AceXtreme®
C Software Development Kit

DDC®
Connectivity Power Control

Model: BU-69092SX

GNAL enc_shift_data : std_log
PROCESS (clk, hsp_rst_l)
BEGIN
if (hsp_rst_l = '0') then
enc_shift_data <= (others => '
elsif ((clk'event) and (clk = '1'))
if (erst_l = '0') then

Software User's Manual

The AceXtreme® C Software Development Kit (SDK) provides the framework for efficient development of applications with DDC's series of MIL-STD-1553 components and cards.

For more information: www.ddc-web.com/BU-69092SX

DDC's Data Networking Solutions

MIL-STD-1553 | ARINC 429 | Fibre Channel

As the leading global supplier of data bus components, cards, and software solutions for the military, commercial, and aerospace markets, DDC's data bus networking solutions encompass the full range of data interface protocols from MIL-STD-1553 and ARINC 429 to USB, and Fibre Channel, for applications utilizing a spectrum of form-factors including PMC, PCI, Compact PCI, PC/104, ISA, and VME/VXI.

DDC has developed its line of high-speed Fibre Channel and Extended 1553 products to support the real-time processing of field-critical data networking netween sensors, compute notes, data storage displays, and weapons for air, sea, and ground military vehicles.

Whether employed in increased bandwidth, high-speed serial communications, or traditional avionics and ground support applications, DDC's data solutions fufill the expanse of military requirements including reliability, determinism, low CPU utilization, real-time performance, and ruggedness within harsh environments. Out use of in-house intellectual property ensures superior mutli-generational support, independent of the life cycles of commercial devices. Moreover, we maintain software compatibility between product generations to protect our customers' investments in software development, system testing, and end-product qualification.

MIL-STD-1553

DDC provides an assortment of quality MIL-STD-1553 commercial, military, and COTS grade cards and components to meet your data conversion and data interface needs. DDC supplies MIL-STD-1553 board level products in a variety of form factors including AMC, USB, PCI, cPCI, PCI-104, PCMCIA, PMC, PC/104, PC/104-Plus, VME/VXI, and ISAbus cards. Our 1553 data bus board solutions are integral elements of military, aerospace, and industrial applications. Our extensive line of military and space grade components provide MIL-STD-1553 interface solutions for microprocessors, PCI buses, and simple systems. Our 1553 data bus solutions are designed into a global network of aircraft, helicopter, and missle programs.

ARINC 429

DDC also has a wide assortment of quality ARINC-429 commercial, military, and COTS grade cards and components, which will meet your data conversion and data interface needs. DDC supplies ARINC-429 board level products in a variety of form factors including AMC, USB, PCI, PMC, PCI-104, PC/104 Plus, and PCMCIA boards. DDC's ARINC 429 components ensure the accurate and reliable transfer of flight-critical data. Our 429 interfaces support data bus development, validation, and the transfer of flight-critical data aboard commercial aerospace platforms.

Fibre Channel

DDC has developed its line of high-speed Fibre Channel network access controllers and switches to support the real-time processing demands of field-critical data networking between sensors, computer nodes, data storage, displays, and weapons, for air, sea, and ground military vehicles. Fibre Channel's architecture is optimized to meet the performance,reliability, and demanding environmental requirements of embedded, real time, military applications, and designed to endure the multi-decade life cycle demands of military/aerospace programs.

BU-69092SX ACEXTREME® C SDK
SOFTWARE USER'S MANUAL

MN-69092SX-002

The information provided in this Software User's Manual is believed to be accurate; however, no responsibility is assumed by Data Device Corporation for its use, and no license or rights are granted by implication or otherwise connection therewith.

Specifications are subject to change without notice.
Please visit our Web site at http://www.ddc-web.com/ for the latest information.

105 Wilbur Place
Bohemia, New York 11716-2426
Tel: (631) 567-5600, Fax: (631) 567-7358
World Wide Web - http://www.ddc-web.com

For Technical Support - 1-800-DDC-5757 ext. 7771
United Kingdom - Tel: +44-(0)1635-811140, Fax: +44-(0)1635-32264
France - Tel: +33-(0)1-41-16-3424, Fax: +33-(0)1-41-16-3425
Germany - Tel: +49-(0)89-15 00 12-11, Fax: +49-(0)89-15 00 12-22
Japan - Tel: +81-(0)3-3814-7688, Fax: +81-(0)3-3814-7689
Asia - Tel: +65- 6489-4801

Please note that this manual was developed from the EMACE PLUS SDK Manual (MN-69092SX-001). Please refer to the last page of this manual for record of change to the original manual.

Revision	Date	Pages	Description
A	2/2009	All	Initial Release - Added the AceXtreme information throughout the manual. New section of functions added – Multi-RT
B	5/2009	71-80, 123, 567, 604, 637, 639, 675, 694, 701, 707	Minor updates on indicated pages
C	6/2009	161, 163, 165, 167, 231, 235	Function reference descriptions edited. Changed acexMRTDataBlkCreate to aceRTDataBlkCreate.
D	8/2009	519 - 954	Minor edits made to aceRTSetAddress, aceRTSetAddrSource, and AceRTModeCodeWriteData. New Function subsections AvionicsI/O and Discrete I/O added. New functions and structures added for software version 3.1.2
E	1/2010	449-461, 509-521	Edited the following functions: aceBCOpCodeCreate, acexBCMemObjCreate, acexBCMemObjDelete, acexBCMemWrdCreate, acexBCMemWrdDelete, acexBCMemWrdRead, acexBCMemWrdWrite
F	7/2010	All	New Format Applied. Function sections removed and created into new manual.
G	10/2010	22	Removed section 1 that had the software licensing Table 4 added dots for RTMTI in both SF and MF columns Added a note for ACE _MODE_MT and ACE_MODE_RTMT as “Not Recommended for new designs” for AceXtreme
H	12/2010	120	In table 45, in the ACE_RT_OPT_ALT_STS row, "aceRTStatusBitsXlear()" was changed to "aceRTStatusBitsClear()"
J	5/2011	17	Updated Table 3 to incorporate BU-67211U
K	12/2011	48	Updated Table 15
L	4/2012	73, 128	Updated aceBCGetMsgFromIDRaw, Changed text from “cognizant” to “contiguous”
M	12/2015	various	Updates to Windows, Linux, and VxWorks sections. New sections added for DIOs and AIOs, updated description of the BSW for all modes.

1 PREFACE

This manual uses typographical conventions to assist the reader in understanding the content. This section will define the text formatting used in the rest of the manual.

1.1 Text Usage

- **BOLD** – text that is written in bold letters indicates important information and table, figure, and chapter references.
- `Courier New –` is used to indicate code examples.
- <…> - Indicates user entered text or commands.

1.2 Special Handling and Cautions

The BU-69092 is delivered on a Compact Disc. Proper care should be used to ensure that the discs are not damaged by heat.

1.3 Trademarks

All trademarks are the property of their respective owners.

1.4 Technical Support

In the event that problems arise beyond the scope of this manual, you can contact In the event that problems arise beyond the scope of this manual, you can contact DDC by the following:

US Toll Free Technical Support:
1-800-DDC-5757, ext. 7771

Outside of the US Technical Support:
1-631-567-5600, ext. 7771

Fax:
1-631-567-5758 to the attention of DATA BUS Applications

DDC Website:
www.ddc-web.com/ContactUs/TechSupport.aspx

Please note that the latest revisions of Software and Documentation are available for download at DDC's Web Site, www.ddc-web.com.

2 OVERVIEW

2.1 Description

The AceXtreme® C Software Development Kit (SDK) provides the framework for developing applications for DDC’s series of MIL-STD-1553 components and cards, while using minimal development time.

This SDK is written such that all low level access to the DDC MIL-STD-1553 communication processor is simplified through a set of API functions in the interface and interrupt control modules. This abstraction allows one common software interface to any DDC MIL-STD-1553 cards or components.

Note: *Version 3.0.1 of the BU-69092S0 only supports the AceXtreme boards.*

2.2 Features

- Library of "C" Routines Available for:
  Windows® XP (32 Bit) and Vista/7/8 (32/64 Bit), Linux®(32/64 Bit), and VxWorks® (32-Bit) Operating Systems
- Documentation Provided
- Provides Modular, Portable, & Readable Code to Reduce Software Development Time
- "C" Structures Eliminate Need to Learn Detailed Address/Bit Maps and Data Formats
- Includes Sample Programs and Compiled Libraries for Quick Startup
- Includes Multiple Environment/Compiler Support

2.3 System Requirements

One or more of the following:

- Windows XP, Windows Vista 32/64-bit, Windows 7 32/64-bit, Windows 7 32/64-bit, Linux, or VxWorks (32-Bit) Operating System.
- Workbench software development environment for VxWorks platforms.
- An appropriate compiler or development environment.
- Contact Factory for additional Operating Systems.

2.4 DDC MIL-STD-1553 Device Families

DDC has been developing MIL-STD-1553 interface cards and components for 30 years. There have been numerous generations of devices to add new functionality and improved performance. The BU-69092Sx Software Development Kit supports the 3 latest generations of MIL-STD-1553 hardware families (AceXtreme, E^{2}MA, and EMA).

1. AceXtreme :
2. E^{2}MA (Extended Enhanced Mini-ACE):
3. EMA (Enhanced Mini-ACE):

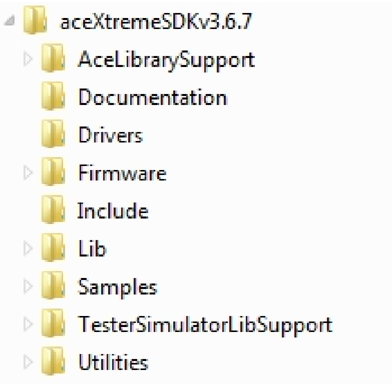
Table 1. DDC MIL-STD-1553 Family Features

Family	***EMA*** Released 1999	***E^{2}MA*** Released 2006	***AceXtreme*** Single-Function Released 2009	***AceXtreme*** Multi-Function Released 2010
Part Number	BU-6555X BU-6556X	BU-65577X BU-65578X BU-6559XX	BU-671XX	BU-672XX
Time-Tag support	16-bit Internal 2µs	48-Bit Internal 1µs	48-Bit Internal 100ns	48-Bit Internal 100ns
Hardware Triggers	Not-Supported	Not-Supported	Not-Supported	Supported
Inter-Message Routines	Not-Supported	Not-Supported	Not-Supported	Supported
Replay	Not-Supported	Not-Supported	Not-Supported	Supported
BC Message Gap	>= 6µs	>= 6µs	>= 6µs	>= 3.5µs
RT Response Time	7µs	7µs	7µs	>= 3.5µs
BC Frame Time	16-bit	16-bit	16-bit	24-bit
BC Streaming/Data Arrays	Not-Supported	Not-Supported	Supported	Supported
RT Streaming/Data Arrays	Not-Supported	Not-Supported	Supported	Supported
Multiple RT Addresses	Not-Supported	Not-Supported	Supported	Supported
MT IRIG106 Chapter 10	Not-Supported	Supported	Supported	Supported
MT Advanced Error Sampling (AES)	Not-Supported	Not-Supported	Not-Supported	Supported
BC Memory OpCodes	Not-Supported	Not-Supported	Supported	Supported
Error Injection	Not-Supported	Not-Supported	Not- Supported	Supported
MIL-STD-1553 Operating Modes	See Table 11. MIL-STD-1553 Channel Modes			

***Note:** Support for most EMACE and E^{2}MA Devices with the AceXtreme SDK ended with version 3.5.3. Check the SDK's release notes for which boards of the EMACE and E^{2}MA families are still supported with the most current version of the SDK.*

2.5 AceXtreme SDK Directory Structure for Windows

The installation of the AceXtreme SDK for Windows will result in the creation of the folder C:\DDC\aceXtremeSDKvX.Y.Z (where XYZ represents the version of the SDK). The "AceXtremeSDKvX.Z.Y" folder contains the drivers, library, header and samples. Also included are the binary firmware files for applicable DDC hardware. The directory also includes the ACE Library Support Package and Tester Simulator Library Support Package. The Ace Library Support Package allows applications written for the ACE Library (BU-6908x) to run on any DDC MIL-STD-1553 device without the need to recompile source code. While the Tester Simulator Library Support package offers support for applications written for the Tester Simulator Library (BU-69068x), allowing the applications to run on any Multi-Function AceXtreme Based card without the need to recompile source code.



aceXtremeSDKv3.6.7
- AceLibrarySupport
- Documentation
- Drivers
- Firmware
- Include
- Lib
- Samples
- TesterSimulatorLibSupport
- Utilities

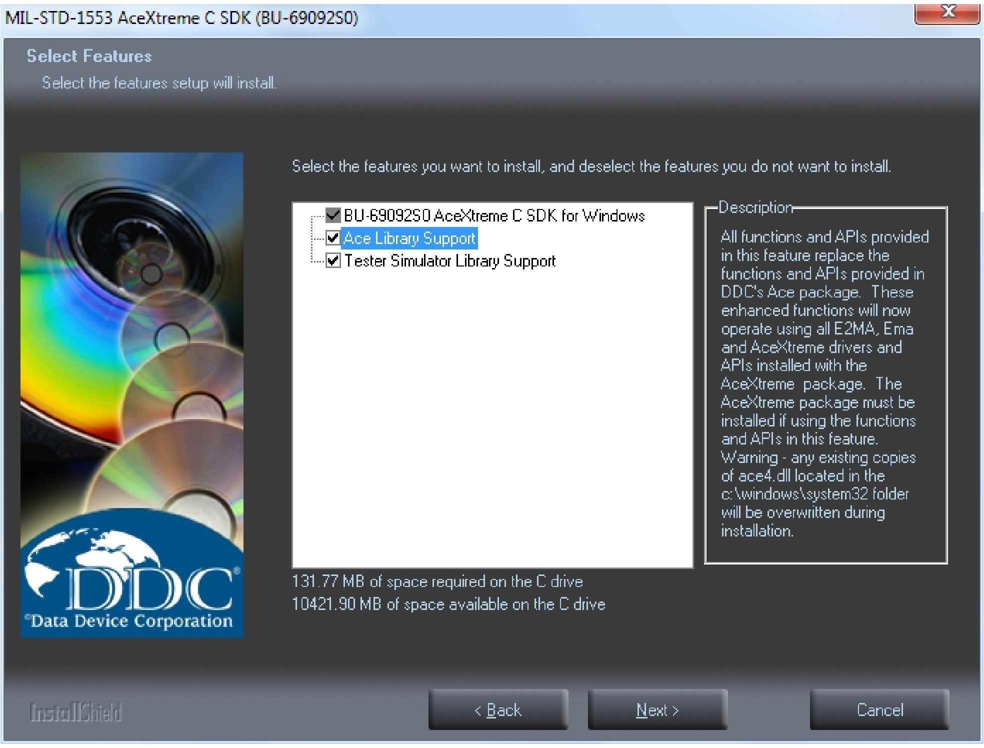
Figure 1. AceXtreme SDK for Windows Directory Structure

Table 2. AceXtreme SDK Directory Structure for Windows

Folder Name	Description of Contents
AceLibrarySupport	Ace4.lib, Ace4.dll, include files and Ace Library samples.
Documentation	ReleaseNotes.txt, and ReadMe.txt. Contains version information on SDK.
Drivers	Driver (.sys) and information files (inf) for driver installation.
Firmware	Firmware for applicable DDC hardware and manual for updating flash
Include	Header files for the AceXtreme SDK.
Lib	Library files required for linking to AceXtreme dll (emacepl.dll and emacepl.lib).
Samples	Samples for AceXtreme SDK (See Section 2.5.7).
TesterSimulatorLibSupport	TestSim32.lib, TestSim.lib, include files and Tester Simulator library samples
Utilities	Application used to enable Tx inhibit and BC Disable on an AceXtreme Device.

2.5.1 AceLibrarySupport

The “AceLibrarySupport” Directory contains the files needed to support current 1553 devices using applications originally written for the ACE Library (BU-6908X). This directory is an optional directory and is only installed when the ACE Library Support option is selected during the AceXtreme SDK installation.



MIL-STD-1553 AceXtreme C SDK (BU-69092S0)

Select Features

Select the features setup will install.

Select the features you want to install, and deselect the features you do not want to install.

- ☑ BU-69092S0 AceXtreme C SDK for Windows
  - ☑ Ace Library Support
  - ☑ Tester Simulator Library Support

Description

All functions and APIs provided in this feature replace the functions and APIs provided in DDC's Ace package. These enhanced functions will now operate using all E2MA, Ema and AceXtreme drivers and APIs installed with the AceXtreme package. The AceXtreme package must be installed if using the functions and APIs in this feature. Warning - any existing copies of ace4.dll located in the c:\windows\system32 folder will be overwritten during installation.

131.77 MB of space required on the C drive
10421.90 MB of space available on the C drive

Figure 2. Ace Library Option Screen

The ACE Library Support option is designed to run applications originally developed using the ACE library on current 1553 devices without requiring a recompile of the source code.   The AceLibrarySupport Directory contains a new Ace4.lib and Ace4.dll, (the new ace4.dll is copied to C:\Windows\System32 and C:\windows\sysWoW64 upon installation of the ACE Library support option), the header files for the ACE Library and the original samples that came with the ACE Library.

2.5.2 Documentation

The "Documentation" directory contains the Release Notes and ReadMe files for the AceXtreme SDK. The release notes contain information regarding each SDK version such as, the current firmware version for applicable DDC hardware, known issues and corrected defects. Other files included in this directory are the software license agreement and a link to this user's manual.

2.5.3 Drivers

The BU-69092Sx software package includes Operating system specific drivers based on the specific release version desired. Table 3 below details the driver file names for Windows (BU-69092S0) based on the DDC MIL-STD-1553 Hardware that are supported in the current version of the AceXtreme SDK.

Table 3. Drivers of DDC Hardware

Windows	Supported Cards
acexpci.sys	BU-67101Q, BU-67104C, BU-67105, BU-67106K, BU-67107X, BU-67108C, BU-67109C, BU-67110X,BU-67112X, BU-67118Y/Z, BU-67114H, BU-67206BK, BU-67210X
acexusb.sys	BU-67102U, BU-67103U, BU-67113, BU-67202U, BU-67211U
e2mausb.sys	BU-65590U, BU-65591U
emapci.sys	BU-65586H, BU-65596F/M
Emaebrpci.sys	BU-65580M
Remote Access	BU-67115W, BU-67116W, BU-67119W, BU-67121W

Support for most EMACE and E^{2}MA Devices with the AceXtreme SDK ended with version 3.5.3. Drivers for these cards are not part of the current release of the AceXtreme SDK. Check the SDK's release notes for which boards of the EMACE and E^{2}MA families are still supported with the most current version of the SDK. Table 4 shows the drivers removed from after version 3.5.3 of the AceXtreme SDK.

Table 4. Discontinued Drivers of DDC Hardware for Windows

Windows	Supported Cards
e2mapci.sys	BU-65577F/M/C, BU-65578 F/M/C, BU-65590F/M/C, BU-65591F/M/C
emapccrd.sys	BU-65553M2
emapci.sys	BU-65565F/M, BU-65566M, BU-65569i, BU-65569T/B

Make sure to install the AceXtreme SDK before installing the device driver for your hardware.

2.5.4 Firmware

DDC AceXtreme and E^{2}MA devices support firmware updates. These updates can be performed via the DDC Card Manager. The “Firmware” directory contains the instructions on how to flash your device (FLASH_UTLITY.pdf) and the latest firmware for all applicable DDC hardware, listed by hardware part number. It is recommended to use the latest firmware for your device. The DDC Card Manager will notify the user if there is an out of date firmware on your device. For more information on flashing your DDC device please refer to the Firmware update procedure.

2.5.5 Include Directory

The “Include” directory contains the header files needed for compiling an application written for the AceXtreme SDK. When creating an application the user will have to configure his compiler to include the path to this folder in order to correctly compile an executable.

2.5.6 Lib Directory

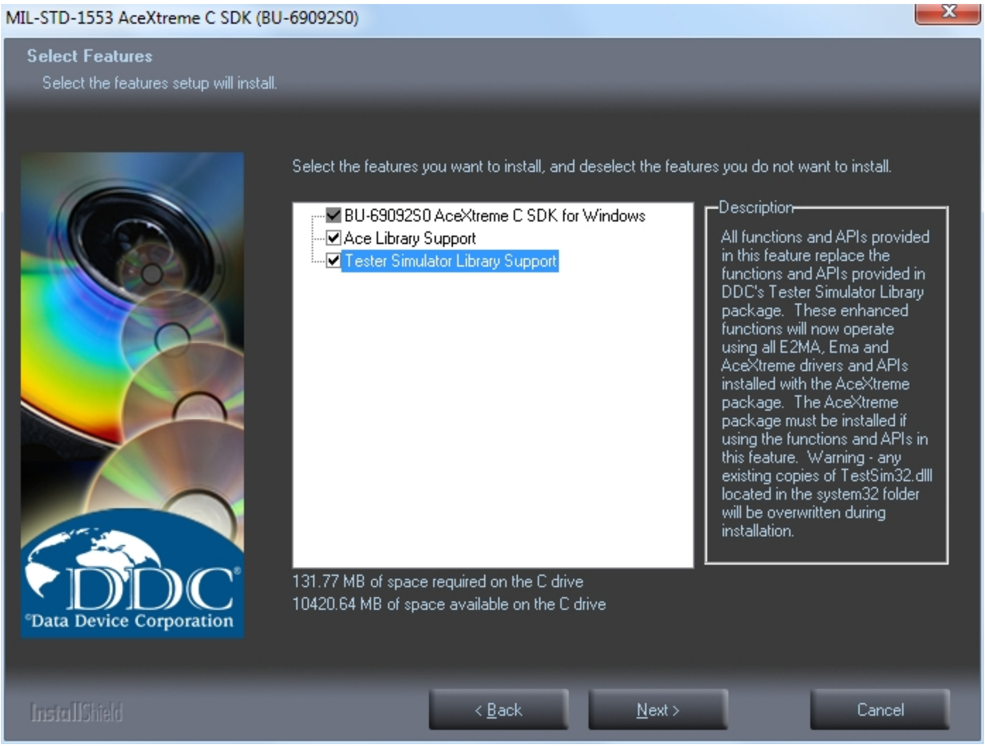
The “Lib” directory contains the emacepl.lib file, which is needed to compile an application written against the AceXtreme SDK. The lib folder contains two sub directories, Win32 and x64. The Win32 directory contains the 32-bit version of the emacepl.lib, while the x64 version contains the 64-bit version of the library file. Also included in each sub- directory is a second copy the emacepl.dll which is also installed in C:\Windows\System32 and C:\Windows\SysWow64. The file emacepls.lib is a statically built copy of the emacepl.lib file.

2.5.7 Samples

The “samples” directory contains the C source samples for the AceXtreme SDK. Each sample sub-directory contains an applicable project file and a compiled executable located in the (Win32\x64)\Release sub-directory of the sample. For more information on the supplied samples, see Section 4.

2.5.8 TesterSimulatorLibrarySupport

The “TesterSimulatorLibrarySupport” Directory contains the files needed to support the AceXtreme Multi-Function 1553 devices using applications originally written for the Tester Simulator Library (BU-69068Sx). This directory is an optional directory and is only installed when the Tester Simulator Library support option is selected during the AceXtreme SDK installation.



MIL-STD-1553 AceXtreme C SDK (BU-69092S0)

Select Features

Select the features setup will install.

Select the features you want to install, and deselect the features you do not want to install.

- ☑ BU-69092S0 AceXtreme C SDK for Windows
  - ☑ Ace Library Support
  - ☑ Tester Simulator Library Support

Description

All functions and APIs provided in this feature replace the functions and APIs provided in DDC's Tester Simulator Library package. These enhanced functions will now operate using all E2MA, Ema and AceXtreme drivers and APIs installed with the AceXtreme package. The AceXtreme package must be installed if using the functions and APIs in this feature. Warning - any existing copies of TestSim32.dlll located in the system32 folder will be overwritten during installation.

131.77 MB of space required on the C drive
10420.64 MB of space available on the C drive

Figure 3. Tester Simulator Library Option Screen

The Tester Simulator Library support option is designed to run applications originally developed using the Tester Simulator library on current AceXtreme Multi-function 1553 devices without requiring a recompile of the source code. The “TesterSimulatorLibSupport” Directory contains a new testsim32.lib and testsim32.dll, (the new testsim32.dll is copied to C:\Windows\System32 and C:\Windows\sysWoW64 upon installation of the Tester Simulator Library support option), the header files for the Tester Simulator Library and the original samples that came with the Tester Simulator Library.

2.5.9 Utilities

The DDC AceXtreme cards have a unique feature allowing the user to enable both “Tx Inhibit” (Transmit Inhibit) and/or “BC Disable” (Bus Controller Disable) on selected MIL-STD-1553 channel(s). This directory contains the utilities to create a binary file which disables transmissions on a 1553 channel on the AceXtreme Device. For more information on Transmit Inhibit and BC Disable see ***Application Note #50*** ***“AceXtreme Transmit Inhibit and Bus Controller Disable Features”***.

2.6 AceXtreme SDK Directory Structure for Linux

The installation of the AceXtreme SDK for Linux will result in the creation of the folders ~/aceXtremeSDKvX.Y.Z (where XYZ represents the version of the SDK). The location and directory name may vary, as it is up to the users discretion on where to store the AceXtreme SDK.

The "AceXtremeSDKvX.Z.Y" folder contains the drivers, library, header and samples, and DDC Card Manager. Also included are the binary firmware files for applicable DDC hardware. The directory also includes the ACE Library Support Package and Tester Simulator Library Support Package. The Ace Library Support Package allows applications written for the ACE Library (BU-6908x) to run on any DDC MIL-STD-1553 device without the need to recompile source code. While the Tester Simulator Library Support package offers support for applications written for the Tester Simulator Library (BU-69068x), allowing the applications to run on any Multi-Function AceXtreme Based card without the need to recompile source code.

The AceXtreme Library files and header files are located in the "~/aceXtremeSDKvX.Y.Z /libraries/emacepl directory. The AceXtreme SDK samples are located in the "~/aceXtremeSDKvX.Y.Z /samples/emacepl directory.

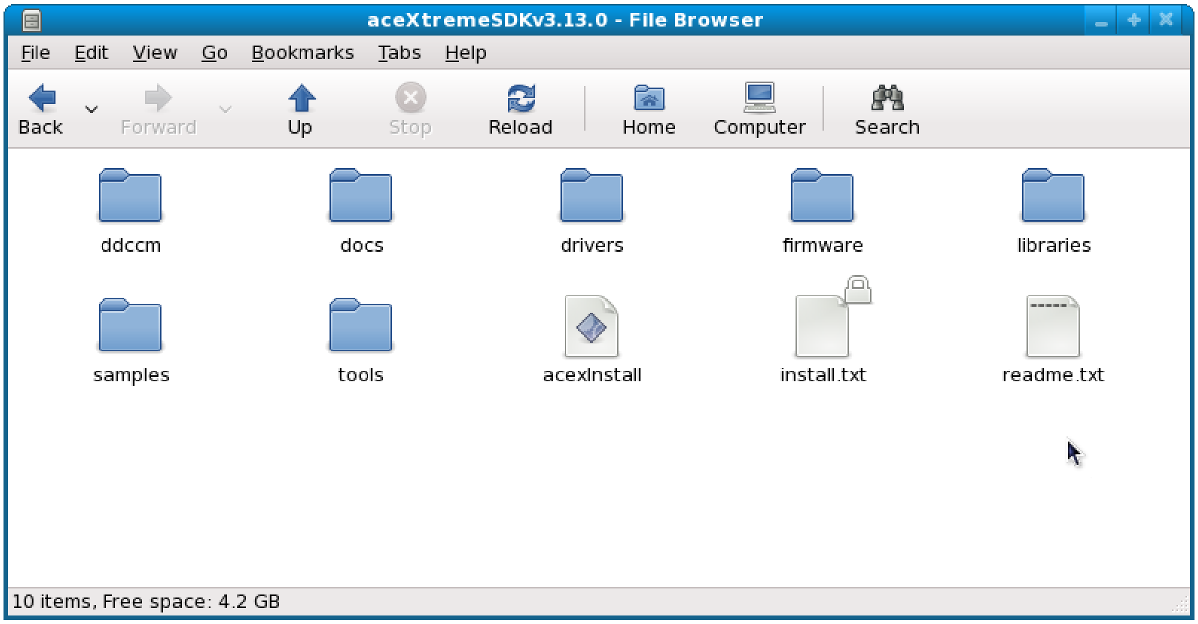


Figure 4. AceXtreme SDK for Linux Directory Structure

Table 5. AceXtreme SDK Directory Structure for Linux

Folder Name	Description of Contents
ddccm	DDC Card Manager source code.
docs	ReleaseNotes.txt, and ReadMe.txt. Contains version information on SDK.
drivers	Driver source directory.
firmware	Firmware for applicable DDC hardware and manual for updating flash.
libraries	Library support files, for AceXtreme (emacepl), Ace Library (acetoemace), and Tester Simulator Library (tstsim).
samples	Samples for AceXtreme SDK (See Section 2.5.7).
tools	Library install scripts, Application to enable Tx inhibit and BC Disable.

2.6.1 ddccm

The “ddccm” directory contains the source files of the DDC Card Manager (located in the ddccm/src directory) and the make file (located in the ddccm/prj directory). Once ddccm is built, the makefile will copy the binary of ddccm to /usr/sbin.

The DDC Card Manager (ddccm) is used to assign a Logical Device Number (LDN) to any MIL-STD-1553 or ARINC 429 devices. Firmware updates are also done through ddccm.

2.6.2 docs

The “docs” directory contains the Release Notes, Install instructions, and software license files for the AceXtreme SDK. The release notes contain information regarding each SDK version such as, the current firmware version for applicable DDC hardware, known issues and corrected defects.

2.6.3 drivers

The “drivers” directory contains two sub-directories, “acex” and “legacy”. The “acex” directory contains the PCI/PCIe Linux devices drivers for the AceXtreme based devices, while the “legacy” folder contains the AceXtreme USB, EMACE PCI drivers.

Table 6 below details the driver file names for Linux (BU-69092S1) based on the DDC MIL-STD-1553 Hardware.

Table 6. Drivers of DDC Hardware for Linux

Linux Name	Supported Cards
acex	BU-67101Q, BU-67104C, BU-67105, BU-67106K, BU-67107X, BU-67108C, BU-67109C, BU-67110X,BU-67112X, BU-67118Y/Z, BU-67114H, BU-67206BK, BU-67210X
legacy/acexusb	BU-67102U, BU-67103U, BU-67113, BU-67202U, BU-67211U
legacy/e2mausb	BU-65590U, BU-65591U
legacy/emapci	BU-65596F/M
Remote Access	BU-67115W, BU-67116W, BU-67119W, BU-67121W

Support for most EMACE and E^{2}MA Devices with the AceXtreme SDK ended with version 3.7.0. Drivers for these cards are not part of the current release of the AceXtreme SDK. Check the SDK's release notes for which boards of the EMACE and E^{2}MA families are still supported with the most current version of the SDK. Table 4 shows the drivers removed from after version 3.7.0 of the AceXtreme SDK.

Table 7. Discontinued Drivers of DDC Hardware for Linux

Linux	Supported Cards
e2mapci.sys	BU-65577F/M/C, BU-65578 F/M/C, BU-65590F/M/C, BU-65591F/M/C
emapci.sys	BU-65565F/M, BU-65566M, BU-65569i, BU-65569T/B

2.6.1 Firmware

Most DDC MIL-STD-1553 devices support firmware updates. These updates can be performed via the **DDC Card Manager**. The "Firmware" directory contains the instructions on how to flash your device (FLASH_UTLITY.pdf) and the latest firmware for all applicable DDC hardware, listed by hardware part number. For more information on flashing your DDC device please refer to the Firmware update procedure.

2.6.2 libraries

The "libraries" directory contains the binary library files and the header files for the AceXtreme SDK (emacepl), the Ace Library (acetoemace) and the Tester Simulator Library (tstsim).

2.6.2.1 AceXtreme Library (emacepl)

The "~/aceXtremeSDKvX.Y.Z /libraries/emacepl" directory contains the AceXtreme files needed to build applications against the AceXtreme SDK.

The emacepl/bin directory contains the binary library files, which are copied to the /usr/lib/ directory with the “install-emacepl.sh” script (located in the tools directory), and the AceXtreme Library header files (located in the /src directory). The samples for the AceXtreme SDK can be found in the “~/aceXtremeSDKvX.Y.Z /samples/emacepl directory.

2.6.2.2 Ace Library (acetoemace)

The “~/aceXtremeSDKvX.Y.Z/libraries/acetoemace” directory contains the files needed to support current 1553 devices using applications originally written for the ACE Library (BU-6908X). The ACE Library support is designed to run applications originally developed using the ACE Library on current 1553 devices without requiring a recompile of the source code.

The acetoemace directory contains the binary library files, which are copied to the /usr/lib/ directory with the “install-acetoemace.sh” script (located in the tools directory), and the Ace Library header files (located in the /src directory). The samples for the Ace Library can be found in the “~/aceXtremeSDKvX.Y.Z /samples/acetoemace directory.

2.6.2.3 Remote Access (ethernet_socket)

The “~/aceXtremeSDKvX.Y.Z/libraries/ethernet_socket” directory contains the files needed to Remote access mode for DDC ABD and AIC devices. The files in the ethernet_socket directory allow for an application written and executed on the Linux host machine to access DDC’s ABD or AIC over Ethernet. For more information on remote access usage, see the ABD or AIC Software manual.

2.6.2.4 Tester Simulator Library (tstsim)

The “~/aceXtremeSDKvX.Y.Z/libraries/tstsim” Directory contains the files needed to support the AceXtreme Multi-Function 1553 devices using applications originally written for the Tester Simulator Library **(**BU-69068Sx). The Tester Simulator Library support is designed to run applications originally developed for the Tester Simulator library without requiring a recompile of the source code.

The “~/aceXtremeSDKvX.Y.Z/libraries/tstsim directory contains the binary library files, which are copied to the /usr/lib/ directory with the “install-tstsim.sh” script (located in the tools directory), and the Tester Simulator Library header files (located in the /src directory). The samples for the Tester Simulator Library can be found in the “~/aceXtremeSDKvX.Y.Z /samples/tstsim directory.

2.6.3 samples

The “samples” directory contains the C source samples for the AceXtreme SDK. There is a subdirectory for each Library supported by the AceXtreme SDK (emacepl, acetoemace, and tstsim). Each library sample sub-directory contains an applicable prj folder which contains the makefile used to build the sample, and a src folder which holds the C source file. For more information on the supplied samples, see Section 4.

2.6.4 tools

The tools folder is used to store the install scripts to install the libraries supported by the AceXtreme SDK.

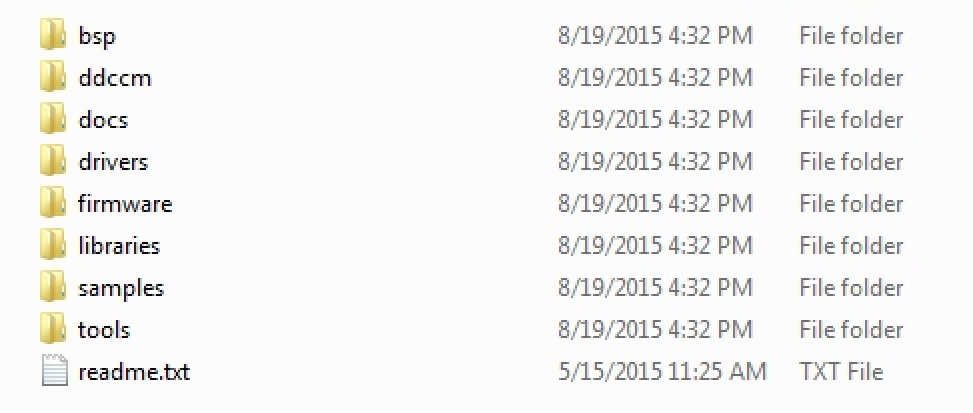
The DDC AceXtreme cards have a unique feature allowing the user to enable both “Tx Inhibit” (Transmit Inhibit) and/or “BC Disable” (Bus Controller Disable) on selected MIL-STD-1553 channel(s). The tx_bc_init_gen directory contains the utilities to create a binary file which disables transmissions on a 1553 channel on the AceXtreme Device. For more information on Transmit Inhibit and BC Disable see ***Application Note #50*** ***“AceXtreme Transmit Inhibit and Bus Controller Disable Features”***.

2.7 AceXtreme SDK Directory Structure for VxWorks

The installation of the AceXtreme SDK for VxWorks will result in the creation of the folders “\BU69092S2” (where XYZ represents the version of the SDK). The location and directory name may vary, as it is up to the users discretion on where to store the AceXtreme SDK.

The “\BU69092S2” folder contains the drivers, library, header and samples, and DDC Card Manager. Also included are the binary firmware files for applicable DDC hardware.

The AceXtreme Library files and header files are located in the “\BU69092S2 \libraries\emacepl” directory. The AceXtreme SDK samples are located in the “\BU69092S2\samples\emacepl” directory.



```
bsp            8/19/2015 4:32 PM     File folder
ddccm          8/19/2015 4:32 PM     File folder
docs           8/19/2015 4:32 PM     File folder
drivers        8/19/2015 4:32 PM     File folder
firmware       8/19/2015 4:32 PM     File folder
libraries      8/19/2015 4:32 PM     File folder
samples        8/19/2015 4:32 PM     File folder
tools          8/19/2015 4:32 PM     File folder
readme.txt     5/15/2015 11:25 AM    TXT File
```

Figure 5. AceXtreme SDK for VxWorks Directory Structure

Table 8. AceXtreme SDK Directory Structure for VxWorks

Folder Name	Description of Contents
bsp	Contains the ddcBspConfig.c files for Pre-VxBus Driver model.
ddccm	DDC Card Manager source code.
docs	ReleaseNotes.txt, and ReadMe.txt. Contains version information on SDK.
drivers	Driver source directory.
firmware	Firmware for applicable DDC hardware and manual for updating flash.
libraries	Support files for AceXtreme (emacepl), Ace (acetoemace), and Tester Simulator (tstsim) libraries.
samples	Samples for AceXtreme SDK (See Section 2.5.7).
tools	Library install scripts, Application to enable Tx inhibit and BC Disable.

The readme.txt file contains instructions on how to integrate the BU-69092S2 AceXtreme SDK for VxWorks with your BSP or Downloadable kernel module. The readme.txt file will point to the appropriate driver install/build instructions and will also list what project settings need to be modified for the AceXtreme SDK.

2.7.1 bsp

The “bsp” folder contains the SBC specific ddcBspConfig.c file needed when using the Pre-VxBus driver model. The Pre-VxBus driver is required for VxWorks 6.5 and earlier, and for SMP VxWorks 6.6 and later. The folder also has a User-Defined folder which contains the skeleton of the ddcBspConfig.c file so that it can be modified to for a SBC that isn’t supported by the BU-69092S2 VxWorks AceXtreme SDK.

2.7.2 ddccm

The “ddcm” directory contains the source files of the DDC Card Manager. The DDC Card Manager (ddccm) is used to assign a Logical Device Number (LDN) to any MIL-

STD-1553 or ARINC 429 devices. Firmware updates are also done through ddccm. Running ddccm is necessary to initialize the device, and must be done once per boot of the system.

2.7.3 docs

The “docs” directory contains the Release Notes, driver setup instructions, and software license files for the ***AceXtreme SDK***. The release notes contain information regarding each SDK version such as, the current firmware version for applicable DDC hardware, known issues and corrected defects.

The “pre-vxbus_driver_readme.txt” file should be used when using the Pre-VxBus Driver model (VxWorks 6.5 and earlier). This file will instruct the user on how to build the driver and integrate the ddcBspConfig.c file into the kernel image.

The “vxbus_driver_readme.txt” file will instruct the user how to build the VxBus driver for VxWorks 6.6 and later. This file will detail the steps to build the driver and how to include the driver in the kernel image.

2.7.4 drivers

The “drivers” directory contains three sub-directories, “acex” and “legacy” and pre-vxbus. The “acex” directory contains the PCI/PCIe VxWorks devices drivers for the AceXtreme based devices, while the “legacy” folder contains the AceXtreme USB, EMACE PCI drivers.

Table 6 below details the driver file names for VxWorks (BU-69092S2) based on the DDC MIL-STD-1553 Hardware.

Table 9. Drivers of DDC Hardware for VxWorks

VxWorks Name	Supported Cards
acex	BU-67101Q, BU-67104C, BU-67105, BU-67106K, BU-67107X, BU-67108C, BU-67109C, BU-67110X,BU-67112X, BU-67118Y/Z, BU-67114H, BU-67206BK, BU-67210X, BU-67301I
legacy/acexusb	BU-67102U, BU-67103U, BU-67113U, BU-67202U, BU-67211U
legacy/emapci	BU-65596F/M
Remote Access	BU-67115W, BU-67116W, BU-67119W, BU-67121W

Support for most EMACE and E^2MA Devices with the AceXtreme SDK ended with version 3.3.4. Drivers for these cards are not part of the current release of the AceXtreme SDK. Check the SDK’s release notes for which boards of the EMACE and E^2MA families are still supported with the most current version of the SDK. Table 4 shows the drivers removed from after version 3.3.4 of the AceXtreme SDK.

Table 10. Discontinued Drivers of DDC Hardware for VxWorks

VxWorks	Supported Cards
e2mapci.sys	BU-65577F/M/C, BU-65578F/M/C, BU-65590F/M/C, BU-65591F/M/C
emapci.sys	BU-65565F/M, BU-65566M/R, BU-65569i, BU-65569T/B,

2.7.5 Firmware

Most DDC MIL-STD-1553 devices support firmware updates. These updates can be performed via the **DDC Card Manager**. The “Firmware” directory contains the instructions on how to flash your device (FLASH_UTLITY.pdf) and the latest firmware for all applicable DDC hardware, listed by hardware part number. For more information on flashing your DDC device please refer to the Firmware update procedure.

2.7.6 libraries

The “libraries” directory contains the binary library files and the header files for the AceXtreme SDK (emacepl), and the Remote Access mode on the ABD and AIC.

2.7.6.1 AceXtreme Library (emacepl)

The “\BU69092S2\libraries\emacepl” directory contains the AceXtreme files needed to build applications against the AceXtreme SDK. The samples for the AceXtreme SDK can be found in the “\BU69092S2\samples\emacepl” directory.

2.7.6.2 Remote Access (ethernet_socket)

The “\BU69092S2\libraries\ethernet_socket” directory contains the files needed to Remote access mode for DDC ABD and AIC devices. The files in the ethernet_socket directory allow for an application written and executed on the VxWorks host machine to access DDC’s ABD or AIC over Ethernet. For more information on remote access usage, see the ABD or AIC Software manual.

2.7.7 samples

The “samples” directory contains the C source samples for the AceXtreme SDK. Each library sample sub-directory contains an the C source file. For more information on the supplied samples, see Section 4.

2.7.8 tools

The tools folder is used to store the install script to copy the driver source to the appropriate VxWorks $(WIND_BASE) directory.   The DDC AceXtreme cards have a unique feature allowing the user to enable both “Tx Inhibit” (Transmit Inhibit) and/or “BC Disable” (Bus Controller Disable) on selected MIL-STD-1553 channel(s). The tx_bc_init_gen directory contains the utilities to create a binary file which disables transmissions on a 1553 channel on the AceXtreme Device. For more information on Transmit Inhibit and BC Disable see ***Application Note #50*** ***“AceXtreme Transmit Inhibit and Bus Controller Disable Features”***.

3 USING THE ACEXTREME C SDK

3.1 Initialization and Setup

3.1.1 Logical Device Numbers

The AceXtreme C SDK uses Logical Device Numbers to access DDC hardware. A Logical Device Number (LDN) is a unique identifier referring to a particular MIL-STD-1553 channel. Most SDK functions will require a target LDN as the first parameter. Depending on your Operating System, the Logical Device Numbers will be auto-assigned or will require some user interaction.

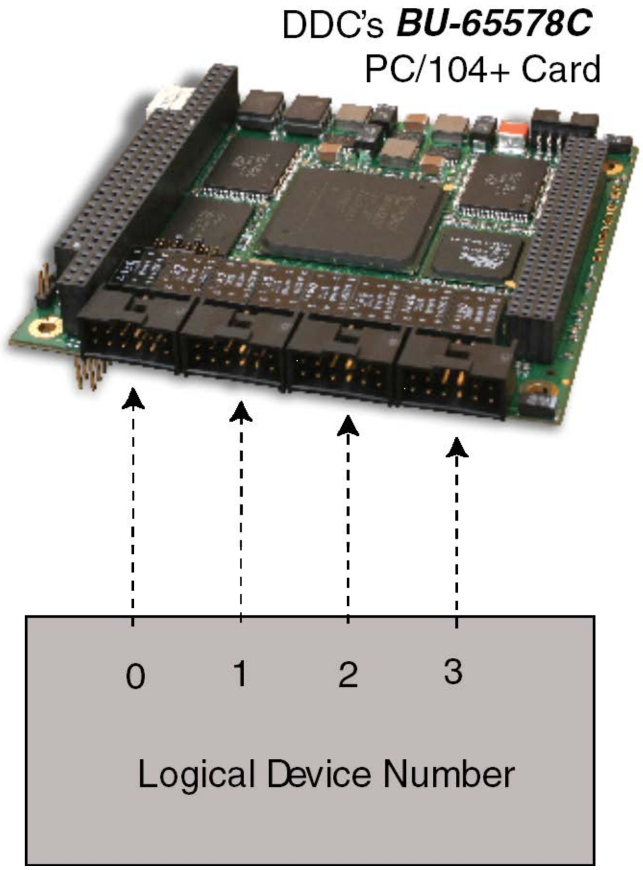


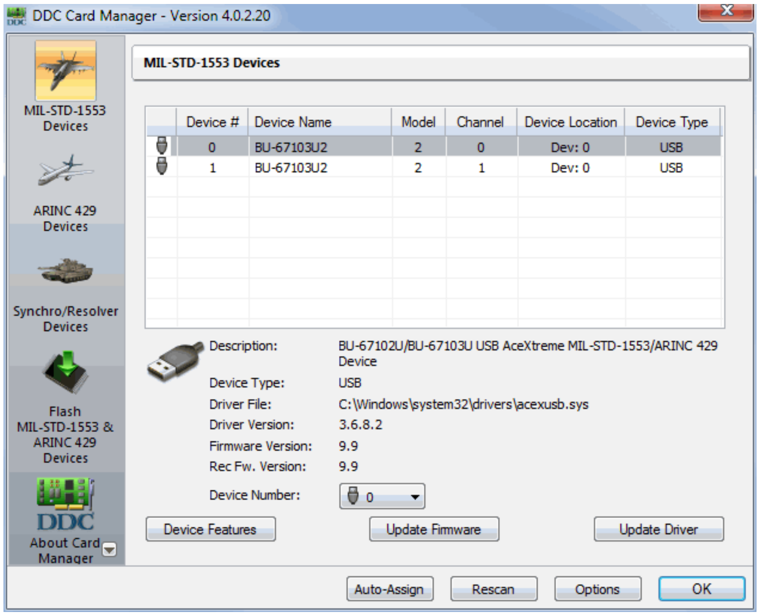
Figure 6. Logical Device Number Assignments

3.1.2 Assigning Logical Device Numbers

3.1.2.1 Windows

Logical Device Numbers in Windows are assigned using the DDC Card Manager Applet, found in the Windows Control Panel. The DDC Card Manager will display all connected hardware and will allow a unique LDN to be assigned to each channel. The DDC Card Manager also supports firmware upgrades (for applicable hardware); along with a driver update utility. To assign an LDN to any MIL-STD-1553 channel:

1. Select “MIL-STD-1553 Devices” in the left-hand column
2. Click on the desired device in the detected hardware list
3. Set/Change the Device Number via the drop-down list provided below



MIL-STD-1553 Devices

Device #	Device Name	Model	Channel	Device Location	Device Type
0	BU-67103U2	2	0	Dev: 0	USB
1	BU-67103U2	2	1	Dev: 0	USB

Description: BU-67102U/BU-67103U USB AceXtreme MIL-STD-1553/ARINC 429 Device
Device Type: USB
Driver File: C:\Windows\system32\drivers\acexusb.sys
Driver Version: 3.6.8.2
Firmware Version: 9.9
Rec Fw. Version: 9.9
Device Number: 0

Figure 7. DDC Card Manager for Windows

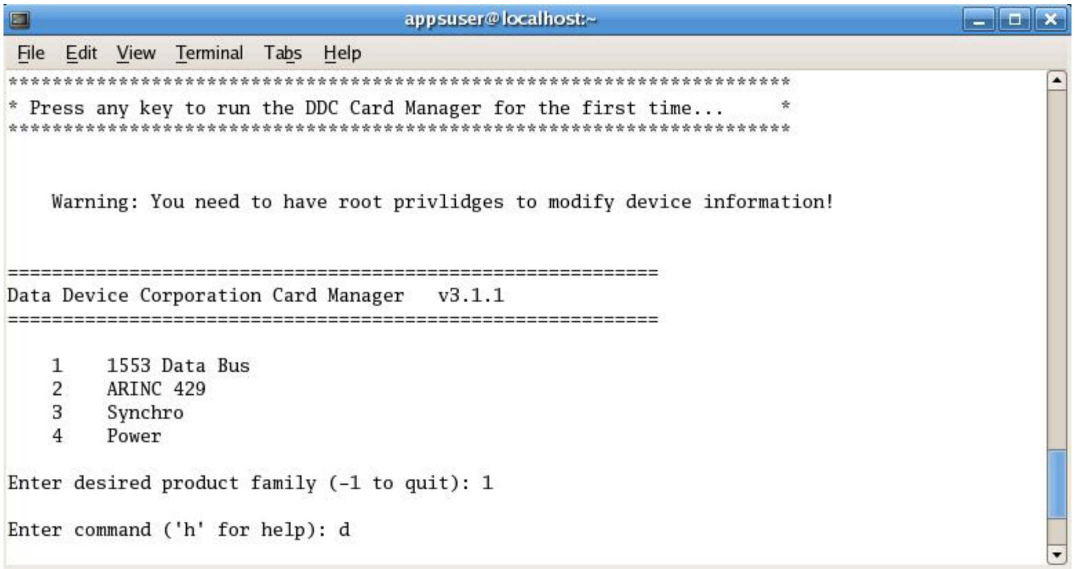
The DDC Card Manager for Windows will display the Driver version for the board, along with the Firmware version on the device (E2MA and AceXtreme devices only). The ‘Update Firmware’ button can be used to update the firmware on an applicable device.

Updating the device driver can also been accomplished by clicking on the ‘Update Driver’ button. A window will open so that the new firmware can be selected and then proceed with the firmware update.

The ‘Options’ and ‘Rescan’ buttons are used to search for any remote access devices on the network. Under the ‘Options’ window, the IP address of the remote access device can be configured, while the ‘Rescan’ button can be pressed to have the card manager look for any device changes to the system.

3.1.2.2 Linux

Logical Device Numbers in Linux are assigned using the DDCCM Text-Based Card Manager (See Figure 8) **./ddccm**, found in the **/usr/bin** folder. The Card Manager will display all connected hardware and will allow a unique LDN to be assigned to each channel. Use the menu-driven interface to assign LDN’s to all detected channels.



```
***********************************************************************
* Press any key to run the DDC Card Manager for the first time...     *
***********************************************************************



    Warning: You need to have root privlidges to modify device information!


===========================================================
Data Device Corporation Card Manager   v3.1.1
===========================================================

    1    1553 Data Bus
    2    ARINC 429
    3    Synchro
    4    Power

Enter desired product family (-1 to quit): 1

Enter command ('h' for help): d
```

Figure 8. DDC Card Manger for Linux

3.1.2.3 VxWorks

Logical Device Numbers in VxWorks are auto-assigned by calling the **ddccm()** function (defined in **ddccm source folder**). This function will call the necessary DDC VxWorks API calls to setup and detected hardware. After executing this function, a message will be displayed to the terminal informing the user of any found devices and their assigned LDN’s. The DDC Card Manager for VxWorks can be used to update the firmware on E^{2}MA and AceXtreme devices.

The call to ddccm() has two optional parameters. The first parameter is for family type, with the second parameter allows for the displaying of the device table which shows the assigned LDNs.

The family types for the first parameter are ‘1’ for MIL-STD-1553 devices only, ‘2’ for ARINC 429 Devices only, or -1 for both 1553 and 429 devices. If both parameters are blank when ddccm is called the DDC Card Manager will then default to -1 to configure both 1553 and 429, and display the assigned LDNs.

3.1.3 Initializing a MIL-STD-1553 Channel

After an LDN is assigned, that particular MIL-STD-1553 channel can be initialized to one of the following modes.

> ***Note:*** *Some modes may not be available on all hardware. For more information on each mode, see Section 3.3.*

Table 11. MIL-STD-1553 Channel Modes

Mode	Description	*EMA/ E^{2}MA*	*AceXtreme*	
			Single Function	Multi-Function
ACE_MODE_BC	Operate as MIL-STD-1553 Bus Controller (BC)	•	•	•
ACE_MODE_RT	Operate as a single MIL-STD-1553 Remote Terminal (RT)	•	•	•
ACE_MODE_MTI	Operate as an IRIG-106 Chapter 10 compliant monitor (MT-I)	•	•	•
ACE_MODE_RTMTI	Operate as a combined IRIG-106 Chapter 10 Monitor and Remote Terminal (RTMT-I)	•	•	•
ACE_MODE_MT	Classic Monitor (MT)	•	•**	•**
ACE_MODE_RTMT	Combined Remote Terminal and Classic Monitor (RTMT)	•	•**	•**
ACE_MODE_BCMTI	Operates as a combined BC and IRIG-106 Chapter 10 Monitor		•	•
ACE_MODE_MRT	Operate in Multi-RT mode		•	•
ACE_MODE_MRTMTI	Operate in combined Multi-RT and IRIG-106 Chapter 10 Monitor		•	•
ACE_MODE_ALL	Operates in combined BC, Multi-RT mode, and IRIG-106 Chapter 10 Monitor			•

*****Note :** ACE_MODE_MT and ACE_MODE_RTMT are “Not Recommended for new designs” with AceXtreme hardware.*

Channel initialization is accomplished via the **aceInitialize()** function. The following minimal inputs are required.

Table 12. Initialization Minimal Input Requirements

Parameter	Name	Function
1	Logical Device Number (LDN)	The LDN of target channel to initialize
2	Fixed Value: ACE_ACCESS_CARD	Informs the SDK that physical DDC hardware is to be used
3	Mode of Operation	Desired mode of Operation (See above)

```
S16BIT nResult;

/* Initialize Device */
nResult = aceInitialize(
    0,                           /* LDN */
    ACE_ACCESS_CARD,             /* Access Mode */
    ACE_MODE_BC,                 /* Mode of Operation */
    0,                           /* Reserved (Adv) */
    0,                           /* Reserved (Adv) */
    0);                          /* Reserved (Adv) */

if(nResult)
    printf(“aceInitialize() Error: Code %d\n”, nResult);
```

Code Example 1. Initializing LDN #0 to Act as a Bus Controller (BC)

3.2 General Concepts

The following general concepts apply to more than one mode of operation within the AceXtreme C SDK. It is recommended that this section be read in its entirety to gain a general understanding and before moving on to Section 3.3.

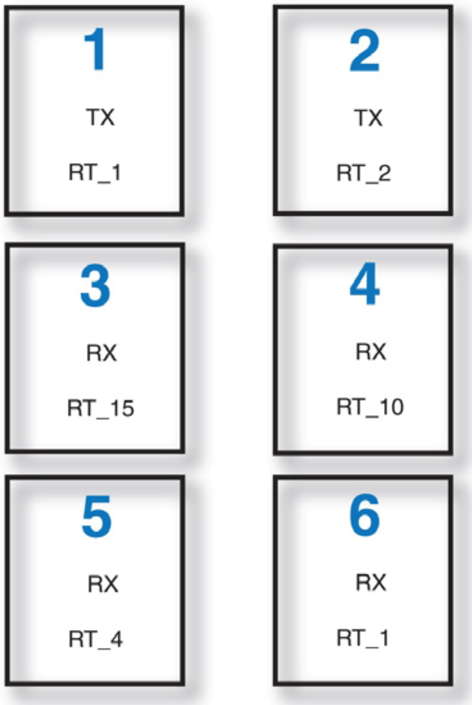
3.2.1 Object Unique Identifiers (OUID)

Every data object created within the AceXtreme C SDK requires an Object Unique Identifier (OUID) to be assigned. An OUID is a unique numerical value assigned by the user that can be used to reference that object.   A valid OUID can range from 1 to 65535 and must not conflict with any other OUID already assigned for that particular object type.

The OUID will be assigned during the object’s Create function. Once assigned, the OUID will be used to reference the object for it to be modified, accessed or deleted.

Table 13. Data Object Types

Operation Mode	Data Object Type
BC	Instruction Opcodes
BC	Data Blocks
BC	Messages
BC	Frames
RT/multi-RT	Data Blocks



To the right are six BC messages defined, with the OUID's in **BLUE**.

Each BC message was defined via **aceBCMsgCreate()** and assigned an OUID.

If any of the messages need to be modified (via **aceBCMsgModify()**), it can be referenced by the OUID initially assigned.

Figure 9. Defined BC Messages Showing OUID and Message Configuration

3.2.2 Hardware Time Tags

All DDC hardware has a built-in, hardware-driven, free running timing device or time tag. This time tag is used as the relative time for all message time stamping.

The resolution of the hardware Tag is configurable. Resolution can be configured at fixed values ranging from 100ns to 64µs/LSb. Please reference the respective hardware manual of the MIL-STD-1553 device for more information on the supported options. In addition, some DDC devices support an external time source. To change the time tag resolution or to use an external time source, see the **aceSetTimeTagRes()** API function.

The hardware time tag can be read or written independently using the following functions. Please see the ***BU-69092SX-003 AceXtreme Reference Guide*** for proper syntax.

16-Bit Time Tag

aceSetTimeTagValue()
aceGetTimeTagValue()

48-bit Time Tag (*AceXtreme / E²MA* Only)

aceSetTimeTagValueEx()
aceGetTimeTagValueEx()

```
S16BIT nResult;

/* Set Timetag Resolution */
nResult = aceSetTimeTagRes(
    0,                          /* LDN */
    ACE_TT_2US);                /* Time Tag Resolution */

if(nResult)
   printf(“aceSetTimeTagValueRes() Error: Code%d\n”,nResult);

/* Set Timetag Value*/
nResult = aceSetTimeTagValueEx(
    0,                          /* LDN */
    0x0011223344556677);        /* New TT Value */

if (nResult)
   printf(“aceSetTimeTagValueEx() Error: Code %d\n”,nResult);
```

Code Example 2. Setting Resolution (2µs/LSB) & Value of the Hardware 48-bit Time Tag

3.2.3 Configuring Hardware Interrupts and Callback Routines

The AceXtreme C SDK allows the user to setup an event-driven callback routine based on a hardware interrupt (when available). This allows the user to be notified of specific events and/or error conditions, thus relieving the need to continuously poll the 1553 data bus.

Each interrupt event, as well as the callback routine, is assigned using the **aceSetIrqConditions()** function. Each call to this function will enable or disable the specified events based on the value of the second parameter (TRUE or FALSE).

The usage of each event is covered in more detail in the Modes of Operation section 3.3.

Table 14. Interrupt Events

Bit	Description
31(MSB)	Master Interrupt
30	BC OP Code Parity Error
29	RT Illegal Command/Message Monitor Message Received
28	General Purpose Queue/Interrupt Status Queue Rollover
27	Call Stack Pointer Register Error
26	BC Trap OP Code
25	RT Command Stack 50% Rollover
24	RT Circular Buffer 50% Rollover
23	Monitor Command Stack 50% Rollover
22	Monitor Data Stack 50% Rollover
21	Enhanced BC IRQ3
20	Enhanced BC IRQ2
19	Enhanced BC IRQ1
18	Enhanced BC IRQ0
17	Bit Test Complete
16	Bit Trigger
15	Reserved
14	RAM Parity Error
13	Transmitter Timeout
12	BC/RT Command Stack Rollover
11	MT Command Stack Rollover
10	MT Data Stack Rollover
9	Handshake Failure
8	BC Retry
7	RT Address Parity Error
6	Time Tag Rollover
5	RT Circular Buffer Rollover
4	RT Subaddress Control Word EOM
3	BC End of Frame
2	Format Error
1	BC Status Set/RT Mode Code/MT Pattern Trigger
0 (LSB)	End of Message

```
S16BIT nResult;

void_DECL MyISR_RT(S16BIT DevNum, U32BIT dwIrqStatus)
{
  printf(“TimeTag Rollover occurred\n”);
}

/* Configure Interrupt Events */
nResult = aceSetIrqConditions(
    0,                            /* LDN */
    TRUE,                         /* Enable Event(s) below */
    ACE_IMRI_TT_ROVER,            /* TT Rollover Event */
    MyISR);                       /* Callback Routine */

if(nResult)
   printf(“aceSetIrqConditions() Error: Code %d\n”, nResult);
```

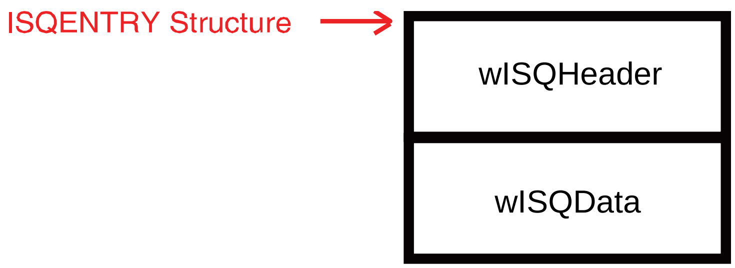
Code Example 3. Setting the Interrupt Callback Routine for Hardware Time Tag Rollover

Note: *The frequency of Time Tag Rollover is dependent on resolution and bit-length of the time tag.*

3.2.4 Interrupt Status Queues

The AceXtreme C SDK includes the capability for generating an Interrupt Status Queue (ISQ). This queue provides a chronological history of interrupt generating conditions and events. The Interrupt Status Queue is 64 words deep, providing the capability to store entries for up to 32 monitored messages. It is available in RT, MT and RTMT modes of operation.

Once enabled, the ISQ logs each event received into a queue using a 2-word (ISQENTRY) structure. Any entries on the queue can be read in FIFO order using the **aceISQRead()** function.



By repeatedly reading the ISQ using the **aceISQRead()** function, the user will have an in-order list of events that have occurred. The wISQHeader word of the ISQENTRY structure will identify the specific event.

Table 15. Interrupt Status Queue Header Values

Bit	Definition for Message Interrupt Event	Definition for Non-Message Interrupt Event
15	Transmitter Timeout	Not Used
14	Illegal Command	Not Used
13	Monitor Data Stack 50% Rollover	Not Used
12	Monitor Data Stack Rollover	Not Used
11	RT Circular Buffer 50% Rollover	Not Used
10	RT Circular Buffer Rollover	Not Used
9	Monitor Command (Descriptor) Stack 50% Rollover	Not Used
8	Monitor Command (Descriptor) Stack Rollover	Not Used
7	RT Command (Descriptor) Stack 50% Rollover	Not Used
6	RT Command (Descriptor) Stack Rollover	Not Used
5	Handshake Fail	Not Used
4	Format Error	Time Tag Rollover
3	Mode Code Interrupt	RT Address Parity Error
2	Subaddress Control Word EOM	Protocol Self-Test Complete
1	End-Of-Message (EOM)	RAM Parity Error
0	“1” For Message Interrupt Event; “0” For Non-Message Interrupt Event	

```
S16BIT nResult;
ISQENTRY sIsqEntry;

/*Enable ISQ */
nResult = aceISQEnable(
    0,                          /* LDN */
    TRUE);                      /* Enable ISQ */

if (nResult)
   printf("aceISQEnable() Error: Code %d\n", nResult);

/* Read an ISQ entry */
nResult = aceISQRead(
    0,                          /* LDN */
    &sIsqEntry);                /* ISQ Entry storage */

if(nResult)
   printf("aceISQRead() Error: Code %d\n", nResult);
else
   printf("Event %04x has occurred!\n",sIsqEntry.wISQHeader);
```

Code Example 4. Reading an Entry from the Interrupt Status Queue (ISQ)

3.2.5 Discrete Digital I/O

DDC's AceXtreme boards include support for discrete digital signals that can be individually programmed as inputs or outputs. The number of discrete digital I/O (DIO) pins on the AceXtreme device varies for each device and it is recommend referring to the device's hardware manual for more information on the digital I/O configuration of a specific device.

The discrete digital I/O can be used for a variety of applications including triggering events, indicating status, or general purpose use. For DDC's AceXtreme Multi-IO boards, the discrete digital I/O may be accessed from either the AceXtreme SDK (BU-69092Sx) or the ARINC 429 Multi-IO SDK (DD-42992Sx).

3.2.5.1 Configuring Discrete Digital

Each discrete line can be set for input or output via the **aceSetDiscDir()** function. The function requires the LDN, the discrete line to modify and the direction to set the discrete line. The direction is configured by passing into the function either **DISC_OUTPUT** (1) for an output or **DISC_INPUT** (0) for an input.

```
S16BIT nResult;
U16BIT wDioChannel_1 = 1;
U16BIT wDioChannel_2 = 2;

/*Set the DIO 1 to an output */
nResult = aceSetDiscDir(
    0,                          /* LDN */
    wDioChannel_1,              /* DIO 1 */
    DISC_OUTPUT);               /* Set to output */

if (nResult)
   printf(“aceSetDiscDir () Error: Code %d\n”, nResult);

/*Set the DIO 2 to an input */
nResult = aceSetDiscDir(
    0,                          /* LDN */
    wDioChannel_2,              /* DIO 2 */
    DISC_INPUT);                /* Set to output */

if (nResult)
   printf(“aceSetDiscDir () Error: Code %d\n”, nResult);
```

Code Example 5. Configuring Discrete lines.

3.2.5.2 Checking Digital I/O line Direction

The direction of a DIO lines can be set for input or output. To retrieve the current state of the DIO line, the function **aceGetDiscDir()** can be used. The function will return either a **DISC_OUTPUT** (1), **DISC_INPUT** (0), or a negative number indicating there was an error in the function call.

```
S16BIT nResult;
U16BIT wDioChannel_1 = 1;
U16BIT wDioChannel_2 = 2;

/* Get the direction of the DIO 1 – should be output */
nResult = aceGetDiscDir (
    0,                      /* LDN */
    wDioChannel_1);         /* DIO 1 */

if(nResult != DISC_OUTPUT)
   printf(“aceGetDiscDir () Error: Code %d\n”, nResult);
else
   printf(“Discrete is set for and output:\n”);


/* Get the direction of the DIO 2 – should be an input */
nResult = aceGetDiscDir (
    0,                      /* LDN */
    wDioChannel_2);         /* DIO 2 */

if(nResult != DISC_INPUT)
   printf(“aceGetDiscDir () Error: Code %d\n”, nResult);
else
   printf(“Discrete is set for and Input\n”);
```

Code Example 6. Reading DIO Channel direction

3.2.5.3 Setting Digital I/O Output

After the DIO line has been configured as an output, the user may configure if the DIO is set **DISC_HIGH** (1), or **DISC_LOW** (0) with the function **aceSetDiscOut()**. By default the DIO line is initialized to DISC_LOW (0).

```
S16BIT nResult;
U16BIT wDioChannel_1 = 1;

/*Set the DIO output to HIGH */
nResult = aceSetDiscOut(
    0,                      /* LDN */
    wDioChannel_1,          /* DIO 1 */
    DISC_HIGH);             /* Set to output to high value */

if (nResult)
   printf("aceSetDiscOut () Error: Code %d\n", nResult);
```

Code Example 7. Setting DIO Line Output Value.

3.2.5.4 Reading Digital I/O Input

After the DIO line has been configured as an input or output, the user may read if the DIO is set **DISC_HIGH** (1), or **DISC_LOW** (0) with the function **aceGetDiscIn()** and **aceGetDisOut()**. By default the DIO line is initialized to DISC_LOW (0).

```
S16BIT nResult;
U16BIT wDioChannel_1 = 1;
U16BIT wDioChannel_2 = 2;

/* Read the DIO line value */
nResult = aceGetDiscOut(
    0,                          /* LDN */
    wDioChannel_1)              /* DIO 1 */

if (nResult < 0)
   printf(“aceGetDiscOut () Error: Code %d\n”, nResult);

else
   printf(“DIO 1 is set to a value %d\n”, nResult);


/*Read the DIO line value */
nResult = aceGetDiscIn(
    0,                          /* LDN */
    wDioChannel_2);             /* DIO 2 */

if (nResult < 0)
   printf(“aceGetDiscIn () Error: Code %d\n”, nResult);

else
   printf(“DIO 1 is set to a value %d\n”, nResult);
```

Code Example 8. Setting DIO Line Output Value.

3.2.5.5 Digital I/O “All” functions

The AceXtreme SDK has two function **aceSetDiscAll()** and **aceGetDiscAll()** which can be used to configure and retrieve all DIO lines within one function call. The DIO functions calls discussed in the previous sections only allow for the configuration of the DIOs lines on an individual basis. When necessary **the aceGetDiscAll()** and **aceSetDiscAll()** may be used to configure all available DIOs based on the mask setting passed into these functions.

```
S16BIT nResult;
U16BIT wDirection = 0xFF;   /* Set DIO 0-7 as output       */
U16BIT wLevel     = 0x55;   /* Set DIO Even as 1, odd as 0 */

/*Set the DIOs to an output and put even DIOs to high */
nResult = aceSetDiscAll(
    0,                      /* LDN */
    wDirection,             /* DIO direction mask */
    wLevel);                /* DIO output levels  */

if (nResult)
   printf("aceSetDiscAll() Error: Code %d\n", nResult);

/* Get the direction and levels of the DIOs */
nResult = aceGetDiscAll (
    0,                      /* LDN */
    &wDirection,            /* DIO direction mask */
    &wLevel);               /* DIO output levels  */

if(nResult < 0)
   printf("aceGetDiscAll () Error: Code %d\n", nResult);
```

Code Example 9. Using aceSetDiscAll() and aceGetDiscAll().

3.2.5.6 DIO Time Tag Recording.

With the BU-67210 Multi-Function AceXtreme card, the time stamp of when a DIO value changes are recorded to a buffer on the card. This feature is currently unique to the BU-67210 card under Windows.

The DIO Time Tag recording feature will store a 48-Bit time stamp based on the configuration specified with **acexDioTtCfg()**. Each time tag entry stored on the buffer is 64-bits (8 Bytes) wide. The upper 16-bits of each entry correspond to the individual discrete signal's rising or falling events, similar to the least significant bit position described in the configuration structure member.

```
S16BIT     nResult;
DIO_TT_CFG sDIOTtCFG;       /* Set DIO 0-7 as output       */


/* User Defined callback function */
Void _DECL DiottIsr(S16BIT DevNum, U32BIT u32Intsts)
{
    /* ISR implementation User Defined callback */
    printf(“An Interrupt has occurred \n”);
}


/* Use Dio 1-8, rising & fall edge */
sDioTtCfg.u32Dio = 0x0000FFFF;

/* Clock Source */
sDioTtCfg.u32TtCfg = TT_RES0_100NS | TT_RO_48BIT;

/* Clock Source */
sDioTtCfg.u32EntCnt = 0;

/*Set the DIOs to an output and put even DIOs to high */
nResult = acexDioTtCfg(
    0,                      /* LDN */
    &sDioTtcfg,             /* DIO direction mask */
    DiottISr);              /* DIO output levels  */

if (nResult)
   printf(“acexDioTtCfg () Error: Code %d\n”, nResult);
```

Code Example 10. Configuring DIO time stamp and ISR.

To turn on or off the time tag recording, the function **acexDioTtCtl()** is used. This function can all be used to reset the timer to zero. To retrieve the stored time tags, the function **acexDioTtRead()** is used.

```
S16BIT     nResult;
U8BIT      *pu8Data;
U32BIT     u32BytesRead = DIO_TT_BUF_LENl

/* Reset the DIO time tag to zero */
nResult = acexDioTtCtl(
    0,                       /* LDN */
    TT_CTL_RESET);           /* Reset DIO Time stamp to zero */

if (nResult)
   printf("acexDioTtCtl () Error: Code %d\n", nResult);


/* Start DIO time tag recording */
nResult = acexDioTtCtl(
    0,                       /* LDN */
    TT_CTL_START);           /* Start the DIO time stamping */

if (nResult)
   printf("acexDioTtCtl () Error: Code %d\n", nResult);


/* Read stored DIO time tag */
nResult = acexDioTtRead(
    0,                       /* LDN */
    pu8Data,                 /* Buffer to store time stamps */
    & u32BytesRead);         /* Number of bytes to read     */

if (nResult)
   printf("acexDioTtRead () Error: Code %d\n", nResult);
```

Code Example 11. Starting Time Tag recording and reading Time tags.

3.2.6 Avionics I/O

DDC's AceXtreme boards include support for Avionic Level Discrete I/O channels that can be individually programmed as inputs or outputs. The number of Avionics Discrete I/O (AIO) pins on the AceXtreme device varies for each device and it is recommend referring to the device's hardware manual for more information on the digital I/O configuration of a specific device.

The Avionics Discrete I/O can be used for a variety of applications including triggering events, indicating status, or general purpose use. For DDC's AceXtreme Multi-IO boards, the Avionics Discrete I/O may be accessed from either the AceXtreme SDK (BU-69092Sx) or the ARINC 429 Multi-IO SDK (DD-42992Sx).

3.2.6.1 Configuring Avionics Discrete I/O

Each Avionics Discrete I/O line can be set for input or output via the **aceSetAioDir()** function. The function requires the LDN, the Avionic Discrete line to modify and the direction to set the discrete line. The direction is configured by passing into the function either **AVIONIC_OUTPUT** (1) for an output or **AVIONIC_INPUT** (0) for an input.

```
S16BIT nResult;
U16BIT wAioChannel_1 = 1;
U16BIT wAioChannel_2 = 2;

/*Set the AIO 1 to an output */
nResult = aceSetAioDir(
    0,                         /* LDN */
    wAioChannel_1,             /* AIO 1 */
    AVIONIC_OUTPUT);           /* Set to output */

if (nResult)
   printf(“aceSetAioDir () Error: Code %d\n”, nResult);

/*Set the AIO 2 to an input */
nResult = aceSetAioDir (
    0,                         /* LDN */
    wAioChannel_2,             /* AIO 2 */
    AVIONIC_INPUT);            /* Set to output */

if (nResult)
   printf(“aceSetAioDir () Error: Code %d\n”, nResult);
```

Code Example 12. Configuring Avionic Discrete lines.

3.2.6.2 Checking Avionic I/O line Direction

The direction of a AIO lines can be set for input or output. To retrieve the current state of the AIO line, the function **aceGetAioDir()** can be used. The function will return either a **AVIONIC_OUTPUT** (1), **AVIONIC_INPUT** (0), or a negative number indicating there was an error in the function call.

```
S16BIT nResult;
U16BIT wAioChannel_1 = 1;
U16BIT wAioChannel_2 = 2;

/* Get the direction of the AIO 1 – should be output */
nResult = aceGetAioDir (
    0,                    /* LDN */
    wAioChannel_1);       /* AIO 1 */

if(nResult != AVIONIC_OUTPUT)
   printf(“aceGetAioDir() Error: Code %d\n”, nResult);
else
   printf(“Avionic I\O is set for and output:\n”);


/* Get the direction of the AIO 2 – should be an input */
nResult = aceGetAioDir (
    0,                    /* LDN */
    wAioChannel_2);       /* AIO 2 */

if(nResult != AVIONIC_INPUT)
   printf(“aceGetAioDir() Error: Code %d\n”, nResult);
else
   printf(“Avionic I\O is set for and Input\n”);
```

Code Example 13. Reading AIO Channel direction

3.2.6.3 Setting Digital I/O Output

After the AIO line has been configured as an output, the user may configure if the AIO is set **AVIONIC_HIGH** (1), or **AVIONIC_LOW** (0) with the function **aceSetAioOut()**. By default the AIO line is initialized to AVIONIC_LOW (0).

```
S16BIT nResult;
U16BIT wAioChannel_1 = 1;

/*Set the AIO output to HIGH */
nResult = aceSetAioOut(
    0,                      /* LDN */
    wAioChannel_1,          /* AIO 1 */
    AVIONIC_HIGH);          /* Set to output to high value */

if (nResult)
   printf("aceSetAioOut () Error: Code %d\n", nResult);
```

Code Example 14. Setting AIO Line Output Value.

3.2.6.4 Reading Digital IO Input

After the AIO line has been configured as an input or output, the user may read if the AIO is set **AVIONIC_HIGH** (1), or **AVIONIC_LOW** (0) with the function **aceGetAioIn()** and **aceGetAioOut()**. By default the AIO line is initialized to AVIONIC_LOW (0).

```
S16BIT nResult;
U16BIT wAioChannel_1 = 1;
U16BIT wAioChannel_2 = 2;

/* Read the AIO line value */
nResult = aceGetAioOut(
    0,                      /* LDN */
    wAioChannel_1)          /* AIO 1 */

if (nResult < 0)
   printf(“aceGetAioOut () Error: Code %d\n”, nResult);

else
   printf(“AIO 1 is set to a value %d\n”, nResult);


/*Read the AIO line value */
nResult = aceGetAioIn(
    0,                      /* LDN */
    wAioChannel_2);         /* AIO 2 */

if (nResult < 0)
   printf(“aceGetAioIn () Error: Code %d\n”, nResult);

else
   printf(“AIO 1 is set to a value %d\n”, nResult);
```

Code Example 15. Setting AIO Line Output Value.

3.2.6.5 Digital I/O “All” functions

The AceXtreme SDK has two function **aceSetAioAll()** and **aceGetAioAll()** which can be used to configure and retrieve all AIO lines within one function call. The AIO functions calls discussed in the previous sections only allow for the configuration of the AIOs lines on an individual basis. When necessary the **aceGetAioAll()** and **aceSetAioAll()** may be used to configure all available AIOs based on the mask setting passed into these functions.

```
S16BIT nResult;
U16BIT wDirection = 0xFF;   /* Set AIO 0-7 as output       */
U16BIT wLevel     = 0x55;   /* Set AIO Even as 1, odd as 0 */

/*Set the AIOs to an output and put even AIOs to high */
nResult = aceSetAioAll(
    0,                      /* LDN */
    wDirection,             /* AIO direction mask */
    wLevel);                /* AIO output levels  */

if (nResult)
   printf("aceSetAioAll() Error: Code %d\n", nResult);

/* Get the direction and levels of the AIOs */
nResult = aceGetAioAll (
    0,                      /* LDN */
    &wDirection,            /* AIO direction mask */
    &wLevel);               /* AIO output levels  */

if(nResult < 0)
   printf("aceGetAioAll () Error: Code %d\n", nResult);
```

Code Example 16. Using aceSetAioAll() and aceGetAioAll().

3.2.7 Triggers

Each 1553 channel can have up to 18 programmable triggers each one uniquely represented by an ID, shown in Table 16.

Triggers are separated into two types: Time/Message (TMT) and General Purpose (GPT). TMT triggers provide message counting and time-related functionality, while GPT triggers are used to identify data matching patterns and synchronize with external equipment. Trigger functionality requires running the 1553 monitor concurrently (see section 3.3.2 for more information on the setup and usage of the Bus Monitor).

Table 16. Trigger IDs

Name / Symbol	Type
ACEX_TRG_ID_TMT1	Time/Message Trigger
ACEX_TRG_ID_TMT2	
ACEX_TRG_ID_GPT1	General Purpose Trigger
ACEX_TRG_ID_GPT2	
ACEX_TRG_ID_GPT3	
ACEX_TRG_ID_GPT4	
ACEX_TRG_ID_GPT5	
ACEX_TRG_ID_GPT6	
ACEX_TRG_ID_GPT7	
ACEX_TRG_ID_GPT8	
ACEX_TRG_ID_GPT9	
ACEX_TRG_ID_GPT10	
ACEX_TRG_ID_GPT11	
ACEX_TRG_ID_GPT12	
ACEX_TRG_ID_GPT13	
ACEX_TRG_ID_GPT14	
ACEX_TRG_ID_GPT15	
ACEX_TRG_ID_GPT16	

Triggers are setup using the following steps, which are to be described in further detail in the rest of this section:

1. Trigger is configured for its Input (that will determine its Arm state) and other operating conditions.
2. The trigger is enabled.
3. Desired Event(s) are enabled.
4. Desired Event(s) are linked to the trigger.
5. (*Optional*) External I/O signals and/or interrupts are configured to operate with the trigger.

3.2.7.1 Trigger Setup

Figure 10 shows the various states a trigger may be in during programming and operation.

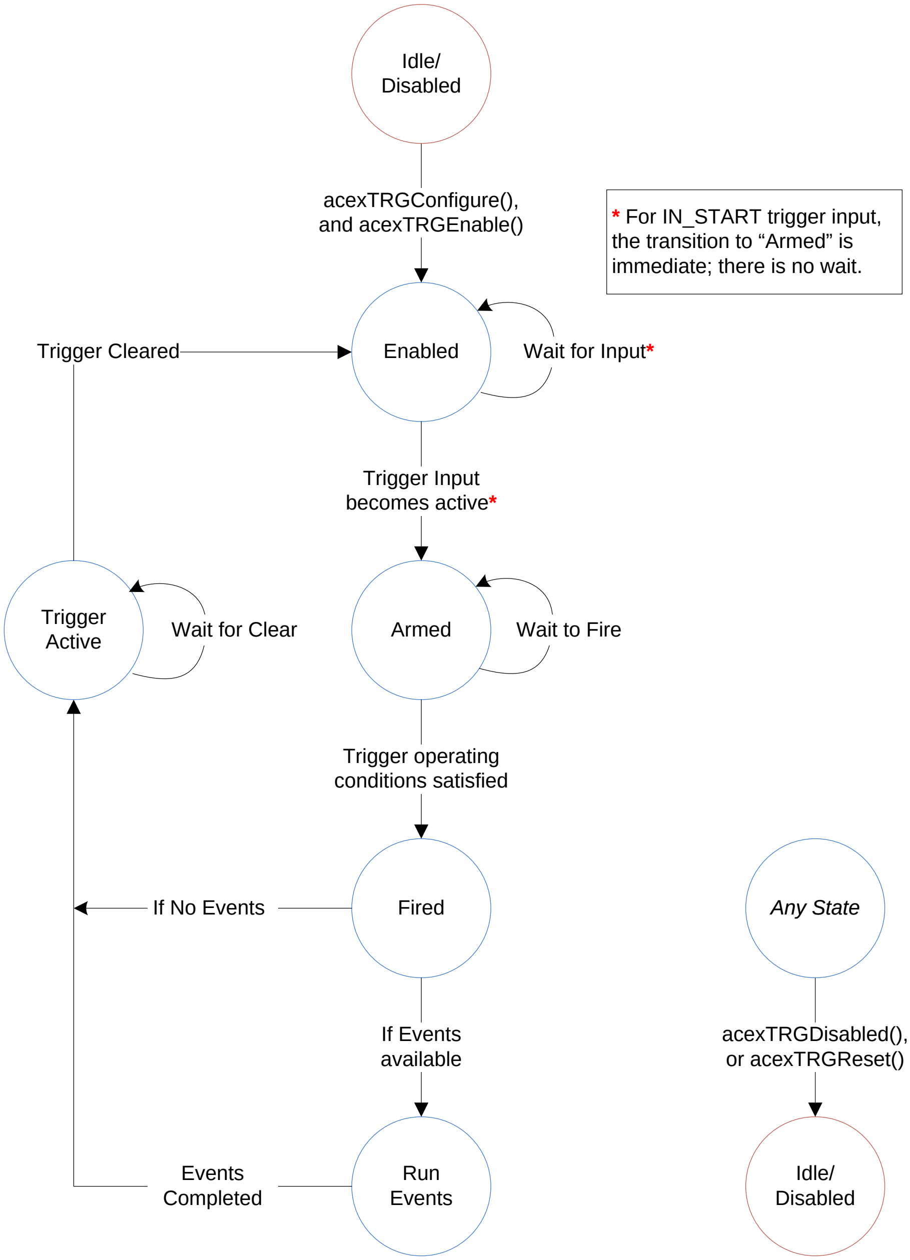


Figure 10. Trigger State Diagram

Triggers are initially configured with the **acexTRGConfigure()** function. The trigger to be configured is specified by its ID from Table 16. The configuration information is provided by the user via the **ACEX_TRG_CONFIG** "C" code structure.

See the section on "Structures" in the ***AceXtreme® C SDK Reference Manual*** (MN-69092SX-003) for detailed information.

```
typedef struct _ACEX_TRG_CONFIG_TMT
{
    U8BIT u8InTmtTrg;
    BOOLEAN bSet;
    BOOLEAN bTimeTrg;
    BOOLEAN bInUs;
    BOOLEAN bClrAuto;
    U16BIT u16TMTCount;
} ACEX_TRG_CONFIG_TMT;
```

Code Example 17. ACEX_TRG_CONFIG_TMT Structure

```
typedef struct _ACEX_TRG_CONFIG_GPT
{
    U8BIT u8InGptTrg;
    BOOLEAN bSet;
    BOOLEAN bClrNotMatched;
    BOOLEAN bClrAuto;
    BOOLEAN bClrNewMsg;
    BOOLEAN bBcCmdEn;
    BOOLEAN bBcDataEn;
    BOOLEAN bRtStatusEn;
    BOOLEAN bRtDataEn;
    BOOLEAN bBswEn;
    BOOLEAN bBusAEn;
    BOOLEAN bBusBEn;
    U8BIT u8DataCnt;
    U8BIT u8TrgCnt;
    U16BIT u16Mask;
    U16BIT u16Data;
} ACEX_TRG_CONFIG_GPT;
```

Code Example 18. ACEX_TRG_CONFIG_GPT Structure

In both the **ACEX_TRG_CONFIG_TMT** and **ACEX_TRG_CONFIG_GPT** structures, the first entry, **u8InTmtTrg** or **u8InGptTrg**, is a value representing the input for the trigger. A trigger's input determines how it will be armed. The available inputs are listed in Table 17.

The primary use of a configurable trigger input is to allow the triggers to be cascaded together, i.e. the output of one trigger will arm the next trigger.

> ***Note**: Each trigger may only have **one** input.*

Table 17. Trigger Inputs (u8InTmtTrg or u8InGptTrg)

Name / Symbol	Description
ACEX_TRG_IN_DISABLE	Trigger will not arm.
ACEX_TRG_IN_START	Trigger will be armed immediately by software when the trigger is enabled.
ACEX_TRG_IN_DISC	This is only valid for GPT. The GPT will be armed by its respective Discrete I/O pin (DIO) transitioning from 0 to 1. When used for triggers, DIO_*n* is hard-wired to ACEX_TRG_ID_GPT*n*, where *n* = 1, 2, … ***Note 1**: The use of DIO pins is limited to the number available on any given DDC hardware.* ***Note 2**: DIO pins are also shared with Trigger outputs and input/output for Intermessage Routines. Hence, not all DIO pins may be available for trigger input use.*
ACEX_TRG_ID_TMT1	Trigger will be armed by the specified Time Message Trigger's output. ***Note**: The user must not configure a trigger to be its own input.*
ACEX_TRG_ID_TMT2	
ACEX_TRG_ID_GPT1	Trigger will be armed by the specified General Purpose Trigger's output. ***Note**: The user must not configure a trigger to be its own input.*
ACEX_TRG_ID_GPT2	
ACEX_TRG_ID_GPT3	
ACEX_TRG_ID_GPT4	
ACEX_TRG_ID_GPT5	
ACEX_TRG_ID_GPT6	
ACEX_TRG_ID_GPT7	
ACEX_TRG_ID_GPT8	
ACEX_TRG_ID_GPT9	
ACEX_TRG_ID_GPT10	
ACEX_TRG_ID_GPT11	
ACEX_TRG_ID_GPT12	
ACEX_TRG_ID_GPT13	
ACEX_TRG_ID_GPT14	
ACEX_TRG_ID_GPT15	
ACEX_TRG_ID_GPT16	

The remaining entries in each of the **ACEX_TRG_CONFIG_TMT** and **ACEX_TRG_CONFIG_GPT** structures define the operating conditions of the trigger, i.e. the conditions that will cause it to fire (i.e. trigger).

Once configured, each trigger in a 1553 channel that is to be used must be enabled by calling the function **acexTRGEnable()**. The trigger is specified by its ID from Table 16. After this, the trigger will be armed according to its input configuration (see Table 17).

Once a trigger is armed, it will fire (i.e. trigger and drives its output signal) when its operating conditions are met. This output signal may be used to drive other triggers, Intermessage Routines (see section 3.3.1.9), or a Discrete I/O pin.

Conversely, calling the function **acexTRGDisable()** will disable the trigger from further use in the given 1553 channel, and clears its output state.

```
S16BIT nResult;
ACEX_TRG_CONFIG sTrgConfig;

sTrgConfig.u.sTmt.u8InTmtTrg = ACEX_TRG_IN_START;
sTrgConfig.u.sTmt.bSet = 0;           /* do not set trigger */
sTrgConfig.u.sTmt.bTimeTrg = 1;       /* choose time trigger */
sTrgConfig.u.sTmt.bInUs = 0;          /* choose ms */
sTrgConfig.u.sTmt.bClrAuto = 0;       /* do not clr */
sTrgConfig.u.sTmt.u16TMTCount = 1000; /* 1000 ms */

/* Configure Trigger */
nResult = acexTRGConfigure(
              0,                    /* LDN */
              ACEX_TRG_ID_TMT1,     /* Trigger ID */
              sTrgConfig);          /* Trigger config */

if(nResult)
    printf(“acexTRGConfigure Error: Code %d\n”,nResult);

/* Enable Trigger */
nResult = acexTRGEnable(0,                  /* LDN */
                        ACEX_TRG_ID_TMT1);  /* Trigger ID */
if(nResult)
    printf(“acexTRGEnable Error: Code %d\n”,nResult);
```

Code Example 19. Selecting an Event

For each channel, all the triggers are reset together by the function **acexTRGReset()**. This resets the trigger configurations, event selections, the trigger enabled/disabled states, and clears the interrupt status and trigger statuses.

3.2.7.2 Event Setup

When a trigger fires, it may be configured to drive other triggers, Intermessage Routines, or a Discrete I/O pin. Additionally, for more flexibility, the trigger may also be programmed to generate up to 12 types of Events, listed in Table 18. To generate more than one event, the values can be logically ORed together when during configuration.

Table 18. Trigger Events

Name / Symbol	Description
ACEX_TRG_EVENT_MTI_MARK_A	Marks a position in the MT capture file.
ACEX_TRG_EVENT_MTI_MARK_B	Marks a position in the MT capture file.
ACEX_TRG_EVENT_MTI_MARK_C	Marks a position in the MT capture file.
ACEX_TRG_EVENT_MTI_MARK_D	Marks a position in the MT capture file.
ACEX_TRG_EVENT_MON_START	Starts MT recording. The MT must previously be configured to wait for a trigger.
ACEX_TRG_EVENT_MON_STOP	Stops MT recording.
ACEX_TRG_EVENT_REPLAY_START	Starts BC Replay. BC Replay must previously be configured to wait for a trigger.
ACEX_TRG_EVENT_REPLAY_STOP	Stops BC Replay.
ACEX_TRG_EVENT_BC_IMR_WAIT	Signals the waiting BC to resume operation. The ACEX_BC_IMR_WAIT_FOR_INPUT_TRIG intermessage routine must first be used to halt the BC.
ACEX_TRG_EVENT_RT_IMR_WAIT	Signals the waiting RT to resume operation. The ACEX_MRT_IMR_WAIT_FOR_INPUT_TRIG intermessage routine must first be used to halt the RT.
ACEX_TRG_EVENT_TIMETAG_LATCH	Captures the current time tag.
ACEX_TRG_EVENT_INTERRUPT	Asserts the ACE_IMR2_BIT_TRIGGER interrupt, causing the user ISR to be called.

Each event that is to be used in a given 1553 channel must be enabled by calling the function **acexTRGEventEnable()**. The event(s) are specified by ORing their values from Table 18.

Conversely, calling the function **acexTRGEventDisable()** will disable the event(s) from further use in the given 1553 channel.

Finally, selected events are linked to a trigger ID by calling the function **acexTRGEventSelect()**.

```
S16BIT nResult;

/* Enable Events */
nResult = acexTRGEventEnable(
              0,                          /* LDN  */
              ACEX_TRG_EVENT_MON_STOP);   /* Events */
if(nResult)
    printf("acexTRGEnable Error: Code %d\n", nResult);

/* Trigger Event Select */
nResult = acexTRGEventSelect(
             0,                           /* LDN */
             ACEX_TRG_ID_TMT1,            /* Trigger ID */
             ACEX_TRG_EVENT_MON_STOP,     /* Events */
             ACEX_TRG_EVENT_TRG);         /* Link to Trigger */
if(nResult)
    printf("acexTRGEventSelect Error: Code %d\n", nResult);
```

Code Example 20. Event Enable and Select

3.2.7.1External Hardware Interactions

The General Purpose Triggers may be configured to interact with external devices via any Discrete or Avionics I/O pins (DIO or AIO) available on the 1553 card. The use of DIO as input to GPT was described previously in Table 17.

DIO may also be configured as the output of GPT. DIO_*n* is hard-wired to ACEX_TRG_ID_GPT*n*, where *n* = 1, 2, …

Note 1*: The use of DIO pins is limited to the number available on any given DDC hardware.*

Note 2*: DIO pins are also shared with Intermessage Routines. Hence, not all DIO pins may be available for trigger use.*

The function **acexSetDiscConfigure()** enables the specified DIO to be used as the external interface for both triggers and intermessage routines. The last parameter of the function is the **ACEX_DISC_CONFIG** structure. It contains parameters that affect the operation of the DIO.

See the section on "Structures" in the ***AceXtreme® C SDK Reference Manual*** (MN-69092SX-003) for detailed information.

```
typedef struct _ACEX_DISC_CONFIG
{
    U16BIT u16Polarity;
    U16BIT u16Control;
    U16BIT u16SelTrgImr;
    BOOLEAN bSSFDisable;
} ACEX_DISC_CONFIG;
```

Code Example 21. ACEX_DISC_CONFIG Structure

```
S16BIT nResult;
S16BIT discrete = DIO_1;           /* DIO_1 connects GPT1 */
ACEX_DISC_CONFIG sDiscConfig;

sDiscConfig.u16Polarity  = ACEX_DISC_ACTIVE_HI;
sDiscConfig.u16Control   = ACEX_DISC_TRGIMR_CTRL;
sDiscConfig.u16SelTrgImr = ACEX_DISC_SEL_TRG;
sDiscConfig.bSSFDisable  = TRUE;

/* Set the Discrete Configuration. */
nResult = acexSetDiscConfigure(0,            /* LDN          */
                               discrete,     /* Discrete I/O */
                               SDiscConfig,  /* Disc Config  */
if(nResult)
    printf("acexSetDiscConfigure Error: Code %d\n", nResult);
```

Code Example 22. Configure a Discrete I/O

For the second entry of the **ACEX_DISC_CONFIG** structure, **u16Control**, the value **ACEX_DISC_TRGIMR_CTRL** allows the GPT (or IMR, in the case of intermessage routines) to drive the DIO output. However, if the value **ACEX_DISC_SW_CTRL** is set, then the host may drive the DIO output by using the Discrete I/O API – **aceSetDiscDir()**, **aceSetDiscAll()**, **aceSetDiscOut**().

All previous Discrete I/O configurations (for both triggers and intermessage routines) in a 1553 channel can be cleared together by calling the function **acexClrDiscConfigure()**.

3.2.7.2 Host Interactions

Any trigger with the **ACEX_TRG_EVENT_INTERRUPT** event enabled will be able to interrupt the host via the **ACE_IMR2_BIT_TRIGGER** (interrupt mask/status bit) user-level interrupt. Please see section 3.2.3 for information on configuring interrupt events and callback routines.

When a trigger interrupt occurs, two registers contain information that is useful to the host. The **Trigger Status Block** contains the current state of every trigger ID and event in the 1553 channel. The **Trigger Interrupt Status Block** shows which trigger ID has interrupted the host since the last interrupt service. The two registers are saved into a buffer each time there is an interrupt. The buffer operates as a circular buffer and has the capacity to save up to 100 interrupt events.

Table 19. Trigger Status Block

Bit	Description
31:29	Reserved
28	Trigger Status TT Latch
27	Trigger Status RT IMR Wait
26	Trigger Status BC IMR Wait
25	Trigger Status Replay Stop
24	Trigger Status Replay Start
23	Trigger Status MT Stop
22	Trigger Status MT Start
21	Trigger Status MT Marker D
20	Trigger Status MT Marker C
19	Trigger Status MT Marker B
18	Trigger Status MT Marker A
17:2	General Purpose Trigger Status [16:1]
1:0	Time/Message Trigger Status [2:1]

Table 20. Trigger Interrupt Status Block

Bit	Description
31:18	Reserved
17:2	General Purpose Trigger Interrupt [16:1]
1:0	Time/Message Trigger Interrupt [2:1]

The host will retrieve the values of the two registers from the buffer by calling the function **acexTRGGetStatus()**.

```
S16BIT nResult;
U32BIT u32TrgStatus = 0;
U32BIT u32IrqStatus = 0;

/* Get trigger and IRQ status. */
nResult = acexTRGGetStatus(0,             /* LDN             */
                           &u32TrgStatus, /* Trigger status  */
                           &u32IrqStatus, /* IRQ Status      */
if(nResult)
    printf(“acexTRGGetStatus Error: Code %d\n”,nResult);
```

Code Example 23. Get Trigger and Interrupt Status

The **ACEX_TRG_EVENT_TIMETAG_LATCH** event causes the 64-bit hardware time tag to be latched. The host reads the last latched value by calling the function **acexTRGGetTimeTag().**

```
S16BIT nResult;
U64BIT u64TimeTag = 0;

/* Get Time tag value. */
nResult = acexTRGGetTimeTag(0,            /* LDN             */
                            &u64TimeTag); /* 64-bit Time Tag */
if(nResult)
    printf(“acexTRGGetTimeTag Error: Code %d\n”,nResult);
```

Code Example 24. Get Latched Time Tag

3.3 1553 Modes of Operation

Based on the user-configurable Mode of Operation (configured via **aceInitialize()**), the AceXtreme C SDK defines API calls to facilitate operation for that specific mode. Each mode has a finite list of functions used to configure, read, and write data to the DDC hardware device.

It is expected that all channels are initialized (see Section 3.1) and assigned a Logical Device Number (LDN) before proceeding.

3.3.1 Bus Controller (ACE_MODE_BC)

The AceXtreme C SDK's Bus Controller mode provides a high degree of flexibility for implementing major and minor frame scheduling and the capability to insert asynchronous messages in the middle or end of a frame. It separates 1553 message data from control/status data for the purpose of implementing advanced buffering and performing bulk data transfers; and implements message retry schemes.

It also includes the capability for automatic bus channel switchover for failed messages as well as for reporting various events to the user host processor via a General Purpose Queue and specific BC events.

The ***AceXtreme*** devices support a combined BC and MT-I mode which allows the user to run both a Bus Controller and an IRIG-106 Chapter 10 monitor on the same channel. To use the combined mode, ACE_MODE_BCMTI must be passed into **aceInitialize()**.

3.3.1.1 Configuration

The AceXtreme C SDK's Bus Controller has numerous configuration options that should be addressed before any data transfers are attempted. Configuration is accomplished via the **aceBCConfigure()** function and is typically called after **aceInitialize()**.

The following options can be "logically OR'ed" into the second parameter of **aceBCConfigure()**.

Table 21. BC Configuration Options

Options	Description
ACE_BC_ASYNC_HMODE	“High-Priority” Asynchronous Messaging Enabled
ACE_BC_ASYNC_LMODE	“Low-Priority” Asynchronous Messaging Enabled
ACE_BC_ASYNC_BOTH	“Both” Asynchronous Messages modes enabled

Table 22. BC Configuration Options for AceXtreme Hardware

Options	Description
ACEX_BC_GPQ_SZ_64,	General Purpose Queue size of 64
ACEX_BC_GPQ_SZ_256,	General Purpose Queue size of 256
ACEX_BC_ASYNCQ_SZ_LP_32,	Asynchronous Low Priority Queue size 32
ACEX_BC_ASYNCQ_SZ_LP_64,	Asynchronous Low Priority Queue size 64
ACEX_BC_ASYNCQ_SZ_LP_128,	Asynchronous Low Priority Queue size 128
ACEX_BC_ASYNCQ_SZ_LP_256,	Asynchronous Low Priority Queue size 256
ACEX_BC_ASYNCQ_SZ_LP_512	Asynchronous Low Priority Queue size 512
ACEX_BC_ASYNCQ_SZ_HP_32,	Asynchronous High Priority Queue size 32
ACEX_BC_ASYNCQ_SZ_HP_64,	Asynchronous High Priority Queue size 64
ACEX_BC_ASYNCQ_SZ_HP_128,	Asynchronous High Priority Queue size 128
ACEX_BC_ASYNCQ_SZ_HP_256,	Asynchronous High Priority Queue size 256
ACEX_BC_ASYNCQ_SZ_HP_512	Asynchronous High Priority Queue size 512
ACEX_BC_RESP_GAP_CHECK	When specified configures a 4µsecond minimum RT response check. If the RT responds earlier than 4 µs, bit 3 of the block status word will be set to ‘1’.
ACEX_BC_BCST_STATUS_CHECK	When this option is specified, the BC will check for status for all broadcast commands. If the status is received before timing out, a word count error will be logged in the block status word.
ACEX_BC_SIMUL_BUS_TX	Simultaneous Bus A and B transmissions for superseding commands.
ACEX_BC_INTERMSG_GAP_TIME	Gap time between two messages (See section 3.3.1.2.2.1.2)
ACEX_BC_TO_ACTIVATE	This Option allows a BC to become active after the DBC switching process. Multiple BCs can be configured, however only one can be activated at a time.

The option **ACEX_BC_INTERMSG_GAP_TIME** gives the user a better accuracy of the gap between messages. Once enabled, all messages will use the gap time, otherwise the time between messages will be measured as time to the next message. Time to the next message is measured from the beginning of the current message to the beginning of the next message.

```
S16BIT nResult;

/* Enable HP Asynchronous Messaging */
nResult = aceBCConfigure(
    0,                          /* LDN */
    ACE_BC_ASYNC_HMODE);    /* Enable HP Async. Messaging */

if(nResult)
    printf(“aceBCConfigure() Error: Code %d\n”, nResult);
```

Code Example 25. Configuring the BC to Support “High-Priority” Asynchronous Messaging

3.3.1.2 Creating BC Objects

As discussed in Section 3.2.1, Bus Controller Mode defines 4 OUID object types: Data Blocks, Message Blocks, Opcodes, and Frames. Each type has a unique operation and a specific purpose, though they all have a specific relation to each other.

3.3.1.2.1 BC Data Blocks

BC Data Blocks are used to store MIL-STD-1553 data words (up to 32 words). Once created, a BC Data Block can be independently read or written to. Typically, a BC Data Block is linked to a BC Message to receive or transmit 1553 data. When a BC Message is sent on the bus, the BC Sequence Engine will place data into or pull data from the linked BC Data Block.

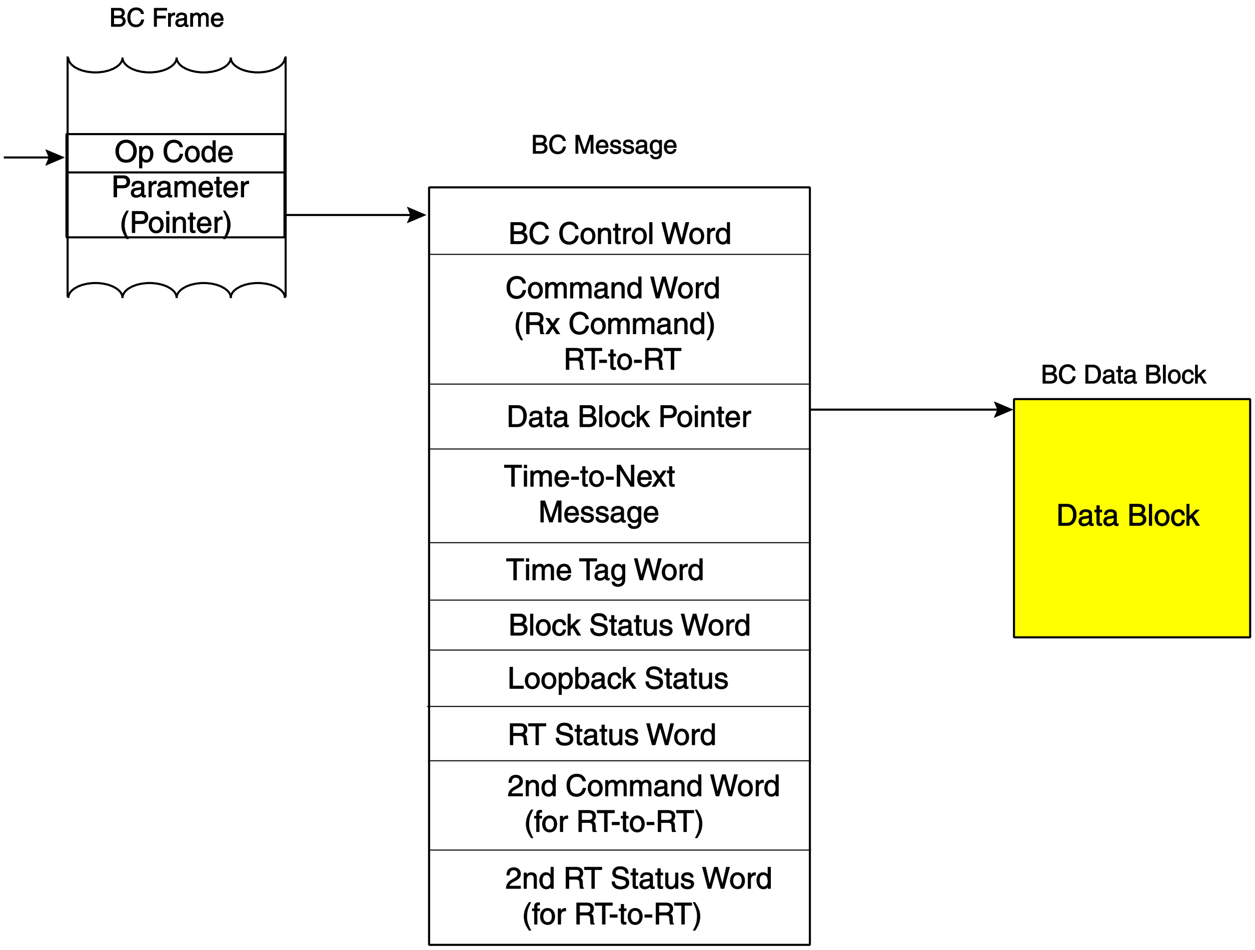


Figure 11. Relationship of BC Data Blocks

BC Data Blocks reside in DDC hardware memory, so they need to be created to guarantee available space. Once created, they can be linked to one or more BC Message Blocks (see 3.3.1.2.2). To create a BC Data Block, use the **aceBCDataBlkCreate()** function.

```
S16BIT nResult;
#define DBLK1 0x0001
U16BIT wBuffer[32] =
{ 0x1111,0x2222,0x3333,0x4444,0x1111,0x2222,0x3333,0x4444,
  0x1111,0x2222,0x3333,0x4444,0x1111,0x2222,0x3333,0x4444,
  0x1111,0x2222,0x3333,0x4444,0x1111,0x2222,0x3333,0x4444,
  0x1111,0x2222,0x3333,0x4444,0x1111,0x2222,0x3333,0x4444  };

/* Create a BC Data Block */
nResult = aceBCDataBlkCreate(
    0,               /* LDN */
    DBLK1,           /* Data Clock OUID to assign */
    32,              /* Data Block Size */
    wBuffer,         /* Initial Data */
    32);             /* Size of Initial Data */

if(nResult)
    printf(“aceBCDataBlkCreate Error: Code %d\n”, nResult);
```

Code Example 26. Creating and Initializing a BC Data Block

3.3.1.2.2 BC Message Blocks

BC Messages Blocks are used to store control and status information about a particular MIL-STD-1553 message. Items such as RT address, RT subaddress, Word Count and others are stored in the BC Message Block to be executed at some point on the 1553 bus.

The AceXtreme C SDK defines two categories of Messages: Synchronous and Asynchronous.

> ***Note:*** *The Message Block does not hold the 1553 data words. Each Message Block should assign the OUID of the desired pre-created BC Data Block Object to hold the associated data words.*

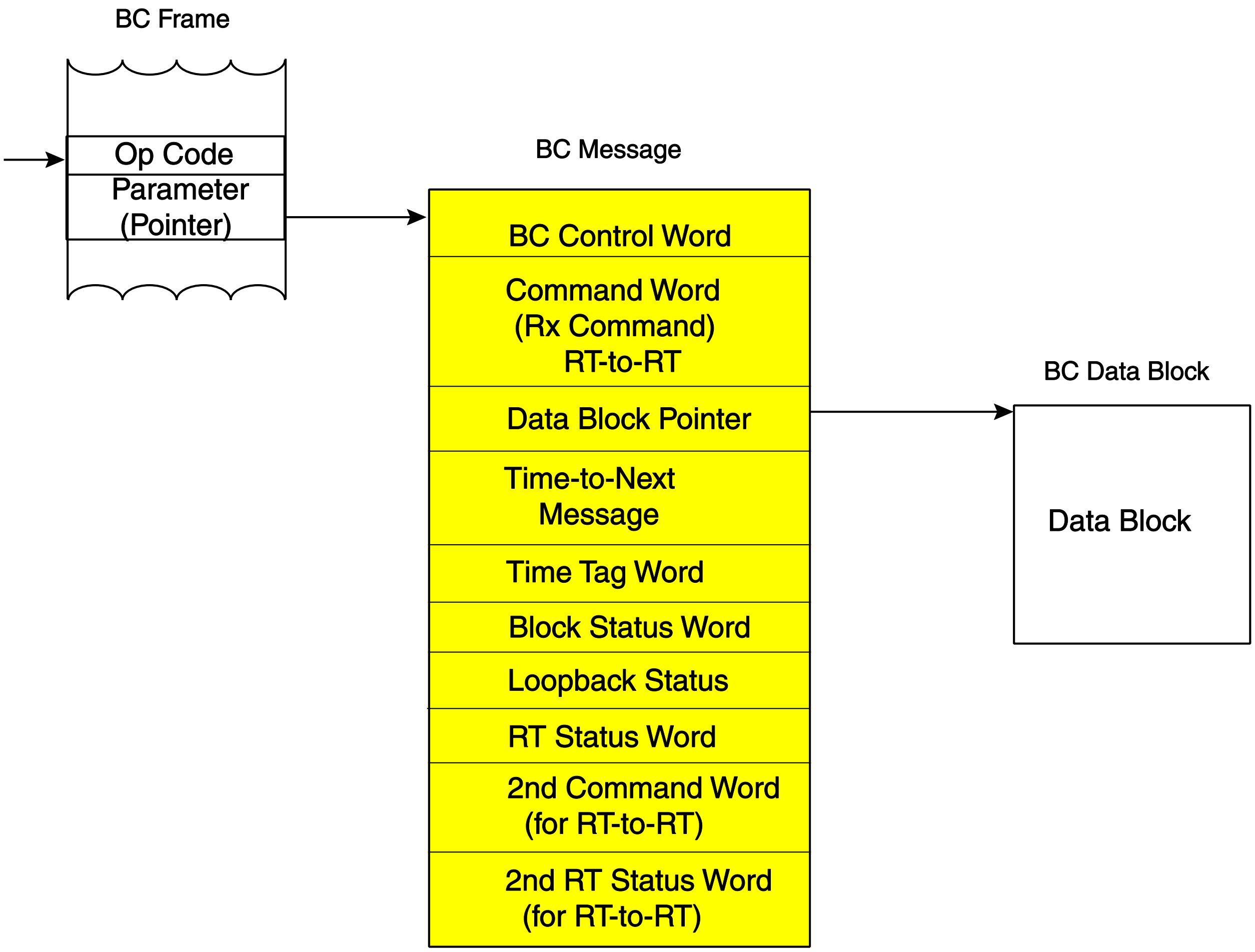


Figure 12. Relationship of BC Message Blocks

3.3.1.2.2.1BC Message Timing

3.3.1.2.2.1.1 BC Time to Next Message

The BC Message Block includes a message timing parameter, which defines the delay time to run the next message. All DDC Hardware supports the Time-To-Next-Message option. This is a delay value calculated from Mid-Parity crossing of the first Command Word crossing to the Mid-Parity crossing of the next Messages Command word (see Figure 13).

This interval is limited to 7us and can be inaccurate based on the RT Status response delay.

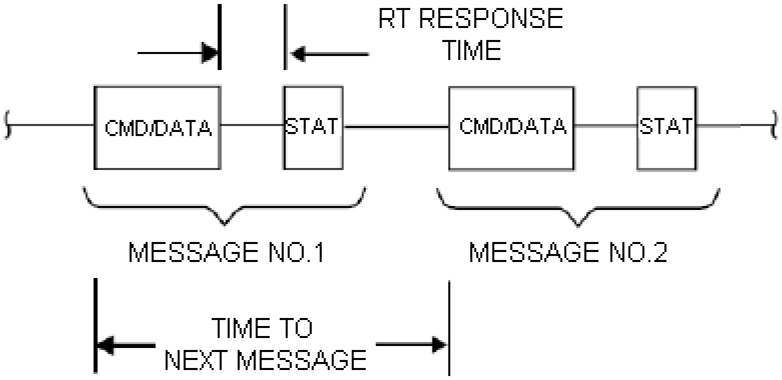


Figure 13. Time To Next Message

3.3.1.2.2.1.2 BC inter-message gap time.

DDC's Multi-Function AceXtreme hardware has the capability of setting the time between messages as either gap time or the time to next message. Inter-Message Gap time is measured from the end of the current message to the beginning of the next message (see Figure 14). The major difference between these two options is the Time to next message field must take into account the RT's response time and the BC's response timeout value. Because of these two dependences from the RT the time to next message field is not as accurate as the Gap time option. To use Gap Time, the parameter **ACEX_BC_INTERMSG_GAP_TIME** must be passed into **aceBCConfigure()**. Doing so, will configure the wMsgGapTime parameter specified in the BC message create functions (**aceBCMsgCreateXXXX()**) to be Gap time and no Time to next message. This is a global setting for all messages.

To have a determinate time interval between messages, the new Multi-Function AceXtreme Architecture allows the user to define this gap time. The inter-message gap time is programmable between 3.5 µseconds and 32,000 µseconds with a resolution of 0.5 µseconds.

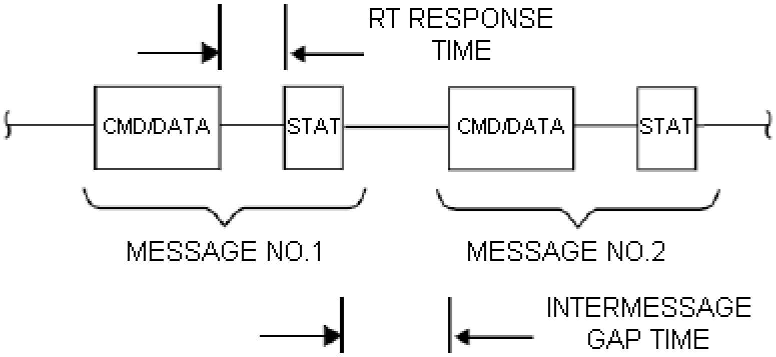


Figure 14. Inter-message Gap Time

3.3.1.2.2.2Synchronous Messages

Synchronous Messages are 1553 messages that will be placed into a Minor or Major Frame (Section 3.3.1.3) for the purpose of being repeated at the scheduled frame rate. For information on sending and receiving Synchronous Messages, see Section 3.3.1.4.2.

The desired 1553 Message Type will determine what API function is used to create the Synchronous Message Block. A Message Block should be created for each unique 1553 Message to be sent on the bus.

Table 23. Synchronous Message Block Type Create Functions

Desired Message Type	Message Block Create Function
BC to RT Message	**aceBCMsgCreateBCtoRT()**
RT to BC Message	**aceBCMsgCreateRTtoBC()**
RT to RT Message	**aceBCMsgCreateRTtoRT()**
Mode Code Command	**aceBCMsgCreateMode()**
BC Broadcast (BC to ALL)	**aceBCMsgCreateBcst()**
RT Broadcast (RT to ALL)	**aceBCMsgCreateBcstRTtoRT()**
Mode Code Broadcast	**aceBCMsgCreateBcstMode()**

```
S16BIT nResult;
#define MSG1 0x0001
#define DBLK1 0x0001

/* Create message block */
nResult =  aceBCMsgCreateBCtoRT(
  0,                     /* Device number */
  MSG1,                  /* Message ID to create */
  DBLK1,                 /* Message will use this data block */
  1,                     /* RT address */
  1,                     /* RT subaddress */
  10,                    /* Word count */
  0,                     /* Next Message Time (TTNM or IMG) */
  ACE_BCCTRL_CHL_A);     /* use chl A options */

if(nResult)
     printf(“aceBCMsgCreateBCtoRT Error: Code %d\n”, nResult);
```

Code Example 27. Creating a BC to RT Synchronous Message Block

3.3.1.2.2.2.1 Synchronous Message Options

The following options can be “logically OR’ed” into the last parameter of any Synchronous Message Block Create (**aceBCMsgCreateXXX**) function.

Table 24. Synchronous Message Options

Message Option	Description (if enabled)
ACE_BCCTRL_1553A	Verifies the validity of the RT response in accordance with MIL-STD-1553A, as opposed to MIL-STD-1553B (default).
ACE_BCCTRL_EOM_IRQ	After this message executes on the 1553 bus, a ACE_IMR1_BC_MSG_EOM device Interrupt Event will be generated. See Section 3.2.3 on how to capture device interrupt events.
ACE_BCCTRL_BCST_MSK	The value of the Broadcast Command Received bit in the received RT Status Word becomes "don't care” in terms of affecting the occurrence of a "Status Set" condition.
ACE_BCCTRL_SELFTST	Inhibits the 1553 transmitter while executing this message for the purpose of offline Self-Test.
ACE_BCCTRL_CHL_A	This message will be sent on Bus A (Primary)
ACE_BCCTRL_CHL_B	This message will be sent on Bus B (Secondary)
ACE_BCCTRL_RETRY_ENA	This message will be retried (per the BC Retry policy) as the result of a response timeout or format error condition.
ACE_BCCTRL_RES_MSK	The value of the 3 Reserved bits in the received RT Status Word becomes "don't care” in terms of affecting the occurrence of a "Status Set" condition.
ACE_BCCTRL_DBC_MSK	This parameter is used to disable the Dynamic Bus Controller interrupt. When the dynamic Bus Controller bit is set in a status word and this masked is used, an interrupt will not be generated.
ACE_BCCTRL_INSTR_MSK	This parameter is used to disable the instrumentation interrupt. When the instrumentation bit is set in a status word and this masked is used, an interrupt will not be generated.
ACE_BCCTRL_TFLG_MSK	The value of the Terminal Flag bit in the received RT Status Word becomes "don't care” in terms of affecting the occurrence of a "Status Set" condition.
ACE_BCCTRL_SSFLG_MSK	The value of the Subsystem Flag bit in the received RT Status Word becomes "don't care” in terms of affecting the occurrence of a "Status Set" condition.
ACE_BCCTRL_SSBSY_MSK	The value of the Busy bit in the received RT Status Word becomes "don't care” in terms of affecting the occurrence of a "Status Set" condition.
ACE_BCCTRL_SREQ_MSK	The value of the Service Request bit in the received RT Status Word becomes "don't care” in terms of affecting the occurrence of a "Status Set" condition.
ACE_BCCTRL_ME_MSK	The value of the Message Error bit in the received RT Status Word becomes "don't care” in terms of affecting the occurrence of a "Status Set" condition.

3.3.1.2.2.3 Asynchronous Messages

Asynchronous Messages are 1553 messages that are not periodic and only need to be sent on the 1553 bus a finite amount of times. This is accomplished by inserting these messages into currently executing synchronous Minor Frame. The AceXtreme C SDK defines two types of Asynchronous Messages; Low-Priority and High-Priority. The method to create the Message Blocks is the same, regardless of type. For information on sending and receiving Asynchronous Messages, see Section 3.3.1.4.2.

Note: *Make sure Asynchronous Messaging is enabled before creating any Message Blocks (See 3.3.1.1)*

Note: *Asynchronous Messaging is designed to work in tandem with Synchronous Frames. It is not recommended to use Asynchronous Messaging as the sole method of sending 1553 BC Traffic.*

The desired 1553 Message Type will determine what API function is used to create the Asynchronous Message Block. A Message Block should be created for each unique 1553 Message to be sent on the bus.

Table 25. Asynchronous Message Block Type Create Functions

Desired Message Type	Message Block Create Function
BC to RT Message	**aceBCAsyncMsgCreateBCtoRT()**
RT to BC Message	**aceBCAsyncMsgCreateRTtoBC()**
RT to RT Message	**aceBCAsyncMsgCreateRTtoRT()**
Mode Code Command	**aceBCAsyncMsgCreateMode()**
BC Broadcast (BC to ALL)	**aceBCAsyncMsgCreateBcst()**
RT Broadcast (RT to ALL)	**aceBCAsyncMsgCreateBcstRTtoRT()**
Mode Code Broadcast	**aceBCAsyncMsgCreateBcstMode()**

```
S16BIT nResult;
#define MSG1 0x0001
#define DBLK1 0x0001
U16BIT wBuffer[32] =
{ 0x1111,0x2222,0x3333,0x4444,0x1111,0x2222,0x3333,0x4444,
  0x1111,0x2222,0x3333,0x4444,0x1111,0x2222,0x3333,0x4444,
  0x1111,0x2222,0x3333,0x4444,0x1111,0x2222,0x3333,0x4444,
  0x1111,0x2222,0x3333,0x4444,0x1111,0x2222,0x3333,0x4444      };

/* Create message block */
nResult =  aceBCAsyncMsgCreateBCtoRT(
   0,                    /* Device number */
   MSG1,                 /* Message ID to create */
   DBLK1,                /* Message will use this data block */
   1,                    /* RT address */
   1,                    /* RT subaddress */
   10,                   /* Word count */
   0,                    /* Default message timer */
   ACE_BCCTRL_CHL_A,     /* use chl A options */
   wBuffer);             /* Load Initial Data */

if(nResult)
    printf("aceBCAsyncMsgCreateBCtoRT Error: Code %d\n", nResult);
```

Code Example 28. Creating a BC to RT Asynchronous Message Block

3.3.1.2.2.3.1 Asynchronous Messages Options

The following options can be “logically OR’ed” into the last parameter of any Asynchronous Message Block Create (**aceBCAsyncMsgCreateXXX**) function.

Table 26. Asynchronous Message Options

Message Option	Description (if enabled)
ACE_BCCTRL_1553A	Verifies the validity of the RT response in accordance with MIL-STD-1553A, as opposed to MIL-STD-1553B (default).
ACE_BCCTRL_EOM_IRQ	After this message executes on the 1553 bus, a ACE_IMR1_BC_MSG_EOM device Interrupt Event will be generated. See Section 3.2.3 on how to capture device interrupt events.
ACE_BCCTRL_BCST_MSK	The value of the Broadcast Command Received bit in the received RT Status Word becomes "don't care" in terms of affecting the occurrence of a "Status Set" condition.
ACE_BCCTRL_SELFTST	Inhibits the 1553 transmitter while executing this message for the purpose of offline Self-Test.
ACE_BCCTRL_CHL_A	This message will be sent on Bus A (Primary)
ACE_BCCTRL_CHL_B	This message will be sent on Bus B (Secondary)
ACE_BCCTRL_RETRY_ENA	This message will be retried (per the BC Retry policy) as the result of a response timeout or format error condition.
ACE_BCCTRL_RES_MSK	The value of the 3 Reserved bits in the received RT Status Word becomes "don't care" in terms of affecting the occurrence of a "Status Set" condition.
ACE_BCCTRL_TFLG_MSK	The value of the Terminal Flag bit in the received RT Status Word becomes "don't care" in terms of affecting the occurrence of a "Status Set" condition.
ACE_BCCTRL_SSFLG_MSK	The value of the Subsystem Flag bit in the received RT Status Word becomes "don't care" in terms of affecting the occurrence of a "Status Set" condition.
ACE_BCCTRL_SSBSY_MSK	The value of the Busy bit in the received RT Status Word becomes "don't care" in terms of affecting the occurrence of a "Status Set" condition.
ACE_BCCTRL_SREQ_MSK	The value of the Service Request bit in the received RT Status Word becomes "don't care" in terms of affecting the occurrence of a "Status Set" condition.
ACE_BCCTRL_ME_MSK	The value of the Message Error bit in the received RT Status Word becomes "don't care" in terms of affecting the occurrence of a "Status Set" condition.

3.3.1.3 Building a Frame

In many 1553 systems, the BC is required to process messages to and from the various RT/subaddresses at a variety of periodicities. For example, some messages may be required to be transmitted at a 5 Hz rate, others at a 10 Hz rate, 20 Hz rate, 100Hz, etc. A common mechanism for supporting varying data rates in a 1553 system is the use of minor and major frames.

3.3.1.3.1 Understanding Minor and Major Frames

The figure below illustrates the concepts of minor and major BC frames. A minor frame has a fixed time duration (e.g., 10 ms or 20 ms), while a major frame is comprised of multiple minor frames. Depending on the periodicity of individual messages, they will either be processed in a single minor frame within the major frame, or in multiple minor frames within the major frame.

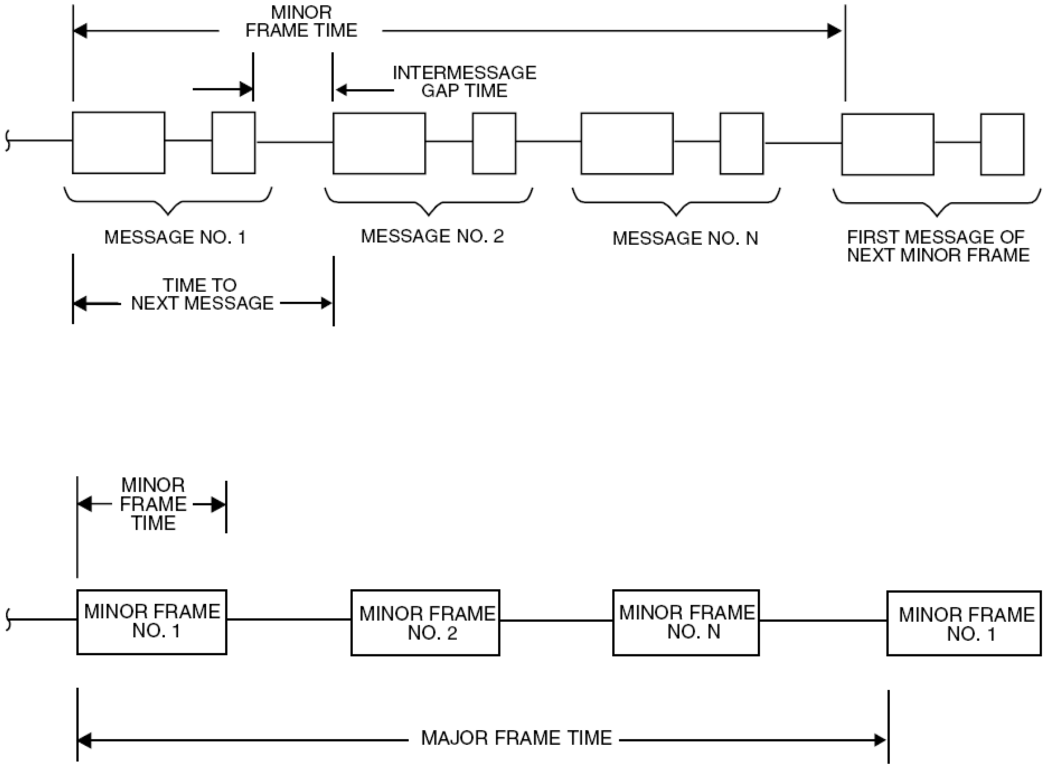


Figure 15. BC Major and Minor Frames

Minor and Major frames may be used as a mechanism for regulating periodic, highly deterministic message traffic. For example, if the BC's minor frame time is 10 ms with a major frame time of one second, it is a relatively simple matter to program specific messages with varying periodicities. For example, a 1 Hz message would appear in one minor frame within a major frame, a 2 Hz message would appear in every 50th minor frame, a 10 Hz message would appear in every tenth minor frame, etc.

MIL-STD-1553B defines inter-message gap time as the time from the mid-parity bit crossing of the last word of one message to the time of the mid-sync crossing of the command word of the subsequent message. As shown in the above figure, inter-message gap time may be controlled by means of the BC's Time-to-Next Message variable (wMsgGapTime) defined in the Message Block (see Section 3.3.1.2.2)

The time-to-next message parameter defines the time from the start of the current message to the start of the subsequent message. The BC's minimum inter-message gap time is approximately 10µs. Therefore, if the programmed time-to-next message is less than the time needed to process the message plus 10µs, the resulting inter-message gap time will be about 10µs.

3.3.1.3.2 BC Opcodes

One of the salient architectural features of the AceXtreme C SDK is the advanced capability for BC message sequence control. The highly autonomous BC operation greatly offloads the operation of the host processor.

The AceXtreme C SDK defines a collection of BC Opcodes, which are used to execute messages, minor and major frames, as well as some advanced conditional control logic. Each Opcode has a unique ability and requires different input parameters. Since Opcodes are one of the BC OUID object types, they need to be created before they can be used in Minor or Major Frames. Some Opcodes have a parameter value, which serves as an input for the Opcode function.

Note: *For detailed information on all BC Opcodes, see the* ***aceBCOpCodeCreate()*** *function definition.*

Table 27. BC Opcode Definitions and Parameter Meanings

Opcode	Parameter	Definition
ACE_OPCODE_XEQ	Message OUID	Execute Message at OUID
ACE_OPCODE_JMP	Jump Offset	Jump within Minor Frame
ACE_OPCODE_CAL	Minor Frame OUID	Call a Minor Frame
ACE_OPCODE_RTN	None	Return from a Minor Frame
ACE_OPCODE_IRQ	BC IRQ to generate	Generate Host Interrupt
ACE_OPCODE_HLT	None	Halt the Bus Controller
ACE_OPCODE_DLY	Time to Delay (ms)	Delay “value” before next Opcode
ACE_OPCODE_WFT	None	Delays until Hardware Frame Timer reaches zero.
ACE_OPCODE_CFT	Value to Compare	Compare “value” to Hardware Frame Timer (100µseconds/LSB)
ACE_OPCODE_FLG	GP Flag to set/clear/toggle	/* Set, clear, toggle 8 GP bits */
ACE_OPCODE_LTT	Time Tag Value (16-bit)	Loads “value” into Hardware Time tag
ACE_OPCODE_LFT	Timer Value	Loads “value” into Hardware Frame Timer (100useconds/LSB)
ACE_OPCODE_SFT	None	Starts the Hardware Frame Timer
ACE_OPCODE_PTT	None	Pushes the current Hardware Time tag onto the General Purpose Queue (GPQ)
ACE_OPCODE_PBS	None	Pushes the Block Status Word (BSW) from the most recent message on to the General Purpose Queue (GPQ)
ACE_OPCODE_PSI	User-Specified Value	Push “value” onto General Purpose Queue (GPQ)
ACE_OPCODE_PSM	DDC Hardware Memory Offset	Pushes data at memory value onto the General Purpose Queue (GPQ)
ACE_OPCODE_WTG	None	Wait for external trigger (BC_EXT) signal to change state
ACE_OPCODE_XQF	Message OUID	Execute and Flip Message at OUID
ACE_OPCODE_IMR	Intermessage Routine (See Table 28)	Generates Bus Controller IMR specified in the parameter field. IMRs may be “logically OR’ed” together, allowing the use of multiple IMRs.

Table 28. Intermessage Routines

Intermessage Routine Macros	Description
ACEX_BC_IMR_IMMEDIATE	Specified Routines will be executed immediately independent of messaging
ACEX_BC_IMR_BREAK	Bus Controller will pause execution after the routines have been executed.
ACEX_BC_IMR_BLK_DATA_SIZE_X	Block Data Size, a data block increment will occur in conjunction with the next message. (see Table 29 on Block Data Size Macros and valid Data block sizes).
ACEX_BC_IMR_SET_DISCRETE_X	Sets discrete output to a logic 1. X is 1 – 4.
ACEX_BC_IMR_RST_DISCRETE_X	Resets discrete output to a logic 0. X is 1 – 4.
ACEX_BC_IMR_WAIT_FOR_INPUT_TRIG	BC operation will be paused until an external BC trigger signal is detected.
ACEX_BC_IMR_NO_RESP_BOTH_BUS	Disables the current RT's transmitter on both buses.
ACEX_BC_IMR_SET_SRQ_IN_STATUS	Indicates the service request bit in the status of the last RT to respond will be set.
ACEX_BC_IMR_RST_SRQ_IN_STATUS	Indicates the service request bit in the status of the last RT to respond will be cleared.
ACEX_BC_IMR_SET_TFG_IN_STATUS	Indicates the terminal flag bit in the status of the last RT to respond will be set.
ACEX_BC_IMR_RST_TFG_IN_STATUS	Indicates the terminal flag bit in the status of the last RT to respond will be cleared.
ACEX_BC_IMR_SET_BSY_IN_STATUS	Indicates the busy bit in the status of the last RT to respond will be set.
ACEX_BC_IMR_RST_BSY_IN_STATUS	Indicates the busy bit in the status of the last RT to respond will be cleared.
ACEX_BC_IMR_RETRY_SAME_ALT_REMAIN_ALT	The next message will be retried on the same bus and then on the alternate bus and remain on the alternated bus, overriding any current message retry settings.
ACEX_BC_IMR_RETRY_ALT_REMAIN_ALT	The next message will be retried and remain on the alternate bus overriding any current message retry settings.
ACEX_BC_IMR_RETRY_ALT	The next message will be retried on the alternate bus overriding any current message retry settings.
ACEX_BC_IMR_RETRY_SAME	The next message will be retried on the same bus overriding any current message retry settings.
ACEX_BC_IMR_EXEC_NEXT_MSG_ONCE	The next message will be executed once and skipped each additional attempt.
ACEX_BC_IMR_SKIP_NEXT_MSG_ONCE	The next message will be skipped once and executed each additional attempt.
ACEX_BC_IMR_SKIP_NEXT_MSG	The next message will always be skipped.

Table 29. Block Data Size

Block Data Size Macros	Description
ACEX_BC_IMR_BLK_DATA_SIZE_64	Data block size of 64 words.
ACEX_BC_IMR_BLK_DATA_SIZE_128	Data block size of 128 words.
ACEX_BC_IMR_BLK_DATA_SIZE_256	Data block size of 256 words.
ACEX_BC_IMR_BLK_DATA_SIZE_512	Data block size of 512 words.
ACEX_BC_IMR_BLK_DATA_SIZE_1024	Data block size of 1K words.
ACEX_BC_IMR_BLK_DATA_SIZE_2048	Data block size of 2K words.
ACEX_BC_IMR_BLK_DATA_SIZE_4096	Data block size of 4K words.
ACEX_BC_IMR_BLK_DATA_SIZE_8192	Data block size of 8K words.
ACEX_BC_IMR_BLK_DATA_SIZE_16384	Data block size of 16K words.
ACEX_BC_IMR_BLK_DATA_SIZE_32768	Data block size of 32K words.

```
S16BIT nResult;
#define OP1 0x0001
#define OP1 0x0001

/* Create opcode to execute MSG1 */
nResult =  aceBCOpCodeCreate(
    0,                       /* IDN */
    OP1,                     /* Opcode OUID to assign */
    ACE_OPCODE_XEQ,          /* Opcode “Execute Msg” */
    ACE_CNDTST_ALWAYS,       /* Conditional Execution? */
    MSG1,                    /* Message OUID to Send */
    0,                       /* Reserved */
    0);                      /* Reserved */

if(nResult)
    printf(“aceBCOpCodeCreate Error: Code %d\n”, nResult);

/* Create opcode to execute MSG2 */
nResult =  aceBCOpCodeCreate(
    0,                       /* LDN */
    OP1,                     /* Opcode OUID to assign */
    ACE_OPCODE_XEQ,          /* Opcode “Execute Msg” */
    ACE_CNDTST_ALWAYS,       /* Conditional Execution? */
    MSG2,                    /* Message OUID to Send */
    0,                       /* Reserved */
    0);                      /* Reserved */

if(nResult)
    printf(“aceBCOpCodeCreate Error: Code %d\n”, nResult);
```

Code Example 29. Creating Two XEQ (Execute Message) Opcodes

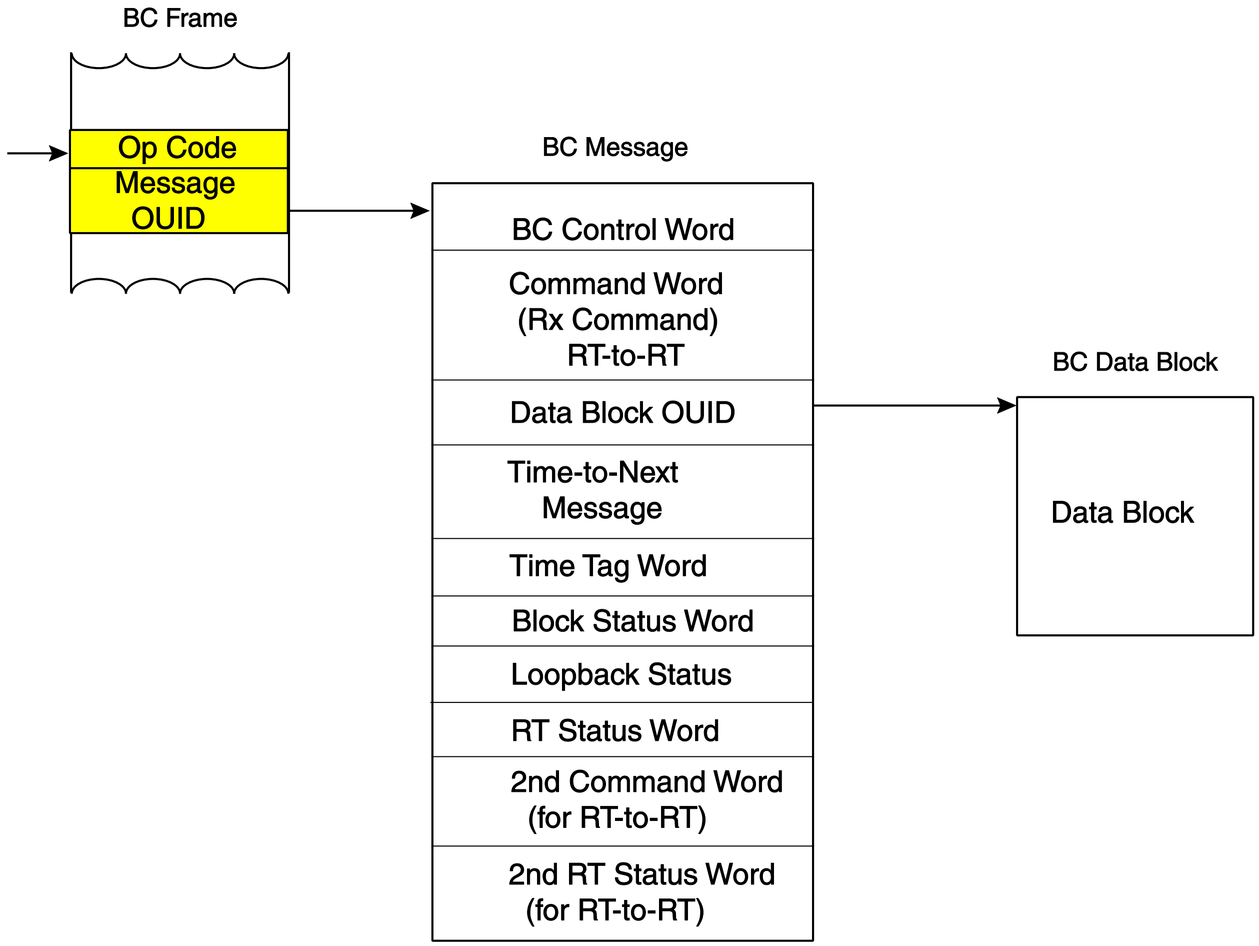


Figure 16. BC Opcode Relationships

3.3.1.3.2.1Conditional Execution

Every BC Opcode has the ability to execute conditionally. The “**wCondition**” variable defines when the Opcode is allowed to execute. For example, an **ACE_OPCODE_XEQ** Opcode can be defined to only execute if there was response from the last accessed RT (**ACE_CNDTST_NO_RES**). This allows certain Opcodes to only execute during specific user conditions. See the **aceBCOpCodeCreate()** function definition for all available execution conditions.

3.3.1.3.2.2Message Execution Opcode

The Execute Message (**ACE_OPCODE_XEQ**) instruction is the primary BC opcode. The “dwParameter1” variable in **aceBCOpCodeCreate()** should reference the OUID of a desired pre-defined Message Block for the desired message to be executed. See Sections 3.3.1.2.2 on how to create Messages. Once the Opcode is created it can be placed into a Minor Frame for processing.

3.3.1.3.2.3 Creating a Minor Frame

Based on the explanation of minor and major frames in Section 3.3.1.3.1, it should be clear that a minor frame is a collection of messages to be executed. Rather than a collection of Messages, the AceXtreme C SDK defines a minor frame as a collection of Opcodes bound in a defined time-window (Minor frame time). Most of the Opcodes in a minor frame will typically be used to execute 1553 messages (**ACE_OPCODE_XEQ**).

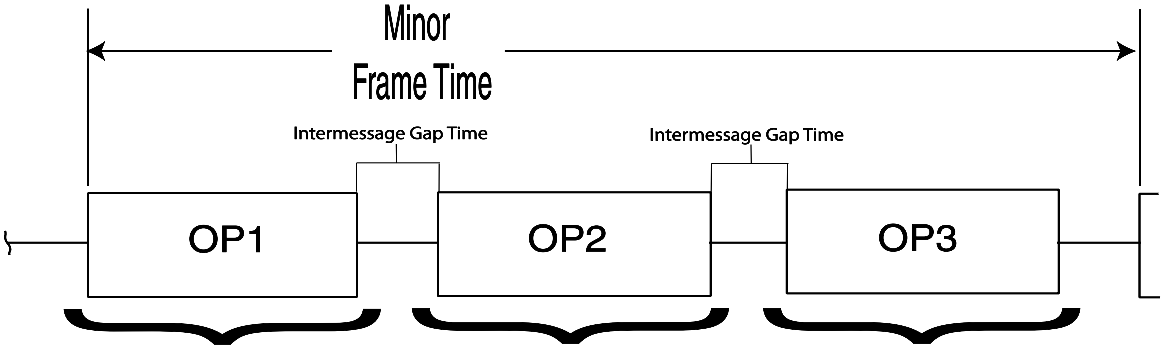


Figure 17. Minor Frame Defined as a Collection of BC Opcodes

Note: *If any Opcodes are Message Execution Opcodes (XEQ), the BC engine will delay for that defined BC Message Blocks' Inter-Message Gap time. See Section 3.3.1.2.2 on how to create BC Message Blocks*

Once all desired Opcodes have been created, a minor frame can be defined. BC Frames are defined using the **aceBCFrameCreate()** for more information. If the time to execute all BC messages exceeds the Minor Frame Time, the Minor Frame time will be increased to accommodate all messages. This occurrence is known as a frame time overrun.

3.3.1.3.2.3.1 32-Bit Expand BC Frame Time Support

The function **acexBCFrameCreate()** allows a Multi-Function AceXtreme card to support a 24-Bit frame time. This function is identical to **aceBCFrameCreate()** with the exception of the frame time parameter is now a unsigned 32-Bit number with the upper 8 bits being reserved.

```
S16BIT nResult;
S16BIT    aOpCodes[2];
 #define MNR1 0x0001

/* Create Minor Frame */
aOpCodes[0] = OP1;
aOpCodes[1] = OP2;
aceBCFrameCreate(
    0,                        /* Device Number */
    FRM1,                     /* OUID */
    ACE_FRAME_MINOR,          /* Frame Type */
    aOpCodes,                 /* Opcodes Array */
    2,                        /* Number of Opcodes */
    1000,                     /* Minor frame time */
    0);                       /* Reserved */

if(nResult)
    printf(“aceBCFrameCreate Error: Code %d\n”, nResult);
```

Code Example 30. Creating a Minor Frame with Two Opcodes

3.3.1.3.2.3.2 User-Defined Interrupt Opcode

The User-Defined Interrupt (**ACE_OPCODE_IRQ**) instruction gives the user “event-driven” control of the Opcode sequencer. Just like all other Opcodes, it can be placed into a Minor Frame and can execute conditionally. When executed, it will generate one of four Opcode Events (depending on its parameter). This event will be marked in the device interrupt callback (see Section 3.2.3) and can be used to trigger data processing. The “dwParameter1” variable in **aceBCOpCodeCreate()** should reference which User-Defined Event should be generated.

3.3.1.3.2.4General Purpose Queue and Flags

In addition to using the User-Defined Interrupt Opcode to synchronize the host to Opcode activity, the AceXtreme C SDK has two more features to aid in event-driven processing. These features are the General Purpose Queue and General Purpose Flags.

3.3.1.3.2.4.1 General Purpose Queue (GPQ)

The General Purpose Queue allows the user to store necessary information needed to process Opcodes. The GPQ is 32 entries deep and can only been filled by the GPQ-based Opcodes (see Section 3.3.1.2.2). Information such as the Hardware Timetag and BC Block Status Word can be placed onto the queue at a specific time during Opcode Execution. By using the GPQ “Push” Opcodes, the user can have a more event-driven interface and have additional information about the current state of the

Minor Frame. Once a “Push” Opcode is created as seen below, it can be added to a Minor Frame just like any other Opcode.

```
S16BIT nResult;
S16BIT aOpCodes[2];
 #define OP2 0x0001

/* Create Subroutine Call Opcode */
nResult =  aceBCOpCodeCreate(
    0,                       /* LDN */
    OP1,                     /* Opcode OUID to assign */
    ACE_OPCODE_PTT,          /* Opcode “Push Timetag” */
    ACE_CNDTST_ALWAYS,       /* Conditional Execution */
    0,                       /* Reserved */
    0,                       /* Reserved */
    0);                      /* Reserved */

 if(nResult)
    printf(“aceBCOpCodeCreate Error: Code %d\n”, nResult);
```

Code Example 31. Creating a Push Timetag to GPQ Opcode

As entries are added to the GPQ via the “Push” Opcodes, the user can read the GPQ directly, and can begin processing data based on the new information. Entries are returned in FIFO order and are automatically removed allow space for new entries. The GPQ can be read via the **aceBCGPQRead()** function.

```
S16BIT nResult;
GPQENTRY sGPQEntry

/* Read an Entry from the GPQ */
nResult =  aceBCGPQRead(
    0,                /* LDN */
    &sGPQEntry);      /* GPQ Entry storage */

if(nResult)
    printf(“aceBCGPQRead Error: Code %d\n”, nResult);
```

Code Example 32. Reading an Entry from the GPQ

3.3.1.3.2.5General Purpose Flags (GPF)

Another method for synchronizing the Autonomous Opcode Engine and the Host is using the seven available General-Purpose Flags. These flags can be set, cleared, and queried directly from the host (via the **aceBCSetGPFState()** function) or from the GPF Opcode (**ACE_OPCODE_FLG**). In addition, any Opcode can execute conditionally based on the state of a GP Flag. See Section 3.3.1.3.2.1 on Condition Execution.

```
S16BIT nResult;
S16BIT aOpCodes[2];
 #define OP2 0x0001

/* Create GP Flag Opcode (Clear GP1) */
nResult =  aceBCOpCodeCreate(
    0,                    /* LDN */
    OP1,                  /* Opcode OUID to assign */
    ACE_OPCODE_FLG,       /* Opcode “Modify GPF Flag” */
    ACE_CNDTST_ALWAYS,    /* Conditional Execution? */
    0x0200,               /* Clear GFP 1 */
    0,                    /* Reserved */
    0);                   /* Reserved */

if(nResult)
    printf(“aceBCOpCodeCreate Error: Code %d\n”, nResult);
```

Code Example 33. Creating a GPF Opcode (Clearing GPF 1)

```
S16BIT nResult;

/* Set GPF1 to active */
nResult =  aceBCSetGPFState(
    0,                  /* LDN */
    ACE_CND_GP1,        /* Modify GPF 1 */
    ACE_GPF_SET);       /* Set GPF 1 */

if(nResult)
    printf(“aceBCSetGPFState Error: Code %d\n”, nResult);
```

Code Example 34. Setting GPF State via Host (Setting GPF 1)

3.3.1.3.2.5.1 General Purpose Flags (GPF) and Conditional Execution

A common use of General Purpose Flags is to control execution of Opcodes.   Any Opcode can be created to conditionally execute only if a specific GP flag is set (or cleared). This can be used to dynamically control execution of Messages (**ACE_OPCODE_XEQ**), Frames (**ACE_OPCODE_CAL**), and any other event that is triggered by an Opcode executing.

```
S16BIT nResult;
#define OP1 0x0001
#define OP2 0x0002

/* Create opcode to execute MSG1 */
nResult =  aceBCOpCodeCreate(
    0,                       /* IDN */
    OP1,                     /* Opcode OUID to assign */
    ACE_OPCODE_XEQ,          /* Opcode “Execute Msg” */
    ACE_CNDTST_GP5_1,        /* Execute if GPF5==1 */
    MSG1,                    /* Message OUID to Send */
    0,                       /* Reserved */
    0);                      /* Reserved */


if(nResult)
    printf(“aceBCOpCodeCreate Error: Code %d\n”, nResult);
```

Code Example 35. Creating an XEQ (Execute Message) Dependant on GPF5

By making the above Opcode dependent on a GP Flag, the user can fully control when that Opcode (and 1553 message) will execute. If GPF5 is set, the Opcode will execute. If it is cleared, the Opcode will not execute and will be treated as a NOOP by the BC Opcode Engine.   GP Flags can be set, cleared, and toggled via the **aceBCSetGPFState()** function (see Section 3.3.1.3.2.5).

> ***Note:*** *By setting ACE_OPCODE_CAL Opcodes to be conditional based on GP Flags, the user can have full control of Minor and Major frames in a similar fashion as above.*

3.3.1.3.2.6Creating a Major Frame

A Major Frame is defined as a collection of Minor Frames (see Section 3.3.1.3.1). The AceXtreme C SDK defines a Major Frame as a collection of Frame Subroutine Call Opcodes. This allows one “Major” Frame to execute all the subsequent minor frames. A specific Opcode (**ACE_OPCODE_CAL**) is used to call a pre-defined minor frame.

3.3.1.3.2.6.1 Frame Subroutine Call Opcode

The Frame Subroutine Call (**ACE_OPCODE_CAL**) instruction is used to execute pre-created Minor Frames. The “dwParameter1” variable in **aceBCOpCodeCreate()** should reference the OUID of a pre-defined Minor Frame for the desired Minor Frame to be executed. See section 3.3.1.3.2.3 on how to create Minor Frames.

```
S16BIT nResult;
S16BIT aOpCodes[2];
 #define OP2 0x0001

/* Create Subroutine Call Opcode */
nResult =  aceBCOpCodeCreate(
    0,                     /* LDN */
    OP1,                   /* Opcode OUID to assign */
    ACE_OPCODE_CAL,        /* Opcode “Call Frame” */
    ACE_CNDTST_ALWAYS,     /* Conditional Execution? */
    MNR1,                  /* OUID of Frame to Call */
    0,                     /* Reserved */
    0);                    /* Reserved */

if(nResult) printf(“aceBCOpCodeCreate Error: Code %d\n”, nResult);
```

Code Example 36. Creating a Subroutine Call Opcode to call Minor Frame “MNR1”

```
S16BIT nResult;
S16BIT    aOpCodes[2];
 #define MJR1 0x0001

/* Create Minor Frame */
aOpCodes[0] = OP1;
aOpCodes[1] = OP2;
nResult = aceBCFrameCreate(
    0,                   /* LDN */
    FRM1,                /* Frame OUID to assign */
    ACE_FRAME_MINOR,     /* Type is Minor Frame */
    aOpCodes,            /* Opcode Array */
    2,                   /* Number of Opcodes in Array */
    1000,                /* Minor Frame Time (100us res) */
    0);                  /* Reserved */

if(nResult)
    printf(“aceBCFrameCreate Error: Code %d\n”, nResult);
```

Code Example 37. Creating a Major Frame (Calling 2 Minor Frames)

3.3.1.3.3 BC Framing/Sequencing Summary

In summary, the highest-level object in BC Sequencing is the Major Frame. Major Frames are a collection of Opcodes (typically **ACE_OPCODE_CAL**) calling Minor Frames. Minor Frames are a collection of Opcodes (typically **ACE_OPCODE_XEQ**) executing Messages. Messages have linked Data Blocks to store 1553 Data.

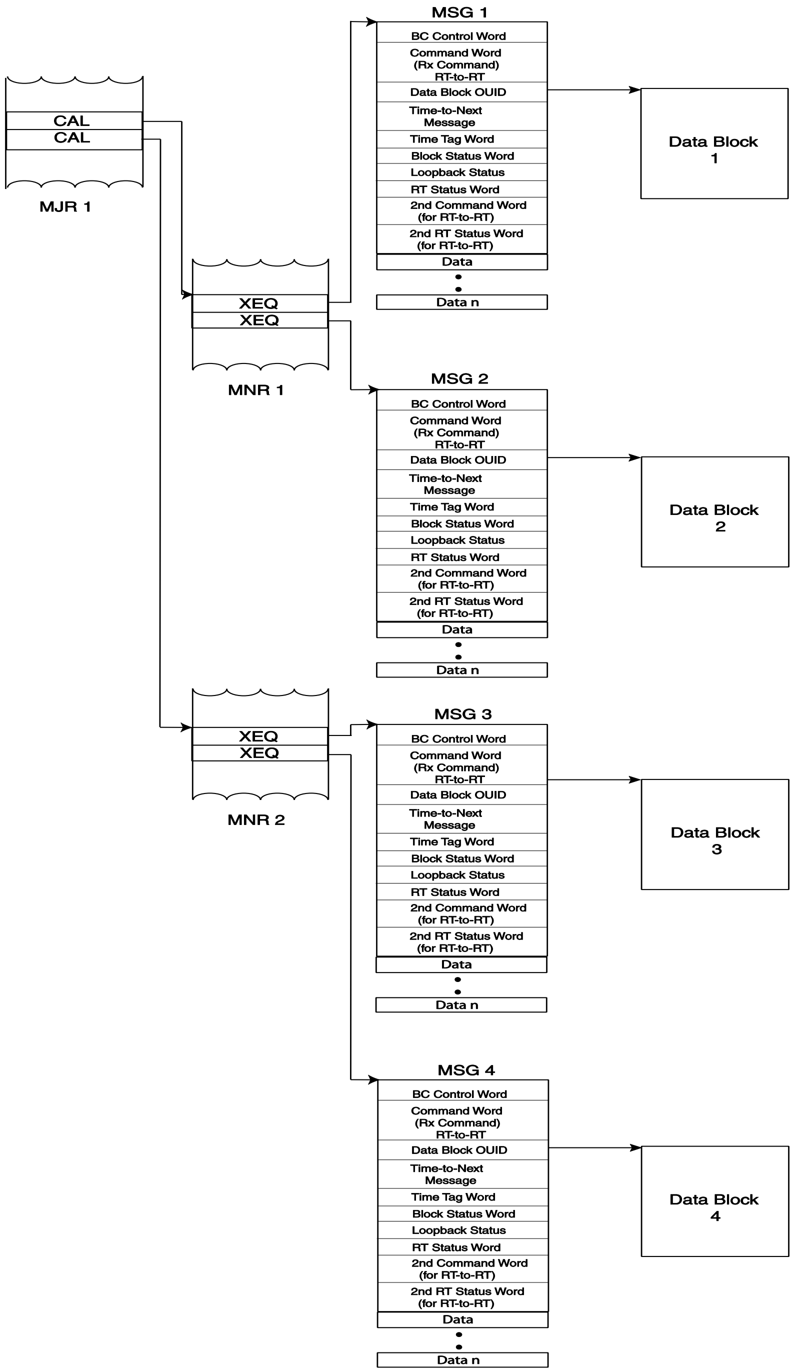


Figure 18. BC Framing/Sequencing Object Relation

3.3.1.4 Activating the BC

Once the Bus Controller is configured and all BC Data Blocks, Message Blocks, Opcodes, Minor and Major Frames have been created, the BC is ready to begin activity on the 1553 bus. By activating the BC, the user-selected Major Frame will start executing its Opcode contents, branching to any Minor Frames and Executing any Synchronous messages defined within.

3.3.1.4.1 Starting and Stopping

The BC is started and stopped dynamically by the user via the **aceBCStart()** and **aceBCStop()** functions. To start the BC, the user must supply the OUID of the Major Frame to execute.

> ***Note:*** *By stopping the Bus Controller via **aceBCStart()**, the BC Engine will complete the currently executing Opcode and will then halt.*

> ***Note:*** *The third parameter of **aceBCStart()** can supply a “Major Frame Execution” value. If this value is “-1”, the Major Frame will continue to run until stopped via **aceBCStop()**. If any other positive value is supplied, the Major Frame will execute X times, where X equals said value.*

```
S16BIT nResult;
#define MAJOR1 0x0001

/* Start the Bus Controller */
nResult =  aceBCStart(
    0,          /* LDN */
    MAJOR1,     /* OUID of Major Frame */
    -1);        /* Run Forever */

if(nResult) printf(“aceBCStart Error: Code %d\n”, nResult);
```

Code Example 38. Starting the Bus Controller (BC)

```
S16BIT nResult;

/* Stop the Bus Controller */
nResult =  aceBCStop(
    0);         /* LDN */

if(nResult) printf(“aceBCStop Error: Code %d\n”, nResult);
```

Code Example 39. Stopping the Bus Controller (BC)

3.3.1.4.2 Sending Asynchronous Messages

Any defined Asynchronous Message can be executed while the BC is running (via **aceBCStart()**). The AceXtreme C SDK defines two methods of sending Asynchronous Messages: High-Priority and Low-Priority. The application need determines which method to use.

> ***Note:*** *Make sure Asynchronous Messaging is enabled before creating any Message Blocks (See section 3.3.1.1)*

3.3.1.4.2.1Sending High-Priority (HP) Asynchronous Messages

Sending an Asynchronous Message High-Priority will execute that message on the 1553 bus immediately following the currently executing message (if a message is in progress). High-Priority Asynchronous Messages can be sent via the **aceBCSendAsyncMsgHP()** function.

```
S16BIT nResult;
#define MSG1 0x0001

/* Send an HP Asynchronous Message */
nResult = aceBCSendAsyncMsgHP(
    0,                  /* LDN */
    MSG1,               /* OUID of Message to send */
    0);                 /* Reserved */

if(nResult) printf(“aceBCSendAsyncMsgHP Error: Code %d\n”,
nResult);
```

Code Example 40. Sending a High-Priority Asynchronous Message

> ***Note:*** *Sending High-Priority Asynchronous Messages increases the Minor Frame time and may potentially cause a frame time overrun.*

3.3.1.4.2.2 Sending Low-Priority Asynchronous Messages

Sending Asynchronous Messages Low-Priority will attempt to send the messages in any gap time left over at the end of the minor frame. Low-Priority Asynchronous Messages are placed in a queue and sent in FIFO order as time allows. When in Low-Priority mode, Asynchronous Messages are automatically added to the end of the LP Queue as they are created.

> ***Note:*** *If more than one Asynchronous Message is on the Low-Priority queue and there is not enough dead time to send all of them, the remaining messages will be sent in the next Minor Frame.*

```
S16BIT nResult;
#define MSG1 0x0001
U16BIT wMsgLeft;

/* Send  LP Asynchronous Messages */
nResult = aceBCSendAsyncMsgLP(
    0,              /* LDN */
    &wMsgLeft,      /* Number of Async. Msgs left to send */
    0);             /* Reserved */

if(nResult)
    printf(“aceBCSendAsyncMsgLP Error: Code %d\n”, nResult);
```

Code Example 41. Sending a Low-Priority Asynchronous Message

> ***Note:*** *The LP Asynchronous Message Queue can be manually cleared via the* ***aceBCEmptyAsyncList ()*** *function.*

3.3.1.5 Consuming Data

The Bus Controller supports three methods of consuming BC Data: Direct Message Block, Direct Data Block, and Host Buffer. Each method has particular advantages based on the user’s application needs.

3.3.1.5.1 Data via Host Buffer

The BC Host Buffer (HBUF) is a circular memory buffer resident on the host that contains the log of all messages sent by the Bus Controller, in the order they appeared on the 1553 bus.

One advantage of using a Host Buffer is that all messages are automatically transferred to the HBUF from the DDC hardware by means of internally configured interrupt events. This ensures that BC data is removed from DDC hardware and placed into the Host Buffer before any data loss can occur.

Another advantage is that the size of the host buffer can be fairly large and can serve as an elasticity buffer for applications that cannot consume data at a high rate.

3.3.1.5.1.1 Installing the Host Buffer

The Host Buffer should be installed before any bus traffic occurs.

> ***Note:*** *The Host Buffer size should typically be 256 times larger that the maximum capacity of the scheduled Message Data (number of executing messages).*

The following equation can be used to calculate the correct Host Buffer size:

HBUFSIZE = [(NUM_MESSSAGES) * 40 words] * 256

```
S16BIT nResult;
#define NUM_MESSAGES = 5

/* Install the BC Host Buffer */
nResult =  aceBCInstallHBuf(
    0,                          /* LDN */
    ((NUM_MESSAGES)*40)*256);   /* Host Buffer Size */

if(nResult)
    printf(“aceBCInstallHBuf Error: Code %d\n”, nResult);
```

Code Example 42. Installing the BC Host Buffer

3.3.1.5.1.2 Reading the Host Buffer

The Host Buffer architecture is designed to automatically remove monitored data from the DDC hardware and place it into the host-allocated Host Buffer. It is the user's responsibility to read entries from the Host Buffer for consumption.

Messages can be read off the Host Buffer in two formats: Raw or Decoded. Depending on which method is used, the messages taken off the Host Buffer will be returned in FIFO order or LIFO order.

3.3.1.5.1.2.1 RAW FORMAT

The Raw Format returns a U16BIT pointer to the binary data. Using this method also allows more than one message to be read off the Host Buffer at one time. Each message is fixed-length of 42 words and uses zero fill words for messages not requiring the word maximum. For example, if two messages have been sent, the binary data will be 84 words deep, with the second message starting at offset 42.

Table 30. BC Host Buffer Raw Format for One Message

Word	**MSB**		**LSB**
Bits	**15**	**7**	**0**
0	BC Control Word		
1	1553 Command Word		
2	Bit [15] EOM	Bits [14:8] Data Length (in Words)	Bits [7:0] 1553 Message Type
3	Time to Next Message		
4	Time Tag Word		
5	Block Status Word		
6	Loopback Word		
7	RT Status Word		
8	2nd Command Word (RT→RT Transfers only)		
9	2nd RT Status Word (RT→RT Transfers only)		
10	Data Word 0		
11	Data Word 1		
n+10	Data Word n		

Messages are read from the Host Buffer in Raw Format using the **aceBCGetHBufMsgsRaw()** function. The function returns up to "*wBufferSize*" words or all messages, whichever is smaller. The "pdwMsgCount" pointer informs the user of the number of messages returned.

Note: *Each message is a fixed length of 42 words.*

```
S16BIT nResult;
U32BIT dwStkLost, dwHBufLost, dwMsgCount;
U16BIT wBuffer[400] = { 0x00000000 };

/* Get Raw Messages from the Host Buffer */
nResult =  aceBCGetHBufMsgsRaw(
    0,              /* LDN */
    wBuffer,        /* Buffer for Data */
    400,            /* Max size of Buffer */
    &dwMsgCount,    /* Number of Msgs placed in Buffer */
    &dwHBufLost);   /* Msgs lost from Hbuf (if any) */

if(nResult)
    printf(“aceBCInstallHBuf Error: Code %d\n”, nResult);
```

Code Example 43. Reading Raw Data From the Host Buffer

3.3.1.5.1.2.2 DECODED FORMAT

The Decoded Format reads one message off the Host Buffer and decodes it into a MSGSTRUCT structure object. In addition, the user can decide whether to read the oldest (next) or latest message and whether or not to remove (purge) the message from the Host Buffer.

A message can be read from the Host Buffer in Decoded Format using the **aceBCGetHBufMsgDecoded()** function. The function returns one message decoded into the “pMsg” MSGSTRUCT variable. The “wMsgLoc” variable is used to define which message to read and whether or not to remove it from the Host Buffer.

Table 31. BC Host Buffer Message Location and Purge Options (wMsgLoc)

Message	Description
ACE_BC_MSGLOC_NEXT_PURGE	Reads next message and takes it off of the host buffer
ACE_BC_MSGLOC_NEXT_NPURGE	Reads next message and leaves it on the host buffer
ACE_BC_MSGLOC_LATEST_PURGE	Reads current message and takes it off of the host buffer
ACE_BC_MSGLOC_LATEST_NPURGE	Reads current message and leaves it on the host buffer

```
/*Global (used for all modes) Message Structure for decoded 1553 msgs */
typedef struct MSGSTRUCT
{
  U16BIT wTYPE;                /* Contains the msg type (see above) */
  U16BIT wBlkSts;             /* Contains the block status word */
  U16BIT wTimeTag;            /* Time Tag of message */
  U16BIT wCmdWrd1;            /* First command word */
  U16BIT wCmdWrd2;            /* Second command word (RT to RT) */
  U16BIT wCmdWrd1Flg;          /* Is command word 1 valid? */
  U16BIT wCmdWrd2Flg;          /* Is command word 2 valid? */
  U16BIT wStsWrd1;            /* First status word */
  U16BIT wStsWrd2;            /* Second status word */
  U16BIT wStsWrd1Flg;          /* Is status word 1 valid? */
  U16BIT wStsWrd2Flg;          /* Is status word 2 valid? */
  U16BIT wWordCount;           /* Number of valid data words */
  U16BIT aDataWrds[32];       /* An array of data words */

/* The following are only applicable in BC mode */
  U16BIT wBCCtrlWrd;           /* Contains the BC control word */
  U16BIT wBCGapTime;           /* Message gap time word */
  U16BIT wBCLoopBack1;        /* First looped back word */
  U16BIT wTimeTag2;           /* wBCLoopBack2 is redefined as TimeTag2 */
  U16BIT wBCLoopback1Flg;      /* Is loopback 1 valid? */
  U16BIT wTimeTag3;           /* wBCLoopBack2Flg is redefined as TimeTag3 */
}MSGSTRUCT;
```

Figure 19. BC Host Buffer MSGSTRUCT Object Definition

```
S16BIT nResult;
U32BIT dwStkLost, dwHBufLost, dwMsgCount;
MSGSTRUCT sMsg;

/* Get a Decoded Message from the Host Buffer */
nResult =  aceBCGetHBufMsgDecoded(
    0,                          /* LDN */
    &sMsg,                      /* Message storage */
    &dwMsgCount,                /* Number of Msgs */
    &dwHBufLost,                /* Hbuf Lost Msgs */
    ACE_BC_MSGLOC_NEXT_PURGE);  /* Read Next & Delete */

if(nResult)
   printf(“aceBCGetHBufMsgDecoded Error: Code %d\n”,nResult);
```

Code Example 44. Reading a Decoded Message from the Host Buffer

3.3.1.5.2 BC Block Status Word

The Block Status Word (BSW) is used to identify the health of the message. The BSW contains information regarding the message, specifying whether the message is in progress or has been completed, what channel the message was processed on, and whether or not there were any errors in the message table. The BC Block status word's bits are defined in

Table 32. BC Block Status Word

Bit	Description	
15 (MSB)	EOM	Set at the completion of a BC message, regardless of whether or not there were any errors in the message.
14	SOM	Set at the start of a BC message and cleared at the end of the message.
13	A/B CHANNEL	This bit will be low if the message was processed on Channel A or high if the message was processed on Channel B
12	ERROR FLAG	If this bit is high, one or more of bits 10, 9, and/or 8 are also set high.
11	STATUS SET	If set, indicates that in one of the lower 11 bits the RT Status Word received from a responding RT contained an unexpected bit value.
10	FORMAT ERROR	If set, indicates the received portion of a message contained one or more violations of the 1553 message validation criteria (sync, encoding, parity, bit count, word count, etc.), or the RT's status word received from a responding RT contained an incorrect RT address field.
9	NO RESPONSE TIMEOUT	If set, indicates that an RT has either not responded or has responded later than the BC No Response Timeout time.
8	LOOP TEST FAIL	A loopback test is performed on the transmitted portion of every message in BC mode. A validity check is performed on the received version of every word transmitted by the BC. In addition, a bit-by-bit comparison is performed on the last word transmitted by the BC for each message. If either the received version of any transmitted word is invalid (sync, encoding, bit count, and/or parity error) and/or the received version of the last word transmitted by the BC does not match the transmitted version, the LOOP TEST FAIL bit will be set.
7	MASKED STATUS SET	It will be set if one or both of the following conditions occur: 1. If one (or more) of the Status Mask bits (14 through 9) in the BC Control Word is logic "0" **and** the corresponding bits are set to logic "1" in the received RT Status Word. In the case of the RESERVED BITS MASK (bit 9) set to logic "0," any or all of the 3 Reserved Status bits being set will result in a MASKED STATUS SET condition; 2. If BROADCAST MASK ENABLED/*XOR* is logic "1" **and** the MASK BROADCAST bit of the message's BC Control Word is logic "0" **and** the BROADCAST COMMAND RECEIVED bit in the received RT Status Word is logic "1".
6	RETRY COUNT 1	Used in conjunction with Bit 5 to indicate the number of retries for a given message. See Table 33 for retry count values.
5	RETRY COUNT 0	Used in conjunction with Bit 6 to indicate the number of retries for a given message. See Table 33 for retry count values.
4	GOOD DATA	This bit is set to logic "1" following completion of a valid (error-free) RT-to-BC transfer, RT-to-RT transfer, or transmit mode code with data message.

Table 32. BC Block Status Word		
Bit	**Description**	
	BLOCK TRANSFER	This bit is set to logic "0" following an invalid message. GOOD DATA BLOCK TRANSFER is always logic "0" following a BC-to-RT transfer, a mode code with data, or a mode code without data.
3	WRONG STATUS ADDRESS / NO GAP	This bit is set if either or both of the following occur: 1. The RT address field of a responding RT does not match the RT address in the Command Word 2. If the GAP CHECK ENABLED bit of Configuration Register #5 is set to logic "1" and a responding RT responds with a response time of less than 4 µs,.
2	WORD COUNT ERROR	If set, indicates that a responding RT did not transmit the correct number of Data Words. Will always be logic "0" following a BC-to-RT transfer, receive mode code message, or transmit mode code without data message.
1	INCORRECT SYNC TYPE	If set, indicates that a responding RT responded with a Data sync in a Status Word and/or a Command/Status sync in a Data Word.
0 (LSB)	INVALID WORD	Indicates an RT responded with one or more words containing one or more of the following error types: sync field error, Manchester encoding error, parity error, and/or bit count error.

Table 33. BC Block Status Word Retry Count

Retry Count 1 (Bit 6)	Retry Count 0 (Bit 5)	Number of Retries
0	0	0
0	1	1
1	0	N/A
1	1	2

3.3.1.5.3 Data via Direct Message

For applications that have strict timing requirements and need quick access to specific 1553 message data, the user can read message status and data directly off of the DDC hardware. This is accomplished using the **aceBCGetMsgFromIDRaw()** and **aceBCGetMsgFromIDDecoded()** functions.

3.3.1.5.3.1Reading a Message Block

The BC Message Block holds all routing information about a particular 1553 message, including a link to the BC Data Block holding the 1553 data words. Every time the Message is executed on the 1553 bus, the BC Message Block is updated with new status information, as well as any detected errors. If a message is to be sent more than once, it is the user’s responsibility to read the Message Block before any new data might overwrite it.

Messages can be read directly from the DDC hardware in two formats: Raw or Decoded. Application needs should influence which method is used. To read a BC Message Block, the user must know the OUID of the desired message.

3.3.1.5.3.1.1 RAW FORMAT

The Raw Format returns a U16BIT pointer to the binary data of one message. Each message is a fixed-length of 42 words and uses zero fill words for messages not requiring the word maximum. In addition, the user can decide to mark the message as read, which will cause any subsequent successful calls to return new data. If no new data is available, an error will be returned.

> ***Note:*** *The **aceBCGetMsgFromIDRaw()** function only returns data (42 words, fixed-length) if that BC Message Block has any new data since the last time it has been accessed.*

Table 34. BC Raw Format for One Message

Word	**MSB**		**LSB**
Bits	**15**	**7**	**0**
0	BC Control Word		
1	1553 Command Word		
2	Bit [15] EOM	Bits [14:8] Data Length (in Words)	Bits [7:0] 1553 Message Type
3	Time to Next Message		
4	Time Tag Word		
5	Block Status Word		
6	Loopback Word		
7	RT Status Word		
8	2nd Command Word (RT→RT Transfers only)		
9	2nd RT Status Word (RT→RT Transfers only)		
10	Data Word 0		
11	Data Word 1		
n+10	Data Word n		

Messages are read directly in Raw Format using the **aceBCGetMsgFromIDRaw()** function. The function returns 42 words if the BC Message Block contains new data. The return value of the function informs the user of the number of messages returned or if an error occurred.

> ***Note:*** *Each message is a fixed length of 42 words.*

```
S16BIT nResult;
U16BIT wBuffer[42] = { 0x00000000 };
#define MSG_MSG_ID 0x0001

/* Get Raw Messages from the Host Buffer */
nResult =  aceBCGetMsgFromIDRaw(
    0,                  /* LDN */
    MY_MSG_ID,          /* OUID of Message to Read */
    wBuffer,            /* Buffer for Data */
    TRUE);              /* Mark Message as “Read” */
if(nResult)
   printf(“aceBCGetMsgFromIDRaw() Error: Code %d\n”,nResult);
```

Code Example 45. Reading Raw Data From a BC Message Block

3.3.1.5.3.1.2 DECODED FORMAT

The Decoded Format reads one message from the DDC hardware and decodes it into a MSGSTRUCT structure object. In addition, the user can decide to mark the message as read, which will cause any subsequent successful calls to return new data. If no new data is available, an error will be returned.

A Message can be directly read from DDC hardware in Decoded Format using the **aceBCGetMsgFromIDDecoded()** function. The function will return one message decoded into the “pMsg” MSGSTRUCT variable.

```
/*Global (used for all modes) Message Structure for decoded 1553 msgs */
typedef struct MSGSTRUCT
{
  U16BIT wTYPE;           /* Contains the msg type (see above) */
  U16BIT wBlkSts;         /* Contains the block status word */
  U16BIT wTimeTag;        /* Time Tag of message */
  U16BIT wCmdWrd1;        /* First command word */
  U16BIT wCmdWrd2;        /* Second command word (RT to RT) */
  U16BIT wCmdWrd1Flg;     /* Is command word 1 valid? */
  U16BIT wCmdWrd2Flg;     /* Is command word 2 valid? */
  U16BIT wStsWrd1;        /* First status word */
  U16BIT wStsWrd2;        /* Second status word */
  U16BIT wStsWrd1Flg;     /* Is status word 1 valid? */
  U16BIT wStsWrd2Flg;     /* Is status word 2 valid? */
  U16BIT wWordCount;      /* Number of valid data words */
  U16BIT aDataWrds[32];   /* An array of data words */

/* The following are only applicable in BC mode */
  U16BIT wBCCtrlWrd;      /* Contains the BC control word */
  U16BIT wBCGapTime;      /* Message gap time word */
  U16BIT wBCLoopBack1;    /* First looped back word */
  U16BIT wTimeTag2;       /* wBCLoopBack2 is redefined as TimeTag2 */
  U16BIT wBCLoopback1Flg;/* Is loopback 1 valid? */
  U16BIT wTimeTag3;       /* wBCLoopBack2Flg is redefined as TimeTag3 */
}MSGSTRUCT;
```

Figure 20. BC Message Block MSGSTRUCT Object Definition

```
S16BIT nResult;
MSGSTRUCT sMsg;
#define MSG_MSG_ID 0x0001

/* Get Decoded Messages from the BC Message Block */
nResult =  aceBCGetMsgFromIDDecoded(
    0,          /* LDN */
    MY_MSG_ID,  /* OUID of Message to Read */
    &sMsg,      /* Message storage */
    TRUE);      /* Mark Msg as “Read” */

if(nResult)
   printf(“aceBCGetMsgFromIDDecoded()Error: %d\n”,nResult);
```

Code Example 46. Reading a Decoded Message from the BC Message Block

3.3.1.5.4 Data via Direct Data Blocks (Read/Write)

In addition to reading BC Data directly and via the Host Buffer, any defined BC Data Block can be read to or written to asynchronously by the user via the **aceBCDataBlkRead()** and **aceBCDataBlkWrite()** functions. See Section 3.3.1.2.1 on creating BC Data Blocks.

Note: *The “wBufferSize” variable should not exceed 32 words.*

```
S16BIT nResult;
U16BIT wData[32];
#define BCDBLK1 0x0001 /* BC Data Block */

/* Read a BC Data Block */
nResult =  aceBCDataBlkRead(
    0,          /* LDN */
    BCDBLK1,    /* OUID of Data Block to Read */
    wData,      /* Data Storage */
    32,         /* Number of words to read */
    0);         /* Read offset in Words */

if(nResult)
    printf(“aceBCDataBlkRead Error: Code %d\n”, nResult);
```

Code Example 47. Reading a BC Data Block

3.3.1.6 Interrupt Events

Some applications may benefit from event notifications regarding the state of the Bus Controller. The following events directly relate to the Bus Controller (BC) mode of operation. For information on how to configure events and callbacks, see Section 3.2.3.

Table 35. BC Interrupt Event Options

Event	Description
ACE_IMR1_EOM (Bit 0)	Enable End-of-Message Event
ACE_IMR1_BC_STATUS_SET (Bit 1)	Enable RT Status Word Error Event
ACE_IMR1_BC_MSG_EOM (Bit 4)	Enable Selective BC End-Of-Message Event
ACE_IMR1_TT_ROVER (Bit 6)	Indicates the Hardware Timetag has rolled over
ACE_IMR1_BC_RETRY (Bit 8)	Indicates that a BC Message has been retried
ACE_IMR1_BCRT_TX_TIMEOUT (Bit 13)	Indicates a BC Message has timed-out
ACE_IMR2_BC_UIRQ0	Indicates User-Defined Opcode Event #0 has occurred.
ACE_IMR2_BC_UIRQ1	Indicates User-Defined Opcode Event #1 has occurred.
ACE_IMR2_BC_UIRQ2	Indicates User-Defined Opcode Event #2 has occurred.
ACE_IMR2_BC_UIRQ3	Indicates a Minor Frame has completed (INTERNAL)

3.3.1.7 Dynamic Bus Controller

Dynamic Bus Control (DBC) is provided to allow the Bus Controller a mechanism to offer a potential bus controller, control of the MIL-STD-1553 data bus. Control is offered by the Bus Controller by sending the mode code Dynamic Bus Controller (00000) to an active RT on the bus. The RT will respond with the dynamic bus controller bit set in its status word if the RT is accepting control over the data bus.

DBC can be enabled to a BC and any RT in a channel. To enable BC with DBC, an optional RT address can be assigned to the BC. The RT address assigned to the BC will activate when the BC relinquishes control of the data bus. The BC will not activate as an RT after it is deactivated when an RT address is not assigned to the BC.

The RT, that BC is to activate, does not have to be inactive. If the RT is already active, BC does nothing but deactivate itself.

Because only one active BC is allowed in a bus, other potential Bus Controllers must thus be configured as inactive. An inactive BC can be activated only after a RT in the same channel accepts DBC request.

When an RT accepts control of the data bus via the Dynamic bus controller mode code, a delay value must be specified in order for the RT to deactivate, and configure itself as the bus controller. This delay value is specified as the “hold-off time” in the

AceXtreme C SDK. Once the hold-off time expires, the newly activated Bus Controller will start issuing commands on the bus. The hold-off time can be specified for each RT with each RT having its own unique hold-off time.

3.3.1.7.1 Enabling /Disable Dynamic Bus Controller

Dynamic Bus Controller (DBC) support can be enabled with the function **acexBCDbcEnable()**. This function requires the LDN.

```
S16BIT nResult;

/* Enable Bus Controller Dynamic BC.. */
nResult = acexBCDbcEnable(0);          /* LDN */
if(nResult)
    printf(“acexBCDbcEnable Error: Code %d\n”,nResult);
```

Code Example 48. Enable/Disbale DBC Support

The AceXtreme C SDK has a function to disable Dynamic Bus Controller called **acexBCDbcDisable()**. This function requires the LDN of the device to disable the DBC.

3.3.1.7.2 Inactive BC

In order for a Multi-Function AceXtreme device in MRT mode to accept control of the data bus, the RT must have an inactive BC configured. An inactive BC, allows the user to configure messages, and frames while leaving the BC inactive when the card is put into the Run state. To create an inactive BC on the card in MRT mode, the **ACEX_BC_INACTIVE_BC** option must be specified in **aceBCConfigure()**.

3.3.1.7.3 Inactive RT

When the Bus Controller is running and has an RT address assigned to it, the RT address maybe inactive. To create an inactive RT for the Bus Controller, the function **acexMRTEnableRT()** must be used and the parameter **ACE_RT_OPT_INACTIVE** must be used.

3.3.1.8 1553 Traffic Replay

The AceXtreme Multi-Function boards support Bus Replay of recorded data. To use Replay the boards can be initialized by passing **ACE_MODE_ALL** into a call to **aceInitialize()**. To enable replay, the functions **acexBCConfigureReplay()**, **acexBCStartReplay()**,and **acexBCGetStatusReplay()** can be used after a call to

aceInitialize(). Monitor functionality can also be used by configuring the MT-I monitor with the **aceMTIConfigure()** function and utilizing the other MT-I function calls. For more on MT-I mode see section 3.3.2.

3.3.1.8.1 Replay Configuration

To configure the replay options on an AceXtreme Multi-Function board, the function **acexBCConfigureReplay()** can be used. The default replay behavior is to enable BC command replay, all RTs will be emulated for Replay, and unknown errors will be replayed. These options can be changed with the parameters passed into **acexBCConfigureReplay()**.

The function **acexBCConfigureReplay()** allows the user to specify if the RT address is being replayed by the AceXtreme Multi-Function card or is an RT external to the card. RTs can be emulated by passing in a bit pattern as the second parameter to **acexBCConfigureReplay()**. A value of 0 will enable the RT for emulation on the AceXtreme Multi-Function card, while a 1 will turn off replay and allow an external device to control the RT address. If RT 5 was desired to be an external RT while the other RT address were to be emulated on the AceXtreme card and value of 0x00000010 would be passed into the second parameter. The other parameters in **acexBCConfigureReplay()** allow the user to select if unknown messages, and errors (when AES is enabled) should be replayed on the bus, along with the Time Tag resolution.

The user may specify a channel ID to replay by using the u16ChannelID parameter if **ACEX_BC_REPLAY_OPT_CHAN_ID_ENA** is specified in the options field of **acexBCConfigureReplay()**. The last parameter passed into **acexBCConfigureReplay()** are the Replay options. The replay options can be logical OR’ed together and are used to enable replaying a specific channel ID, and to use triggers to start or stop replay.

```
S16BIT nResult;

/* Configure BC/RT replay mode */
nResult =  acexBCConfigureReplay(
    0,              /* LDN */
    0x0,            /* bitmask of RTs to disable during replay */
    FALSE,          /* Enable the replay of unknown messages   */
    FALSE,          /* Disable replay of BC commands (unused)  */
    FALSE,          /* Enable the Replay of 1553 errors        */
    ACE_TT_1US,     /* Time Tag Resolution set to 1 µsec       */
    0,              /* Ch ID to Replay, Ignored by default     */
    0);             /* Options                                 */

if(nResult)
   printf(“acexBCConfigureReplay Error: Code %d\n”, nResult);
```

Code Example 49. Configuring Replay

3.3.1.8.2 Start/Stop Replay

After configuring the AceXtreme Multi-Function device for Replay with the function **acexBCConfigureReplay()**, a call to **acexBCStartReplay()** will begin transmission of recorded messages. The function **acexBCStartReplay()** requires the logical device number of the device, a chapter 10 file, the channel ID and how many times to replay the chapter 10 file. Please see section 3.3.2.7 for more information on creating and working with Chapter 10 files.

```
S16BIT nResult;
S32BIT wLoopCount = 1;
U16BIT wChannelID = 10;

/* Start Replaying Replay.ch10 file*/
nResult =  acexBCStartReplay(
    0,                     /* LDN  */
    “Replay.ch10”,         /* Path/filename to replay file */
    wChannelId,            /* Chapter 10 channel to replay */
    wLoopCount);           /* Number of times to replay file */

if(nResult)
   printf(“acexBCStartReplay Error: Code %d\n”, nResult);
```

Code Example 50. Starting Replay

The function **aceBCStop()** can be used to stop replay activity. There are functions to pause and restart replay activity, they are **acexBCPause()**, and **acexBCContinue()**.

3.3.1.8.3 Pause / Continue Replay

Pausing replay with the function **acexBCPause()** will temporarily pause all replay activity until **acexBCContinue()** is called. Once **acexBCContinue()** is called, replay will activity will begin with the next message in the Chapter 10 replay file.

```
S16BIT nResult = 0;
S16BIT DevNum  = 0;

/* Pause Replay */
nResult =  acexBCPause(DevNum);

if(nResult)
   printf(“acexBCPause Error: Code %d\n”, nResult);

sleep(100);

/* Restart or Continue replay */
nResult =  acexBCContinue(DevNum);

if(nResult)
     printf(“acexBCContinue Error: Code %d\n”, nResult);
```

Code Example 51. Pausing / Continue Replay

3.3.1.8.4 Replay Activity Status

The status of the replay engine on a AceXtreme Multi-Function card can be retrieved by calling **acexBCGetStatusReplay()**. The function will return the following states **ACEX_BC_REPLAY_STATUS_RUN** if the Replay is currently running, **ACEX_BC_REPLAY_STATUS_PAUSE** if Replay has been paused with the **acexBCPause()** function call, or **ACEX_BC_REPLAY_STATUS_STOP** when the Replay file has reached the end of file and the number of times to repeat.

```
S16BIT nResult;
U32BIT dwStatus;

/* Get Replay status */
nResult =  acexBCGetStatusReplay(
    0,              /* LDN */
    &dwStatus);     /* Status of Replay, Busy, idle, or stopped */

 if(nResult)
    printf("acexBCGetStatusReplay Error: Code %d\n",nResult);
```

Code Example 52. Retrieve Replay Status

3.3.1.9 BC Intermessage Routines

The Multi-Function AceXtreme boards have support for Intermessage routines (IMRs). IMRs are a set of tasks executed in real-time after any configured 1553 message. Multiple IMRS can be grouped together for one message, however caution must be taken in order to avoid conflicts between the IMRS and sufficient intermessage gap time must be available for proper operation. IMRs are used by using the opcode **ACE_OPCODE_IMR** and the frame type **OTHER.** The usage of IMRs will be described in Section 0.

3.3.1.9.1 BC Intermessage Routine Types

Intermessage routines can be categorized into several groups, IMRs used for message response, IMRs used for discrete and triggers, IMRs used for retry schemes, and IMR control of message execution. There is one other IMR called **ACEX_BC_IMR_IMMEDIATE** which is used when the IMR is to be executed immediately, regardless of messaging.

Table 36. Intermessage Routines

Intermessage Routine	Description
ACEX_BC_IMR_IMMEDIATE	Specified Routines will be executed Immediately independent of messaging
Discretes and Triggers	
ACEX_BC_IMR_SET_DISCRETE_X	Sets discrete output to a logic 1. X is 1 – 4.
ACEX_BC_IMR_RST_DISCRETE_X	Resets discrete output to a logic 0. X is 1 – 4.
ACEX_BC_IMR_WAIT_FOR_INPUT_TRIG	BC operation will be paused until an external BC trigger signal is detected.
Message Response	
ACEX_BC_IMR_NO_RESP_BOTH_BUS	Disables the current RT's transmitter on both buses.

Table 36. Intermessage Routines

Intermessage Routine	Description
ACEX_BC_IMR_SET_SRQ_IN_STATUS	Indicates the service request bit in the status of the last RT to respond will be set.
ACEX_BC_IMR_RST_SRQ_IN_STATUS	Indicates the service request bit in the status of the last RT to respond will be cleared.
ACEX_BC_IMR_SET_TFG_IN_STATUS	Indicates the terminal flag bit in the status of the last RT to respond will be set.
ACEX_BC_IMR_RST_TFG_IN_STATUS	Indicates the terminal flag bit in the status of the last RT to respond will be cleared.
ACEX_BC_IMR_SET_BSY_IN_STATUS	Busy bit in the status of the last RT to respond will be set.
ACEX_BC_IMR_RST_BSY_IN_STATUS	Indicates the busy bit in the status of the last RT to respond will be cleared.
BC Retry Schemes	
ACEX_BC_IMR_RETRY_SAME_ALT_REMAIN_ALT	The next message will be retried on the same bus and then on the alternate bus and remain on the alternated bus, overriding any current message retry settings.
ACEX_BC_IMR_RETRY_ALT_REMAIN_ALT	The next message will be retried and remain on the alternate bus overriding any current message retry settings.
ACEX_BC_IMR_RETRY_ALT	The next message will be retried on the alternate bus overriding any current message retry settings.
ACEX_BC_IMR_RETRY_SAME	The next message will be retried on the same bus overriding any current message retry settings.
BC Message Execution	
ACEX_BC_IMR_EXEC_NEXT_MSG_ONCE	The next message will be executed once and skipped each additional attempt.
ACEX_BC_IMR_SKIP_NEXT_MSG_ONCE	The next message will be skipped once and executed each additional attempt.
ACEX_BC_IMR_SKIP_NEXT_MSG	The next message will always be skipped.
ACEX_BC_IMR_BREAK	Bus Controller will pause execution after the routines have been executed.
Data Blocks Size	
ACEX_BC_IMR_BLK_DATA_SIZE_X	Block Data Size, a data block increment will occur in conjunction with the next message. Valid Block sizes are 64, 128, 256, 512, 1K, 2K, 4K, 8K, 16K and 32K words.

The IMRS listed in Table 39 may be “Logically OR’ed” together to form multiple intermessage actions with the exception of the **ACEX_BC_IMR_SKIP_NEXT_MSG_ONCE**, **ACEX_BC_IMR_SKIP_NEXT_MSG**, **ACEX_BC_IMR_EXEC_NEXT_MSG_ONCE** and the Retry IMRS.

3.3.1.9.2 Usage

In order to use IMRs in BC mode, the opcode **ACE_OPCODE_IMR** must be used with the IMRs listed in Table 39. The opcode for the **ACE_OPCODE_IMR** must appear in the frame before an opcode linked to a message in order for the opcode to run once the message has completed.

It is recommended to use the “OTHER” frame in constructing a message that will use IMRS. The “Other” frame will be a subset of the minor frame which contains the IMR, a message and a call to return from the “Other” frame. The follow of frames, from Major to Minor to Other can be seen in Figure 21.

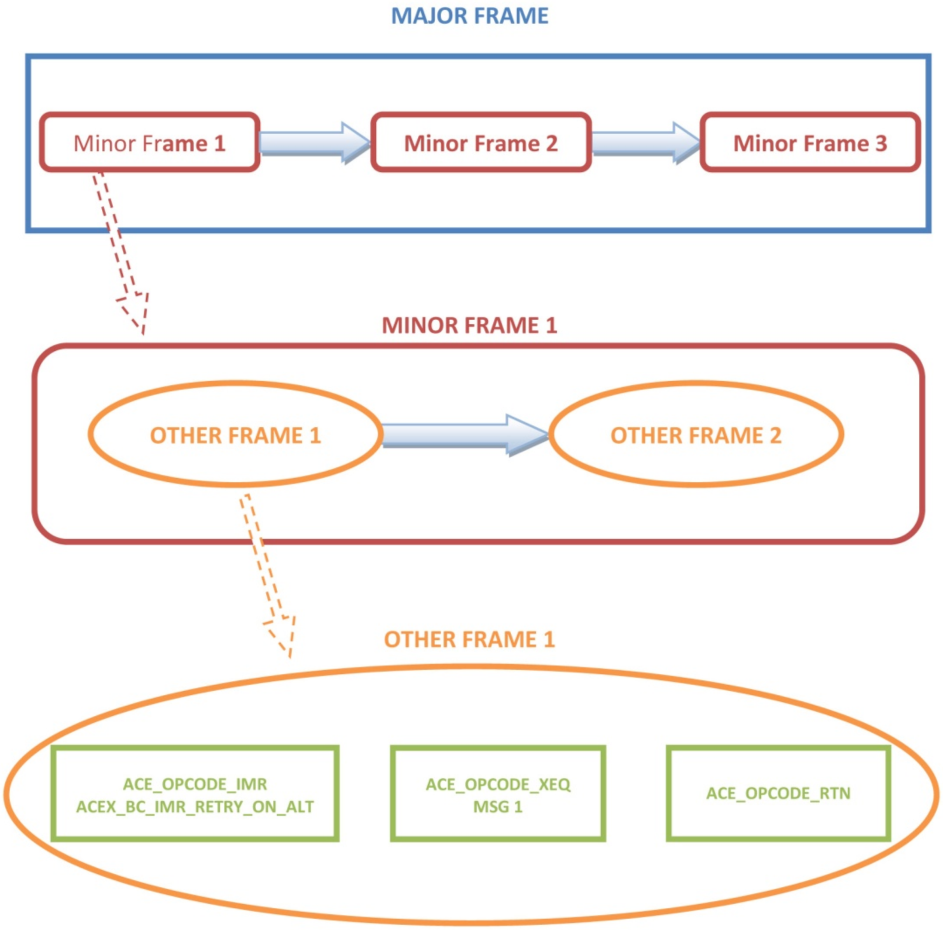


Figure 21. IMR Frame Sequence

```
/* Create IMR opcode that will use msg block */
nResult = aceBCOpCodeCreate(
            0,                     /* LDN             */
            OP1                    /* Opcode ID       */
            ACE_OPCODE_IMR,        /* Opcode for IMR  */
            ACE_CNDTST_ALWAYS,     /* Always operate IMR  */
            ACEX_BC_IMR_RETRY_ALT, /* IMR Retry on alt bus */
            0,                     /* Reserved        */
            0);                    /* Reserved        */

/* Create XEQ opcode that Msg 1 */
nResult = aceBCOpCodeCreate(
            0,                     /* LDN             */
            OP2                    /* Opcode ID       */
            ACE_OPCODE_XEQ,        /* Opcode for Execute  */
            ACE_CNDTST_ALWAYS,     /* Always operate XEQ  */
            MSG1,                  /* Message one     */
            0,                     /* Reserved        */
            0);                    /* Reserved        */

/* Create RTN opcode */
nResult = aceBCOpCodeCreate(
            0,                      /* LDN
*/
            OP3                     /* Opcode ID
*/
            ACE_OPCODE_RTN,         /* Opcode for Return
*/
            ACE_CNDTST_ALWAYS,      /* Always operate IMR
*/
            0,                      /* Reserved
*/
            0,                      /* Reserved
*/
            0);                     /* Reserved
*/

/* Create OTHER FRAME opcode */
nResult = aceBCOpCodeCreate(
            0,                      /* LDN
*/
            OP4                     /* Opcode ID
*/
            ACE_OPCODE_CAL,         /* Opcode for Call
*/
            ACE_CNDTST_ALWAYS,      /* Always operate IMR
*/
            OTHER1,                 /* Other Frame ID
*/
            0,                      /* Reserved
```

```
*/
            0);                     /* Reserved
*/

/* Create IMR Message Other Frame */
  aOpCodes[0] = OP1;               /* IMR Opcode            */
  aOpCodes[1] = OP2;               /* XEQ Opcode            */
  aOpCodes[2] = OP3;               /* RTN Opcode            */

nResult = aceBCFrameCreate(
            0,                     /* LDN                   */
            OTHER1                 /* Frame ID              */
            ACE_FRAME_OTHER,       /* Frame Type            */
            aOpCodes,              /* Opcode list           */
            3,                      /* # of Opcodes in list
*/
            0,                      /* Reserved
*/
            0);                     /* Reserved
*/
/* Create MINOR FRAME opcode */
nResult = aceBCOpCodeCreate(
            0,                      /* LDN
*/
            OP5                     /* Opcode ID
*/
            ACE_OPCODE_CAL,         /* Opcode for Call
*/
            ACE_CNDTST_ALWAYS,      /* Always operate IMR
*/
            MNR1,                   /* Major Frame ID
*/
            0,                      /* Reserved
*/
            0);                     /* Reserved
*/


aOpCodes[0] = OP4;                  /* Load Minor Frame
*/

/* Create Minor Frame */
nResult = aceBCFrameCreate(
            0,                      /* LDN
*/
            MNR1                    /* Frame ID
*/
            ACE_FRAME_MINOR,        /* Frame Type
*/
            aOpCodes,               /* Opcode list
```

```
*/
            1,                      /* # of Opcodes in list
*/
            0,                      /* Reserved
*/
            0);                     /* Reserved
*/

aOpCodes[0] = OP5;                  /* Load Major Frame
*/

/* Create Major Frame */
nResult = aceBCFrameCreate(
            0,                      /* LDN
*/
            MJR                     /* Frame ID
*/
            ACE_FRAME_MAJOR,        /* Frame Type
*/
            aOpCodes,               /* Opcode list
*/
            1,                      /* # of Opcodes in list
*/
            100,                    /* Frame time
*/
            0);                     /* Reserved
*/
```

Code Example 53. Configure BC IMRs

3.3.1.9.3 IMR and Triggers

BC intermessage routines may also generate or be generated by external triggers through the discrete I/O pins on the Multi-Function AceXtreme board. The function **acexBCImrTrigSelect()** to link a discrete I/O pin to the IMRs using the discrete IO pins or the wait for trigger IMR. The function requires the LDN, and the discrete pin number (0 – 15 depending on the number of discrete on the Multi-Function AceXtreme board).

```
S16BIT nResult;

/* BC IMR Trigger Select */
nResult = acexBCImrTrigSelect(0,          /* LDN        */
                              DIO 1);     /* discrete 1 */

if(nResult)
    printf(“acexBCImrTrigSelect Error: Code %d\n”,nResult);
```

Code Example 54. Configure Discrete to IMR

3.3.2 IRIG-106 Chapter 10 Monitor (ACE_MODE_MTI)

The IRIG-106 Chapter 10 Monitor (MT-I) provides a definitive solution for systems requiring IRIG-106 Chapter 10 support. In addition, DDC hardware supporting MT-I adds Direct Memory Access (DMA) and event performance enhancements to maximize monitor data throughput. Please note that this mode is only available for the E²MA and AceXtreme family of DDC devices.

Note: *Due to the numerous performance improvements of MT-I mode as compared to Classic Monitor mode (MT), it is recommended that this mode be used for 1553 Monitor Applications using DDC E²MA and AceXtreme Hardware and requiring high performance. For existing or retrofit Applications requiring Classic Monitor API Support, see Section 3.3.3.*

3.3.2.1 What is IRIG-106 Chapter 10?

IRIG-106 is a comprehensive telemetry standard to ensure interoperability in aeronautical telemetry application at United States Military RCC member ranges. IRIG-106 is developed and maintained by the Telemetry Group of the Range Commanders Council. Chapter 10 of the IRIG-106 document addresses the Digital On-Board Recorder Standard, which defines the operation and interfaces for digital flight data recorders. This new file format standard supports MIL-STD-1553, as well as other telemetry protocols (PCM, ARINC 429).

For more information on IRIG-106 Chapter 10, visit http://www.irig106.org/.

3.3.2.2 Theory of Operation

The MT-I Monitor creates IRIG-106 Chapter Data packets, including a packet header and trailer, containing any monitored 1553 traffic (after filter is applied, see Section 3.3.2.4). The MT-I Data Packet is compliant to MIL-STD-1553, Format 1 (MIL-STD-1553B) defined in RCC IRIG-106 Chapter 10 version 2004.

MT-I Data Packets consist of 3 parts: Packet Header, Packet Body and Packet Trailer. The size of each Data packet is variable based on the number of messages contained within (For controlling packet size, see Packet Generation Events, Section 3.3.2.2.4).

Table 37. MT-I General Data Packet Format

PACKET SYNC PATTERN	PACKET HEADER
CHANNEL ID	
PACKET LENGTH	
DATA LENGTH	
HEADER VERSION	
SEQUENCE NUMBER	
PACKET FLAGS	
DATA TYPE	
RELATIVE TIME COUNTER	
HEADER CHECKSUM	
CHANNEL SPECIFIC DATA	PACKET BODY
INTRA-PACKET TIME STAMP 1	
INTRA-PACKET DATA HEADER 1	
DATA 1	
• • •	
INTRA-PACKET TIME STAMP n	
INTRA-PACKET DATA HEADER n	
DATA	
DATA CHECK SUM	PACKET TRAILER

3.3.2.2.1 Packet Header

The MT-I Packet Header is included at the start of every MT-I Packet. The Header always contains the same 10 fields, which are defined in sections 3.3.2.2.1.1 through 3.3.2.2.1.10. For more detailed information on IRIG-106 Chapter 10 Packet Headers, see section 3.3.2.1.

3.3.2.2.1.1PACKET SYNC PATTERN (2 Bytes)

Packet Sync Pattern (2 Bytes) contains a static sync value for every packet. The Packet Sync Pattern value shall be 0xEB25.

3.3.2.2.1.2CHANNEL ID (2 Bytes)

Channel ID (2 Bytes) contains a value representing the Packet Channel ID. All channels in a system must have a unique value (data channels). Channel value 0x0000 is reserved, and is used to insert computer-generated messages into the composite data stream. Channel values 0x0001 through 0xFFFF are available.

Note: *The Channel ID can be set via the* ***aceMTIConfigure()*** *function. See Section 3.3.2.3.*

3.3.2.2.1.3PACKET LENGTH (4 Bytes)

Packet Length (4 Bytes) contains a value representing the length of the entire packet. The value shall be in bytes and is always a multiple of four (bits 1 and 0 shall always be zero). This Packet Length includes the Packet Header, Packet Secondary Header (if enabled), Channel Specific Data, Intra-Packet Data Headers, Data, Filler, and Data Checksum.

3.3.2.2.1.4DATA LENGTH (4 Bytes)

Data Length (4 Bytes) contains a value representing the length within the packet. This value shall be represented in bytes. Data length includes Channel Specific Data, Intra-Packet Data Headers, Intra-Packet Time Stamp, and Data but does not include Filler and Data Checksum.

3.3.2.2.1.5HEADER VERSION (1 Byte)

MT-I mode will always supply a fixed Header Version value of 0x02, indicating that the TG-78 header format is being used.

3.3.2.2.1.6SEQUENCE NUMBER (1 Byte)

Sequence Number (1 Byte) contains a value representing the packet sequence number for each Channel ID. This is simply a counter that increments by n + 0x01 to 0xFF for every packet transferred from a particular channel and is not required to start at 0x00 for the first occurrence of a packet for the Channel ID.

3.3.2.2.1.7PACKET FLAGS (1 Byte)

MT-I mode will always supply a fixed Packet Flags value of x00, which translates to the following:

- Packet Secondary Header is not present (Bit 7)
- Intra-Packet Time Stamp Source is the Packet Header 48-Bit Relative Time Counter (Bit 6)

- No Relative Time Counter sync error (Bit 5)
- No Data Overflow (Bit 4)
- No Packet Secondary Header Format (Bits 3-2)
- No Data Checksum Present (Bits 1-0)

3.3.2.2.1.8DATA TYPE (1 Byte)

Data Type (1 Byte) contains a value representing the type and format of the data. All values not used to define a data type are reserved for future data type growth:

MT-I mode supports 2 Data Types: MIL-STD-1553B (Format 1 (0x19)) and Time Data (Format 1 (0x11)). This value will change depending on whether you have received a 1553 Data Packet (Section 3.3.2.6.1) or a Time Data Packet (Section 3.3.2.6.2).

3.3.2.2.1.9RELATIVE TIME COUNTER

Relative Time Counter (6 Bytes) contains a value representing the Relative Time Counter. This is a free-running 10 MHz binary counter represented by 48 bits common to all data channels. The counter shall be derived from an internal crystal oscillator and shall remain free running during each recording session. The applicable data bit to which the 48-bit value applies, unless defined in each data type section, shall correspond to the first bit of the data in the packet body.

Note: *The Relative Time Counter can be an external 10MHz source on supported DDC hardware. The Clock source and resolution (Internal only) can be set via the* ***aceSetTimeTagRes()*** *function.*

Special Note: *When operating the following DDC cards, the ARINC Time Tag selection will override the Relative Time counter selection in the 1553 section. If IRIG is selected on the 1553 and the ARINC then selects the global 48-bit counter, the 1553 will be modified to use the global 48-bit counter as well. Cards affected:* ***BU-65590/91Ux****,* ***BU-65590F/Mx****, and* ***BU-65590Cx****.*

3.3.2.2.1.10 HEADER CHECKSUM (2 Bytes)

Header Checksum (2 Bytes) contains a value representing a 16 bit arithmetic sum of all 16-bit words in the header excluding the Header Checksum Word.

3.3.2.2.2 1553 Data Packet Body

The MT-I Packet Body consists of a Channel Specific Data Section, followed by 1 or more Message Data Sections. The number of 1553 messages in each Packet is controlled by the Packet Generation Events (see Section 3.3.2.2.4). Each Message

Data Section consists of 3 sections; an Intra-Packet Time Stamp, Intra-Packet Data Header and the actual monitored 1553 message data (Command, Data, Status).

Table 38. MT-I 1553 Complete Data Packet
PACKET HEADER
CHANNEL SPECIFIC DATA
INTRA-PACKET TIME STAMP FOR MESSAGE 1
INTRA-PACKET DATA HEADER FOR MESSAGE 1
MESSAGE 1
INTRA-PACKET TIME STAMP FOR MESSAGE 2
INTRA-PACKET DATA HEADER FOR MESSAGE 2
MESSAGE 2
• • •
INTRA-PACKET TIME STAMP FOR MESSAGE n
INTRA-PACKET DATA HEADER FOR MESSAGE n
MESSAGE n
PACKET HEADER

3.3.2.2.2.11553 Data Channel Specific Data

IRIG-106 Chapter 10 defines a “Channel Specific Data” section, which has a unique definition for each protocol supported by the standard. MT-I supports the MIL-STD-1553B, Format 1 protocol (0x19). The 1553B Channel Specific Data is included before any monitored messages and is decoded as shown in Figure 22.

MSB		LSB
31	15	0
Bits [31:30] TTB	Bits [29:24] RESERVED	Bits [23:0] MSGCOUNT

- Message Count (MSGCOUNT). (bits 23-0) indicate the binary value of the number of messages included in the packet. An integral number of complete messages will be in each packet.
- Reserved. (bits 29-24) are reserved.
- Time Tag Bits (TTB). (bits 31-30) indicate which bit of the MIL-STD-1553 message the Intra-Packet Header time tags.

00 = Last bit of the last word of the message
01 = First bit of the first word of the message
10 = Last bit of the first (command) word of the message <——— **Supported E^{2}MA Option**
11 = 0xE Reserved

Figure 22. MT-I 1553 Data Packet – Channel Specific Data

3.3.2.2.2.21553 Data Intra-Packet Time Stamp (8 Bytes)

Each 1553 Message will be stamped with an Intra-Packet Time Stamp, marking the relative reception time of the message. The value posted will contain a 48-bit Relative Time Counter (plus 16 high-order zero bits).

3.3.2.2.2.31553 Data Intra-Packet Data Header

Each 1553 Message that is monitored will contain an Intra-Packet Data Header. This header is used to identify size, timing and status of the 1553 message, including any observed errors.

Table 39. MT-I 1553 Data Packet - Intra-Packet Data Header

Word Offset	MSB	LSB
	15	0
+2	BLOCK STATUS WORD [15:0]	
+1	GAP TIMES WORD [15:0]	
+0	LENGTH WORD [15:0]	

The Block Status Word (BSW) is used to identify the health of the message. It can be used to identify erroneous message.

MSB														LSB
15						7								0
Bits	Bit	Bit	Bit	Bit	Bit	Bit	Bit	Bit	Bit	Bit	Bit	Bit	Bit	Bit
[15:14]	[13]	[12]	[11]	[10]	[9]	[8]	[7]	[6]	[5]	[4]	[3]	[2]	[1]	[0]
R	BID	ME	RR	FE	TM	R	R	R	LE	SE	WE	R	R	R

- Reserved. (Bits 15-14) are reserved

- Bus ID (BID). (Bit 13) indicates the bus ID for the message.
  0 = Message was from Channel A
  1 = Message was from Channel B
- Message Error (ME). (Bit 12) indicates a message error was encountered.
  0 = No message error
  1 = Message error
- RT to RT Transfer (RR). (Bit 11) indicates an RT to RT transfer: message begins with two command words.
  0 = No RT to RT transfer
  1 = RT to RT transfer
- Format Error (FE). (Bit 10) indicates a format error.
  0 = No format error
  1 = Format error
- Response Time Out. (Bit 9) indicates a response time out has occurred.
  0 = No response time out
  1 = Response time out
- Reserved. (Bit 8-6) are reserved
- Word Count Error (LE). (Bit 5) indicates a word count error has occurred.
  0 = No word count error
  1 = Word count error
- Sync Type Error (SE). (Bit 4) indicates an incorrect sync type occurred.
  0 = No sync type error
  1 = Sync type error
- Invalid Word Error (WE). (Bit 3) indicates an invalid word error has occurred.
  0 = No invalid word error
  1 = Invalid word error
- Reserved. (Bit 2-0) are reserved

Figure 23. MT-I 1553 Data Packet – Block Status Word

3.3.2.2.2.41553 Message Data Section

Table 40. MT-I 1553 Data Packet – Data Portion (1553 Command / Data / Status Words)
COMMAND WORD
COMMAND, STATUS, OR DATA WORD
DATA OR STATUS WORD
• • •
DATA OR STATUS WORD

3.3.2.2.3 Packet Trailer

MT-I mode supplies a Filler Packet trailer of 8 bytes. All bytes are set to 0x00.

3.3.2.2.4 Packet Generation Events

The MT-I architecture allows the user to define numerous events to trigger MT-I Packet generation. Each event can be used independently or in conjunction with each other to make sure Data Packets are generated at the desired rate, size, and conditions. If any of the interrupt event conditions are met, the MT-I engine will generate a valid Data Packet with the monitored 1553 data collected at the time of the event. The events will then be reset to collect a new Data Packet. Event conditions can be configured via the **aceMTIConfigure()** function (see Section 3.3.2.3).

3.3.2.2.4.1MTI_OVERFLOW_INT

This event condition will trigger a Data Packet to be generated if the DDC hardware has overflowed (i.e. reached the maximum amount of monitored data without being depleted).

3.3.2.2.4.2MTI_HOST_INT

This event condition will trigger a Data Packet to be generated if asynchronously requested by the host Application (via **aceMTIInitiateHostIrq()**).

3.3.2.2.4.3MTI_TIME_MSG_TRIG_INT

This event condition will trigger a Data Packet to be generated if a user-defined time value has expired since the reception of the last 1553 message.

3.3.2.2.4.4MTI_TIME_INT

This event condition will trigger a Data Packet to be generated if a user-defined time value has expired.

3.3.2.2.4.5 MTI_NUM_MSGS

This event condition will trigger a Data Packet to be generated if a user-defined number of 1553 messages have been monitored.

3.3.2.2.4.6MTI_NUM_WORDS

This event condition will trigger a Data Packet to be generated if a user-defined number of words (command and data) have been monitored.

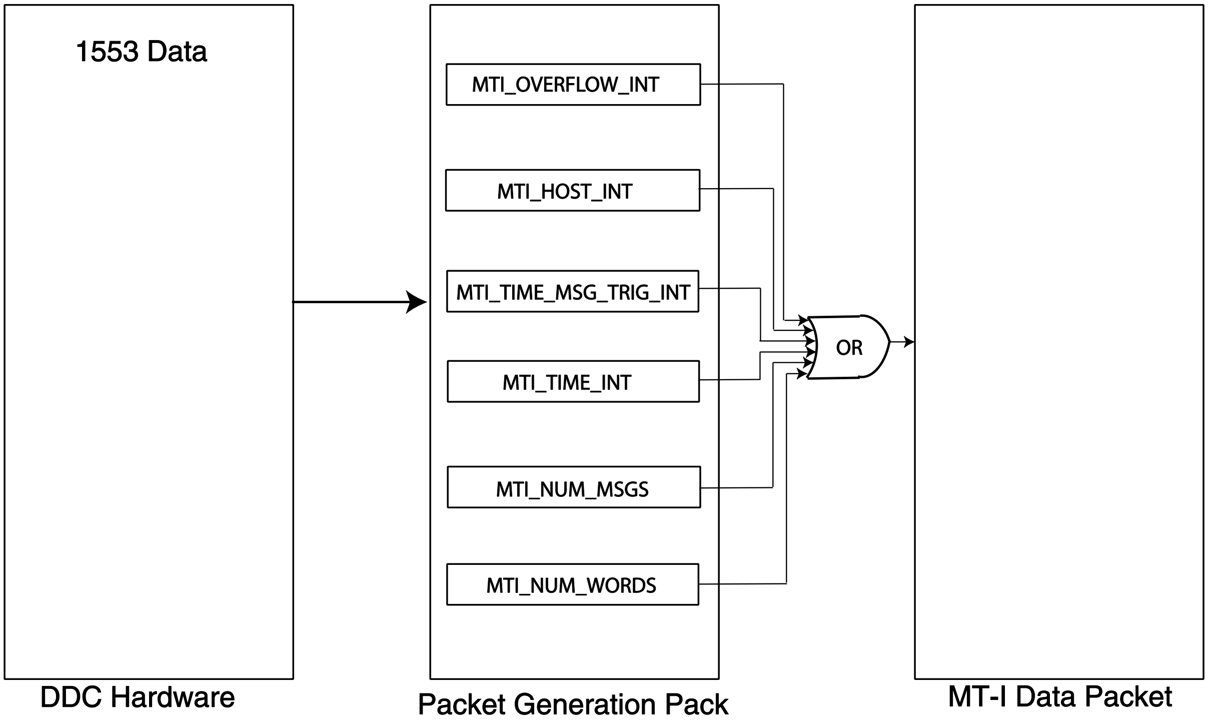


Figure 24. MT-I Data Packet Generation – Interrupt Events

3.3.2.3 Configuration

The AceXtreme C SDK’s MT-I Monitor has numerous configuration options that should be addressed before any monitoring is attempted. Configuration is accomplished via the **aceMTIConfigure()** function and is typically called after **aceInitialize()**.

Table 41. MT-I Configuration Parameters

Variable	Description	Valid Options or Range

u32DevBufByteSize	Size of DDC hardware memory (bytes) allocated for MT-I monitored data	MTI_DEVBUF_SIZE_128K = 128 KB
		MTI_DEVBUF_SIZE_256K = 256 KB
		MTI_DEVBUF_SIZE_512K = 512 KB
		MTI_DEVBUF_SIZE_1M = 1 MB

u32NumBufBlks	Number of memory blocks allocated for chapter 10 data packets	Target Host Memory Dependent
u32BufBlkByteSize	Bytes allocated for MT-I Data Packet buffer	Target Host Memory Dependent

fZeroCopyEnable	Enable Zero-Copy (Needs to be supported by target Operating System)	TRUE = Enable Zero-Copy
		FALSE = Disable Zero-Copy

u32IrqDataLen	Interrupt Event Option (MTI_NUM_WORDS): Number of data words necessary to generate a MT-I Data Packet	System Dependent
u32IrqMsgCnt	Interrupt Event Option (MTI_NUM_MSGS): Number of messages necessary to generate a MT-I Data Packet	System Dependent
u16IrqTimeInterval	Interrupt Event Option (MTI_TIME_MSG_TRIG_INT) (MTI_TIME_INT) Time Limit (µs) necessary to generate a MT-I Data Packet	System Dependent

u32IntConditions	Interrupt Events enabled to generate MT-I Data Packet	MTI_OVERFLOW_INT = Generate packet after DDC Hardware Overflow
		MTI_HOST_INT = Generate packet after Host Request
		MTI_TIME_MSG_TRIG_INT = Generate Packet after time limit reached, triggered by 1553 message
		MTI_TIME_INT = Generate Packet after time limit reached
		MTI_NUM_MSGS = Generate Packet after receiving “X” 1553 Messages
		MTI_NUM_WORDS = Generate Packet after receiving “X” words

u16Ch10ChnlId	IRIG-106 Chapter 10 assigned Channel ID for this device	0 - 65535
u8HdrVer	Reserved for Future Use	Reserved (0)
u8RelAbsTime	Reserved for Future Use	Reserved (0)
u8Ch10Checksum	Reserved for Future Use	Reserved (0)

The following options can be "logically OR'ed" into the 14th parameter (dwOptions) of **aceMTIConfigure()**.

Table 42. MT-I Configuration Options

Options	Description
ACE_MT_OPT_1553A_MC	Enable 1553A Mode Code Support
ACE_MT_OPT_BCST_DIS	Disable Broadcast Address (RT 31)
ACE_MTI_OPT_RTBUSY_DISABLE	Busy/Illegal bit and data valid format disable
ACE_MTI_OPT_EOM_TT_ENABLE	Enables EOM for TT
ACE_MTI_OPT_ERR_MON_ENA	MT-I Error monitor mode
ACE_MTI_OPT_REPLAY_MON_ENA	MT-I Replay Monitor Mode enabled
ACE_MTI_OPT_DDC_DATA_TYPE	Use DDC custom data types for MT-I, MT-IE and MTR packets

3.3.2.4 Message Filtering

The MT-I Monitor can be setup to filter messages based on RT address, Transmit or Receive command, and RT Subaddress. By default, the MT-I Monitor is configured to monitor all 1553 bus messages. Filtering out (disabling) messages is accomplished via the **aceMTDisableRTFilter()** function. For more information, see the **aceMTEnableRTFilter()** and **aceMTDisableRTFilter()** sections.

```
S16BIT nResult;

/* Disable RT5 TX SA19 */
nResult =  aceMTDisableRTFilter(
    0,                     /* LDN */
    5,                     /* RT Address to Filter */
    ACE_MT_FILTER_TX,      /* RT Type to Filter */
    ACE_MT_FILTER_SA19);   /* Subaddress(es) to Filter */

 if(nResult)
    printf("aceMTDisableRTFilter Error: Code %d\n", nResult);
```

Code Example 55. Filtering Out (Disabling) RT5 Transmit SubAddress 19

Note: *To read back the current Filter status of all of a particular RT, TX/RX, Subaddress combination, use the* ***aceMTGetRTFilter()*** *function.*

3.3.2.5 Activating the MT-I Monitor

Once the MT-I Monitor is configured and the RT Filtering has been setup, the Monitor is ready to begin storing 1553 bus traffic.

3.3.2.5.1 Starting and Stopping

The MT-I Monitor can be started and stopped dynamically by the user. Starting and Stopping is accomplished by the **aceMTIStart()** and **aceMTIStop()** functions.

> ***Note:*** *By Stopping the Monitor with **aceMTIStop()**, any traffic that has not been consumed by the user will be discarded. To temporarily pause monitoring without discarding data, see Section 3.3.2.5.2*

```
S16BIT nResult;

/* Start the MT-I Monitor */
nResult =  aceMTIStart(0);      /* LDN */

 if(nResult)
    printf(“aceMTIStart Error: Code %d\n”, nResult);
```

Code Example 56. Starting the MT-I Monitor

```
S16BIT nResult;

/* Stop the MT-I Monitor */
nResult =  aceMTIStop(0);       /* LDN */

 if(nResult)
    printf(“aceMTIStop Error: Code %d\n”, nResult);
```

Code Example 57. Stopping the MT-I Monitor

3.3.2.5.2 Continue and Pause

The **aceMTIPause()** function will temporarily pause the monitoring of bus traffic, leaving all unconsumed data intact. After pausing, bus monitoring can be restarted using the **aceMTIContinue()** function.

3.3.2.6 Consuming Data

Once the MT-I engine has been configured and the monitor is started, the user may begin to consume monitored data. The MT-I interface is event-driven and contains some blocking options to limit device polling.

3.3.2.6.1 Getting 1553 Data Packets

Available MT-I 1553 Data Packets can be consumed by the user via the **aceMTIGetCh10DataPkt()** function. For a non-zero copy configuration, the user must supply an allocated **MTI_CH10_DATA_PKT** buffer able to hold the largest achievable packet size (depending on Packet Generation options). In addition, the function can block until the packet is available.

```
S16BIT nResult;
MTI_CH10_DATA_PKT* pPkr;

/* Get a 1553 Data Packet */
pPkt = (MTI_CH10_DATA_PKT*) malloc(0x1000);
nResult =  aceMTIGetCh10DataPkt(
    0,          /* LDN */
    &pPkt,      /* MT-I Packet Storage */
    -1);        /* Wait forever (block) */

if(nResult)
    printf(“aceMTIGetCh10DataPkt Error: Code %d\n”, nResult);
```

Code Example 58. Getting a MT-I 1553 Data Packet

Note: *If the target Operating System supports zero-copy and it is enabled via* ***aceMTIConfigure()****, the buffer supplied to* ***aceMTIGetCh10DataPkt()*** *will be allocated by the* ***AceXtreme C SDK*** *and does not need to be allocated by the user.*

3.3.2.6.1.1Blocking Options

The Third Parameter (Timeout) of **aceMTIGetCh10DataPkt()** contains the timeout value. Depending on the value supplied, the function will return immediately, block until data is received, or block until a user-supplied timeout occurs. This functionality allows applications to poll or be event-driven depending on needs.

Table 43. Getting 1553 Data Packets: Blocking Options

"Timeout" Value	Observed Blocking Behavior of aceMTIGetCh10DataPacket()
-1	Function will block (not return) until a valid MT-I 1553 Data Packet is available (NOTE: Completion of a packet is based on one or more Packet Generation Events occurring).
0	Function will return immediately with or without an MT-I 1553 Data Packet
1 – 65535 (n)	Function will return as soon as a valid MT-I 1553 Data Packet is available or after "n milliseconds", which ever comes first.

3.3.2.6.1.2Using the Host-Initiated IRQ

One of the Packet Generation Event Options (MTI_HOST_INT) is a Host-Generated IRQ event. This option allows the user to cause an MT-I 1553 Data Packet to generate at their request. To initiate the Host IRQ, use the **aceMTIInitiateHostIrq()** function.

> ***Note:*** *In order to use the Host-Initiated IRQ event, the MTI_HOST_INT event option must be enabled. See section 3.3.2.3 for configuration information.*

3.3.2.6.2 Getting Time Data Packets (TDP)

Supporting DDC hardware will create Time Data Packets (TDP's) under MT-I mode. A Time Data Packet contains information from an internal or external time source (including IRIG-B) and will be generated at a frequency of 1 Hz. The TDP Packet Header and Trailer are in the same format as 1553 Data Packets, refer to Sections 3.3.2.2.1 and 3.3.2.2.3 for more information.

Table 44. MT-I Time Data Packet Format
PACKET HEADER
CHANNEL SPECIFIC DATA
TIME DATA
PACKET TRAILER

The Packet Header is identical to the Packet Header described in Section 3.3.2.2.1. Please refer to this section for more information.

3.3.2.6.2.1TDP Channel Specific Data

Since the Time Data Packets is treated as a separate protocol, IRIG-106 Chapter 10 defines a unique "Channel Specific Data" portion, which is defined as follows.

MSB				LSB
31	15			0
Bits [31:12] RESERVED		Bits [11:8] DATE	Bits [7:4] FMT	Bits [3:0] SRC

- Time Source (SRC). (bits 3-0) indicates the source of the time in the payload of each time packet.

  0x0 = Internal (Time derived from the Clock in the Recorder)
  0x1 = External (Time derived from a Clock not in the Recorder)
  0x2 = Internal from RMM (Internal Time derived from the Clock in the RMM)
  0x3 = 0xE Reserved
  0xF = None

- Time Format (FMT). (bits 7-4) indicate the Time Data Packet format. All bit patterns not used to define a time format type are reserved for future data type growth.

  0x0 = IRIG-B <——— **Supported E^{2}MA Option**
  0x1 = IRIG-A
  0x2 = IRIG-G
  0x3 = Internal Real time clock
  0x4 = UTC Time from GPS
  0x5 = Native GPS Time
  0x6 though 0xE = Reserved
  0xF = None (time packet payload invalid)

- Date Format (DATE). (bits 11-8) indicate the Date format. All bit patterns not used to define a date format type are reserved for future growth.

  Bits 11-10: Reserved

  Bit 9: Indicates Date Format

  0 = IRIG day available <——— **Supported E^{2}MA Option**
  1 = Month and Year available

  Bit 8: Indicates if this is a leap year

  0 = Not a leap year
  1 = Is a leap year

- Reserved. (bits 31-12) are reserved.

Figure 25. MT-I Time Data Packet – Channel Specific Data

3.3.2.6.2.2TDP Time Data

After the Channel Specific Data word, the time data words are inserted in the packet in Binary Coded Decimal (BCD) Day format as shown below.

Table 45. BCD Day Format

Word Offset	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
					Sn								Tmn			
+1	RSVD(00)		THn		Hn				0	TMn			Mn			
							HDn						Dn			

Table 46. MT-I Time Data Packet - Time Data

Code	Description	Code	Description
Tmn	Tens of milliseconds	TDn	Tens of days
Hmn	Hundreds of milliseconds	HDn	Hundreds of days
Sn	Units of seconds	On	Units of months
TSn	Tens of seconds	Ton	Tens of months
Mn	Units of minutes	Yn	Units of years
TMn	Tens of Minutes	Tyn	Tens of years
Hn	Units of hours	Hyn	Hundreds of years
THn	Tens of hours	Oyn	Thousands of years
Dn	Units of days	0	Always zero

3.3.2.6.2.3Packet Trailer

MT-I mode supplies a Filler Packet trailer of 8 bytes. All bytes are set to 0x00.

3.3.2.6.2.4Enabling TDPs

Once Time Data Packets are enabled, the ***AceXtreme C SDK*** will generate a new packet every second (1HZ rate).

```
S16BIT nResult;

/* Enable TDPs */
nResult =  aceMTICh10TimePktEnable(
    0,          /* LDN */
    TRUE);      /* Enable TDP’s */

if(nResult)
    printf(“aceMTICh10TimePktEnable Error: %d\n”,nResult);
```

Code Example 59. Enabling Time Data Packets

3.3.2.6.2.5 Consuming Time Data Packets (TDP’s)

Available MT-I 1553 Time Data Packets can be consumed by the user via the **aceMTIGetCh10TimePkt()** function. For a non-zero copy configuration, the user must supply an allocated **MTI_CH10_TIME_PKT** buffer able to hold the TDP. In addition, the function can block until a TDP packet is available.

Note: *A TDP will be available every second (1 Hz rate).*

```
S16BIT nResult;
MTI_CH10_TIME_PKT* pPkr;

/* Get a Time Data Packet */
pPkt = (MTI_CH10_TIME_PKT*) malloc(0x1000);
nResult =  aceMTIGetCh10TimePkt(
    0,          /* LDN */
    &pPkt,      /* TDP Packet Storage */
    -1);        /* Wait forever (block) */

if(nResult)
    printf(“aceMTIGetCh10TimePkt Error: Code %d\n”, nResult);
```

Code Example 60. Getting a MT-I 1553 Time Data Packet (TDP)

3.3.2.6.2.6 Blocking Options

The Third Parameter (Timeout) of **aceMTIGetCh10TimePkt()** contains the timeout value. Depending on the value supplied, the function will return immediately, block until data is received, or block until a user-supplied timeout occurs. This functionality allows applications to poll or be event-driven depending on needs.

Table 47. Getting Time Data Packets - Blocking Options

"Timeout" Value	Observed Blocking Behavior of aceMTIGetCh10DataPacket()
-1	Function will block (not return) until a valid MT-I 1553 Data Packet is available (Note: Completion of a packet is based on one or more Packet Generation Events occurring).
0	Function will return immediately with or without an MT-I 1553 Data Packet
1 – 65535 (n)	Function will return as soon as a valid MT-I 1553 Data Packet is available or after "n milliseconds", whichever comes first.

3.3.2.7 Chapter 10 File Access

The AceXtreme C SDK has functions allowing the user File IO access. These functions allow the user to save MT-I packets to an IRIG Chapter 10 file. This file can be used in **ACE_MODE_ALL** mode for replay. There is also a function allowing the user to read packets from the IRIG Chapter 10 file.

3.3.2.7.1 File open

The function **acexMTICh10FileOpen()** is used to open a Chapter 10 capture file for read / write access. A call to **acexMTICh10FileOpen()** will return pointer to a Chapter 10 file handle.   Parameters passed into the function are the Chapter 10 file name and path, the access mode (**MTI_CH10_FILE_READ** or **MTI_CH10_FILE_WRITE)**, a pointer to the TMATS header, and the length of the TMATS header in bytes.

If the file does not exist an empty file will be created. If the file is opened for write access and does exists, the contents of the file will be erased and it will be treated as a new empty file. If a TMATS head packet is provided, the header will be saved as the first packet in the file.

```
PMTI_CH10_FILE_HANDLE pCh10FileHandle;
S16BIT nResult;

/* Open CH10 File */
nResult =  aceMTICh10FileOpen(
    &pCh10FileHandle,     /* Pointer to CH10 File Handle */
    "Replay.ch10",        /* Replay file path\name       */
    MTI_CH10_FILE_WRITE,  /* read or written to file     */
    NULL,                 /* Pointer to TMATS header     */
    0);                   /* Length of TMATS header      */

if(nResult)
    printf("aceMTICh10FileOpen Error: Code %d\n",nResult);
```

Code Example 61. Open MT-I File with Write Access

3.3.2.7.2 File Close

Once all user access to the chapter 10 file has been completed, the handle to the open file must be closed. The handle can be closed by calling **acexMTICh10FileClose()**.

```
PMTI_CH10_FILE_HANDLE pCh10FileHandle;
S16BIT nResult;

/* Close CH10 File */
nResult =  aceMTICh10FileClose(
    &pCh10FileHandle);     /* Pointer to CH10 File Handle */

if(nResult)
    printf("aceMTICh10FileClose Error: Code %d\n",nResult);
```

Code Example 62. Open MT-I File

3.3.2.7.3 File Read

The **acexMTICh10FileRead()** function is used to read the current of next Packet in the Chapter 10 file. The function requires the handle (pCh10FileHandle) to the Chapter 10 file to be passed into the call to **acexMTICh10FileRead()**. The next parameter (u8PacketReadType) specifies which packet to read from the file, either the current packet (**MTI_CH10_FILE_READ_CURRENT_PACKET**) or the next packet (**MTI_CH10_FILE_READ_NEXT_PACKET**) in the chapter 10 file. The next parameter (pDataPacket) is a pointer where the packet from the chapter 10 file will be stored and returned to the user. This parameter may be set to NULL if reading only the packet header. The last parameter passed into **acexMTICh10FileRead()** is the packet length in bytes (excluding the header).

```
PMTI_CH10_FILE_HANDLE pCh10FileHandle;
S16BIT nResult;

/* Read next packet from File. */
nResult =  aceMTICh10FileRead(
    &pCh10FileHandle,              /* Pointer to File Handle */
    MTI_CH10FILE_READ_NEXT_PACKET, /* Read the next packet.  */
    pMtiCh10Header,                /* Pointer to File Handle */
    pDataPacket,                   /* Pointer to packet data */
    U32DataPacketLen,              /* Length of data packet  */

if(nResult)
    printf(“aceMTICh10FileRead Error: Code %d\n”,nResult);
```

Code Example 63. Read Packet from File

3.3.2.7.4 File Write

The **acexMTICh10FileWrite()** function is used to write a Packet to the Chapter 10 file. The function requires the handle (pCh10FileHandle) to the Chapter 10 file to be passed into the call to **acexMTICh10FileWrite()**. The next parameter (pPacket) is a pointer to valid packet data buffer. The last parameter passed into **acexMTICh10FileWrite()** is the packet length in bytes (excluding the header).

```
PMTI_CH10_FILE_HANDLE pCh10FileHandle;
MTI_CH10_DATA_PKT* pPacket = NULL;
S16BIT nResult;

/* Get packet. */
nResult = aceMTIGetCh10DataPkt(
    0,              /* LDN                          */
    &pPacket,       /* Pointer to packet            */
    10);            /* Timeout value in milliseconds */
if(nResult)
    printf(“aceMTIGetCh10DataPkt Error: Code %d\n”,nResult);

/* write packet to File. */
nResult =  aceMTICh10FileWrite(
    &pCh10FileHandle,               /* Pointer to File Handle */
    pPacket,                        /* Pointer to packet      */
    pPacket->u32PktLength);         /* Length of packet       */

if(nResult)
    printf(“aceMTICh10FileWrite Error: Code %d\n”,nResult);
```

Code Example 64. Write Packet to File

3.3.2.7.5 Offset

The **acexMTICh10FileGetOffset()** function is used to get the current offset of a packet from the beginning of the file. The **acexMTICh10FileGetOffset()** can only be called on files opened for reading. The function **acexMTICh10FileGetOffset()** can be used with **acexMTICh10FileSetOffset()**. The offset will always point to the beginning of a packet. Both functions require the handle to the Chapter 10 file and a Signed 64 bit value used for the offset.

If the offset is set to an invalid location such as, the middle of a packet, the read function will fail and return an error. It is recommend to only use the values returned by **acexMTICh10FileGetOffset()** when attempting to set the offset via the call to **acexMTICh10FileSetOffset()**.

```
PMTI_CH10_FILE_HANDLE pCh10FileHandle;
S64BIT pOffset
S16BIT nResult;

/* Get the current offset of Ch10 file. */
nResult =  aceMTICh10GetOffset(
    &pCh10FileHandle,      /* Pointer to File Handle */
    &pOffset);             /* Offset of file.        */

if(nResult)
    printf(“aceMTICh10GetOffset Error: Code %d\n”,nResult);
```

Code Example 65. Get File Offset

```
PMTI_CH10_FILE_HANDLE pCh10FileHandle;
S64BIT pOffset
S16BIT nResult;

/* Set the offset for the CH10 File. */
nResult =  aceMTICh10SetOffset(
    &pCh10FileHandle,      /* Pointer to File Handle */
    pOffset);              /* Offset into file.      */

if(nResult)
    printf(“aceMTICh10SetOffset Error: Code %d\n”,nResult);
```

Code Example 66. Set File Offset

3.3.3 Classic Monitor (ACE_MODE_MT)

The ***AceXtreme C SDK*** Classic Monitor provides a flexible interface that allows selective monitoring of 1553 messages based on RT Address, T/R, and Subaddress with very little host processor intervention. This mode recreates all command/response messages on the 1553 bus on channels A and B, and stores them on DDC hardware memory based on a user programmable filter (RT Address, T/R, and Subaddress). This monitor can be used as a monitor alone or in a combined RT/Monitor mode (see Section 3.3.6.2).

> ***Note:*** *Classic Monitor support should only be used for existing applications or if the target DDC hardware is not of the* ***E^{2}MA*** *or* ***AceXtreme*** *Family. New designs using* ***E^{2}MA*** *or* ***AceXtreme*** *hardware should use MT-I mode (see Section 3.3.2).*

The Classic Monitor hardware uses a command stack to store 1553 command information and a data stack to store 1553 data words. Depending on the application, these stacks can vary in size (see Section 3.3.3.1). In addition, interrupt events can be used to notify the user when the stacks are filling up.

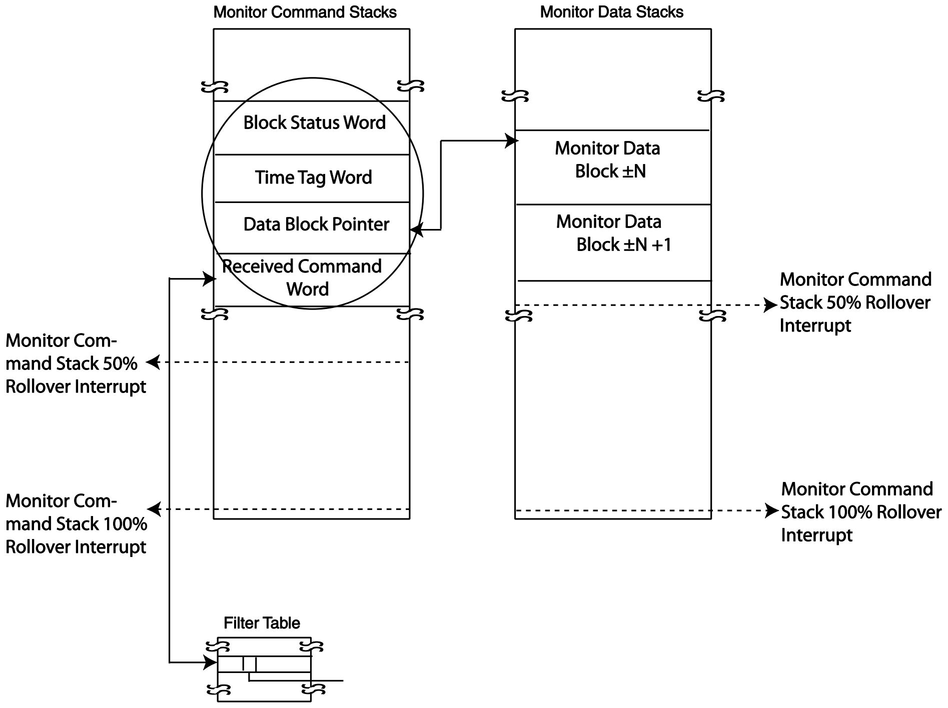

Monitor Command Stacks
Monitor Data Stacks
Block Status Word
Time Tag Word
Data Block Pointer
Received Command Word
Monitor Data Block ±N
Monitor Data Block ±N +1
Monitor Command Stack 50% Rollover Interrupt
Monitor Com-mand Stack 50% Rollover Interrupt
Monitor Com-mand Stack 100% Rollover Interrupt
Monitor Command Stack 100% Rollover Interrupt
Filter Table

Figure 26. Monitor Command and Data Stacks Relationship

3.3.3.1 Configuration

The AceXtreme C SDK’s Classic Monitor has numerous configuration options that should be addressed before any monitoring is attempted. Configuration is accomplished via the **aceMTConfigure()** function and is typically called after **aceInitialize()**.

Variable	Description	Valid Options

wMTStkType	Stack type to use for command and data stacks	ACE_MT_SINGLESTK = Single-Buffered Stack (default)
		ACE_MT_DOUBLESTK = Double-Buffered Stack

wCmdStkSize	Size (in words) of the command stack	ACE_MT_CMDSTK_256 -> 256 words ACE_MT_CMDSTK_1K -> 1K words ACE_MT_CMDSTK_4K -> 4K words (default) ACE_MT_CMDSTK_16K -> 16K words
wDataStkSize	Size (in words) of the data stack	ACE_MT_DATASTK_512 -> 512 words ACE_MT_DATASTK_1K -> 1K words ACE_MT_DATASTK_2K -> 2K words ACE_MT_DATASTK_4K -> 4K words ACE_MT_DATASTK_8K -> 8K words ACE_MT_DATASTK_16K -> 16K words (default) ACE_MT_DATASTK_32K -> 32K words ACE_MT_DATASTK_64K -> 64K words

The following options can be “logically OR’ed” into the Fifth parameter of **aceMTConfigure()**.

Options	Description
ACE_MT_OPT_1553A_MC	Enable 1553A Mode Code Support
ACE_MT_OPT_BCST_DIS	Disable Broadcast Address (RT 31)

3.3.3.2 Message Filtering

The Classic Monitor can be setup to filter messages based on RT address, Transmit or Receive command, and RT Subaddress. By default, the Classic Monitor is configured to monitor all 1553 bus messages. Filtering out (disabling) messages is accomplished via the**aceMTDisableRTFilter()** function. For more information, see the **aceMTEnableRTFilter()** and **aceMTDisableRTFilter()** sections.

```
S16BIT nResult;

/* Disable RT5 TX SA19 */
nResult =  aceMTDisableRTFilter(
    0,                      /* LDN */
    5,                      /* RT Address to Filter */
    ACE_MT_FILTER_TX,       /* RT Type Filter */
    ACE_MT_FILTER_SA19);  /* Subaddress(es) to Filter */

if(nResult)
    printf(“aceMTDisableRTFilter Error: Code %d\n”, nResult);
```

Code Example 67. Filtering Out (Disabling) RT5 Transmit SubAddress 19

Note: *To read back the current Filter status of all of a particular RT, TX/RX, Subaddress combination, use the **aceMTGetRTFilter()** function.*

3.3.3.3 Activating the Monitor

Once the Classic Monitor is configured and the RT Filtering has been setup, the Monitor is ready to begin storing 1553 bus traffic.

3.3.3.3.1 Starting and Stopping

The Classic Monitor can be started and stopped dynamically by the user. Starting and Stopping is accomplished by the **aceMTStart()** and **aceMTStop()** functions.

Note: *By stopping the Monitor with **aceMTStop()**, any traffic that has not been consumed by the user will be discarded. To temporarily pause monitoring without discarding data, see Section 3.3.3.3.2.*

```
S16BIT nResult;

/* Start the MT Monitor */
nResult =  aceMTStart(0);      /* LDN */

if(nResult)
    printf(“aceMTStart Error: Code %d\n”, nResult);
```

Code Example 68. Starting the MT Monitor

```
S16BIT nResult;

/* Stop the MT Monitor */
nResult =  aceMTStop(0);        /* LDN */

if(nResult)
    printf(“aceMTStop Error: Code %d\n”, nResult);
```

Code Example 69. Stopping the MT Monitor

3.3.3.3.2 Continue and Pause

The **aceMTPause()** function will temporarily pause the monitoring of bus traffic, leaving all unconsumed data intact. After pausing, bus monitoring can be restarted using the **aceMTContinue()** function.

3.3.3.4 Consuming Data

The Classic Monitor supports two methods of consuming monitored data: Stack Access and Host Buffer. Each method has particular advantages based on the user’s Application needs.

3.3.3.4.1 Data via Host Buffer

The MT Host Buffer (HBUF) is a circular memory buffer resident on the host that contains the log of all monitored messages in the order they appeared on the 1553 bus.

One advantage of using a Host Buffer is that all monitored messages are automatically transferred to the HBUF by means of internally configured interrupt events. This will make sure that monitored data is removed from DDC hardware and placed into the Host Buffer before any data loss can occur.

Another advantage is that the size of the host buffer can be fairly large and can serve as an elasticity buffer for applications that cannot consume data at a high rate.

3.3.3.4.1.1Installing the Host Buffer

The Host Buffer should be installed before any traffic monitoring occurs.

***Note:** The Host Buffer size should typically be 8-10 times larger than the maximum capacity of the DDC hardware (MT Command Stack).*

The following equation can be used to calculate the correct Host Buffer size:

[(CMD_STACK_SIZE / 4) * 40] * 8

```
S16BIT nResult;
#define CMD_STK_SIZE = 4096

/* Install the host buffer */
nResult =  aceMTInstallHBuf(
    0,                             /* LDN */
    ((CMD_STK_SIZE/4)*40)*8);  /* Host Buffer Size */

if(nResult)
    printf(“aceMTInstallHBuf Error: Code %d\n”, nResult);
```

Code Example 70. Installing the MT Host Buffer

3.3.3.4.1.2Reading the Host Buffer

The Host Buffer architecture is designed to automatically remove monitored data from the DDC hardware and place it into the host-allocated Host Buffer. It is the user’s responsibility to read entries from the Host Buffer for consumption.

Messages can be read off the Host Buffer in two formats: Raw or Decoded. Depending on which method is used, messages taken off the Host Buffer will be returned in FIFO order or LIFO order.

3.3.3.4.1.2.1 RAW FORMAT

The Raw Format will return a U16BIT pointer to the binary data. Using this method will also allow more than one message to be read off the Host Buffer at one time. Each message will be fixed-length of 40 words and will use zero fill words for messages not meeting the required word maximum. For example, if two messages have been monitored, the binary data will be 80 words deep, with the second message starting at offset 40.

Table 48. MT Host Buffer Raw Format for One Message

Word	MSB		LSB
Bits	**15**	**7**	**0**
0	Block Status Word		
1	Time Tag Word		
2	Bit [15] EOM	Bits [14:8] Data Length (in Words)	Bits [7:0] 1553 Message Type
3	1553 Command Word		
4	2nd 1553 Command Word (RT→ RT Transfers only)		
5	RT Status Word		
6	2nd RT Status Word (RT→RT Transfers only)		
7	Data Word 0		
8	Data Word 1		
n+6	Data Word n		

Messages can be read from the Host Buffer in Raw Format using the **aceMTGetHBufMsgsRaw()** function. The function will return up to “*wBufferSize*” words or all messages, whichever is smaller. The “pdwMsgCount” pointer will inform the user of the number of messages returned.

Note: *Each message is a fixed length of 40 words.*

```
S16BIT nResult;
U32BIT dwStkLost, dwHBufLost, dwMsgCount;
U16BIT wBuffer[400] = { 0x00000000 };

/* Get Raw Messages from the Host Buffer */
nResult =  aceMTGetHBufMsgsRaw(
    0,                /* LDN */
    wBuffer,          /* Buffer Storage */
    400,              /* Max Size of Buffer */
    &dwMsgCount,      /* Number of Msgs copied */
    &dwStkLost,       /* Messages lost on stack (if any) */
    &dwHBufLost);     /* Messages lost on Hbuf (if any) */

if(nResult)
    printf(“aceMTGetHBufMsgsRaw Error: Code %d\n”, nResult);
```

Code Example 71. Reading Raw Data From the Host Buffer

3.3.3.4.1.2.2 DECODED FORMAT

The Decoded Format will read one message off the Host Buffer and decode it into a MSGSTRUCT structure object. In addition, the user can decide whether to read the oldest (next) or latest message and whether or not to remove (purge) the message from the Host Buffer.

A Message can be read from the Host Buffer in Decoded Format using the **aceMTGetHBufMsgDecoded()** function. The function will return one message decoded into the “pMsg” MSGSTRUCT variable. The “wMsgLoc” variable is used to define which message to read and whether or not to remove it from the Host Buffer.

Table 49. Host Buffer Message Location and Purge Options (wMsgLoc)

Option	Description
ACE_MT_MSGLOC_NEXT_PURGE	Reads next message and takes it off of the host buffer
ACE_MT_MSGLOC_NEXT_NPURGE	Reads next message and leaves it on the host buffer
ACE_MT_MSGLOC_LATEST_PURGE	Reads current message and takes it off of the host buffer
ACE_MT_MSGLOC_LATEST_NPURGE	Reads current message and leaves it on the host buffer

```
/*Global (used for all modes) Message Structure for decoded 1553 msgs */
typedef struct MSGSTRUCT
{
  U16BIT wTYPE;               /* Contains the msg type (see above) */
  U16BIT wBlkSts;             /* Contains the block status word */
  U16BIT wTimeTag;            /* Time Tag of message */
  U16BIT wCmdWrd1;            /* First command word */
  U16BIT wCmdWrd2;            /* Second command word (RT to RT) */
  U16BIT wCmdWrd1Flg;         /* Is command word 1 valid? */
  U16BIT wCmdWrd2Flg;         /* Is command word 2 valid? */
  U16BIT wStsWrd1;            /* First status word */
  U16BIT wStsWrd2;            /* Second status word */
  U16BIT wStsWrd1Flg;         /* Is status word 1 valid? */
  U16BIT wStsWrd2Flg;         /* Is status word 2 valid? */
  U16BIT wWordCount;          /* Number of valid data words */
  U16BIT aDataWrds[32];       /* An array of data words */

/* The following are only applicable in BC mode */
  U16BIT wBCCtrlWrd;          /* Contains the BC control word */
  U16BIT wBCGapTime;          /* Message gap time word */
  U16BIT wBCLoopBack1;        /* First looped back word */
  U16BIT wTimeTag2;           /* wBCLoopBack2 is redefined as TimeTag2 */
  U16BIT wBCLoopback1Flg;     /* Is loopback 1 valid? */
  U16BIT wTimeTag3;           /* wBCLoopBack2Flg is redefined as TimeTag3 */
}MSGSTRUCT;
```

Figure 27. MT Host Buffer MSGSTRUCT Object Definition

```
S16BIT nResult;
U32BIT dwStkLost, dwHBufLost, dwMsgCount;
MSGSTRUCT sMsg;

/* Get a Decoded Message from the Host Buffer */
nResult =  aceMTGetHBufMsgDecoded(
    0,                         /* LDN */
    &sMsg,                     /* Message storage */
    &dwMsgCount,               /* Num of Msgs copied */
    &dwStkLost,                /* Messages Lost (Stack) */
    &dwHBufLost,               /* Messages Lost (Hbuf) */
    ACE_MT_MSGLOC_NEXT_PURGE); /* Purge messages */

if(nResult)
  printf(“aceMTGetHBufMsgDecoded Error: Code %d\n”, nResult);
```

Code Example 72. Reading a Decoded Message from the Host Buffer

3.3.3.4.2 Data via Stack

For Applications that have strict timing requirements and need quick access to 1553 data, the user can read monitored data directly off of the DDC hardware. This can be accomplished using the **aceMTGetStkMsgsRaw()** and **aceMTGetStkMsgDecoded()** functions.

3.3.3.4.2.1Reading the Stack

The MT Stacks are the lowest level of data storage. It consists of a command stack holding message/routing information and a data stack holding 1553 data words. The stacks will store monitored data (after filtering, see Section 3.3.3.2) until full and will then start overwriting the oldest data. It is the user's responsibility to read entries from the stacks before an overrun occurs.

Messages can be read off of the Stacks in two formats: Raw or Decoded. Depending on which method is used, Messages taken off of the stacks will be returned in FIFO order or LIFO order.

3.3.3.4.2.1.1 RAW FORMAT

The **Raw Format** will return a U16BIT pointer to the binary data. This method will also allow more than one message to be read off the stacks at one time. Each message will be fixed-length of 40 words and will use zero fill words for messages not required the word maximum. For example, if two messages have been monitored, the binary data will be 80 words deep, with the second message starting at offset 40.

Table 50. MT Raw Format for Message One

Word	MSB		LSB
Bits	15	7	0
0	Block Status Word		
1	Time Tag Word		
2	Bit [15] EOM	Bits [14:8] Data Length (in Words)	Bits [7:0] 1553 Message Type
3	1553 Command Word		
4	2nd 1553 Command Word (RT→ RT Transfers only)		
5	RT Status Word		
6	2nd RT Status Word (RT→RT Transfers only)		
7	Data Word 0		
8	Data Word 1		
n+6	Data Word n		

Messages can be read from the stacks in Raw Format using the **aceMTGetStkMsgsRaw()** function. The function will return up to “*wBufferSize*” words or all messages, whichever is smaller. The return value of the function will inform the user of the number of messages returned or if an error occurred.

Note: *Each message is a fixed length of 40 words.*

```
S16BIT nResult;
U16BIT wBuffer[400] = { 0x00000000 };

/* Get Raw Messages from the Stacks */
nResult =  aceMTGetStkMsgsRaw(
    0,                       /* LDN */
    wBuffer,                 /* Buffer storage */
    400,                     /* Max size of Buffer */
    ACE_MT_STKLOC_ACTIVE);   /* Read active stack */

if(nResult)
    printf(“aceMTGetStkMsgsRaw() Error: Code %d\n”, nResult);
```

Code Example 73. Reading Raw Data From the Stacks

3.3.3.4.2.1.2 DECODED FORMAT

The Decoded Format will read one message off the Stacks and decode it into a MSGSTRUCT structure object. In addition, the user can decide whether to read the oldest (next) or latest message and whether or not to remove (purge) the message from the Stacks.

A message can be read from the Stacks in Decoded Format using the **aceMTGetStkMsgDecoded()** function. The function will return one message decoded into the “pMsg” MSGSTRUCT variable. The “wMsgLoc” variable is used to define which message to read and whether or not to remove it from the Host Buffer.

Table 51. Stacks Message Location and Purge Options (wMsgLoc)

Option	Description
ACE_MT_MSGLOC_NEXT_PURGE	Reads next message and takes it off of the host buffer
ACE_MT_MSGLOC_NEXT_NPURGE	Reads next message and leaves it on the host buffer
ACE_MT_MSGLOC_LATEST_PURGE	Reads current message and takes it off of the host buffer
ACE_MT_MSGLOC_LATEST_NPURGE	Reads current message and leaves it on the host buffer

```
/*Global (used for all modes) Message Structure for decoded 1553 msgs */
typedef struct MSGSTRUCT
{
  U16BIT wTYPE;               /* Contains the msg type (see above) */
  U16BIT wBlkSts;             /* Contains the block status word */
  U16BIT wTimeTag;            /* Time Tag of message */
  U16BIT wCmdWrd1;            /* First command word */
  U16BIT wCmdWrd2;            /* Second command word (RT to RT) */
  U16BIT wCmdWrd1Flg;         /* Is command word 1 valid? */
  U16BIT wCmdWrd2Flg;         /* Is command word 2 valid? */
  U16BIT wStsWrd1;            /* First status word */
  U16BIT wStsWrd2;            /* Second status word */
  U16BIT wStsWrd1Flg;         /* Is status word 1 valid? */
  U16BIT wStsWrd2Flg;         /* Is status word 2 valid? */
  U16BIT wWordCount;          /* Number of valid data words */
  U16BIT aDataWrds[32];       /* An array of data words */

/* The following are only applicable in BC mode */
  U16BIT wBCCtrlWrd;          /* Contains the BC control word */
  U16BIT wBCGapTime;          /* Message gap time word */
  U16BIT wBCLoopBack1;        /* First looped back word */
  U16BIT wTimeTag2;           /*wBCLoopBack2 is redefined as TimeTag2 */
  U16BIT wBCLoopback1Flg;     /* Is loopback 1 valid? */
  U16BIT wTimeTag3;           /* wBCLoopBack2Flg is redefined as TimeTag3 */
}MSGSTRUCT;
```

Figure 28. MT Stack MSGSTRUCT Object Definition

```
S16BIT nResult;
MSGSTRUCT sMsg;

/* Get a Decoded Message from the Stacks */
nResult =  aceMTGetStkMsgDecoded(
    0,                         /* LDN */
    &sMsg,                     /* Message Storage */
    ACE_MT_MSGLOC_NEXT_PURGE,  /* Read and Purge */
    ACE_MT_STKLOC_ACTIVE);     /* Read active stack */

if(nResult)
   printf(“aceMTGetStkMsgDecoded Error: Code %d\n”, nResult);
```

Code Example 74. Reading a Decoded Message from the Stacks

3.3.3.4.3 MT Block Status Word

The Block Status Word (BSW) is used to identify the health of the message. The BSW contains information regarding the message, specifying whether the message is in progress or has been completed, what channel the message was processed on, and whether or not there were any errors in the message table. The MT Block status word's bits are defined in

Table 52. MT Block Status Word

Bit	Description	
15 (MSB)	EOM	Set at the completion of a BC message, regardless of whether or not there were any errors in the message.
14	SOM	Set at the start of a BC message and cleared at the end of the message.
13	A/B CHANNEL	This bit will be low if the message was processed on Channel A or high if the message was processed on Channel B
12	ERROR FLAG	If this bit is high, one or more of bits 10, 9, and/or 8 are also set high.
11	RT-RT FORMAT	Is set when to indicate the message was an RT-to-RT command.
10	FORMAT ERROR	If set, indicates the received portion of a message contained one or more violations of the 1553 message validation criteria (sync, encoding, parity, bit count, word count, etc.), or the RT's status word received from a responding RT contained an incorrect RT address field.
9	NO RESPONSE TIMEOUT	If set, indicates that an RT has either not responded or has responded later than the BC No Response Timeout time.
8	GOOD DATA BLOCK TRANSFER	Set to '1' following completion of a valid (error-free) message.
7	DATA STACK ROLLOVER	Indicates the current message results in the value of the Monitor Data Stack Pointer rolling over from the bottom to the top of its range.
6	RESERVED	Reserved for future use.
5	WORD COUNT ERROR	Indicates the BC did not transmit the correct number of Data Words.
4	INCORRECT SYNC	Indicates the BC transmitted a Command sync in a Data Word.
3	INVALID WORD	Indicates the BC (or transmitting RT in an RT-to-RT transfer) transmitted with one or more words containing one or more of the following error types: sync field error, Manchester encoding error, parity error, and/or bit count error.
2	RT-RT GAP / SYNC ADDRESS ERROR	This bit is set if the RT is the receiving RT for an RT-to-RT transfer and one or more of the following occur: 1. The GAP CHECK ENABLED bit is set to logic "1" and the transmitting RT responds with a response time of less than 4 µs. 2. There is an incorrect sync type or format error (encoding, bit count, and/or parity error) in the transmitting RT Status Word. 3. The RT address field of the transmitting RT Status Word does not match the RT address in the transmit Command Word.
1	RT-RT 2ND COMMAND ERROR	If the device is the receiving RT for an RT-to-RT transfer, this bit set indicates one or more of the following error conditions in the transmit Command Word: 1. T/R bit = logic "0" 2. Subaddress = 00000 or 11111 3. Same RT Address field as the receive Command Word.

Table 52. MT Block Status Word

Bit	Description	
0 (LSB)	COMMAND WORD CONTENTS ERROR	This bit indicates a received command word is not defined in accordance with MIL-STD-1553B. This includes the following undefined Command Words: 1. BROADCAST DISABLED **and** the Command Word is a non-mode code, broadcast, transmit command. 2. The OVERRIDE MODE T/R ERROR bit is logic "0" **and** a message with a T/R bit of "0," a subaddress/mode field of 00000 or 11111 and a mode code field between 00000 and 01111. 3. BROADCAST DISABLED **and** a mode code command that is not permitted to broadcast (e.g.. Transmit status) is sent to the broadcast address (11111).

3.3.3.4.4 Using Interrupt Events

Some applications may benefit from event notifications regarding the monitoring of 1553 traffic. The following events directly relate to the Classic Monitor (MT) mode of operation. For information on how to configure events and callbacks, see Section 3.2.3.

Table 53. MT Interrupt Events

Event	Description
ACE_IMR1_TT_ROVER (Bit 6)	Indicates the Hardware Timetag has rolled over
ACE_IMR1_MT_DATASTK_ROVER (Bit 10)	Indicates the MT Data Stack has rolled over
ACE_IMR1_MT_CMDSTK_ROVER (Bit 11)	Indicates the MT Command Stack has rolled over
ACE_IMR2_MT_DSTK_50P_ROVER (Bit 22)	Indicates the MT Data Stack has passed the 50% mark
ACE_IMR2_MT_CSTK_50P_ROVER (Bit 23)	Indicates the MT Command Stack has passed the 50% mark

3.3.4 Remote Terminal (ACE_MODE_RT)

The AceXtreme C SDK Remote Terminal (RT) implements all of the MIL-STD-1553B message formats and dual redundant mode codes for MIL-STD-1553B RT operation. The RT performs comprehensive error checking, word and format validation, and checks for various RT to RT transfer errors. One of the main features of the AceXtreme C SDK RT is the choice of Data Block memory management schemes. These include single buffering by subaddress, double buffering for individual receive subaddresses, circular buffering by individual subaddresses and a global circular buffering option for multiple (or all) subaddresses. Other features include a set of interrupt conditions, internal command illegalization, programmable busy by subaddress, and multiple options for time tagging.

The RT hardware uses a command stack to store 1553 command information and Data Blocks to store 1553 data words. Depending on the application, the command stack can be varied in size (see Section 3.3.4.1) and the Data Blocks can be configured in numerous memory management schemes (see Section 3.3.4.3). The SubAddress (SA) Mapping Table will determine which Data Block object is used for any particular command word. In addition, interrupt events can be used to notify the user when the stack and Data Blocks are filling up or have new data.

The AceXtreme devices support a Multi-RT mode. This mode allows for the use of more than one RT on one channel. Multi-RT mode can be used by passing **ACE_MODE_MRT** into **aceInitialize()**.

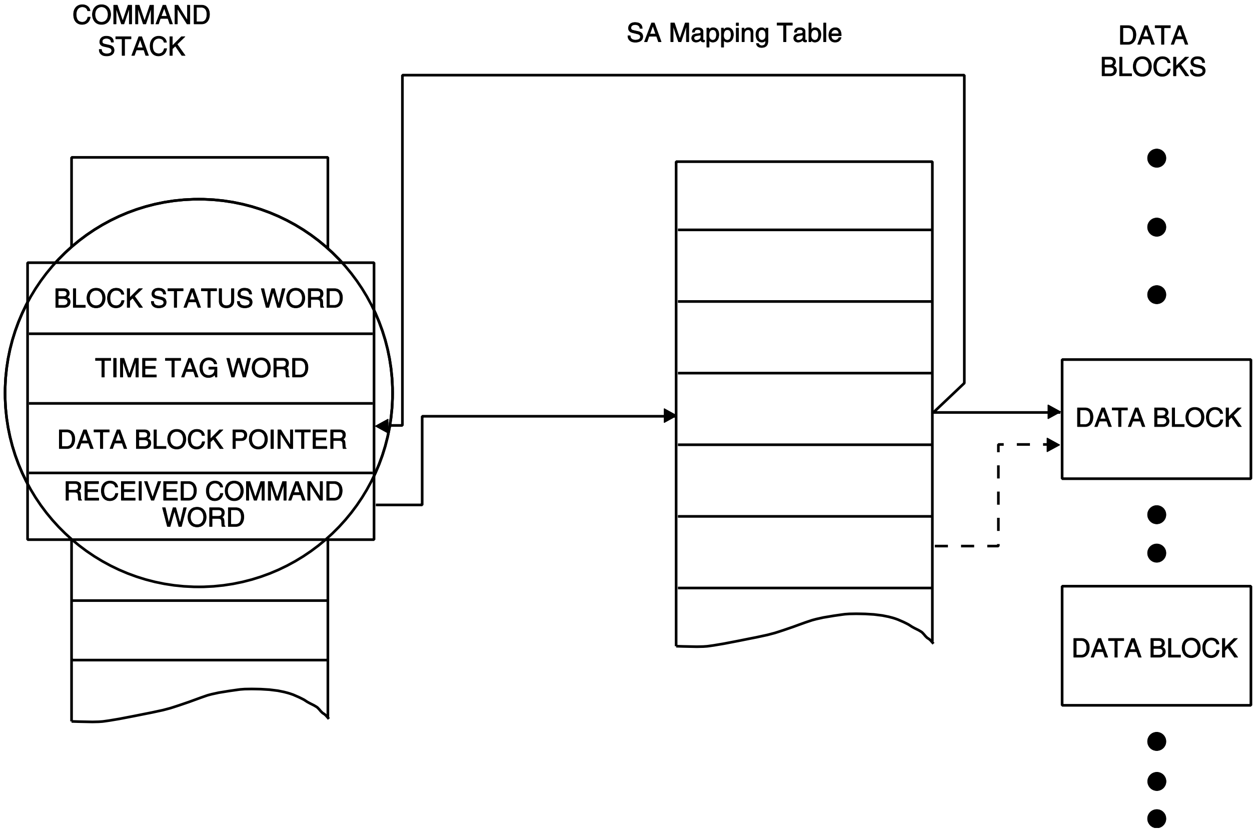


Figure 29. RT Command Stack, SA Mapping Table and Data Blocks Relationship

3.3.4.1 Configuration

The AceXtreme C SDK s Remote Terminal has numerous configuration options that should be addressed before the RT is brought online. Configuration is accomplished via the **aceRTConfigure()** function and is typically called after **aceInitialize()**.

Table 54. RT Configuration Parameters

Variable	Description	Valid Options
wCmdStkSize	Size (in words) of the command stack	ACE_RT_CMDSTK_256-> 256 words ACE_RT_CMDSTK_512-> 512 words ACE_RT_CMDSTK_1K-> 1K words ACE_RT_CMDSTK_2K-> 2K words (default)

The following options can be “logically OR’ed” into the Third parameter (dwOptions) of **aceRTConfigure()**.

Table 55. RT Configuration Options

Options	Description
ACE_RT_OPT_CLR_SREQ	Sets the Clear Service Request bit 2 to a 1. This will clear a service request after a tx vector word.
ACE_RT_OPT_LOAD_TT	With the reception of a Synchronize (with data) mode command, this will cause the Data Word from the Synchronize message to be loaded into the Hardware Time Tag Register.
ACE_RT_OPT_CLEAR_TT	With the reception of a Synchronize (without data) mode command, this will cause the value of the Hardware Time Tag Register to clear to 0x0000.
ACE_RT_OPT_OVR_DATA	This option affects the operation of the RT subaddress circular buffer memory management scheme. The Lookup Table address pointer will only be updated following a transmit message or following a valid receive or broadcast message to the respective Rx/Bcst subaddress. If this option is enabled, the Lookup Table pointer will not be updated following an invalid receive or broadcast message. In addition, an interrupt request for a circular buffer rollover condition (if enabled) will only occur following the end of a transmit message during which the last location in the circular buffer has been read or following the end of a valid receive or Broadcast message in which the last location in the circular buffer has been written to.
ACE_RT_OPT_OVR_MBIT	Enabling this option will cause a mode code Command Word with a T/R* bit of 0 and an MSB of the mode code field of 0 to be considered a defined (reserved) mode Command Word. The DDC hardware will respond to such a command and the Message Error bit will not become set.
ACE_RT_OPT_ALT_STS	Enabling this option will cause all 11 RT Status Word bits to be under control of the user via the **aceRTStatusBitsSet()** and **aceRTStatusBitsClear()** functions.
ACE_RT_OPT_IL_RX_D	Enabling this option will cause the device to not store the received data words if the DDC hardware device receives a receive command that has been illegalized.

Table 55. RT Configuration Options

Options	Description
ACE_RT_OPT_BSY_RX_D	If a particular Command Word (broadcast, T/R* bit, subaddress) has been programmed to be busy by means of the Busy lookup table and the RT receives a receive command, the 1553 device will respond with its Status Word with the Busy bit set and will not store the received Data Words. See Section 0 on modifying the Busy Lookup Table.
ACE_RT_OPT_SET_RTFG	If enabled, the Terminal flag status word bit will also become set if either a transmitter timeout (660.5 µs) condition had occurred or the ACE RT had failed its loopback test for the previous non-broadcast message. The loopback test is performed on all non-broadcast messages processed by the RT. The received version of all transmitted words is checked for validity (sync and data encoding, bit count, parity) and correct sync type. In addition, a 16-bit comparison is performed on the received version of the last word transmitted by the RT. If any of these checks or comparisons do not verify, the loopback test is considered to have failed.
ACE_RT_OPT_1553A_MC	This option causes the RT to consider only subaddress 0 to be a mode code subaddress. Subaddress 31 is treated as a standard non-mode code subaddress. In this configuration, the 1553 hardware will consider valid and respond only to mode code commands containing no data words. In this configuration, the RT will consider all mode commands followed by data words to be invalid and will not respond. In addition the 1553 hardware will not decode for the MIL-STD-1553B "Transmit Status" and "Transmit Last Command" mode codes. As a result, the internal RT Status Word Register will be updated as a result of these commands.
ACE_RT_OPT_MC_O_BSY	If a particular Command Word (broadcast, T/R* bit, subaddress) has been programmed to be busy by means of the Busy Lookup Table, the 1553 hardware will transmit its Status Word with its BUSY bit set, followed by a single Data Word, in response to either a Transmit Vector Word mode command or a Reserved transmit mode command with data (transmit mode codes 10110 through 11111). See Section 0 on modifying the Busy Lookup Table.
ACE_RT_OPT_BCST_DIS	The 1553 hardware will **not** recognize RT address 31 as the broadcast address. In this instance, RT address 31 may be used as a discrete RT address.

3.3.4.1.1 Setting the RT Address

The DDC device’s RT address on the 1553 bus can come from two sources; Externally via specific pins on the DDC hardware or programmatically (Internally) via an API function. The **aceRTSetAddrSource()** function can be used to set the address source. If the RT address is to be programmed by the user, the **aceRTSetAddress()** function should be used.

> ***Note:*** *If using an “External” RT Address Source, the RT Address can be reached via the **aceRTRelatchAddr()** function.*

```
S16BIT nResult;
#define RT_ADDRESS 10

/* Set RT Address Source */
nResult =  aceRTSetAddrSource(
    0,                          /* LDN */
    ACE_RT_INTERNAL_ADDR);      /* Use Internal Address Src */

if(nResult)
    printf(“aceRTSetAddrSource Error: Code %d\n”, nResult);

/* Set RT Address Value */
nResult =  aceRTSetAddress(
    0,                          /* LDN */
    RT_ADDRESS);                /* Internal Address Value */

if(nResult)
    printf(“aceRTSetAddress Error: Code %d\n”, nResult);
```

Code Example 75. Setting the RT Address Source and Value

3.3.4.2 RT Lookup Tables

The AceXtreme C SDK RT architecture contains a number of lookup tables to store specific user choices with regards to how the RT will respond on the 1553 bus. The tables’ functions are described in Sections 0, 3.3.4.2.2, and 0, including information on how to configure them.

3.3.4.2.1 Busy Bit Table

The Busy Bit Table holds information on which subaddresses will respond with the Busy Bit (Bit 3) set in the Status Word. The table is based on address (own address or broadcast) and command direction (RT Transmit/Receive). By default, the Busy Bit will be OFF for all subaddresses.

> ***Note:*** *Entries in the Busy Bit Lookup Table can be cleared and queried using the* ***aceRTBusyBitsTblClear()*** *and* ***aceRTBusyBitsTblStatus()*** *functions.*

Table 56. Busy Bit Lookup Table

Own Addr / Broadcast	Command Direction	Bus Subaddresses
OWN (1)	TX (1)	Each binary bit corresponds to Subaddress (i.e. 0x00000100 is Subaddress 8)
OWN (1)	RX (0)	Each binary bit corresponds to Subaddress (i.e. 0x00000100 is Subaddress 8)
BCST (0)	RX (0)	Each binary bit corresponds to Subaddress (i.e. 0x00000100 is Subaddress 8)

```
S16BIT nResult;

/* Set Busy Bit (OWN Address  (1) , TX (1) , SA19) */
nResult =  aceRTBusyBitsTblSet(
    0,              /* LDN */
    1,              /* Own Address (not BCAST) */
    1,              /* Transmit Cmds */
    ACE_RT_SA19);   /* Subaddress 19 */

if(nResult)
   printf(“aceRTBusyBitsTblSet Error: Code %d\n”, nResult);
```

Code Example 76. Setting the Busy Bit for all TRANSMIT Commands to SA19 (“Own Addr”, TX, SA 19)

3.3.4.2.2 Status Word Table

The Status Word Table holds information on what bits are active (set) in the RT Status Word. By default, the “Standard” 1553 Status Word is used allowing configuration of optional Status Word bits. Alternatively, an “Alternate Status Word” can be defined (See **ACE_RT_OPT_ALT_STS** option in **aceRTConfigure()**). Using the Alternate Status word will allow the user to control bits 0-10 of the RT Status Word.

Note: *When using the "Standard" Status Word, some bits are internally set and clear by the DDC Hardware and cannot be configured via the API.*

Note: *Entries in the Status Word Lookup Table can be cleared and queried using the **aceRTStatusBitsClear()** and **aceRTStatusBitsStatus()** functions.*

Table 57. Standard RT Status Word

BIT	DESCRIPTION
15 (MSB)	Remote Terminal Address Bit 4
14	Remote Terminal Address Bit 3
13	Remote Terminal Address Bit 2
12	Remote Terminal Address Bit 1
11	Remote Terminal Address Bit 0
10	Message Error
9	Instrumentation
8	Service Request
7	Reserved
6	Reserved
5	Reserved
4	Broadcast Command Received
3	Busy
2	Subsystem Flag
1	Dynamic Bus Control Acceptance
0 (LSB)	Terminal Flag

```
S16BIT nResult;

/* Set Service Request Status Bit */
nResult =  aceRTStatusBitsSet(
    0,                      /* LDN */
    ACE_RT_STSBIT_SREQ);   /* Set "Service Reg" bit */

if(nResult)
    printf("aceRTStatusBitsSet Error: Code %d\n", nResult);
```

Code Example 77. Setting the "Service Request" Bit in the RT Status Word

3.3.4.2.3 Built-in-Test (BIT) Word Table

The BIT Word is maintained by the RT device to store advanced error information. The internal contents of the BIT data word are provided to supplement the appropriate bits already available in the RT Status Word. The BIT Word definition can be the standard Internal BIT Data Word (default). Alternatively, the user can define a custom BIT Word for unique applications.

Whether internal or external, the BIT Word can be read via the **aceRTBITWrdRead()** function.

Table 58. Internal Built-in-Test (BIT) Data Word

BIT	DESCRIPTION
15 (MSB)	Transmitter Timeout
14	Loop Test Failure B
13	Loop Test Failure A
12	Handshake Failure
11	Transmitter Shutdown B
10	Transmitter Shutdown A
9	Terminal Flag Inhibited
8	BIT Test Fail
7	High Word Count
6	Low Word Count
5	Incorrect Sync Received
4	Parity/Manchester Error Received
3	RT-RT Gap/Sync/Address Error
2	RT-RT No Response Error
1	RT-RT 2nd Command Word Error
0 (LSB)	Command Word Contents Error

```
S16BIT nResult;
U16BIT nBITWord;

/* Configure RT BIT Word */
nResult =  aceRTBITWrdConfig(
    0,                    /* LDN */
    ACE_RT_BIT_INTERNAL,  /* Use Internal BIT */
    0);                   /* Reserved */

if(nResult)
    printf(“aceRTBITWrdConfig Error: Code %d\n”, nResult);

/* Read RT BIT Word */
nResult =  aceRTBITWrdRead(
    0,                    /* LDN */
    ACE_RT_BIT_INTERNAL,  /* Read from Internal BIT */
    &nBITWord);           /* Bit Word Storage */

if(nResult)
    printf(“aceRTBITWrdRead Error: Code %d\n”, nResult);
```

Code Example 78. Configure and Read the Internal BIT Word

Note: *If using an “Externally Supplied” BIT Word, the function* ***aceRTBITWrdWrite()*** *is used to set the BIT Word value.*

3.3.4.3 RT Data Blocks

RT Data Blocks are used to store MIL-STD-1553 data words being received or to be transmitted by the Remote Terminal. Once created, a RT Data Block can be independently read or written to. In order to use an RT Data Block it must be linked to a specific RT Subaddress. When a 1553 message involving that Subaddress is sent on the bus, the RT Engine will place or pull data from the linked RT Data Block.

3.3.4.3.1 RT Data Block Types

RT Data Blocks come in many different types and sizes for use in different applications. Each type deals with 1553 data differently and may require additional control. For information on reading RT Data Blocks, see Section 3.3.4.6.1.

The AceXtreme C SDK provides a number of memory management schemes for RT Data Block Objects. The choice of scheme is fully programmable on a transmit/receive/broadcast basis. Schemes available include a single message mode, a circular buffer mode to support bulk data transfers and a double buffering mode for individual receive subaddresses to ensure data consistency. Circular buffers may be allocated on an individual transmit and/or receive subaddress basis. In addition, there is a global circular buffer option, by which data words received to any subset of subaddresses (or all subaddresses) may be stored to a single circular buffer.

Note: *Creation (Allocation) of RT Data Blocks may be limited based on DDC hardware memory.*

Table 59. RT Data Block Types

RT Data Block Type	Description
ACE_RT_DBLK_SINGLE	Single-Buffered Data Block (32 Words)
ACE_RT_DBLK_DOUBLE	Double-Buffered Data Block (64 Words)
ACE_RT_DBLK_C_128	Circular-Buffered Data Block (128 Words)
ACE_RT_DBLK_C_256	Circular-Buffered Data Block (256 Words)
ACE_RT_DBLK_C_512	Circular-Buffered Data Block (512 Words)
ACE_RT_DBLK_C_1K	Circular-Buffered Data Block (1K Words)
ACE_RT_DBLK_C_2K	Circular-Buffered Data Block (2K Words)
ACE_RT_DBLK_C_4K	Circular-Buffered Data Block (4K Words)
ACE_RT_DBLK_C_8K	Circular-Buffered Data Block (8K Words)
ACE_RT_DBLK_GBL_C_128	Global Circular-Buffered Data Block (128 Words)
ACE_RT_DBLK_GBL_C_256	Global Circular-Buffered Data Block (256 Words)
ACE_RT_DBLK_GBL_C_512	Global Circular-Buffered Data Block (512 Words)
ACE_RT_DBLK_GBL_C_1K	Global Circular-Buffered Data Block (1K Words)
ACE_RT_DBLK_GBL_C_2K	Global Circular-Buffered Data Block (2K Words)
ACE_RT_DBLK_GBL_C_4K	Global Circular-Buffered Data Block (4K Words)
ACE_RT_DBLK_GBL_C_8K	Global Circular-Buffered Data Block (8K Words)

3.3.4.3.1.1Single-Buffered RT Data Block

Single-Buffered RT Data Blocks allocate a 32-word buffer for 1553 data. This Data Block type can be used for all RT Transmit, Receive, and Broadcast commands. Based on the SA Mapping Table, any 1553 data involving that Subaddress will be stored in the destined RT Data Block.

Note: *If a new 1553 Message destined for a Single-Buffered Data Block is received before data is consumed by the user, the data will be automatically overwritten.*

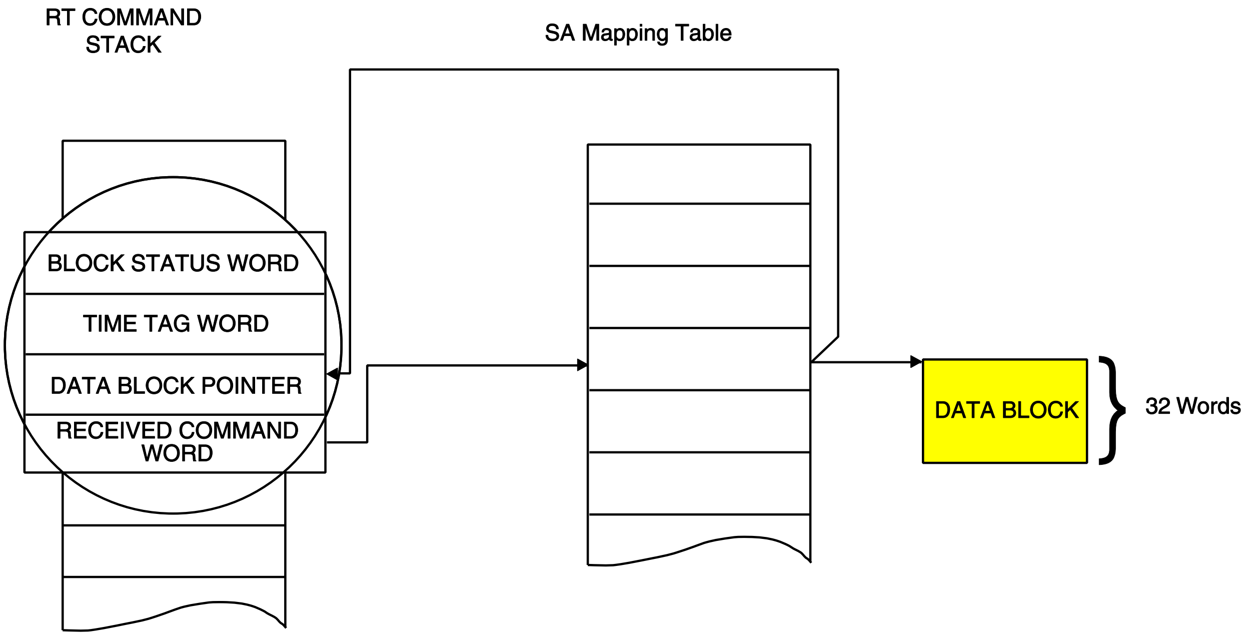


Figure 30. Single-Buffer RT Data Block Storage

3.3.4.3.1.2 Double-Buffered RT Data Block

Double-Buffered RT Data Blocks allocate two 32-word buffers for 1553 data. This Data Block type can be used for all RT receive and Broadcast commands. Based on the SA Mapping Table, any 1553 data involving that Subaddress will be stored in the two contiguous RT Data Blocks in an alternating fashion defined as follows.

One of the 32-word buffers will be designated as the active RT Data Block while the other will be considered inactive. The data accompanying the next receive command to that subaddress will be stored in the active Data Block. Upon completion of the message (if valid), the DDC Hardware will automatically switch the active and inactive blocks for that subaddress. This means that the latest, valid, complete data block is always readily available to the user.

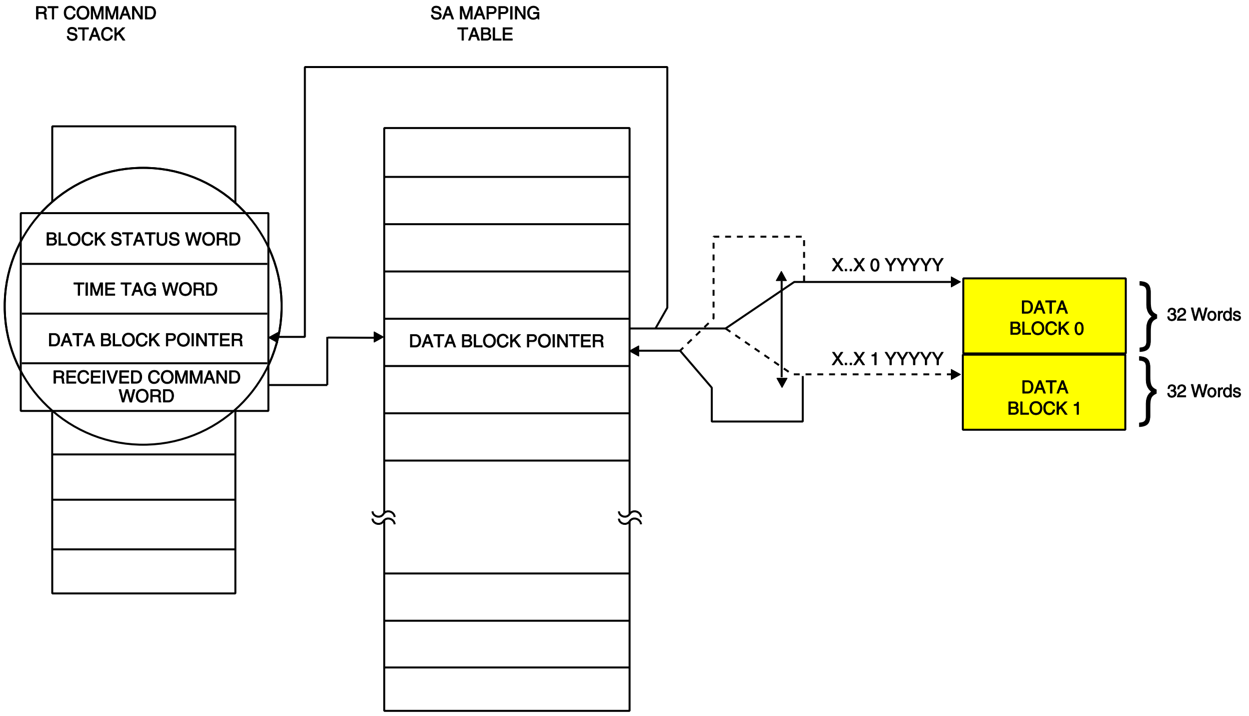


Figure 31. Double-Buffered RT Data Block Storage

3.3.4.3.1.3Circular-Buffered RT Data Blocks

Circular-Buffered RT Data Blocks allocate a contiguous buffer that will store 1553 data. When the buffer is filled up, the DDC Hardware will jump back to the start of the buffer, thus making it circular. This Data Block type can be used for all RT Transmit, Receive, and Broadcast commands. Based on the SA Mapping Table, any 1553 data involving that Subaddress will be stored in the next contiguous location in the Circular Buffer. Circular Buffers can be created in the following lengths: 128, 256, 512, 1K, 2K, 4K, 8K words.

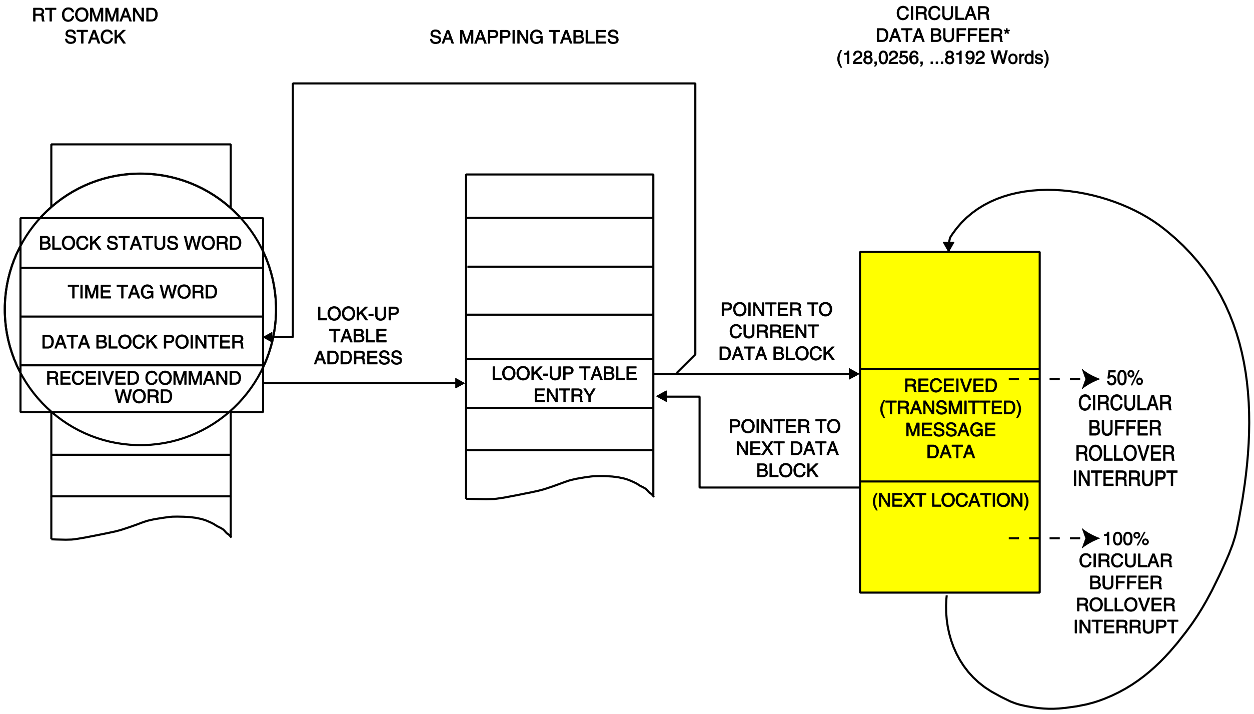


Figure 32. Circular-Buffered RT Data Block Storage

3.3.4.3.1.4Global Circular-Buffered RT Data Block

The final RT Data Block type is the Global Circular-Buffered RT Data Block. This type is unique, in that only **one** of this type can be created per Remote Terminal (device). This buffer operates in the same fashion as standard Circular-Buffered Data Blocks (see Section 3.3.4.3.1.3), except for one key difference: the Global Circular-Buffered RT Data Block can be linked to more than one Subaddress. This allows one contiguous buffer to store 1553 data for an arbitrary group of transmit, receive and broadcast messages. The Global Circular-Buffered RT Data Block can be created in the following lengths: 128, 256, 512, 1K, 2K, 4K, 8K words.

3.3.4.3.2 Creating an RT Data Block

Once an RT Data Block Type has been chosen, the RT Data Block can be simply created using the **aceRTDataBlkCreate()** function. If the initial data is available, it can be placed into the data block.

```
S16BIT nResult;
#define RTDBLK1 0x0001
U16BIT wBuffer[32] =
  {0x1111,0x2222,0x3333,0x4444,0x1111,0x2222,0x3333,0x4444,
  0x1111,0x2222,0x3333,0x4444,0x1111,0x2222,0x3333,0x4444,
  0x1111,0x2222,0x3333,0x4444,0x1111,0x2222,0x3333,0x4444,
  0x1111,0x2222,0x3333,0x4444,0x1111,0x2222,0x3333,0x4444
  };


/* Create an RT Data Block */
nResult =  aceRTDataBlkCreate(
    0,                        /* LDN */
    RTDBLK1,                  /* Data Block OUID to assign */
    ACE_RT_DBLK_SINGLE,       /* Use Single-Buffering */
    wBuffer,                  /* Initial Data */
    32);                      /* Size of Initial Data */

if(nResult)
    printf(“aceRTDatBlkCreate Error: Code %d\n”, nResult);
```

Code Example 79. Creating a Single-Buffered RT Data Block

3.3.4.3.3 Mapping an RT Data Block to a RT Subaddress

Once an RT Data Block (see Section 3.3.4.3.2) has been created, it needs to be linked to an RT Subaddress to identify which 1553 message data will be stored in which RT Data Blocks. Except for the Global Circular-Buffered RT Data Block (Section 3.3.4.3.1.4), only one Data Block can be linked to each Subaddress/Message Type combination. Applicable Message Types are **ACE_RT_MSGTYPE_RX** (Receive), **ACE_RT_MSGTYPE_TX** (Transmit), and **ACE_RT_MSGTYPE_BCST** (Broadcast).

3.3.4.4 Subaddress Interrupt Options

The Fifth Parameter of **aceRTDataBlkMapToSA()** defines interrupt options for this Subaddress/Message Type combination. The following options can be “logically OR’ed” into the Third parameter (wIrqOptions) of **aceRTDataBlkMapToSA()**. See Section 3.2.3 on how to capture Device Interrupt Events.

Interrupt Option	Description (if Enabled)
ACE_RT_DBLK_EOM_IRQ	The ACE_IMR1_RT_SUBADDR_EOM Device Interrupt Event will be generated when this Data Block receives 1553 message data.
ACE_RT_DBLK_CIRC_IRQ	The ACE_IMR1_RT_CIRCBUF_ROVER Device Interrupt Event will be generated when this Circular-Buffered Data Block rolls over (100%). The ACE_IMR2_RT_CIRC_50P_ROVER Device Interrupt Event will be generated when this Circular-Buffered Data Block reaches 50%.

***Note:** If the Sixth Parameter of **aceRTDataBlkMapToSA()** is TRUE, the specified Message Type/ Subaddress combination will be marked “Legal” in the Command Legalization Table (see Section 3.3.4.7).*

```
S16BIT nResult;
#define RTDBLK1 0x0001

/* Map RTDBLK1 to SA 19 */
nResult =  aceRTDataBlkMapToSA(
    0,                       /* LDN */
    RTDBLK1,                 /* Data Block OUID to map */
    19,                      /* Subaddress to map */
    ACE_RT_MSGTYPE_RX |      /* Map for RX Messages */
    ACE_RT_MSGTYPE_TX,       /* Map for TX Messages */
    0,                       /* Interrupt Mask */
    TRUE);                   /* “Legalize” above commands */

if(nResult)
    printf(“aceRTDatBlkMapToSA Error: Code %d\n”, nResult);
```

Code Example 80. Mapping RTDBLK1 to TX ad RX messages for Subaddress 19

3.3.4.5 Activating the Remote Terminal

Once the Remote Terminal is configured and the RT Lookup Tables have been setup, the RT is ready to begin responding to 1553 bus traffic.

3.3.4.5.1.1Starting and Stopping

The Remote Terminal can be started and stopped dynamically by the user. Starting and Stopping is accomplished by the **aceRTStart()** and **aceRTStop()** functions.

> ***Note:*** *By stopping the Remote Terminal with **aceRTStop()**, any traffic that has not been consumed by the user will be discarded.*

```
S16BIT nResult;

/* Start the Remote Terminal */
nResult =  aceRTStart(0);         /* LDN */

if(nResult)
    printf(“aceRTStart Error: Code %d\n”, nResult);
```

Code Example 81. Starting the Remote Terminal (RT)

```
S16BIT nResult;

/* Stop the Remote Terminal (RT) */
nResult =  aceRTStop(0);          /* LDN */

if(nResult)
    printf(“aceRTStop Error: Code %d\n”, nResult);
```

Code Example 82. Stopping the Remote Terminal (RT)

3.3.4.6 Consuming Data

The Remote Terminal supports two methods of consuming monitored data: Command Stack/Data Block Access and Host Buffer. Each method has particular advantages based on the user’s Application needs.

3.3.4.6.1 Data via Command Stack

For Applications that have strict timing requirements and need quick access to 1553 RT data, the user can read 1553 messages directly off of the DDC hardware. This can be accomplished using the **aceRTGetStkMsgsRaw()** and **aceRTGetStkMsgDecoded()** functions.

3.3.4.6.1.1 Reading the Stack

The RT Command Stack is the lowest level of data storage. It consists of command stack holding message/routing information with a link to an RT Data Block holding 1553 data words. The stack will store "Legal" 1553 command data until full and then will start overwriting the oldest data. It is the user's responsibility to read entries from the Command Stack before an overrun occurs.

Note: *The Stack Access functions **aceRTGetStkMsgsRaw()** and **aceRTGetStkMsgDecoded()** will return 1553 command information, as well as 1553 data words.*

Messages can be read off of the Command Stack in two formats: Raw or Decoded. Depending on which method is used, Messages taken off the stack will be returned in FIFO order or LIFO order.

3.3.4.6.1.1.1 RAW FORMAT

The Raw Format will return a U16BIT pointer to the binary data. Using this method will also allow more than one message to be read off the stack at one time. Each message will be fixed-length of 36 words and will use zero fill words for messages not meeting the required word maximum. For example, if two messages have been monitored, the binary data will be 72 words deep, with the second message starting at offset 36.

Table 60. RT Command Stack Raw Format for One Message

Word	**MSB**		**LSB**
Bits	**15**	**7**	**0**
0	Block Status Word		
1	Time Tag Word		
2	Bit [15] EOM	Bits [14:8] Data Length (in Words)	Bits [7:0] 1553 Message Type
3	1553 Command Word		
4	Data Word 0		
5	Data Word 1		
n+4	Data Word n		

Messages can be read from the RT Command stack in Raw Format using the **aceRTGetStkMsgsRaw()** function. The function will return up to "*wBufferSize*" words or all messages, whichever is smaller. The **Return Value** of the function will inform the user of the number of messages returned or if an error occurred.

***Note**: Each message is a fixed length of 36 words.*

```
S16BIT nResult;
U16BIT wBuffer[400] = { 0x00000000 };

/* Get Raw Messages from the RT Command Stack */
nResult =  aceRTGetStkMsgsRaw(
    0,                /* LDN */
    wBuffer,          /* Buffer Storage */
    400);             /* Max Size of Buffer */

if(nResult)
    printf(“aceRTGetStkMsgsRaw() Error: Code %d\n”, nResult);
```

Code Example 83. Reading Raw Data from the RT Command Stack

3.3.4.6.1.1.2 DECODED FORMAT

The Decoded Format will read one message off of the RT Command Stack and decode it into a MSGSTRUCT structure object. In addition, the user can decide whether to read the oldest (next) or latest message and whether or not to remove (purge) the message from the Stack.

A Message can be read from the Command Stack in Decoded Format using the **aceRTGetStkMsgDecoded()** function. The function will return one message decoded into the "pMsg" MSGSTRUCT variable. The "wMsgLoc" variable is used to define which message to read and whether or not to remove it from the Stack.

Stack Message Location and Purge Options (wMsgLoc)	
ACE_MT_MSGLOC_NEXT_PURGE	Reads next message and takes it off of the stack
ACE_MT_MSGLOC_NEXT_NPURGE	Reads next message and leaves it on the stack
ACE_MT_MSGLOC_LATEST_PURGE	Reads current message and takes it off of the stack
ACE_MT_MSGLOC_LATEST_NPURGE	Reads current message and leaves it on the stack

```
/*Global (used for all modes) Message Structure for decoded 1553 msgs */
typedef struct MSGSTRUCT
{
  U16BIT wTYPE;                /* Contains the msg type (see above) */
  U16BIT wBlkSts;              /* Contains the block status word */
  U16BIT wTimeTag;             /* Time Tag of message */
  U16BIT wCmdWrd1;             /* First command word */
  U16BIT wCmdWrd2;             /* Second command word (RT to RT) */
  U16BIT wCmdWrd1Flg;          /* Is command word 1 valid? */
  U16BIT wCmdWrd2Flg;          /* Is command word 2 valid? */
  U16BIT wStsWrd1;             /* First status word */
  U16BIT wStsWrd2;             /* Second status word */
  U16BIT wStsWrd1Flg;          /* Is status word 1 valid? */
  U16BIT wStsWrd2Flg;          /* Is status word 2 valid? */
  U16BIT wWordCount;           /* Number of valid data words */
  U16BIT aDataWrds[32];        /* An array of data words */

/* The following are only applicable in BC mode */
  U16BIT wBCCtrlWrd;           /* Contains the BC control word */
  U16BIT wBCGapTime;           /* Message gap time word */
  U16BIT wBCLoopBack1;         /* First looped back word */
  U16BIT wTimeTag2;            /* wBCLoopBack2 is redefined as TimeTag2 */
  U16BIT wBCLoopback1Flg;      /* Is loopback 1 valid? */
  U16BIT wTimeTag3;            /* wBCLoopBack2Flg is redefined as TimeTag3 */
}MSGSTRUCT;
```

Figure 33. RT Command Stack MSGSTRUCT Object Definition

```
S16BIT nResult;
MSGSTRUCT sMsg;

/* Get a Decoded Message from the RT Command Stack */
nResult =  aceRTGetStkMsgDecoded(
    0,                             /* LDN */
    &sMsg,                         /* Message Storage */
    ACE_RT_MSGLOC_NEXT_PURGE);     /* Read and Purge Msg */

if(nResult)
    printf(“aceRTGetStkMsgDecoded Error: Code %d\n”,nResult);
```

Code Example 84. Reading a Decoded Message from the RT Command Stack

3.3.4.6.2 RT Block Status Word

The Block Status Word (BSW) is used to identify the health of the message. The BSW contains information regarding the message, specifying whether the message is in progress or has been completed, what channel the message was processed on, and whether or not there were any errors in the message table. The RT Block status word's bits are defined in

Table 61. RT Block Status Word

Bit	Description	
15 (MSB)	EOM	Set at the completion of a BC message, regardless of whether or not there were any errors in the message.
14	SOM	Set at the start of a BC message and cleared at the end of the message.
13	A/B CHANNEL	This bit will be low if the message was processed on Channel A or high if the message was processed on Channel B
12	ERROR FLAG	If this bit is high, one or more of bits 10, 9, and/or 8 are also set high.
11	RT-RT FORMAT	This bit is set when the device is the receiving RT in an RT-to-RT command.
10	FORMAT ERROR	If set, indicates the received portion of a message contained one or more violations of the 1553 message validation criteria (sync, encoding, parity, bit count, word count, etc.), or the RT's status word received from a responding RT contained an incorrect RT address field.
9	NO RESPONSE TIMEOUT	If set, indicates that an RT has either not responded or has responded later than the BC No Response Timeout time.
8	LOOP TEST FAIL	A loopback test is performed on the transmitted portion of every message in BC mode. A validity check is performed on the received version of every word transmitted by the BC. In addition, a bit-by-bit comparison is performed on the last word transmitted by the BC for each message. If either the received version of any transmitted word is invalid (sync, encoding, bit count, and/or parity error) and/or the received version of the last word transmitted by the BC does not match the transmitted version, the LOOP TEST FAIL bit will be set.
7	CIRCULAR BUFFER ROLLOVER	This bit will be set if the lookup table address pointer crossed the upper boundary of its circular buffer, resulting in a rollover.
6	ILLEGAL COMMAND WORD	If this bit is set, it indicates that the message has been illegalized.
5	WORD COUNT ERROR	Indicates the BC did not transmit the correct number of Data Words.
4	INCORRECT DATA SYNC	Indicates the BC transmitted a Command sync in a Data Word.
3	INVALID WORD	Indicates the BC (or transmitting RT in an RT-to-RT transfer) transmitted with one or more words containing one or more of the following error types: sync field error, Manchester encoding error, parity error, and/or bit count error.
2	RT-RT GAP / SYNC ADDRESS ERROR	This bit is set if the RT is the receiving RT for an RT-to-RT transfer and one or more of the following occur: 4. The GAP CHECK ENABLED bit is set to logic "1" and the transmitting RT responds with a response time of less than

Table 61. RT Block Status Word

Bit	Description	
		4 µs. 5. There is an incorrect sync type or format error (encoding, bit count, and/or parity error) in the transmitting RT Status Word. 6. The RT address field of the transmitting RT Status Word does not match the RT address in the transmit Command Word.
1	RT-RT 2ND COMMAND ERROR	If the device is the receiving RT for an RT-to-RT transfer, this bit set indicates one or more of the following error conditions in the transmit Command Word: 4. T/R bit = logic "0" 5. Subaddress = 00000 or 11111 6. Same RT Address field as the receive Command Word.
0 (LSB)	COMMAND WORD CONTENTS ERROR	This bit indicates a received command word is not defined in accordance with MIL-STD-1553B. This includes the following undefined Command Words: 4. BROADCAST DISABLED **and** the Command Word is a non-mode code, broadcast, transmit command. 5. The OVERRIDE MODE T/R ERROR bit is logic "0" **and** a message with a T/R bit of "0," a subaddress/mode field of 00000 or 11111 and a mode code field between 00000 and 01111. 6. BROADCAST DISABLED **and** a mode code command that is not permitted to broadcast (e.g.. Transmit status) is sent to the broadcast address (11111).

3.3.4.6.3 Data via RT Data Block

3.3.4.6.3.1 Reading and Writing Individual RT Data Blocks

In addition to reading RT Command Stack, any defined RT Data Block can be read or written to asynchronously by the user via the **aceRTDataBlkRead()** and **aceRTDataBlkWrite()** functions. See Section 3.3.4.3.1 on RT Data Block Types.

3.3.4.6.3.1.1 Single-Buffered and Double-Buffered RT Data Blocks

Single-Buffered and Double-Buffered Data Blocks will return the inactive 1553 data to the user. Returning the inactive data guarantees that the data is complete and does not contain any partial from an active message on the 1553 bus.

Note: *The "wBufferSize" variable should not exceed 32 words.*

```
S16BIT nResult;
U16BIT wData[32];
#define RTDBLK1 0x0001 /* Single-Buffered RT Data Block */

/* Read an RT Data Block */
nResult =  aceRTDataBlkRead(
    0,          /* LDN */
    RTDBLK1,    /* Data Block OUID to read */
    wData,      /* Buffer Storage */
    32,         /* Number of Words to read */
    0);         /* Offset into Data Block */

if(nResult)
    printf(“aceRTDataBlkRead Error: Code %d\n”, nResult);
```

Code Example 85. Reading a “Single-Buffered” Data Block

3.3.4.6.3.1.2 Circular-Buffered RT Data Blocks

Reading and Writing to Circular-Buffered RT Data Blocks (including the Global Circular-Buffered Data Block) requires knowledge of where the Data Pointers are within the Circular Buffer. There are two Data Pointers: “pUserRWOffset”, which defines the last location read/written by the user and “pAceRWOffset”, which defines the last location read/written by DDC Hardware. These pointers can be used to determine the right offset into the Circular Buffer to read or write data. Circular Buffer Data Pointers can be retrieved via the **aceRTDataBlkCircBufInfo()** function.

```
S16BIT nResult;
U16BIT wData[32];
U16BIT nUserRWOffset, nAceRWOffset;
#define RTDBLK1 0x0001 /* Circular-Buffered RT Data Block */

/* Get Circular Buffer Data Pointers */
nResult = aceRTDataBlkCircBufInfo(
    0,                 /* LDN */
    RTDBLK1,           /* OUID of Data Block to query */
    &nUserRWOffset,    /* Current User Offset */
    &nAceRWOffset)     /* Current Hardware Offset */

if(nResult)
    printf(“aceRTDataBlkCircBufInfo Error: %d\n”, nResult);

/* Read an RT Data Block */
nResult =  aceRTDataBlkRead(
    0,                 /* LDN */
    RTDBLK1,           /* OUID of Data Block to read */
    wData,             /* Buffer Storage */
    32,                /* Number of Stat Words to Read */
    nAceRWOffset);     /* Offset into Data Block */

if(nResult)
    printf(“aceRTDataBlkRead Error: %d\n”, nResult);
```

Code Example 86. Reading a “Circular-Buffered” Data Block

3.3.4.6.4 Data via Host Buffer

The RT Host Buffer (HBUF) is a circular memory buffer resident on the host that contains the log of all messages involving the configured RT address, in the order they appeared on the 1553 bus.

One advantage of using a Host Buffer is that all messages are automatically transferred to the HBUF by means of internally configured interrupt events. This will make sure that RT data is removed from DDC hardware and placed into the Host Buffer before any data loss can occur.

Another advantage is that the size of the host buffer can be fairly large and can serve as an elasticity buffer for applications that cannot consume data at a high rate.

3.3.4.6.4.1 Installing the Host Buffer

The Host Buffer should be installed before RT is active on the 1553 bus (See **aceRTStart()**)

> ***Note:*** *The Host Buffer size should be typically be 4-5 times larger than the maximum capacity of the DDC hardware (RT Command Stack).*

The following equation can be used to calculate the correct Host Buffer size:

HBUFSIZE = (CMD_STACK_SIZE * ACE_MSGSIZE_RT) * 4

```
S16BIT nResult;
#define CMD_STK_SIZE = 512

/* Setup the RT Host Buffer */
nResult =  aceRTInstallHBuf(
    0,                                        /* LDN */
    (CMD_STK_SIZE * ACE_MSGSIZE_RT) * 4));  /* Hbuf size */

if(nResult)
    printf("aceRTInstallHBuf Error: Code %d\n", nResult);
```

Code Example 87. Installing the RT Host Buffer

3.3.4.6.4.2 Reading the Host Buffer

The Host Buffer architecture is designed to automatically remove data from the DDC hardware and place it into the host-allocated Host Buffer. It is the user's responsibility to read entries from the Host Buffer for consumption.

Messages can be read off of the Host Buffer in two formats: Raw or Decoded. Depending on which method is used, Messages taken off of the Host Buffer will be returned in FIFO order or LIFO order.

3.3.4.6.4.2.1 RAW FORMAT

The Raw Format will return a U16BIT pointer to the binary data. This method will also allow more than one message to be read off of the Host Buffer at one time. Each message will be fixed-length of 36 words and will use zero fill words for messages not required the word maximum. For example, if two Messages have been monitored, the binary data will be 72 words deep, with the second message starting at offset 36.

Table 62. RT Host Buffer Raw Format for One RT Message

Word	MSB		LSB
Bits	**15**	**7**	**0**
0	Block Status Word		
1	Time Tag Word		
2	Bit [15] EOM	Bits [14:8] Data Length (in Words)	Bits [7:0] 1553 Message Type
3	1553 Command Word		
4	Data Word 0		
5	Data Word 1		
n+4	Data Word n		

Messages can be read from the Host Buffer in Raw Format using the **aceRTGetHBufMsgsRaw()** function. The function will return up to “*wBufferSize*” words or all messages, whichever is smaller. The “pdwMsgCount” pointer will inform the user of the number of messages returned.

Note: *Each message is a fixed length of 36 words.*

```
S16BIT nResult;
U32BIT dwStkLost, dwHBufLost, dwMsgCount;
U16BIT wBuffer[400] = { 0x00000000 };

/* Get Raw Messages from the Host Buffer */
nResult =  aceRTGetHBufMsgsRaw(
    0,              /* LDN */
    wBuffer,        /* Buffer Storage */
    400,            /* Max size of Buffer */
    &dwMsgCount,    /* Number of Msgs read */
    &dwStkLost,     /* Lost Messages (Stack) */
    &dwHBufLost);   /* Lost Messages (Hbuf) */

if(nResult)
    printf(“aceRTInstallHBuf Error: Code %d\n”, nResult);
```

Code Example 88. Reading Raw Data From the Host Buffer

3.3.4.6.4.2.2 DECODED FORMAT

The Decoded Format will read one message off the Host Buffer and decode it into a MSGSTRUCT structure object. In addition, the user can decide whether to read the oldest (next) or latest message and whether or not to remove (purge) the message from the Host Buffer.

A message can be read from the Host Buffer in Decoded Format using the **aceRTGetHBufMsgDecoded()** function. The function will return one message decoded into the “pMsg” MSGSTRUCT variable. The “wMsgLoc” variable is used to define which message to read and whether or not to remove it from the Host Buffer.

Host Buffer Message Location and Purge Options (wMsgLoc)	
ACE_RT_MSGLOC_NEXT_PURGE	Reads next message and takes it off of the host buffer
ACE_RT_MSGLOC_NEXT_NPURGE	Reads next message and leaves it on the host buffer
ACE_RT_MSGLOC_LATEST_PURGE	Reads current message and takes it off of the host buffer
ACE_RT_MSGLOC_LATEST_NPURGE	Reads current message and leaves it on the host buffer

```
/*Global (used for all modes) Message Structure for decoded 1553 msgs */
typedef struct MSGSTRUCT
{
  U16BIT wTYPE;                /* Contains the msg type (see above) */
  U16BIT wBlkSts;              /* Contains the block status word */
  U16BIT wTimeTag;             /* Time Tag of message */
  U16BIT wCmdWrd1;             /* First command word */
  U16BIT wCmdWrd2;             /* Second command word (RT to RT) */
  U16BIT wCmdWrd1Flg;          /* Is command word 1 valid? */
  U16BIT wCmdWrd2Flg;          /* Is command word 2 valid? */
  U16BIT wStsWrd1;             /* First status word */
  U16BIT wStsWrd2;             /* Second status word */
  U16BIT wStsWrd1Flg;          /* Is status word 1 valid? */
  U16BIT wStsWrd2Flg;          /* Is status word 2 valid? */
  U16BIT wWordCount;           /* Number of valid data words */
  U16BIT aDataWrds[32];        /* An array of data words */

/* The following are only applicable in BC mode */
  U16BIT wBCCtrlWrd;           /* Contains the BC control word */
  U16BIT wBCGapTime;           /* Message gap time word */
  U16BIT wBCLoopBack1;         /* First looped back word */
  U16BIT wTimeTag2;            /* wBCLoopBack2 is redefined as TimeTag2 */
  U16BIT wBCLoopback1Flg;      /* Is loopback 1 valid? */
  U16BIT wTimeTag3;            /* wBCLoopBack2Flg is redefined as TimeTag3 */
}MSGSTRUCT;
```

Figure 34. RT Data Block MSGSTRUCT Object Definition

```
S16BIT nResult;
U32BIT dwStkLost, dwHBufLost, dwMsgCount;
MSGSTRUCT sMsg;

/* Get a Decoded Message from the Host Buffer */
nResult =  aceRTGetHBufMsgDecoded(
    0,                           /* LDN */
    &sMsg,                       /* Message Storage */
    &dwMsgCount,                 /* Number of Msgs read */
    &dwStkLost,                  /* Lost Msgs (stack) */
    &dwHBufLost,                 /* Lost Msgs (Hbuf) */
    ACE_MT_MSGLOC_NEXT_PURGE);   /* Read and Purge */

if(nResult)
   printf(“aceRTGetHBufMsgDecoded Error: Code %d\n”,nResult);
```

Code Example 89. Reading a Decoded Message from the Host Buffer

3.3.4.6.5 Mode Code Support

The AceXtreme C SDK RT supports all MIL-STD-1553B Mode Codes. Please reference DDC’s *MIL-STD-1553 Designer’s Guide* for more information on individual Mode Code definitions.

3.3.4.6.5.1Reading and Writing Mode Code Data

The data portion of Mode Codes (that have associated data) is stored internally and can be accessed at any time via the **aceRTModeCodeReadData()** and **aceRTModeCodeWriteData()** functions.

```
S16BIT nResult;
U16BIT nMCData;

/* Read Mode Code Data */
nResult =  aceRTModeCodeReadData(
    0,                              /* LDN */
    ACE_RT_MCDATA_RX_SYNCHRONIZE,   /* “Synchronize” MC */
    &nMCData);                      /* MC Data Storage */

if(nResult)
    printf(“aceRTModeCodeReadData Error: Code %d\n”,nResult);
```

Code Example 90. Reading Mode Code Data for “Synchronize”(10001)

***Note:** In order for Mode Codes to be properly processed, they need to be "Legal". See Section 3.3.4.7 on how to legalize commands.*

3.3.4.6.5.2Mode Code Events

The AceXtreme C SDK Remote Terminal has the ability to notify the user when a specific Mode Code has been received (via an Interrupt Event). This is accomplished via the **aceRTModeCodeIrqEnable()** function. For each Mode Code Family, an individual (applicable) Mode Code can be configured to generate an **ACE_IMR1_RT_MODE_CODE** event when received. (See Section 3.2.3 on how to configure the **ACE_IMR1_RT_MODE_CODE** event.)

Table 63. Mode Code Family Types

Family Type	Description
ACE_RT_MCTYPE_RX_NO_DATA	"Own Address" RT-Receive Mode Codes without Data
ACE_RT_MCTYPE_RX_DATA	"Own Address" RT-Receive Mode Codes with Data
ACE_RT_MCTYPE_TX_NO_DATA	"Own Address" RT-Transmit Mode Codes without Data
ACE_RT_MCTYPE_TX_DATA	"Own Address" RT-Transmit Mode Code with Data
ACE_RT_MCTYPE_BCDT_RX_NO_DATA	Broadcast RT-Receive Mode Code without Data
ACE_RT_MCTYPE_BCST_RX_DATA	Broadcast RT-Receive Mode Code with Data
ACE_RT_MCTYPE_BCST_TX_NO_DATA	Broadcast RT-Transmit Mode Code without Data
ACE_RT_MCTYPE_BCST_TX_DATA	Broadcast RT-Transmit Mode Code with Data

Table 64. Mode Code Interrupt Event Options

Mode Code Value	Description	Applicable Family
ACE_RT_MCIRQ_DYN_BUS_CTRL	Dynamic Bus Control	ACE_RT_MCTYPE_TX_NO_DATA
ACE_RT_MCIRQ_SYNCHRONIZE	Synchronize	ACE_RT_MCTYPE_RX_DATA ACE_RT_MCTYPE_TX_NO_DATA ACE_RT_MCTYPE_TX_DATA ACE_RT_MCTYPE_BCST_RX_DATA ACE_RT_MCTYPE_BCST_TX_NO_DATA
ACE_RTMCIRQ_TRNS_STATUS	Transmit Status Word	ACE_RT_MCTYPE_TX_NO_DATA
ACE_RT_MCIRQ_INIT_SELF_TEST	Initiate Self-Test	ACE_RT_MCTYPE_TX_NO_DATA ACE_RT_MCTYPE_BCST_TX_NO_DATA
ACE_RT_MCIRQ_TRNS_SHUTDOWN	Transmitter Shutdown	ACE_RT_MCTYPE_TX_NO_DATA ACE_RT_MCTYPE_BCST_TX_NO_DATA
ACE_RT_MCIRQ_OVR_TRNS_SHUTDOWN	Override Transmitter Shutdown	ACE_RT_MCTYPE_TX_NO_DATA ACE_RT_MCTYPE_BCST_TX_NO_DATA

Table 64. Mode Code Interrupt Event Options

Mode Code Value	Description	Applicable Family
ACE_RT_MCIRQ_INH_TERM_FLAG	Inhibit terminal flag bit	ACE_RT_MCTYPE_TX_NO_DATA ACE_RT_MCTYPE_BCST_TX_NO_DATA
ACE_RT_MCIRQ_OVR_INH_TERM_FLG	Override inhibit terminal flag bit	ACE_RT_MCTYPE_TX_NO_DATA ACE_RT_MCTYPE_BCST_TX_NO_DATA
ACE_RT_MCIRQ_RESET_REMOTE_TERM	Reset remote terminal	ACE_RT_MCTYPE_TX_NO_DATA ACE_RT_MCTYPE_BCST_TX_NO_DATA
ACE_RT_MCIRQ_TRNS_VECTOR	Transmit vector word	ACE_RT_MCTYPE_TX_DATA
ACE_RT_MCIRQ_TRNS_LAST_CMD	Transmit last command	ACE_RT_MCTYPE_TX_DATA
ACE_RT_MCIRQ_TRNS_BIT	Transmit bit word	ACE_RT_MCTYPE_TX_DATA
ACE_RT_MCIRQ_SEL_TRNS_SHUTDOWN	Selected transmitter shutdown	ACE_RT_MCTYPE_RX_DATA ACE_RT_MCTYPE_BCST_RX_DATA
ACE_RT_MCIRQ_OVRD_SEL_TRNS_SHUTDWN	Override selected transmitter shutdown	ACE_RT_MCTYPE_RX_DATA ACE_RT_MCTYPE_BCST_RX_DATA

```
S16BIT nResult;

/* Enable Mode Code Event */
nResult = aceRTModeCodeIrqEnable(
    0,                              /* LDN */
    ACE_RT_MCTYPE_TX_DATA,          /* TX Mode Code */
    ACE_RT_MCIRQ_TRNS_VECTOR);      /* Transmit Vec. Wrd */

if(nResult)
   printf(“aceRTModeCodeIrqEnable Error: Code %d\n”,nResult);
```

Code Example 91. Enabling an “Transmit Vector Word” Mode Code Interrupt Event

3.3.4.7 Command Legalization

A “*Legal Command*” is a 1553 Command Word that is acceptable to be processed by the configured Remote Terminal. Since a particular RT may not be able to receive all valid 1553 Commands, the AceXtreme C SDK allows all invalid command words to be deemed illegal. If an illegal command word is sent to the DDC Remote Terminal to be processed by RT, the RT will return a Status Word with the Message Error (Bit 10) Set. Commands can be legalized by Address (Broadcast or “own address”), RT Transmit or Receive, Subaddress, and Word Count (including Mode Code commands).

Note: *Entries in the Command Legalization Lookup Table can be “Illegalized” and queried in a similar fashion using the* ***aceRTMsgLegalityDisable()*** *and* ***aceRTMsgLegalityStatus()*** *functions.*

Table 65. Command Legalization Lookup Table

SUBADDRESS 0		
Own Addr / Broadcast	**Command Direction**	**Word Count / Mode Code (32-bit Value)**
OWN (1)	TX (1)	Each binary value corresponds to WC/MC value (i.e. 0x00000100 is Word Count of 8)
OWN (1)	RX (0)	Each binary value corresponds to WC/MC value (i.e. 0x00000100 is Word Count of 8)
BCST (0)	RX (0)	Each binary value corresponds to WC/MC value (i.e. 0x00000100 is Word Count of 8)
• • •	• • •	• • •
SUBADDRESS 31		
Own Addr / Broadcast	**Command Direction**	**Word Count / Mode Code (32-bit Value)**
OWN (1)	TX (1)	Each binary value corresponds to WC/MC value (i.e. 0x00000100 is Word Count of 8)
OWN (1)	RX (0)	Each binary value corresponds to WC/MC value (i.e. 0x00000100 is Word Count of 8)
BCST (0)	RX (0)	Each binary value corresponds to WC/MC value (i.e. 0x00000100 is Word Count of 8)

Note: *By default, all 1553 Command Words are marked ILLEGAL until they are “legalized” via the* ***aceRTDataBlkMapToSA()*** *or* ***aceRTMsgLegalityEnable()*** *functions.*

```
S16BIT nResult;
#define RT_ADDRESS 10

/* Legalize Command Word */
nResult =  aceRTMsgLegalityEnable(
    0,             /* LDN */
    1,             /* Own address (not BCAST) */
    1,             /* Transmit Messages */
    19,            /* Subaddress 19 */
    0xFFFFFFFF);   /* “All” Word Counts */

if(nResult)
   printf(“aceRTMsgLegalityEnable Error: Code %d\n”,nResult);
```

Code Example 92. Legalizing a Specific 1553 Command Word (“Own Addr”, TX, SA19, All Word Counts)

3.3.4.8 Using Interrupt Events

Some applications may benefit from event notifications regarding the state of Remote Terminal. The following events directly relate to the Remote Terminal (RT) mode of operation. For information on how to configure events and callbacks, see Section 3.2.3.

Table 66. RT Interrupt Event Options

Event	Description
ACE_IMR1_RT_MODE_CODE (Bit 1)	Enable RT Mode Code Events
ACE_IMR1_RT_SUBADDR_EOM (Bit 4)	Enable RT Subaddress Access Events
ACE_IMR1_RT_CIRCBUF_ROVER (Bit 5)	Indicates an RT Circular Buffer has rolled over
ACE_IMR1_TT_ROVER (Bit 6)	Indicates the Hardware Timetag has rolled over
ACE_IMR1_BCRT_CMDSTK_ROVER (Bit 12)	Indicates the RT Command Stack has reached rolled over
ACE_IMR2_RT_CIRC_50P_ROVER	Indicates an RT Circular Buffer reached the 50% mark
ACE_IMR2_RT_CSTK_50P_ROVER	Indicates the RT Command Stack reached the 50% mark
ACE_IMR2_RT_ILL_CMD	Indicates the RT received an illegal command

3.3.5 Multi-RT (ACE_MODE_MRT)

The AceXtreme family includes a Multi-RT engine. The Multi-RT engine provides the capability of implementing up to 31 RTs on a single MIL-STD-1553 channel. The Multi-RT architecture provides multiprotocol support, with full compliance to all of the commonly used data bus standards, including MIL-STD-1553A, MIL-STD-1553B

(Notice 2), and MIL-STD-1760. Programmable flexibility enables the RT to be configured to fulfill any set of system requirements. This includes the capability to meet the MIL-STD-1553A response time requirement of 2µs to 5µs, and multiple options for mode code subaddresses, mode codes, RT status word, and RT BIT word. The Multi-RT engine implements all of the Protocol options and support as with the Single RT mode.

This section on the AceXtreme Multi-RT mode will only cover the difference between the Single RT mode (**ACE_MODE_RT**) and the Multi-RT mode. For all other information on RT mode see Section 3.3.4 on the **ACE_MODE_RT**.

3.3.5.1 Configuration

The AceXtreme C SDK's Multi-Remote Terminal mode has numerous configurations options that should be addressed before the RTs are brought online. Configuration is accomplished via the **acexMRTConfigure()** function and is typically called after the **aceInitialize()** function call.

Table 67. MRT Configuration Parameters

Variable	Description	Valid Options
wCmdStkSize	Size (in words) of the command stack	ACE_RT_CMDSTK_256 -> 256 words ACE_RT_CMDSTK_512 -> 512 words ACE_RT_CMDSTK_1K -> 1K words ACE_RT_CMDSTK_2K -> 2K words (default)
u32GlbDataStkType	The size of the desired Global RT data stack	ACE_RT_DBLK_GBL_C_128 -> 128 Words ACE_RT_DBLK_GBL_C_256 -> 256 Words ACE_RT_DBLK_GBL_C_512 -> 512 Words ACE_RT_DBLK_GBL_C_1K -> 1K Words ACE_RT_DBLK_GBL_C_2K -> 2K Words ACE_RT_DBLK_GBL_C_4K -> 4K Words ACE_RT_DBLK_GBL_C_8K -> 8K Words
u16GblDataBlkID	Identification number of global data block.	1 – 2048

After the call to **acexMRTConfigure()** has been completed the next task is to enable the desired RT address the AceXtreme device will emulate. This is done by the use of the function call **acexMRTEnableRT()**. This function must be called for each RT the AceXtreme device will be emulating. The second parameter passed into the function is the RT address while the third parameter has several options which can be "logically OR'ed" together. These parameters are:

Table 68. Multi-RT Configuration Options

Options	Description
ACE_RT_OPT_CLR_SREQ	Sets the Clear Service Request bit 2 to a 1. This will clear a service request after a tx vector word.
ACE_RT_OPT_LOAD_TT	With the reception of a Synchronize (with data) mode command, this will cause the Data Word from the Synchronize message to be loaded into the Hardware Time Tag Register.
ACE_RT_OPT_CLEAR_TT	With the reception of a Synchronize (without data) mode command, this will cause the value of the Hardware Time Tag Register to clear to 0x0000.
ACE_RT_OPT_OVR_DATA	This option affects the operation of the RT subaddress circular buffer memory management scheme. The Lookup Table address pointer will only be updated following a transmit message or following a valid receive or broadcast message to the respective Rx/Bcst subaddress. If this option is enabled, the Lookup Table pointer will not be updated following an invalid receive or broadcast message. In addition, an interrupt request for a circular buffer rollover condition (if enabled) will only occur following the end of a transmit message during which the last location in the circular buffer has been read or following the end of a valid receive or Broadcast message in which the last location in the circular buffer has been written to.
ACE_RT_OPT_OVR_MBIT	Enabling this option will cause a mode code Command Word with a T/R* bit of 0 and an MSB of the mode code field of 0 to be considered a defined (reserved) mode Command Word. The DDC hardware will respond to such a command and the Message Error bit will not become set.
ACE_RT_OPT_ALT_STS	Enabling this option will cause all 11 RT Status Word bits to be under control of the user via the**aceRTStatusBitsSet()** and **aceRTStatusBitsClear()** functions.
ACE_RT_OPT_IL_RX_D	Enabling this option will cause the device to not store the received data words if the DDC hardware device receives a receive command that has been illegalized.
ACE_RT_OPT_BSY_RX_D	If a particular Command Word (broadcast, T/R* bit, subaddress) has been programmed to be busy by means of the Busy lookup table and the RT receives a receive command, the 1553 device will respond with its Status Word with the Busy bit set and will not store the received Data Words. See Section 0 on modifying the Busy Lookup Table.
ACE_RT_OPT_SET_RTFG	If enabled, the Terminal flag status word bit will also become set if either a transmitter timeout (660.5 µs) condition had occurred or the ACE RT had failed its loopback test for the previous non-broadcast message. The loopback test is performed on all non-broadcast messages processed by the RT. The received version of all transmitted words is checked for validity (sync and data encoding, bit count, parity) and correct sync type. In addition, a 16-bit comparison is performed on the received version of the last word transmitted by the RT. If any of these checks or comparisons do not verify, the loopback test is considered to have failed.
ACE_RT_OPT_1553A_MC	This option causes the RT to consider only subaddress 0 to be a mode code subaddress. Subaddress 31 is treated as a standard non-mode code subaddress. In this configuration, the 1553 hardware will consider valid and respond only to mode code commands containing no data words. In this configuration, the RT will consider all mode commands followed by data words to be invalid and will not respond. In addition the 1553 hardware will not decode for the MIL-STD-1553B "Transmit Status" and "Transmit Last Command" mode codes. As a result, the internal RT Status Word Register will be updated as a result of these commands.

Table 68. Multi-RT Configuration Options	
Options	**Description**
ACE_RT_OPT_MC_O_BSYx	If a particular Command Word (broadcast, T/R* bit, subaddress) has been programmed to be busy by means of the Busy Lookup Table, the 1553 hardware will transmit its Status Word with its BUSY bit set, followed by a single Data Word, in response to either a Transmit Vector Word mode command or a Reserved transmit mode command with data (transmit mode codes 10110 through 11111). See Section 0 on modifying the Busy Lookup Table.
ACE_RT_OPT_BCST_DIS	The 1553 hardware will **not** recognize RT address 31 as the broadcast address. In this instance, RT address 31 may be used as a discrete RT address.
ACE_RT_OPT_INACTIVE	This option enables the RT, but leaves it inactive. Although the inactive RT does not respond to messages, all API calls to RT are available.
ACE_RT_OPT_TO_ACTIVATE	This option allows a BC to become active after the DBC switching process. Multiple RTs can be setup and activated at the same time.

3.3.5.2 Multi-RT Lookup Tables

The AceXtreme C SDK Multi-RT architecture contains a number of lookup tables to store specific user choices with regards to how the RTs will respond on the 1553 bus. The tables' functions are described in Sections 3.3.5.2.1, 3.3.5.2.2, and 3.3.5.2.3, including information on how to configure them.

3.3.5.2.1 Busy Bit Table

The Busy Bit Table holds information on which subaddresses will respond with the Busy Bit (Bit 3) set in the Status Word. The table is based on address (own address or broadcast) and command direction (RT Transmit/Receive). By default, the Busy Bit will be OFF for all subaddresses.

Note: *Entries in the Busy Bit Lookup Table can be cleared and queried using the* ***aceRTBusyBitsTblClear()*** *and* ***aceRTBusyBitsTblStatus()*** *functions.*

Table 69. Busy Bit Lookup Table

Own Addr / Broadcast	Command Direction	Bus Subaddresses
OWN (1)	TX (1)	Each binary bit corresponds to Subaddress (i.e. 0x00000100 is Subaddress 8)
OWN (1)	RX (0)	Each binary bit corresponds to Subaddress (i.e. 0x00000100 is Subaddress 8)
BCST (0)	RX (0)	Each binary bit corresponds to Subaddress (i.e. 0x00000100 is Subaddress 8)

```
S16BIT nResult;

/* Set Busy Bit for RT 1 (OWN Address  (1) , TX (1) , SA19) */
nResult =  acexMRTSetRTBusyBitsTbl(
    0,                  /* LDN */
    1,                  /* RT Address */
    1,                  /* Own Address (not BCAST) */
    1,                  /* Transmit Cmds */
    ACE_RT_SA19);       /* Subaddress 19 */

if(nResult)
    printf(“acexMRTSetRTBusyBitsTbl Error: %d\n”,nResult);
```

Code Example 93. Setting the Busy Bit for all TRANSMIT Commands to SA19 (“Own Addr”, TX, SA 19)

3.3.5.2.2 Status Word Table

The Status Word Table holds information on what bits are active (set) in the RT Status Word. By default, the “Standard” 1553 Status Word is used allowing configuration of optional Status Word bits. Alternatively, an “Alternate Status Word” can be defined (See **ACE_RT_OPT_ALT_STS** option in **aceRTConfigure()**). Using the Alternate Status word will allow the user to control bits 0-10 of the RT Status Word.

> ***Note:*** *When using the “Standard” Status Word, some bits are internally set and clear by the DDC Hardware and cannot be configured via the API.*
>
> ***Note:*** *Entries in the Status Word Lookup Table can be cleared and queried using the **aceRTStatusBitsClear()** and **aceRTStatusBitsStatus()** functions*

.

Table 70. Standard RT Status Word

BIT	Description
15 (MSB)	Remote Terminal Address Bit 4
14	Remote Terminal Address Bit 3
13	Remote Terminal Address Bit 2
12	Remote Terminal Address Bit 1
11	Remote Terminal Address Bit 0
10	Message Error
9	Instrumentation
8	Service Request
7	Reserved
6	Reserved
5	Reserved
4	Broadcast Command Received
3	Busy
2	Subsystem Flag
1	Dynamic Bus Control Acceptance
0 (LSB)	Terminal Flag

```
S16BIT nResult;

/* Set Service Request Status Bit  for RT 1*/
nResult =  acexMRTSetRTStatusBits(
    0,                    /* LDN */
    1,                    /* RT Address */
    ACE_RT_STSBIT_SREQ);  /* Set “Service Reg” bit */

if(nResult)
    printf(“acexMRTSetRTStatusBits Error: %d\n”, nResult);
```

Code Example 94. Setting the “Service Request” Bit in the RT Status Word

3.3.5.2.3 Built-in-Test (BIT) Word Table

The BIT Word is maintained by the RT device to store advanced error information. The internal contents of the BIT data word are provided to supplement the appropriate bits already available in the RT Status Word. The BIT Word definition can be the standard Internal BIT Data Word (default). Alternatively, the user can define a custom BIT Word for unique applications. Each RT has its own unique Built-in-Test word.

Whether internal or external, the BIT Word can be read via the **aceRTBITWrdRead()** function.

Table 71. Internal Built-In-Test (BIT) Data Word

BIT	Description
15 (MSB)	Transmitter Timeout
14	Loop Test Failure B
13	Loop Test Failure A
12	Handshake Failure
11	Transmitter Shutdown B
10	Transmitter Shutdown A
9	Terminal Flag Inhibited
8	BIT Test Fail
7	High Word Count
6	Low Word Count
5	Incorrect Sync Received
4	Parity/Manchester Error Received
3	RT-RT Gap/Sync/Address Error
2	RT-RT No Response Error
1	RT-RT 2nd Command Word Error
0 (LSB)	Command Word Contents Error

```
S16BIT nResult;
U16BIT nBITWord;

/* Configure RT BIT Word for RT 1*/
nResult =  acexMRTConfigRTBITWrd(
    0,                    /* LDN */
    1,                    /* RT Address */
    ACE_RT_BIT_INTERNAL,  /* Use Internal BIT */
    0);                   /* Reserved */

if(nResult)
    printf(“acexMRTConfigRTBITWrd Error: Code %d\n”,
nResult);

/* Read RT BIT Word for RT 1 */
nResult =  acexMRTReadRTBITWrd(
    0,                    /* LDN */
    1,                    /* RT Address */
    ACE_RT_BIT_INTERNAL,  /* Read from Internal BIT */
    &nBITWord);           /* Bit Word Storage */

if(nResult)
    printf(“acexMRTReadRTBITWrd Error: Code %d\n”, nResult);
```

Code Example 95. Configure and Read the Internal BIT Word

3.3.5.3 Multi-RT Data Blocks

RT Data Blocks are used to store MIL-STD-1553 data words being received or to be transmitted by the Remote Terminal. Once created, a RT Data Block can be independently read or written to. In order to use an RT Data Block it must be linked to a specific RT Subaddress. When a 1553 message involving that Subaddress is sent on the bus, the RT Engine will place or pull data from the linked RT Data Block.   For more information on the different data block types available see section 4.3.4.3 (RT Data Blocks in the Remote Terminal (**ACE_MODE_RT** section)).

3.3.5.3.1 Mapping a Multi-RT Data Block to a RT Subaddress

Once an RT Data Block (see Section 3.3.4.3.2) has been created, it needs to be linked to an RT Subaddress to identify which 1553 message data will be stored in which RT Data Blocks. Except for the Global Circular-Buffered RT Data Block (Section 3.3.4.3.1.4), only one Data Block can be linked to each Subaddress/Message Type combination. Applicable Message Types are **ACE_RT_MSGTYPE_RX** (Receive), **ACE_RT_MSGTYPE_TX** (Transmit), and **ACE_RT_MSGTYPE_BCST** (Broadcast). The AceXtreme Multi-RT functions use the call **acexMRTDataBlkMapToRTSA()** to map the data block to the RT’s subaddress. This function is similar to

aceRTDataBlkMapToSA() with the added (2nd) parameter of s8RtAddr which represents the address of the RT linked to the data block.

3.3.5.3.2 Subaddress Interrupt Options

The sixth Parameter of **aceRTDataBlkMapToSA()** defines interrupt options for this Subaddress/Message Type combination. The following options can be “logically OR’ed” into the Third parameter (wIrqOptions) of **aceRTDataBlkMapToSA()**. See Section 3.2.3 on how to capture Device Interrupt Events.

Interrupt Option	Description (if Enabled)
ACE_RT_DBLK_EOM_IRQ	The ACE_IMR1_RT_SUBADDR_EOM Device Interrupt Event will be generated when this Data Block receives 1553 message data.
ACE_RT_DBLK_CIRC_IRQ	The ACE_IMR1_RT_CIRCBUF_ROVER Device Interrupt Event will be generated when this Circular-Buffered Data Block rolls over (100%). The ACE_IMR2_RT_CIRC_50P_ROVER Device Interrupt Event will be generated when this Circular-Buffered Data Block reaches 50%.
0	For no IRQ

Note: *If the Seventh Parameter of **aceRTDataBlkMapToSA()** is TRUE, the specified Message Type/ Subaddress combination will be marked “Legal” in the Command Legalization Table (see Section 3.3.4.7).*

```
S16BIT nResult;
#define RTDBLK1     0x0001
#define RTADDRESS_1 0x0001

/* Map RTDBLK1 to RT 1 SA 19 */
nResult =  acexRTDataBlkMapToRTSA(
    0,                    /* LDN */
    RTADDRESS_1,          /* Data Block OUID to map */
    RTDBLK1,              /* Data Block OUID to map */
    19,                   /* Subaddress to map */
    ACE_RT_MSGTYPE_RX |   /* Map for RX Messages */
    ACE_RT_MSGTYPE_TX,    /* Map for TX Messages */
    0,                    /* Interrupt Mask */
    TRUE);                /* “Legalize” above commands */

if(nResult)
      printf(“acexRTDatBlkMapToRTSA Error: Code %d\n”, nResult);
```

Code Example 96. Mapping RTDBLK1 to TX ad RX messages for RTADDRESS_1, Subaddress 19

3.3.5.4 Activating the Remote Terminals

Once the Remote Terminal is configured and the RT Lookup Tables have been setup, the RT is ready to begin responding to 1553 bus traffic.

3.3.5.4.1 Starting and Stopping

The Remote Terminals can be started and stopped dynamically by the user. Starting and Stopping is accomplished by the **aceRTStart()** and **aceRTStop()** functions. Individual RTs can be start by calling **aceRTStart()** on an RT basis, or all enabled RTs (which were enabled by the call **acexMRTEnableRT()** can be started by passing a -1 into the second parameter of **acexMRTStart()**. The same mechanism also stands for the **acexMRTStop()** function.

```
S16BIT nResult;

/* Start the all Remote Terminals */
nResult =  acexRTStart(
    0,          /* LDN */
   -1);         /* Start all RTs */

if(nResult)
     printf(“acexRTStart Error: Code %d\n”, nResult);
```

Code Example 97. Starting the Remote Terminal (RT)

```
S16BIT nResult;

/* Stop all Remote Terminals (RT) */
nResult =  acexRTStop(
    0,          /* LDN */
   -1);         /* LDN */

if(nResult)
     printf(“acexRTStop Error: Code %d\n”, nResult);
```

Code Example 98. Stopping the Remote Terminal (RT)

3.3.5.5 Consuming Data

Just as in the single Remote Terminal mode (**ACE_MODE_RT**), the Multi-RT mode will behave in the same fashion when consuming monitored data, either via the Command Stack/Data Block access or the Host Buffer. For more information on consuming data see section 3.3.4.6.

3.3.5.5.1 Mode Code Support

The AceXtreme C SDK Multi-RT supports all MIL-STD-1553B Mode Codes. Please reference DDC's *MIL-STD-1553 Designer's Guide* for more information on individual Mode Code definitions.

3.3.5.5.2 Reading and Writing Mode Code Data

The data portion of Mode Codes (that have associated data) is stored internally and can be accessed at any time via the **aceRTModeCodeReadData()** and **aceRTModeCodeWriteData()** functions.

```
S16BIT nResult;
U16BIT nMCData;

/* Read Mode Code Data for RT 1 */
nResult =  acexMRTReadRTModeCodeData(
    0,                                 /* LDN */
    1,                                 /* RT Address */
    ACE_RT_MCDATA_RX_SYNCHRONIZE,      /* “Synchronize” MC */
    &nMCData);                         /* MC Data Storage */

if(nResult)
   printf(“acexMRTReadRTModeCodeData Error: Code %d\n”, nResult);
```

Code Example 99. Reading Mode Code Data for “Synchronize”(10001) for RT Address 1

Note: *In order for Mode Codes to be properly processed, they need to be “Legal”. See Section 3.3.4.5 on how to legalize Commands.*

3.3.5.5.3 Mode Code Events

The AceXtreme C SDK Multi-Remote Terminal has the ability to notify the user when a specific Mode Code has been received (via an Interrupt Event). This is accomplished via the **aceRTModeCodeIrqEnable()** function. For each Mode Code Family, an individual (applicable) Mode Code can be configured to generate an **ACE_IMR1_RT_MODE_CODE** event when received. (See Section 3.2.3 on how to configure the **ACE_IMR1_RT_MODE_CODE** event.)

Table 72. Mode Code Family Types

Family Type	Description
ACE_RT_MCTYPE_RX_NO_DATA	“Own Address” RT-Receive Mode Codes without Data
ACE_RT_MCTYPE_RX_DATA	“Own Address” RT-Receive Mode Codes with Data
ACE_RT_MCTYPE_TX_NO_DATA	“Own Address” RT-Transmit Mode Codes without Data
ACE_RT_MCTYPE_TX_DATA	“Own Address” RT-Transmit Mode Code with Data
ACE_RT_MCTYPE_BCDT_RX_NO_DATA	Broadcast RT-Receive Mode Code without Data
ACE_RT_MCTYPE_BCST_RX_DATA	Broadcast RT-Receive Mode Code with Data
ACE_RT_MCTYPE_BCST_TX_NO_DATA	Broadcast RT-Transmit Mode Code without Data
ACE_RT_MCTYPE_BCST_TX_DATA	Broadcast RT-Transmit Mode Code with Data

Table 73. Mode Code Interrupt Event Options

Mode Code Value	Description	Applicable Family
ACE_RT_MCIRQ_DYN_BUS_CTRL	Dynamic Bus Control	ACE_RT_MCTYPE_TX_NO_DATA
ACE_RT_MCIRQ_SYNCHRONIZE	Synchronize	ACE_RT_MCTYPE_RX_DATA ACE_RT_MCTYPE_TX_NO_DATA ACE_RT_MCTYPE_TX_DATA ACE_RT_MCTYPE_BCST_RX_DATA ACE_RT_MCTYPE_BCST_TX_NO_DATA
ACE_RTMCIRQ_TRNS_STATUS	Transmit Status Word	ACE_RT_MCTYPE_TX_NO_DATA
ACE_RT_MCIRQ_INIT_SELF_TEST	Initiate Self-Test	ACE_RT_MCTYPE_TX_NO_DATA ACE_RT_MCTYPE_BCST_TX_NO_DATA
ACE_RT_MCIRQ_TRNS_SHUTDOWN	Transmitter Shutdown	ACE_RT_MCTYPE_TX_NO_DATA ACE_RT_MCTYPE_BCST_TX_NO_DATA
ACE_RT_MCIRQ_OVR_TRNS_SHUTDOWN	Override Transmitter Shutdown	ACE_RT_MCTYPE_TX_NO_DATA ACE_RT_MCTYPE_BCST_TX_NO_DATA
ACE_RT_MCIRQ_INH_TERM_FLAG	Inhibit terminal flag bit	ACE_RT_MCTYPE_TX_NO_DATA ACE_RT_MCTYPE_BCST_TX_NO_DATA
ACE_RT_MCIRQ_OVR_INH_TERM_FLG	Override inhibit terminal flag bit	ACE_RT_MCTYPE_TX_NO_DATA ACE_RT_MCTYPE_BCST_TX_NO_DATA
ACE_RT_MCIRQ_RESET_REMOTE_TERM	Reset remote terminal	ACE_RT_MCTYPE_TX_NO_DATA ACE_RT_MCTYPE_BCST_TX_NO_DATA
ACE_RT_MCIRQ_TRNS_VECTOR	Transmit vector word	ACE_RT_MCTYPE_TX_DATA

Table 73. Mode Code Interrupt Event Options		
Mode Code Value	**Description**	**Applicable Family**
ACE_RT_MCIRQ_TRNS_LAST_CMD	Transmit last command	ACE_RT_MCTYPE_TX_DATA
ACE_RT_MCIRQ_TRNS_BIT	Transmit bit word	ACE_RT_MCTYPE_TX_DATA
ACE_RT_MCIRQ_SEL_TRNS_SHUTDOWN	Selected transmitter shutdown	ACE_RT_MCTYPE_RX_DATA ACE_RT_MCTYPE_BCST_RX_DATA
ACE_RT_MCIRQ_OVRD_SEL_TRNS_SHUTDWN	Override selected transmitter shutdown	ACE_RT_MCTYPE_RX_DATA ACE_RT_MCTYPE_BCST_RX_DATA

```
S16BIT nResult;

/* Enable Mode Code Event */
nResult = acexMRTEnableRTModeCodeIrq(
    0,                                /* LDN */
    1,                                /* RT Address 1 */
    ACE_RT_MCTYPE_TX_DATA,            /* TX Mode Code */
    ACE_RT_MCIRQ_TRNS_VECTOR);        /* Transmit Vec. Wrd */

if(nResult)
    printf(“acexMRTEnableRTModeCodeIrq Error: Code %d\n”, nResult);
```

Code Example 100. Enabling an “Transmit Vector Word” Mode Code Interrupt Event

3.3.5.6 Command Legalization

A “*Legal Command*” is a 1553 Command Word that is acceptable to be processed by the configured Remote Terminal. Since a particular RT may not be able to receive all valid 1553 Commands, the ***AceXtreme C SDK*** allows all invalid command words to be deemed illegal. If an illegal command word is sent to the DDC Remote Terminal to be processed by RT, the RT will return a Status Word with the Message Error (Bit 10) Set. Commands can be legalized by Address (Broadcast or “own address”), RT Transmit or Receive, Subaddress, and Word Count (including Mode Code commands).

> ***Note:*** *Entries in the Command Legalization Lookup Table can be “Illegalized” and queried in a similar fashion using the* ***aceRTMsgLegalityDisable()*** *and* ***aceRTMsgLegalityStatus()*** *functions.*

Table 74. Command Legalization Lookup Table

SUBADDRESS 0		
Own Addr / Broadcast	**Command Direction**	**Word Count / Mode Code (32-bit Value)**
OWN (1)	TX (1)	Each binary value corresponds to WC/MC value (i.e. 0x00000100 is Word Count of 8)
OWN (1)	RX (0)	Each binary value corresponds to WC/MC value (i.e. 0x00000100 is Word Count of 8)
BCST (0)	RX (0)	Each binary value corresponds to WC/MC value (i.e. 0x00000100 is Word Count of 8)
• • •	• • •	• • •
SUBADDRESS 31		
Own Addr / Broadcast	**Command Direction**	**Word Count / Mode Code (32-bit Value)**
OWN (1)	TX (1)	Each binary value corresponds to WC/MC value (i.e. 0x00000100 is Word Count of 8)
OWN (1)	RX (0)	Each binary value corresponds to WC/MC value (i.e. 0x00000100 is Word Count of 8)
BCST (0)	RX (0)	Each binary value corresponds to WC/MC value (i.e. 0x00000100 is Word Count of 8)

Note: *By default, all 1553 Command Words are marked ILLEGAL until they are "legalized" via the* ***aceRTDataBlkMapToSA()*** *or* ***aceRTMsgLegalityEnable()*** *functions.*

```
S16BIT nResult;
#define RT_ADDRESS 10

/* Legalize Command Word */
nResult =  acexMRTEnableRTMsgLegality(
    0,              /* LDN */
    RT_ADDRESS,     /* RT address */
    1,              /* Own address (not BCAST) */
    1,              /* Transmit Messages */
    19,             /* Subaddress 19 */
    0xFFFFFFFF);    /* “All” Word Counts */

if(nResult)
   printf(“acexMRTEnableRTMsgLegality Error: Code %d\n”, nResult);
```

Code Example 101. Legalizing a Specific 1553 Command Word (“Own Addr”, TX, SA19, All Word Counts)

3.3.5.7 Using Interrupt Events

Some applications may benefit from event notifications regarding the state of Remote Terminal. The following events directly relate to the Remote Terminal (RT) mode of operation. For information on how to configure events and callbacks, see Section 3.2.3.

Table 75. RT Interrupt Event Options

Event	Description
ACE_IMR1_RT_MODE_CODE (Bit 1)	Enable RT Mode Code Events
ACE_IMR1_RT_SUBADDR_EOM (Bit 4)	Enable RT Subaddress Access Events
ACE_IMR1_RT_CIRCBUF_ROVER (Bit 5)	Indicates an RT Circular Buffer has rolled over
ACE_IMR1_TT_ROVER (Bit 6)	Indicates the Hardware Timetag has rolled over
ACE_IMR1_BCRT_CMDSTK_ROVER (Bit 12)	Indicates the RT Command Stack has reached rolled over
ACE_IMR2_RT_CIRC_50P_ROVER	Indicates an RT Circular Buffer reached the 50% mark
ACE_IMR2_RT_CSTK_50P_ROVER	Indicates the RT Command Stack reached the 50% mark
ACE_IMR2_RT_ILL_CMD	Indicates the RT received an illegal command

3.3.5.8 MRT Response Time

The AceXtreme C SDK supports a programmable response time for the RTs on a Multi-Function AceXtreme Device. The programmable response time allows the user the control over when the Remote Terminal will respond to a MIL-STD-1553 command word.

The function **acexMRTRespTimeEnable()** and **acexMRTRespTimeDisable()**can be utilized to enable or disable the programmability of the RT’s response time.   The function **acexMRTSetRespTime()** can then be used to specify the RT’s response time.   The second parameter passed into **acexMRTSetRespTime()** represents the RT address, while the third parameter uses a range of 7 to 60 to represent the minimum and maximum values. A value of 7 represents 3.5 µseconds while a 60 represents 30 µseconds.

```
S16BIT nResult;
S8BIT  S8RTAddr =1;
U32BIT u32Time = 28;


/* Enable RT response time Command Word */
nResult =  acexMRTRespTimeEnable(
    0,                 /* LDN */
    s8RTAddr);         /* RT address */

if(nResult)
   printf("acexMRTRespTimeEnable Error: Code %d\n", nResult);

/* Enable RT response time Command Word */
nResult =  acexMRTSetRespTime(
    0,                 /* LDN */
    s8RTAddr,          /* RT address */
    u32Time);          /* RT response time of 14 µseconds */

if(nResult)
   printf("acexMRTSetRespTime Error: Code %d\n", nResult);
```

Code Example 102. Configuring RT 1's Response Time Value.

Note: AceXtreme *Multi-Function Devices Only*

3.3.5.9 MRT Response Timeout

The AceXtreme C SDK supports a programmable response timeout for the RTs on a Multi-Function AceXtreme Device. The programmable response timeout allows the user the control over when the Remote Terminal will designate a particular message as a no response and wait for the next message. Setting the RT's Response Time value is used for RT to RT commands.

The Remote Terminal with the AceXtreme C SDK manual supports a programmable timeout from 3.5 µseconds to 30 µseconds in steps of 500 nanoseconds. The function **acexMRTSetRespTimeout()** is used to program the timeout value for the Remote Terminal. The second parameter passed into **acexMRTSetRespTimeout()** represents the RT address, while the third parameter uses a range of 7 to 60 to represent the minimum and maximum values. A value of 7 represents 3.5 µseconds while a 60 represents 30 µseconds.

```
S16BIT nResult;
S8BIT  S8RTAddr =1;
U32BIT u32Timeout = 28;

/* Set Response timeout value for RT 1 */
nResult =  acexMRTSetRespTimeout(
    0,             /* LDN */
    s8RTAddr,      /* RT address */
    U32Timeout);   /* Timeout value 14 µseconds */

if(nResult)
   printf(“acexMRTSetRespTimeout Error: Code %d\n”, nResult);
```

Code Example 103. Configuring RT 1’s Response Timeout Value.

Note: AceXtreme *Multi-Function Devices Only*

3.3.5.10 DBC Acceptance

The Multi-Function AceXtreme hardware has the ability to accept the Dynamic Bus Controller mode code and accept command of the data bus. To enable DBC on an RT basis, the function **acexMRTDbcEnable()** must be called. The function requires the LDN of the channel, the RT address, and the hold-off time. When an RT accepts control of the data bus via the Dynamic bus controller mode code, a delay value must be specified in order for the RT to deactivate, and configure itself as the bus controller. This delay value is specified as the “hold-off time” in the AceXtreme C SDK.

```
S16BIT nResult;

/* Enable Dynamic Bus Controller for RT. */
nResult = acexMRTDbcEnable(0,           /* LDN            */
                           1,           /* RT Address     */
                           40);         /* RT Holdoff time   */

if(nResult)
    printf(“acexMRTDbcEnable Error: Code %d\n”,nResult);
```

Code Example 104. Clearing Discrete Configuration

Acceptance of the Dynamic Bus Controller mode code can also be disabled on an RT basis with the function **acexMRTDbcDisable()**. This function requires the LDN, and the RT address which is disabling acceptance of DBC.

3.3.5.11 MRT Intermessage Routines

The Multi-Function AceXtreme boards have support for Intermessage routines (IMRs). IMRs are a set of tasks executed between messages. In Multi-RT mode for a Multi-Function AceXtreme Device IMRs are assigned to the RT's subaddress.

3.3.5.11.1 MRT IMR types

Intermessage routines can be categorized into two groups, IMRs used for message response, and IMRs used for discrete and triggers. The IMRS listed in Table 76 may be "Logically OR'ed" together to form multiple intermessage actions.

Table 76. MRT Intermessage Routines

Intermessage Routine MacRos	Description
IMRS using Discretes and Triggers	
ACEX_MRT_IMR_SET_DISCRETE_X	Sets discrete output to a logic 1. X is 1 – 4
ACEX_MRT_IMR_RST_DISCRETE_X	Resets discrete output to a logic 0. X is 1 – 4
ACEX_BC_IMR_WAIT_FOR_INPUT_TRIG	BC operation will be paused until an external BC trigger signal is detected.
IMRS used for Message Response	
ACEX_MRT_IMR_NO_RESP_BOTH_BUS	Disables the current RT's transmitter on both buses.
ACEX_MRT_IMR_SET_SRQ_IN_STATUS	Indicates the service request bit in the status of the last RT to respond will be set.
ACEX_MRT_IMR_RST_SRQ_IN_STATUS	Indicates the service request bit in the status of the last RT to respond will be cleared.
ACEX_MRT_IMR_SET_TFG_IN_STATUS	Indicates the terminal flag bit in the status of the last RT to respond will be set.
ACEX_MRT_IMR_RST_TFG_IN_STATUS	Indicates the terminal flag bit in the status of the last RT to respond will be cleared.
ACEX_MRT_IMR_SET_BSY_IN_STATUS	Busy bit in the status of the last RT to respond will be set.
ACEX_MRT_IMR_RST_BSY_IN_STATUS	Indicates the busy bit in the status of the last RT to respond will be cleared.

3.3.5.11.2 Usage

IMRs are mapped to the RT's subaddress, and can be unique for receive, transmit and broadcast commands. The message types can also be "Logically OR'ed" together to share the IMRs for all message types. To link an IMR to a subaddress the function **acexMRTImrMapToRTSA()** can be used. The function requires the LDN, the RT's address, the sub address, message type and the IMR type.

```
S16BIT nResult;

/* BC IMR Trigger Select */
nResult = acexMRTImrMapToRTSA(
             0,                              /* LDN          */
             1,                              /* RT Address   */
             2,                              /* SA Address   */
             ACE_RT_MSGTYPE_RX,              /* Message Type */
             ACEX_MRT_IMR_NO_RESP_BOTH_BUS,  /* IMR          */

if(nResult)
    printf("acexMRTImrMapToRTSA Error: Code %d\n",nResult);
```

Code Example 105. Configure Discrete to BC IMR

3.3.5.11.3 IMR and Triggers

MRT intermessage routines may also generate or be generated by external triggers through the discrete I/O pins on the Multi-Function AceXtreme board. The function **acexMRTImrTrigSelect()** to link a discrete I/O pin to the IMRs using the discrete IO pins or the wait for trigger IMR. The function requires the LDN, and the discrete pin number (0 – 15 depending on the number of discrete on the Multi-Function AceXtreme board).

```
S16BIT nResult;

/* MRT IMR Trigger Select */
nResult = acexMRTImrTrigSelect(0,          /* LDN        */
                               1);         /* discrete 1  */

if(nResult)
    printf("acexMRTImrTrigSelect Error: Code %d\n",nResult);
```

Code Example 106. Configure Discrete to MRT IMR

3.3.6 Combination Modes

The AceXtreme C SDK has the ability to run certain modes of operation simultaneously. By running more than one mode at one time, the user can have a variety of information and features readily available.

3.3.6.1 Combined RT and MT-I (ACE_MODE_RTMTI)

The AceXtreme C SDK supports running the MT-I Monitor simultaneously with Remote Terminal operation. Excluding some caveats covered in this section, the functionality of MT-I and RT mode remain intact, as covered in their respected sections. See Sections 3.3.2 and 3.3.4 for all available MT-I and RT functionality

3.3.6.1.1 Configuration

The AceXtreme C SDK's combined MT-I Monitor / Remote Terminal (**ACE_MODE_RTMTI**) has numerous configuration options that should be addressed before any data processing is attempted. Configuration is accomplished via the **aceRTMTIConfigure()** function and is typically called after **aceInitialize()**.

Note: *Since both modes are configured via the* ***aceRTMTIConfigure()*** *function, there is no need to call each modes individual configuration function [****aceRTConfigure()*** *and* ***aceMTIConfigure()****].*

Table 77. RTMT-I Configuration Parameters

Variable	Description	Valid Options or Range
wCmdStkSize	Size (in words) of the RT command stack	ACE_RT_CMDSTK_256-> 256 words ACE_RT_CMDSTK_512-> 512 words ACE_RT_CMDSTK_1K-> 1K words ACE_RT_CMDSTK_2K-> 2K words (default)

u32DevBufByteSize	Size of DDC hardware memory (bytes) allocated for MT-I monitored data	MTI_DEVBUF_SIZE_128K = 128 KB
		MTI_DEVBUF_SIZE_256K = 256 KB
		MTI_DEVBUF_SIZE_512K = 512 KB
		MTI_DEVBUF_SIZE_1M = 1 MB

u32NumBufBlks	Number of memory blocks allocated for chapter 10 data packets	Target Host Memory Dependent
u32BufBlkByteSize	Bytes allocated for MT-I Data Packet buffer	Target Host Memory Dependent

fZeroCopyEnable	Enable Zero-Copy (Needs to be supported by target Operating System)	TRUE = Enable Zero-Copy
		FALSE = Disable Zero-Copy

u32IrqDataLen	Interrupt Event Option (MTI_NUM_WORDS): Number of data words necessary to generate an MT-I Data Packet	System Dependent
u32IrqMsgCnt	Interrupt Event Option (MTI_NUM_MSGS): Number of messages necessary to generate an MT-I Data Packet	System Dependant
u16IrqTimeInterval	Interrupt Event Option (MTI_TIME_MSG_TRIG_INT) (MTI_TIME_INT) Time Limit (us) necessary to generate a MT-I Data Packet	System Dependent

Table 77. RTMT-I Configuration Parameters		
Variable	**Description**	**Valid Options or Range**

u32IntConditions	Interrupt Events enabled to generate MT-I Data Packet	MTI_OVERFLOW_INT = Generate packet after DDC Hardware Overflow
		MTI_HOST_INT = Generate packet after Host Request
		MTI_TIME_MSG_TRIG_INT = Generate Packet after time limit reached, triggered by 1553 message
		MTI_TIME_INT = Generate Packet after time limit reached
		MTI_NUM_MSGS = Generate Packet after receiving “X” 1553 Messages
		MTI_NUM_WORDS = Generate Packet after receiving “X” words

u16Ch10ChnlId	IRIG-106 Chapter 10 assigned Channel ID for this device	0 - 65535
u8HdrVer	Reserved for Future Use	Reserved (0)
u8RelAbsTime	Reserved for Future Use	Reserved (0)
u8Ch10Checksum	Reserved for Future Use	Reserved (0)
dwOptions	Configuration Options	See Table 78

The following options can be “logically OR’ed” into the last parameter (dwOptions) of **aceRTMTIConfigure()**.

Table 78. RTMT-I Configuration Options	
Options	**Description**
ACE_RT_OPT_CLR_SREQ	Sets the Clear Service Request bit 2 to a 1. This will clear a service request after a tx vector word.
ACE_RT_OPT_LOAD_TT	With the reception of a Synchronize (with data) mode command, it will cause the Data Word from the Synchronize message to be loaded into the Hardware Time Tag Register.
ACE_RT_OPT_CLEAR_TT	With the reception of a Synchronize (without data) mode command, it will cause the value of the Hardware Time Tag Register to clear to 0x0000.
ACE_RT_OPT_OVR_DATA	This option affects the operation of the RT subaddress circular buffer memory management scheme. The Lookup Table address pointer will only be updated following a transmit message or following a valid receive or broadcast message to the respective Rx/Bcst subaddress. If this option is enabled, the Lookup Table pointer will not be updated following an invalid receive or broadcast message. In addition, an interrupt request for a circular buffer rollover condition (if enabled) will only occur following the end of a transmit message during which the last location in the circular buffer has been read or following the end of a valid receive or Broadcast message in which the last location in the circular buffer has been written to.

Table 78. RTMT-I Configuration Options

Options	Description
ACE_RT_OPT_OVR_MBIT	Enabling this option will cause a mode code Command Word with a T/R* bit of 0 and an MSB of the mode code field of 0 will be considered a defined (reserved) mode Command Word. The DDC hardware will respond to such a command and the Message Error bit will not become set.
ACE_RT_OPT_ALT_STS	Enabling this option will cause all 11 RT Status Word bits to be under control of the User via the **aceRTStatusBitsSet()** and **aceRTStatusBitsClear()** functions.
ACE_RT_OPT_IL_RX_D	Enabling this option will cause the device to not store the received data words if the DDC hardware device receives a receive command that has been illegalized.
ACE_RT_OPT_BSY_RX_D	If a particular Command Word (broadcast, T/R* bit, subaddress) has been programmed to be busy by means of the Busy lookup table and the RT receives a receive command, the 1553 device will respond with its Status Word with the Busy bit set and will not store the received Data Words. See Section 0 on modifying the Busy Lookup Table.
ACE_RT_OPT_SET_RTFG	If enabled, the Terminal flag status word bit will also become set if either a transmitter timeout (660.5 µs) condition had occurred or the ACE RT had failed its loopback test for the previous non-broadcast message. The loopback test is performed on all non-broadcast messages processed by the RT. The received version of all transmitted words is checked for validity (sync and data encoding, bit count, parity) and correct sync type. In addition, a 16-bit comparison is performed on the received version of the last word transmitted by the RT. If any of these checks or comparisons do not verify, the loopback test is considered to have failed.
ACE_RT_OPT_1553A_MC	If this option is chosen, the RT considers only subaddress 0 to be a mode code subaddress. Subaddress 31 is treated as a standard nonmode code subaddress. In this configuration, the 1553 hardware will consider valid and respond only to mode code commands containing no data words. In this configuration, the RT will consider all mode commands followed by data words to be invalid and will not respond. In addition the 1553 hardware will not decode for the MIL-STD-1553B "Transmit Status" and "Transmit Last Command" mode codes. As a result, the internal RT Status Word Register will be updated as a result of these commands.
ACE_RT_OPT_MC_O_BSY	If a particular Command Word (broadcast, T/R* bit, subaddress) has been programmed to be busy by means of the Busy Lookup Table, the 1553 hardware will transmit its Status Word with its BUSY bit set, followed by a single Data Word, in response to either a Transmit Vector Word mode command or a Reserved transmit mode command with data (transmit mode codes 10110 through 11111). See Section 0 on modifying the Busy Lookup Table.
ACE_RT_OPT_BCST_DIS	The 1553 hardware will not recognize RT address 31 as the broadcast address. In this instance, RT address 31 may be used as a discrete RT address.
ACE_MT_OPT_BCST_DIS	Disable Broadcast Address (RT 31) for the Monitor
ACE_MT_OPT_1553A_MC	Enable 1553A Mode Code Support for the Monitor

3.3.6.1.2 Activating the RT and MT-I Monitor

Once the MT-I Monitor and Remote Terminal have been configured and the Monitor RT Filtering has been setup, the RTMT-I Engine is ready to begin processing 1553 bus traffic.

3.3.6.1.2.1Starting and Stopping

The MT-I Monitor / Remote Terminal can be started and stopped dynamically by the user. Starting and Stopping is accomplished by the **aceRTMTIStart()** and **aceRTMTIStop()** functions.

> ***Note:*** *When running in* ***ACE_RTMTI_MODE****, the functions* ***aceRTMTIStart()*** *and* ***aceRTMTIStop()*** *must be used to start and stop both modes (RT and MT-I). While in this mode, the individual start and stop functions of each mode will not operate.*

> ***Note:*** *By Stopping the Monitor with* ***aceRTMTIStop()****, any traffic that has not been consumed by the user will be discarded. To temporarily pause monitoring without discarding data, see Section 3.3.3.3.2. Please note that is not possible to pause the Remote Terminal.*

```
S16BIT nResult;

/* Start the RTMT-I Monitor */
nResult =  aceRTMTIStart(
    0);       /* LDN */

if(nResult)
    printf(“aceRTMTIStart Error: Code %d\n”, nResult);
```

Code Example 107. Starting the MT-I Monitor and Remote Terminal

```
S16BIT nResult;

/* Stop the RTMT-I Monitor */
nResult =  aceRTMTIStop(
    0);       /* LDN */

if(nResult)
    printf(“aceRTMTIStop Error: Code %d\n”, nResult);
```

Code Example 108. Stopping the MT-I Monitor and Remote Terminal

3.3.6.1.2.2MT-I Continue and Pause

The **aceMTIPause()** function will temporarily pause the monitoring of bus traffic, leaving all unconsumed data intact. After pausing, bus monitoring can be restarted using the **aceMTIContinue()** function.

***Note:** The Pause and Continue functionality referenced above will only affect the Monitor (MT-I) portion of this mode. The Remote Terminal (RT) portion will be unaffected.*

3.3.6.2 Combined RT and MT (ACE_MODE_RTMT)

The AceXtreme C SDK supports running the Classic Monitor simultaneously with Remote Terminal operation. Excluding some caveats covered in this section, the functionality of MT and RT mode remain intact, as covered in their respected sections. See Sections 3.3.3 and 3.3.4 for all available MT and RT functionality.

***Note:** Classic Monitor support should only be used for existing applications or if the target DDC hardware is not of the* E²MA or AceXtreme *Family. New designs using* E²MA or AceXtreme *hardware that require a combined Monitor and Remote Terminal should use RTMT-I mode (see Section 3.3.6.1).*

3.3.6.2.1 Configuration

The AceXtreme C SDK's combined Classic Monitor / Remote Terminal mode has numerous configuration options that should be addressed before any data processing is attempted. Configuration is accomplished via the **aceRTMTConfigure()** function and is typically called after **aceInitialize()**.

Table 79. RTMT Configuration Parameters

Variable	Description	Valid Options
wRTCmdStkSize	Size (in words) of the Remote Terminal (RT) command stack	ACE_RT_CMDSTK_256-> 256 words ACE_RT_CMDSTK_512-> 512 words ACE_RT_CMDSTK_1K -> 1K words ACE_RT_CMDSTK_2K -> 2K words (default)

wMTStkType	Stack type to use for Monitor (MT) command and data stacks	ACE_MT_SINGLESTK = Single-Buffered Stack (default)
		ACE_MT_DOUBLESTK = Double-Buffered Stack

wMTCmdStkSize	Size (in words) of the Monitor (MT) command stack	ACE_MT_CMDSTK_256-> 256 words ACE_MT_CMDSTK_1K -> 1K words ACE_MT_CMDSTK_4K -> 4K words (default) ACE_MT_CMDSTK_16K-> 16K words
wMTDataStkSize	Size (in words) of the Monitor (MT) data stack	ACE_MT_DATASTK_512-> 512 words ACE_MT_DATASTK_1K-> 1K words ACE_MT_DATASTK_2K-> 2K words ACE_MT_DATASTK_4K-> 4K words ACE_MT_DATASTK_8K-> 8K words ACE_MT_DATASTK_16K-> 16K words (default) ACE_MT_DATASTK_32K-> 32K words ACE_MT_DATASTK_64K-> 64K words
dwOptions	RTMT Configuration Options	See Table 80

The following options can be "logically OR'ed" into the Sixth parameter (dwOptions) of **aceRTMTConfigure()**.

Table 80. RTMT Configuration Options

Options	Description
ACE_RT_OPT_CLR_SREQ	Sets the Clear Service Request bit 2 to a 1. This will clear a service request after a tx vector word.
ACE_RT_OPT_LOAD_TT	With the reception of a Synchronize (with data) mode command, it will cause the Data Word from the Synchronize message to be loaded into the Hardware Time Tag Register.
ACE_RT_OPT_CLEAR_TT	With the reception of a Synchronize (without data) mode command, it will cause the value of the Hardware Time Tag Register to clear to 0x0000.
ACE_RT_OPT_OVR_DATA	This option affects the operation of the RT subaddress circular buffer memory management scheme. The Lookup Table address pointer will only be updated following a transmit message or following a valid receive or broadcast message to the respective Rx/Bcst subaddress. If this option is enabled, the Lookup Table pointer will not be updated following an invalid receive or broadcast message. In addition, an interrupt request for a circular buffer rollover condition (if enabled) will only occur following the end of a transmit message during which the last location in the circular buffer has been read or following the end of a valid receive or Broadcast message in which the last location in the circular buffer has been written to.
ACE_RT_OPT_OVR_MBIT	Enabling this option will cause a mode code Command Word with a T/R* bit of 0 and an MSB of the mode code field of 0 will be considered a defined (reserved) mode Command Word. The DDC hardware will respond to such a command and the Message Error bit will not become set.
ACE_RT_OPT_ALT_STS	Enabling this option will cause all 11 RT Status Word bits to be under control of the User via the **aceRTStatusBitsSet()** and **aceRTStatusBitsClear()** functions.
ACE_RT_OPT_IL_RX_D	Enabling this option will cause the device to not store the received data words if the DDC hardware device receives a receive command that has been illegalized.
ACE_RT_OPT_BSY_RX_D	If a particular Command Word (broadcast, T/R* bit, subaddress) has been programmed to be busy by means of the Busy lookup table and the RT receives a receive command, the 1553 device will respond with its Status Word with the Busy bit set and will not store the received Data Words. See Section 0 on modifying the Busy Lookup Table.
ACE_RT_OPT_SET_RTFG	If enabled, the Terminal flag status word bit will also become set if either a transmitter timeout (660.5 µs) condition had occurred or the ACE RT had failed its loopback test for the previous non-broadcast message. The loopback test is performed on all non-broadcast messages processed by the RT. The received version of all transmitted words is checked for validity (sync and data encoding, bit count, parity) and correct sync type. In addition, a 16-bit comparison is performed on the received version of the last word transmitted by the RT. If any of these checks or comparisons do not verify, the loopback test is considered to have failed.
ACE_RT_OPT_1553A_MC	If this option is chosen, the RT considers only subaddress 0 to be a mode code subaddress. Subaddress 31 is treated as a standard nonmode code subaddress. In this configuration, the 1553 hardware will consider valid and respond only to mode code commands containing no data words. In this configuration, the RT will consider all mode commands followed by data words to be invalid and will not respond. In addition, the 1553 hardware will not decode for the MIL-STD-1553B "Transmit Status" and "Transmit Last Command" mode codes. As a result, the internal RT Status Word Register will be updated as a result of these commands.
ACE_RT_OPT_MC_O_BSY	If a particular Command Word (broadcast, T/R* bit, subaddress) has been programmed to be busy by means of the Busy Lookup Table, the 1553 hardware will transmit its Status Word with its BUSY bit set, followed by a single Data Word, in response to either a Transmit Vector Word mode command or a Reserved transmit mode command with data (transmit mode codes 10110 through 11111). See Section 0 on modifying the Busy Lookup Table.
ACE_RT_OPT_BCST_DIS	The 1553 hardware will not recognize RT address 31 as the broadcast address. In this instance, RT address 31 may be used as a discrete RT address.
ACE_MT_OPT_BCST_DIS	Disable Broadcast Address (RT 31) for the Monitor

Table 80. RTMT Configuration Options

Options	Description
ACE_MT_OPT_1553A_MC	Enable 1553A Mode Code Support for the Monitor

3.3.6.2.2 Activating the RT and Classic Monitor

Once the Classic Monitor and Remote Terminal have been configured and the Monitor RT Filtering has been setup, the RTMT Engine is ready to begin processing 1553 bus traffic.

3.3.6.2.2.1Starting and Stopping

The Classic Monitor / Remote Terminal can be started and stopped dynamically by the user. Starting and Stopping is accomplished by the **aceRTMTStart()** and **aceRTMTStop()** functions.

Note: *When running in **ACE_RTMT_MODE**, the functions **aceRTMTStart()** and **aceRTMTStop()** must be used to start and stop both modes (RT and MT). While in this mode, the individual start and stop functions of each mode will not operate.*

Note: *By stopping the Monitor with **aceRTMTStop()**, any traffic that has not been consumed by the user will be discarded. To temporarily pause monitoring without discarding data, see Section 3.3.3.3.2. Please note that is not possible to pause the Remote Terminal.*

```
S16BIT nResult;

/* Start the RTMT Monitor */
nResult =  aceRTMTStart(0);       /* LDN */

if(nResult)
    printf(“aceRTMTStart Error: Code %d\n”, nResult);
```

Code Example 109. Starting the MT Monitor and Remote Terminal

```
S16BIT nResult;

/* Stop the RTMT Monitor */
nResult =  aceRTMTStop(0);       /* LDN */

if(nResult) printf(“aceRTMTStop Error: Code %d\n”, nResult);
```

Code Example 110. Stopping the MT Monitor and Remote Terminal

3.3.6.2.2.2MT Continue and Pause

The **aceMTPause()** function will temporarily pause the monitoring of bus traffic, leaving all unconsumed data intact. After pausing, bus monitoring can be restarted using the **aceMTContinue()** function.

> ***Note:*** *The Pause and Continue functionality referenced above will only affect the Classic Monitor (MT) portion of this mode. The Remote Terminal (RT) portion will be unaffected.*

3.3.6.3 Combined BC and MT-I (ACE_MODE_BCMTI)

The AceXtreme C SDK supports running the BC and MT-I Monitor with an AceXtreme device. The combined BC and MT-I mode has several configuration options. These options can be made by calling **aceBCConfigure()**. For more information of these configurations, see Sections 3.3.1 and 3.3.2 for the BC and MT-I configurations.

> ***Note:*** *The* ***ACE_MODE_BCMTI*** *is only supported by* AceXtreme *devices.*

3.3.6.4 Combined MRT and MT-I (ACE_MODE_MRTMTI)

The AceXtreme C SDK supports running the Multi-RT and MT-I Monitor with an AceXtreme device. The combined Multi-RT and MT-I mode has several configuration options. These options can be made by calling **acexMRTConfigure()** and **aceMTIConfigure()**. For more information of these configurations, see Sections 3.3.4 and 3.3.2 for the RT and MT-I configurations.

> ***Note:*** *The* ***ACE_MODE_MRTMTI*** *is only supported by* AceXtreme *devices.*

3.3.6.5 AceXtreme Multi-Function Modes

The Multi-Function AceXtreme devices support a running the Bus Controller, all 31 RTs and a monitor concurrently. There are two supported modes of operation, a combined BC and Multi-RT mode, and a BC, Multi-RT, and MTI-I mode.

3.3.6.5.1 Combined BC, MRT and MT-I (ACE_MODE_ALL)

The AceXtreme C SDK supports running the Bus Controller, Multi-RT and MT-I Monitor with an AceXtreme device. The combined Bus Controller, Multi-RT and MT-I mode has several configuration options. These options can be made by calling **aceBCConfigure()**, **acexMRTConfigure()** and **aceMTIConfigure()**. For more information of these configurations, see Sections 3.3.1, 3.3.2 and 3.3.5 for the BC, MRT, and MTI-I configurations.

Note: *The* ***ACE_MODE_ALL*** *is only supported by* AceXtreme *devices.*

3.3.6.5.2 BC Response timeout

The AceXtreme C SDK supports a programmable response timeout for the Bus Controller with a Multi-Function AceXtreme Device. The programmable response timeout allows the user the control over when the Bus Controller will designate a particular message as a no response and move onto the next message.

The Bus Controller with the AceXtreme C SDK manual supports a programmable timeout from 3.5 µseconds to 30 µseconds in steps of 500 nanoseconds. The function **acexBCSetRespTimeout()** is used to program the timeout value for the Bus Controller. The second parameter passed into **acexBCSetRespTimeout()** uses a range of 7 to 60 to represent the minimum and maximum values. A value of 7 represents 3.5 µseconds while a 60 represents 30 µseconds.

```
S16BIT nResult;
U32BIT u32Timeout = 28;

/* Set Response timeout value for RT 1 */
nResult =  acexBCSetRespTimeout(
    0,             /* LDN */
    U32Timeout);   /* Timeout value 14 µseconds */

if(nResult)
   printf(“acexBCSetRespTimeout Error: Code %d\n”, nResult);
```

Code Example 111. Configuring the BC’s Response timeout value.

Note: AceXtreme *Multi-Function Devices Only*

3.3.7 Error Injection

The AceXtreme C SDK support error injection with DDC's Multi Function AceXtreme devices. Errors can be injected on either a Bus Controller Message or via an emulated RT on the Multi-Function AceXtreme Device. Error injection is supported on a message by message basis allowing one error to be injected per message.

Note: AceXtreme *Multi-Function Devices Only*

3.3.7.1 BC Error Injection

The Bus Controller on a Multi-Function AceXtreme Device supports error injection on a message by message basis. The error types the can be injected by the BC are Length Errors and Encoding Errors. Length Errors include modifying data word count, or Bit Counter Errors for a specific message.

Data Word Count Errors include increasing or decreasing the number of data words in a message, while Bit Count Errors allow for up to an extra three bits to be added onto a specific word for a given message. Bit Errors also allows for up to three bits to be removed from a word of a given message.

Encoding errors supports a Glitch or Inverse error. The Glitch error forces the output of the encoder to an idle bus condition for a specified period of time. The Inverse error will invert the output of the encoder for a specific time period. The Glitch and Inverse errors allow for error types such as invalid sync patterns, parity errors and Manchester bi-phase errors. The Glitch and Inverse errors can be anywhere in the message (command words, or data words) with a resolution of 50 nanoseconds, and can be injected from 50 to 3000 nanoseconds, in steps of 50 nanoseconds.

Error injection on a Multi-Function AceXtreme device operating in Bus Controller mode can be enabled or disabled with the commands **acexBCMsgErrorEnable()** and **acexBCMsgErrorDisable()**.

After error injection has been enabled for the Bus Controller, the function **acexBCSetMsgError()** is used to define the errors to be injected on a message. The call to **acexBCSetMsgError()** requires three parameters, the LDN of the BC, the Message ID of the message on which the error will be injected, and a structure (**ACEX_ERR_INJ**) which indicates the error that will be injected into the message.

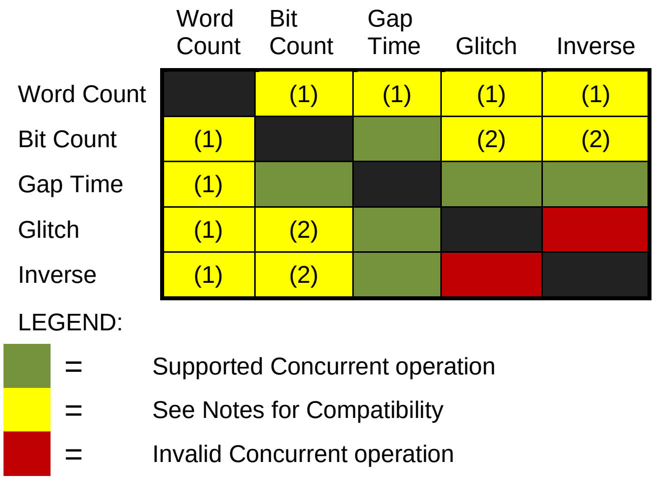
Table 81. ACEX_ERR_INJ Structure – BC Error Injection

Event	Description	Valid Values
u32ErrorType	The type of error being injected	ACEX_EI_NONE ACEX_EI_WORD_COUNT ACEX_EI_BIT_COUNT ACEX_EI_GLITCH ACEX_EI_INVERSE ACEX_EI_GAP
s16WordSel	Specifies the word to inject the error.	0 = CMD, 1-32 for data word
s16WordCount	Number of words to be added or removed from message.	-32 to -1, 0, 1 to 32 in Words
s16BitCount	Number of bits to be added or removed from message	-3, -2, -1, 0, 1, 2, 3 in bits
s16GapTime	Amount of time between words in a message	0 – 32 µseconds
s16GlitchLoc	Location of the Glitch Error	0 – 400 in 50 ns increments (range 0 -20 µseconds)
S16GlitchDur	Duration of the Glitch Error	0 – 60 in 50 ns increments (range 0 – 3 µseconds)
s16InverseLoc	Location of Inverse Error	0 – 400 in 50 ns increments (range 0 – 20 µseconds)
s16InverseDur	Duration of Inverse Error	0 – 60 in 50 ns increments (range 0 – 3 µseconds)
U16BITStatusBitMask	RT Status Bit Mask	0x0000 – 0xFFFF
U16StatusBits	RT Status Bit Modification	0x0000 – 0xFFFF

```
S16BIT       nResult;
ACEX_ERR_INJ sError;

/* Set error type to remove one word from message */
sError->u32ErrorType = ACEX_EI_WORD_COUNT;
sError->S16WordCount = -1;

/* Inject error on MSG1 */
nResult =  acexBCMsgError(
    0,             /* LDN */
    MSG1,          /* Message 1 */
    &sErrror);     /* Error Structure */

if(nResult)
   printf(“acexBCMsgError Error: Code %d\n”, nResult);


/* Enable Error injection for BC mode */
nResult =  acexBCMsgErrorEnable(0);            /* LDN */

if(nResult)
   printf(“acexBCMsgErrorEnable Error: Code %d\n”, nResult);
```

Code Example 112. Configuring the BC’s Error Injection.

The AceXtreme C SDK does not protect the user from using error types that may conflict with one another, such as using a Glitch and Inverse error on the same word. The table below shows a matrix of supported error injection types. Care needs to be taken when mixing multiple error injection types.



	Word Count	Bit Count	Gap Time	Glitch	Inverse
Word Count	⬛	🟨 (1)	🟨 (1)	🟨 (1)	🟨 (1)
Bit Count	🟨 (1)	⬛	🟩	🟨 (2)	🟨 (2)
Gap Time	🟨 (1)	🟩	⬛	🟩	🟩
Glitch	🟨 (1)	🟨 (2)	🟩	⬛	🟥
Inverse	🟨 (1)	🟨 (2)	🟩	🟥	⬛

LEGEND:

🟩 = Supported Concurrent operation

🟨 = See Notes for Compatibility

🟥 = Invalid Concurrent operation

Notes:

1. User must ensure errors do not occur within removed words due to the Word Count error.
2. User must ensure that these errors do not occur on the removed bits due to Bit Count Error.
3. These errors are invalid if NO RSP A and NO RSP B are enabled.

Figure 35. AceXtreme BC Error Injection

3.3.7.2 RT Error Injection

The Remote Terminal on a Multi-Function AceXtreme Device supports error injection on a message by message basis. The error types the can be injected by the RT are Length Errors and Encoding Errors. Length Errors include modifying data word count, or Bit Counter Errors for a specific message.

Data Word Count Errors include increasing or decreasing the number of data words in a message, while Bit Count Errors allow for up to an extra three bits to be added onto a specific word for a given message. Bit Errors also allows for up to three bits to be removed from a word of a given message.

Encoding errors supports a Glitch or Inverse error. The Glitch error forces the output of the encoder to an idle bus condition for a specified period of time. The Inverse error will invert the output of the encoder for a specific time period. The Glitch and Inverse errors allow for error types such as invalid sync patterns, parity errors and Manchester bi-phase errors. The Glitch and Inverse errors can be anywhere in the message (command words, or data words) with a resolution of 50 nanoseconds, and can be injected from 50 to 3000 nanoseconds, in steps of 50 nanoseconds.

Error injection on a Multi-Function AceXtreme device operating in Remote Terminal mode can be enabled or disabled with the commands **acexMRTMsgErrorEnable()** and **acexMRTMsgErrorDisable()**. These two functions require two parameters. The first parameter is the LDN of the device, and the second is the Remote Terminal address.

After error injection has been enabled for the Remote Terminal, the function **acexMRTSetMsgError()** is used to define the errors to be injected on a message. The call to **acexMRTSetMsgError()** requires three parameters, the LDN of the BC, RT address, and a structure (**ACEX_ERR_INJ**) which indicates the errors that will be injected into the message.

Table 82. ACEX_ERR_INJ Structure – RT Error Injection

Event	Description	Valid Values
u32ErrorType	The type of error being injected	ACEX_EI_NONE ACEX_EI_WORD_COUNT ACEX_EI_BIT_COUNT ACEX_EI_GLITCH ACEX_EI_INVERSE ACEX_EI_GAP ACEX_EI_NO_RESP_A ACEX_EI_NO_RESP_B ACEX_EI_RESP_LATE ACEX_EI_RESP_WRONG_BUS ACEX_EI_RESP_WRONG_ADDR ACEX_EI_RESP_STATUS_BIT_SET
s16WordSel	Specifies the word to inject the error.	0 = CMD, 1-32 for data word
s16WordCount	Number of words to be added or removed from message.	-32 to -1, 0, 1 to 32 in Words
s16BitCount	Number of bits to be added or removed from message	-3, -2, -1, 0, 1, 2, 3 in bits
s16GapTime	Amount of time between words in a message	0 – 32 µseconds
s16GlitchLoc	Location of the Glitch Error	0 – 400 in 50 ns increments (range 0 -20 µseconds)
S16GlitchDur	Duration of the Glitch Error	0 – 60 in 50 ns increments (range 0 – 3 µseconds)
s16InverseLoc	Location of Inverse Error	0 – 400 in 50 ns increments (range 0 – 20 µseconds)
s16InverseDur	Duration of Inverse Error	0 – 60 in 50 ns increments (range 0 – 3 µseconds)
s16RespLateTime	RT delays response to a command word.	7 – 60 in 50 ns increments (range 3.5 µs – 30 µs)
s16RespWrongAddr	RT responds with wrong address in status word.	0 – 31 (RT address)
U16BITStatusBitMask	RT Status Bit Mask	0x0000 – 0xFFFF
U16StatusBits	RT Status Bit Modification	0x0000 – 0xFFFF

```
S16BIT       nResult;
ACEX_ERR_INJ sError;

/* Set error type to remove one word from message */
sError->u32ErrorType    = ACEX_EI_RESP_LATE;
sError->S16RespLateTime = 14;

/* Inject a Late Response error on to RT1 */
nResult =  acexMRTMsgError(
    0,             /* LDN */
    1,              /* Message 1 */
    &sErrror);     /* Error Structure */

if(nResult)
   printf("acexMRTMsgError Error: Code %d\n", nResult);


/* Enable Error injection for MRT mode */
nResult =  acexMRTMsgErrorEnable(
    0,             /* LDN */
    1);            /* RT address */

if(nResult)
   printf("acexMRTMsgErrorEnable Error: Code %d\n", nResult);
```

Code Example 113. Configuring the RT's Error Injection.

The AceXtreme C SDK does not protect the user from using error types that may conflict with one another, such as using a Glitch and Inverse error on the same word. The table below shows a matrix of supported error injection types. Care needs to be taken when mixing multiple error injection types.

	Word Count	Bit Count	Gap Time	Glitch	Inverse	No RSP A	No RSP B	LATE RSP	Wrong Bus RSP	RSP Wrong ADDR
Word Count	⬛	(1)	(1)	(1)	(1)	🟩	🟩	🟩	🟩	🟩
Bit Count	(1)	⬛	🟩	(2)	(2)	🟩	🟩	🟩	🟩	🟩
Gap Time	(1)	🟩	⬛	🟩	🟩	🟩	🟩	🟩	🟩	🟩
Glitch	(1)	(2)	🟩	⬛	🟥	🟩	🟩	🟩	🟩	🟩
Inverse	(1)	(2)	🟩	🟥	⬛	🟩	🟩	🟩	🟩	🟩
No RSP A	🟩	🟩	🟩	🟩	🟩	⬛	🟩	(3)	(3)	(3)
No RSP B	🟩	🟩	🟩	🟩	🟩	🟩	⬛	(3)	(3)	(3)
LATE RSP	🟩	🟩	🟩	🟩	🟩	(3)	(3)	⬛	🟩	🟩
Wrong Bus RSP	🟩	🟩	🟩	🟩	🟩	(3)	(3)	🟩	⬛	🟩
RSP Wrong ADDR	🟩	🟩	🟩	🟩	🟩	(3)	(3)	🟩	🟩	⬛

LEGEND:

🟩	=	Supported Concurrent operation
🟨	=	See Notes for Compatibility
🟥	=	Invalid Concurrent operation

Notes:

1. User must ensure errors do not occur within removed words due to the Word Count error.
2. User must ensure that these errors do not occur on the removed bits due to Bit Count Error.
3. These errors are invalid if NO RSP A and NO RSP B are enabled.

Figure 36. AceXtreme RT Error Injection

3.3.8 Amplitude

The AceXtreme C SDK allows for software programmable transceivers on the Multi-Function AceXtreme devices which have variable voltage transceivers. The AceXtreme C SDK has the ability to set the amplitude on the variable voltage transceiver with the function **acexSetAmplitude().** The function **acexGetAmplitude()** will return the current value of the software programmable transceivers.

3.3.9 Self-Test Capabilities (ACE_MODE_TEST)

The AceXtreme C SDK defines a Self-Test (TEST) mode to allow numerous diagnostic tests to be executed on the DDC hardware. Each test has a specific function and must be run independently.

> ***Note:*** *All Tests are memory destructive and will delete any configuration options and/or unconsumed 1553 data residing on the DDC hardware. In addition, while in Test Mode, the device will not be active on the 1553 bus.*

After executing, every Self-Test function will return a TESTRESULT structure. This structure will inform the user if the test passed or failed. If the test failed, expected and actual data at the point of failure is supplied for additional debugging.

```
/* Test result structure /
typedef struct TESTRESULT
{
  U16BIT wResult;        /* pass or fail code */
  U16BIT wExpData;       /* expected data on fail */
  U16BIT wActData;       /* actual data on fail */
  U16BIT wAceAddr;       /* address of failure*/
  U16BIT wCount;         /* test count index */

} TESTRESULT
```

Code Example 114. TESTRESULT Structure

3.3.9.1 Testing Hardware Registers

The AceXtreme C SDK supplies a test to verify that the DDC hardware registers are operating properly. This test can be executed via the **aceTestRegisters()** function.

```
S16BIT nResult;
TESTRESULT sTest;

/* Test card registers */
nResult = aceTestRegisters(
    0,            /* LDN */
    &sTest);      /* Test Results Storage */

if(nResult)
    printf(“aceTestRegisters Error: Code %d\n”, nResult);

/* Display Test results */
if(sTest.wResult == ACE_TEST_PASSED)
{
  printf("Registers Passed test.\n");
}
else
{
  printf("Register Failed test, expected=%04x
          read=%04x!!!\n",sTest.wExpData, sTest.wActData);
}
```

Code Example 115. Running a Hardware Register Test

3.3.10 Testing Hardware Memory

The AceXtreme C SDK supplies a test to verify that the DDC hardware memory can be accessed and read back correctly. The user has the ability to supply a 16-bit verify pattern. This test can be executed via the **aceTestMemory()** function.

```
S16BIT nResult;
TESTRESULT sTest;

/* Test card memory */
nResult = aceTestMemory(
    0,          /* LDN */
    &sTest,     /* Test Results Storage */
    0xAA55);    /* Pattern to Test */

if(nResult)
  printf(“aceTestMemory Error: Code %d\n”, nResult);

/* Display Test results */
if(sTest.wResult == ACE_TEST_PASSED)
{
  printf("Ram Passed %04x test.\n",0xAA55);
}
else
{
  printf("Ram Failed %04x test, data read = %04x addr =
          %d!!!\n", 0xAA55, sTest.wActData, sTest.wAceAddr);
}
```

Code Example 116. Running a Hardware Memory Test

3.3.10.1 Testing 1553 Protocol

The AceXtreme C SDK supplies a test to verify that the DDC hardware can properly transmit and receive 1553B protocol. The device is placed into internal loopback and 1553 traffic is sent, received and verified. This test can be executed via the **aceTestProtocol()** function.

```
S16BIT nResult;
TESTRESULT sTest;

/* Test card protocol */
nResult = aceTestProtocol(
    0,                     /* LDN */
    &sTest);               /* Test Results Storage */

if(nResult)
  printf(“aceTestProtocol Error: Code %d\n”, nResult);

/* Display Test results */
if(sTest.wResult == ACE_TEST_PASSED)
{
  printf("Protocol Unit Passed test.\n");
}
else
{
  printf("Protocol Unit Failed test, expected=%04x read=%04x
          addr=%04x!!!\n", sTest.wExpData, sTest.wActData,
          sTest.wAceAddr);
}
```

Code Example 117. Running a 1553 Protocol Test

3.3.10.2 Testing Hardware Interrupts

The AceXtreme C SDK supplies a test to verify that the DDC hardware interrupt can be properly generated and accessed by the SDK. This test can be executed via the **aceTestIrqs()** function.

```
S16BIT nResult;
TESTRESULT sTest;

/* Test card Interrupts */
nResult = aceTestIrqs(
    0,             /* LDN */
    &sTest);       /* Test Results Storage */

 if(nResult) printf(“aceTestIrqs Error: Code %d\n”, nResult);

/* Display Test results */
if(sTest->wResult == ACE_TEST_PASSED)
{
  printf("Interrupt Occurred, Passed test.\n");
}
else
{
  printf("Interrupt Test Failure, %s %s!!!\n",
         (sTest.wCount&1)?"NO TimeTag Rollover":"",
         (sTest.wCount&2)?"NO IRQ":"");
}
```

Code Example 118. Running a Hardware Interrupt Test

3.3.10.3 Running Hardware Vectors

The AceXtreme C SDK supplies the ability to run DDC Vector Tests. These tests (encapsulated in a file having a “VEC” extension) are used to run more detailed and extensive testing. DDC supplies a standard Vector File (test.vec) for this purpose. Vector files can be executed via **aceTestVectors()** function.

```
S16BIT nResult;
TESTRESULT sTest;

/* Run Hardware Vectors */
nResult = aceTestVectors(
    0,                   /* LDN */
    &sTest,              /* Test Results Storage */
    "test.vec");         /* Vector File to Use */

if(nResult)
    printf(“aceTestVectors Error: Code %d\n”, nResult);

/* Display Test results */
if(sTest.wResult == ACE_TEST_PASSED)
{
  printf("Vectors Passed, EOF at line #%d.\n",sTest.wCount);
}
else
{
  printf("Vectors Failed!\n");
  printf("Failure...at line #%d\n",sTest.wCount);
  printf("          location=%s\n",
(sTest.wResult==ACE_TEST_FAILED_MVECTOR)?"memory":"register");
  printf("          address=%04x\n",sTest.wAceAddr);
  printf("          expected=%04x\n",sTest.wExpData);
  printf("          actual=%04x\n",sTest.wActData);
}
```

Code Example 119. Running Hardware Vectors

4 INCLUDED DEMOS

This software package is supplied with many examples of the use of the SDK and the capabilities of the hardware. The examples provided demonstrate the six main categories for use of the SDK. The six categories are BC mode, RT mode, MT mode, RTMT mode, MT-I mode, and RTMT-I mode.

In all cases, the examples have been provided as source codes with an appropriate Make file that may be used to build the executable.

The Demo programs are split into three groups:

- General – These programs will operate on all DDC 1553 Hardware
- AceXtreme – These programs will operate on AceXtreme family DDC 1553 hardware for both Single-Function and Multi-Function
- AceXtreme MF – These programs will only operate on AceXtreme Multi-Function 1553 Hardware.

4.1 General Demo Programs

4.1.1 AIO.c

This application demonstrates the use of the avionic discrete ports found on supported DDC cards. It runs four tests on each of the avionic discrete channels.

The first test verifies that each avionic discrete direction can be set internally as an input or output. The second will check the input level of each channel, as it is expected to default to high. The third test will check the output level of each channel, which is expected to be low by default. Finally this application will set the direction of communication, and set the level for each output channel. The connected input channel will then be read to test the communication. If the input level is not read as the output level, an error will be displayed.

This demo works without an external loopback connection, as these devices are equipped with internal sensing circuitry specifically to perform these tests. They internally connect the first half of the I/O channels directly to the second half of the I/O channels.

4.1.2 BcAsync.c

This sample demonstrates the ability of sending out three asynchronous messages in the low priority mode. This means that messages will be inserted at the end of a minor frame. The user sets the logic device used for the BC and chooses how they want to send the asynchronous messages. The user will see a few lines of BC to RT messages and then a series of asynchronous messages.

4.1.3 BCAsync2.c

This sample demonstrates the ability of sending out three asynchronous messages in the high priority mode. The user will see three different asynchronous messages transmitted on the bus.

4.1.4 BCDemo.c

This demonstration program creates a basic BC to RT message that uses a single data block of 32 words. Two opcodes are then created. One opcode is the XEQ (execute) opcode that will cause the BC message to be transmitted over the bus. The second opcode is created as a jump to minor frame by using the CAL (call) instruction.

The opcode 1 is then placed in a minor frame and opcode 2 is entered in the major frame. The major frame is run and will call the minor frame that contains the XEQ opcode. The XEQ opcode is tied to a message through the **aceBCOpCodeCreate()** function. One of the input parameters is the ID of the message that was previously created with the **aceBCMsgCreateBCtoRT()** function.

When the major frame is run, it will run the major frame forever and will never halt. The frame will run forever because it was specified to do so in the **aceBCStart()** function.

4.1.5 BCDBuf.c

This demonstration program creates three messages that are included into one minor frame. The three messages are BC to RT, RT to BC, and RT to RT. A fourth opcode is created that will call minor frame 1. Message 1, 2, and 3 are attached to minor frame 1. The major frame will call minor frame 1 using opcode 4 and then run message 2 by attaching opcode 2.

When the major frame is run, it will run the major frame forever and will never halt. The frame will run forever because it was specified to do so in the **aceBCStart()** function.

If a ‘0’ is pressed, then the keystroke routine will quit. The program will enter a data display mode until any key is pressed. Finally, the BC will halt, and all allocated memory will be freed.

This sample also utilizes the function **aceBCBreateImageFiles()** that would allow for a user to create an image file for their code.

4.1.6 RTDBuf.c

This program creates an RT stack file in ASCII text using all messages read from the hardware.

A user buffer is created and initialized. The RT address is set to 5 and the buffer is attached to a double-buffered data block. This data block is then attached to Subaddress 1 for transmit command and to Subaddress 2 for receive commands that the RT receives.

After these steps, the RT is set to the run state and the program enters a loop looking for keystrokes. If a ‘0’ is input, the program halts, if any other number (1 through 9) is input, the number will be placed in the RT buffer for transmission.

4.1.7 RTMode.c

This demo program initializes the device to operate as an RT using the Logical Device number that is input in the command line parameters. It then sets the RT address to the value input by the user. After setting the address, the program attaches a double-buffered data block to Subaddress 1.

The software will read the synchronize MODE code data by using the **aceRTModeCodeReadData()** function in a loop. The loop will exit and the device will close if the ENTER key is pressed.

4.1.8 RTMTDemo.c

This program demonstrates the minimum setup needed to run the device simultaneously as an RT and a Monitor. The program initializes the device and sets the RT/MT address to the address specified by the user. The user sets the RT address, as well as the RT subaddress. It starts the Remote Terminal / Monitor and then stops both. This program gets messages from the RT hardware stack and the MT hardware stack. In RT/MT mode of operation, the monitor will monitor the entire 1553 data bus except for its own RT address. The MT stack on the hardware will contain all contents of the data bus except anything received or sent by the device’s assigned RT address. This is a function of the device and cannot be changed. When using the AceXtreme C SDK**,** some post processing is performed to combine the MT stack and the RT stack into this one RT/MT host buffer, which will contain all monitored messages and data on the 1553 data bus.

If using this SDK in RT/MT mode without a host buffer installed, the MT stack will contain all monitored data on the 1553 data bus, except for the RT defined for that particular channel.

4.1.9 MTPoll.c

This program demonstrates the setup and operation necessary to run the device in the Monitor mode. Interrupts are not used in this example; instead the software periodically polls the monitor stack and transfers the message data received to a user buffer. This program initializes the hardware to the monitor mode.

The user may choose to set up an RT filter and then choose which RT to filter, which subaddres to filter, and which data type to filter. The monitored data is displayed on the terminal. If any key is pressed, the program will remove any dynamically allocated buffers and quit.

4.1.10 MTIrq.c

This program demonstrates the setup and operation of the device as a Monitor operating with interrupts. The program will monitor the bus and display the data to the screen. If a file is specified, the data will be stored to disk.

The device is initialized to Monitor Mode. After initialization, the storage file will open if it is defined in the parameter list. The MT stack is created based on the memory size of the card. The software allocates a host buffer based on the size of the Monitor stack which must be at least three times greater than the number of messages that can be stored in the command stack * 40 (fixed length messages). The Monitor is then started and the captured data will be displayed on the terminal and sent to a file based on the parameter list.

4.1.11 DIO.c

This application demonstrates the use of the digital discrete ports found on supported DDC cards. It utilizes access of the digital discretes in conjunction with initialization of MIL-STD-1553 functionality. Moreover, it tests each digital discrete one at a time.

This application will set the direction of communication, and set the level for the output channel. The input channel will then be read to test the communication. The process will then be repeated with the channel‘s direction reversed.

This demo works with a loopback connection; with the first half of the I/O channels connected directly to the second half of the I/O channels.

4.1.12 DIOALL.c

This application demonstrates the use of the digital discrete ports found on supported DDC cards. It utilizes access of the digital discretes independent of MIL-STD-1553 functionality. This functionality can be initialized and/or used without the use of MIL-STD-1553 functionality. Moreover, it tests all of the digital discretes at the same time.

This demo works with a loopback connection; with the first half of the I/O channels connected directly to the second half of the I/O channels.

This application will set the direction of communication, and set the level for the output channel. The input channel will then be read to test the communication. The process will then be repeated with the channel‘s direction reversed.

4.1.13 Irigdemo.c

This sample demonstrates the use of IRIG 106 Chapter 10 Monitor (MTi) mode. The main function illustrates how to configure resources to receive IRIG 106 Chapter 10 time packets. These packets will be generated based on an internal or external IRIG106 time source.

4.1.14 Looptest.c

This sample performs a wrap around self-test of the 1553 device channel. Two different tests are run based on the device under test:

For EMACE devices, the test transmits a word on one channel and receives the transmitted word on the other channel.

For E^{2}MA and AceXtreme devices, the test connect the Channel A receive lines to the Channel B receiver and operates the BC with message. This exercises the BC on-line loopback test as the failure test.

This test require that a loop back cable be used which will connect channel A to channel B with the appropriate termination. If the expected data is received (EMACE devices) or the on-line loopback test does not fail (AceXtreme devices), then the test returns a PASS.

4.1.15 Mti2disk.c

This sample demonstrates the use of IRIG 106 Chapter 10 Monitor (MTi) mode. It saves the monitored data packets to a file in raw IRIG106 Chapter 10 format. When running this sample, the user inputs the device number and configures the device to monitor a 1553 Bus and saves the packets.

4.1.16 Mti2disk2.c

This sample acts in the same manner as MTi2Disk. This also actively displays the amount of packets intercepted. The file generated is the same as the MTi2Disk sample.

4.1.17 Mtidemo.c

This sample demonstrates the use of IRIG 106 Chapter 10 Monitor (MTi) mode. The main function illustrates how to configure resources to receive IRIG 106 Ch10 data packets. The user inputs a logical device number for the MTi and a value for Ch10 IRIG ID. This sample does not store any data into a file for later review.

4.1.18 Mtiread.c

This sample reads and converts a saved IRIG106 Chapter 10 binary file into a readable decoded message. It exports the message to an ASCII text file.

4.1.19 MtiRead2.c

This sample reads and converts a saved IRIG106 Chapter 10 binary file into a readable decoded message and prints the message to screen as well as exporting the message to an ASCII text file.

4.1.20 RTirq.c

This sample demonstrates how using a host buffer in RT mode decreases the number of messages lost, as opposed to polling the stack.

4.1.21 RTMTiDemo.c

This sample demonstrates the RTMTi on one channel. The user sets the logical device number, the Ch10 channel ID, and an RT address. The sample then returns the amount of messages at a rate of every 10 seconds.

4.1.22 RTPoll.c

This sample uses simple polling to read messages off the RT stack. The user specifies the RT address and the RT subaddress. The RT will then display all the data send from an external BC in a decoded format.

4.1.23 Tester.c

This application tests all hardware and software components (registers, hardware memory, protocol, interrupts, and vectors) to make sure the device is in full working order. Any problems that arise are reported back in descriptive errors. E2MA and AceXtreme devices do not support the Protocol and Vector test. These tests will return with a failure error message of “Function not supported”.

4.2 AceXtreme Demo Programs

4.2.1 Aesdemo.c

The Aesdemo demonstrates the use of the AceXtreme MTi monitor in Advanced Error Sampling (AES) mode. When the MTi channel is placed into AES mode, a bit by bit sampling of the bus is recorded in the data packets when an error is detected. The sample shows how to configure AES mode, retrieve packets and store them to a CH10 file.

4.2.2 Bcmti.c

This sample demonstrates the combination of the IRIG 106 Chapter 10 Monitor (MTi) and Bus Controller (BC) modes working in concurrently on the same channel. The BC device will transmit a predefined amount of data and then the MTi will compare the message count and data to the expected values.

4.2.3 MTIedemo.c

This sample demonstrates the use of IRIG 106 Chapter 10 Monitor (MTi) mode, while the MTi channel is configured to use the MTi Error monitor. The main function illustrates how to configure resources to receive IRIG 106 Ch10 data packets. The user inputs a logical device number for the MTi and a value for Ch10 IRIG ID. This sample does not store any data into a file for later review.

4.2.4 MRTMTi.c

This sample demonstrates the Multi-RT (MRT) and IRIG 106 Chapter 10 Monitor (MTi) operating concurrently on one channel. BC mode is run on another channel to allow for simultaneous monitoring and for purposes of an integrity check at the end.

This sample will run until the user terminates it. While the sample is running, three threads will display three different sources of information, one thread retrieves BC Host Buffer Data, another retrieves the monitored Data Packets in an IRIG 106 chapter 10 format, and the final retrieves MRT Message Data. If the message count is incorrect, the system will return an error.

4.2.5 RTDataArray.c

This sample demonstrates the use of Static Data Arrays for RT Mode.

4.2.6 DataArray.c

This sample application demonstrates the transmission data arrays in BC Mode.

4.2.7 DataStrm.c

****Note**: This sample requires four physical 1553 channels*

This sample application demonstrates the transmission of a bulk data transfer. The program will show all of the necessary API calls to configure buffers, and stream the data.

4.2.8 Mrtdemo.c

This sample demonstrates the use of the Multi-RT mode with an AceXtreme device. The sample will configure and run several RTs. The program will ask the user to specify which RTs the AceXtreme device should emulate.

4.3 AceXtreme MF Samples

4.3.1 BCei .c (AceXtreme MF only)

This sample application demonstrates the use of BC error injection via a menu driven interface. The user can select between word count, bit count, glitch inverse or gap errors to inject into a message.

Three BC to RT messages will be created with the selected error injected into message #2.

The application demonstrates the configuration of the error parameters as well as the enabling of such within a message for transmission. It will then retrieve the transmitted messages from either the stack or host buffer as per user selection.

4.3.2 BCIMR.c (AceXtreme MF only)

This sample application demonstrates the use of BC inter-message routines via a menu driven interface. Inter-message routines are operations that are performed between the execution of two messages within a frame. The available inter-message routines are described in Table 28.

This application uses both BC and multi-RT mode. It will create a frame consisting of two BC to RT messages and the inter-message routine selected by the user which will be executed immediately after the first message. The sample also initializes RT1 and RT2 to demonstrate the interaction between the BC and these remote terminals with certain inter-message routines.

Upon completion of this sample, a description of the expected results is displayed on the console. The user can compare this with the actual results displayed during operation.

4.3.3 BCMemobj .c (AceXtreme MF only)

This application demonstrates the use of BC memory objects to implement conditional messaging. BC memory objects are created, and manipulated depending on the user's choices to control the transmission of messages.

In this example three messages are created, the first of which is always transmitted. Each time a frame is executed, the two previously created memory words will be compared. If they are not equal, the second memory word is incremented. If they are equal, the second and third messages are also transmitted within that frame. Also if the words are equal, the second memory word is reset back to zero.

The memory words are initialized to 5 and 10; therefore the user will observe the additional messages transmitted every fifth frame. This is verified successful by the sample by comparing memory words and the number of messages transmitted. The results are also displayed on the console.

4.3.4 BCMRT.c (AceXtreme MF only)

***** Note** – This demo requires an additional channel to act as a bus monitor ***

This sample application will demonstrate the use of concurrent bus controller and multi-RT on a single channel.

It configures the BC to send three BC to RT messages in a synchronous frame until the user stops it by key press. It also configures three concurrent remote terminals to receive these three messages.

During execution of the sample, the number of messages sent by the BC and the number of messages received by the RTs and Monitor are compared and displayed to the console to ensure that the messages transmitted are as expected.

4.3.5 BCMRTMTI.c (AceXtreme MF only)

This sample application will demonstrate the use of concurrent bus controller, multi-RTs and a bus monitor on a single channel.

It configures the BC to send three BC to RT messages in a synchronous frame until the user stops it by key press. It also configures three concurrent remote terminals to receive these three messages.

During execution of the sample, the number of messages sent by the BC and the number of messages received by the RTs and Monitor are compared and displayed to the console to ensure that the messages transmitted are as expected.

4.3.6 BCOpcode.c (AceXtreme MF only)

**** Note** – This demo requires an additional channel to act as a bus monitor ***

The sample application demonstrates the capability of opcode modification for changing messages or frames during runtime. The steps to run a frame, not run a frame, modify a message, and modify a frame are demonstrated in this application. The device is initialized in concurrent BC and MRT modes.

Seven messages are created of which the first three are inserted into minor frame #1, and the second three are inserted into minor frame #2. The seventh message is left unassigned for use later. A major frame is initially created containing minor frame #1. During the execution of this program, the major frame is modified to use minor frame #1 by opcode modification. Also during the execution, message #7 is used to replace other messages originally inserted into each minor frame. The capability to start and stop running a specific frame by modification of opcodes is also demonstrated in this sample.

At the conclusion of this program, verification is performed by comparison of the number of BC messages transmitted to messages received by the bus monitor.

4.3.7 BCTime.c (AceXtreme MF only)

This example shows a user how to take advantage of different BC message timing functions. It demonstrates how to configure messages using “Inter-Message Gap Time” versus “Time to Next Message” parameters.

It configures the BC to send three BC to RT messages in a synchronous frame until the user stops it by key press. It also configures three concurrent remote terminals to receive these three messages. However, the user is given the choice via menu to select the timing mechanism as either Time to Next Message or Inter-Message Gap Time. The user must select whether the resolution is 1 microsecond or 100 microseconds.   The bus controller frames are then configured according to the user's selection.

4.3.8 DBCDEMO.c (AceXtreme MF only)

** Note this sample requires two physical channels **

This application is an example of the method used to initiate the transfer of bus control from a bus controller to a remote terminal.

The program will configure two separate channels in concurrent BC/multi-RT mode. It will alternate between transferring control from the BC of the first channel to an RT on the second, and vice versa.

4.3.9 MRTEI.c (AceXtreme MF only)

This sample application demonstrates the use of RT error injection via a menu driven interface. The user can select between word count, bit count, glitch inverse or gap errors to inject into a message.

The application demonstrates the configuration of the error parameters as well as the enabling of such within a message for transmission. It will then retrieve the transmitted messages from the host buffer and display to the console.

4.3.10 MTRDemo.c (AceXtreme MF only)

This sample application acts as an “out of the box” bus recorder. It demonstrates the functionality required to monitor the 1553 bus and record all traffic including errors to a file. This file can be replayed back to the bus at a later time as demonstrated by the replaydemo sample application.

4.3.11 ReplayDemo.c (AceXtreme MF only)

This sample application demonstrates use of the replay engine in BC mode to playback previously recorded 1553 traffic on the bus. This demo can replay a file created by the above described mtrdemo sample application.

4.3.12 Resptime.c (AceXtreme MF only)

This sample application demonstrates the capability of configuring both the RT response timeout, and the RT response for a given RT. It is a very simplistic example of setting up multi-RT mode and configuring the timing parameters via API calls.

4.3.13 Trigdio.c (AceXtreme MF only)

*** **Note** this sample requires the following DIO connections be wired:*

DIO1 -> DIO5

DIO2 -> DIO6

DIO3 -> DIO7

DIO4 -> DIO8

This sample application loops through setting the level of each the discrete I/O to high which will trigger an interrupt event associated with the DIO it is connected to, as per the description above.

For example DIO#1 is set to high, thus triggering an interrupt event configured to DIO#5.

4.3.14 Trigger.c (AceXtreme MF only)

This sample application will demonstrate the API calls necessary to configure, and utilize the available triggers and events on the AceXtreme device. It will loop through all of the possible triggers, and for each one trigger all of the possible events associated with each. Please refer to section 3.2.7 for a description of the available triggers and events.

4.3.15 Voltage.c (AceXtreme MF only)

This example application configures a simple bus controller that transmits one frame containing one message twenty times. The program allows the user to enter a desired voltage at which to transmit the messages.

This demonstrates the API's necessary to get the current voltage, as well as setting the voltage to a desired value.

EMACE PLUS SDK MANUAL RECORD OF CHANGE

Revision	Date	Pages	Description
A	10/2007	All	Initial Release
B	12/2007	All	Replacement of aceDioCtl, aceDioDir, and aceDioBits. Edits to table 48.
C	12/2007	121, 571,	Added note and changed function name
D	2/2008	428	Edited ACE_OPCODE_FLG
E	3/2008	All	Added new functions throughout document
F	6/2008	All	Added new function in general function section
G	9/2008	49, 52, 183, 702	Warning added for global time-tag on pgs. 42 and 183. changed web link to irig106.org on pg. 49, Changed the return values on pg. 702.
H	11/2008	41, 14, 693, 427, 356	Removed line from code example on page 41. Fixed minor error on page 14. Added dwOptions to page 693. Changed dwDualType to dwMsgOptions on page 356. Removed ACE_MSGOPT_DOUBLE_BUFFER once on page 427

Data Device Corporation

Leadership Built on Over 50 Years of Innovation

Military | Commercial Aerospace | Space | Industrial

Data Device Corporation (DDC) is the world leader in the design and manufacture of high-reliability data bus products, motion control, and solid-state power controllers for aerospace, defense, and industrial automation applications. For more than 50 years, DDC has continuously advanced the state of high-reliability data communications and control technology for MIL-STD-1553, ARINC 429, AFDX®, Synchro/Resolver interface, and Solid-State Power Controllers with innovations that have minimized component size and weight while increasing performance. DDC offers a broad product line consisting of advanced data bus technology for Fibre Channel networks; MIL-STD-1553 and ARINC 429 Data Networking cards, components, and software; Synchro/Resolver interface components; and Solid-State Power Controllers and Motor Drives.

Product Families

Data Bus | Synchro/Resolver Digital Conversion| Power Controllers | Motor Controllers

DDC is a leader in the development, design, and manufacture of highly reliable and innovative military data bus solutions. DDC's Data Networking Solutions include MIL-STD-1553, ARINC 429, AFDX®, Ethernet and Fibre Channel. Each Interface is supported by a complete line of quality MIL-STD-1553 and ARINC 429 commercial, military, and COTS grade cards and components, as well as software that maintain compatibility between product generations. The Data Bus product line has been field proven for the military, commercial and aerospace markets.

DDC is also a global leader in Synchro/Resolver Solutions. We offer a broad line of Synchro/Resolver instrument-grade cards, including angle position indicators and simulators. Our Synchro/Resolver-to-Digital and Digital-to-Synchro/Resolver microelectronic components are the smallest, most accurate converters, and also serve as the building block for our card-level products. All of our Synchro/Resolver line is supported by software, designed to meet today's COTS/MOTS needs. The Synchro/Resolver line has been field proven for military and industrial applications, including radar, IR, and navigation systems, fire control, flight instrumentation/simulators, motor/motion feedback controls and drivers, and robotic systems.

As the world's largest supplier of Solid-State Power Controllers (SSPCs) and Remote Power Controllers (RPCs), DDC was the first to offer commercial and fully-qualified MIL-PRF-38534 and Class K Space-level screening for these products. DDC's complete line of SSPC and RPC boards and components support real-time digital status reporting and computer control, and are equipped with instant trip, and true I^2T wire protection. The SSPC and RPC product line has been field proven for military markets, and are used in the Bradley fighting vehicles and M1A2 tank.

DDC is the premier manufacturer of hybrid motor drives and controllers for brush, 3-phase brushless, and induction motors operating from 28 Vdc to 270 Vdc requiring up to 18 kilowatts of power. Applications range from aircraft actuators for primary and secondary flight controls, jet or rocket engine thrust vector control, missile flight controls, to pumps, fans, solar arrays and momentum wheel control for space and satellite systems.

Certifications

Data Device Corporation is ISO 9001: 2008 and AS 9100, Rev. C certified.

DDC has also been granted certification by the Defense Supply Center Columbus (DSCC) for manufacturing Class D, G, H, and K hybrid products in accordance with MIL-PRF-38534, as well as ESA and NASA approved.

Industry documents used to support DDC's certifications and Quality system are: AS9001 OEM Certification, MIL-STD-883, ANSI/NCSL Z540-1, IPC-A-610, MIL-STD-202, JESD-22, and J-STD-020.

UL REGISTERED FIRM

DATA DEVICE CORPORATION
REGISTERED TO:
ISO 9001:2008, AS9100C:2009-01
EN9100:2009, JIS Q9100:2009
FILE NO. 10001296 ASH09

The first choice for more than 50 years—DDC
DDC is the world leader in the design and manufacture of high reliability data interface products, motion control, and solid-state power controllers for aerospace, defense, and industrial automation.

Inside the U.S. - Call Toll-Free 1-800-DDC-5757

Headquarters and Main Plant
105 Wilbur Place, Bohemia, NY 11716-2426
Tel: (631) 567-5600 Fax: (631) 567-7358
Toll-Free, Customer Service: 1-800-DDC-5757

Web site: www.ddc-web.com

Data Device Corporation

Outside the U.S. - Call 1-631-567-5700

United Kingdom: DDC U.K., LTD
James House, 27-35 London Road, Newbury,
Berkshire RG14 1JL, England
Tel: +44 1635 811140 Fax: +44 1635 32264

France: DDC Electronique
84-88 Bd de la Mission Marchland
92411 Courbevoie Cedex, France
Tel: +33-1-41-16-3424 Fax: +33-1-41-16-3425

Germany: DDC Elektronik GmbH
Triebstrasse 3, D-80993 München, Germany
Tel: +49 (0) 89-15 00 12-11
Fax: +49 (0) 89-15 00 12-22

Japan: DDC Electronics K.K.
Dai-ichi Magami Bldg, 8F, 1-5, Koraku 1-chome,
Bunkyo-ku, Tokyo 112-0004, Japan
Tel: 81-3-3814-7688 Fax: 81-3-3814-7689
Web site: www.ddcjapan.co.jp

Asia: Data Device Corporation - RO Registered in Singapore
Blk-327 Hougang Ave 5 #05-164
Singapore 530327
Tel: +65 6489 4801

The information in this Manual is believed to be accurate; however, no responsibility is assumed by Data Device Corporation for its use, and no license or rights are granted by implication or otherwise in connection therewith. Specifications are subject to change without notice.