTrac® are grouped into so-called Personality Profiles. These profiles may be purchased separately so that the user may configure their system based on individual requirements

The Personality Profiles allow the solution to be adapted for use in embedded systems which typically have limited resources, and/or with powerful host processors that can utilize even the fully featured profile.
The following profiles are available:

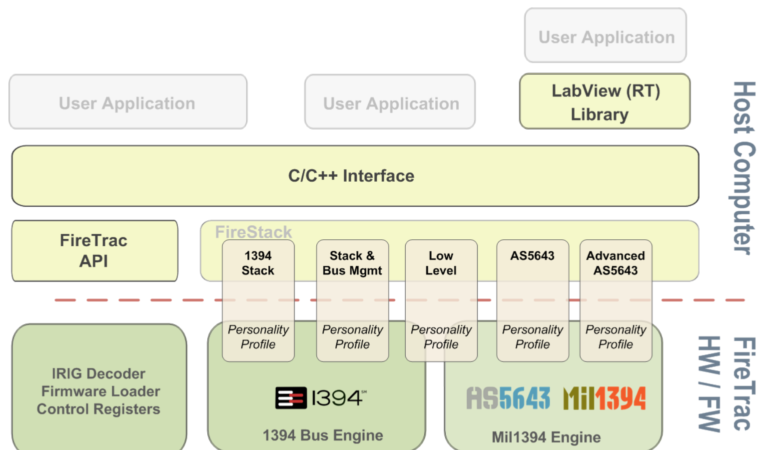
- FPP: Combination of all profiles listed below
- LLP: Low-Level API Profile
- SSP: 1394 Software Stack Profile + LLP
- SSBMP: Bus Management Profile + SSP
- AS5643P: AS5643 Bus Interface Profile + LLP

For detailed features and components of these profiles see the following table.

Functional Modules	FPP	LLP	SSP	SSBMP	AS5643P
Low-level 1394	Y	Y	Y	Y	Y
Inbound Transactions	Y		Y	Y	
Outbound Transactions	Y		Y	Y	
Isochronous Reception*	Y		Y	Y	
Isochronous Transmission*	Y		Y	Y	
Bus Management*	Y			Y	
AS5643 Frame Timing	Y				Y
AS5643 Reception	Y				Y
AS5643 Transmission	Y				Y
Time Input Device	Y				Y

* Module that is either not yet available or partly available and under development

 Copyright(C) 2008-2020, DapTechnology, Version 2/26/2020

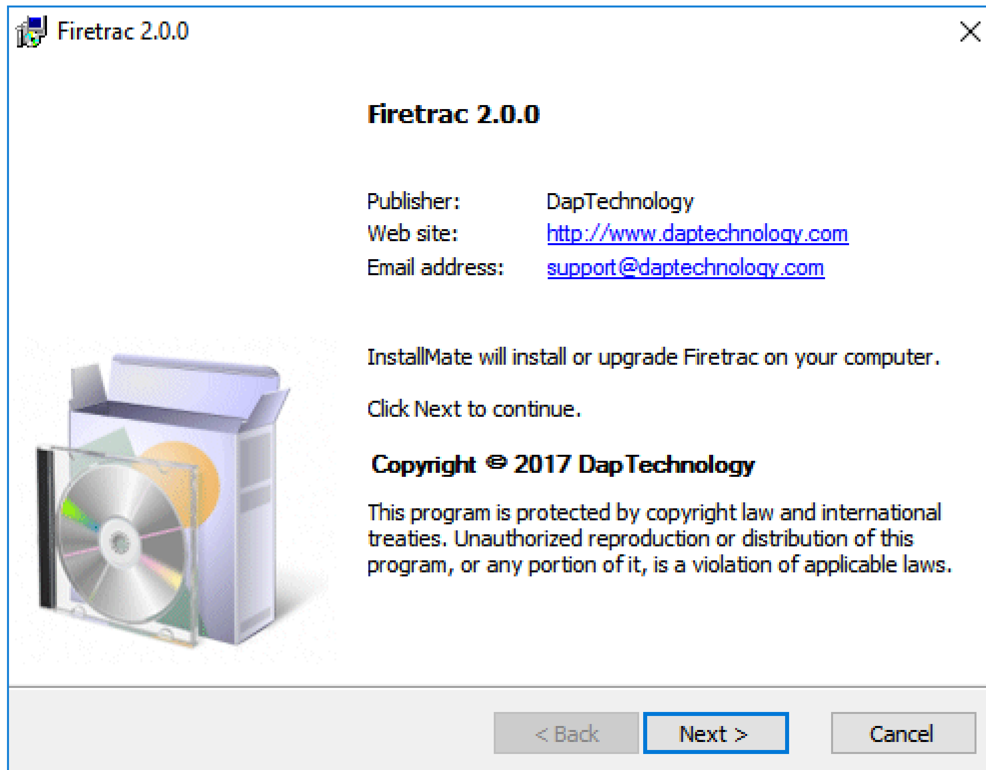

User Application
User Application
User Application
LabView (RT) Library
Host Computer
C/C++ Interface
FireTrac API
FireStack
1394 Stack
Stack & Bus Mgmt
Low Level
AS5643
Advanced AS5643
Personality Profile
Personality Profile
Personality Profile
Personality Profile
Personality Profile
IRIG Decoder
Firmware Loader
Control Registers
1394
1394 Bus Engine
AS5643 Mil1394
Mil1394 Engine
FireTrac HW / FW

2.2. Windows

2.2.1. Software Installation

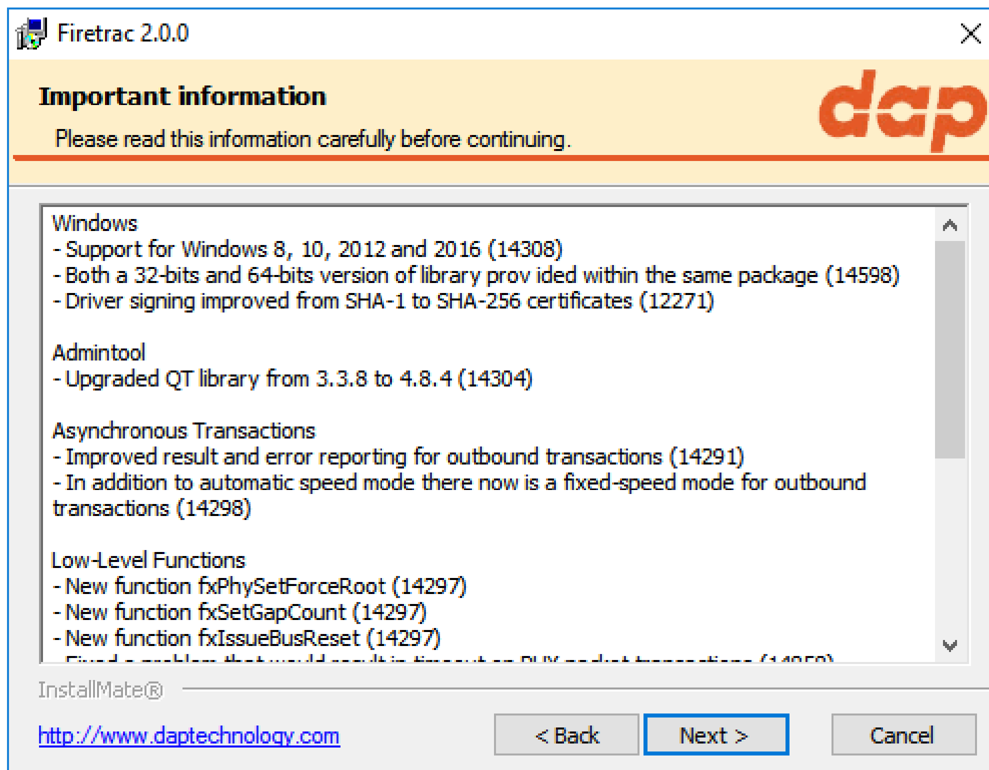
firetrac Software library comes with an installer that only allows to have one version installed at the same time. If you are using a device that is field-upgradeable, then there is a specific firmware version included that matches this version of the API. In that case, please update the firmware each time you switch to another version (either a newer or an older one) after running the installer.

To install the firetrac software, please open the '.zip' file you downloaded and run setup.exe. A dialog will open as shown below.

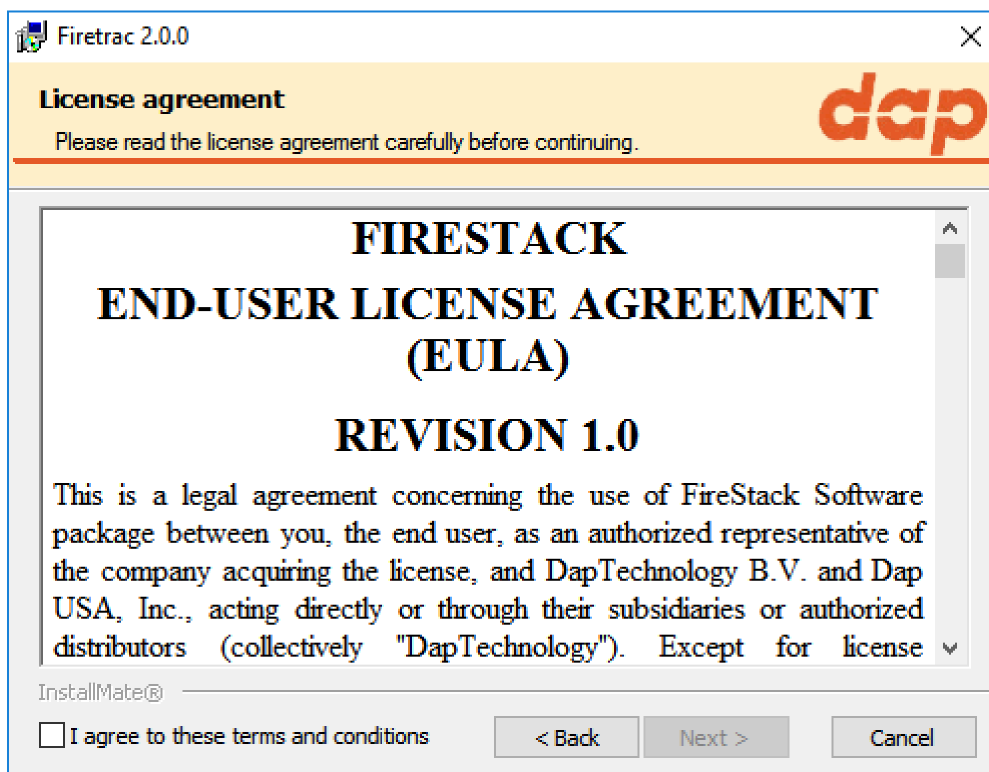


Please press Next to go to the next step. A page will be shown that contains an overview of the changes in this and the latest software versions.

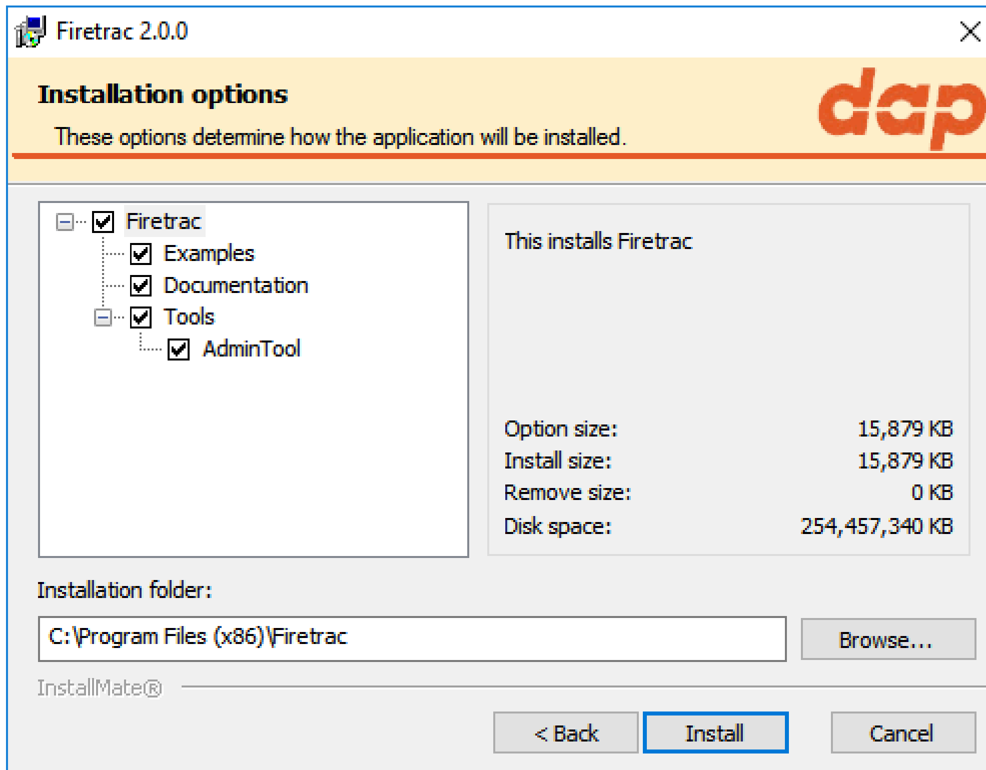
Copyright(C) 2008-2020, DapTechnology, Version 2/26/2020



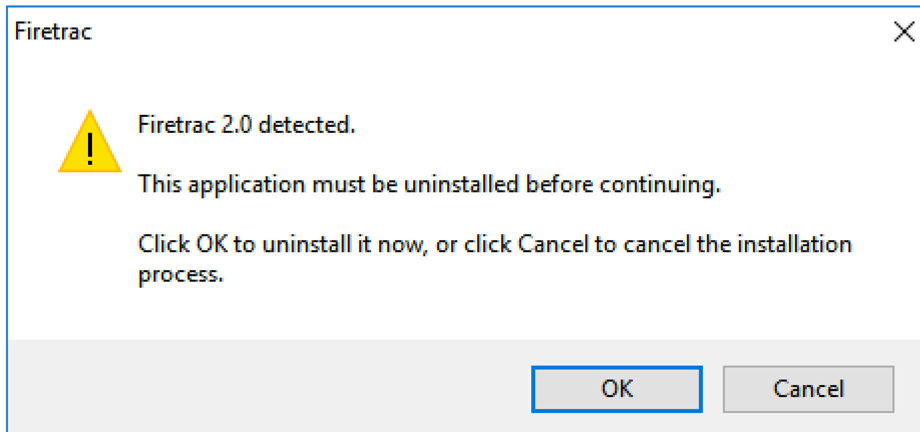
Press next to continue to the next step. A page will be shown that contains the End-user License Agreement. Please read the presented information with care and click on the checkbox when you agree with our licensing terms and press next.



You will then be asked to choose the location where the files should be installed and which components to install. It is recommended to always install all components.

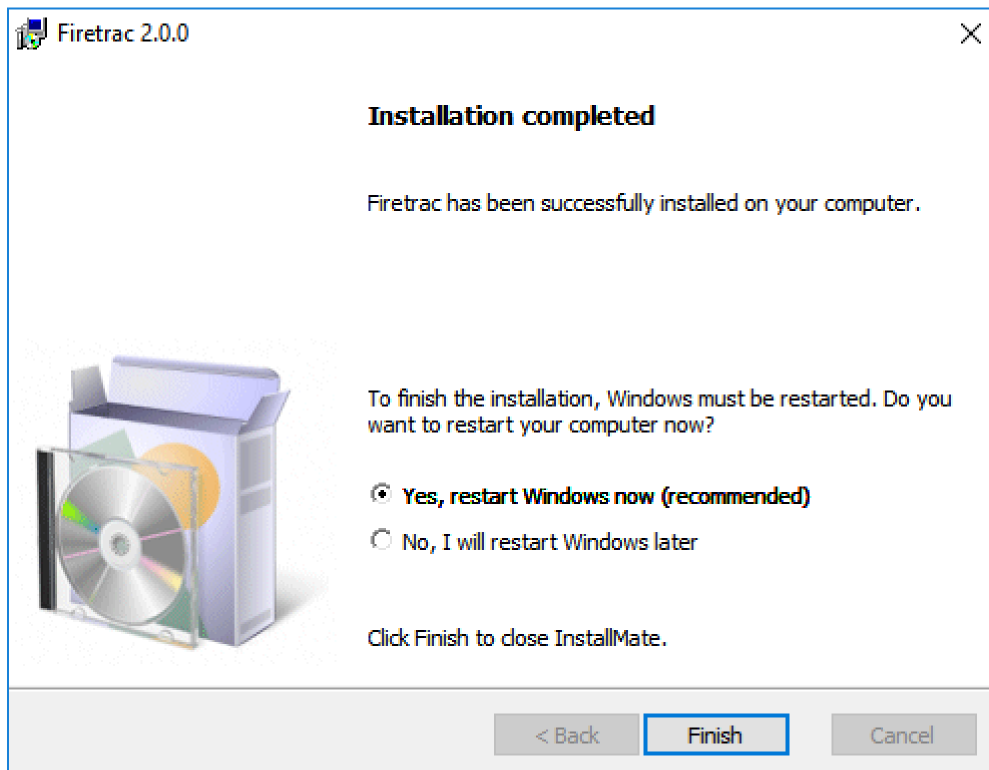


In case you are re-installing or installing over another version you will be asked to agree with the removal of the previous installation first.



A page will show installation progress and once it completes the following page will be shown.

 Copyright(C) 2008-2020, DapTechnology, Version 2/26/2020



At this point it is required to reboot the machine in order to install the device drivers. The device drivers will pre-allocate some memory that can be used for performing DMA operations and this needs to be done as early as possible after system startup. If done at a later point during system run it can take a very long time to load the driver.

Once the software is installed a complete firetrac operation manual is available through the Windows start menu.

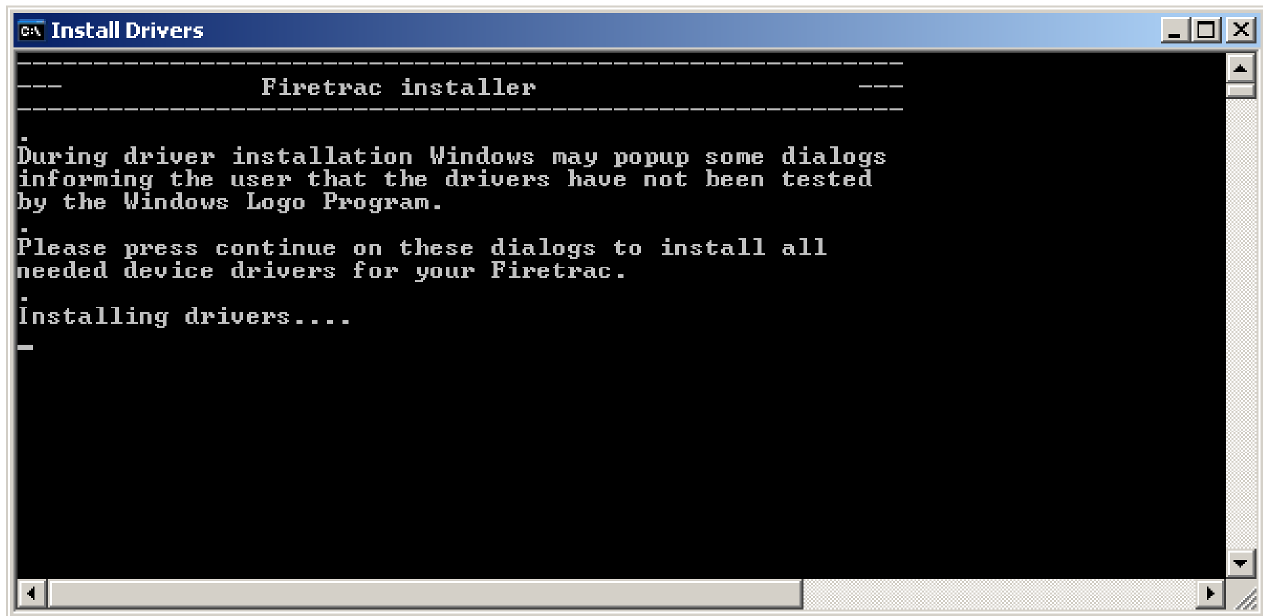
2.2.2. Manual driver (un)installation

When a supported 1394 host adapter card was installed in the system after installing the software, it may be needed to manually install the firetrac drivers to replace the Windows provided driver. This section describes both how to uninstall the firetrac drivers and revert back to the Windows default drivers and how to install the firetrac drivers instead of the Windows default drivers.

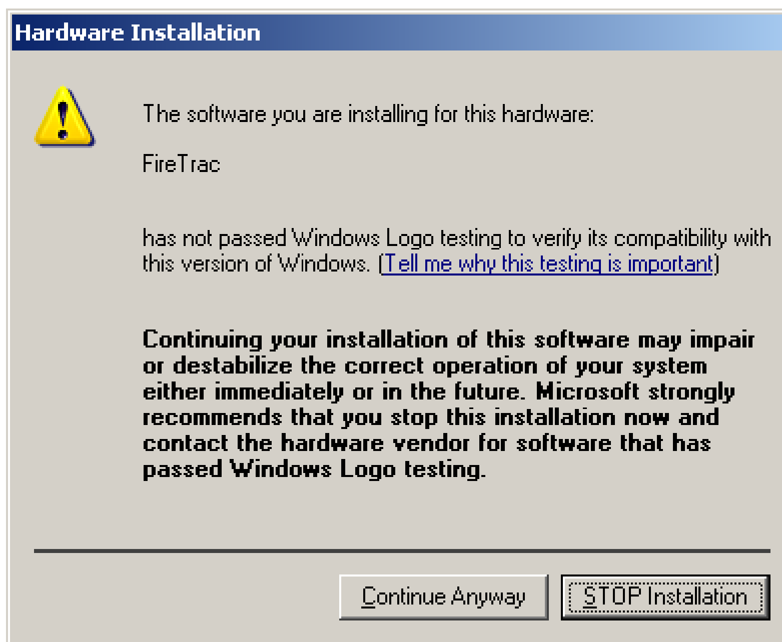
For uninstalling the drivers, simply open the Windows Start menu, Programs, firetrac Software x.x, Drivers, Uninstall Drivers. A dialog will popup and it will uninstall the drivers. This finishes quite quickly.

Important: When installing the drivers, please first reboot your machine and then follow the steps below as soon as the PC finishes startup. If the steps below are performed after the PC has been running for a while it can take a very very long time to finish.

For installing the drivers, simply open the Windows Start menu, Programs, firetrac Software x.x, Drivers, Install Drivers. A dialog will popup and it will install the drivers. The dialog looks as follows.

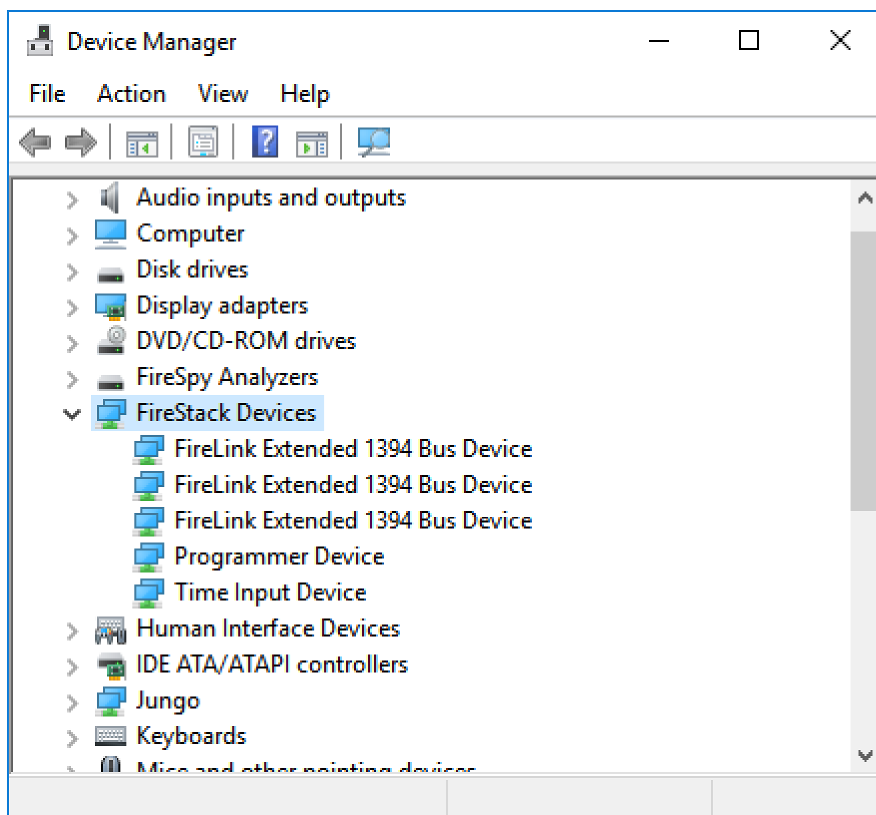


During driver installation depending on the exact Windows version used, it may popup several dialogs informing the user that the driver has not passed Windows Logo testing. Please press continue on these dialogs to install the driver.



To verify the drivers have been installed correctly, please open the Windows device manager. It should contain a new category named "FireStack Devices" with some items as child nodes. Your device is split up in several functions, each one represented here. (individual device names may differ from the picture)

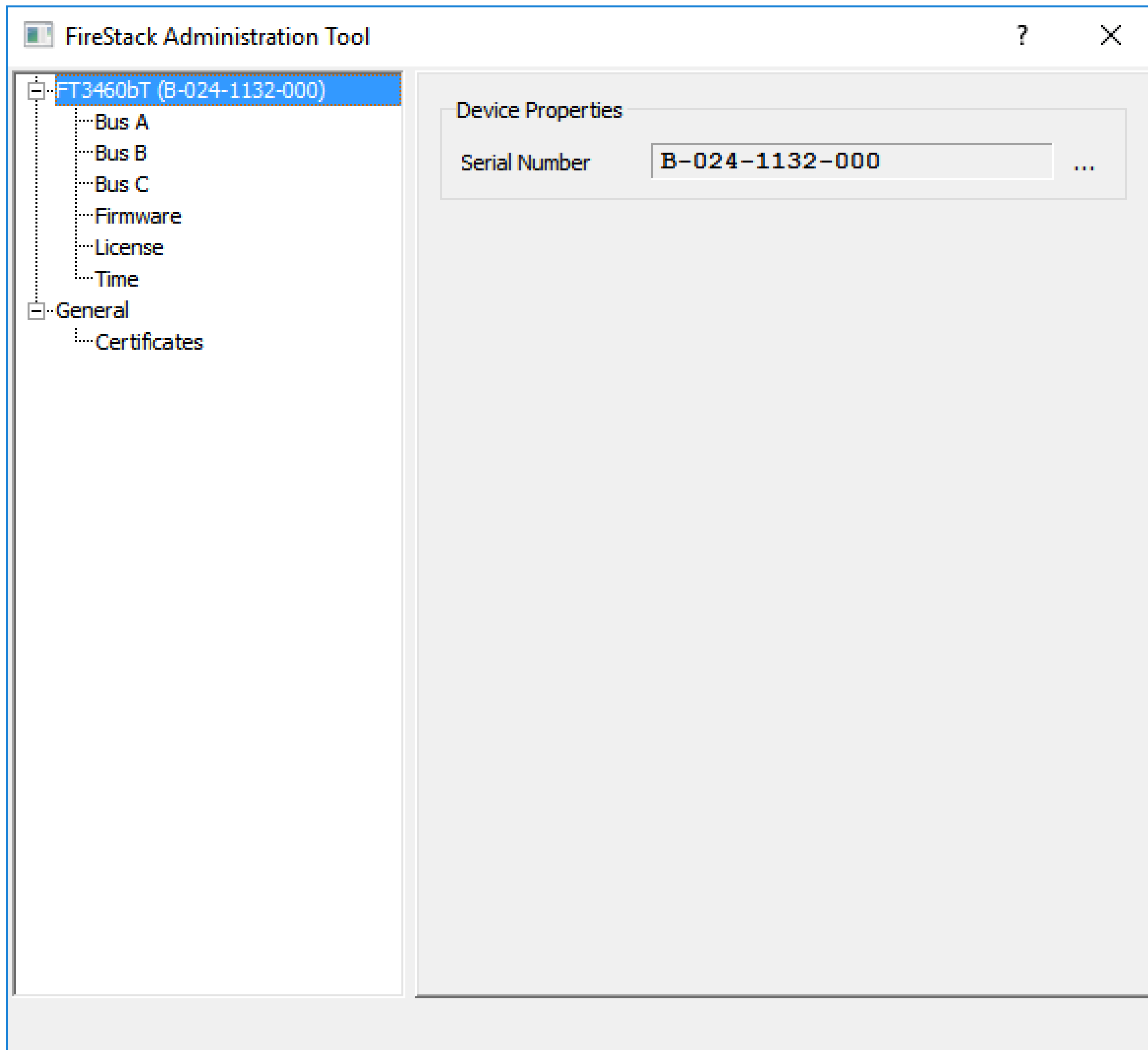
Copyright(C) 2008-2020, DapTechnology, Version 2/26/2020



2.2.3. Administration Tool

The Administration Tool can be used to manage your license certificates, update the firmware and view information about the devices that are present in your system.

When the device(s) and its driver have been properly installed, the devices will be listed on the left, as shown below. Here you can see the serial number and the busses of the device.



2.2.3.1. Updating Firmware

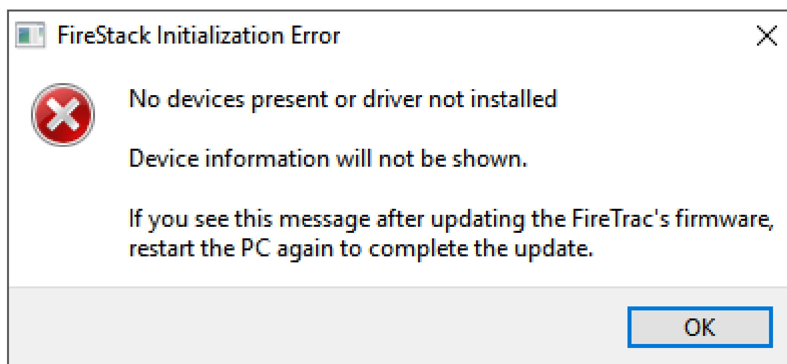
In order to use the device, the firmware version must match the FireStack software version. You can update the device using the firmware update page, as shown below. This page can be accessed by selecting the "Firmware Updater" tree item of the device that needs to be updated as shown below.

Copyright(C) 2008-2020, DapTechnology, Version 2/26/2020

You can determine whether the device has been updated using the Check button, or update the firmware with the Update button. After the update, the firmware will be automatically checked to see if everything went well. The check and update procedures can be canceled using the Cancel button within the Firmware Status frame.

The firmware will be loaded in the card when the PCI bus power is reset, which usually requires turning the machine off and on again(!). Please make sure to turn off your PC completely after updating the firmware.

After installing a new FireStack software version on Windows 10, and updating the Firmware of your device, the new drivers may not be loaded correctly after the power-cycle. The following message will appear when starting the Admintool. Reboot the PC to let Windows load the drivers correctly.

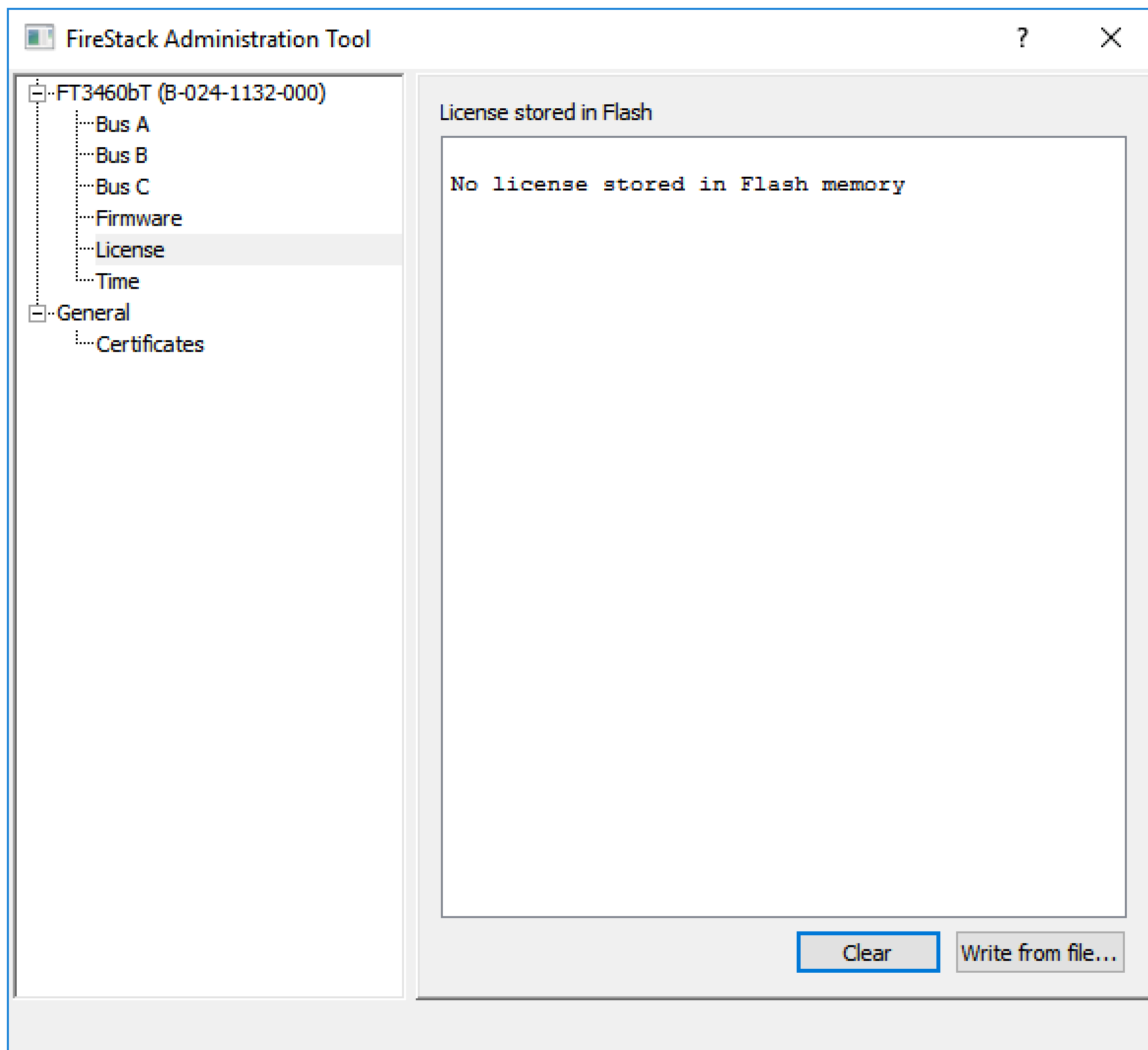


In case the firmware was not updated properly, the device will use its backup firmware after the restart, which enables you to try to update it again in the same manner. The busses will not be visible under the Device node, since no other FireStack functionality can be used as long as the proper firmware has not been uploaded.

2.2.3.2. Registering Licenses

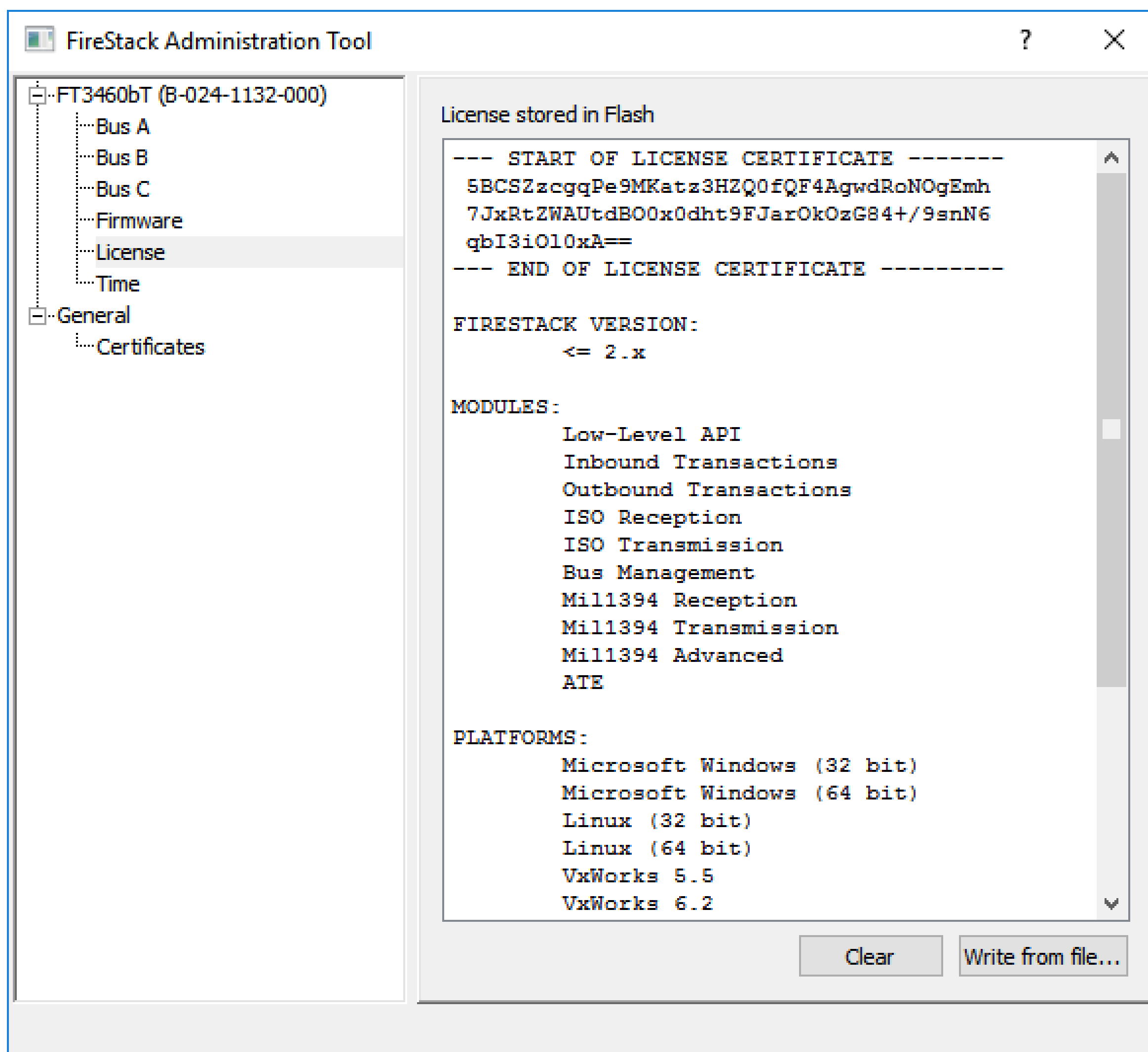
Without a license certificate the firetrac software will run in Demo mode and will only allow the software to run continuously for about 15 minutes. In this mode all features and modules are enabled. In order to use the device in normal mode a license certificate is required. A license certificate can be loaded programmatically from within the user-defined application by using the firetrac software API.

It is also possible to store a license certificate in Flash memory on the device. The Administration Tool can be used to program a license certificate into Flash memory of the device by selecting the 'Licenses' node as shown below. The picture below shows that for the selected device no license key is loaded into Flash memory.

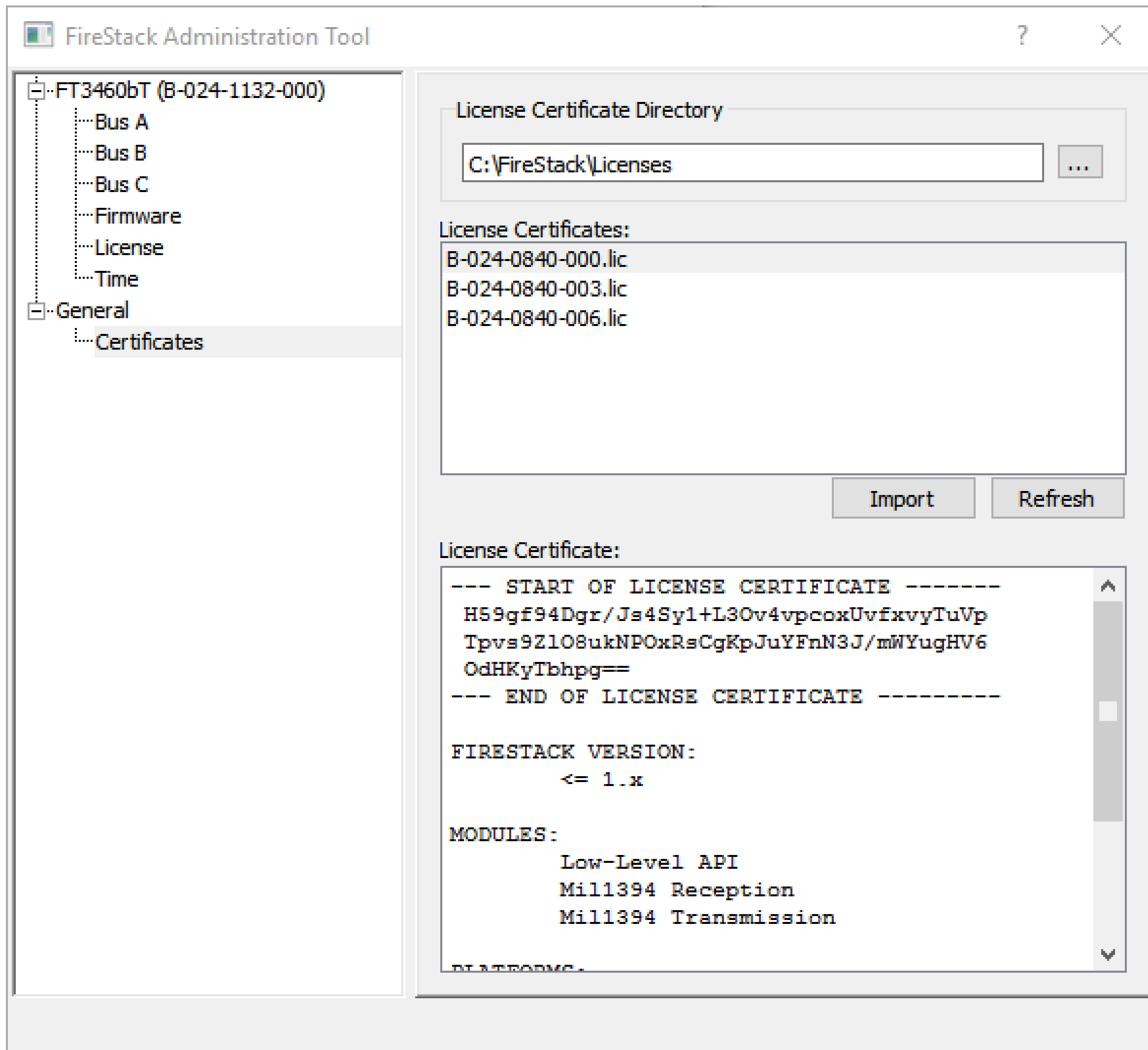


By pressing "Write from file..." and selecting the right license certificate file for the selected device, the Admintool will program the license certificate into the card. If successful the license details will be shown as in the picture below. Please note that "SERIAL NUMBERS" will output an item for each 1394 node of the device. Only the first item is the actual serial number of the card but all items are used as unique identifier for the device busses.

 Copyright(C) 2008-2020, DapTechnology, Version 2/26/2020



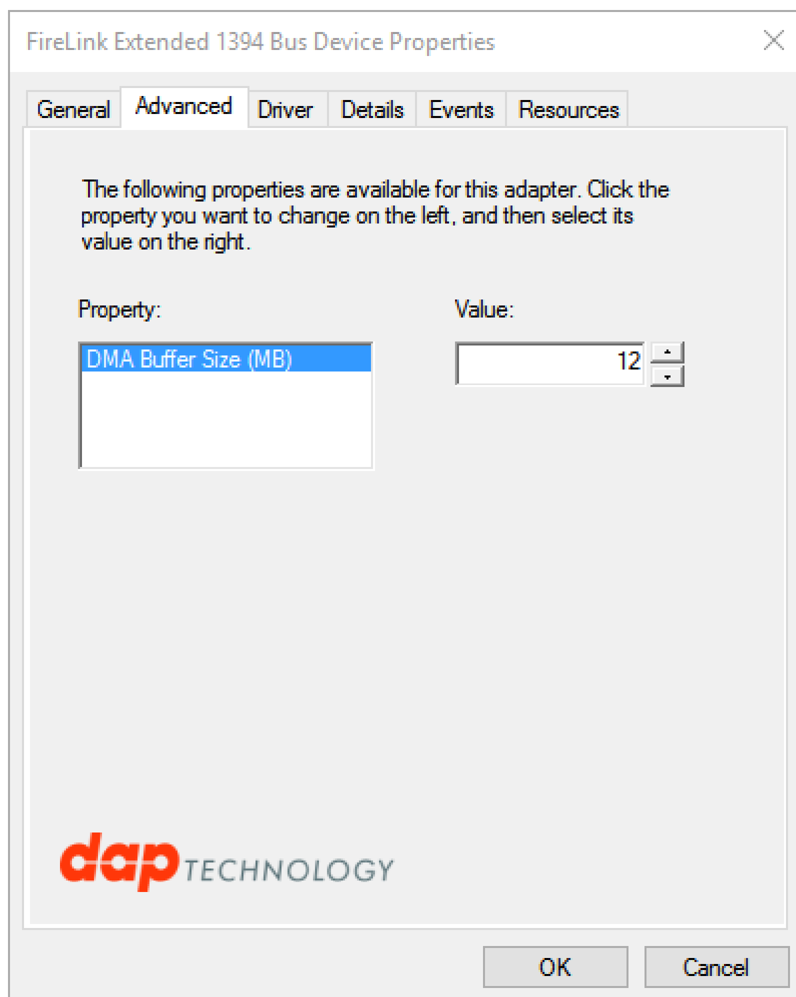
For devices without flash memory, license certificate files can be stored on the local file system. The location of the files can be changed in the Admintool by clicking the browse (...) button in the "License Certificate Directory" box.
After copying a new license file to this location a refresh might be needed before the new license is visible.
The content of a license can be made visible by selecting it.



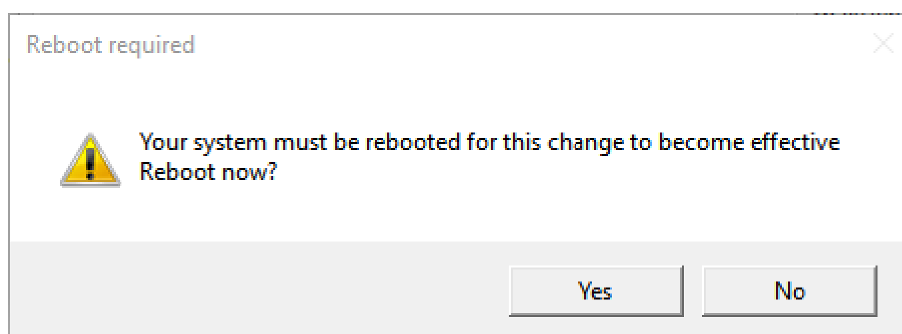
2.2.4. Driver parameter settings

The firetrac driver has user configurable options that can be set via the Windows device manager. The options for each function of the device driver can be changed in the device driver properties. Expand the category named FireStack Devices, right click on the device's function and open the properties window.

Copyright(C) 2008-2020, DapTechnology, Version 2/26/2020



Click on OK to accept any changes made to the properties or Cancel to cancel the action, leaving all properties unchanged.
After changing an option, it may be necessary to reboot the system. Windows will show a dialog to reboot the system when the properties windows is closed by pressing OK.



If parameters of multiple drivers need to be changed, the system can be rebooted after changing the parameters for all drivers.

Configurable option:

- DMA Buffer Size: Size of the pre-allocated buffer size for DMA transfers, please refer to Memory Management in the API Reference for more information.

2.2.5. Notes on Borland Builder

The import library file (".lib") that ships with the library is generated using Visual C++ and uses the COFF library format. Borland uses the OMF format. This means that the included import library can not be included

Copyright(C) 2008-2020, DapTechnology, Version 2/26/2020

directly in a C++ Builder project.

Borland Builder comes with a tool called "COFFtoOMF.EXE" to convert from COFF to OMF format. After converting the firetrac import library it should be possible to include it in a Borland project.

When using the firetrac library in a Borland project, the following option must be turned on:
- Treat enum types as int

 Copyright(C) 2008-2020, DapTechnology, Version 2/26/2020

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.

Copyright(C) 2008-2020, DapTechnology, Version 2/26/2020

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.

Copyright(C) 2008-2020, DapTechnology, Version 2/26/2020

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

Copyright(C) 2008-2020, DapTechnology, Version 2/26/2020

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

Copyright(C) 2008-2020, DapTechnology, Version 2/26/2020

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

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()`

Copyright(C) 2008-2020, DapTechnology, Version 2/26/2020

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.

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

7.1.2. Special Types

FXReturnCode	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.
FXBusHandle	Handle to a bus opened by the fxCreateBusHandle function.

Copyright(C) 2008-2020, DapTechnology, Version 2/26/2020

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;
```


Copyright(C) 2008-2020, DapTechnology, Version 2/26/2020

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.

 Copyright(C) 2008-2020, DapTechnology, Version 2/26/2020

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()
```

8.2. 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(
```

Copyright(C) 2008-2020, DapTechnology, Version 2/26/2020

```
        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.

Copyright(C) 2008-2020, DapTechnology, Version 2/26/2020

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. 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.

 Copyright(C) 2008-2020, DapTechnology, Version 2/26/2020

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.

The memory needed for DMA transfers is allocated by the device driver during system startup, this assures that large buffers of contiguous memory are always available to the user without any delays.
Per default the FireStack will allocate an area of memory that will be suitable for most general applications. If a user needs more than the amount of memory assigned during installation, this can be changed via the Device Manager. The device manager allows the allocation of 1 - 32 MB of memory in 1 MB increments.
See Driver parameter settings for a detailed description on how to change the available DMA memory size.

8.3.1. Functions

8.3.1.1. fxMemAlloc

Description
This function can be used to allocate memory that can later by 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.

Copyright(C) 2008-2020, DapTechnology, Version 2/26/2020

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
)
```


Copyright(C) 2008-2020, DapTechnology, Version 2/26/2020

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;
```

Copyright(C) 2008-2020, DapTechnology, Version 2/26/2020

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(
```

 Copyright(C) 2008-2020, DapTechnology, Version 2/26/2020

```
        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_INVALID_DRIVERVERSION
FX_ERR_DRIVER_NOT_FOUND
FX_ERR_WIN_COULD_NOT_LOAD_LICENSE

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.

Copyright(C) 2008-2020, DapTechnology, Version 2/26/2020

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

 Copyright(C) 2008-2020, DapTechnology, Version 2/26/2020

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	One of the modes documented in Frame Synchronization Input Modes or-ed with one of the modes documented in Frame Synchronization Output Modes. default: (`FX_MIL_SYNC_FREERUNNING | FX_MIL_SYNC_OUT_NONE`)
frameLength	The duration of a frame expressed in micro seconds. A negative value means that the frame length will not be changed. minimum: 3000, maximum: 30000, default: 12500
syncMargin	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. When a synchronization event is detected before the frame offset time reaches frameLength-syncMargin, the synchronization event will result in clearing the frame offset time, but no new frame is started (frame counter is not incremented). When no synchronization events are detected and the frame offset reaches a value of frameLength+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). A negative value means that the synchronization margin will not be changed. minimum:10, maximum 3000, default: 100
channel	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. A negative value means that the synchronization channel will not be changed. minimum: 0, maximum: 63, default: 31
controlFlags	Combination of one or more of the flags documented in Frame Control Flags.
dmaPreloadTime	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. 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. 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

 Copyright(C) 2008-2020, DapTechnology, Version 2/26/2020

	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

Copyright(C) 2008-2020, DapTechnology, Version 2/26/2020

9.1.4.2. Frame Synchronization Input Modes

The following frame synchronization input modes are available for use with the FXMilFrameTimingOptions 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 FXMilFrameTimingOptions 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

Copyright(C) 2008-2020, DapTechnology, Version 2/26/2020

	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.`

 Copyright(C) 2008-2020, DapTechnology, Version 2/26/2020

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.

Copyright(C) 2008-2020, DapTechnology, Version 2/26/2020

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
)
```

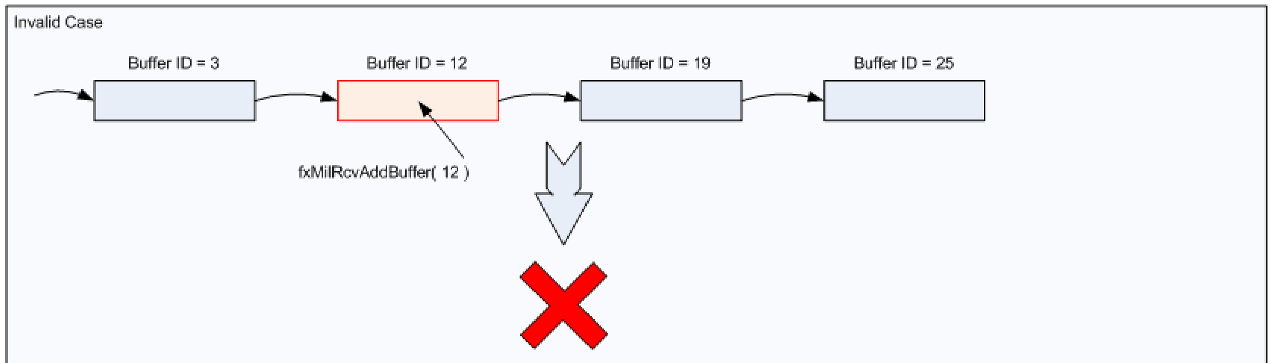
9.2.2.3. Buffer Control

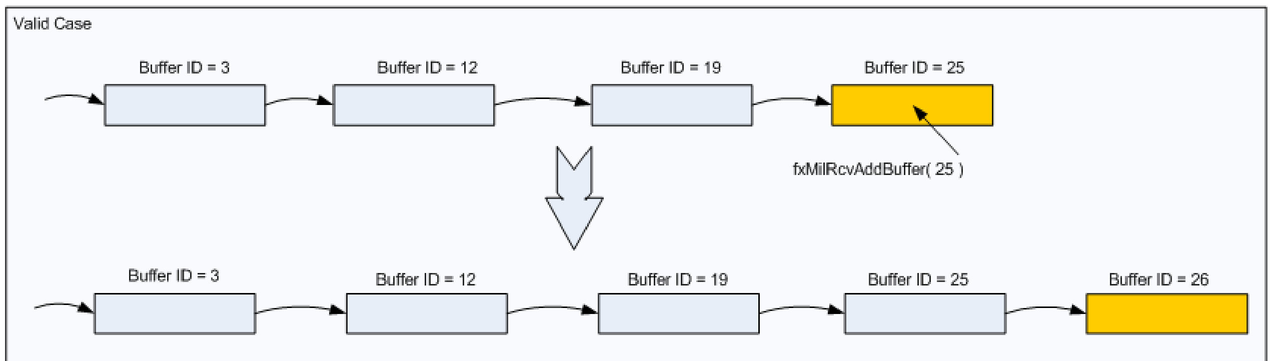
In order to start a context for reception, the user first needs to setup the necessary buffers for packet storage. Buffer Control functions can be used to register memory buffers with the AS5643 reception module. The user needs to take care of allocating memory blocks that can be used as reception buffer. 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

Copyright(C) 2008-2020, DapTechnology, Version 2/26/2020

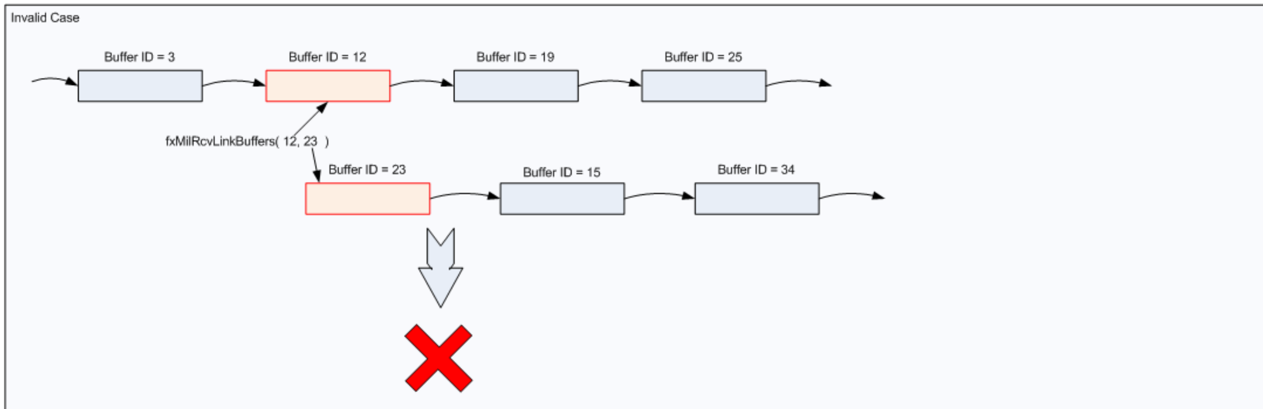
all times. After removing a buffer from the reception list, the user may write and/or free memory again.

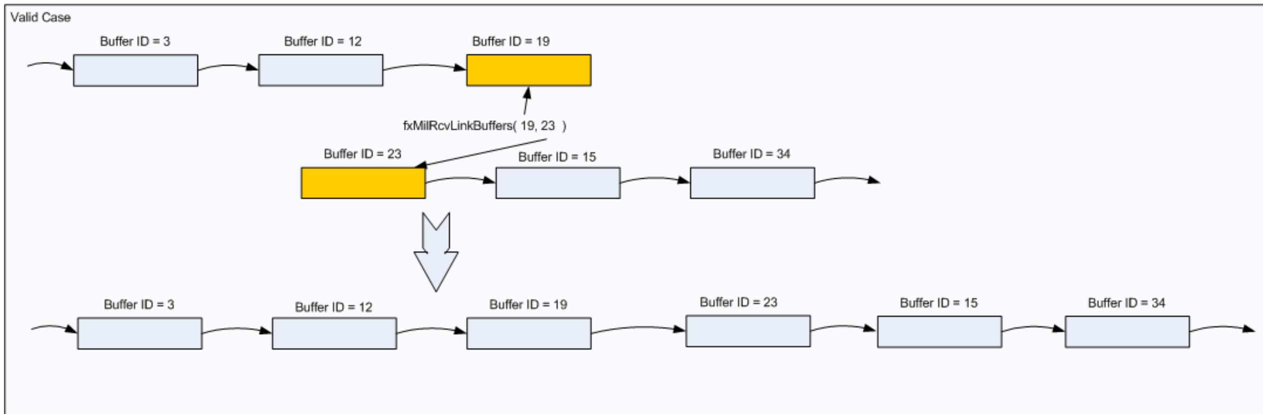
Buffers need to be setup such that they form a list. It is not allowed to link buffers in a loop. Having buffers in a loop fashion would yield unspecified results.
For each buffer the user may set buffer options. However, please note that options need to be the same for all buffers in the same list.
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.



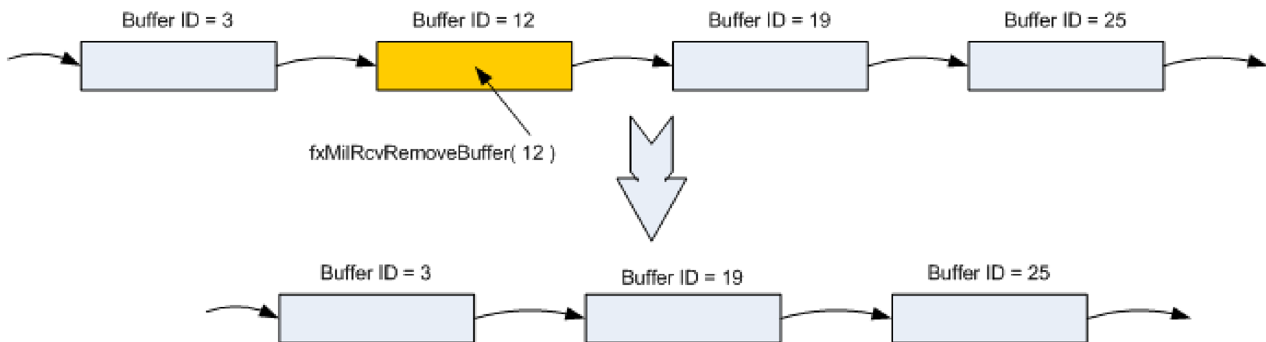


Add a new buffer to existing list - previous buffer must be the last one of a list





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 fxMilRcvAddBuffer

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

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

Copyright(C) 2008-2020, DapTechnology, Version 2/26/2020

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
```

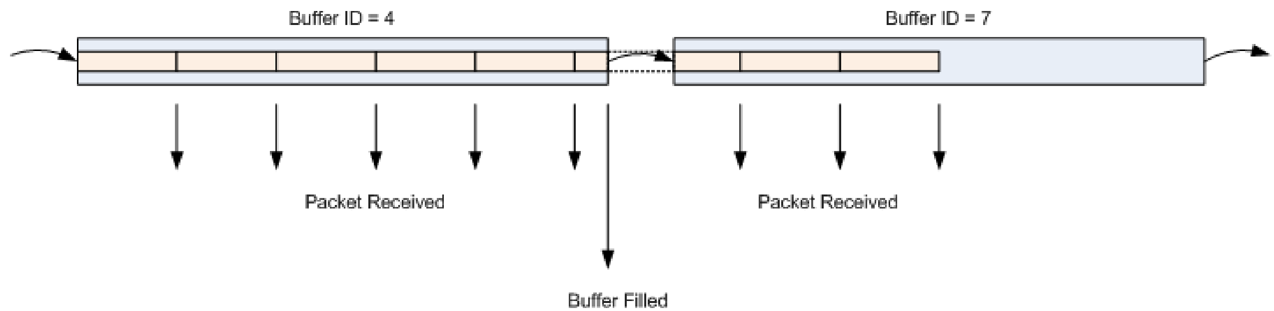
 Copyright(C) 2008-2020, DapTechnology, Version 2/26/2020

)

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 then 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

`busHandle`	`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
)
```

 Copyright(C) 2008-2020, DapTechnology, Version 2/26/2020

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.

Copyright(C) 2008-2020, DapTechnology, Version 2/26/2020

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

Synopsis

```
typedef struct {
      bool_t                    anyMessage;
      uint32_t                  messageID;
      boolt_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 Staus 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

Copyright(C) 2008-2020, DapTechnology, Version 2/26/2020

	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;
```

 Copyright(C) 2008-2020, DapTechnology, Version 2/26/2020

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.

<table>
<tr><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td colspan="16">data length</td><td colspan="2">tag</td><td colspan="6">channel</td><td colspan="4">tcode</td><td colspan="4">sy</td></tr>
<tr><td colspan="32">stream data (include padding if needed)</td></tr>
<tr><td colspan="32">timestamp seconds</td></tr>
<tr><td colspan="2">time stamp status</td><td></td><td></td><td></td><td></td><td></td><td></td><td colspan="24">timestamp sub seconds</td></tr>
<tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td>VO</td><td colspan="3">speed</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
</table>

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

Copyright(C) 2008-2020, DapTechnology, Version 2/26/2020

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

busHandle	Reference handle to the bus to control. (see fxCreateBusHandle)
mode	Controls the transmission mode the context will be opened in. (see Context Modes)
contextHandle	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

busHandle	Reference handle to the bus to control. (see fxCreateBusHandle)
optionList	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.
listSize	Specifies the number of items in the optionList.
contextHandle	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

Copyright(C) 2008-2020, DapTechnology, Version 2/26/2020

```
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

 Copyright(C) 2008-2020, DapTechnology, Version 2/26/2020

	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.

 Copyright(C) 2008-2020, DapTechnology, Version 2/26/2020

	(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.

Copyright(C) 2008-2020, DapTechnology, Version 2/26/2020

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.

 Copyright(C) 2008-2020, DapTechnology, Version 2/26/2020

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.

 Copyright(C) 2008-2020, DapTechnology, Version 2/26/2020

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
)
```


Copyright(C) 2008-2020, DapTechnology, Version 2/26/2020

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.

Copyright(C) 2008-2020, DapTechnology, Version 2/26/2020

	(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.

Copyright(C) 2008-2020, DapTechnology, Version 2/26/2020

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 {
```

 Copyright(C) 2008-2020, DapTechnology, Version 2/26/2020

```
        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	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. Network Bus Mode Bits: • 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 Dirtections. 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

 Copyright(C) 2008-2020, DapTechnology, Version 2/26/2020

	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 jiter 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_BOTH	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_POS	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_NEG	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 harwdare capabilities): • 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 transmitted on. Value needs to be positve 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)

Copyright(C) 2008-2020, DapTechnology, Version 2/26/2020

FX_MIL_TRM_OPT_ISFRAMESEPARATOR	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: • 0 - disabled meaning it is a regular message • 1 - enabled, meaning it is not a message but a wait start of frame item Default: 0

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 thejitter 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().

<table>
<tr><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td colspan="32">Item Type</td></tr>
<tr><td colspan="32">Data (1 ~ N quadlets)</td></tr>
</table>

Item Type (32 bits)

1: Frame Separator

Copyright(C) 2008-2020, DapTechnology, Version 2/26/2020

3: Stream Packet

Item Type 1 - Frame Separator

<table>
<tr><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td colspan="32">1</td></tr>
<tr><td colspan="32">Reserved</td></tr>
</table>

Item Type 3 - Stream Packet

<table>
<tr><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td colspan="32">3</td></tr>
<tr><td colspan="32">Offset</td></tr>
<tr><td colspan="32">Speed</td></tr>
<tr><td colspan="32">Payload Size</td></tr>
<tr><td colspan="32">Header</td></tr>
<tr><td colspan="32">Payload Data</td></tr>
</table>

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

 Copyright(C) 2008-2020, DapTechnology, Version 2/26/2020

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

Copyright(C) 2008-2020, DapTechnology, Version 2/26/2020

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
);
```

 Copyright(C) 2008-2020, DapTechnology, Version 2/26/2020

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
)
```

 Copyright(C) 2008-2020, DapTechnology, Version 2/26/2020

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 Functions

10.2.1.2.1 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

Copyright(C) 2008-2020, DapTechnology, Version 2/26/2020

	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 whichFXRequestHandlerCallback 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,

Copyright(C) 2008-2020, DapTechnology, Version 2/26/2020

	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

Copyright(C) 2008-2020, DapTechnology, Version 2/26/2020

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 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.

Copyright(C) 2008-2020, DapTechnology, Version 2/26/2020

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
)
```

 Copyright(C) 2008-2020, DapTechnology, Version 2/26/2020

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,
```


Copyright(C) 2008-2020, DapTechnology, Version 2/26/2020

```
        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 functionfxGetErrorMessage 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

 Copyright(C) 2008-2020, DapTechnology, Version 2/26/2020

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

 Copyright(C) 2008-2020, DapTechnology, Version 2/26/2020

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. Low-Level 1394

10.4.1. Settings

10.4.1.1. 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

10.4.2. Functions

10.4.2.1. Asynchronous Packet Reception Functions

10.4.2.1.1 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

Copyright(C) 2008-2020, DapTechnology, Version 2/26/2020

Synopsis

```
FXReturnCode fxAsyRcvWaitSingleRequest(
      FXBusHandle        handle,
      uint8_t*           data,
      uint32_t*          size,
      uint32_t*          speed,
      uint32_t*          ackCode,
      uint32_t           timeout
)
```

10.4.2.1.2 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
)
```

10.4.2.1.3 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
);
```

10.4.2.2. Single Packet Transmission Functions

10.4.2.2.1 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(
```

 Copyright(C) 2008-2020, DapTechnology, Version 2/26/2020

```
        FXBusHandle               handle,
        uint32_t                  speed,
        uint32_t                  nodeID,
        uint32_t                  transactionLabel,
        uint32_t                  retryCode,
        const FXAddress64*        offset,
        uint32_t                  data
)
```

10.4.2.2.2 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
)
```

10.4.2.2.3 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
)
```

10.4.2.2.4 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

Copyright(C) 2008-2020, DapTechnology, Version 2/26/2020

Synopsis

```
FXReturnCode fxAsyTrmReadQuadletRequest(
      FXBusHandle              handle,
      uint32_t                 speed,
      uint32_t                 nodeID,
      uint32_t                 transactionLabel,
      uint32_t                 retryCode,
      const FXAddress64*       offset
)
```

10.4.2.2.5 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
)
```

10.4.2.2.6 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
)
```

10.4.2.2.7 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

Copyright(C) 2008-2020, DapTechnology, Version 2/26/2020

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
)
```

10.4.2.2.8 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
)
```

10.4.2.2.9 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
)
```

10.4.2.2.10 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.

 Copyright(C) 2008-2020, DapTechnology, Version 2/26/2020

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
)
```

10.4.2.3. PHY Packets and registers

10.4.2.3.1 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
)
```

10.4.2.3.2 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
)
```

10.4.2.3.3 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

 Copyright(C) 2008-2020, DapTechnology, Version 2/26/2020

```
FXReturnCode fxWriteLocalPhyRegister(
      FXBusHandle        handle,
      uint8_t            offset,
      uint8_t            value
)
```

10.4.2.3.4 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
)
```

10.4.2.3.5 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 + 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
)
```

10.4.2.3.6 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
)
```

10.4.2.3.7 fxPhyRemoteCommand

Description
This function can be used to perform a PHY Remote Command on a remote node.

Parameters

 Copyright(C) 2008-2020, DapTechnology, Version 2/26/2020

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
);
```

10.4.2.3.8 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
)
```

10.4.2.3.9 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
)
```

10.4.2.3.10 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

Copyright(C) 2008-2020, DapTechnology, Version 2/26/2020

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
)
```

10.4.2.3.11 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
);
```

10.4.2.3.12 fxPhyPacketTrmRaw

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
);
```

10.4.2.4. Topology Functions

10.4.2.4.1 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
)
```

10.4.2.4.2 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)

 Copyright(C) 2008-2020, DapTechnology, Version 2/26/2020

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
)
```

10.4.3. Type Definitions

10.4.3.1. 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
);
```

10.4.3.2. 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
);
```

10.4.4. Structures

10.4.4.1. 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;
};
```

10.4.5. Constants

10.4.5.1. 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

10.4.5.2. PHY Remote Commands

```
FX_PHY_CMD_NOP                          0
FX_PHY_CMD_TX_DISABLE_NOTIFY            1
FX_PHY_CMD_INIT_SUSPEND                 2
```


Copyright(C) 2008-2020, DapTechnology, Version 2/26/2020

```
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
```

10.4.5.3. 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
```

Copyright(C) 2008-2020, DapTechnology, Version 2/26/2020

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.

 Copyright(C) 2008-2020, DapTechnology, Version 2/26/2020

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

Copyright(C) 2008-2020, DapTechnology, Version 2/26/2020

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
)
```

 Copyright(C) 2008-2020, DapTechnology, Version 2/26/2020

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
);
```

Copyright(C) 2008-2020, DapTechnology, Version 2/26/2020

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

deviceId	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.
pciBus	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.
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.
subSecondResolution	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;
```


Copyright(C) 2008-2020, DapTechnology, Version 2/26/2020

```
        /* 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.

FX_TIMEINPUTMODE_FREERUNNING	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().
FX_TIMEINPUTMODE_IRIG	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 FX_TIMEINPUTMODE_OPTION_TTL or FX_TIMEINPUTMODE_OPTION_RS422.
FX_TIMEINPUTMODE_IRIG_1344	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 FX_TIMEINPUTMODE_OPTION_TTL or FX_TIMEINPUTMODE_OPTION_RS422.

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:

FX_TIMEINPUTMODE_OPTION_TTL	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 FX_TIMEINPUTMODE_OPTION_INVERTED option. This option is only allowed in combination with FX_TIMEINPUTMODE_OPTION_IRIG or FX_TIMEINPUTMODE_OPTION_IRIG_1344.
FX_TIMEINPUTMODE_OPTION_RS422	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 FX_TIMEINPUTMODE_OPTION_INVERTED option. This option is only allowed in combination with FX_TIMEINPUTMODE_OPTION_IRIG or FX_TIMEINPUTMODE_OPTION_IRIG_1344.
FX_TIMEINPUTMODE_OPTION_INVERTED	This option changes the signaling from non-inverted active high to inverted active-low. This option is only allowed in combination with FX_TIMEINPUTMODE_OPTION_TTL or FX_TIMEINPUTMODE_OPTION_RS422.

Copyright(C) 2008-2020, DapTechnology, Version 2/26/2020

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.

Before you generate the project file, please copy this directory to a location outside the program files. This prevents problems with write rights and your changes will be untouched when a new version is installed.
In order to generate the project files for the examples, please run the batch file that can be found in the Examples folder of the Installation folder.

 Copyright(C) 2008-2020, DapTechnology, Version 2/26/2020

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 repsponse, 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 0x00FFFF000020 to 0x00FFFF00009F (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 repsponse
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.

Copyright(C) 2008-2020, DapTechnology, Version 2/26/2020

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.

 Copyright(C) 2008-2020, DapTechnology, Version 2/26/2020

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.

 Copyright(C) 2008-2020, DapTechnology, Version 2/26/2020

12.4. AS5643 Reception

12.4.1. Mil1394 Data Logger

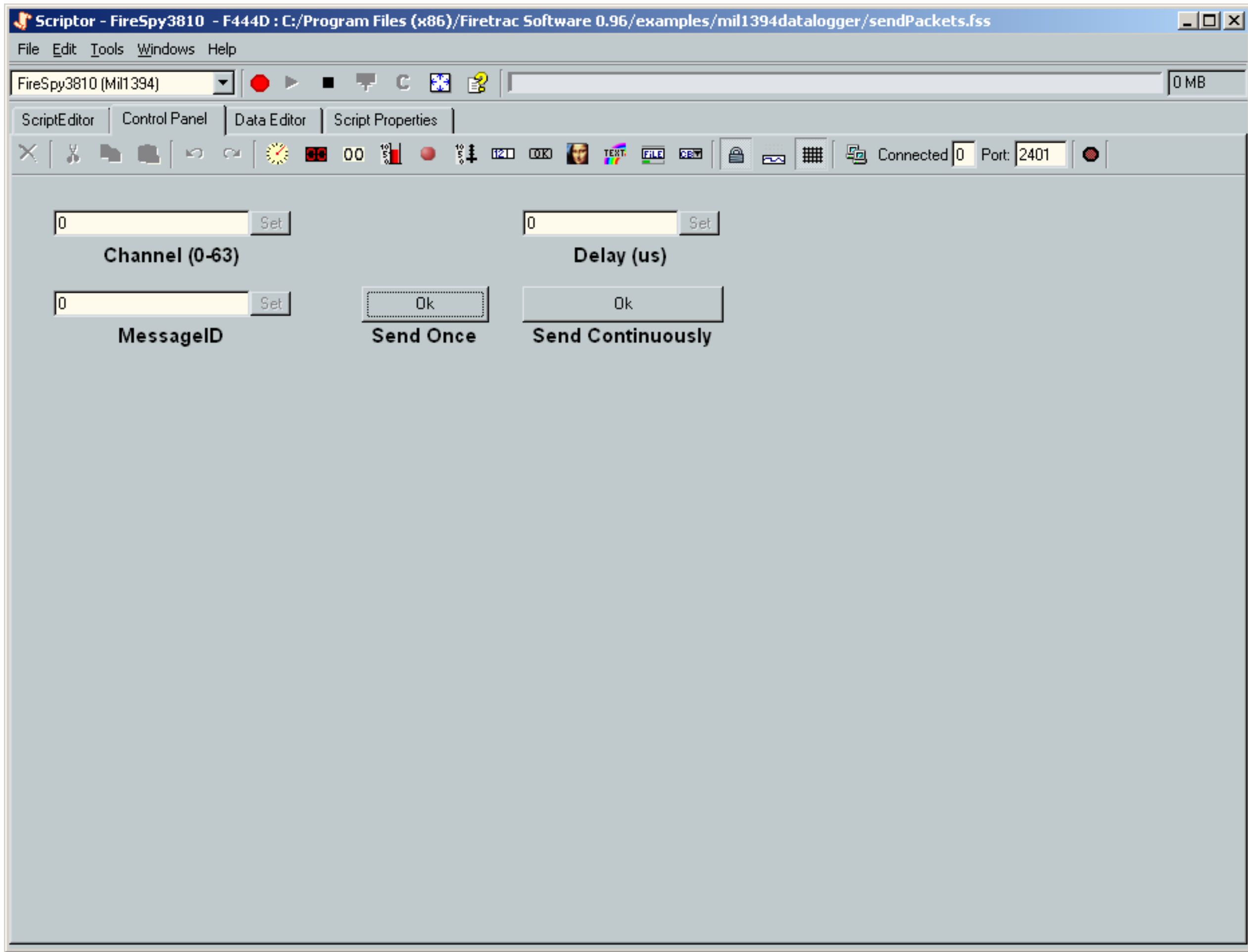
The Mil1394 Data Logger demonstrates functions defined in AS5643 Reception.

When the demo application starts it will open up-to three busses and configures them to start receiving specific messages. The console displays the following output.

```
Bus[0]: Opened
Bus[0]: Channelmask:1111100000000000000000000000000100000000000000000000000000000000
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:1111100000000000000000000000000100000000000000000000000000000000
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:1111100000000000000000000000000100000000000000000000000000000000
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
```

For each bus it can be seen from the output that some channels are set to AS5643 Reception mode (channelmask) and a couple of message filter entries have been set. Another device connected to the 1394 bus can be used to transmit packets that match these 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:

Copyright(C) 2008-2020, DapTechnology, Version 2/26/2020

```
Bus[0]: Opened
Bus[0]: Channelmask:1111100000000000000000000000000100000000000000000000000000000000
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:1111100000000000000000000000000100000000000000000000000000000000
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:1111100000000000000000000000000100000000000000000000000000000000
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.

12.4.2. 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 Mil1394 Receive Demo!!!
----------------------------------------------------------

Bus opened
Channelmask:1111100000000000000000000000000100000000000000000000000000000000
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 Mil1394 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.

 Copyright(C) 2008-2020, DapTechnology, Version 2/26/2020

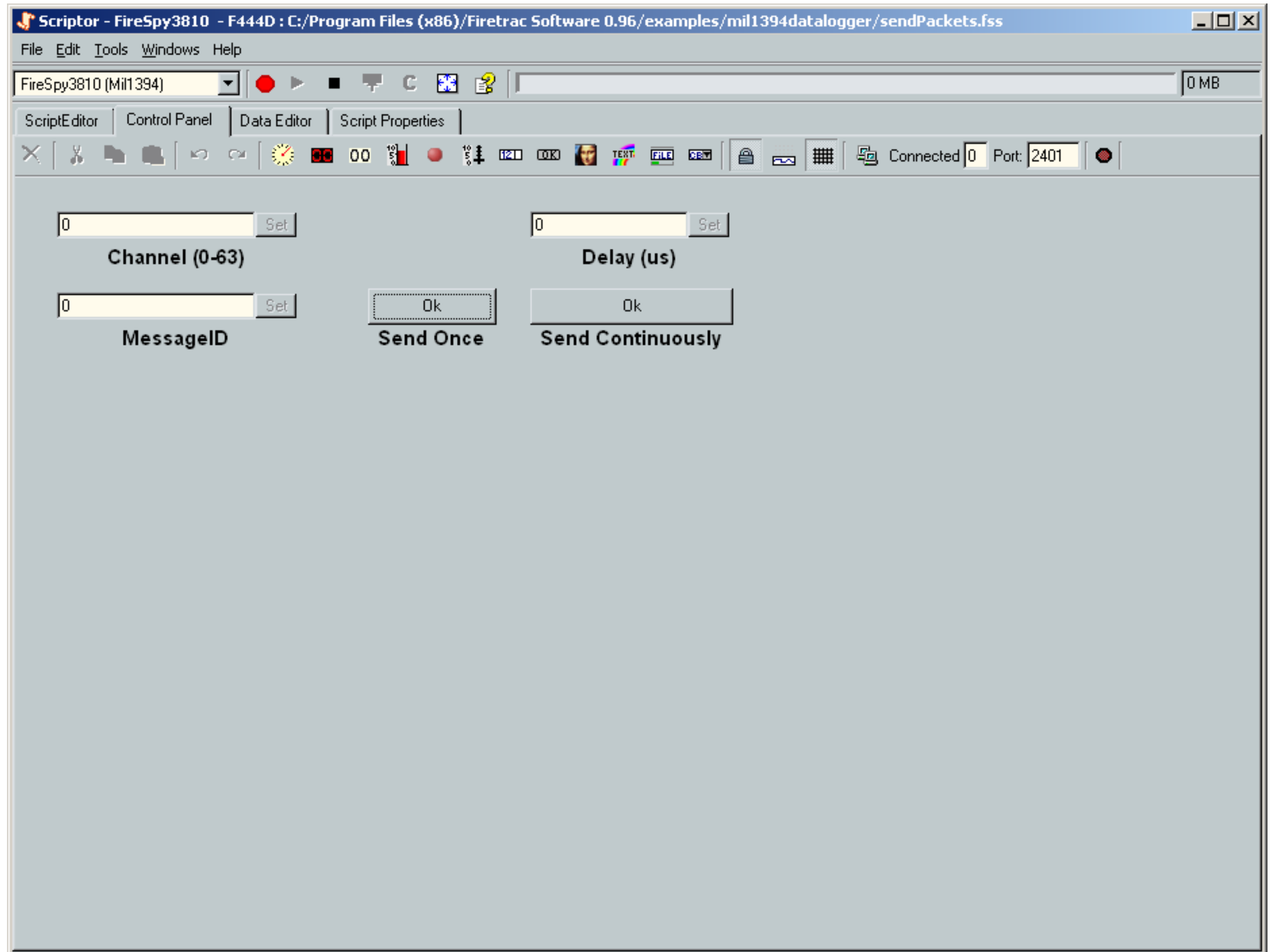
Other commands

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:

Copyright(C) 2008-2020, DapTechnology, Version 2/26/2020

```
---------------------------------------------------------
-- Welcome to FireLinkExtended Mil1394 Receive Demo!!!
---------------------------------------------------------

Bus opened
Channelmask:1111100000000000000000000000000100000000000000000000000000000000
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 Mil1394 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]
```


Copyright(C) 2008-2020, DapTechnology, Version 2/26/2020

12.5. AS5643 Transmission

12.5.1. 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 Mil1394 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

----------------------------------------------------------
---  Mil1394 Transmit Demo Menu                        ---
----------------------------------------------------------
?                             help (this menu)

--- Bus Commands ---

ob <device num>               open bus
cb                            close bus

--- Mil1394 Trm Single Mode ---

sms                           transmit single message
spms                          transmit single split message

--- Mil1394 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

--- Mil1394 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

--- Mil1394 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

--- Mil1394 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
```


Copyright(C) 2008-2020, DapTechnology, Version 2/26/2020

```
>
>
```

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 singel 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 bij 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

Copyright(C) 2008-2020, DapTechnology, Version 2/26/2020

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 wil 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

 Copyright(C) 2008-2020, DapTechnology, Version 2/26/2020

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.

12.6. Extensions

12.6.1. 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

Copyright(C) 2008-2020, DapTechnology, Version 2/26/2020

```
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 2.1.8

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)

 Copyright(C) 2008-2020, DapTechnology, Version 2/26/2020

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

- fxAsyRcvWaitSingleRequest 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)

Copyright(C) 2008-2020, DapTechnology, Version 2/26/2020