MCX API

for Sital Tester IP Core Device

Programmer and Reference Guide

Rev 2.27

March 2020

Table of Contents

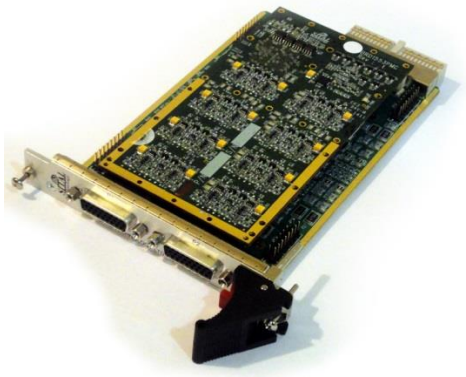
1 Introduction

1.1 Scope

This document is the Programmer and Reference Guide for programming with the MCX Tester's API for the the following protocols; Mil-Std-1553, WB194 (pp194), H009, RS422/485.
MCX API serves the following devices
MultiComBox

Grip2

PXI (cPCI)



1.2 Audience

This document assumes that the reader is familiar with the above specified protocols.

1.3 Related Documentation

TESTER1553 User Manual Ver62

1.4 Support

If you have any question or require further assistance, use any of the following methods to contact Sital customer support:

- By Email: support@sitaltech.com
- By Phone: +972-9-7633300
- By Fax: +972-9-7663394

1.5 About the MultiComBox

The MultiComBox™ system connects a standard USB 2.0 port to one or two dual-redundant Mil-Std-1553 buses.

Depending on the loaded firmware and configuration, MultiComBox™ can operate as Bus Controller (BC), Remote Terminal (RT), Monitor Terminal (MT), Multi-RT or as a full Mil-Std-1553 Tester.

Depending on model, MultiComBox also supports additional avionics protocols like WB194, H009 and Extended Bit Rate 1553 (EBR1553).

The system uses the high-speed USB 2.0 port from any desktop or laptop computer; loaded with Windows™ XP and higher.
As a 1553 bus tester, the MultiComBox™ unit provides full MIL-STD-1553B test, simulation and bus analysis capability in a compact self-contained unit. It supports concurrent Bus Controller (BC) and up to 31 Remote Terminals (RT) with Bus Monitor (MT). Full error injection capability is available in BC and RT modes, with full error detection in BC, RT and MT modes.

The unit is supplied with Dynamic Link Library (DLL), together with a Windows Graphical User Interface (GUI), which includes a Monitor and Simulator in the Composer application, providing a user-friendly software tool for all 1553 set-ups, simulation, data management and storage.

It is possible to create your own testing program, using the supplied DLL and its functions.

2 Concept & High Level Workflow

The MCX toolbox of API functions are intended to a SW developer who wishes to perform serial avionics bus communications through Sital Technology's MultiComBox in one of its hardware implementations, i.e. in USB, PCI, cPCI, PCIe and others.

The target of the SW development is to load the HW (Hardware) with the messages to transmit, and set it to go.
The HW supports a stack of message elements, where each element manages a single message, and works with a data buffer to transmit or receive. The SW equivalent of this HW is a BusList of Elements, and each Element points to a data buffer.

The API functions let you manage multiple devices.
You may also manage multiple BusLists each one having a different list of Elements.
You may create multiple DataBlocks, and map these DataBlocks to the Elements.
The Elements are mapped to the bus list, and the order of which is the order of the messages on the bus.

For the purpose of this guide;
BusList – relates to a Frame.
Element – relates to Message
DataBlock – relates to Message's data

At minimum:

1. Initialize a device.
2. Create a single BusList (Frame).
3. Create a single Element (Message).
4. Create a single DataBlock (Message's Data).
5. Map the DataBlock to the Element.
6. Map the Element to the BusList.
7. Start running the BusList on a device.
8. Collect run results.

It is advised to start the coding from examples which are provided.

2.1 Entities Relations

User can create stacks of BusLists (Frames), Elements (Messages) and DataBlocks (Messages Data).
Each entity is created by its designated create function (mcx_Create_BusList(…), mcx_Create_BusList_Element(…), mcx_Create_Element_DataBlock(…)).

BusLists
A BusList (Frame) entity can contain a single Element (Message) and up to 100 Elements.
BusList is the basic entity that a device is running on a 'mcx_Start(…)' call. When calling the mcx_Start, the specified BusList must exists and contain at list a single valid message.
Assigning an Element to a BusList is done by 'mcx_Map_Element_To_BusList(…)' function.
Un-assigning an Element from a BusList is done by 'mcx_UnMap_Element_From_BusList(…)' function.

Element
An Element (Frame) entity contains Commands (Command2 serves RT to RT Element structure) and Statuses. It can also point to a DataBlock. Once a Device is set to 1553 and PP194 Protocols, the Options of an Element specifies the Element's Protocol and selected Bus for running.

Bus	BusA = 0x80. BusB =0 (Default)

Pp194 messagetype	For pp194 – 0x0004. For 1553 - 0 (Default)

Mapping a DataBlock (Message's data) to an Element (Message) is done by 'mcx_Map_DataBlock_To_Element(…)' function.
Un-mapping a DataBlock from an Element is done by 'mcx_UnMap_DataBlock_From_Element (…)' function.
Note that Element structure, mapping and data assigning is identical for all supported Protocols. It is the User reposnsibility to map relevant data and data size, relevant Commands and so on according to the desired Protocl.

DataBlock
DataBlock (Message's data) entity contains an array of data words. The array and its size is assigned by the User.
The DataBlock entity structure and data is identical for all supported Protocols.
Note that it is the User's responsibility that DataBlock's data buffer and data buffer's size match the Protocol's limitations.

2.2 Cyber Attack Emulation

Note – Cyber Attack capabilities can be used via API function calls or by Composer. The Composer currently supports Attack type 1.

2.2.1 Introduction

Sital Technology's MultiComBox has been elevated to being able to emulate a cyber-attack for multi-drop bus protocols.

Development groups of aerospace products that would request to protect their products from cyber-attacks may use this emulation mode of operations to attack their product under development, and enhance their counter measures and firewalls.
For activating the following attack modes, refer to function mcx_SetCyberAttack(..).

2.2.2 Supported Attacks in Emulation Mode

2.2.2.1 Periodic Attack - After Period of Time

This type of attack would follow this algorithm:

1. Wait for predefined period of time.
2. Wait for bus idle on both bus A and B of the first message.
3. Transmit all frame messages to the bus based on the frame rate parameters.

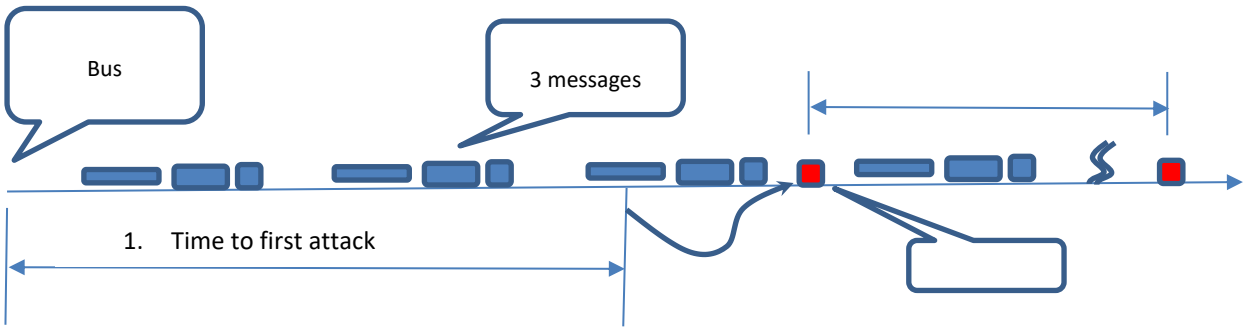
This attack allows the attacker to delay an attack, and then be persistent with it.

Resources: the frame length counter is used for the delay. 16 bits, two resolutions, one with LSB=65 milliseconds second with LSB=100 us. Maximum delay for LSB=65 ms is 65ms X 64K => 4295 seconds which are 71 minutes => 1 hour and 10 minutes.

Rate of attack: Message gap counter of all messages in the frame. This is typically 16 bits gap of micro seconds, up to a total of 65 ms.

Example attack: Wait for 10 minutes, and then transmit broadcast reset time tag every 65 milliseconds.

In this case there would be a frame with one message with message gap time set to 0xFFFF, and frame length counter set to 10x60x(1000/65) = 9230.



Set attack type to 1 to enable this type of attack.

2.2.2.2 Triggering Command

This type of attack would follow this algorithm:

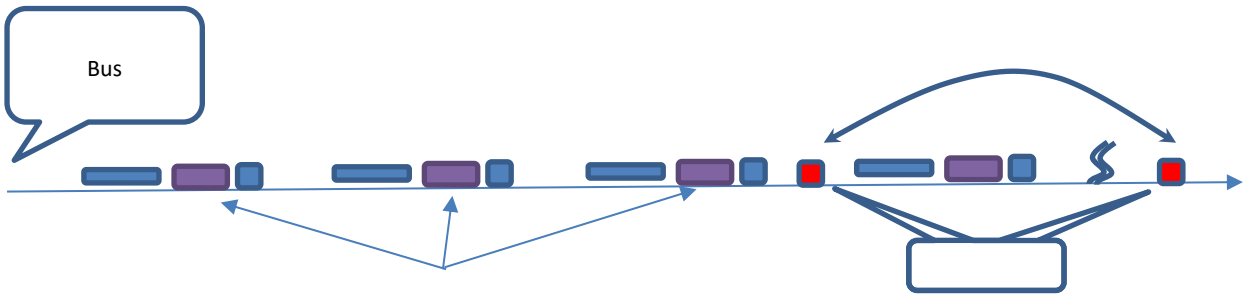
1. Wait for BC to transmit a particular command for N times.
2. Wait for destination buses idle, and transmit frame (without frame delay).

This attack allows the attacker to wait for a particular event on the 1553 bus in the form of a specific message, count N such occurrences, and then transmit the preplanned frame to the bus.

Resources: the frame length counter is used for N. N can be in the range of 0 to 64K. 0 would transmit without delay, 1 would indicate right after first occurrence of trigger message, 2 would wait for 2 such occurrences…

The Sync pattern register (0x46) defines the triggering command.

The attacker chooses to wait for an event such as a particular station (RT) becoming armed and replying to the bus. When that event happens, the attack includes transmitting predefined messages to that particular RT, to damage its operation.



•

Set attack type to 2 to enable this type of attack.

2.2.3 External Loopback Tests

The testers contain two types of external loopback tests; Device to Device and Bus A to Bus B within a single device.

External Loopback Test – Device to Device
General notes and requirements:

- Requires at least 2 devices – PMC and MultiComBox devices. This test does not apply to Grip2 tester devices.
- Required wiring scheme can be found in **{TBD}**
- Connection is done between 2 devices where Device 0 Bus A is connected to Device 1 Bus A and Device 0 Bus B is connected to Device 1 Bus B.

External Loopback Test – Single Device, Bus A to Bus B
General notes and requirements:

- Requires a single device – this test applies to all tester device types.
- Required wiring scheme can be found in **{TBD}**
- Connection is done within a single device between Bus A and Bus B.

2.3 Asynchronous message mode

As of March 20th 2018 a new Async message sending mode was added to the transmission capabilities of MCX BC.
This new mode of operation allows the controller to inject a new message instantaneously to the transmissions on the bus.
1. If the MCX is running and transmitting a bus list, say message #3, and the Async is initiated, then it will be transmitted ONCE after #3 ends, but before message #4.
2. If MCX is running, but it is in a passive phase, i.e., between two frames, the message would go out instantly with no delay, and the first message on the next frame might be delayed until the Async message has been transmitted.
3. If the MCX is in idle mode, i.e. no bus lists are running, the Async message would be instantly transmitted ONCE and MCX returns to idle mode.
4. If the MCX is in idle mode, i.e. no bus lists are running, the Async message would be instantly transmitted ONCE and if during transmission, bus-list transmission is engaged, MCX would start the bus-list transmission back-to-back with the Async message completion.

Some avionic systems make use of asynchronous messages, and the above method facilitates this mode to MCX.
The usage of Async message on IDLE simplifies the procedure for sending messages, and avoids using bus lists. The controller can transmit any message one after the other, and each can be different from the previous one. This mode of operation might be useful for some application that want the controller intimately managing the bus list. Please note that if message results are tested, there would be no bus activity periods between two messages, depending on this result analysis takes.
An Async message is defined by a standard message block format, but one which resides between address 0 and 7. Writing word 7 initiates the transmission.
A second Async 2 message is provided in addresses 8..F. Using these two async messages, one can work in pipe line mode, and achieve very high bus utilization even if using USB interface.

It is recommended that the data blocks and state blocks be located after the last block. Assuming there are 64 blocks supported, block location 65 and 66 should be used.

3 Configurations

Note – the minimal virtual requirement for local PC: a minimum of 8K (8065) MB.
In order to access and modify (Windows7):
My Computer --> Right click, Properties --> Advanced System Settings --> Advanced Tab--> Performance, press Settings button --> Advanced Tab --> Change button --> Initial Size 8065

3.1 Protocols & Modes

The device can be initialized to 1553 / PP194 mode (default) or H009 or Multiple RTs only mode.
The workflow in SW is the same for all modes of operation, the difference is the signaling on the bus wires for H009 or 1553/PP194.
For Multiple RT mode (MultiRT), the workflow is the same, except that when device Start command is issued, the MultiRT does not start transmission, but rather waits for an incoming message from an external bus controller.

When a message arrives and that RT (or RIU) is enabled for simulation, it is scanned in the bus list of elements, and if a match is found, that element will service the message, either transmitting data or receiving it. If no match is found zero data is transmitted, and no data is stored.

NOTE – on MultiRT only modes, the Tester disragards any bus selection (selected by the user) and answers to messages on both buses.

3.1.1 Frame Gap Mode

When setting a frame to run in a Gap mode, each of the frame messages can define a gap, which is the amount of microseconds from the beginning of this message to the beginning of the next message. A value could be in the range of 0 us to 64K us =~ 65 ms.

NOTE – this mode was the mode available in the Tester API and UI up to Release 4.3.0.32 (release on April 18, 2019).

3.1.2 Frame Rate Mode

For each message define its rate. Possible rates are
0 – skip this message.
1/1 – send every frame.
1/2 – send every second frame.
1/4 – send every forth frame.
1/8 – send every 8th frame.
:
1/N – Where N is a power of 2 and Nmax = 2^14, i.e., once every 16,384 frames.
15 – send only once. Core will change to 0 after transmission.
If two stack entries point to the same message, than the resulting rate of that message would be higher. The resulting rate would be the sum of both rates. For example 1/4 + 1/16 would be 5/16, which is almost 1/3 of the frame rate.
The HW core sequences the messages at lower rates, such that for each frame, only the 1/1 rate messages are transmitted along side only ONE slower rate messages. Such that 1/N (N>1) messages and 1/M (M>1, N<>M) messages are not transmitted in the same frame.

The HW core provides a register that indicates the frame number. After start command this frame counter starts counting up until the operation is stopped. The host can determine which message is transmitted on which frame using a simple equation. This will allow the host to update the transmitted data buffers on time.

It might happen that the transmission length in time of all messages of rate 1/1 and messages of lower rate in a specific frame sum up to a total length which is too long for the frame length. If one of the following frames is not crowded, the HW core supports message skew definition. A specific stack entry message can be skewed forward 1 to 15 frames ahead from its designated frame.

Example

Definition.
Typically MIL-STD-1553 (“MuxBus”) has a frame time definition.
The frame is a period of time, typically 10 or 20 milliseconds long.
Several messages are transmitted every frame. These messages manage the system.
In more complex MuxBus systems, not all messages are transmitted every frame. For example, the direction that a Radar Antenna is pointing at, should be transmitted every frame, i.e. 50 times a second, for display units to display target position. On the other hand, button position on one of the panels can be transmitted twice a second, since its not practical that the pilot would press that button faster than that.
The Operational Flight Program (OFP) programmer tailors the frames based on the Interface Control Document (ICD) that defines all message types required. In the ICD, each message is tagged by its usage rate.
Existing sequencing mechanisms.
For example, let’s define a system with a rate of 50 frames per second (50 Hz).
Message A50 and B50 are transmitted every frame.
Message E25 is transmitted every 2nd frame.

Message G125 is transmitted every 4th frame.

A possible order of the system would be:

Frame #1	Frame #2	Frame #3	Frame #4
A50 B50 E25	A50 B50 G125	A50 B50 E25	A50 B50

Frame #5 repeats frame #1 and so on…
As exampled, message G125 is transmitted in the frame that does not serve 25 Hz messages. This is done for load balancing as explained below.
Existing Bus Controllers (BC) use a stack of message entries to control their message sequencing. The Host CPU updates the stack, and initiates the bus controller to execute the messages automatically and autonomic.
Each Stack entry points to a location in its memory where the actual message command and words are stored.
Existing BCs define minor and major frames. In the above example, there is one Major Frame that is 4x20ms => 80ms, and 4 minor frames, each one take 20ms.
The stack would look like this: A50 B50 E25 A50 B50 G125 A50 B50 E25 A50 B50.

This list would be transmitted every 80ms, BUT there need to be a tool to force a gap between the end of E25 of frame #1 and A50 of frame #2 in order to make frame #1 20ms

in length. So, existing BCs also hold in their stack a ‘message-length’ parameter. So the stack would now look like (message-length in parenthesis):
A50 (1ms) B50 (1ms) E25 (18ms) A50 (1ms) B50 (1ms) G125 (18ms) …
Message-length parameter in stack is a way of composing a minor frame of 20ms.
This technique starts falling apart when lower rate messages have to be sequenced. If the slowest message is 50Hz/64 => 0.78 Hz then a complete list of more than a second has to be stacked. Most of the stack entries point to the very same message, it’s simply a very big stack.

3.1.3 Additions

As of Composer version 4.3.1.18, the following protocols and modes are available for licensed devices:

RS485, 4 channels (MCX B|C)

Arinc429, 4 channels (MCX C)

Scope (MCX C)

NOTE – The Scope feature can be coupled and used via Composer with any of the following – 1553 | Arinc429 | RS485.

NOTE II – The following protocols can be used individually via Composer – 1553 | Arinc429 | RS485.

3.2 Devices

3.2.1 MultiComBox

MulitComBox or MCX is a tester device. This device type requires loading a compatible .rbf firmware file. The file is loaded via USB on initialization.
Each MCX device contains 2 Mil-Std-1553 devices (Bus A and B for each of the 1553 devices) or a single EBR 1553 device or a H009 device.

3.2.2 PMC

PMC device is a PCI tester device that contains a static firmware version. Upgrading the firmware can be done by reflashing the PMC device via Sital's reflasher.
Each PMC contains 1 | 2 | 4 Mil-Std-1553 devices (Bus A and B for each of the 1553 devices) or a 1 | 2 EBR 1553 devices or a combination of 2 Mil-Std-1553 and 1 EBR 1553 devices.
The PMC can also contain 1 | 2 H009 devices (Bus A and Bus B).

In the configuration of 2x1553&1xEBR, the ordering of the devices is as follow; devices 0 and 1 are 1553 and device 2 is EBR device.

3.2.3 Grip2

Grip2 is a light weight tester device that contains a static firmware version. Upgrading the firmware can be done by reflashing the Grip2 device via Sital's reflasher.
The Grip2 contains a single Mil-Std-1553 device (Bus A and B).

3.2.4 PCI

As of McxAPI version 4.1.1.53, the McxAPI supports in 2 types of PCI devices: PCI 1553 (optionally with RS485) and Arinc429 (with 8Tx channels and 16 Rx channels or 12Tx channels and 16Rx channels).
For using the Arinc429 PCI card, use the '`mcx_A429_Pci_`' set of cuntions insead of '`mcx_A429_`' functions, described in the document

4 MultiComBox Hardware

4.1 USB Data

MultComBox™ connects to a host PC via a USB 2.0 connection. This connection uses high speed 480Mbps data transfer, and thus requires an appropriate cable. Please use only the provided USB cable. Using other USB cables may cause the unit not to work properly or not to work at all.

The USB cable should be connected to the USB connection and to any USB 2.0 port of your PC.

4.2 USB Connection

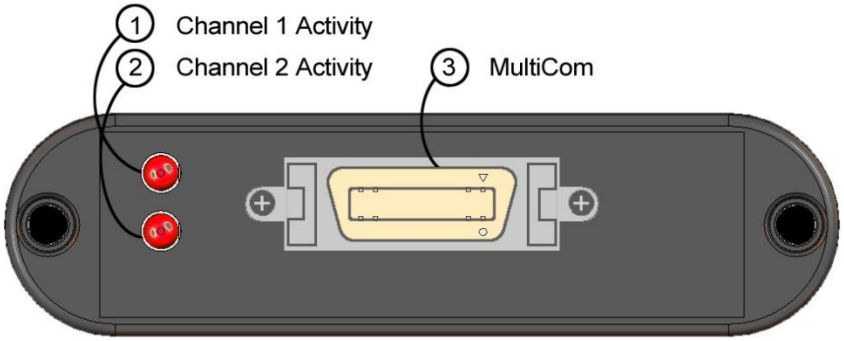
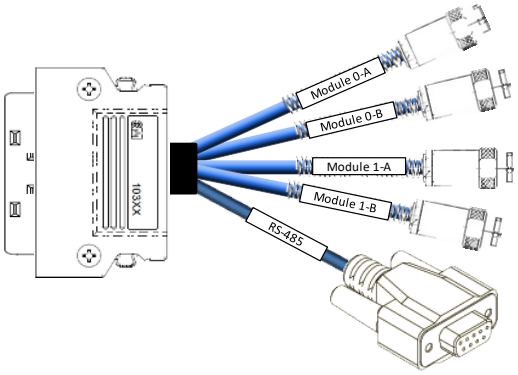
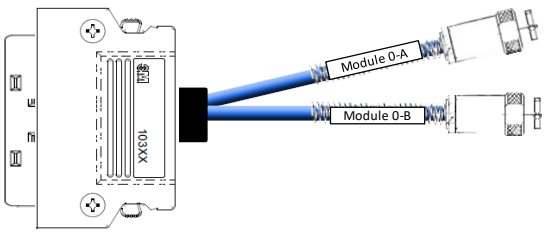


Figure 1: MultiCom Panel

The MultiCom Panel includes a 20 pin Mini-D-Ribbon connector (3) and two LED indicators (1 and 2). The MultiCom connectors in used for the 1553 and RS-485 connections and the LED indicators are used to monitor the activity in each 1553 module.

In addition, the unit comes with a cable assembly that, according to the configuration you purchased, contains 4 Triax connectors for 1553 and 1 female 9 pin D-type connector for RS 485, or 2 Triax connectors for a single Dual-Redundant 1553 channel. These connectors are marked with accordance to the bus they should be connected to.





1x 1553 channel

2 x 1553 and 4 x Serial channels

Figure 1:MultiCom Cable Assemblies

This cable should be connected to the “MultiCom” connector at the MultiComBox unit.

A MultiComBox unit contains an internal termination of 240 Ohm per each 1553 channel. Therefore, for a very simple test environment it is possible to connect the MultiComBox 1553 ports directly to the unit that is under test.

Figure 1:Direct connection to unit under test

Note that this is not a standard/recommended way to use Mil-Std-1553. Yet, for a simple test environment, if you plan to test your unit for its protocol capabilities, then this would be the simplest way to use MultiComBox.

If you wish to connect via a 1553 coupler, then a simple Mil-Std-1553 test environment is typically connected in the following way:

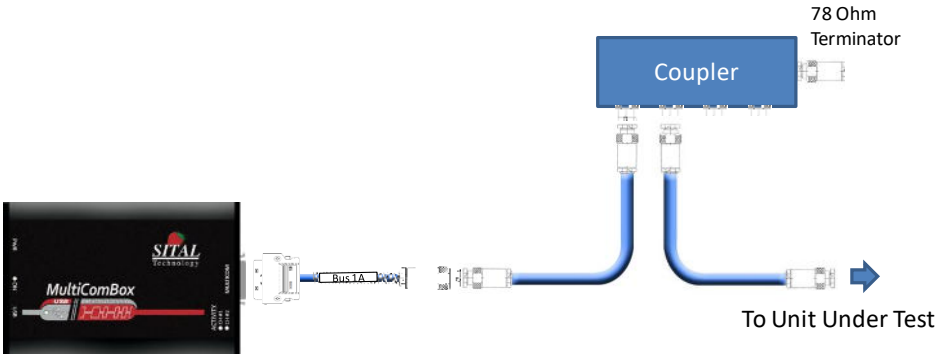


Figure 1:Mil-Std-1553 connection environment for single channel, via short stubs

When a long cable is required, or if more units need to connect to the bus, then it is required to connect more than one coupler. Such connection will typically be done in the following way:

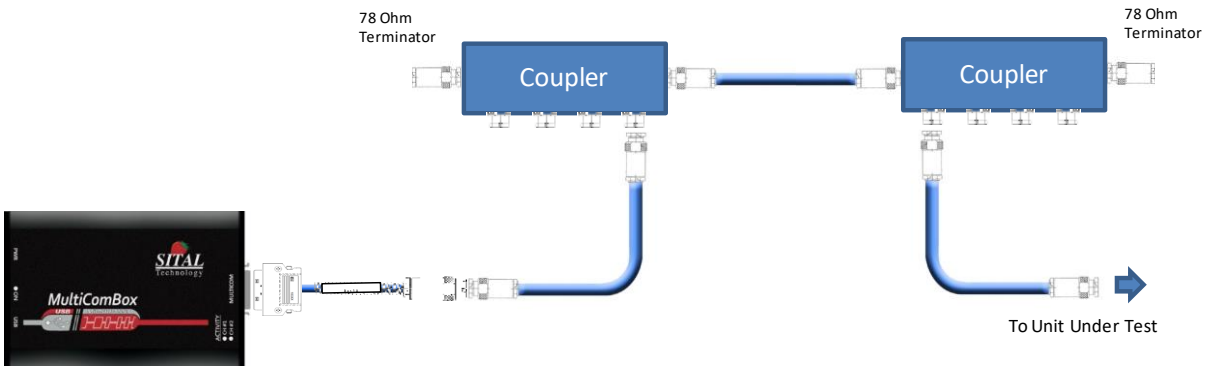


Figure 1:Mil-Std-1553 connection environment for single channel, via long stubs

In this example, only Bus 1A is used. When you need to connect other busses as well, for example Bus 2A, you would need to duplicate this connection, using an additional 1553 Coupler and cables.

MultiComBox™ enables you to connect up to two dual-redundant Mil-Std-1553 channels. These are marked at the cable assembly as “Module 0-A”, “Module 0-B”, “Module 1-A” and “Module 1-B”. When operating MuxSim™ and MuxMonitor™ software you will notice that the two channels are defined as Modules – “Module 0” and “Module 1”.

- **Module 0-A** represents BUS A in Module 0.
- **Module 0-B** represents BUS B in Module 0.
- **Module 1-A** represents BUS A in Module 1.
- **Module 1-B** represents BUS B in Module 1.

When operating a Dual-Redundant environment, you should not connect Module 0-A and Module 0-B on the same 1553 Coupler, nor Module 1-A and Module 1-B. There must be a complete duplication of the connection in the following way:

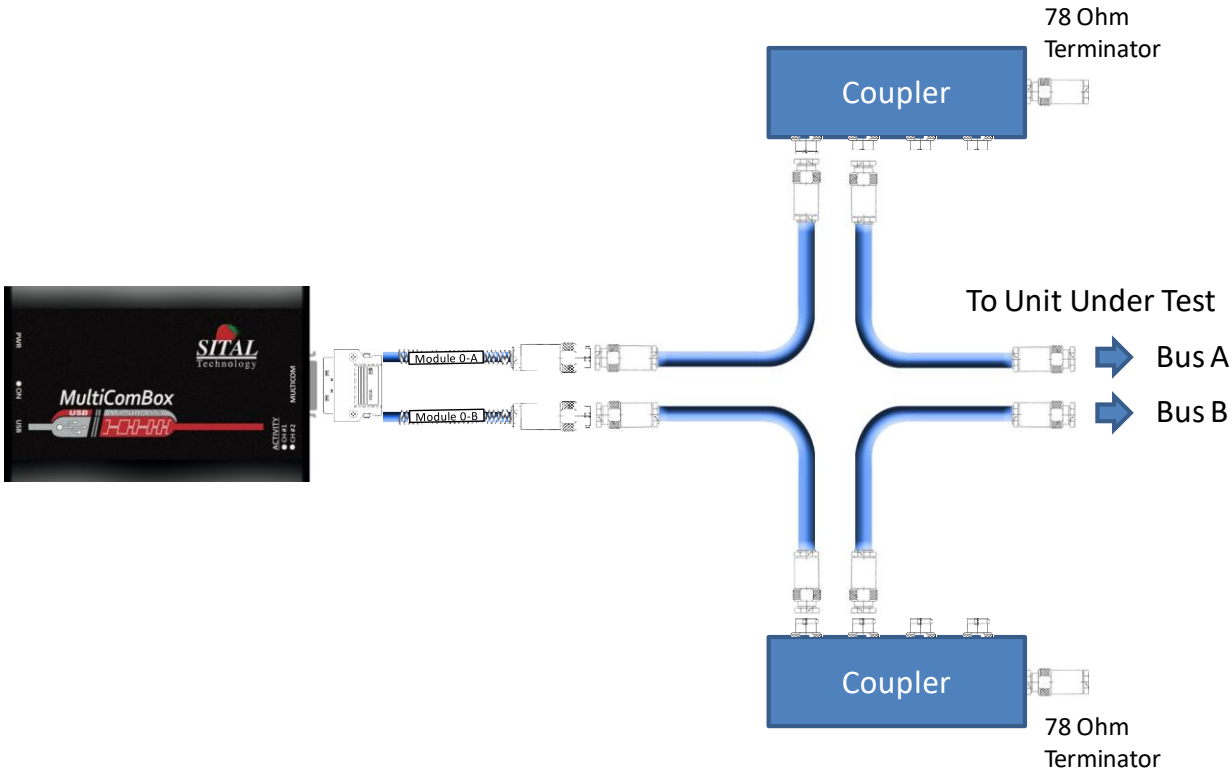


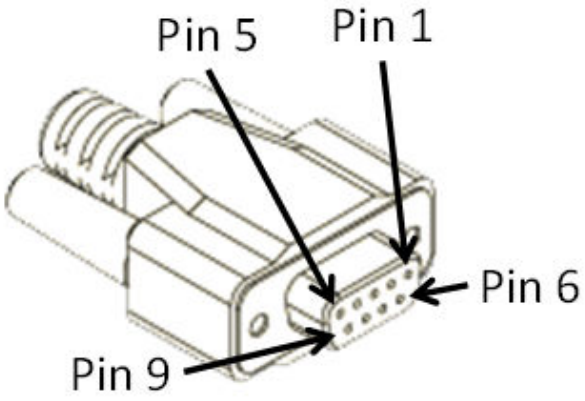
Figure 2: Dual Redundant 1553 test environment connection.

When using both 1553 Modules on the same bus, for example – it is possible to use Module 0 as BC, RT or MultiRT and Module 1 as Monitor Terminal, or vice-versa. In such case, channels A of both modules and channels B of both modules can be connected to the same couplers, and so channel A of Module 0 will be connected to channel A of Module 1, and channel B of Module 0 will be connected to channel B of Module 1.

4.3 RS485 (and EBR1553) Connection

MultiComBox™ enables up to 4 channels of RS-485. RS-485 is a two-wire, half-duplex, multipoint serial communications channel, that can be used for serial protocols or for discrete line as an event trigger.

The four channels of RS-485 are



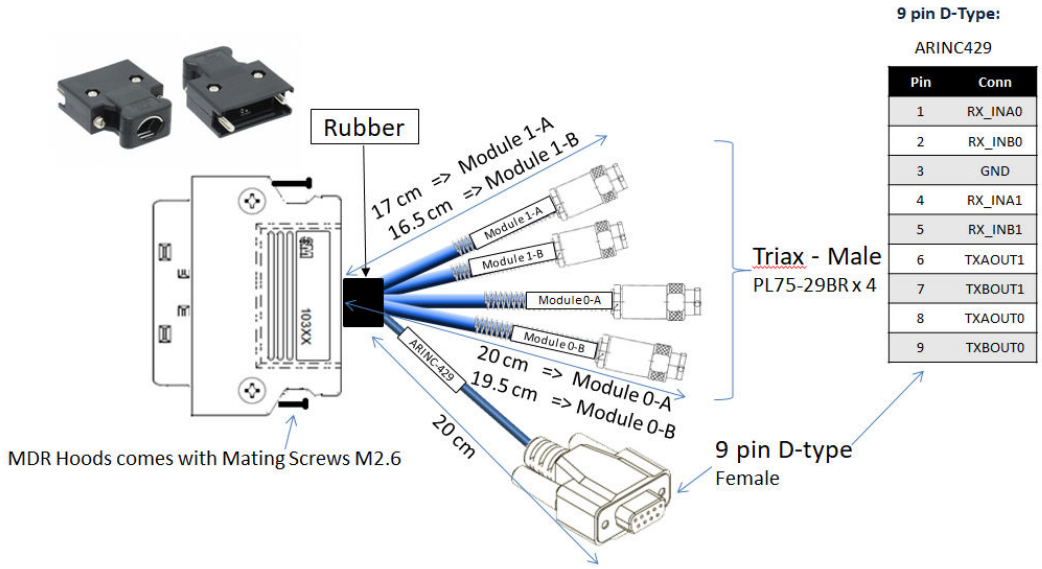
Pin	Conn
1	CH 1 +
2	CH 1 -
3	NC
4	CH 2 +
5	CH 2 -
6	CH 3 +
7	CH 3 -
8	CH 4 +
9	CH 4 -

available via the Female 9 pin D-type connector in the following manner:

Figure 1:9 Pin D-type for RS-485

These channels are also used for Extended Bit Rate 1553 (EBR1553) where applicable.

4.4 ARINC429 Connection



Pin	Conn
1	RX_INA0
2	RX_INB0
3	GND
4	RX_INA1
5	RX_INB1
6	TXAOUT1
7	TXBOUT1
8	TXAOUT0
9	TXBOUT0

4.5 PCI MIL-STD-1553 + RS485 Connection

	BRD1553PCI I/O Connector Pinout Mapping				
	1553 Bus	RS485	P1 pin #	1553 Description	RS485 Description
GROUP1	BUSA_P0	RS485_A0	1	MIL-STD-1533 bus P, positive signal	Non-inverting receiver input and non-inverting driver output
	BUSA_N0	RS485_B0	20	MIL-STD-1533 bus N, negative signal	Inverting receiver input and inverting driver output.
	BUSB_P0	NC	2	MIL-STD-1533 bus P, positive signal	Not Connected
	BUSB_N0	NC	21	MIL-STD-1533 bus N, negative signal	Not Connected
	BUSA_P1	RS485_A1	3	MIL-STD-1533 bus P, positive signal	Non-inverting receiver input and non-inverting driver output
	BUSA_N1	RS485_B1	22	MIL-STD-1533 bus N, negative signal	Inverting receiver input and inverting driver output.
	BUSB_P1	NC	5	MIL-STD-1533 bus P, positive signal	Not Connected
	BUSB_N1	NC	24	MIL-STD-1533 bus N, negative signal	Not Connected
	BUSA_P2	RS485_A2	6	MIL-STD-1533 bus P, positive signal	Non-inverting receiver input and non-inverting driver output
	BUSA_N2	RS485_B2	25	MIL-STD-1533 bus N, negative signal	Inverting receiver input and inverting driver output.
	BUSB_P2	NC	7	MIL-STD-1533 bus P, positive signal	Not Connected
	BUSB_N2	NC	26	MIL-STD-1533 bus N, negative signal	Not Connected
	BUSA_P3	RS485_A3	8	MIL-STD-1533 bus P, positive signal	Non-inverting receiver input and non-inverting driver output
	BUSA_N3	RS485_B3	27	MIL-STD-1533 bus N, negative signal	Inverting receiver input and inverting driver output.
	BUSB_P3	NC	9	MIL-STD-1533 bus P, positive signal	Not Connected
	BUSB_N3	NC	28	MIL-STD-1533 bus N, negative signal	Not Connected
GROUP2	BUSA_P4	RS485_A4	11	MIL-STD-1533 bus P, positive signal	Non-inverting receiver input and non-inverting driver output
	BUSA_N4	RS485_B4	29	MIL-STD-1533 bus N, negative signal	Inverting receiver input and inverting driver output.
	BUSB_P4	NC	12	MIL-STD-1533 bus P, positive signal	Not Connected
	BUSB_N4	NC	30	MIL-STD-1533 bus N, negative signal	Not Connected
	BUSA_P5	RS485_A5	13	MIL-STD-1533 bus P, positive signal	Non-inverting receiver input and non-inverting driver output
	BUSA_N5	RS485_B5	31	MIL-STD-1533 bus N, negative signal	Inverting receiver input and inverting driver output.
	BUSB_P5	NC	14	MIL-STD-1533 bus P, positive signal	Not Connected
	BUSB_N5	NC	32	MIL-STD-1533 bus N, negative signal	Not Connected
	BUSA_P6	RS485_A6	15	MIL-STD-1533 bus P, positive signal	Non-inverting receiver input and non-inverting driver output
	BUSA_N6	RS485_B6	33	MIL-STD-1533 bus N, negative signal	Inverting receiver input and inverting driver output.
	BUSB_P6	NC	17	MIL-STD-1533 bus P, positive signal	Not Connected
	BUSB_N6	NC	35	MIL-STD-1533 bus N, negative signal	Not Connected
	BUSA_P7	RS485_A7	18	MIL-STD-1533 bus P, positive signal	Non-inverting receiver input and non-inverting driver output
	BUSA_N7	RS485_B7	36	MIL-STD-1533 bus N, negative signal	Inverting receiver input and inverting driver output.
	BUSB_P7	NC	19	MIL-STD-1533 bus P, positive signal	Not Connected
	BUSB_N7	NC	37	MIL-STD-1533 bus N, negative signal	Not Connected
	GND	GND	4	Ground (Can be left uncconected)	Ground
	GND	GND	10	Ground (Can be left uncconected)	Ground
	GND	GND	16	Ground (Can be left uncconected)	Ground
	GND	GND	23	Ground (Can be left uncconected)	Ground
	IRIG_IN/NC		34	IRIG_B or Ground (default=NC-Can be left uncconected)	IRIG_B or Ground (default=NC-Can be left uncconected)
	STANDARD CONNECTIVITY COULD NOT MIX 1553 AND RS485 SIGNALS IN THE SAME GROUP				

4.6 PCI ARINC429 Connection

BRD429PCI-12-16 I/O Connector Pinout Mapping					
Pin #	**Pin Name**	**Pin #**	**Pin Name**	**Pin #**	**Pin Name**
62	arinc_txa0	42	arinc_rxa9	21	arinc_rxa0
61	arinc_txb0	41	arinc_rxb9	20	arinc_rxb0
60	arinc_txa1	40	cgnd	19	arinc_rxa1
59	arinc_txb1	39	arinc_rxa10	18	arinc_rxb1
58	arinc_txa2	38	arinc_rxb10	17	arinc_rxa2
57	arinc_txb2	37	cgnd	16	arinc_rxb2
56	arinc_txa3	36	arinc_rxa11	15	arinc_rxa3
55	arinc_txb3	35	arinc_rxb11	14	arinc_rxb3
54	arinc_txa4	34	cgnd	13	arinc_rxa4
53	arinc_txb4	33	arinc_rxa12	12	arinc_rxb4
52	arinc_txa5	32	arinc_rxb12	11	arinc_rxa5
51	arinc_txb5	31	cgnd	10	arinc_rxb5
50	arinc_txa6	30	arinc_rxa13	9	arinc_rxa6
49	arinc_txb6	29	arinc_rxb13	8	arinc_rxb6
48	arinc_txa7	28	cgnd	7	arinc_rxa7
47	arinc_txb7	27	arinc_rxa14	6	arinc_rxb7
46	arinc_txa8	26	arinc_rxb14	5	arinc_rxa8
45	arinc_txb8	25	arinc_rxa15	4	arinc_rxb8
44	arinc_txa9	24	arinc_rxb15	3	cgnd
43	arinc_txb9	23	arinc_txa10	2	arinc_txa11
		22	arinc_tx10	1	arinc_tx11

5 API Reference

5.1 mcx_Initialize

```
INT16 mcx_Initialize (
    UINT16  deviceId
    UINT16  protocol
)
```

Parameters

deviceId	Unique Device ID 0 - (sitalMaximum_DEVICES - 1)
protocol	User Code option for setting the device to work in a protocol/state. The following definitions can be found in McxAPI.h: // UserCode Options MIL_STD_1553_AND_PP194 0x0000 H009 0x0001 MultiRT 0x0002 MIL_STD_1553 0x0004 EBR_1553 0x0008 DIGIBUS_F16 0x0016

Description

Mode: Ready

This function initializes device to a protocol and state according to initialization protocol parameter. Release any past allocations of device memory and pointers.

This function loads the FPGA to the MCX Tester device. Loading FPGA operation may last up to 8-10 seconds. The FPGA loading action occurres on the first mcx_Initialize. Once loaded successfully, re-using mcx_Initialize will use previously loaded FPGA.

Mode: Runtime

Since this is a “configurations and settings” function, it stops the device activities and data transfer.

5.2 mcx_SetFpgaFileDirectory

```
INT16 mcx_SetFpgaFileDirectory    (
                                    UINT16      deviceId
                                    char *      fpgaFileDir
                                  )
```

Parameters

deviceId	Unique Device ID 0 - (sitalMaximum_DEVICES - 1)
fpgaFileDir	A string representing

Description

Mode: Ready & Runtime
This function changes the default FPGA File directory from McxAPI.dll location to the specified folder in fpgaFileDir parameter.
If an existing and valid directory is specified, the FPGA file is loaded from the new directory location for this device.

Note

This function must be called prior to 'mcx_Initialize(…)' in order to take effect.

5.3 mcx_EnableRts

```
INT16 mcx_EnableRts (
    UINT16  deviceId
    UINT32  rtsVector
)
```

Parameters

deviceId	Unique Device ID 0 - (sitalMaximum_DEVICES - 1)
rtsVector	RTs vector

Description

Mode: Ready
This function enable Remote Terminal simulation for up to 31 RTs according to the specified bits in the rtsVector.
Each call of this function overwrites the enabled RTs and configers it as specified in the vector parameter.
The device ID must be within the allowed range (0 - sitalMaximum_DEVICES - 1) and in Ready Mode.

Note I: This function supports Mil-Std-1553 and H009 protocols. For pp194 (WB194) protocol, refer to mcx_EnableRius function.

Mode: Runtime
This function is not supported in Runtime mode. In case of calling this function in Running mode an error is returned: STL_ERR_BUSLIST_IS_RUNNING.

5.4 mcx_Get_EnabledRts

```
INT16 mcx_Get_EnabledRts (
    UINT16      deviceId
    UINT32*     rtsVector
)
```

Parameters

deviceId	Unique Device ID 0 - (sitalMaximum_DEVICES - 1)
rtsVector	Pointer to RTs vector

Description

Mode: Ready & Runtime
This function returns a list of Remote Terminal simulated for up to 31 RTs according to the specified bits in the rtsVector.
The device ID must be within the allowed range (0 - sitalMaximum_DEVICES - 1) and in Ready Mode.

5.5 mcx_EnableRius

```
INT16 mcx_EnableRts (
    UINT16 deviceId
    UINT16 riusVector
)
```

Parameters

deviceId	Unique Device ID 0 - (sitalMaximum_DEVICES - 1)
riusVector	RIUs vector

Description

Mode: Ready
This function enable Remote Terminal simulation for up to 16 RIUs according to the specified bits in the riusVector.
Each call of this function overwrites the enabled RIUs and configers it as specified in the vector parameter.
The device ID must be within the allowed range (0 - sitalMaximum_DEVICES - 1) and in Ready Mode.

Note I: This function supports pp194 (WB194) protocol. For Mil-Std-1553 and H009 protocols, refer to mcx_EnableRts function.

Mode: Runtime
This function is not supported in Runtime mode. In case of calling this function in Running mode an error is returned: STL_ERR_BUSLIST_IS_RUNNING.

5.6 mcx_Create_BusList

```
INT16 mcx_Create_BusList (
    UINT16 deviceId
    UINT16 busList
)
```

Parameters

deviceId	Unique Device ID 0 - (sitalMaximum_DEVICES - 1)
busList	Unique ID of BusList 0 - (MAX_BUSLISTS - 1)

Description

Mode: Ready
This function creates a BusList with a unique ID.
The ID must be within the allowed range (0 – sitalMaximum_BusLists -1). If it is not a valid BusList ID an error is returned:
Note: In case that this ID already exists, an error is returned. In order re-create a BusList with existing ID, the user must delete the BusList using 'mcx_Delete_BusList' fisrt.

Mode: Runtime
While in runtime mode, this function creates only new BusLists with new IDs. In case that the specified ID is mapped and running an error is returned.

5.7 mcx_Create_BusList_Element

```
INT16 mcx_Create_BusList_Element (
                                    UINT16      deviceId
                                    UINT16      element
                                    UINT16      command
                                    UINT16      options
                                    UINT16      Command2
                                    UINT16      StatusWord1
                                    UINT16      StatusWord2
                                   )
```

Parameters

deviceId Unique Device ID 0 - (sitalMaximum_DEVICES - 1)

element Unique ID of Element 0 - (MAX_ELEMENTS - 1)

command Unique, MIL-STD-1553 | H009 | pp194 Command word that this Element services

Options Element's optional configuration parameter. The option is a logic OR combination of the following configs:

Bus	BusA = 0x80. BusB =0
Pp194 messagetype	For pp194 – 0x0004. For 1553 - 0

Command2 Unique, MIL-STD-1553 | H009 | pp194 Command word that this Element services. This Command is relevant for RT to RT and RT to Broadcast only as second RT's Command

StatusWord1 First status for simulated (Multi) RT / RIU responses.

StatusWord2 Second status for simulated (Multi) RT / RIU responses.

Description

Mode: Ready
Create an Element with a unique ID. The command word specified gets serviced by this Element.
In case of RT to RT or RT to Broadcast, the second RT's command is specified in

The ID must be within the allowed range (0 – sitalMaximum_ Elements -1). If it is not a valid Element ID, an error is returned:
Note: In case that this ID already exists, an error is returned. In order re-create an Element with existing ID, the user must delete the BusList using 'mcx_Delete_BusList_Element' fisrt.

Mode: Runtime

While in runtime mode, this function creates only new Elements with new IDs. In case that the specified ID is mapped and running an error is returned.

Notes

Note I - that specifying statuses on StatusWord1/2 it is injected and applies only to a situation when the Tester is BC and the RT/s is set to be simulated by the Tester.

The use case for it is when testing a monitor unit (UUT) that is connected to the Tester and you want to verify various statuses are received ok on the monitor's side.

Note II -

Field	Description
Command	MIL-STD-1553 command word #1. Bits 11–15: 00000 – 11110: Case – One command word: RT ID number of the transmitting RT (data source) Case – Two command words: RT ID number of the receiving RT (data sink) 11111: Broadcast (BCST) Bit 10: 1 – transmit command 0 – receive command Bits 5 – 9: 00001 – 11110: Sub-address 00000, 11111: Mode command Bits 0 – 4: For either the count of data words or the mode code, depending on the value in bits 5-9.
wCommand2	MIL-STD-1553 command word #2 (for RT-to-RT type and RT-to-broadcast type only) Bits 11–15: RT ID number of the transmitting RT (data source) Bit 10 must be 1 See the *table in the Message Formats* section of the Sital Tester-1553 User Manual, which lists the possible commands and shows where a second RT participates.
StatusWord1	Transmission status, if any, with most recent transmission of this message. This status word is filled only where relevant.
StatusWord2	Reception status, if any, with most recent transmission of this message. This status word is filled only where relevant.

5.8 mcx_Create_BusList_Element1

```
INT16
mcx_Create_BusList_Element1     (
                                  UINT16      deviceId
                                  UINT16      element
                                  UINT16      command
                                  UINT16      options
                                )
```

Parameters

deviceId Unique Device ID 0 - (sitalMaximum_DEVICES - 1)

element Unique ID of Element 0 - (MAX_ELEMENTS - 1)

command Unique, MIL-STD-1553 | H009 | pp194 Command word that this Element services

options Element's optional configuration parameter. The option is a logic OR combination of the following configs:

Bus	BusA = 0x80. BusB =0
Pp194 messagetype	For pp194 – 0x0004. For 1553 - 0

Description

Mode: Ready
Create an Element with a unique ID. The command word specified gets serviced by this Element.

The ID must be within the allowed range (0 – sitalMaximum_ Elements -1). If it is not a valid Element ID, an error is returned:
Note: In case that this ID already exists, an error is returned. In order re-create an Element with existing ID, the user must delete the BusList using 'mcx_Delete_BusList_Element' fisrt.
Note II – this function provides the capability to create a BC2RT or RT2BC commands. In order to create an RT2RT commands, use mcx_Create_BusList_Element(..) function.

Mode: Runtime
While in runtime mode, this function creates only new Elements with new IDs. In case that the specified ID is mapped and running an error is returned.

Notes

Field	Description
Command	MIL-STD-1553 command word #1. Bits 11–15: 00000 – 11110: Case – One command word: RT ID number of the transmitting RT (data source) Case – Two command words: RT ID number of the receiving RT (data sink) 11111: Broadcast (BCST) Bit 10: 1 – transmit command 0 – receive command Bits 5 – 9: 00001 – 11110: Sub-address 00000, 11111: Mode command Bits 0 – 4: For either the count of data words or the mode code, depending on the value in bits 5-9.

5.9 mcx_Create_Element_DataBlock

```
INT16
mcx_Create_Element_DataBlock    (
                                  UINT16      deviceId
                                  UINT16      dataBlock
                                  UINT16      dataBlockMode
                                  UINT16  *   buffer
                                  UINT16      bufferSize
                                )
```

Parameters

deviceId	Unique Device ID 0 - (sitalMaximum_DEVICES - 1)
dataBlock	Unique ID of DataBlock 0 - (MAX_DATABLOCKS- 1)
dataBlockMode	DataBlockMode_64_WORDS
buffer	A pointer to an array of data words to be copied into the new data block, or NULL if isn't required
bufferSize	The buffersize (unsigned int 16)

Description

Mode: Ready
Create a DataBlock with unique ID. The DataBlockMode specified will set the type.
Currently, the MCX API support a single data block mode of 64 words for all message types. The DataBlockMode_64_WORDS(0x0010) can be found in the API's header file.

The ID must be within the allowed range (0 – sitalMaximum_DataBlock -1). If it is not a valid DataBlock ID, an error is returned:
Note: In case that this ID already exists, an error is returned. In order re-create a DataBlock with existing ID, the user must delete the DataBlock using 'mcx_Delete_Element_DataBlock' fisrt.
Mode: Runtime
While in runtime mode, this function creates only new DataBlock with new IDs. In case that the specified ID is mapped and running an error is returned

Limitations

The User must allocate a Buffer Size of 64 words in order to match the supported DataBlockMode.

5.10 mcx_Map_DataBlock_To_Element

```
INT16
mcx_Map_DataBlock_To_Element    (
                                  UINT16      deviceId
                                  UINT16      element
                                  UINT16      dataBlock
                                )
```

Parameters

deviceId	Unique Device ID 0 - (sitalMaximum_DEVICES - 1)
element	Unique ID of Element 0 - (MAX_ELEMENTS - 1)
dataBlock	Unique ID of DataBlock 0 - (MAX_DATABLOCKS - 1)

Description

Mode: Ready
This functions maps a DataBlock by its unique ID to an Element by its unique ID.
In case that DataBlock or Element are null, does not exist or not created, error code will be returned.

Note: Datablock can be mapped once to an Element. In case that the user is mapping a Datablock (by unique ID) that is already mapped to this Element, an error is returne.

Mode: Runtime
This function cannot run while the specified DataBlock OR Element are in use by the HW.

5.11 mcx_Map_Element_To_BusList

```
INT16
mcx_Map_Element_To_BusList      (
                                  UINT16      deviceId
                                  UINT16      busList
                                  UINT16      element
                                )
```

Parameters

deviceId	Unique Device ID 0 - (sitalMaximum_DEVICES - 1)
busList	Unique ID of BusList 0 - (MAX_BUSLISTS - 1)
element	Unique ID of Element 0 - (MAX_ELEMENTS - 1)

Description

Mode: Ready
This functions maps a Element by its unique ID to a BusList by its unique ID.
In case that BusList or Element are null, does not exist or not created, error code will be returned.

Note: Element can be mapped once to a BusList. In case that the user is mapping an Element (by unique ID) that is already mapped to this BusList, an error is returned.

Mode: Runtime
This function cannot run while the specified BusList OR Element are in use by the HW.

5.12 mcx_Start

```
INT16 mcx_Start (
    UINT16 deviceId
    UINT16 busList
    UINT16 numberOfIterations
)
```

Parameters

deviceId	Unique Device ID 0 - (sitalMaximum_DEVICES - 1)
busList	Unique ID of BusList 0 - (MAX_BUSLISTS - 1)
numberOfIterations	Number of iterations/cycles for the bus list. 0 signals the device to run forever.

Description

Mode: Ready
This function sets the specified device to start handling messages.
Setting the device to numberOfIterations will apply #N cycles for the specified BusList. In case of setting this parameter to 0, the bus list will iterate for ever (until stopped via mcx_Stop(..)).
The states of the device, bus, bus's Elements and datablock are set to Running on success.
This function assumes that device was initialized by 'mcx_Initialize'.
In Case that specified device was not initialized as described, an error code will be returned.
In case that the HW state is not set to Running the following error is returned: STL_ERR_START_RUN_FAILED

Mode: Runtime
This function cannot run while the specified Device is in use by the HW (device is not in Ready Mode). In such a case, an error is returned: STL_ERR_INVALID_STATE

NOTE – Buslist in this function settings runs on frame GAP mode. For using RATE mode, use 'mcx_Start_RateMode(..)' function.
For elaborated info about the difference between Gap and Rate modes, see section 3.1 of this document.

5.13 mcx_Start_RateMode

```
INT16 mcx_Start_RateMode (
    UINT16 deviceId
    UINT16 busList
    UINT16 numberOfIterations
)
```

Parameters

deviceId	Unique Device ID 0 - (sitalMaximum_DEVICES - 1)
busList	Unique ID of BusList 0 - (MAX_BUSLISTS - 1)
numberOfIterations	Number of iterations/cycles for the bus list. 0 signals the device to run forever.

Description

Mode: Ready
This function sets the specified device to start handling messages.
Setting the device to numberOfIterations will apply #N cycles for the specified BusList. In case of setting this parameter to 0, the bus list will iterate for ever (until stopped via mcx_Stop(..)).
The states of the device, bus, bus's Elements and datablock are set to Running on success.
This function assumes that device was initialized by 'mcx_Initialize'.
In Case that specified device was not initialized as described, an error code will be returned.
In case that the HW state is not set to Running the following error is returned: STL_ERR_START_RUN_FAILED

Mode: Runtime
This function cannot run while the specified Device is in use by the HW (device is not in Ready Mode). In such a case, an error is returned: STL_ERR_INVALID_STATE

NOTE – Buslist in this function settings runs on frame Rate mode. For using GAP mode, use 'mcx_Start (..)' function.
For elaborated info about the difference between Gap and Rate modes, see section 3.1 of this document.

5.14 mcx_Stop

```
INT16 mcx_Stop (
    UINT16 deviceId
)
```

Parameters

deviceId	Unique Device ID 0 - (sitalMaximum_DEVICES - 1)

Description

Mode: Ready
This function cannot run while the specified Device is in Ready mode. In such a case, an error is returned: STL_ERR_INVALID_STATE

Mode: Runtime
This function sets the specified device to stop running.
On success, the states of the device, bus, bus's Elements and DataBlock are set to Ready.
In case that the HW fails to stop, a stop retry occurs after 1 mS. If this retry fails an error returns: STL_ERR_STOP_RUN_FAILED.

5.15 mcx_Stop2

```
INT16 mcx_Stop2 (
    UINT16 deviceId
)
```

Parameters

deviceId	Unique Device ID 0 - (sitalMaximum_DEVICES - 1)

Description

Mode: Ready
This function cannot run while the specified Device is in Ready mode. In such a case, an error is returned: STL_ERR_INVALID_STATE

Mode: Runtime
This function sets the specified device to stop running.
On success, the states of the device, bus, bus's Elements and DataBlock are set to Ready.
In case that the HW fails to stop, a stop retry occurs after 1 mS. If this retry fails an error returns: STL_ERR_STOP_RUN_FAILED.
In this function, no hardware reset applied, should work on MultiRT in 100% bus utilization.

5.16 mcx_Get_Element_Results

```
INT16 mcx_Get_Element_Results (
                                UINT16      deviceId
                                UINT16      busList
                                UINT16      elementIndex
                                UINT16 *    blockStatusWord
                                WORD *      buffer
                                UINT16      bufferSize
                                UINT16 *    status1
                                UINT16 *    status2
                                UINT16 *    tag
                               )
```

Parameters

deviceId	Unique Device ID 0 - (sitalMaximum_DEVICES - 1)
busList	Unique ID of BusList 0 - (MAX_BUSLISTS - 1)
elementIndex	Element's Index (not Element ID) within the busList
blockStatusWord	Returns Element/Message findings; see Message State table below.
buffer	Returns the buffer according to requested size of data transmitted.
bufferSize	Buffer size for the retuned buffer.
status1	The first status that was received from a real non-simulated RT
status2	Only in RT2RT command format, the second status (Rx Status) that was received from a real non-simulated RT
tag	This parameter works in either one of two modes of operations: the message Rate mode, or the message Gap mode. In Gap mode - The 16 LSBs of the 32 bit time tag counter are stored here when the message was launched by the core. In Rate mode – A frame counter is incremented by 1 at EOF. This frame counter value is stored in this entry when the message is transmitted

Description

Mode: Ready

This function gets the results of a transmission of a specific Element within a specified BusList. Message results comprise the message words that were actually transmitted along the internal bus together with the statistics (diagnostics) of the transmitted message. The diagnostics include an indicator of whether the message transmission was successful, status words, the data payload that was actually transmitted on the bus.
The difference between this function and the Word Monitor family of functions is that the Word Monitor sits on the bus in the Tester device and simply records all the words that go by; the Word Monitor has no concept

of BusLists or Elements. This function, on the other hand, returns a specific Element's results from the specified BusList.

Notes

This function requires an Element's index parameter rather than its ID since the same Elements may appear more than once within a single frame.

Message State Table

	Name	Bit	Description
0	Time Tag Word 16 LSBs. (Gap mode)	15..0	16 LSBs of the real time counter. Written by core when the message started.
0	Frame Number (Rate mode)	15..0	Frame number when this message was transmitted. Frame number is incremented every EOF. It is recommended to init this value to 0xFFFF before run.
1	Message findings	15	End Of Message – Set to '1' by the core when the message has been complete.
		14	Start Of Message - Set to '1' by the core when the message has been started. In most cases, this bit is stuck at '1' after end of message if there is a 1553 bus-coupling problem.
		13	'0' – Was sent on Bus A. '1' – Was sent on Bus B.
		12	'1' – Error was found in the message. Bits 10, 9, 8, 3, 2, 1, 0 indicate cause of error.
		11	Status Set. One of the status bits (excluding BCST bit) of the status return was '1'. Masking ignored. BCST bit works in either mask mode or compare mode. In mask mode it works like other mask bits on the BCST bit. In compare mode, Status set occurs if BCST bit is different from bit 5 of BC control word.
		10	Format Error. The returned echo from the RT contained 1553 violations. See bits 3, 2, 1, 0 for a more accurate guess of the source of the problem.
		9	Response timeout. The RT responded too late or didn't respond at all. In PP194 – The RIU did not respond properly.
		8	Loop back failed. The nature of 1553 bus is that every word transmitted, is also echoed back. The core verifies that the echo is correct and equal to the transmitted word. If not, this bit is set to '1'. Also set in messages with error injected. Tip: The source of this type of error could be transceiver fault, or bus coupling problem. In PP194 –Loop back Failed.
		7	Unmasked Status bit set. This bit will be set to '1' if one of the status bits are set high and its appropriate mask bit in the BC control word is unmasked ('0'). BCST bit influences only in mask mode. See registers section for description of BCST bit.
		6..5	Number of retries done for this message. "11" is 3, "10" is 2…
		4	Good data block received by TestersChoice, waiting in Data Block. '1' – after an RT-BC, RT2RT, and Transmit Mode code with data commands if the message ended OK. '0' – after other message types, or if the above type of message was invalid. '0' – for received words that did not match the expected values if "Write Verify" mode is enabled for the message. Loop back test failure does not cripple this bit result. In PP194 – Both phases completed successfully and a real RIU sent its status and saved to memory.
		3	'1' indicates the RT responded with wrong RT address.

			In PP194 – RIU status respond with wrong RIU address.
		2	'1' indicates that the RT transmitted a wrong number of words. In PP194 – RIU Data phase error.
		1	'1' – Incorrect sync type response by RT. In PP194 – RIU Status phase error.
		0	'1' – Invalid word. Indicates that the RT responded with a word containing 1553 errors. In PP194 – The RIU responded with Manchester / parity error.
2	Received 1st status	15..0	First status received from un-simulated RT. In PP194 – Status bits of status word.
3	Received 2nd status	15..0	Second status received from un-simulated RT.

5.17 mcx_Get_Element_Results_PP194

```
INT16
mcx_Get_Element_Results_PP194 (
                                UINT16      deviceId
                                UINT16      busList
                                UINT16      elementIndex
                                UINT16 *    blockStatusWord
                                UINT16 *    buffer
                                UINT16      bufferSize
                                UINT16 *    status1
                                UINT16 *    status2
                                UINT16 *    tag
                               )
```

Parameters

deviceId	Unique Device ID 0 - (sitalMaximum_DEVICES - 1)
busList	Unique ID of BusList 0 - (MAX_BUSLISTS - 1)
elementIndex	Element's Index (not Element ID) within the busList
blockStatusWord	Returns Element/Message findings; see Message State table below.
buffer	Returns the buffer according to requested size of data transmitted.
bufferSize	Buffer size for the retuned buffer.
status1	The first status that was received from a real non-simulated RT
status2	Only in RT2RT command format, the second status (Rx Status) that was received from a real non-simulated RT
tag	This parameter works in either one of two modes of operations: the message Rate mode, or the message Gap mode. In Gap mode - The 16 LSBs of the 32 bit time tag counter are stored here when the message was launched by the core. In Rate mode – A frame counter is incremented by 1 at EOF. This frame counter value is stored in this entry when the message is transmitted

Description

Remarks
This function deferes from mcx_Get_Element_Results(..) by the type of the buffer (UINT16*) retruned.
Both functions (mcx_Get_Element_Results(..) and mcx_Get_Element_Results_PP194(..)) can be used to retrieve PP194 element results.

Mode: Ready
This function gets the results of a transmission of a specific Element within a specified BusList. Message results comprise the message words that were actually transmitted along the internal bus together with the statistics

(diagnostics) of the transmitted message. The diagnostics include an indicator of whether the message transmission was successful, status words, the data payload that was actually transmitted on the bus.
The difference between this function and the Word Monitor family of functions is that the Word Monitor sits on the bus in the Tester device and simply records all the words that go by; the Word Monitor has no concept of BusLists or Elements. This function, on the other hand, returns a specific Element's results from the specified BusList.

Notes

This function requires an Element's index parameter rather than its ID since the same Elements may appear more than once within a single frame.

Message State Table

	Name	Bit	Description
0	Time Tag Word 16 LSBs. (Gap mode)	15..0	16 LSBs of the real time counter. Written by core when the message started.
0	Frame Number (Rate mode)	15..0	Frame number when this message was transmitted. Frame number is incremented every EOF. It is recommended to init this value to 0xFFFF before run.
1	Message findings	15	End Of Message – Set to '1' by the core when the message has been complete.
		14	Start Of Message - Set to '1' by the core when the message has been started. In most cases, this bit is stuck at '1' after end of message if there is a 1553 bus-coupling problem.
		13	'0' – Was sent on Bus A. '1' – Was sent on Bus B.
		12	'1' – Error was found in the message. Bits 10, 9, 8, 3, 2, 1, 0 indicate cause of error.
		11	Status Set. One of the status bits (excluding BCST bit) of the status return was '1'. Masking ignored. BCST bit works in either mask mode or compare mode. In mask mode it works like other mask bits on the BCST bit. In compare mode, Status set occurs if BCST bit is different from bit 5 of BC control word.
		10	Format Error. The returned echo from the RT contained 1553 violations. See bits 3, 2, 1, 0 for a more accurate guess of the source of the problem.
		9	Response timeout. The RT responded too late or didn't respond at all. In PP194 – The RIU did not respond properly.
		8	Loop back failed. The nature of 1553 bus is that every word transmitted, is also echoed back. The core verifies that the echo is correct and equal to the transmitted word. If not, this bit is set to '1'. Also set in messages with error injected. Tip: The source of this type of error could be transceiver fault, or bus coupling problem. In PP194 –Loop back Failed.
		7	Unmasked Status bit set. This bit will be set to '1' if one of the status bits are set high and its appropriate mask bit in the BC control word is unmasked ('0'). BCST bit influences only in mask mode. See registers section for description of BCST bit.
		6..5	Number of retries done for this message. "11" is 3, "10" is 2…
		4	Good data block received by TestersChoice, waiting in Data Block. '1' – after an RT-BC, RT2RT, and Transmit Mode code with data commands if the message ended OK. '0' – after other message types, or if the above type of message was invalid. '0' – for received words that did not match the expected values if "Write Verify" mode is enabled for the message.

			Loop back test failure does not cripple this bit result. In PP194 – Both phases completed successfully and a real RIU sent its status and saved to memory.
		3	'1' indicates the RT responded with wrong RT address. In PP194 – RIU status respond with wrong RIU address.
		2	'1' indicates that the RT transmitted a wrong number of words. In PP194 – RIU Data phase error.
		1	'1' – Incorrect sync type response by RT. In PP194 – RIU Status phase error.
		0	'1' – Invalid word. Indicates that the RT responded with a word containing 1553 errors. In PP194 – The RIU responded with Manchester / parity error.
2	Received 1st status	15..0	First status received from un-simulated RT. In PP194 – Status bits of status word.
3	Received 2nd status	15..0	Second status received from un-simulated RT.

5.18 mcx_Element_DataBlock_Write

```
INT16
mcx_Element_DataBlock_Write (
    UINT16      deviceId
    UINT16      element
    UINT16      dataBlock
    UINT16 *    buffer
    UINT16      bufferSize
)
```

Parameters

deviceId	Unique Device ID 0 - (sitalMaximum_DEVICES - 1)
element	Unique ID of Element 0 - (MAX_ELEMENTS - 1)
dataBlock	Unique ID of DataBlock 0 - (MAX_DATABLOCKS - 1)
buffer	A pointer to an array of data words to be copied into the new data block, or NULL if isn't required
bufferSize	The size (in words) of the data buffer to write

Description

Mode: Ready
This function writes the buffer of the DataBlock (by its unique ID) by the buffer size.

5.19 mcx_Element_DataBlock_Read

```
INT16
mcx_Element_DataBlock_Read      (
                                  UINT16      deviceId
                                  UINT16      element
                                  UINT16      dataBlock
                                  UINT16 *    buffer
                                  UINT16      bufferSize
                                )
```

Parameters

deviceId	Unique Device ID 0 - (sitalMaximum_DEVICES - 1)
element	Unique ID of Element 0 - (MAX_ELEMENTS - 1)
dataBlockId	Unique ID of DataBlock 0 - (MAX_DATABLOCKS - 1)
buffer	A pointer to an array of data words to be copied into the new data block, or NULL if isn't required
bufferSize	The size (in words) of the data buffer to read.

Description

Mode: Ready & Runtime
In both ready and run-time modes, this function's call access the HW and reads the data to return in buffer and buffer size.

5.20 mcx_DevicePassiveTimeStarted

```
INT16 mcx_DevicePassiveTimeStarted (
    UINT16      deviceId
    UINT16 *    isFirstPassive
)
```

Parameters

deviceId	Unique Device ID 0 - (sitalMaximum_DEVICES - 1)
isFirstPassive	A pointer to the state of passive phase: 0 == not passive, 1 == is first passive.

Description

Mode: Ready
In this mode, when calling this function before running (mcx_Start(..)) will always return passive time 1.
After running and stop, the function will behave as in runtime mode.

Mode: Runtime
This function checks the state of the currently running BusList.
For each BusList running, the first occurrence of passive phase (first query after active phase) returned 1.
Quesries sent within the same running frame in the passive phase return 0.

5.21 mcx_GetDescriptors

```
INT16 mcx_GetDescriptors (
    UINT16      deviceId
    char *      deviceName
    char *      deviceManufacturer
    char *      deviceFirmware
    char *      deviceSerial
)
```

Parameters

deviceId	Unique Device ID 0 - (sitalMaximum_DEVICES - 1)
deviceName	A pointer to the specified device's name.
deviceManufacturer	A pointer to the specified device's manufacturer.
deviceFirmware	A pointer to the specified device's firmware.
deviceSerial	A pointer to the specified device's serial number.

Description

Mode: Ready
This function returns in the pointers the relevant device's following details; name, manufacturer, firmware and serial number.

Mode: Runtime
Same as Ready Mode.

5.22 mcx_Set_Error

```
INT16 mcx_Set_Error (
    UINT16  deviceId
    UINT16  errorType
    UINT16  messageNumber
    UINT16  wordNumber
    UINT16  injectionParameters
    INT16   zXDistortion
)
```

Parameters

deviceId Unique Device ID 0 - (sitalMaximum_DEVICES - 1)

errorType Error type corresponding to the following constants (defined in Header file):
mcx_NO_ERROR 0x0000
mcx_PARITY_ERROR 0x1000
mcx_BIPHASE_ERROR 0x3000
mcx_SYNC_ERROR 0x4000
mcx_ZERO_CROSSING_ERROR 0x8000
mcx_NOISE_ERROR 0xF000

messageNumber Message number to insert the error to. Valid values: 0 – 3.

wordNumber Word number to insert the error to. Valid values: 0 – 35.

Word number to insert the error. 0 to 35.
The word number type depends on the Message Type:
For example:
Rx Message type is structured: Command, Data0, Data1…Data N, Status.
Word 0 = Command
Word 1 = Data0
Word..N (up to 35) = Status
Example 2:
RT 2 RT type structured: Rx Command, Tx Command, Tx Status, Data0…DataN, Rx Status.
Word 0 = Rx Command
Word 1 = Tx Command
Word 2 = Tx Status
Word 3 = Data0
Word N = Rx Status

PP194 word numbering:
0 for WORD1
1 for WORD2
2 for Rx bus A RIU response for WORD2 emulation
3 for Rx bus B RIU response for WORD2 emulation
4 for WORD3
5 for WORD4
6 for Rx bus A RIU STATUS response emulation
7 for Rx bus B RIU STATUS response emulation

injectionParameters Injection parameters corresponds to the specified Error Type:

Parity Error : This field is ignored.
Word Length: 0x0 – Decrease 2 bits from specified word. 0x1 – Decrease 1 bit from specified word. 0x2 – Increase 1 bit to specified word. 0x3 – Increase 2 bit to specified word. 0x4 – Increase 3 bit to specified word.
Bi Phase: 0x0 for bit 0 0x1 for bit 1 : 0xF for bit 15 0x10 for parity bit. In case Auto increment is selected, go from parity of Word N to first bit of Word N+1. **0 to 25 for PP194**
Sync: 0x0 - 111100, 0x1 - 110000, 0x2 - 111001, 0x3 - 011000, 0x4 - 000011, 0x5 - 001111, 0x6 - 000110, 0x7 - 100111, 0xF - Inverted of what is expected. 000111 ⇔ 111000 **For PP194 works as 0xF only.**
Zero Crossing: 0 for 1st half of bit 0 1 for 2nd half of for bit 0 2 for 1st half of bit 1 3 for 2nd half of for bit 1 : 38 for 1st half of bit 19 (parity bit) 39 for 2nd half of for bit 19 Zero Xing is inserted in one of the ½ bits of each word. This change would either expedite the arrival of the next zero Xing, or skew it away by the amount defined in bits 7..4 below. Note that Zero Xing in some of the bits might skew the zero Xing of the next bit. This will always happen during the sync in bit 0 and 2, and in 2nd half of bits that are followed by an opposite bit value. **0 to 51 for PP194**

zXDistortion Zero Crossing distortion parameter applied only when selecting Zero Crossing ErrorType

Zero Xing:
Signed field in the Range of -8 to +7.
-8 : skew next Xing by 8 * 1000/30 nano seconds = +266 ns.
-7 : skew next Xing by 7 * 1000/30 nano seconds = +233 ns.
-6 : skew next Xing by 6 * 1000/30 nano seconds = +200 ns.
:
6 : expedite next Xing by 6 * 1000/30 nano seconds = -200 ns.
7 : expedite next Xing by 7 * 1000/30 nano seconds = -233 ns.

Description

Mode: Ready
This function inject error according to selected error type. Once set, the error would be injected in the BusList following the earliest Start command.
setup defines the following parameters:
1. The message number in the BusList that should have the error.
2. The word number inside the Element to insert the fault.
3. The bit number in the word to inject the fault (where relevant).
4. Additional parameters per fault.

The errors supported are:
1. Parity error – the Parity bit is inverted.
2. Bi-Phase error –The second half of the bit is identical to the first half.
3. Sync Error – Various bit patterns for the 6 half microseconds are supported.
* For PP194 – applies for inverted pattern only.
4. Zero crossing distortion – Distortion of the zero crossing compared with the previous zero crossing.
5. Noise ** FOR EBR1553 messages only** – inserted to word 0. Injecting noise/distortion to the bus and creating signal permutation, imitating non-standard bus signal.

Mode: Runtime
Same as in Ready Mode with a single difference – when error is injected while running, it is injected with 0 delay (in oppose to ealiest start).

Limitations

The setup limitations are:
1. A single error per ‘start’ command.
2. No HW checking if the fault location in the message is within the Element length.
3. The error is injected to each and every occurance of that Element after started.

5.23 mcx_Reload

```
INT16 mcx_Reload (
    UINT16 deviceId
    UINT16 protocol
)
```

Parameters

deviceId	Unique Device ID 0 - (sitalMaximum_DEVICES - 1)
protocol	User Code option for setting the device to work in a protocol/state. The following definitions can be found in McxAPI.h: // UserCode Options Protocol_1553_PP194 Protocol_H009 MultiRT

Description

Mode: Ready

This function dismisses the currently used FPGA HW file and re-loads an FPGA to the MCX Tester device. After successful re-load, it initializes device to a protocol and state according to initialization protocol parameter.

Loading FPGA operation may last up to 8-10 seconds.

Mode: Runtime

Since this is a “configurations and settings” function, it stops the device activities and data transfer.
Note – this function does not wait till the end of message / frame, it is executed with no delay.

Note

Loading FPGA file applies to MultiComBox devices. For Grip2 and PMC (PCI) tester types, the hardware file is burned into the device.

5.24 mcx_SetFrameTime

```
INT16 mcx_SetFrameTime (
    UINT16  deviceId
    UINT32  microSeconds
)
```

Parameters

deviceId	Unique Device ID 0 - (sitalMaximum_DEVICES - 1)
microSeconds	Frame time in micro seconds.

Description

Mode: Ready
This function sets the BusList time in resotuion of 100 microseconds.
Note – the acceptable values are 0 to 100*64K microseconds.

Mode: Runtime
N/A

Note

If the frame length is shorter than the active part of the BusList (when it is transmitting the Elements), then the tester will run back to back at 100% bus utilization.

5.25 mcx_MapDevices

```
INT16 mcx_MapDevices (
    UINT16*    numberOfDevices
)
```

Parameters

numberOfDevices	A pointer in which the number of available / connected devices is returned

Description

Mode: Ready
This function sense all connected devices and returns an accumulated number that contains all types of connected devices.
For example, if a Grip2 (a single device) and a MultiComBox (inherentaly contains 2 devices) are both connected to a machine, this function will return numberOfDevices = 3.

Mode: Runtime
N/A

5.26 mcx_Free

INT16 mcx_Free	**(**		
	UINT16		***deviceId***
	)		

Parameters

deviceId	Unique Device ID 0 - (sitalMaximum_DEVICES - 1)

Description

Mode: Ready
This function releases all internal relations between mapped BusLists, Elements, DataBlocks and Devices.

Mode: Runtime
This function stops the run and releases all internal relations between mapped BusLists, Elements, DataBlocks and Devices.

5.27 mcx_FreeBusList

```
INT16 mcx_FreeBusList (
    UINT16 deviceId
    UINT16 busList
)
```

Parameters

deviceId	Unique Device ID 0 - (sitalMaximum_DEVICES - 1)
busList	Unique ID of BusList 0 - (MAX_BUSLISTS - 1)

Description

Mode: Ready
This function releases all internal relations between BusList, Elements, DataBlocks.

Mode: Runtime
This function stops the run and releases all internal relations between BusList, Elements, DataBlocks.

5.28 mcx_SetUserPort

```
INT16 mcx_SetUserPort (
    UINT16  deviceId
    UINT16  userPort
)
```

Parameters

deviceId	Unique Device ID 0 - (sitalMaximum_DEVICES - 1)
userPort	User port to apply for this device. The following definitions can be found in McxAPI.h: // User Port Options Protocol_1553_PP194 Protocol_H009 MultiRT

Description

Mode: Ready
User Port is one of the Testers' configuration register which controls various device activities.
This function provides a subset of this config adjustment in the area of changing a specific device's protocol between 1553 and PP194 to H009. It also provides the capability to define a device to be a BC + MultiRTs + Monitor or as MultiRTs + Monitor ony.

Mode: Runtime
N/A – applies to Ready mode only.

Note

Setting the user port overwrites the existing data in the user port.
The best practice of this function is calling mcx_GetUserPort(..) and modify the relevant data only.

5.29 mcx_GetUserPort

```
INT16 mcx_GetUserPort (
    UINT16      deviceId
    UINT16*     userPort
)
```

Parameters

deviceId	Unique Device ID 0 - (sitalMaximum_DEVICES - 1)
userPort	The User port that this device is set to. The following definitions can be found in McxAPI.h: // User Port Options Protocol_1553_PP194 Protocol_H009 MultiRT

Description

Mode: Ready
This function gets the protocol of the device.

Mode: Runtime
N/A – applies to Ready mode only.

5.30 mcx_GetCurrentFrameNumber

```
INT16 mcx_GetCurrentFrameNumber (
    UINT16      deviceId
    UINT16*     currFrameNumber
)
```

Parameters

deviceId	Unique Device ID 0 - (sitalMaximum_DEVICES - 1)
currFrameNumber	Returns the frame number

Description

Mode: Ready and Runtime
This function reads the current BusList number from the hardware.
Bset practice for this function –

- When running #N times – read the current BusList number before the run and after it is completed to verify that the number of sent BusLists is identical to the requested number.
- When running forever and stopping the run – read the current BusList number before the run and after stop to verify the number of BusLists sent till run stopped.

Note

The frame counter is 16 bits long, and it cycles back from 64K-1 to 0, continuing thereafter.

5.31 mcx_Element_SetGap

```
INT16 mcx_Element_SetGap (
    UINT16 deviceId
    UINT16 elementId
    UINT16 gap
)
```

Parameters

deviceId	Unique Device ID 0 - (sitalMaximum_DEVICES - 1)
elementId	Element ID (message ID) to set a gap to.
gap	Gap value for the Element.

Description

Mode: Ready
This function sets the gap value to an Element.
The gap is detailed in microseconds and is the time from the beginning of the current Element to the beginning of the next Element.
In case the gap is shorter than the current Element, the next element is transmitted back to back.
Note, the Element must be created in order to apply a value to it.

Mode: Running
N/A – not applied for Running mode.

5.32 mcx_Element_SetRate

```
INT16 mcx_Element_SetRate (
    UINT16  deviceId
    UINT16  elementId
    UINT16  rate
    UINT16  skew
    UINT16  elementSpacing
)
```

Parameters

deviceId	Unique Device ID 0 - (sitalMaximum_DEVICES - 1)
elementId	Element ID (message ID) to set a gap to.
rate	Rate value for the Element, value are 0 – 15: Case N (decimal value) is 0: Skip this message. 1: transmit this message every frame. 2..14: transmit this message every 2^(N-1) frames. Will be transmitted in frames who's frame N-1 (unless skewed). 15: Transmit this message once. The HW core resets this value to 0 after message has been transmitted once.
skew	Skew value (0 – 15): Frame skew: Defines a number M between 0 and 15. Will skew the message M frames away from its rate planned location (defined above).
elementSpacing	Spacing time: Number of microseconds between end of message and start of next message. Could be left zero.

Description

Mode: Ready
This function sets the rate value to an Element to appear within a Buslist.
The rate is 'rate' parameter with values of 0 – 15, see above.

Mode: Running
N/A – not applied for Running mode.

NOTE – in order to use Rate mode, use 'mcx_Start_RateMode (..)' function instead of 'mcx_Start (..)'.

5.33 mcx_Grip2_GetTemperature

```
INT16 mcx_Grip2_GetTemperature (
    UINT16      deviceId
    UINT16*     temperature
)
```

Parameters

deviceId	Unique Device ID 0 - (sitalMaximum_DEVICES - 1)
temperature	Returned temperature

Description

Mode: Ready + Running
This function gets Grip2 device's temperature (Celsius).

Note

Applies only to Grip2 devices.

5.34 mcx_GetTemperature

```
INT16 mcx_GetTemperature (
    UINT16      deviceId
    UINT16*     temperature
)
```

Parameters

deviceId	Unique Device ID 0 - (sitalMaximum_DEVICES - 1)
temperature	Returned temperature

Description

Mode: Ready + Running
This function gets Grip2 and/or MCX C device's temperature (Celsius).

Note

Applies only to Grip2 and MCX C (Release in Feb 2020) devices.

5.35 mcx_Get_Version

```
INT16 mcx_Get_Version (
    UINT16      deviceId
    UINT16*     version
)
```

Parameters

deviceId	Unique Device ID 0 - (sitalMaximum_DEVICES - 1)
version	A pointer that returns the firmware version

Description

Mode: Ready & Runtime

This function returns the firmware version of the current device (MultiComBox, Grip2, cPCI card).

5.36 mcx_wm_GetNextSymbol

```
INT16 mcx_wm_GetNextSymbol (
    UINT16      deviceId
    UINT32*     swPointer
    WORD*       descriptor
    WORD*       data
    WORD*       bufferSize
)
```

Parameters

deviceId	Unique Device ID 0 - (sitalMaximum_DEVICES - 1)
swPointer	Last software pointer
descriptor	descriptor
data	data
bufferSize	returned buffer size

Description

Mode: Ready + Running
This function gets data and descriptor pair from the Monitor's stack.
The pair contains raw data (time tags, data, command, status, etc). its decoding is done by message decoders; mcx_wm_GetNextMsg_1553_194(..) and mcx_wm_GetNextMsg_H009(..).

Note

swPointer is managed by the API function and must be tampered by the user.

5.37 mcx_wm_GetNextMsg_1553_194

```
INT16 mcx_wm_GetNextMsg_1553_194 (
                                   UINT16          deviceId
                                   INT16*          msgType
                                   UINT32*         swPointer
                                   WORD*           rxCommand
                                   WORD*           txCommand
                                   WORD*           data
                                   WORD*           bufferSize
                                   WORD*           rxStatus
                                   WORD*           txStatus
                                   UINT32*         BSW
                                   unsigned long*  tTag
                                 )
```

Parameters

deviceId	Unique Device ID 0 - (sitalMaximum_DEVICES - 1)
msgType	Specified the type. The following definitions can be found in McxAPI.h: UnParsed BC2RT RT2BC RT2RT BC2BCST RT2BCST BCST2RT_Invalid BCST2BC_Invalid BC2RT_Mode_No_Data BC2RT_Mode_With_Data RT2BC_Mode_No_Data RT2BC_Mode_With_Data RT2RT_Mode_No_Data RT2RT_Mode_With_Data BC2BCST_Mode_No_Data BC2BCST_Mode_With_Data RT2BCST_Mode_No_Data RT2BCST_Mode_With_Data BCST2RT_Mode_No_Data BCST2BC_Mode_No_Data
swPointer	Last software pointer
rxCommand	Receive Command
txCommand	Transmit Command
data	Data buffer of the read message
bufferSize	Data buffer size

rxStatus	Receive Status
txStatus	Transmit Status
BSW	Block Status Word, Or-ed combination. The following definitions can be found in McxAPI.h: mcx_wm_WRONG_CMD_SYNC mcx_wm_INVALID_WORD mcx_wm_NO_RESPONSE mcx_wm_LOW_WORD_COUNT_ERROR mcx_wm_HIGH_WORD_COUNT_ERROR mcx_wm_BUS_SWITCHED_ERROR mcx_wm_LOST_SYNC_ON_BAD_TIME_SYMBOL mcx_wm_DATA_OVERRUN mcx_wm_BUS_A mcx_wm_BUS_B mcx_wm_PP194
tTag	Time tag in 0.5 microseconds resolution

Description

Mode: Ready + Running
This function decodes a 1553 or PP194 (WB194) Element from the Monitor's stack.
The parameters returned from this function contains decoded data, updated pointers and filtered out time symbols.

Next Message Data's Validity

Please note that this function is called and returns Message Type UnParsed (*msgType==0*).
No new messages have been found or are being processed, hence, all returned data should be ignored.

Note

swPointer is managed by the API function and must be tampered by the user.

5.38 mcx_wm_GetNextMsg_H009

```
INT16 mcx_wm_GetNextMsg_H009 (
    UINT16          deviceId
    UINT32*         swPointer
    WORD*           command
    WORD*           isCommandValid
    WORD*           data
    WORD*           bufferSize
    UINT32*         BSW
    unsigned long*  tTag
)
```

Parameters

deviceId	Unique Device ID 0 - (sitalMaximum_DEVICES - 1)
swPointer	Last software pointer
command	H009 Command
isCommandValid	A flag indicating if the command is valid (1) or not (0)
data	Data buffer
bufferSize	Data buffer size
BSW	Block Status Word, Or-ed combination. The following definitions can be found in McxAPI.h: mcx_wm_WRONG_CMD_SYNC mcx_wm_INVALID_WORD mcx_wm_NO_RESPONSE mcx_wm_LOW_WORD_COUNT_ERROR mcx_wm_HIGH_WORD_COUNT_ERROR mcx_wm_BUS_SWITCHED_ERROR mcx_wm_LOST_SYNC_ON_BAD_TIME_SYMBOL mcx_wm_DATA_OVERRUN mcx_wm_BUS_A mcx_wm_BUS_B mcx_wm_PP194
tTag	Time tag in 0.5 microseconds resolution

Description

Mode: Ready + Running

This function decodes a H009 Element from the Monitor's stack.

The parameters returned from this function contains decoded data, updated pointers and filtered out time symbols.

Note

swPointer is managed by the API function and must be tampered by the user.

5.39 mcx_Restart

```
INT16 mcx_Restart (
    UINT16 deviceId
    UINT16 busList
)
```

Parameters

deviceId	Unique Device ID 0 - (sitalMaximum_DEVICES - 1)
busList	Unique ID of BusList 0 - (MAX_BUSLISTS - 1)

Description

Mode: Ready + Running
This function allows the user to re-run a running bus list whose data was updated.
Using this function saves the user from the need to wait for the run finish + mcx_Stop + mcx-Start.
User's data is updated to the bus once it is called and the messages are transmitted.

Notes & Limitations

- mcx_Restart function does not run if the mcx_Start is running forever and will return an error in such a case.
- mcx_Restart is intended for re-running an mcx_Start once (one shot frames)
- For RT2MCX messages' data, one has to read the data (mcx_Get_Element_Results) before calling mcx_Restart if needed.

5.40 mcx_BusList_UpdateData

```
INT16 mcx_BusList_UpdateData (
    UINT16   deviceId
    UINT16   busList
    INT32*   updatedFrame
)
```

Parameters

deviceId	Unique Device ID 0 - (sitalMaximum_DEVICES - 1)
busList	Unique ID of BusList 0 - (MAX_BUSLISTS - 1)
updatedFrame	A pointer that returns the BusList's number that was updated by this function

Description

Mode: Running
This function allows the user to update a running bus list's data during run.
Using this function saves the user from the need to wait for the run finish + mcx_Stop + mcx-Start.
User's data is updated to the bus once a passive phase (no data is being transmitted). If a passive phase is not detected for 5 retries a 'no passive phase detected' error returns.
Passive phase can be missing if the bus utilization is very high. On average, the passive part should be longer than 1 ms, and for MultiRT mode, it should be greater than 2 ms for the function to detect the passive phase.

Mode: Ready
This function updates the data to be transmitted.

Notes & Limitations

- This function is intended for re-running an mcx_Start multiple shots, data update occurs during the run.
- The buslist for this functionmust be mapped to a device and contin the relevant messages and data.

5.41 mcx_GetMonitorErrorsDescription

```
INT16 mcx_GetMonitorErrorsDescription (
    UINT32      bsw
    Char*       errorDescription
)
```

Parameters

bsw	Block Status Word as received in Word Monitor functions mcx_wm_GetNextMsg_1553_194(..) or mcx_wm_GetNextMsg_H009(..)
errorCodeDescription	Returned description

Description

Mode: Ready + Running
This function gets a string by a BSW (Block Status Word) value as received in Word Monitor functions mcx_wm_GetNextMsg_1553_194(..) or mcx_wm_GetNextMsg_H009(..)

Notes

Note I - In case that the Block Status Word contains few error messages the returned 'errorDescription' string contains a concatenated strings with '&' separators.
For example; "Wrong command sync & No reponse & ".
Note II – The Monitor error codes can be found in header file McxAPI.h under `// WORD MONITOR Returned codes`

5.42 mcx_GetReturnCodeDescription

```
INT16 mcx_GetReturnCodeDescription (
    INT16   errorCode
    Char*   errorCodeDescription
)
```

Parameters

errorCode	Error code to get its description
errorCodeDescription	Returned description

Description

Mode: Ready + Running
This function gets a string by an error code value.

5.43 mcx_GetSimulatorErrorsDescription

```
INT16 mcx_GetSimulatorErrorsDescription (
    UINT32      bsw
    Char*       errorDescription
    UINT16      protocol
)
```

Parameters

bsw	Block Status Word as received in mcx_Get_Element_Results(..)
errorCodeDescription	Returned description
protocol	protocol: 0==1553 | 1==PP194 | 2==H009

Description

Mode: Ready + Running
This function gets a string by a BSW (Block Status Word) value as received in mcx_Get_Element_Results(..).

Notes

Note I - In case that the Block Status Word contains few error messages the returned 'errorDescription' string contains a concatenated strings with '&' separators.
For example; "Wrong command sync & No reponse & ".
Note II – The simulator error codes can be found in header file McxAPI.h under `// SIMULATOR ELEMENT | MESSAGE FINDINGS`
Note III – currently, this function supports protocol Mil-Std-1553 only.

5.44 mcx_SetConfigurationRegisters

INT16 mcx_SetConfigurationRegisters	**(**	
	UINT16	***deviceId***
	UINT16	***configRegisters***
	UINT16	***configRegisters2***
	)	

Parameters

deviceId	Unique Device ID 0 - (sitalMaximum_DEVICES - 1)
configRegisters	Configuration Register Address 0x004A, see description below
configRegisters2	Configuration Register Address 0x004B, see description below

Description

Mode: Ready

This function sets configuration registers (addresses 0x4A & 0x4B) with selected values, see elaboration below. In order to support Bus swap on fail and retries, use this function to set retry on and specify the bus swap configuration as well.

In addition, on creating a bus list, specify in message options bit 8 high.

The following code example demonstrates PP194 message creation with retry enabled and on first retry swap bus (marked in YELLOW):

```
static UINT16 BusList1 = 0;
static UINT16 Element1 = 0;
static UINT16 DB1 = 0;
static UINT16 datablock32[64];
short iResult = 0;
unsigned short elementCommand =  0x0c43;
unsigned short numberOfIterations = 1;
unsigned short messageOptions = 0x0104;
UINT16 simulatedStatus = 0x1234;
unsigned short wDataWord0 = 0x2345;
unsigned short wDataWord1 = 0x00ff;

iResult += mcx_Initialize(0, Protocol_1553_PP194);
// Enabling all RIUs
//iResult += mcx_EnableRius(0, 0xffff);
iResult = mcx_SetConfigurationRegisters(deviceID, 0xC, 0);
iResult += mcx_Create_BusList(BusList1);
iResult += mcx_Create_BusList_Element (Element1, elementCommand, 0x80 /*Bus A*/ |
messageOptions, 0 , simStatus, 0);
iResult += mcx_Create_Element_DataBlock (DB1, DataBlockMode_64_WORDS, dataBlock,
DataBlockS);
dataBlock[0] = data0;
dataBlock[1] = data1;
iResult += mcx_Map_DataBlock_To_Element (Element1, DB1);
iResult += mcx_Map_Element_To_BusList (BusList1, Element1);
iResult += mcx_Start (mrtDeviceID, BusList1, numberOfIterations);
Sleep(100);
```

Configuration Register Address 0x004A

Bit number	Read/Write/ Default	
15	Write/Read/'0'	'0' – generates a 500ns low pulse on the INTn signal. '1' – Level mode. INTn stays low until the host reads interrupt status register.
14	Write/Read/'0'	'1' – Loop back transmission in the FGPA, no bus transmission. '0' – Normal operation.
13	Write/Read/'0'	Mask / Compare the BCST bit in returned status bit. ** '1' – Mask. The relevant bit in the BC control word operates as mask for the BCST bit of the received status. '0' – Compare. The relevant bit in the BC control word is compared with the BCST bit of the received status.
12	Write/Read/'0'	'1' – Stop at end of Message error. If optional retry succeeded => messages continue!
11	Write/Read/'0'	'1' – Stop at end of frame if error. If optional retry succeeded => frames continue!
10	Write/Read/'0'	'1' – Stop messages if unexpected, non-masked status bits are set. If optional retry succeeded => messages continue!
9	'0'	'0' is preset for this bit.
8	'0'	'0' is preset for this bit.
7	'0'	'0' is preset for this bit.
6	Write/Read/'0'	'1' - PP194 enabled - only module 0 monitor is functional. PP194 Tx bus is mapped to module 1 busses. '0' – only 1553, both monitor modules are functional.
5	Write/Read/'0'	'1' - Message gap mode enable. Gap is defined in 1st word in stack. If enabled will start next message after gap*1us. If message gap is smaller than message length, will use default 10 microseconds inter-message gap. '0' – Rate mode. 1st word in each stack entry defines the rate of the message.
4..3	Write/Read /"00"	"00" – Global retry off. Do not retry. "01" – Retry once. "10" – Retry twice. "11" – Retry three times.
2	Write/Read/'0'	'0' – First & third retry on same bus if message failed. '1' – First & third retry on opposite bus if message failed.
1	Write/Read/'0'	'0' – Second retry on same bus as original failed message. '1' – Second retry on opposite bus of original message.
0	Write/Read/'0'	'1' – Retry a message if retry enabled and one of the unmasked status bits are set high in the returned RT status word. BCST bit pass/fail has a special treatment as seen above in bit 11 setup. '0' – No retry if status bits that are not masked are set.

Default Value: 0x0000.
** Note that the BCST bit in the status return is only set by the RT in the proceeding message's status word after a broadcast message FOR a transmit status or transmit command mode commands. Otherwise the BCST bit should be '0'.

Configuration Register 2 Address 0x004B

Bit number	Read/Write/ Default	
7	Write/Read/'0'	MRT mode – '1' - when searching if command exists, ignore Word Count field, bits 0 to 4. Also covered by USER CODE bit 7.

5.45 mcx_GetConfigurationRegisters

```
INT16 mcx_GetConfigurationRegisters (
    UINT16      deviceId
    UINT16 *    configRegisters
    UINT16 *    configRegisters2
)
```

Parameters

deviceId	Unique Device ID 0 - (sitalMaximum_DEVICES - 1)
configRegisters	A pointer to returned Configuration Register Address 0x004A, see description below
configRegisters2	A pointer to returned Configuration Register Address 0x004B, see description below

Description

Mode: Ready
This function gets configuration registers values (addresses 0x4A & 0x4B), see elaboration below.

Configuration Register Address 0x004A

Bit number	Read/Write/ Default	
15	Write/Read/'0'	'0' – generates a 500ns low pulse on the INTn signal. '1' – Level mode. INTn stays low until the host reads interrupt status register.
14	Write/Read/'0'	'1' – Loop back transmission in the FGPA, no bus transmission. '0' – Normal operation.
13	Write/Read/'0'	Mask / Compare the BCST bit in returned status bit. ** '1' – Mask. The relevant bit in the BC control word operates as mask for the BCST bit of the received status. '0' – Compare. The relevant bit in the BC control word is compared with the BCST bit of the received status.
12	Write/Read/'0'	'1' – Stop at end of Message error. If optional retry succeeded => messages continue!
11	Write/Read/'0'	'1' – Stop at end of frame if error. If optional retry succeeded => frames continue!
10	Write/Read/'0'	'1' – Stop messages if unexpected, non-masked status bits are set. If optional retry succeeded => messages continue!
9	'0'	'0' is preset for this bit.
8	'0'	'0' is preset for this bit.
7	'0'	'0' is preset for this bit.
6	Write/Read/'0'	'1' - PP194 enabled - only module 0 monitor is functional. PP194 Tx bus is mapped to module 1 busses. '0' – only 1553, both monitor modules are functional.
5	Write/Read/'0'	'1' - Message gap mode enable. Gap is defined in 1st word in stack. If enabled will start next message after gap*1us. If message gap is smaller than message length, will use default 10 microseconds inter-message gap. '0' – Rate mode. 1st word in each stack entry defines the rate of the message.
4..3	Write/Read /"00"	"00" – Global retry off. Do not retry. "01" – Retry once. "10" – Retry twice. "11" – Retry three times.
2	Write/Read/'0'	'0' – First & third retry on same bus if message failed. '1' – First & third retry on opposite bus if message failed.
1	Write/Read/'0'	'0' – Second retry on same bus as original failed message. '1' – Second retry on opposite bus of original message.
0	Write/Read/'0'	'1' – Retry a message if retry enabled and one of the unmasked status bits are set high in the returned RT status word. BCST bit pass/fail has a special treatment as seen above in bit 11 setup. '0' – No retry if status bits that are not masked are set.

Default Value: 0x0000.

** Note that the BCST bit in the status return is only set by the RT in the proceeding message's status word after a broadcast message FOR a transmit status or transmit command mode commands. Otherwise the BCST bit should be '0'.

Configuration Register 2 Address 0x004B

Bit number	Read/Write/ Default	
7	Write/Read/'0'	MRT mode – '1' - when searching if command exists, ignore Word Count field, bits 0 to 4. Also covered by USER CODE bit 7.

5.46 mcx_GetTime

INT16 mcx_GetTime	**(**	
	UINT16	***deviceId***
	Unsinged long long *	***time***
	)	

Parameters

deviceId	Unique Device ID 0 - (sitalMaximum_DEVICES - 1)
time	A pointer to returned device time tag

Description

Mode: Ready & Running
This function returns the card's time tag. The time tag resolution is 0.5 microseconds.

Notes

Note I – if time tag reads zero as time tag an error is returned.
Note II – a time tag is shared to all devices within a card: 2 devices of MultiComBox, single device of a Grip2, 1-8 devices of PMC card.

5.47 mcx_SetTime

INT16 mcx_GetTime	**(**	
	UINT16	***deviceId***
	Unsinged long long	***time***
	)	

Parameters

deviceId	Unique Device ID 0 - (sitalMaximum_DEVICES - 1)
time	A value to apply the device's internal time tag

Description

Mode: Ready & Running
This function sets the card's time tag. The time tag resolution is 0.5 microseconds.

Notes

Note – a time tag is shared to all devices within a card: 2 devices of MultiComBox, single device of a Grip2, 1-8 devices of PMC card.

5.48 mcx_ RS485_Setup

```
INT16 mcx_ RS485_Setup (
    UINT16      moduleId
    UINT16      line
    UINT16      bitsCount
    UINT16      parity
    UINT16      stopBits
    UINT16      rateDivider
    UINT16      rxTxMode
    Uint16 *    offset
)
```

Parameters

moduleId	Module ID in initialized device. MultiComBox device contains 2 modules, each module contains 2 RS485 lines.
line	RS485 line in selected module. There are 2 lines for each Module.
bitsCount	Number of bits per word, range is 5-9.
parity	Type of parity: None = 0; Even = 1; Odd = 2.
stopBits	Number of Stop Bits: (number of stop bits) 1 = 0; (number of stop bits) 1.5 = 1; (number of stop bits) 2 = 2.
rateDivider	RS485-line's frequency divider register. the internal UART frequency is 90 MHz. The RS485 baud rate is: 90,000,000 / uiRateDivider.
rxTxMode	RS485 Rx / Tx mode: Tx only = 0; RxTx = 1.
offset	Pointer to a variable to receive the offset within the data receive buffer to start reading data words. This is the offset portion of the related RS485-line-in's status register (i.e., the portion that contains the RS485-line RX buffer offset – in units of words) where the next received RS485 word will be stored.

Description

Mode: Ready + Running
This function configures the bit, rate, and other characteristics of the module's RS485 line.

Notes

The device must be initialized before using this function.

5.49 mcx_RS485_Put

```
INT16 mcx_RS485_Put (
    UINT16  moduleId
    UINT16  line
    UINT16  length
    WORD *  buffer
)
```

Parameters

moduleId	Module ID in initialized device. MultiComBox device contains 2 modules, each module contains 2 RS485 lines.
line	RS485 line in selected module. There are 2 lines for each Module.
length	Number of data words to send. Range: 1 – 1008 (`SIZE_OF_RS485_TX_BUFFER`)
buffer	Pointer to a buffer containing the words to send. The buffer must be large enough to contain 'length' words.

Description

Mode: Ready + Running
This function sends data words directly out over the module's specified RS485 line.

Notes

The mcx_RS485_Setup(..) function must be called prior to using this function.

5.50 mcx_RS485_Get

```
INT16 mcx_RS485_Get (
    UINT16      moduleId
    UINT16      line
    UINT16 *    offset
    UINT16      length
    WORD *      buffer
)
```

Parameters

moduleId	Module ID in initialized device. MultiComBox device contains 2 modules, each module contains 2 RS485 lines.
line	RS485 line in selected module. There are 2 lines for each Module.
offset	Pointer to the variable that contains the offset (in units of words) where the reading operation should start within the specified RS485 line's RX buffer. Range: 0 <= offset < 1024 (SIZE_OF_RS485_RX_BUFFER) The offset is then automatically incremented.
length	Number of data words to send. Range: 1 – 1024 (SIZE_OF_RS485_RX_BUFFER)
buffer	Pointer to a buffer containing the words to recieve. The buffer must be large enough to contain 'length' words.

Description

Mode: Ready + Running
This function reads the specified number of data words from words that the module has most recently received into its RX buffer from the specified RS485 line.

Notes

The mcx_RS485_Setup(..) function must be called prior to using this function.

5.51 mcx_RS485_GetNumberOfReceivedWords

```
INT16
mcx_RS485_GetNumberOfReceivedWords (
    UINT16      moduleId
    UINT16      line
    UINT16      offset
    UINT16 *    length
)
```

Parameters

moduleId	Module ID in initialized device. MultiComBox device contains 2 modules, each module contains 2 RS485 lines.
line	RS485 line in selected module. There are 2 lines for each Module.
offset	Offset (in units of words) where the reading operation should start within the RS485 line's RX buffer. Range: 0 <= offset < 1024 (SIZE_OF_RS485_RX_BUFFER)
length	Pointer to a variable for receiving the word count of the words received in the module's RX buffer.

Description

Mode: Ready + Running

This function gets the count of the number of data words that the module has newly received into its RX buffer from the RS485 line. The data words have not yet been read by the application.

Notes

The mcx_RS485_Setup(..) function must be called prior to using this function.

5.52 mcx_RS485_GetStatus

```
INT16 mcx_RS485_GetStatus (
    UINT16      moduleId
    UINT16 *    Line0
    UINT16 *    Line1
)
```

Parameters

moduleId	Module ID in initialized device. MultiComBox device contains 2 modules, each module contains 2 RS485 lines.
Line0	A pointer which returns the Line0 status – 0 ok, 1 data corrupted
Line1	A pointer which returns the Line1 status – 0 ok, 1 data corrupted

Description

Mode: Ready + Running
This function gets the activity state for both lines of a Module: once a corruption of data occurs (2 units are transmitting at once and the data is overlapping) the relevant Line returns with 1.
0 value indicated a good data integrity.

Notes

The mcx_RS485_Setup(..) function must be called prior to using this function.

5.53 mcx_A429_Channel_GetCount

INT16 mcx_A429_Channel_GetCount	**(**		
	UINT32*		***numOfChannels***
	)		

Parameters

numOfChannels	Returns the number of exiting ARINC429 channels on the card

Description

Mode: Ready + Running
This function returns the number of detected Arinc429 channel on the card

Notes

Currently, Arinc429 capabilities applies only to PMC (PCI) cards.

5.54 mcx_A429_Channel_GetInformation

```
INT16 mcx_A429_Channel_GetInformation (
    UINT16              channel
    mcx_A429ChannelInfo* channelInfo
)
```

Parameters

channel	Arinc429 channel number
*mcx_A429ChannelInfo**	A pointer to channel information

Description

Mode: Ready
This function returns information on the specified channel.
Channel info struct specification can be found in Appendix B of this document as well as in McxAPI.h file

Notes

Currently, Arinc429 capabilities applies only to PMC (PCI) cards.

5.55 mcx_A429_Channel_Open

```
INT16 mcx_A429_Channel_Open (
    UINT16          channel
    mcx_A429ChannelInfo*  channelInfo
)
```

Parameters

channel	Arinc429 channel number
*mcx_A429ChannelInfo**	A pointer to channel information

Description

Mode: Ready
This function opens the specified channel and returns updated information on the specified channel.
This function must be used for each channel before communicating with the it in order to prepare it for Tx Rx operations (bring the channel out of reset state).
Channel info struct specification can be found in Appendix B of this document as well as in McxAPI.h file.

Notes

Currently, Arinc429 capabilities applies only to PMC (PCI) cards.

5.56 mcx_A429_Channel_Close

INT16 mcx_A429_Channel_Close	**(**		
	UINT16		***channel***
	)		

Parameters

channel	Arinc429 channel number

Description

Mode: Ready
This function closes the specified channel.
Once a channel is closed it cannot perform Arinc429 operations until mcx_A429_Channel_Open (..) it invoked.

Notes

Currently, Arinc429 capabilities applies only to PMC (PCI) cards.

5.57 mcx_A429_Channel_SetConfigRegister

```
INT16 mcx_A429_Channel_SetConfigRegister (
    UINT16      channel
    UINT32      chanFlags
)
```

Parameters

channel	Arinc429 channel number
chanFlags	Channel flags by the Tx and Rx vectors specified below

Description

Mode: Ready
This function sets the configuration for the specified channel.
The channel flags are described below and can be used with the following constants, also defined in McxAPI.h file.

```
// Following bits are same for TX and RX modes:
#define MCX_A429_CFG_HIGH_RATE          0x0001  ///< Data rate: 100 KHz
#define MCX_A429_CFG_LOW_RATE           0x0000  ///< Data rate: 12.5 KHz
#define MCX_A429_CFG_MASK_RATE          0x0001
#define MCX_A429_CFG_PARITY_NONE        0x0000  ///< Data parity: none. MSB can be
used as data.
#define MCX_A429_CFG_PARITY_EVEN        0x0002  ///< Data parity: even
#define MCX_A429_CFG_PARITY_ODD         0x0006  ///< Data parity: odd
#define MCX_A429_CFG_MASK_PARITY        0x0006  //   Mask for parity bits

#define MCX_A429_NUMBER_OF_WORDS_MASK   0x00FF0000  //   Number of words in FIFO mask
- Bits 16-23
#define MCX_A429_FIFO_FULL                0x01000000  //   FIFO full
#define MCX_A429_FIFO_EMPTY               0x02000000  //   FIFO empty
#define MCX_A429_RX_LABEL_TABLE_READY    0x04000000  //   Labels table ready

// Following bits apply only for RX mode:
#define MCX_A429_CFG_RX_LABEL_MATCH     0x0008  ///< enable label matching
#define MCX_A429_CFG_MASK_RX_LABELS     0x0008
#define MCX_A429_CFG_RX_DECODER_ENABLE  0x0010
#define MCX_A429_CFG_RX_DECODER_DISABLE 0x0000
#define MCX_A429_CFG_MASK_RX_DECODER    0x0070
```

Tx Configuration

Tx Control and Status : (need to set once during system bring up)
Bit 0 – if '1' perform 100Khz messages, if '0' do it in 12.5Khz.
Bit 1 – if '0' then no parity is used.
Bit 2..1 – if "01" then Even parity, if "11" then Odd parity.
Read this register to get its status:
Bit 24 – if '1' Tx FIFO is Full.
Bit 25 – if '1' Tx FIFO is empty.
Bit 23..16 – Number of words in the Tx FIFO. 0 to 255.

Rx Configuration

Rx Control and Status : (need to set once during system bring up)
Bit 0 – if '1' perform 100Khz messages, if '0' do it in 12.5Khz.
Bit 1 – if '0' then no parity is used.
Bit 2..1 – if "01" then Even parity, if "11" then Odd parity.
Bit 3 – '1' Enable Label Recognition,
Bit 4 – '1' Rx Core Decoder Enabled
(ARINC bit 9 must match Rx Control bit 5 and ARINC bit 10 must match Rx Control bit 6)
Bit 5 and 6 – Used for Decoder Enable matching.

Read this register to get its status:
Bit 24 – if '1' Rx FIFO is Full.
Bit 25 – if '1' Rx FIFO is empty.
Bit 26 – '1' when Labels Table Ready for writing.
Bit 23..16 – Number of words in the Rx FIFO - 0 to 255.

Notes

Currently, Arinc429 capabilities applies only to PMC (PCI) cards.

5.58 mcx_A429_Channel_GetConfigRegister

```
INT16 mcx_A429_Channel_GetConfigRegister (
                                  UINT16      channel
                                  UINT32*     chanFlags
                                 )
```

Parameters

channel	Arinc429 channel number
chanFlags	Pointer to channel flags by the Tx and Rx vectors specified below

Description

Mode: Ready
This function gets the configuration for the specified channel.
The channel flags are described below and can be used with the following constants, also defined in McxAPI.h file.

```
// Following bits are same for TX and RX modes:
#define MCX_A429_CFG_HIGH_RATE          0x0001  ///< Data rate: 100 KHz
#define MCX_A429_CFG_LOW_RATE           0x0000  ///< Data rate: 12.5 KHz
#define MCX_A429_CFG_MASK_RATE          0x0001
#define MCX_A429_CFG_PARITY_NONE        0x0000  ///< Data parity: none. MSB can be
used as data.
#define MCX_A429_CFG_PARITY_EVEN        0x0002  ///< Data parity: even
#define MCX_A429_CFG_PARITY_ODD         0x0006  ///< Data parity: odd
#define MCX_A429_CFG_MASK_PARITY        0x0006  //   Mask for parity bits

#define MCX_A429_NUMBER_OF_WORDS_MASK   0x00FF0000  //   Number of words in FIFO mask
- Bits 16-23
#define MCX_A429_FIFO_FULL                0x01000000  //   FIFO full
#define MCX_A429_FIFO_EMPTY               0x02000000  //   FIFO empty
#define MCX_A429_RX_LABEL_TABLE_READY    0x04000000  //   Labels table ready

// Following bits apply only for RX mode:
#define MCX_A429_CFG_RX_LABEL_MATCH     0x0008  ///< enable label matching
#define MCX_A429_CFG_MASK_RX_LABELS     0x0008
#define MCX_A429_CFG_RX_DECODER_ENABLE  0x0010
#define MCX_A429_CFG_RX_DECODER_DISABLE 0x0000
#define MCX_A429_CFG_MASK_RX_DECODER    0x0070
```

Tx Configuration

Tx Control and Status : (need to set once during system bring up)
Bit 0 – if '1' perform 100Khz messages, if '0' do it in 12.5Khz.
Bit 1 – if '0' then no parity is used.
Bit 2..1 – if "01" then Even parity, if "11" then Odd parity.
Read this register to get its status:
Bit 24 – if '1' Tx FIFO is Full.
Bit 25 – if '1' Tx FIFO is empty.
Bit 23..16 – Number of words in the Tx FIFO. 0 to 255.

Rx Configuration

Rx Control and Status : (need to set once during system bring up)
Bit 0 – if '1' perform 100Khz messages, if '0' do it in 12.5Khz.
Bit 1 – if '0' then no parity is used.
Bit 2..1 – if "01" then Even parity, if "11" then Odd parity.
Bit 3 – '1' Enable Label Recognition,
Bit 4 – '1' Rx Core Decoder Enabled
(ARINC bit 9 must match Rx Control bit 5 and ARINC bit 10 must match Rx Control bit 6)
Bit 5 and 6 – Used for Decoder Enable matching.

Read this register to get its status:
Bit 24 – if '1' Rx FIFO is Full.
Bit 25 – if '1' Rx FIFO is empty.
Bit 26 – '1' when Labels Table Ready for writing.
Bit 23..16 – Number of words in the Rx FIFO - 0 to 255.

Notes

Currently, Arinc429 capabilities applies only to PMC (PCI) cards.

5.59 mcx_A429_Channel_GetStatusRegister

```
INT16 mcx_A429_Channel_GetStatusRegister (
                                    UINT16      channel
                                    UINT32*     chanStats
                                  )
```

Parameters

channel	Arinc429 channel number
chanFlags	Pointer to channel status registers by the Tx and Rx vectors specified below

Description

Mode: Ready+ Runtime

This function gets the status registers for the specified channel. It can be used to determined the FIFO state and number of received words in case of Rx bus.

The channel flags are described below and can be used with the following constants, also defined in McxAPI.h file.

```
// Following bits are same for TX and RX modes:
#define MCX_A429_CFG_HIGH_RATE          0x0001  ///< Data rate: 100 KHz
#define MCX_A429_CFG_LOW_RATE           0x0000  ///< Data rate: 12.5 KHz
#define MCX_A429_CFG_MASK_RATE          0x0001
#define MCX_A429_CFG_PARITY_NONE        0x0000  ///< Data parity: none. MSB can be
used as data.
#define MCX_A429_CFG_PARITY_EVEN        0x0002  ///< Data parity: even
#define MCX_A429_CFG_PARITY_ODD         0x0006  ///< Data parity: odd
#define MCX_A429_CFG_MASK_PARITY        0x0006  //   Mask for parity bits

#define MCX_A429_NUMBER_OF_WORDS_MASK   0x00FF0000  //   Number of words in FIFO mask
- Bits 16-23
#define MCX_A429_FIFO_FULL                0x01000000  //   FIFO full
#define MCX_A429_FIFO_EMPTY               0x02000000  //   FIFO empty
#define MCX_A429_RX_LABEL_TABLE_READY    0x04000000  //   Labels table ready

// Following bits apply only for RX mode:
#define MCX_A429_CFG_RX_LABEL_MATCH     0x0008  ///< enable label matching
#define MCX_A429_CFG_MASK_RX_LABELS     0x0008
#define MCX_A429_CFG_RX_DECODER_ENABLE  0x0010
#define MCX_A429_CFG_RX_DECODER_DISABLE 0x0000
#define MCX_A429_CFG_MASK_RX_DECODER    0x0070
```

Tx Configuration

Tx Control and Status : (need to set once during system bring up)
Bit 0 – if '1' perform 100Khz messages, if '0' do it in 12.5Khz.
Bit 1 – if '0' then no parity is used.
Bit 2..1 – if "01" then Even parity, if "11" then Odd parity.
Read this register to get its status:
Bit 24 – if '1' Tx FIFO is Full.
Bit 25 – if '1' Tx FIFO is empty.
Bit 23..16 – Number of words in the Tx FIFO. 0 to 255.

Rx Configuration

Rx Control and Status : (need to set once during system bring up)
Bit 0 – if '1' perform 100Khz messages, if '0' do it in 12.5Khz.
Bit 1 – if '0' then no parity is used.
Bit 2..1 – if "01" then Even parity, if "11" then Odd parity.
Bit 3 – '1' Enable Label Recognition,
Bit 4 – '1' Rx Core Decoder Enabled
(ARINC bit 9 must match Rx Control bit 5 and ARINC bit 10 must match Rx Control bit 6)
Bit 5 and 6 – Used for Decoder Enable matching.

Read this register to get its status:
Bit 24 – if '1' Rx FIFO is Full.
Bit 25 – if '1' Rx FIFO is empty.
Bit 26 – '1' when Labels Table Ready for writing.
Bit 23..16 – Number of words in the Rx FIFO - 0 to 255.

Notes

Currently, Arinc429 capabilities applies only to PMC (PCI) cards.

5.60 mcx_A429_Channel_ Receive

```
INT16 mcx_A429_Channel_ Receive (
    UINT16      channel
    UINT32      bufferSize
    UINT32*     buffer
    UINT32*     numberOfReceivedWords
)
```

Parameters

channel	Arinc429 channel number
bufferSize	Size of assigned buffer
buffer	A pointer to the buffer in which the data returned
numberOfReceivedWords	Number of words actually received on the bus

Description

Mode: Ready + Runtime
This function gets the data received and number of words received on the bus, it is returned in a buffer.
This function can be coupled with mcx_A429_GetRxWordsPending(..) to check if data is received and waiting in the FIFO

Notes

Currently, Arinc429 capabilities applies only to PMC (PCI) cards.

5.61 mcx_A429_Channel_ Send

```
INT16 mcx_A429_Channel_ Send (
    UINT16      channel
    UINT32      bufferSize
    UINT32*     buffer
    UINT32*     numberOfWrittenWords
)
```

Parameters

channel	Arinc429 channel number
bufferSize	Size of assigned buffer
buffer	A pointer to the data buffer to send
numberOfWrittenWords	Number of words actually sent on the bus

Description

Mode: Ready + Runtime
This function transmits data buffer on the bus and returns the number of words sent.

Notes

Currently, Arinc429 capabilities applies only to PMC (PCI) cards.

5.62 mcx_A429_GetRxWordsPending

```
INT16 mcx_A429_GetRxWordsPending (
    UINT16      channel
    UINT32*     numberOfWords
)
```

Parameters

channel	Arinc429 channel number
numberOfWords	Number of pending words in the Rx FIFO

Description

Mode: Ready + Runtime
This function gets the number of words pending on the Rx bus.

Notes

Currently, Arinc429 capabilities applies only to PMC (PCI) cards.

5.63 mcx_A429_Card_SetConfiguration

INT16 mcx_A429_Card_Configuration	**(**	
	UINT32	***cardFlags***
	)	

Parameters

cardFlags	A vector signaling the card on the setup. The following constants can be found in McxAPI.h file:
	`MCX_A429_CARD_DISABLE_INTERNAL_LOOPBACK 0x00000000`
	`MCX_A429_CARD_ENABLE_INTERNAL_LOOPBACK 0x00000001`

Description

Mode: Ready
This function sets internal loopback on and off on the card for all Arinc429 Tx and Rx data buses.
Once the card is set to perform internal loopback, external Tx Rx are disabled and vice versa.
The initial state of the card is internal loopback is disable (accepting external Tx and Rx data).

The internal loopback loops channel 0 to channel 2 and channel 1 to channel 3.

Notes

Currently, Arinc429 capabilities applies only to PMC (PCI) cards.

5.64 mcx_GetPciProductIds

```
INT16 mcx_GetPciProductIds (
    S32BIT*   pIds
    S16BIT*   numberOfCardsFound
)
```

Parameters

pIds	Pointer to the beginning of an array of Product IDs found for PCI cards.
numberOfCardsFound	Returns the number of cards found

Description

Mode: Ready+ Running
This function returns array of Product IDs of all PCI cards identified on the PCI slots and the number of found cards.
For example, if a PCI card of 1553 + RS485 (identifies as 2 cards on the PCI since it contains 2 IP cores) and in addition Arinc429 card with 8Tx channels and 16Rx channels, the pIds will return 3 product ids (2002, 2002 and 429) and the numberOfCards will return 3.

5.65 mcx_A429_Pci_Channel_GetCount

```
INT16 mcx_A429_Pci_Channel_GetCount (
    U16BIT      card
    UINT32*     numOfChannels
)
```

Parameters

card	Card number to get the number of channels it contains
numOfChannels	Returns the number of exiting ARINC429 channels on the card

Description

Mode: Ready + Running
This function returns the number of detected Arinc429 channel on the card specified

Notes

Currently, Arinc429 capabilities applies only to PCI cards.

5.66 mcx_A429_Pci_Channel_GetInformation

```
INT16 mcx_A429_Pci_Channel_GetInformation (
                                    U16BIT              card
                                    U16BIT              channel
                                    mcx_A429ChannelInfo* channelInfo
                                   )
```

Parameters

card	Card number
channel	Arinc429 channel number
*mcx_A429ChannelInfo**	A pointer to channel information

Description

Mode: Ready
This function returns information on the specified channel within a specified card.
Channel info struct specification can be found in Appendix B of this document as well as in McxAPI.h file

5.67 mcx_A429_Pci_Channel_Open

```
INT16 mcx_A429_Pci_Channel_Open (
    U16BIT              card
    U16BIT              channel
    mcx_A429ChannelInfo* channelInfo
)
```

Parameters

channel	Arinc429 channel number
*mcx_A429ChannelInfo**	A pointer to channel information

Description

Mode: Ready

This function opens the specified channel and returns updated information on the specified channel within the specified card.

This function must be used for each channel before communicating with the it in order to prepare it for Tx Rx operations (bring the channel out of reset state).

Channel info struct specification can be found in Appendix B of this document as well as in McxAPI.h file.

5.68 mcx_A429_Pci_Channel_Close

INT16 mcx_A429_Pci_Channel_Close	**(**	
	U16BIT	***Card***
	U16BIT	***channel***
	)	

Parameters

card	Card number
channel	Arinc429 channel number

Description

Mode: Ready
This function closes the specified channel within specified card.
Once a channel is closed it cannot perform Arinc429 operations until mcx_A429_Channel_Open (..) it invoked.

5.69 mcx_A429_Pci_Channel_SetConfigRegister

```
INT16
mcx_A429_Pci_Channel_SetConfigRegister (
    U16BIT  Card
    U16BIT  channel
    U32BIT  chanFlags
)
```

Parameters

card	Card number
channel	Arinc429 channel number
chanFlags	Channel flags by the Tx and Rx vectors specified below

Description

Mode: Ready
This function sets the configuration for the specified channel within specified card.
The channel flags are described below and can be used with the following constants, also defined in McxAPI.h file.

```
// Following bits are same for TX and RX modes:
#define MCX_A429_CFG_HIGH_RATE          0x0001  ///< Data rate: 100 KHz
#define MCX_A429_CFG_LOW_RATE           0x0000  ///< Data rate: 12.5 KHz
#define MCX_A429_CFG_MASK_RATE          0x0001
#define MCX_A429_CFG_PARITY_NONE        0x0000  ///< Data parity: none. MSB can be
used as data.
#define MCX_A429_CFG_PARITY_EVEN        0x0002  ///< Data parity: even
#define MCX_A429_CFG_PARITY_ODD         0x0006  ///< Data parity: odd
#define MCX_A429_CFG_MASK_PARITY        0x0006  //   Mask for parity bits

#define MCX_A429_NUMBER_OF_WORDS_MASK   0x00FF0000  //   Number of words in FIFO mask
- Bits 16-23
#define MCX_A429_FIFO_FULL                0x01000000  //   FIFO full
#define MCX_A429_FIFO_EMPTY               0x02000000  //   FIFO empty
#define MCX_A429_RX_LABEL_TABLE_READY    0x04000000  //   Labels table ready

// Following bits apply only for RX mode:
#define MCX_A429_CFG_RX_LABEL_MATCH     0x0008  ///< enable label matching
#define MCX_A429_CFG_MASK_RX_LABELS     0x0008
#define MCX_A429_CFG_RX_DECODER_ENABLE  0x0010
#define MCX_A429_CFG_RX_DECODER_DISABLE 0x0000
#define MCX_A429_CFG_MASK_RX_DECODER    0x0070
```

Tx Configuration

Tx Control and Status : (need to set once during system bring up)
Bit 0 – if '1' perform 100Khz messages, if '0' do it in 12.5Khz.
Bit 1 – if '0' then no parity is used.
Bit 2..1 – if "01" then Even parity, if "11" then Odd parity.
Read this register to get its status:
Bit 24 – if '1' Tx FIFO is Full.
Bit 25 – if '1' Tx FIFO is empty.
Bit 23..16 – Number of words in the Tx FIFO. 0 to 255.

Rx Configuration

Rx Control and Status : (need to set once during system bring up)
Bit 0 – if '1' perform 100Khz messages, if '0' do it in 12.5Khz.
Bit 1 – if '0' then no parity is used.
Bit 2..1 – if "01" then Even parity, if "11" then Odd parity.
Bit 3 – '1' Enable Label Recognition,
Bit 4 – '1' Rx Core Decoder Enabled
(ARINC bit 9 must match Rx Control bit 5 and ARINC bit 10 must match Rx Control bit 6)
Bit 5 and 6 – Used for Decoder Enable matching.

Read this register to get its status:
Bit 24 – if '1' Rx FIFO is Full.
Bit 25 – if '1' Rx FIFO is empty.
Bit 26 – '1' when Labels Table Ready for writing.
Bit 23..16 – Number of words in the Rx FIFO - 0 to 255.

5.70 mcx_A429_Pci_Channel_GetConfigRegister

```
INT16
mcx_A429_Pci_Channel_GetConfigRegister (
    U16BIT      Card
    U16BIT      channel
    U32BIT*     chanFlags
)
```

Parameters

card	Card number
channel	Arinc429 channel number
chanFlags	Pointer to channel flags by the Tx and Rx vectors specified below

Description

Mode: Ready
This function gets the configuration for the specified channel within specified card.
The channel flags are described below and can be used with the following constants, also defined in McxAPI.h file.

```
// Following bits are same for TX and RX modes:
#define MCX_A429_CFG_HIGH_RATE          0x0001  ///< Data rate: 100 KHz
#define MCX_A429_CFG_LOW_RATE           0x0000  ///< Data rate: 12.5 KHz
#define MCX_A429_CFG_MASK_RATE          0x0001
#define MCX_A429_CFG_PARITY_NONE        0x0000  ///< Data parity: none. MSB can be
used as data.
#define MCX_A429_CFG_PARITY_EVEN        0x0002  ///< Data parity: even
#define MCX_A429_CFG_PARITY_ODD         0x0006  ///< Data parity: odd
#define MCX_A429_CFG_MASK_PARITY        0x0006  //   Mask for parity bits

#define MCX_A429_NUMBER_OF_WORDS_MASK   0x00FF0000  //   Number of words in FIFO mask
- Bits 16-23
#define MCX_A429_FIFO_FULL                0x01000000  //   FIFO full
#define MCX_A429_FIFO_EMPTY               0x02000000  //   FIFO empty
#define MCX_A429_RX_LABEL_TABLE_READY    0x04000000  //   Labels table ready

// Following bits apply only for RX mode:
#define MCX_A429_CFG_RX_LABEL_MATCH     0x0008  ///< enable label matching
#define MCX_A429_CFG_MASK_RX_LABELS     0x0008
#define MCX_A429_CFG_RX_DECODER_ENABLE  0x0010
#define MCX_A429_CFG_RX_DECODER_DISABLE 0x0000
#define MCX_A429_CFG_MASK_RX_DECODER    0x0070
```

Tx Configuration

Tx Control and Status : (need to set once during system bring up)
Bit 0 – if '1' perform 100Khz messages, if '0' do it in 12.5Khz.
Bit 1 – if '0' then no parity is used.
Bit 2..1 – if "01" then Even parity, if "11" then Odd parity.
Read this register to get its status:
Bit 24 – if '1' Tx FIFO is Full.
Bit 25 – if '1' Tx FIFO is empty.
Bit 23..16 – Number of words in the Tx FIFO. 0 to 255.

Rx Configuration

Rx Control and Status : (need to set once during system bring up)
Bit 0 – if '1' perform 100Khz messages, if '0' do it in 12.5Khz.
Bit 1 – if '0' then no parity is used.
Bit 2..1 – if "01" then Even parity, if "11" then Odd parity.
Bit 3 – '1' Enable Label Recognition,
Bit 4 – '1' Rx Core Decoder Enabled
(ARINC bit 9 must match Rx Control bit 5 and ARINC bit 10 must match Rx Control bit 6)
Bit 5 and 6 – Used for Decoder Enable matching.

Read this register to get its status:
Bit 24 – if '1' Rx FIFO is Full.
Bit 25 – if '1' Rx FIFO is empty.
Bit 26 – '1' when Labels Table Ready for writing.
Bit 23..16 – Number of words in the Rx FIFO - 0 to 255.

5.71 mcx_A429_Pci_Channel_GetStatusRegister

```
INT16
mcx_A429_Pci_Channel_GetStatusRegister (
    U16BIT      Card
    U16BIT      channel
    U32BIT*     chanStats
)
```

Parameters

Card	Card number
channel	Arinc429 channel number
chanFlags	Pointer to channel status registers by the Tx and Rx vectors specified below

Description

Mode: Ready+ Runtime

This function gets the status registers for the specified channel within specified card. It can be used to determined the FIFO state and number of received words in case of Rx bus.

The channel flags are described below and can be used with the following constants, also defined in McxAPI.h file.

```
// Following bits are same for TX and RX modes:
#define MCX_A429_CFG_HIGH_RATE          0x0001  ///< Data rate: 100 KHz
#define MCX_A429_CFG_LOW_RATE           0x0000  ///< Data rate: 12.5 KHz
#define MCX_A429_CFG_MASK_RATE          0x0001
#define MCX_A429_CFG_PARITY_NONE        0x0000  ///< Data parity: none. MSB can be
used as data.
#define MCX_A429_CFG_PARITY_EVEN        0x0002  ///< Data parity: even
#define MCX_A429_CFG_PARITY_ODD         0x0006  ///< Data parity: odd
#define MCX_A429_CFG_MASK_PARITY        0x0006  //   Mask for parity bits

#define MCX_A429_NUMBER_OF_WORDS_MASK   0x00FF0000  //   Number of words in FIFO mask
- Bits 16-23
#define MCX_A429_FIFO_FULL                0x01000000  //   FIFO full
#define MCX_A429_FIFO_EMPTY               0x02000000  //   FIFO empty
#define MCX_A429_RX_LABEL_TABLE_READY    0x04000000  //   Labels table ready

// Following bits apply only for RX mode:
#define MCX_A429_CFG_RX_LABEL_MATCH     0x0008  ///< enable label matching
#define MCX_A429_CFG_MASK_RX_LABELS     0x0008
#define MCX_A429_CFG_RX_DECODER_ENABLE  0x0010
#define MCX_A429_CFG_RX_DECODER_DISABLE 0x0000
#define MCX_A429_CFG_MASK_RX_DECODER    0x0070
```

Tx Configuration

Tx Control and Status : (need to set once during system bring up)
Bit 0 – if '1' perform 100Khz messages, if '0' do it in 12.5Khz.
Bit 1 – if '0' then no parity is used.
Bit 2..1 – if "01" then Even parity, if "11" then Odd parity.
Read this register to get its status:
Bit 24 – if '1' Tx FIFO is Full.
Bit 25 – if '1' Tx FIFO is empty.
Bit 23..16 – Number of words in the Tx FIFO. 0 to 255.

Rx Configuration

Rx Control and Status : (need to set once during system bring up)
Bit 0 – if '1' perform 100Khz messages, if '0' do it in 12.5Khz.
Bit 1 – if '0' then no parity is used.
Bit 2..1 – if "01" then Even parity, if "11" then Odd parity.
Bit 3 – '1' Enable Label Recognition,
Bit 4 – '1' Rx Core Decoder Enabled
(ARINC bit 9 must match Rx Control bit 5 and ARINC bit 10 must match Rx Control bit 6)
Bit 5 and 6 – Used for Decoder Enable matching.

Read this register to get its status:
Bit 24 – if '1' Rx FIFO is Full.
Bit 25 – if '1' Rx FIFO is empty.
Bit 26 – '1' when Labels Table Ready for writing.
Bit 23..16 – Number of words in the Rx FIFO - 0 to 255.

5.72 mcx_A429_Pci_Channel_ Receive

```
INT16 mcx_A429_Pci_Channel_ Receive (
    U16BIT      Card
    U16BIT      channel
    U32BIT      bufferSize
    U32BIT*     buffer
    U32BIT*     numberOfReceivedWords
)
```

Parameters

card	Card number
channel	Arinc429 channel number
bufferSize	Size of assigned buffer
buffer	A pointer to the buffer in which the data returned
numberOfReceivedWords	Number of words actually received on the bus

Description

Mode: Ready + Runtime
This function gets the data received and number of words received on the bus, it is returned in a buffer.
This function can be coupled with mcx_A429_GetRxWordsPending(..) to check if data is received and waiting in the FIFO

5.73 mcx_A429_Pci_Channel_ Send

```
INT16 mcx_A429_Pci_Channel_ Send (
    U16BIT      Card
    U16BIT      channel
    U32BIT      bufferSize
    U32BIT*     buffer
    U32BIT*     numberOfWrittenWords
)
```

Parameters

card	Card number
channel	Arinc429 channel number
bufferSize	Size of assigned buffer
buffer	A pointer to the data buffer to send
numberOfWrittenWords	Number of words actually sent on the bus

Description

Mode: Ready + Runtime
This function transmits data buffer on the bus and returns the number of words sent.

5.74 mcx_A429_Pci_GetRxWordsPending

```
INT16 mcx_A429_Pci_GetRxWordsPending (
    U16BIT      Card
    U16BIT      channel
    U32BIT*     numberOfWords
)
```

Parameters

card	Card number
channel	Arinc429 channel number
numberOfWords	Number of pending words in the Rx FIFO

Description

Mode: Ready + Runtime
This function gets the number of words pending on the Rx bus.

5.75 mcx_A429_Pci_Card_SetConfiguration

```
INT16 mcx_A429_Pci_Card_Configuration (
    U16BIT      Card
    UINT32      cardFlags
)
```

Parameters

card	Card number
cardFlags	A vector signaling the card on the setup. The following constants can be found in McxAPI.h file: MCX_A429_CARD_DISABLE_INTERNAL_LOOPBACK 0x00000000 MCX_A429_CARD_ENABLE_INTERNAL_LOOPBACK 0x00000001

Description

Mode: Ready

This function sets internal loopback on and off on the card specified for all Arinc429 Tx and Rx data buses.
Once the card is set to perform internal loopback, external Tx Rx are disabled and vice versa.
The initial state of the card is internal loopback is disable (accepting external Tx and Rx data).

The internal loopback loops channel 0 to channel 2 and channel 1 to channel 3.

5.76 mcx_GetLicenseDescription

```
INT16 mcx_GetLicenseDescription (
  UINT16   deviceId
  char*    description
)
```

Parameters

deviceId	Unique Device ID 0 - (sitalMaximum_DEVICES - 1)
description	A pointer in which the license features returned

Description

Mode: Ready + Runtime

This function gets a string concatenating all available features and their license state. For each feature an indication of 'Licensed' or 'Unlicensed' provided. The features are concatenated as a single characters' string, separated by ' & ' character. For example: "`Licensed - Mil-Std-1553 & Unlicensed - PP194 & Licensed - H009 & "` etc.

5.77 mcx_SetCyberAttack

```
INT16 mcx_SetCyberAttack (
    UINT16  deviceId
    UINT16  cyberAttackType
    UINT16  triggerCommand
)
```

Parameters

deviceId	Unique Device ID 0 - (sitalMaximum_DEVICES - 1)
cyberAttackType	Attack types of the following mcx_No_Attack 0x0000 mcx_Attack_Mode1 0x0001 mcx_Attack_Mode2 0x0002 mcx_Attack_Mode3 0x0003
triggerCommand	Triggering command for Attack Mode 3

Description

Mode: Ready + Runtime

This function sets the firmware to No Attack mode or to any of the following modes:

```
mcx_No_Attack        0x0000 // no Cyber-attack mode
mcx_Attack_Mode1     0x0001 // Time delayed attack mode with 65 ms steps of delay
mcx_Attack_Mode2     0x0002 // Time delayed attack mode with 100 us steps of delay
mcx_Attack_Mode3     0x0003 // Trigger Message delay mode
```

for elaboration about different mode types, see section 2.2 (Set Cyber Emulation) in this document.
Once setting the attack to an active attack type (other than 0 = No Attack), the firmware automatically acts upon selected settings; time to first attack (is set by function 'mcx_SetFrameTime(..)'), messages gaps and trigger command (attack type 3).

5.78 mcx_TestExternalLoopback_DevicetoDevice

```
INT16 mcx_TestExternalLoopback_DevicetoDevice (
                                    UINT16      device0
                                    UINT16      Device1
                                    UINT16 *    resultD0A
                                    UINT16 *    resultD0B
                                    UINT16 *    resultD1A
                                    UINT16 *    resultD1B
                                    Bool*       badDataFound
                                    )
```

Parameters

Device0	Unique Device ID 0 - (sitalMaximum_DEVICES - 1).
Device1	Unique Device ID 0 - (sitalMaximum_DEVICES - 1).
resultD0A	A pointer returning the results BC side for device 0, Bus A
resultD0B	A pointer returning the results BC side for device 0, Bus B
resultD0A	A pointer returning the results BC side for device 1, Bus A
resultD0A	A pointer returning the results BC side for device 1, Bus B
badDataFound	A pointer returning true if received data is different than transmitted (from RT)

Description

Mode: Ready

This test performs the following test:

- transmit 0xC20 command (RT to BC, 32 words) on bus A from BC device to MultiRT device and then on bus B
- RT is simulated in MultiRT side, data is incremental from 0x5555
- command is transmitted once
- data is checked in the BC side
- then devices are switched, repeating the test
- this test is a blocking command
- 4 results are returned - device0A, device0B, device1A, device1B

Important Notes

- The code implementation can be found in this document, appendix 7.3.
- Pre-requisit for theis function is that
    - 2 devices at least exist on the device
    - Both devices are initialized prior to using this function
    - Relevant wiring required, see document **{TBD}**

5.79 mcx_Send_AsynchMsg1

```
INT16 mcx_Send_AsynchMsg1 (
                            UINT16      deviceId
                            UINT16      command
                            UINT16      options
                            UINT16      statusWord
                            UINT16*     buffer
                            UINT16      bufferSize
                           )
```

Parameters

deviceId Unique Device ID 0 - (sitalMaximum_DEVICES - 1)

command Unique, MIL-STD-1553 Command word that this Element services.
Currently supported BC2RT and RT2BC commands (no RT2RT)

options Element's optional configuration parameter. The option is a logic OR combination of the following configs:

Bus	BusA = 0x80. BusB =0
Pp194 messagetype	For pp194 – 0x0004. For 1553 - 0

statusWord Status for simulated (Multi) RT / RIU responses.

Buffer A pointer to the beginning of the data buffer for this message

bufferSize Buffer size to apply

Description

Mode: Ready & Runtime
This function create an Async message and sends it. The message can be created and run when bus is idle and when other frames and messages are running.
Once this message is created it is transmitted instantly, serving as Async (High Prioirty) message.
Additional Async message can be sent using mcx_Send_AsynchMsg2.

For getting the Async message's results, use mcx_Get_Asynch1_Results(..) function.

5.80 mcx_Send_AsynchMsg2

```
INT16 mcx_Send_AsynchMsg2 (
                            UINT16      deviceId
                            UINT16      command
                            UINT16      options
                            UINT16      statusWord
                            UINT16*     buffer
                            UINT16      bufferSize
                          )
```

Parameters

deviceId Unique Device ID 0 - (sitalMaximum_DEVICES - 1)

command Unique, MIL-STD-1553 Command word that this Element services.
Currently supported BC2RT and RT2BC commands (no RT2RT)

options Element's optional configuration parameter. The option is a logic OR combination of the following configs:

Bus	BusA = 0x80. BusB =0
Pp194 messagetype	For pp194 – 0x0004. For 1553 - 0

statusWord Status for simulated (Multi) RT / RIU responses.

Buffer A pointer to the beginning of the data buffer for this message

bufferSize Buffer size to apply

Description

Mode: Ready & Runtime
This function create an Async message and sends it. The message can be created and run when bus is idle and when other frames and messages are running.
Once this message is created it is transmitted instantly, serving as Async (High Prioirty) message.

For getting the Async message's results, use mcx_Get_Asynch2_Results(..) function.

5.81 mcx_Get_Asynch1_Results

```
INT16 mcx_Get_Asynch1_Results    (
                                   UINT16      deviceId
                                   UINT16*     blockStatusWord
                                   WORD*       buffer
                                   UINT16      bufferSize
                                   UINT16*     Status
                                   UINT16*     tag
                                 )
```

Parameters

deviceId	Unique Device ID 0 - (sitalMaximum_DEVICES - 1)
blockStatusWord	A pointer that returns the BSW, see elaboration below
buffer	A pointer to the returned buffer
bufferSize	Size of buffer to return
Status	A pointer returning the 1553 Status of the message
Tag	The 16 LSBs of the 32 bit time tag counter

Description

Mode: Ready & Runtime

This function gets the results of a transmission of Async message 1. Message results comprise the message words that were actually transmitted along the internal bus together with the statistics (diagnostics) of the transmitted message. The diagnostics include an indicator of whether the message transmission was successful, status words, the data payload that was actually transmitted on the bus.
The difference between this function and the Word Monitor family of functions is that the Word Monitor sits on the bus in the Tester device and simply records all the words that go by; the Word Monitor has no concept of BusLists or Elements. This function, on the other hand, returns a specific Element's results from the specified BusList.

Message State Table

	Name	Bit	Description
0	Time Tag Word 16 LSBs. (Gap mode)	15..0	16 LSBs of the real time counter. Written by core when the message started.
0	Frame Number (Rate mode)	15..0	Frame number when this message was transmitted. Frame number is incremented every EOF. It is recommended to init this value to 0xFFFF before run.
1	Message findings	15	End Of Message – Set to '1' by the core when the message has been complete.
		14	Start Of Message - Set to '1' by the core when the message has been started. In most cases, this bit is stuck at '1' after end of message if there is a 1553 bus-coupling problem.

		13	'0' – Was sent on Bus A. '1' – Was sent on Bus B.
		12	'1' – Error was found in the message. Bits 10, 9, 8, 3, 2, 1, 0 indicate cause of error.
		11	Status Set. One of the status bits (excluding BCST bit) of the status return was '1'. Masking ignored. BCST bit works in either mask mode or compare mode. In mask mode it works like other mask bits on the BCST bit. In compare mode, Status set occurs if BCST bit is different from bit 5 of BC control word.
		10	Format Error. The returned echo from the RT contained 1553 violations. See bits 3, 2, 1, 0 for a more accurate guess of the source of the problem.
		9	Response timeout. The RT responded too late or didn't respond at all. In PP194 – The RIU did not respond properly.
		8	Loop back failed. The nature of 1553 bus is that every word transmitted, is also echoed back. The core verifies that the echo is correct and equal to the transmitted word. If not, this bit is set to '1'. Also set in messages with error injected. Tip: The source of this type of error could be transceiver fault, or bus coupling problem. In PP194 –Loop back Failed.
		7	Unmasked Status bit set. This bit will be set to '1' if one of the status bits are set high and its appropriate mask bit in the BC control word is unmasked ('0'). BCST bit influences only in mask mode. See registers section for description of BCST bit.
		6..5	Number of retries done for this message. "11" is 3, "10" is 2…
		4	Good data block received by TestersChoice, waiting in Data Block. '1' – after an RT-BC, RT2RT, and Transmit Mode code with data commands if the message ended OK. '0' – after other message types, or if the above type of message was invalid. '0' – for received words that did not match the expected values if "Write Verify" mode is enabled for the message. Loop back test failure does not cripple this bit result. In PP194 – Both phases completed successfully and a real RIU sent its status and saved to memory.
		3	'1' indicates the RT responded with wrong RT address. In PP194 – RIU status respond with wrong RIU address.
		2	'1' indicates that the RT transmitted a wrong number of words. In PP194 – RIU Data phase error.
		1	'1' – Incorrect sync type response by RT. In PP194 – RIU Status phase error.
		0	'1' – Invalid word. Indicates that the RT responded with a word containing 1553 errors. In PP194 – The RIU responded with Manchester / parity error.
2	Received 1st status	15..0	First status received from un-simulated RT. In PP194 – Status bits of status word.
3	Received 2nd status	15..0	Second status received from un-simulated RT.

5.82 mcx_Get_Asynch2_Results

```
INT16 mcx_Get_Asynch2_Results    (
                                   UINT16      deviceId
                                   UINT16*     blockStatusWord
                                   WORD*       buffer
                                   UINT16      bufferSize
                                   UINT16*     Status
                                   UINT16*     tag
                                 )
```

Parameters

deviceId	Unique Device ID 0 - (sitalMaximum_DEVICES - 1)
blockStatusWord	A pointer that returns the BSW, see elaboration below
buffer	A pointer to the returned buffer
bufferSize	Size of buffer to return
Status	A pointer returning the 1553 Status of the message
Tag	The 16 LSBs of the 32 bit time tag counter

Description

Mode: Ready & Runtime

This function gets the results of a transmission of Async message 2. Message results comprise the message words that were actually transmitted along the internal bus together with the statistics (diagnostics) of the transmitted message. The diagnostics include an indicator of whether the message transmission was successful, status words, the data payload that was actually transmitted on the bus.
The difference between this function and the Word Monitor family of functions is that the Word Monitor sits on the bus in the Tester device and simply records all the words that go by; the Word Monitor has no concept of BusLists or Elements. This function, on the other hand, returns a specific Element's results from the specified BusList.

Message State Table

	Name	Bit	Description
0	Time Tag Word 16 LSBs. (Gap mode)	15..0	16 LSBs of the real time counter. Written by core when the message started.
0	Frame Number (Rate mode)	15..0	Frame number when this message was transmitted. Frame number is incremented every EOF. It is recommended to init this value to 0xFFFF before run.
1	Message findings	15	End Of Message – Set to '1' by the core when the message has been complete.
		14	Start Of Message - Set to '1' by the core when the message has been started. In most cases, this bit is stuck at '1' after end of message if there is a 1553 bus-coupling problem.

		13	'0' – Was sent on Bus A. '1' – Was sent on Bus B.
		12	'1' – Error was found in the message. Bits 10, 9, 8, 3, 2, 1, 0 indicate cause of error.
		11	Status Set. One of the status bits (excluding BCST bit) of the status return was '1'. Masking ignored. BCST bit works in either mask mode or compare mode. In mask mode it works like other mask bits on the BCST bit. In compare mode, Status set occurs if BCST bit is different from bit 5 of BC control word.
		10	Format Error. The returned echo from the RT contained 1553 violations. See bits 3, 2, 1, 0 for a more accurate guess of the source of the problem.
		9	Response timeout. The RT responded too late or didn't respond at all. In PP194 – The RIU did not respond properly.
		8	Loop back failed. The nature of 1553 bus is that every word transmitted, is also echoed back. The core verifies that the echo is correct and equal to the transmitted word. If not, this bit is set to '1'. Also set in messages with error injected. Tip: The source of this type of error could be transceiver fault, or bus coupling problem. In PP194 –Loop back Failed.
		7	Unmasked Status bit set. This bit will be set to '1' if one of the status bits are set high and its appropriate mask bit in the BC control word is unmasked ('0'). BCST bit influences only in mask mode. See registers section for description of BCST bit.
		6..5	Number of retries done for this message. "11" is 3, "10" is 2…
		4	Good data block received by TestersChoice, waiting in Data Block. '1' – after an RT-BC, RT2RT, and Transmit Mode code with data commands if the message ended OK. '0' – after other message types, or if the above type of message was invalid. '0' – for received words that did not match the expected values if "Write Verify" mode is enabled for the message. Loop back test failure does not cripple this bit result. In PP194 – Both phases completed successfully and a real RIU sent its status and saved to memory.
		3	'1' indicates the RT responded with wrong RT address. In PP194 – RIU status respond with wrong RIU address.
		2	'1' indicates that the RT transmitted a wrong number of words. In PP194 – RIU Data phase error.
		1	'1' – Incorrect sync type response by RT. In PP194 – RIU Status phase error.
		0	'1' – Invalid word. Indicates that the RT responded with a word containing 1553 errors. In PP194 – The RIU responded with Manchester / parity error.
2	Received 1st status	15..0	First status received from un-simulated RT. In PP194 – Status bits of status word.
3	Received 2nd status	15..0	Second status received from un-simulated RT.

5.83 mcx_Element_UpdateData

INT16 mcx_Element_UpdateData	**(**	
	UINT16	***deviceId***
	UINT16	***busList***
	UINT16	***element***
	)	

Parameters

deviceId	Unique Device ID 0 - (sitalMaximum_DEVICES - 1)
busList	Unique ID of BusList 0 - (MAX_BUSLISTS - 1)
updatedFrame	The Element's index within the Buslist to update it's data

Description

Mode: Running
This function allows the user to update a running bus list element's data during run.
User's data is updated to the bus once on this function call, therefore, it is the user's responsibility to invoke this function on the relevant timing.
In order to update the Element's data once a message is transmitted, use function 'mcx_Get_Buslist_TransmittedElements(..)' to check if a message was transmitted or not.

Mode: Ready
This function updates the data to be transmitted.

Example

```
U8BIT transmitted[64];
U16BIT count = 0;

while(count < times){
        iResult = mcx_Get_Buslist_TransmittedElements(DeviceId, BusList1, transmitted); if (iResult < 0) return iResult;
        if(transmitted[0]) {
                for (UINT16 i = 0x0000; i < 8; i++) datablock1[i] = payload0++;
                iResult = mcx_Element_UpdateData(DeviceId, BusList1, Element1); if (iResult < 0) return iResult;

        }
        if(transmitted[1]) {
                for (UINT16 i = 0x0000; i < 16; i++) datablock2[i] = payload1++;
                iResult = mcx_Element_UpdateData(DeviceId, BusList1, Element2); if (iResult < 0) return iResult;

        }
        if(transmitted[2]) {
                for (UINT16 i = 0x0000; i < 8; i++) datablock3[i] = payload2++;
                iResult = mcx_Element_UpdateData(DeviceId, BusList1, Element3); if (iResult < 0) return iResult;

        }
        if(transmitted[3]) {
                for (UINT16 i = 0x0000; i < 16; i++) datablock4[i] = payload3++;
                iResult = mcx_Element_UpdateData(DeviceId, BusList1, Element4); if (iResult < 0) return iResult;
                count++;
        }
}
```

5.84 mcx_Get_Buslist_TransmittedElements

```
INT16 mcx_Get_Buslist_TransmittedElements (
                                    UINT16      deviceId
                                    UINT16      busList
                                    U8BIT*      elementsTransmitted
                                   )
```

Parameters

deviceId	Unique Device ID 0 - (sitalMaximum_DEVICES - 1)
busList	Unique ID of BusList 0 - (MAX_BUSLISTS - 1)
elementsTransmitted	A pointer to an array that returns all elements transmitted states: Not transmitted – '0' Transmitted – '1'

Description

Mode: Running

This function returns all the Buslists' Elements' states – up to maximum 64 messages allowed in a single Buslist.
Each of this array index represents it's complementary Element's states:
For index 0, the message 0 state is '0' (Not transmitted / Completed) or '1' (Transmitted).
For index 6, the message 6 state is '0' (Not transmitted / Completed) or '1' (Transmitted).
And so on…

Example

```
U8BIT transmitted[64];
U16BIT count = 0;

while(count < times){
        iResult = mcx_Get_Buslist_TransmittedElements(DeviceId, BusList1, transmitted); if (iResult < 0) return iResult;
        if(transmitted[0]) {
                for (UINT16 i = 0x0000; i < 8; i++) datablock1[i] = payload0++;
                iResult = mcx_Element_UpdateData(DeviceId, BusList1, Element1); if (iResult < 0) return iResult;

        }
        if(transmitted[1]) {
                for (UINT16 i = 0x0000; i < 16; i++) datablock2[i] = payload1++;
                iResult = mcx_Element_UpdateData(DeviceId, BusList1, Element2); if (iResult < 0) return iResult;

        }
        if(transmitted[2]) {
                for (UINT16 i = 0x0000; i < 8; i++) datablock3[i] = payload2++;
                iResult = mcx_Element_UpdateData(DeviceId, BusList1, Element3); if (iResult < 0) return iResult;

        }
        if(transmitted[3]) {
                for (UINT16 i = 0x0000; i < 16; i++) datablock4[i] = payload3++;
                iResult = mcx_Element_UpdateData(DeviceId, BusList1, Element4); if (iResult < 0) return iResult;
                count++;
        }
}
```

5.85 mcx_Element_UpdateStatuses

```
INT16 mcx_Element_UpdateStatuses (
    UINT16 deviceId
    UINT16 element
    UINT16 rxStatus
    UINT16 txStatus
)
```

Parameters

deviceId	Unique Device ID 0 - (sitalMaximum_DEVICES - 1)
element	Unique ID of Element's ID
rxStatus	Rx status to update for this Element
txStatus	Tx status to update for this Element

Description

Mode: Ready & Running
This function allows the user to update an Element's statuses during running frames.

5.86 mcx_SetRTsResponseDelay

```
INT16 mcx_SetRTsResponseDelay (
    UINT16 deviceId
    UINT16 rtResponseHalfUs
    UINT16 respondAnyway
)
```

Parameters

deviceId	Unique Device ID 0 - (sitalMaximum_DEVICES - 1)
rtResponseHalfUs	Response time value to change the global RTs response time. Values are 6 – 258 in half of micro seconds.
respondAnyway	'1' - Force RTs to reply once *rtResponseHalfUs* value is between 0 – 6 (1/2 micro seconds) '0' – Do not force RTs to reply.

Description

Mode: Ready & Running
This function sets all RTs response time to the defined value in parameter *rtResponseHalfUs.*

In order for an RT to spoof another RT, do the following:
1. Find out what is your spoofed RT response time.
2. Find out what is your spoofed RT address.
3. Enable simulation for that RT address.
4. Set the MultiRT response time to either shorter / longer response time than measured RT - *rtResponseHalfUs*
5. Add the attacked messages into the MultiRT frame.
6. Run the frame when spoofing is required.

NOTE - If the MultiRT response is shorter than the spoofed RT, some spoofed RTs would back off,
Others would transmit at their response time.
The latter case might error out the spoofed RT response.
In case that the MultiRT's response time is greater than the spoofed RT response time,
the MultiRT's response would probably overlap the spoofed RT.
NOTE II - 200 nano seconds are added to any user's requested MultiRT response time.
NOTE III - in case the MultiRT response is greater than the standard allows (14 us), unexpected behavior might occur.

5.87 mcx_TransmitSingleMessageOnce

INT16 mcx_TransmitSingleMessageOnce	**(**		
		UINT16	***deviceId***
		UINT16	***command***
		Bool	***emulateRt***
		UINT16*	***buffer***
		UINT16	***size***
		UINT16*	***BSW***
		UINT16*	***rtStatus***
	)		

Parameters

deviceId	Unique Device ID 0 - (sitalMaximum_DEVICES - 1)
command	1553 command.
emulateRt	'1' -emulate RT '0' – Do emulate RT
buffer	data buffer
size	buffer size
BSW	Bit status word result
rtStatus	RT Status result

Description

Mode: Ready & Running
This function sends one 1553 command once and returns Bit Status word and rtStatus.
It is an immediate one command that includes all functionality.

6 Service Functions

6.1 Mcx_Read

```
INT16 mcx_Read (
  UINT16   deviceId
  U16BIT   address
  WORD     bufferSize
  WORD*    buffer
)
```

Parameters

deviceId	Unique Device ID 0 - (sitalMaximum_DEVICES - 1)
address	Address of Memory to read data from.
bufferSize	Size of buffer to read.
buffer	A pointer to the buffer that returns the data read.

Description

Mode: Ready & Runtime
This service function returns (in the Data pointer) the data available according to the specified address.
It is advised to use this function for debug and print-outs purposes.

6.2 Mcx_Write

```
INT16 mcx_Write (
    UINT16  deviceId
    U16BIT  address
    WORD    bufferSize
    WORD*   buffer
)
```

Parameters

deviceId	Unique Device ID 0 - (sitalMaximum_DEVICES - 1)
address	Address to write data to.
bufferSize	Size of buffer to write.
Buffer	A pointer to the buffer to write.

Description

Mode: Ready & Runtime
This service function writes the data in buffer to the specified address.
It is advised to use this function for debug and print-outs purposes.

6.3 mcx_Transmit_1553_Message

```
INT16 mcx_Transmit_1553_Message (
    UINT16      deviceId
    UINT16      command
    UINT16*     blockStatusWord
    WORD*       buffer
    UINT16*     actualWordCount
    UINT16*     status
    UINT16*     tTag
    UINT16*     options
)
```

Parameters

deviceId	Unique Device ID 0 - (sitalMaximum_DEVICES - 1)
command	Mil-Std-1553 Command to transmit of type BC to RT or RT to BC.
blockStatusWord	Unused
Buffer	A pointer to the data words buffer to transmit.
Actual WordCount	Unused
Status	Unused
tTag	Unused
options	Unused

Description

Mode: Ready
This service function creates and transmits a single Mil-Std-1553 command on Bus A.
The command is of type BC2RT or RT2BC. Frame length is configured to 0.

Note I - A device initialization is required in order to run this function successfully.
Note II – in order to call this function in a loop, the buslist (This function uses buslist == 0 internally) must be free on each iteration (mcx_Free(..) or mcx_FreeBusList(..)).

6.4 mcx_Transmit_1553_Messages

```
INT16 mcx_Transmit_1553_Messages (
    UINT16      deviceId
    UINT16      command
    UINT16*     blockStatusWord
    WORD*       buffer
    UINT16      numberOfShots
    UINT16*     status
    UINT16*     options
)
```

Parameters

deviceId	Unique Device ID 0 - (sitalMaximum_DEVICES - 1)
command	Mil-Std-1553 Command to transmit of type BC to RT or RT to BC.
blockStatusWord	Unused
Buffer	A pointer to the data words buffer to transmit.
numberOfShots	Number of cycles to transmit the requested command. 0 means, transmit forever.
Status	Unused
options	Unused

Description

Mode: Ready

This service function creates and transmits Mil-Std-1553 command on Bus A. The command is transmitted as specified in numberOfShots parameter, while selecting 0 will tramit the command forever.
The command is of type BC2RT or RT2BC. Frame length is configured to 0.

Note I - A device initialization is required in order to run this function successfully.
Note II – In order to stop the run, use free function (mcx_Free(..)).

7 Code Samples

7.1 MIL-STD-1553

```
// Create a single Rx message and run it. Get results.

static UINT16 BusList1 = 0;
static UINT16 Element1 = 0;
static UINT16 DB1 = 0;
static UINT16 datablock32[64];
short iResult = 0;
unsigned short elementCommand =  0x20;
unsigned short numberOfIterations = 3;
unsigned short messageOptions = 0x0000;
iResult += mcx_Initialize(0, Protocol_1553_PP194);
iResult += mcx_EnableRts(0, 0x01);
iResult += mcx_Create_BusList(BusList1);
iResult += mcx_Create_BusList_Element (Element1, elementCommand, 0x80 /*Bus A*/ |
messageOptions, 0, 0, 0);
iResult += mcx_Create_Element_DataBlock (DB1, DataBlockMode_64_WORDS, datablock32,
64);
for (UINT16 i = 0x0000; i < DataBlockS ; i++)
{
       dataBlock[i] = i;
}
iResult += mcx_Map_DataBlock_To_Element (Element1, DB1);
iResult += mcx_Map_Element_To_BusList (BusList1, Element1);
iResult += mcx_Start (mrtDeviceID, BusList1, numberOfIterations);
Sleep(100);

UINT16 blockStatusWord;
WORD buffer[32];
UINT16 bufferSize = 32;
UINT16 status1;
UINT16 status2;
UINT16 tag;
iResult += mcx_Get_Element_Results(0, 0, 0, &blockStatusWord, buffer, bufferSize,
&status1, &status2, &tag);
```

7.2 H009

```
// Create a single Rx message and run it, function as Multi RT. Get results.

static UINT16 BusList1 = 0;
static UINT16 Element1 = 0;
static UINT16 DB1 = 0;
static UINT16 datablock32[64];
short iResult = 0;
unsigned short elementCommand =  0x6033;
unsigned short numberOfIterations = 1;
unsigned short messageOptions = 0x0000;
iResult += mcx_Initialize(0, Protocol_H009 | MultiRT);
iResult += mcx_EnableRts(0, 0x40);
iResult += mcx_Create_BusList(BusList1);
```

```
iResult += mcx_Create_BusList_Element (Element1, elementCommand, 0x80 /*Bus A*/ |
messageOptions, 0, 0, 0);
iResult += mcx_Create_Element_DataBlock (DB1, DataBlockMode_64_WORDS, datablock32,
64);
for (UINT16 i = 0x0000; i < DataBlockS ; i++)
{
       dataBlock[i] = i;
}
iResult += mcx_Map_DataBlock_To_Element (Element1, DB1);
iResult += mcx_Map_Element_To_BusList (BusList1, Element1);
iResult += mcx_Start (mrtDeviceID, BusList1, numberOfIterations);
Sleep(100);

UINT16 blockStatusWord;
WORD buffer[32];
UINT16 bufferSize = 32;
UINT16 status1;
UINT16 status2;
UINT16 tag;
iResult += mcx_Get_Element_Results(0, 0, 0, &blockStatusWord, buffer, bufferSize,
&status1, &status2, &tag);
```

7.3 PP194 (WB194)

```
static UINT16 BusList1 = 0;
static UINT16 Element1 = 0;
static UINT16 DB1 = 0;
static UINT16 datablock32[64];
short iResult = 0;
unsigned short elementCommand =  0x0c43;
unsigned short numberOfIterations = 1;
unsigned short messageOptions = 0x0004;
UINT16 simulatedStatus = 0x1234;
unsigned short wDataWord0 = 0x2345;
unsigned short wDataWord1 = 0x00ff;

iResult += mcx_Initialize(0, Protocol_1553_PP194);
// Enabling all RIUs
//iResult += mcx_EnableRius(0, 0xffff);
iResult += mcx_Create_BusList(BusList1);
iResult += mcx_Create_BusList_Element (Element1, elementCommand, 0x80 /*Bus A*/ |
messageOptions, 0 , simStatus, 0);
iResult += mcx_Create_Element_DataBlock (DB1, DataBlockMode_64_WORDS, dataBlock,
DataBlockS);
dataBlock[0] = data0;
dataBlock[1] = data1;
iResult += mcx_Map_DataBlock_To_Element (Element1, DB1);
iResult += mcx_Map_Element_To_BusList (BusList1, Element1);
iResult += mcx_Start (mrtDeviceID, BusList1, numberOfIterations);
Sleep(100);

UINT16 blockStatusWord;
WORD buffer[32];
UINT16 bufferSize = 32;
UINT16 status1;
UINT16 status2;
UINT16 tag;
```

```
iResult += mcx_Get_Element_Results(0, 0, 0, &blockStatusWord, buffer, bufferSize,
&status1, &status2, &tag);
```

7.4 RS485

Conceptual Workflow

- Init MCX device
- Setup a module and RS485 line
- Put (Tx) data
- Verify the number of words received
- Get (Rx) into buffer by the number of words received

```
/*****************************************************************************
The following test assumes
- the tested device is a PCI (not PMC) tester device -> the RS485 devices are devices 4-7
- coupling between devices 4-5 and 6-7
NOTE - 4 == device 4 channel 0, 5 == device 4  channel 1, device 6 == device 5 channel 0, device 7 ==
device 5 channel 1
- input required from user -> device | baud rate | number of iterations
*****************************************************************************/
#include "stdafx.h"
#include <stdio.h>
#include <stdlib.h>
#include <winsock2.h>
#include "windows.h"
#include <stdio.h>
#include "CommonTypes.h"
#include "McxAPI.h"
#include "McxAPIReturnCodes.h"

UINT16 DeviceId = 0;
char errorCode[1000];
UINT16 offset = 0;
const UINT16 size = 16;
UINT16 txBuff[1024];
UINT16 rxBuff[1024];
UINT16 rxLength;

char input [256];

UINT16 module;
UINT16 line;

int _tmain(int argc, _TCHAR* argv[])
{
        INT16 iResult = 0;
        UINT16 map = 0;
        iResult = mcx_MapDevices(&map);
        module = 4;
        line = 0;

        printf("Up and running");
        if(iResult < 0)
        {
                printf("\nError reading card %04X, HIT enter to exit", iResult);
                getchar();
                return -5;
        }

        printf("\nDetected %i devices", map);

        for(int i = 0 ; i < map ; i++)
        {       iResult = mcx_Initialize(i, MIL_STD_1553);
                if (iResult < 0) {
                        mcx_GetReturnCodeDescription(iResult, errorCode);
```

```
                        printf("\nError INIT -> %s\nHit Enter to exit", errorCode);
                        getchar();
                        return -1;
                }
        }
        printf("\nDevices initialized successfully");
        printf("\n\n\n");
        UINT16 dev;
        UINT16 linerTx;
        UINT16 linerRx;
        UINT16 baud;
        UINT16 iterations;
        char *p;
        while(true)
        {
                printf("\nPress ENTER after each enty...:");
                printf("\nDevice ID:\n");
                gets (input);
                dev = atoi(input);
                printf("\nBaud Rate:\n");
                gets (input);
                baud = atoi(input);
                printf("\nIterations:\n");
                gets (input);
                iterations = atoi(input);

                dev = (dev / 2) * 2;
                if((dev % 2) == 0) {
                        linerTx = 0;
                        linerRx = 1;
                }
                else  {
                        linerTx = 1;
                        linerRx = 0;
                }
                iResult = mcx_RS485_Setup(dev, linerTx, 8 /*bit count*/, 1 /*no parity*/, 0 /*stop
bits single*/, baud /*rate divider...*/, 1/*rxtx mode*/, &offset);
                if (iResult < 0) {
                        mcx_GetReturnCodeDescription(iResult, errorCode);
                        printf("Error -> %s\n", errorCode);
                        getchar();
                        return -1;
                }
                iResult = mcx_RS485_Setup(dev, linerRx, 8 /*bit count*/, 1 /*no parity*/, 0 /*stop
bits single*/, baud /*rate divider...*/, 1/*rxtx mode*/, &offset);
                if (iResult < 0) {
                        mcx_GetReturnCodeDescription(iResult, errorCode);
                        printf("Error -> %s\n", errorCode);
                        getchar();
                        return -1;
                }

                printf("\nTX device %i", dev);
                if((dev % 2) == 0) printf("\nRX device %i", dev + 1);
                else printf("\nRX device %i", dev - 1);

                for(int i = 0 ; i < size ; i++) txBuff[i] = i + 0xAAAA;
                for(int i = 0 ; i < iterations; i++){
                // Tx a buffer to the bus..
                        iResult = mcx_RS485_Put(dev, linerTx, size, txBuff);
                        if (iResult < 0) {
                                mcx_GetReturnCodeDescription(iResult, errorCode);
                                printf("Error -> %s\n", errorCode);
                                getchar();
                                return -1;
                        }
                        Sleep(100);

                        iResult = mcx_RS485_GetNumberOfReceivedWords(dev, linerRx, offset, &rxLength);
                        if (iResult < 0) {
```

```
                        mcx_GetReturnCodeDescription(iResult, errorCode);
                        printf("Error -> %s\n", errorCode);
                        getchar();
                        return -1;
                }
                else{
                        printf("\nRx %i words", rxLength);
                }

                if(rxLength > 0)
                {
                        iResult = mcx_RS485_Get(dev, linerRx, &offset, rxLength, rxBuff);
                        if (iResult < 0) {
                                mcx_GetReturnCodeDescription(iResult, errorCode);
                                printf("Error -> %s\n", errorCode);
                                getchar();
                                return -1;
                        }
                        else{
                                printf("\n");
                                for(int i = 0 ; i < rxLength ; i++)
                                {
                                        if(txBuff[i] != rxBuff[i])
                                        {
                                                printf("DATA err %i ", i);
                                        }
                                }
                        }
                }
        }
    }
    return 0;
}
```

7.5 Arinc 429

Conceptual Workflow

- Get number of existing channels on the card
- Open all channels
- Set each channel's config
- Send data on Tx bus
- Check the number of words pending in the Rx FIFO
- Get (Rx) into buffer by the number of words received

```
#include "stdafx.h"
#include <stdio.h>
#include <stdlib.h>
#include "windows.h"
#include "McxAPI.h"
#include "McxAPI.h"
#include "McxAPIReturnCodes.h"

static mcx_A429ChannelInfo cInfo[4] = {};


void _tmain(int argc, _TCHAR* argv[])
{
        INT16 result;
        UINT32 nc;

        result = mcx_A429_Channel_GetCount(&nc);
        if(result != 0) printf("\nmcx_A429_Channel_GetCount Failed");

        //result = mcx_A429_Channel_Reset(0, 0);
```

```
	for(int i = 0 ; i < (int)nc ; i++){
		result = mcx_A429_Channel_GetInformation(i, &cInfo[i]);
		if(result != 0) printf("\nmcx_A429_Channel_GetInformation Failed");
		result = mcx_A429_Channel_Open(i, &cInfo[i]);
		if(result != 0) printf("\nmcx_A429_Channel_Open Failed");
	}
	UINT32 cf = (MCX_A429_CFG_HIGH_RATE | MCX_A429_CFG_PARITY_NONE);
	//cf = 0x98761234;
	UINT32 gcf;
	result = mcx_A429_Channel_SetConfigRegister(0, cf);
	if(result != 0) printf("\nmcx_A429_Channel_SetConfigRegister ch0 Failed");
	result = mcx_A429_Channel_GetConfigRegister(0, &gcf);
	if(result != 0) printf("\nmcx_A429_Channel_GetConfigRegister ch0 Failed");
	if(cf != (gcf & 0xFF)) printf("\nFailed to configure ch0");

	result = mcx_A429_Channel_SetConfigRegister(1, cf);
	if(result != 0) printf("\nmcx_A429_Channel_SetConfigRegister ch1 Failed");
	result = mcx_A429_Channel_SetConfigRegister(2, cf);
	if(result != 0) printf("\nmcx_A429_Channel_SetConfigRegister ch2 Failed");
	result = mcx_A429_Channel_SetConfigRegister(3, cf);
	if(result != 0) printf("\nmcx_A429_Channel_SetConfigRegister ch3 Failed");

	UINT32 buff[100];
	UINT32 buff1[100];
	UINT32 buff2[100];
	UINT32 buff3[100];
	UINT32 written, rcv;
	for(UINT32 i = 0; i < 100 ; i++) {
		buff[i] = (0x00000000 | i);
		buff2[i] = i + 50;
	}
	//while(1){
		result = mcx_A429_Send(0, 100, buff, &written);
	//}
	//if(result != 0) printf("\nmcx_A429_Send ch0 Failed");
	result = mcx_A429_Channel_GetConfigRegister(0, &gcf);
	if(result != 0) printf("\nmcx_A429_Channel_GetConfigRegister ch0 Failed");
	Sleep(40);
	result = mcx_A429_Receive(2, 100, buff1, &rcv);
	if(result != 0) printf("\nmcx_A429_Receive ch1 Failed");

	for(int i = 0 ; i < 100 ; i++){
		if(buff[i] != buff1[i]) printf("\nBuffers not the same!");
	}

	written = 0; rcv = 0;

	result = mcx_A429_Send(1, 100, buff2, &written);
	if(result != 0) printf("\nmcx_A429_Send ch1 Failed");
	result = mcx_A429_Channel_GetConfigRegister(1, &gcf);
	if(result != 0) printf("\nmcx_A429_Channel_GetConfigRegister ch1 Failed");
	Sleep(40);
	result = mcx_A429_Receive(3, 100, buff3, &rcv);
	if(result != 0) printf("\nmcx_A429_Receive ch1 Failed");

	for(int i = 0 ; i < 100 ; i++){
		if(buff2[i] != buff3[i]) printf("\nbuff2 != buff3!");
	}

	getchar();
	return;
}
```

7.6 MIL-STD-1760

```
#include "stdafx.h"
#include <stdio.h>
#include <time.h>
#include <stdlib.h>
#include "windows.h"
#include "McxAPI.h"
#include "McxAPIReturnCodes.h"

INT16 iResult = 0;
UINT16 DeviceId = 0;
char errorCode[1000];
/*
1. Power up the MIL-STD-1760 UUT
2. Run full speed contimuous RT1 with Tx 30 words to SA1
Repeat -
3. Record first time of reply
4. Record first time of reply not busy
until
5. Report UUT powerup response time
6. Report UUT not-busy response time
7. Report message transmission resolution
8. Exit
*/

UINT16 blockStatus = 0;
UINT16 buffer[32];
UINT16 aWC = 0;
UINT16 status = 0;
UINT16 tTag = 0;
UINT16 options = 0;
UINT16 numberOfShots = 0;// 0 == run forever
clock_t t1, t2, t3;


int _tmain(int argc, _TCHAR* argv[])
{
        iResult = mcx_Initialize(DeviceId, MIL_STD_1553);
        if (iResult < 0) {
                mcx_GetReturnCodeDescription(iResult, errorCode);
                printf("Error -> %s\n", errorCode);
                printf("Hit Enter to Exit");
                getchar();
                return -1;
        }

        // POWER UP the UUT
        printf("\nPower Up the UUT and hit Enter to Continue\n");
        getchar();
        printf("Waiting for Response..");
        t1 = clock();

        iResult = mcx_FreeBusList(DeviceId, 0);
                if (iResult < 0) {
                        mcx_GetReturnCodeDescription(iResult, errorCode);
                        printf("Error -> %s\n", errorCode);
                        printf("Hit Enter to Exit");
                        getchar();
                        return -1;
                }
        iResult = mcx_Transmit_1553_Messages(DeviceId, 0x0C02, &blockStatus, buffer, numberOfShots,
&status, &options);
                if (iResult < 0) {
                        mcx_GetReturnCodeDescription(iResult, errorCode);
                        printf("Error -> %s\n", errorCode);
                        printf("Hit Enter to Exit");
                        getchar();
                        return -1;
                }
```

```
        UINT32 BSW = 0;
        INT16 msgType;
        unsigned long long ttag = 0;
        WORD data[32];
        UINT32 swPointer = 0;
        WORD rxCommand = 0xAAAA;
        WORD txCommand = 0xAAAA;
        WORD rxStat;
        WORD txStat;
        WORD bufferSize = 0;
        bool answered = false;

        while(1){

                iResult = mcx_wm_GetNextMsg_1553_194(0, &msgType, &swPointer, &rxCommand, &txCommand,
data, &bufferSize, &rxStat, &txStat, &BSW, &ttag);
                if (iResult < 0) {
                        mcx_GetReturnCodeDescription(iResult, errorCode);
                        printf("Error -> %s\n", errorCode);
                        printf("Hit Enter to Exit");
                        getchar();
                        return -1;
                }

                if((BSW & mcx_wm_NO_RESPONSE) == 0 && answered == false && (rxCommand != 0xCCCC ||
txStat != 0xCCCC)) {
                        t2 = clock();
                        answered = true;
                }
                // bit 3 is busy
                if((answered == true) && ((txStat & 0x8) == 0)){
                        t3 = clock();
                        break;
                }
        }

        float diff = ((float)(t2 - t1) / 1000000.0F ) * 1000;
    printf("\nTime from power up until first respose - %f",diff);

        diff = ((float)(t3 - t2) / 1000000.0F ) * 1000;
    printf("\nTime from first respose until not busy - %f",diff);

        printf("\n\nProgram finished, please press Enter");
        getchar();
        mcx_Free(DeviceId);

        return 0;
}
```

8 Appendices

8.1 Appendix A – Returned Error Codes

Note I – returned error codes can be found in McxAPIReturnCodes.h

Note II – The function '`mcx_GetReturnCodeDescription(..)`' can be used in order to retrieve a string description.

Codes:

```
#define STL_ERR_SUCCESS                                                 ( 0)
#define STL_ERR_INVALID_DEVICE_NUMBER                                   (-1)

#define STL_ERR_VERSION_ERROR                                           (-2)

#define STL_ERR_DEVICE_NOT_FOUND                                        (-3)

#define STL_ERR_FAILED_TO_OPEN_DEVICE                                   (-4)

#define STL_ERR_FAILED_TO_GET_DESCRIPTORS                               (-5)

#define STL_ERR_FAILED_TO_CONFIGURE_FPGA                                (-6)

#define STL_ERR_FAILED_TO_READ_DEVICE_PORT                              (-7)

#define STL_ERR_FAILED_TO_SET_DEVICE_PORT                               (-8)

#define STL_ERR_DEVICE_POWER_SUPPLY_ERRPR                               (-9)

#define STL_ERR_FAILED_TO_START_FPGA_CONFIG                             (-10)

#define STL_ERR_FAILED_TO_OPEN_FPGA_FILE                                (-11)

#define STL_ERR_FAILED_TO_WRITE_FPGA_CHUNK                              (-12)

#define STL_ERR_FAILED_TO_LOAD_FPGA                                     (-13)

#define STL_ERR_NOT_IMPLEMENTED                                         (-14)

#define STL_ERR_DATABLOCK_SIZE_EXCEEDS_LIMIT                            (-15)

#define STL_ERR_QUSB_WRITE_COMMAND_FAILED                               (-16)

#define STL_ERR_QUSB_READ_FAILED                                        (-17)

#define STL_ERR_QUSB_WRITE_FAILED                                       (-18)

#define STL_ERR_TOO_MANY_ELEMENTS_FOR_MEM_SPCAE                         (-19)

#define STL_ERR_NULL_POINTER_PARAMETER                                  (-20)

#define STL_ERR_STRING_TOO_LONG                                         (-21)

#define STL_ERR_INVALID_DIRECTORY_NAME                                  (-22)

#define STL_ERR_FAILED_TO_ALLOCATE_MEMORY                               (-23)

#define STL_ERR_H009_DEVICE_ID_MUST_BE_EVEN                             (-24)
#define STL_ERR_PP194_ELEMENT_ON_ODD_DEVICE_ID                          (-25)
#define STL_ERR_DEVICE_NOT_MAPPED                                       (-26)
#define STL_ERR_LONG_READ_5_LSB_NOT_0                                   (-27)
#define STL_ERR_TOO_MANY_WRONG_SYMBOLS_IN_WM                            (-28)
#define STL_ERR_DEVICE_NOT_INITIALIZED                                  (-29)
#define STL_ERR_PP194_DEVICE_ID_MUST_BE_EVEN                            (-30)

#define STL_ERR_DEVICE_BUSY
                (-51)
#define STL_ERR_DEVICE_WAS_FORCED_HW_RESET_DURING_STOP                  (-52)

#define STL_ERR_PCI_READ_WIDTH_NOT_MODULU_4                             (-100)


#define STL_ERR_BUSLIST_ALREADY_EXISTS                                  (-2000)
                        ///User tried to create a buslist that was previously created.
#define STL_ERR_ELEMENT_ALREADY_EXISTS                                  (-2001)
                        ///
#define STL_ERR_DATABLOCK_ALREADY_EXISTS                                (-2002)
                    ///
#define STL_ERR_DATABLOCK_SIZE_ASSINGMENT_ERROR                         (-2003)

#define STL_ERR_DATABLOCK_SIZE_TOO_SMALL                                (-2004)

#define STL_ERR_MAPPING_UNREADY_CONSTRUCTS                              (-2005)
```

```
#define STL_ERR_BUSLIST_CONTAINS_TOO_MANY_ELEMENTS                    (-2006)

#define STL_ERR_BUSLIST_IS_RUNNING                                    (-2007)

#define STL_ERR_CODE_IN_REG_1A_INCORRECT                              (-2008)

#define STL_ERR_WRITING_TO_UNINITIALIZED_DATABLOCK                    (-2009)

#define STL_ERR_WRITING_TO_UNINITIALIZED_ELEMENT                      (-2010)

#define STL_ERR_WRITING_TO_UNMAPPED_DATABLOCK                         (-2011)

#define STL_ERR_ACCESSING_INVALID_DEVICE_ID                           (-2012)

#define STL_ERR_REQUESTED_ID_EXCEEDED_MAX_ALLOWED                     (-2013)
#define STL_ERR_ELEMENT_IS_RUNNING                                    (-2014)
#define STL_ERR_DATABLOCK_IS_RUNNING                                  (-2015)
#define STL_ERR_REQUESTED_ID_UNINITIALIZED                            (-2016)
#define STL_ERR_DATABLOCK_NOT_MAPPED_TO_ELEMENT                       (-2017)
#define STL_ERR_ELEMENT_NOT_MAPPED_TO_BUSLIST                         (-2018)
#define STL_ERR_READ_DATA_WHILE_TX_NOT_ALLOWED                        (-2019)
#define STL_ERR_COMMAND_MODE_CODE_NOT_SUPPORTED                       (-2020)
#define STL_ERR_READING_EMPTY_BUFFER                                  (-2021)
#define STL_ERR_NOT_IN_RUNNING_MODE                                   (-2022)
#define STL_ERR_STOP_RUN_FAILED                                       (-2023)
#define STL_ERR_START_RUN_FAILED                                      (-2024)
#define STL_ERR_GET_TIMETAG_FAILED_DATA_INCONSISTENT                  (-2025)
#define STL_ERR_TIME_REQUESTED_NOT_IN_VALID_RANGE                     (-2026)
#define STL_ERR_TIME_VALUE_REQUESTED_IS_INVALID                       (-2027)
#define STL_ERR_MESSGE_NUMBER_TO_INSERT_ERROR_IS_INVALID              (-2028)
#define STL_ERR_WORD_NUMBER_TO_INSERT_ERROR_IS_INVALID                (-2029)
#define STL_ERR_NO_ERROR_SPECIFIED                                    (-2030)
#define STL_ERR_SYNC_INJECTION_PARAM_INVALID                          (-2031)
#define STL_ERR_SPECIFIED_ERROR_INJECTION_NOT_SUPPORTED               (-2032)
#define STL_ERR_ZERO_CROSSING_INJECTION_PARAM_INVALID                 (-2033)
#define STL_ERR_DEVICE_MEMORY_FULL                                    (-2034)
/// During mRt start the datablock's allocation spiled over the memory limit of the device
#define STL_ERR_RUI_0_ENABLE_IS_REDUNDANT_IN_PP194
                                                                      (-2035)
#define STL_ERR_ELEMENT_NOT_INITIALIZED                               (-2036)
#define STL_ERR_DATABLOCK_SIZE_TOO_BIG                                (-2037)
#define STL_ERR_BUSLIST_IS_NOT_IN_RUNNING_MODE                        (-2038)
#define STL_ERR_SPECIFIED_PROTOCOL_NOT_SUPPORTED                      (-2039)
#define STL_ERR_INVALID_PROTOCOL_TYPE                                 (-2040)
#define STL_ERR_MULTI_SHOTS_NOT_SUPPORTED                             (-2041)
#define STL_ERR_BUSLIST_PASSIVE_PHASE_NOT_FOUND                       (-2042)
#define STL_ERR_DATA_VECTOR_OVERFLOW                                  (-2043)
#define STL_ERR_DEPRECATED_FUNCTION                                   (-2044)

#define STL_ERR_INVALID_ARGUMENTS                                     (-2200)
#define STL_ERR_NULL_POINTER                                          (-2201)
#define STL_ERR_READ_FAILED
        (-2202)
#define STL_ERR_WRITE_FAILED                                          (-2203)
//#define STL_ERR_DEVICE_BUSY
        (-2204)
#define STL_ERR_TIMETAG_ZERO_READ_AGAIN                               (-2205)

#define STL_ERR_CANNOT_SEND_ON_RX_CHANNEL                             (-
2300)// Arinc 429
#define STL_ERR_CANNOT_GET_ON_TX_CHANNEL                              (-
2301)// Arinc 429
#define STL_ERR_TIMEOUT
                (-2302)// Arinc 429
#define STL_ERR_IO_OVERFLOW
        (-2303)// Arinc 429
#define STL_ERR_A429_SIGNATURE_MISSING                                (-2304)// Arinc 429
#define STL_ERR_A429_DEVICE_ALREADY_OPENED                            (-2305)// Arinc 429
// licensing
#define STL_ERR_LICENSE_PARAM_NOT_FOUND                               (-2350)
#define STL_ERR_LICENSE_STRING_NOT_FOUND                              (-2351)
#define STL_ERR_LICENSE_STRING_TOO_SHORT                              (-2352)
#define STL_ERR_LICENSE_INVALID_FEATURE                               (-2353)
#define STL_ERR_LICENSE_INVALID_CHECKSUM                              (-2354)
#define STL_ERR_LICENSE_INVALID_KEY                                   (-2355)
#define STL_ERR_LICENSE_PROTOCOL_DISABLED                             (-2356)
#define STL_ERR_LICENSE_SINGLE_DEVICE_PERMISSION                      (-2357)
#define STL_ERR_LICENSE_DIGIBUS_REQUIRE_1553                          (-2358)
#define STL_ERR_LICENSE_BUSLIST_CONTAINS_UNLICENSED_PP194_MESSAGE     (-2359)
#define STL_ERR_LICENSE_BUSLIST_CONTAINS_UNLICENSED_1553_MESSAGE      (-2360)
#define STL_ERR_LICENSE_H009_UNLICENSED                               (-2361)
#define STL_ERR_LICENSE_1553_UNLICENSED                               (-2362)
#define STL_ERR_LICENSE_PP194_UNLICENSED                              (-2363)
#define STL_ERR_LICENSE_EBR_UNLICENSED                                (-2364)
#define STL_ERR_LICENSE_DIGIBUS_F16_UNLICENSED                        (-2365)
#define STL_ERR_LICENSE_ENGINEERING_UNITS_UNLICENSED                  (-2366)
#define STL_ERR_LICENSE_WIRING_FAULT_LOCATION_UNLICENSED              (-2367)
#define STL_ERR_LICENSE_SMART_CYBER_EMULATION_UNLICENSED              (-2368)
#define STL_ERR_LICENSE_NO_LICENSED_FEATURES_FOUND                    (-2369)
#define STL_ERR_LICENSE_REQUIRED_AND_NOT_FOUND_1553_ONLY_ENABLED      (-2370)

#define STL_ERR_FUNCTION_NOT_IMPLEMENTED                              (-3000)

#define STL_ERR_SW_POINTER_INPUT_IS_ODD                               (-3001)
#define STL_ERR_ODD_NUMBER_OFF_ELEMENTS_IN_HOST_BUFFER                (-3002)
```

```
#define STL_ERR_LICENSE_FILE_MISSING_OR_DAMAGED (-3003)
#define STL_ERR_LICENSE_FILE_EMPTY (-3004)

#define STL_ERR_ETH_W_SOCKET_FAIL (-7000)
#define STL_ERR_ETH_R_SOCKET_FAIL (-7001)
#define STL_ERR_ETH_W_SOCKET_ADDRESS_ERROR (-7002)
#define STL_ERR_ETH_R_SOCKET_ADDRESS_ERROR (-7003)
#define STL_ERR_ETH_W_DATA_BUFF_SIZE (-7004)
#define STL_ERR_ETH_W_REQUEST_TIMEOUT (-7005)
#define STL_ERR_ETH_R_SENDTO_REQUEST_TIMEOUT (-7006)
#define STL_ERR_ETH_R_RECVFROM_REQUEST_TIMEOUT (-7007)
#define STL_ERR_ETH_SERVER_LIST_ITEMS (-7008)
```

8.2 Appendix B – mcx_A429ChannelInfo

```
//Arinc429
typedef struct mcxA429ChannelInformation
{
    /// The size in bytes of this structure.
    /// Caller must set this field to sizeof(stla429ChannelInformationStructure).
    /// This is to prevent buffer overwrite when compilers are incompatible or
definitions change.
    UINT32  dwStructureSize;
    UINT32 dwUserTag;        // Arbitrary user provided value
    union {
        struct /*characteristics*/ {
           UINT32 channelIsAvailable : 1;    // true = available for use
           UINT32 channelRunning : 1;        // true = running, false = not running
           UINT32 channelFailure : 1;        // true = failure detected
           UINT32 channelIsTX : 1;           // true = configured as TX, false = as RX
           UINT32 channelIsHighSpeed : 1;    // true = configured for high speed, else
low speed
           UINT32 channelSupportsTX : 1;     // true = can be configured as TX
           UINT32 channelSupportsRX : 1;     // true = can be configured as RX
           UINT32 channelSupportsHighSpeed : 1;  // true = can be configured for high
speed
           UINT32 channelSupportsLowSpeed : 1;   // true = can be configured for low
speed
        };
        UINT32  dwFlags; // above struct as integer
    };
    UINT32  dwTransferSize;   // Size of transfer buffer required for bulk RX
    UINT32  dwCardNumber;     // Card number on which channel is located
    UINT32  dwReserved1;      // Padding
} mcx_A429ChannelInfo;
```

8.3 Appendix C – External Loopback Device to Device

Code implementation:

```
// transmit the command of type RT2BC on bus A from BC device to MultiRT device and
then on bus B
// RT is simulated in MultiRT side, data is incremental
// command is transmitted once
// data is checked in the BC side
// then devices are switched, repeating the test
// this test is a blocking command
// 4 results are returned - device0A, device0B, device1A, device1B
// NOTE - assuming devices are initialized

INT16 mcx_TestExternalLoopback_DevicetoDevice(UINT16 device0, UINT16 device1, UINT16* resultD0A,
UINT16* resultD0B, UINT16* resultD1A, UINT16* resultD1B, bool* badDataFound){

        if (((INT16)0 > device0) || ((INT16)sitalMaximum_DEVICES <= device0)) return
STL_ERR_INVALID_DEVICE_NUMBER;
        if (((INT16)0 > device1) || ((INT16)sitalMaximum_DEVICES <= device1)) return
STL_ERR_INVALID_DEVICE_NUMBER;


        INT16 iResult = 0;
        UINT16 lBus       = 0x80;
        unsigned short rxStt0 = 0;
        unsigned short txStt0 = 0;
        unsigned short rxStt1 = 0;
        unsigned short txStt1 = 0;
        UINT16 BusList0 = 0;
        UINT16 Element0 = 0;
        UINT16 DB0 = 0;
        UINT16 BusList1 = 1;
        UINT16 Element1 = 1;
        UINT16 DB1 = 1;
        UINT16 datablock32_0[64];
        UINT16 datablock32_1[64];
        UINT16 command0 = 0xC20;// RT1 to BC, 32 words
        UINT16 command1 = 0xC20;// RT1 to BC, 32 words
        UINT16 localD0 = 0;
        UINT16 localD1 = 1;

        // results
        INT16 results = 0;
        UINT16 blockStatus = 0;
        UINT16 buffer[32];
        UINT16 status1 = 0;
        UINT16 status2 = 0;
        UINT16 tTag = 0;

        for(int i = 0 ; i < 2 ; i++)/* iteration for device 0 to 1 and then device 1 to 0*/{

                iResult = mcx_Stop2(localD1); if (iResult < 0) return iResult;
                iResult = mcx_Stop2(localD0); if (iResult < 0) return iResult;

                if((i % 2) == 0)
                {
                        localD0 = device0;
                        localD1 = device1;
                }
                else
                {
                        localD0 = device1;
                        localD1 = device0;
                }
                for(int j = 0 ; j < 2 ; j++)/*bus selection..*/{
                        if((j % 2) == 0) lBus = 0x80;
```

```
                        else lBus = 0x00;

                        iResult = mcx_FreeBusList(localD0, BusList0);
                        iResult = mcx_FreeBusList(localD1, BusList1);

                        UINT16 userPort = MIL_STD_1553_AND_PP194 | MultiRT;
                        iResult = mcx_SetUserPort(localD1, userPort); if(iResult < 0) return iResult;
                        userPort = MIL_STD_1553_AND_PP194;
                        iResult = mcx_SetUserPort(localD0, userPort); if(iResult < 0) return iResult;

                        // MultiRT create and go..
                        iResult = mcx_EnableRts(localD1, 0xFFFFFFFF);  if(iResult < 0) return
iResult;// Enable all RTs, incremental data is injected
                        iResult = mcx_Create_BusList(localD1, BusList1); if(iResult < 0) return
iResult;
                        iResult = mcx_Create_BusList_Element (localD1, Element1, command1, lBus,
0x0000, rxStt1, txStt1); if(iResult < 0) return iResult;
                        iResult = mcx_Create_Element_DataBlock (localD1, DB1, 0, datablock32_1, 64);
if(iResult < 0) return iResult;
                        iResult = mcx_Map_DataBlock_To_Element (localD1, Element1, DB1); if(iResult <
0) return iResult;
                        iResult = mcx_Map_Element_To_BusList (localD1, BusList1, Element1); if(iResult
< 0) return iResult;

                        for(int idx = 0 ; idx < 32 ; idx++) datablock32_1[idx] = 0x5555 + i;

                        iResult = mcx_Start(localD1, BusList1, 0); if (iResult < 0) return iResult;
                        //Sleep(1);

                        // BC side..
                        iResult = mcx_EnableRts(localD0, 0);  if(iResult < 0) return iResult;
                        iResult = mcx_Create_BusList(localD0, BusList0); if(iResult < 0) return
iResult;
                        iResult = mcx_Create_BusList_Element (localD0, Element0, command0, lBus,
0x0000, rxStt0, txStt0); if(iResult < 0) return iResult;
                        iResult = mcx_Create_Element_DataBlock (localD0, DB0, 0, datablock32_0, 64);
if(iResult < 0) return iResult;
                        iResult = mcx_Map_DataBlock_To_Element (localD0, Element0, DB0); if(iResult <
0) return iResult;
                        iResult = mcx_Map_Element_To_BusList (localD0, BusList0, Element0); if(iResult
< 0) return iResult;

                        iResult = mcx_Start(localD0, BusList0, 1); if (iResult < 0) return iResult;

                        // let the frame end
                        Sleep(1);

                        blockStatus = 0;
                        // get the results..
                        results = mcx_Get_Element_Results(localD0, BusList0, 0, &blockStatus, buffer,
32, &status1, &status2, &tTag); if (results < 0) return results;
                        if(i == 0 && j == 0) *resultD0A = blockStatus;
                        if(i == 0 && j == 1) *resultD0B = blockStatus;
                        if(i == 1 && j == 0) *resultD1A = blockStatus;
                        if(i == 1 && j == 1) *resultD1B = blockStatus;

                        bool badData = false;
                        // check data
                        for(int idx2 = 0 ; idx2 < 32 ; idx2++)
                        {
                                if(datablock32_1[idx2] != 0x5555 + i)
                                {
                                        badData = true;
                                        break;
                                }
                        }
                        *badDataFound = badData;
                        iResult = mcx_Stop2(localD1); if (iResult < 0) return iResult;

                }
```

```
        }
        return iResult;
}
```

17 Atir Yeda St., Kfar-Saba, ISRAEL 44643

Email: info@sitaltech.com

Website: http://www.sitaltech.com