MIL-STD-1553 eTPU Driver Manual

Version 1.1

June 5, 2020

ASH WARE Inc.

Table of Contents

[1 Overview 5](#_Toc42244918)

[1.1 Revision History 5](#_Toc42244919)

[1.2 Tools Versions 5](#_Toc42244920)

[1.3 References 5](#_Toc42244921)

[2 Hardware Interface 5](#_Toc42244922)

[2.1 eTPU Channel Assignment Requirements 6](#_Toc42244923)

[3 eTPU Software 7](#_Toc42244924)

[3.1 Message Processing Functionality 7](#_Toc42244925)

[3.1.1 Monitor Terminal 7](#_Toc42244926)

[3.1.2 Remote Terminal 7](#_Toc42244927)

[3.1.3 Bus Controller 7](#_Toc42244928)

[3.1.4 Error Detection 8](#_Toc42244929)

[3.2 Interface 9](#_Toc42244930)

[3.2.1 Driver Configuration 9](#_Toc42244931)

[3.2.2 Inputs/Outputs 10](#_Toc42244932)

[3.2.3 Interrupts 11](#_Toc42244933)

[3.2.4 Shutdown 12](#_Toc42244934)

[3.3 Software Structure 13](#_Toc42244935)

[3.3.1 Files and Build 14](#_Toc42244936)

[3.4 Performance 15](#_Toc42244937)

[4 Host API Software 15](#_Toc42244938)

[4.1 1553 Driver Instance 15](#_Toc42244939)

[4.2 Interfaces 16](#_Toc42244940)

[4.2.1 eTPU Module 16](#_Toc42244941)

[4.2.2 1553 Driver 17](#_Toc42244942)

[4.3 Interrupt Timing 20](#_Toc42244943)

[4.3.1 Monitor Terminal 20](#_Toc42244944)

[4.3.2 Remote Terminal 24](#_Toc42244945)

[4.3.3 Bus Controller 29](#_Toc42244946)

[5 Testing Artifacts 33](#_Toc42244947)

[5.1 eTPU Tests 33](#_Toc42244948)

[5.1.1 Code Coverage 33](#_Toc42244949)

[5.1.2 Testing-Only Code 33](#_Toc42244950)

[5.2 System Tests 33](#_Toc42244951)

List of Figures

[Figure 1 : Interface between eTPU and MIL-STD-1553 Bus 6](#_Toc42244952)

[Figure 2 : Bus Differential Signal and.Matching Digital Signals 6](#_Toc42244953)

[Figure 3: Host/eTPU Data Interface Overview 10](#_Toc42244954)

[Figure 4 : BC -> RT (Receive) Message Timing Diagram 20](#_Toc42244955)

[Figure 5 : RT-> BC (Transmit) Message Timing Diagram 21](#_Toc42244956)

[Figure 6 : RT -> RT Message Timing Diagram 21](#_Toc42244957)

[Figure 7 : Mode Command (Without Data) Timing Diagram 21](#_Toc42244958)

[Figure 8 : Mode Command (Data Transmit) Timing Diagram 22](#_Toc42244959)

[Figure 9 : Mode Command (Data Receive) Timing Diagram 22](#_Toc42244960)

[Figure 10 : BC -> RT (Receive) Broadcast Message Timing Diagram 23](#_Toc42244961)

[Figure 11 : RT -> RT Broadcast Message Timing Diagram 23](#_Toc42244962)

[Figure 12 : Mode Command (No Data) Broadcast Timing Diagram 24](#_Toc42244963)

[Figure 13 : Mode Command (Data Receive) Broadcast Timing Diagram 24](#_Toc42244964)

[Figure 14 : BC -> RT (Receive) Message Timing Diagram 25](#_Toc42244965)

[Figure 15 : RT -> BC (Transmit) Message Timing Diagram 25](#_Toc42244966)

[Figure 16 : RT -> RT (Transmitter Perspective) Message Timing Diagram 25](#_Toc42244967)

[Figure 17 : RT -> RT (Receiver Perspective) Message Timing Diagram 26](#_Toc42244968)

[Figure 18 : Mode Command (Without Data) Timing Diagram 26](#_Toc42244969)

[Figure 19 : Mode Command (Data Transmit) Timing Diagram 26](#_Toc42244970)

[Figure 20 : Mode Command (Data Receive) Timing Diagram 27](#_Toc42244971)

[Figure 21 : BC -> RT (Receive) Broadcast Message Timing Diagram 27](#_Toc42244972)

[Figure 22 : RT -> RT (Transmitter Perspective) Broadcast Message Timing Diagram 28](#_Toc42244973)

[Figure 23 : RT -> RT (Receiver Perspective) Broadcast Message Timing Diagram 28](#_Toc42244974)

[Figure 24 : Mode Command (No Data) Broadcast Timing Diagram 28](#_Toc42244975)

[Figure 25 : Mode Command (Data Receive) Broadcast Timing Diagram 29](#_Toc42244976)

[Figure 26 : BC -> RT (Receive) Message Timing Diagram 29](#_Toc42244977)

[Figure 27 : RT -> BC (Transmit) Message Timing Diagram 30](#_Toc42244978)

[Figure 28 : RT -> RT Message Timing Diagram 30](#_Toc42244979)

[Figure 29 : Mode Command (Without Data) Timing Diagram 30](#_Toc42244980)

[Figure 30 : Mode Command (Data Transmit) Timing Diagram 31](#_Toc42244981)

[Figure 31 : Mode Command (Data Receive) Timing Diagram 31](#_Toc42244982)

[Figure 32 : BC -> RT (Receive) Broadcast Message Timing Diagram 31](#_Toc42244983)

[Figure 33 : RT -> RT Broadcast Message Timing Diagram 32](#_Toc42244984)

[Figure 34 : Mode Command (No Data) Broadcast Timing Diagram 32](#_Toc42244985)

[Figure 35 : Mode Command (Data Receive) Broadcast Timing Diagram 33](#_Toc42244986)

# Overview

This manual describes the functionality of the ASH WARE MIL-STD-1553 eTPU driver, and how to use it. The driver supports a Bus Controller (BC) configuration, a Remote Terminal (RT) configuration, or a Monitor Terminal (MT, also known as Bus Monitor) that can be addressed or unaddressed. The eTPU is a timing co-processor onboard NXP microcontrollers (MPC55xx, MPC56xx and MPC57xx). The eTPU is capable of managing the bit-level signal encoding and decoding required for MIL-STD-1553, as well as the low-level message protocol handling. This minimizes the workload on the host processor, which minimizing the amount of expensive external hardware needed to interface to the 1553 data bus. This guide assumes the reader is familiar with the eTPU and an expert on MIL-STD-1553.

The first section of this document describes the hardware interface – the signals expected and generated at the MCU (eTPU) pins, and the requirements for eTPU channel layout. Note that this driver functions at the level of interfacing to a single 1553 data bus. To meet the needs of a full dual-redundant MIL-STD-1553 data bus, two instances of this driver must be configured, and user software must manage dual-bus interaction.

The middle sections of the document describe the eTPU low-level driver software and the host API layer software that serves as the intermediary between user software and the eTPU. The final section of the document describes the simulation testing that is part of the driver package.

## Revision History

Version 1.0 5/5/2020 - initial document

Version 1.1 6/5/2020 - support BC terminal type, general improvements

## Tools Versions

This software release and associated testing makes use of the following tools and versions:

* ETEC Compiler, version 2.62C
* eTPU2+ Development Tool, version 2.72C
* System Development Tool, version 2.72C

## References

Enhanced Time Processing Unit (eTPU) Preliminary Reference Manual (ETPURM/D 5/2004 REV 1), NXP

MIL-STD-1553 Designer’s Guide, Sixth Edition, ILC Data Device Corporation

# Hardware Interface

The MIL-STD-1553 data bus physical structure is a twisted shielded pair of lines, with data communicated via differential voltage. Connected to this bus are three possible terminal types – bus controllers, remote terminals and monitor terminals. The external hardware consists for a direct coupled or transformer coupled interface and hardware to convert between the differential bus voltage encoded data and digital Manchester II encoded data. This is illustrated in the figures below. The “P” and “N” designators refer to the positive and negative polarity signals. The “A/B” designator refers to the signal being either bus A or B, in the dual redundant scheme.

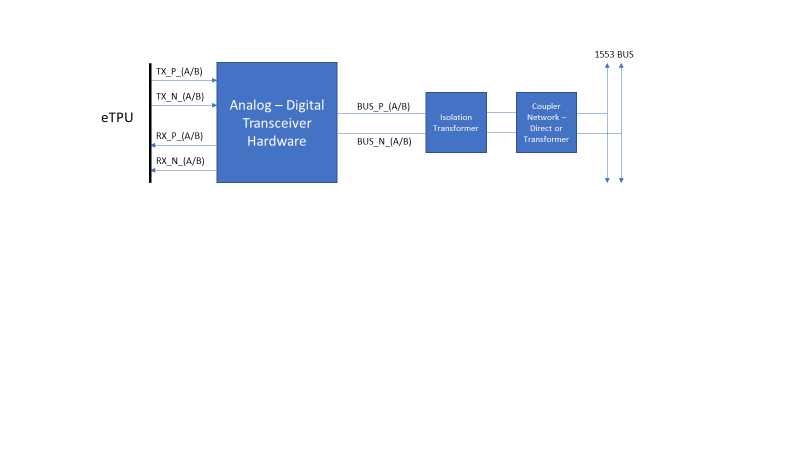


Figure : Interface between eTPU and MIL-STD-1553 Bus

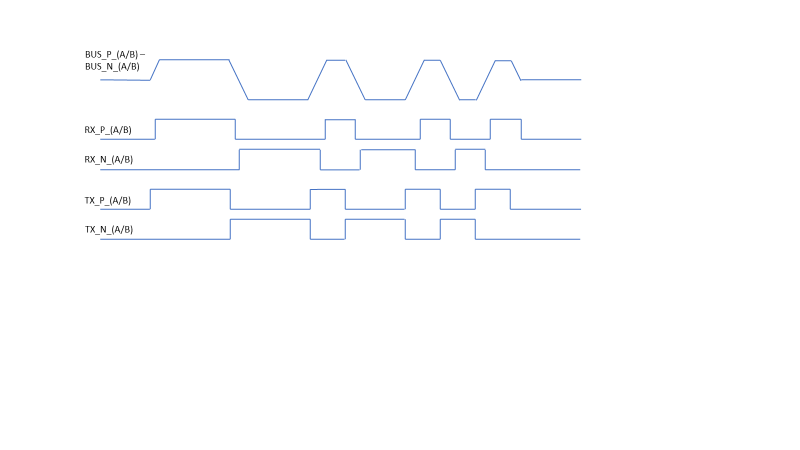


Figure : Bus Differential Signal and.Matching Digital Signals

Note that a Monitor Terminal does not generate digital TX\_P/TX\_N signals – it is a receiver only.

## eTPU Channel Assignment Requirements

Each digital RX pair, or TX pair, is constrained to certain eTPU channel assignments in order to simplify the software and make it more efficient. In both cases, the pairs must be consecutive channels. So for example, an RX pair could be assigned to eTPU-A channels 7 and 8, but a TX pair could not be assigned to eTPU-B channels 13 and 20. The RX\_P or TX\_P is signal must be connected to the lower channel of a pair – these lower channels are sometimes referred to as the “master” channel of a pair. A remote terminal or bus controller consists of both an RX pair and a TX pair. These pairs must be part of the same eTPU module (AB, or in the case of an MPC5676R or MPC5777C, module C), but can be on different eTPU engines. However, it is recommended for performance reasons that they be on the same eTPU engine.

The driver performance is such that for eTPU modules running at a clock rate under 200MHz (MPC55xx, MPC56xx), an eTPU engine can support a single driver instance. Thus, to support a dual-redundant 1553 solution, a driver must be assigned to each of two eTPU engines. For MPC57xx parts, with the eTPU configured to run at 200MHz or higher, two driver instances can run on a single eTPU engine. Note that in either case, an eTPU engine with a 1553 driver can perform only very limited other functions. Performance is discussed on more detail in a later section.

# eTPU Software

The eTPU software driver implements the MIL-STD-1553 protocol for a single data bus. It provides the capability and memory storage to process one complete MIL-STD-1553 message. The driver may be configured as a Bus Controlle (BC), Remote Terminal (RT), or Monitor Terminal (MT). The MT can be addressed (only receives messages involving its address), or unaddressed (receives all valid messages on the bus). The eTPU portion of these 1553 drivers is described in more detail in the sections below.

## Message Processing Functionality

The message processing capabilities for each terminal type is described in this section.

### Monitor Terminal

When an MT 1553 eTPU driver instance is initialized, it begins by waiting to see an initial idle period on the bus. Once this has succeeded, the driver awaits detection of the first sync of a message. The message processing of the driver is capable of storing the contents of one complete error-free message. Thus for most purposes, the host processor utilizing an MT eTPU driver may only listen to message complete (and error) interrupts. However, all the interrupt types are supported by the MT driver. More details on interrupt timing and the various message types is provided in the host software Interrupt Timing section. Following a message complete interrupt, the host processor has at least one word time (20 us) in order to read all of the message contents, before they potentially begin to be overwritten by a new message.

### Remote Terminal

When an RT 1553 eTPU driver instance is initialized, it begins by waiting to see an initial idle period on the bus. Once this has succeeded, the driver awaits detection of the first sync of a message. Once a command word (or command words in the case of an RT-RT message), and the driver determines that this RT needs to be involved in the message (terminal address match, or a broadcast), then a command valid interrupt is generated (if enabled). This gives the host processor the opportunity to respond appropriately to the command, depending upon the message type. If the message calls for the RT to respond with status, and optionally data (mode or regular message), the host processor should update those input parameters appropriately – the eTPU RT driver will begin to output its response based upon configured timing, regardless of whether the host has written new data or not. More details on interrupt timing and the various message types is provided in the host software Interrupt Timing section.

### Bus Controller

Once a BC 1553 eTPU driver instance is initialized, it is readying to start sending messages. The driver does have a feature that allows it to test the bus for an idle state, if that is necessary. A message is sent by first writing the command word(s), the message type, and if the message type includes receive data, the data word(s) to the eTPU. Then a host service request begins the transmission process. Interrupts (if enabled) during the message processing provide feedback on the message status. All messages end with a message complete (or error) interrupt. More details on interrupt timing for the various message types is provided in the host software Interrupt Timing section.

### Error Detection

There are two basic kind of errors detected by the software. The first category involves issues found in the low-level transmission – parity errors or missing bits. The second category involves protocol issues:

*Too few data words* : A gap/idle is detected before the number of words specified by the command word have been received.

*Expected idle (gap) missing* : A gap/idle is expected, but activity on the bus has been detected too soon.

*Data received when status expected* : Occurs when given message protocol, a status word is expected, but instead a data word has been received.

*Status received when data expected* : Occurs when given message protocol, a data word is expected, but instead a status/command word has been received.

*Timeout waiting for status* : The RT status response maximum time has been exceeded while awaiting a response.

*Message starts with data* : A new message (command word) is expected, but instead a data word has been received.

*Invalid mode command* : An invalid mode command has been encountered – this can occur one of two ways. One is if a broadcast transmit mode command is decoded – this is not a valid combination. The second is if a zero data words is specified by a receive mode command – this is considered an invalid combination.

*Expected mode data missing* : The mode command specified the inclusion of mode data in the message, but it did not occur on the bus.

*BC invalid message type* : The message type specified at the time of a start message host request is invalid; message request is ignored (BC only).

*BC bus busy* : A start message request was received by the host, but a previous message request is still processing (BC only).

If the message processing logic encounters an error, the driver will raise an error interrupt if the error interrupt type is enabled in the mask. This is done even if the message is being filtered by an (addressed) MT or RT due to no terminal address match.

Upon receipt of an error interrupt, the host, as part of its error handling logic, can directly clear the error status. Or, the eTPU driver will automatically clear the error status once a complete and error-free message is received or sent. In an MT/RT eTPU driver, if the error was detected due to gap/idle detection, it will begin immediately looking for the start of a new message, otherwise, it will await a gap before trying to begin receipt of a new message. For the BC driver, it is recommended that after an error the bus idle detection feature be used to ensure the bus is clear before starting the next message transmission.

## Interface

The interaction with the eTPU consists of the following pieces:

* eTPU module configuration
* Host service requests to the eTPU (similar to an interrupt request)
* Shared memory
* Interrupts of the host processor from the eTPU

The 1553 eTPU driver uses the TCR1 counter for all of its timing. It is recommended it be set to a high resolution – the eTPU module clock divided by 2 is suggested (this is the highest setting in normal, or eTPU1 mode). All digital filtering should be minimized in order to minimize any delay in processing the signals. Given that the host API software handles the details of eTPU initialization, only general information is provided in this section.

### Driver Configuration

Most of the driver eTPU configuration is hidden from the user by the provided host API, but some additional detail is provided here. All the eTPU channels associated with a terminal share the same channel memory (2 RX channels for MT, and 2 TX channels for RT/BC) and thus are able to share much configuration. First, there are a number of parameters that hold TCR1 bit timing constants. The initialized values depend upon the system TCR1 counter frequency. Although all of these timing parameters could be derived from a single value, to maximize the efficiency of the eTPU code, the following all most be pe-computed and configured at initialization time – 0.5 bit time (0.5microsecond) in TCR1 counts, 1.25 bit time, 1.5 bit time, 1.75 bit time, 2.25 bit time, 2.5 bit time, and 18.75 bit time.

Other basic configuration includes base RX channel number (with appropriate engine link bit set), base TX channel number (RT/BC only, with appropriate engine link bit set), and the RT/MT terminal address (31 for unaddressed MT), and the message interrupt mask. The interrupt mask is described in more detail in its own section below.

There are 4 final timing configuration parameters that must be initialized. The host API opens these to user configuration since there is some leeway into their values. As with all other timing parameters, they are held in the eTPU in terms of TCR1 counts:

*Initial idle timeout* : Once the eTPU driver is initialized the RX portion requires the bus to be idle for at least this amount of time before beginning to process and accept messages (MT/RT).

*RT status timeout* : In a message, when an RT response (status, or status and data) is expected, if no response is seen in this amount of time, the message is considered to be in error / invalid.

*Transmit delay* : Applies to RT only. When the RT is expected to respond in a message, this is the amount of time between the end of the previous word, and the point at which the eTPU RT driver will begin transmitting the response (starting with a status word), minus the following: the bus idle detect time (2 us – minimum time between end of parity and start of a new word sync), plus the processing time of the eTPU thread that detects the idle and requests the transmit (~ 0.5 us), plus transmit setup time (0.5 us). Thus for example, configuring a transmit delay of 4 us results in a total response gap of ~ 7 us.

*BC seek idle timeout* : There may be situations wherein a BC needs to look for a bus idle situation, e.g., after the detection of a message error. For the BC to declare the bus idle, it must detect no activity for this amount of time. Note that this parameter is set when the seek idle API is used rather than at initialization time.

Once the driver data parameters are configured, the remaining eTPU channel setup can be completed, the channel enabled, and an initialization host service request placed.

### Inputs/Outputs

During operation, a number of outputs, and inputs for BC messages or RT responses, are used for communicating message status and data. For MT and RT, note that as this only applies to messages addressed to the MT (if in addressed mode) or RT – other messages do not update the data outputs or trigger interrupts; however, if an error is detected in a message not destined for the terminal, it is reported.

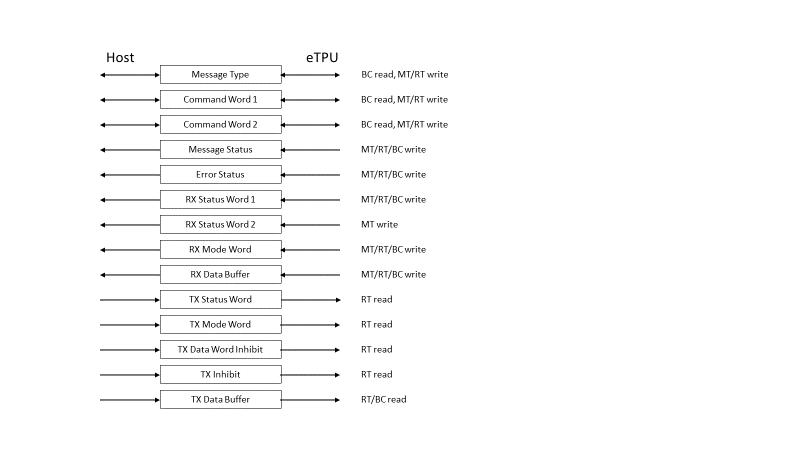


Figure : Host/eTPU Data Interface Overview

The data I/O parameters are described below:

#### eTPU Data Input/Output Parameters

*Message type* : For MT/RT terminal types, this is the message type for the last received message, or the one in the process of being received (input). For a BC terminal, it is the commanded message type (input). It is one of the 10 possible 1553 message types, and the values are defined in header file etec\_MS1553\_common.h.

*Command word 1* : For MT and RT terminal types, this holds the first received command word of a message. For a BC, it is the leading command word of the message being sent.

*Command word 2* : For MT and RT terminal types, this holds the transmit command of a received RT-RT message. For a BC, it is the second command word (RT -> RT messages only, transmit command).

#### eTPU Data Output Parameters

*Message status* : The status value represents stage of the message that is currently being processed. This output is set in conjunction with message interrupts. The possible status values ae defined etec\_MS1553\_common.h, and include command valid, data valid, message complete, etc.

*Error status* : When a message error is detected by the software, the error type is output in this data item, and depending upon interrupt configuration, an error interrupt is generated. The error codes are defined in etec\_MS1553\_common.h.

*RX status word 1* : For an MT or BC, this holds the first status word received in a message. For an RT, this data output is only set when it is the transmitting RT in an RT-RT message – in that case, this data holds the receiving RT’s status response.

*RX status word 2* : MT and BC only. Holds the second status word in an RT-RT message. This is the status response provided by the receiving RT.

*RX mode word* : Holds the received mode word from a mode command message.

*RX data buffer* : For an MT, this holds all the data (up to 32 words) that was transferred as part of the message. If the RT was a receiver in the message, this buffer holds that message’s data.

#### eTPU Data Inputs

*TX status word* : This is the status word a responding RT outputs as part of the message. The status word can be updated by the host any time up to mid-sync of the status word transmission.

*TX mode word* : For transmit mode commands that include a mode word, this is the mode word transmitted by the RT. The mode word may be updated by the host any time up to mid-sync of the mode word transmission. Note: for a BC, a mode word being sent to an RT(s) is placed in the TX data buffer, instead of using this input.

*TX data word inhibit* : When this is enabled, data and mode word transmission is inhibited for the RT. This supports the command illegalization requirement.

*TX inhibit* : Inhibit all transmission from the RT when enabled.

*TX data buffer* : For a transmitting RT or BC, this data buffer holds data words to be transmitted, up to 32 words. Each word needs to be updated by mid-sync of the transmission on the data bus.

### Interrupts

Each eTPU channel can generate two unique interrupts – a data/DMA interrupt and a channel interrupt. The MT/RT/BC eTPU driver does not make use of the DMA interrupt, just channel interrupts. Interrupts can be generated by either RX channel. On a BC, bus busy and invalid message type errors come via TX base channel interrupt. The activation of interrupts is controlled by the message interrupt mask configuration parameter. Seven types of interrupts are defined (see etec\_MS1553\_common.h), and each type can be enabled individually. A single interrupt activation may correspond to multiple types. They correspond to different message processing stages and status:

*Command word valid* : The command word (or command words for RT-RT) has been received and the message type is known. This interrupt occurs right after the command word completes on the bus, or in the case of receive messages, right after the second word of the message. Note that this means if the message is a BC-RT receive with only one data word, the command word and data valid interrupts are shared (and also message complete if it is a BC-RT broadcast). A BC does not issue command word valid interrupts.

*Mode word data valid* : The mode word has been seen on the bus (MT) or received (RT/BC). For a BC, the mode data valid piggybacks with message complete.

*Message data valid* : Message data receipt is complete. On a BC, this piggybacks with message complete except on non-broadcast RT -> RT messages.

*Message status valid* : A message status has been received. An RT can only generate this interrupt when it is the transmitting RT as part of an RT-RT message, and it has received status from the receiving RT. Note that for many message types, this interrupt is shared with message complete.

*Message complete* : The message is complete and valid. This interrupt is generated after the post-message gap has been detected.

*Message transmit inhibited* : Notify the host that an expected RT transmit did not occur because TX inhibit is enabled.

*Bus idle detected* : This interrupt is generated when a requested BC bus idle seek succeeds.

*Message error* : An error was detected during message processing. Note that an error in any message on the bus triggers this, even if the message is being filtered due to a non-matching terminal address.

In most cases, an MT will only want to enabled message complete and error interrupts. An RT is likely to require message complete, command word, mode word, data valid, as well as error interrupts to be enabled as a minimum. A BC would typically enable message complete, bus idle detected and error interrupts.

There will be no more than one interrupt per 1553 word (20 us), the one possible exception being the error interrupt. In some cases an error interrupt could be generated asynchronous to the receipt of a word. For example, if the expected last data word of a message has been received, then the data complete interrupt will have been raised, but then shortly thereafter, if the expected bus gap/idle is not detected, an error is raised – this would occur within a few microseconds, much sooner than a 20 us word time.

### Shutdown

A 1553 eTPU driver can be shut down via host service request. For an RT or BC, shutdown host service requests must be issued to both the RX base channel and the TX base channel. On the RX side, the eTPU will not respond to any input signal activity once deactivated, while on the TX side, the signal outputs will continuously stay low. In order to reactivate a driver, initialization host service request(s) must be made.

## Software Structure

The MIL-STD-1553 eTPU driver is encapsulated in one eTPU class that contains all the data, event handlers (threads) and event entry tables, and methods for an MT or RT instance. An MT uses two consecutive eTPU channels connected to input pins (RX\_P/RX\_N), while an RT or BC uses two sets of consecutive eTPU channels – one set connected to input pins (RX\_P/RX\_N), and one set connected to output pins (TX\_P/TX\_N). In both cases, all channels associated with a driver instance share the same channel frame memory (same CPBA value).

The all terminal driver types share the same low-level bit-handling code on the RX side – again, sharing common code is simple using the eTPU class architecture. The shared code is implemented as either threads or callable fragments. Callable functions are avoided in this implementation as they typically generate more overhead and performance is critical in this application.

On the MT/RT/BC RX side, the RX\_P and RX\_N channels have their own unique entry tables. Most of the work is performed by the base RX\_P channel – the RX\_N channel only has a role in detecting syncs and decoding command/status sync vs. data sync – all other processing is handled by the RX\_P channel. When a new word’s sync is identified, the RX\_P sets a match for just beyond the end of the word – it is this match event that performs the final word error checking (both bit level issues and protocol issues) and protocol management. In between, the low-level bit handling logic receives the word bit by bit.

The RX logic for MT/RT, and to some extent a BC, follows these basic sets of states:

* Seek for a sync (bus currently idle)
* Once sync identified, enter bit reception handling
* Word time complete, handle the word. Depending upon the state of the message protocol, either a gap/idle will be expected, or the RX function will immediately begin seeking the next sync
* Found gap/idle. If more pieces of the message are expected post idle, prepare for that and return to seeking for a sync. If the message is complete, return to seeking for a sync, awaiting the next message. On an RT, this may be a point where it needs to transmit its response. If that is the case, a transmit on the TX base channel is requested.

On a BC, the RX is subservient to the TX logic. When a message is started, the BC transmits the command word(s) and optionally data, depending upon message type. It then enables the RX side, which does the following (high level):

* Seek for post BC transmit idle. Depending on message type (e.g. BC -> RT broadcast), upon verification of idle the RX work may be done
* Complete message protocol (or detect error) – receive status and optionally data.
* Upon completion of message (or error), disable RX and mark the BC message processing as no longer busy. Issue message complete interrupt if enabled.

To support TX in an RT, only the base TX channel (TX\_P) actively executes eTPU code – it also directly controls TX\_N in order to drive the proper signal levels synchronized with TX\_P. The basic RT TX state flow is as follows:

* Idle, receive link from RX\_P (RT only) to begin transmission.
* Wait transmit delay and then start to generate a command sync.
* After half way through the sync, sample the word to be transmitted
* Transmit the word bit by bit
* At end of word, finish parity calculation and output parity bit
* If more words to transmit as part of message, start next sync (data), and then go to half sync step above.
* Otherwise, send a transmit complete link to the driver instance’s RX\_P channel and go idle

At that point, the RX\_P channel will complete checking for now expected bus gap/idle, and then continue with the message protocol. Note that when an RT instance is transmitting, the RX portion does not listen to this transmission.

The BC TX driver shares the same low-level bit and word handling code as the RT, and thus only executes code on the TX\_P (master) channel. The basic state flow for the BC TX is as follows:

* Activate via start message host service request. Message type and command word 1 must be specified at this point so that the protocol processing for the message can be initialized.
* Begin transmission a half bit after servicing of start message request. Sync and bit processing code is shared with RT TX (described above).
* Once all words have been transmitted, a link is sent to the master RX channel of the BC to complete message processing.

### Files and Build

The MIL-STD-1553 eTPU driver consists of the following files:

etec\_MS1553.h : Contains the eTPU class declaration for the 1553 eTPU driver, common to all driver modes.

etec\_MS1553\_common.h : Holds definitions that are needed to be accessed by both the eTPU driver code and host code interacting with the driver.

etec\_MS1553\_RX.c : The basic 1553 receive bit-handling code. Also has code to fully act as a basic word receiver.

etec\_MS1553\_TX.c : The basic 1553 transmit bit-handling code. Also has code to fully act as a basic word transmitter.

etec\_MS1553\_MT\_RX.c : The MT protocol handling code.

etec\_MS1553\_RT\_RX.c : The RT protocol handling code, receive portion.

etec\_MS1553\_RT\_TX.c : The RT protocol handling code, transmit portion.

etec\_MS1553\_BC\_RX.c : The BC protocol handling code, receive portion.

etec\_MS1553\_BC\_TX.c : The BC protocol handling code, transmit portion.

The eTPU code is built using the project in the top-level package directory. It can be opened and the build can be generated manually, or it can be done from command line by using the -AutoBuild option. See the Mk.bat file to see how that works.

## Performance

The ETEC linker outputs an analysis file that includes worst-case thread latency (WCTL) information (see etpu\_ab\_set\_ana.html for details). Based upon this information, it can be seen that an eTPU engine running at 128 MHz or higher can support a single MT or RT 1553 driver. A dual-redundant implementation requires two eTPU engines.

# Host API Software

The host API software for this 1553 eTPU-based driver is written in C, and is designed to be called from C or C++ code. All the data, configuration, and state of an MT, RT or BC driver instance is kept in a single data structure, described in detail in the next section. A pointer to an instance is a required parameter to every 1553 API. Additionally, all code to initialize and interact with the eTPU module is included in the package.

## 1553 Driver Instance

All of the configuration and state of a MIL-STD-1553 instance is held in a data structure – the higher level host code must declare or allocate one instance per 1553 bus (therefor 2 for a dual-redundant configuration). The structure is declared as follows:

typedef struct ms1553\_instance\_struct

{

ms1553\_terminal\_type\_t terminal\_type;

/\* eTPU configuration \*/

ETPU\_MODULE module;

uint8\_t rx\_chan\_base\_num;

uint8\_t tx\_chan\_base\_num;

uint8\_t priority;

uint32\_t cpba;

etpu\_if\_MS1553\_CHANNEL\_FRAME\* MS1553\_etpu\_access\_8bit;

etpu\_if\_MS1553\_CHANNEL\_FRAME\_PSE\* MS1553\_etpu\_access\_24bit;

/\* common terminal configuration \*/

uint32\_t terminal\_address;

uint8\_t interrupt\_mask;

/\* interrupt handlers \*/

void (\*p\_cmd\_word\_handler)(struct ms1553\_instance\_struct \*);

void (\*p\_mode\_data\_handler)(struct ms1553\_instance\_struct \*);

void (\*p\_message\_data\_handler)(struct ms1553\_instance\_struct \*);

void (\*p\_status\_word\_handler)(struct ms1553\_instance\_struct \*);

void (\*p\_message\_handler)(struct ms1553\_instance\_struct \*);

void (\*p\_transmit\_inhibited\_handler)(struct ms1553\_instance\_struct \*);

void (\*p\_idle\_detected\_handler)(struct ms1553\_instance\_struct \*);

void (\*p\_bus\_busy\_handler)(struct ms1553\_instance\_struct \*);

void (\*p\_error\_handler)(struct ms1553\_instance\_struct \*);

/\* message state/status retrieved from eTPU drivers \*/

ms1553\_instance\_state\_t state;

ms1553\_instance\_message\_status\_t status;

/\* configurable timing parameters \*/

float initialIdleTimeout;

float rtStatusTimeout;

/\* MT-specific configuration \*/

/\* none currently \*/

/\* RT-specific configuration \*/

float transmitDelay;

/\* BC-specific configuration \*/

/\* none currently \*/

} ms1553\_instance\_t;

The message state and status retrieved from the eTPU driver is contained in additional structures:

typedef struct

{

uint8\_t message\_state;

uint8\_t message\_type;

uint8\_t error\_status;

} ms1553\_instance\_state\_t;

typedef struct

{

uint16\_t cmdWord1;

uint16\_t cmdWord2;

uint16\_t statusWord1;

uint16\_t statusWord2;

uint16\_t modeWord;

} ms1553\_instance\_message\_status\_t;

A pointer to an instance structure is the first parameter to every host 1553 API. The contents of the structure will be discussed in more detail in relevant API documentation below.

## Interfaces

The host code in this package consists of generic eTPU initialization and utility code, and host code specific to the 1553 driver functions.

### eTPU Module

This software package includes a utility library to interact with the eTPU – it is primarily implemented in the source files etpu\_util\_ext.[c, h]. This code is an extension of the original library in that it allows access to both the eTPU-AB module and eTPU-C module. The original utility library, etpu\_util.[c, h] is also included.

eTPU module initialization is performed by a combination of code from etpu\_config.h (defines eTPU module register configuration), etpu\_init.[c, h] (pulls in the eTPU executable code and initialized data and calls the utility eTPU initialization function), and etpu\_util\_ext.[c, h] (performs eTPU module initialization). The user must properly update the following configuration items in etpu\_config.h:

* my\_etpu\_config initializer
* my\_etpu\_c\_config initializer (optional, for parts with an eTPU-C module)
* etpu\_[a,b,c]\_tcr[1,2]\_freq initializers (configured frequency of eTPU timers)

Once the eTPU module initialization is complete, 1553 eTPU driver instances can be initialized and the global timers (TCR1) can be activated. With this eTPU function, initialization can occur before or after the timers are started, but the initial bus idle detection cannot complete without the timers running.

### 1553 Driver

The provided package has the following API calls, organized by category.

#### Initialization and Shutdown

The initialization of a 1553 driver instance is accomplished by invoking the following API:

uint32\_t etpu\_ms1553\_init(

ms1553\_instance\_t \*p\_ms1553\_instance

);

The instance structure passed to the call must have all of its configuration items initialized prior to making the call – here are those items with description:

* terminal\_type : MT, RT or BC
* module : eTPU module – AB or C
* rx\_chan\_base\_num : base channel number of the RX pair – 0 to 31 for first engine, 64 to 95 for the second engine (eTPU-B)
* tx\_chan\_base\_num : base channel number of the TX pair – 0 to 31 for first engine, 64 to 95 for the second engine (eTPU-B). For MT, must be set to 0xFF (not used)
* priority : eTPU processing priority low, medium, high (recommended high)
* cpba : must be set to 0 initially, and not touched after that
* terminal\_address : the terminal address for RT or addressed MT. For unaddressed MT, set to 31
* interrupt\_mask : determines what kinds of events trigger interrupts from the eTPU (see etec\_MS1553\_common.h for bit definitions) – a specific bit set to 1 enables interrupt
* initial\_idle\_timeout : the bus idle time length that must be found after initialization, in microseconds
* rt\_status\_timeout : when an RT status response has not occurred within this time (microseconds), a timeout error is set
* transmit\_delay : the delay between when the RT protocol processing requests a transmit to begin, and when it actually starts

Each interrupt handler function pointer should be initialized to 0/NULL if not used, or to the proper function pointer if the host is planning to deal with a particular interrupt type.

In order to shutdown the driver, the following should be called:

uint32\_t etpu\_ms1553\_shutdown(

ms1553\_instance\_t \*p\_ms1553\_instance

);

After a shutdown, the driver can be re-enabled by initializing again, potentially with different parameters, however, the eTPU channels are expected not to change, and the ‘cpba’ data member must not be modified.

#### General

These APIs allow the user to get message state/status, data, a well as reset error status. They are for use by all terminal types. Note that if these calls are made after a message completes, they must be made before the first word of the next message can complete (approximately 20us).

uint32\_t etpu\_ms1553\_get\_state(

ms1553\_instance\_t \*p\_ms1553\_instance

);

uint32\_t etpu\_ms1553\_get\_message\_status(

ms1553\_instance\_t \*p\_ms1553\_instance

);

uint32\_t etpu\_ms1553\_get\_message\_data(

ms1553\_instance\_t \*p\_ms1553\_instance,

uint32\_t word\_count,

uint16\_t\* p\_data\_buf);

Note: When getting message data, rather than read the full 32 word buffer every time, it is recommended the user decode the number of words from the appropriate command word that is part of the message status, and just read the expected number of words.

uint32\_t etpu\_ms1553\_reset\_error\_status(

ms1553\_instance\_t \*p\_ms1553\_instance

);

Note: the user can allow the eTPU driver to auto-clear the error status when it next successfully receives a message (recommended).

#### RT-specific

The following APIs are specific to supporting an RT driver instance. The first three are used to set the data that makes up an RT response. In all cases, data must be written before mid-sync of the data going out onto the 1553 bus. The message type determines the exact response contents and length, although if a response is required, it always includes a status word.

uint32\_t etpu\_ms1553\_set\_tx\_status\_word(

ms1553\_instance\_t \*p\_ms1553\_instance,

uint16\_t status\_word

);

uint32\_t etpu\_ms1553\_set\_tx\_mode\_word(

ms1553\_instance\_t \*p\_ms1553\_instance,

uint16\_t mode\_word

);

uint32\_t etpu\_ms1553\_set\_tx\_data(

ms1553\_instance\_t \*p\_ms1553\_instance,

uint32\_t data\_word\_cnt,

uint16\_t\* p\_data

);

The last two APIs allow the user to disable (or re-enable) data word transmission, or all transmission. These capabilities are useful for command illegalization, or perhaps for switching between bus A and B in a dual redundant architecture.

uint32\_t etpu\_ms1553\_set\_tx\_data\_inhibit(

ms1553\_instance\_t \*p\_ms1553\_instance,

uint8\_t tx\_data\_inhibit

);

uint32\_t etpu\_ms1553\_set\_tx\_inhibit(

ms1553\_instance\_t \*p\_ms1553\_instance,

uint8\_t tx\_inhibit

);

Note: in both cases above, a ‘1’ indicates to inhibit transmission, ‘0’ to enable transmission.

#### BC-specific

The following APIs are specific to supporting a BC driver instance. The first API is for generating a message on the 1553 bus. In addition to the message type, it takes all necessary command(s) and data as parameters, although technically only the first command word is needed right at the start. However, providing everything up front simplifies the process. A message complete interrupt (if enabled) lets the host know that the message is completed, and any expected response data/status can be collected.

uint32\_t etpu\_ms1553\_start\_message(

ms1553\_instance\_t \*p\_ms1553\_instance,

uint8\_t message\_type,

uint16\_t cmd\_word\_1,

uint16\_t cmd\_word\_2,

uint16\_t\* p\_data,

uint8\_t word\_count

);

The other BC-specific API is for detecting an idle bus. This may be useful at initial startup, but is best used after an error is detected, as the terminal generating the failure may still be transmitting on the bus at the time the error is reported. A bus idle interrupt is generated when the specified idle is detected.

uint32\_t etpu\_ms1553\_seek\_idle(

ms1553\_instance\_t \*p\_ms1553\_instance,

float seek\_idle\_timeout\_us

);

#### Interrupt Support

The interrupt support consists of two pieces – one is an instance registration which allows the proper instance structure to be found from an eTPU channel interrupt source. This should not be needed by the user, as registration is performed as part of the initialization API.

uint32\_t etpu\_ms1553\_register\_instance(

ms1553\_instance\_t \*p\_ms1553\_instance

);

The second part if the interrupt handler. It is expected that this will be called from a user’s lower-level interrupt routine, which can be written in proper form to match any compiler-specific notation. The lower level routine is expected to perform the interrupt clear operation by writing the appropriate eTPU CISR register. This handler takes as a parameter the eTPU channel number (0 to 31, or 64 to 95 for eTPU-B).

uint32\_t etpu\_ms1553\_ISR(

uint32\_t etpu\_channel

);

The handler gets the message state which provide information on what event(s) triggered the interrupt. The possible message state types (defined in etec\_MS1553\_common.h) that trigger an interrupt are described in more detail in section Interrupts.

The message complete interrupt has priority over all others, and can occur in conjunction with several other types depending upon the message (command word valid, mode data valid, message data valid, status valid). If the message complete handler pointer is non-zero, it is called.

Otherwise, the message state is checked for any other types and the appropriate handlers, if non-zero, are called.

## Interrupt Timing

This section provides information on interrupt timing for all possible message types – it is documented as if every interrupt type is enabled (i.e. mask set to 0xFF).

### Monitor Terminal

Since the monitor terminal does not respond to messages, it may only make sense to listen to message complete and error interrupts. However, all interrupt types except transmit inhibited and bus idle are implemented and can be enabled if desired. Timing diagrams for each of the 10 possible message formats are found in the below sections.

#### BC -> RT

The timing for a BC to RT transfer is shown below. Both the multi-data word and single data word cases are shown as the interrupt timing and associated status differs between the two cases. Note that the diagramed interrupts only occur if enabled.

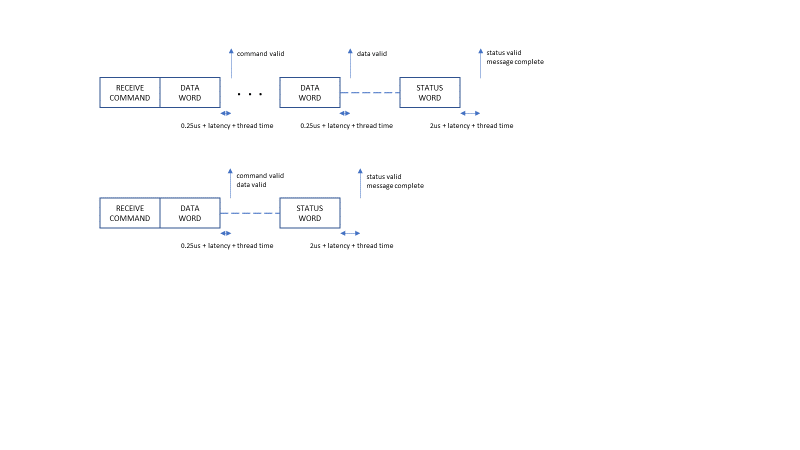


Figure : BC -> RT (Receive) Message Timing Diagram

#### RT-> BC

The timing for an RT to BC transfer is shown below. Note that the diagramed interrupts only occur if enabled.

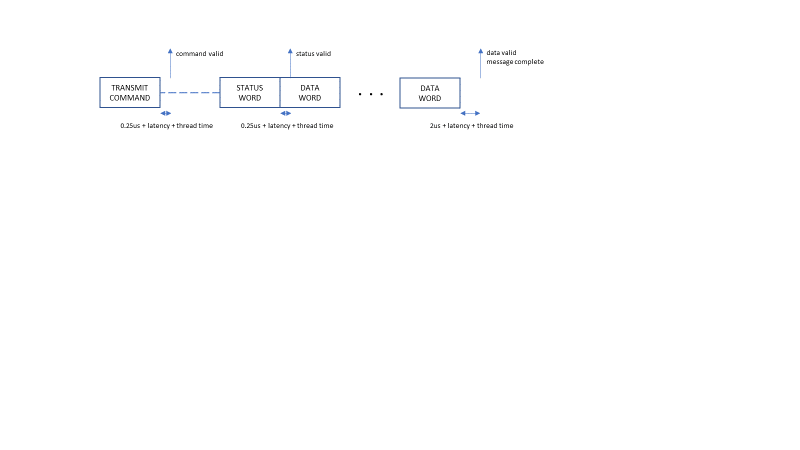


Figure : RT-> BC (Transmit) Message Timing Diagram

#### RT -> RT

The timing for an RT to RT transfer is shown below. Note that the diagramed interrupts only occur if enabled.

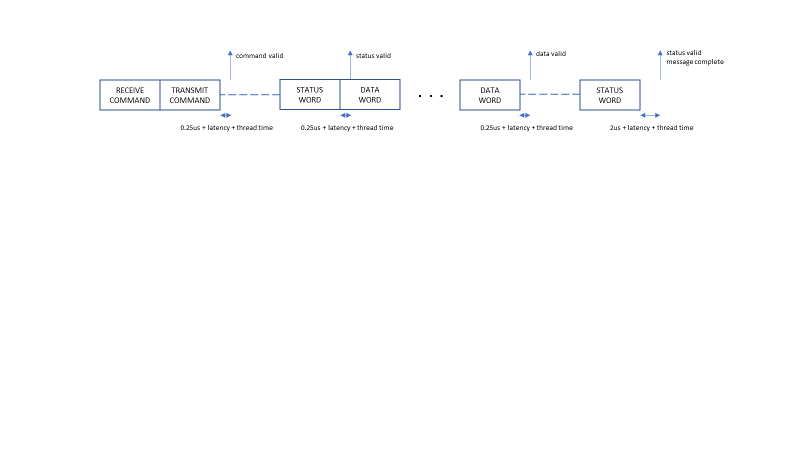


Figure : RT -> RT Message Timing Diagram

#### Mode Command Without Data

The timing for a mode command with no data is shown below. Note that the diagramed interrupts only occur if enabled.

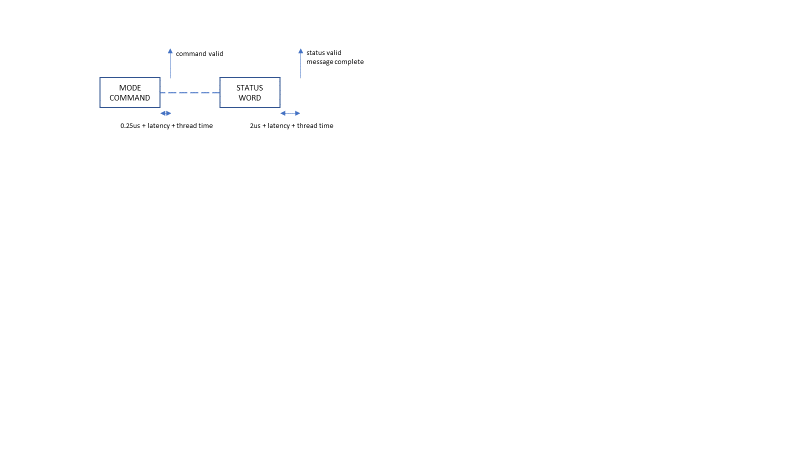


Figure : Mode Command (Without Data) Timing Diagram

#### Mode Command With Data Transmit

The timing for a mode command with a transmit data word is shown below. Note that the diagramed interrupts only occur if enabled.

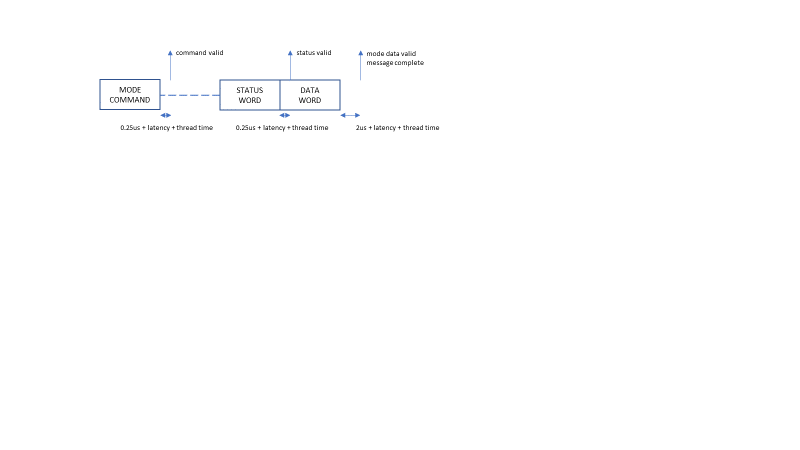


Figure : Mode Command (Data Transmit) Timing Diagram

#### Mode Command With Data Receive

The timing for a mode command with a receive data word is shown below. Note that the diagramed interrupts only occur if enabled.

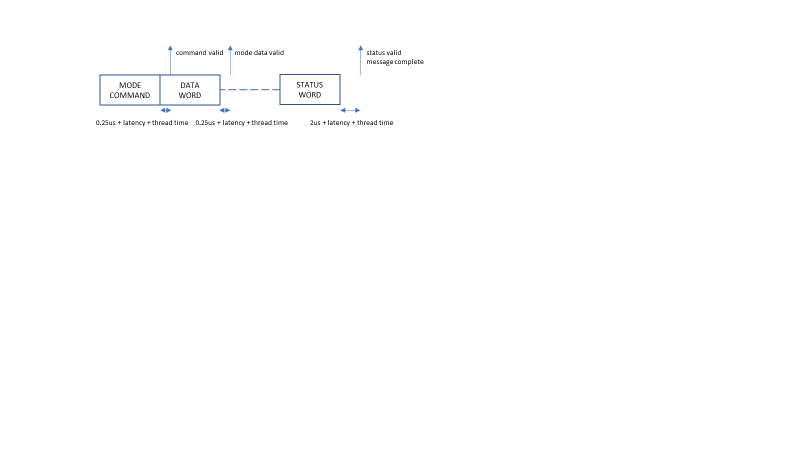


Figure : Mode Command (Data Receive) Timing Diagram

#### BC -> RT Broadcast

The timing for a BC to RT broadcast transfer is shown below. Both the multi-data word and single data word cases are shown as the interrupt timing and associated status differs between the two cases. Note that the diagramed interrupts only occur if enabled.

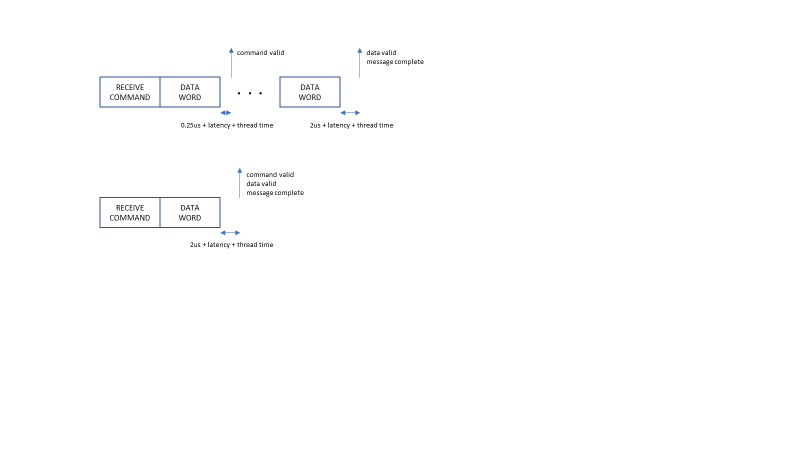


Figure : BC -> RT (Receive) Broadcast Message Timing Diagram

#### RT -> RT Broadcast

The timing for an RT to RT broadcast transfer is shown below. Note that the diagramed interrupts only occur if enabled.

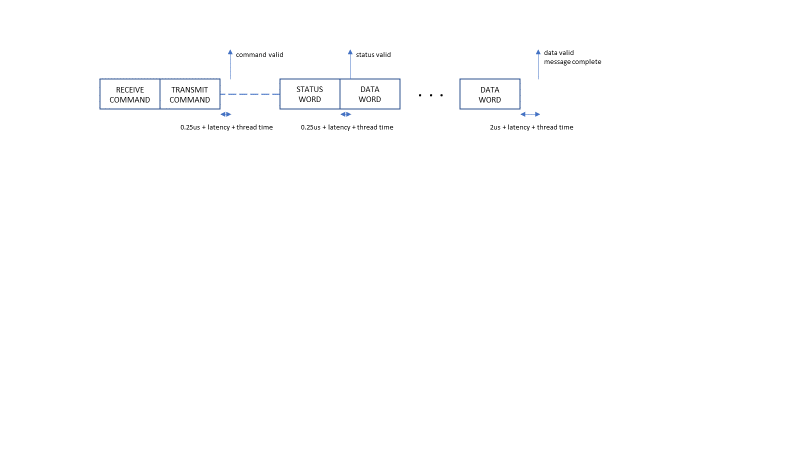


Figure : RT -> RT Broadcast Message Timing Diagram

#### Mode Command Without Data Broadcast

The timing for a broadcast mode command with no data is shown below. Note that the diagramed interrupt only occurs if enabled.

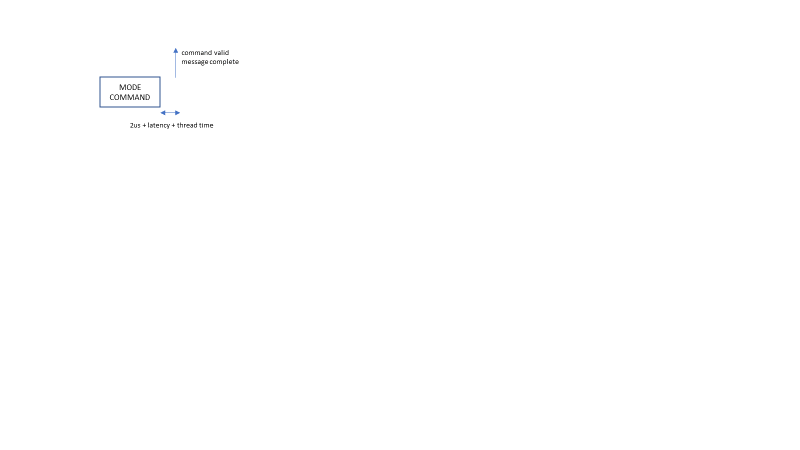


Figure : Mode Command (No Data) Broadcast Timing Diagram

#### Mode Command With Data Receive Broadcast

The timing for a broadcast mode command with data is shown below. Note that the diagramed interrupts only occur if enabled.

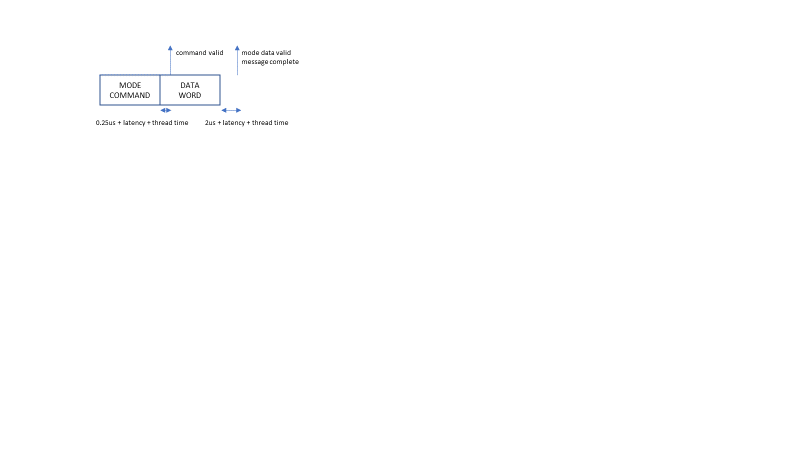


Figure : Mode Command (Data Receive) Broadcast Timing Diagram

### Remote Terminal

A remote terminal should listen to command word valid interrupts, because it will need to decode the message format and possibly respond within a short amount of time. It should also enable the data valid (and possibly mode data valid) interrupt in order to be able to provide a proper status-only response when required. Last, the error interrupt should be enabled. All interrupt types are implemented except for bus idle and can be enabled if desired. Timing diagrams for each of the 10 possible message formats are found in the below sections. 1553 words transmitted by the RT under consideration are marked in blue. Note that messages that do involved the RT (no terminal address match) are ignored, although may trigger message error interrupts if a protocol error is identified.

#### BC -> RT

The timing for a BC to RT transfer is shown below. Both the multi-data word and single data word cases are shown as the interrupt timing and associated status differs between the two cases. Note that the diagramed interrupts only occur if enabled.

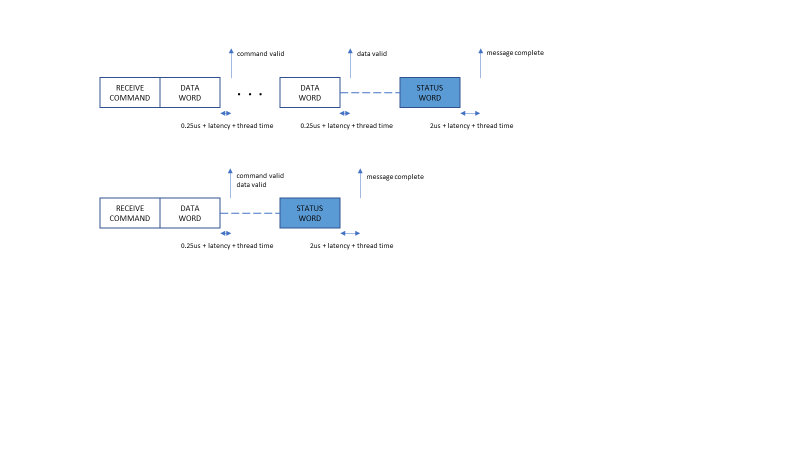


Figure : BC -> RT (Receive) Message Timing Diagram

#### RT-> BC

The timing for an RT to BC transfer is shown below. Note that the diagramed interrupts only occur if enabled.

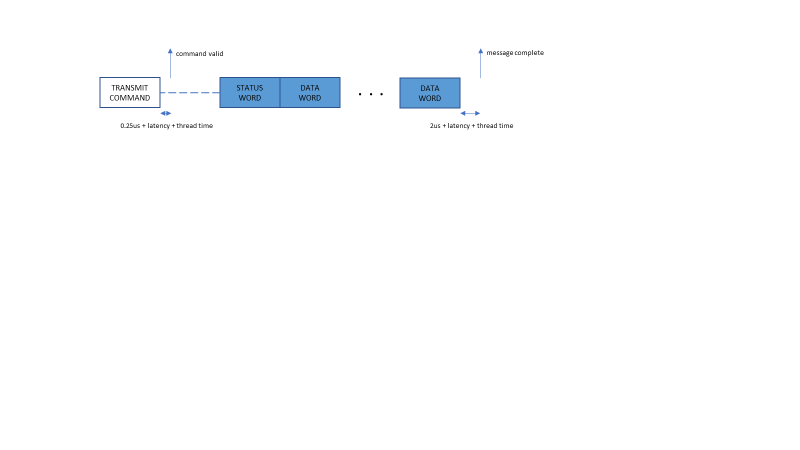


Figure : RT -> BC (Transmit) Message Timing Diagram

#### RT -> RT (transmitter)

The timing for an RT to RT transfer, from the perspective of the transmitting RT, is shown below. Note that the diagramed interrupts only occur if enabled.

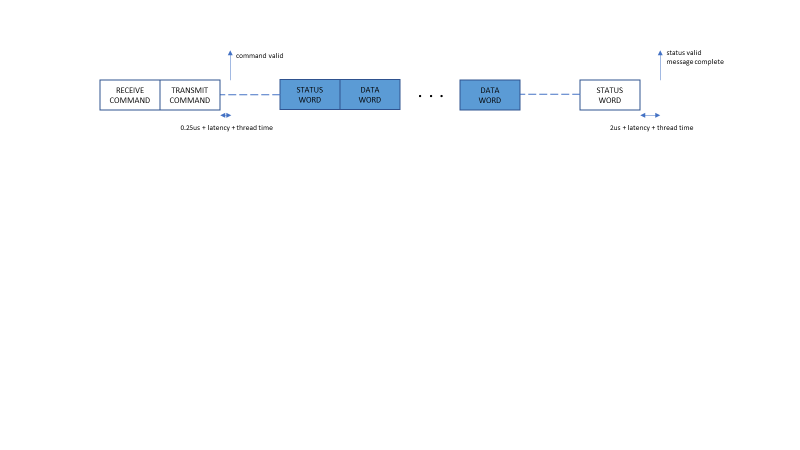


Figure : RT -> RT (Transmitter Perspective) Message Timing Diagram

#### RT -> RT (receiver)

The timing for an RT to RT transfer, from the perspective of the receiving RT, is shown below. Note that the diagramed interrupts only occur if enabled.

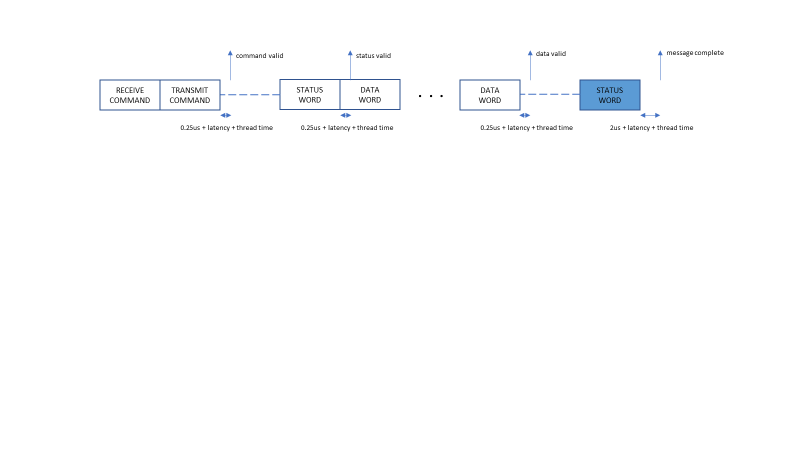


Figure : RT -> RT (Receiver Perspective) Message Timing Diagram

#### Mode Command Without Data

The timing for a mode command with no data is shown below. Note that the diagramed interrupts only occur if enabled.

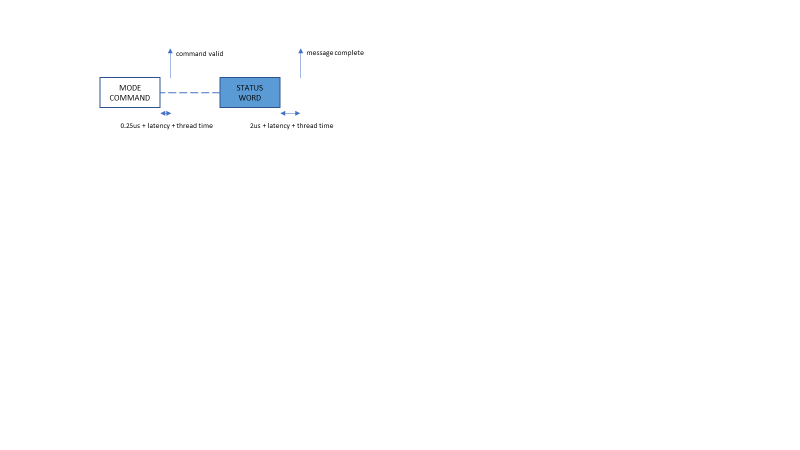


Figure : Mode Command (Without Data) Timing Diagram

#### Mode Command With Data Transmit

The timing for a mode command with a transmit data word is shown below. Note that the diagramed interrupts only occur if enabled.

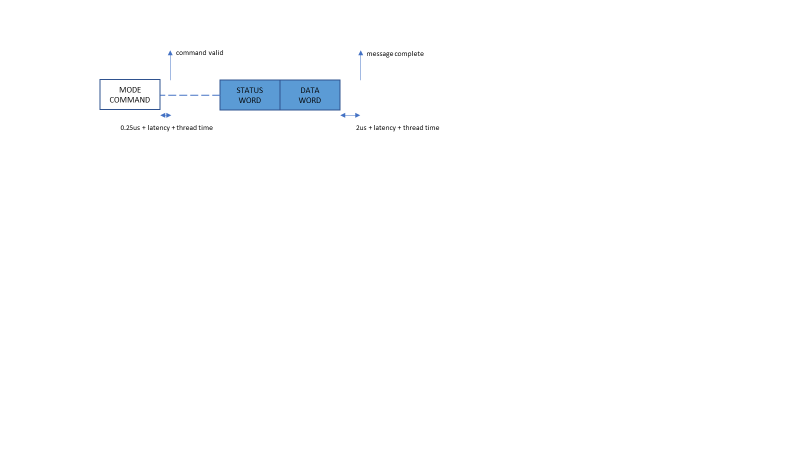


Figure : Mode Command (Data Transmit) Timing Diagram

#### Mode Command With Data Receive

The timing for a mode command with a receive data word is shown below. Note that the diagramed interrupts only occur if enabled.

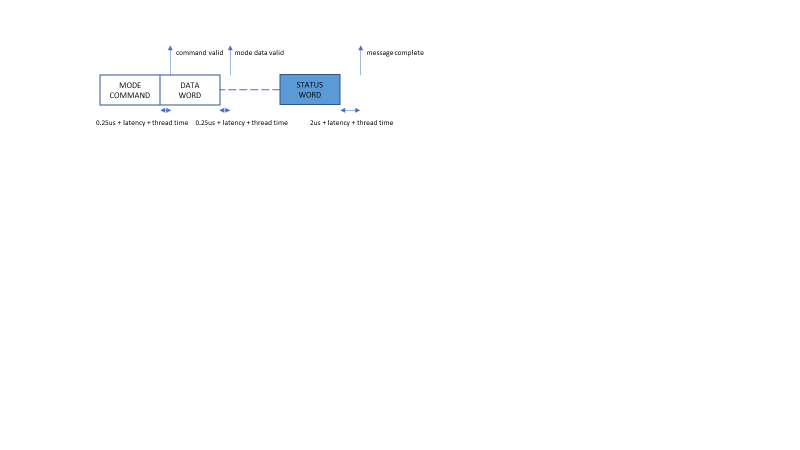


Figure : Mode Command (Data Receive) Timing Diagram

#### BC -> RT Broadcast

The timing for a BC to RT broadcast transfer is shown below. Both the multi-data word and single data word cases are shown as the interrupt timing and associated status differs between the two cases. Note that the diagramed interrupts only occur if enabled.

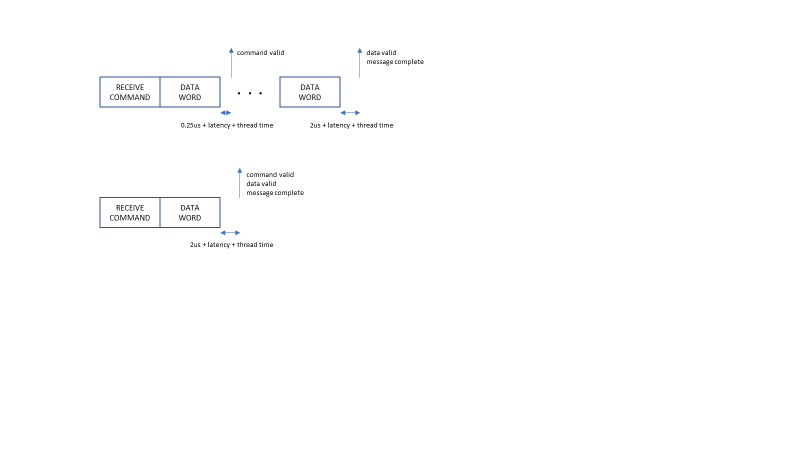


Figure : BC -> RT (Receive) Broadcast Message Timing Diagram

#### RT -> RT Broadcast (transmitter)

The timing for an RT to RT broadcast transfer, from the perspective of the transmitting RT, is shown below. Note that the diagramed interrupts only occur if enabled.

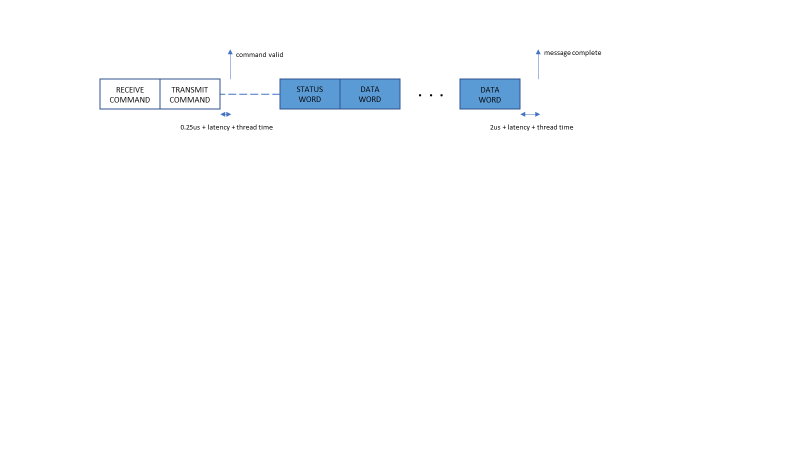


Figure : RT -> RT (Transmitter Perspective) Broadcast Message Timing Diagram

#### RT -> RT Broadcast (receiver)

The timing for an RT to RT broadcast transfer, from the perspective of the receiving RT, is shown below. Note that the diagramed interrupts only occur if enabled.

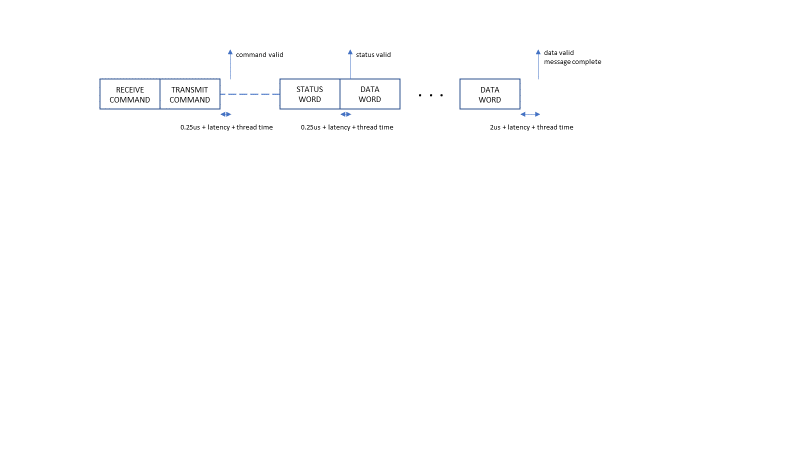


Figure : RT -> RT (Receiver Perspective) Broadcast Message Timing Diagram

#### Mode Command Without Data Broadcast

The timing for a broadcast mode command with no data is shown below. Note that the diagramed interrupt only occurs if enabled.

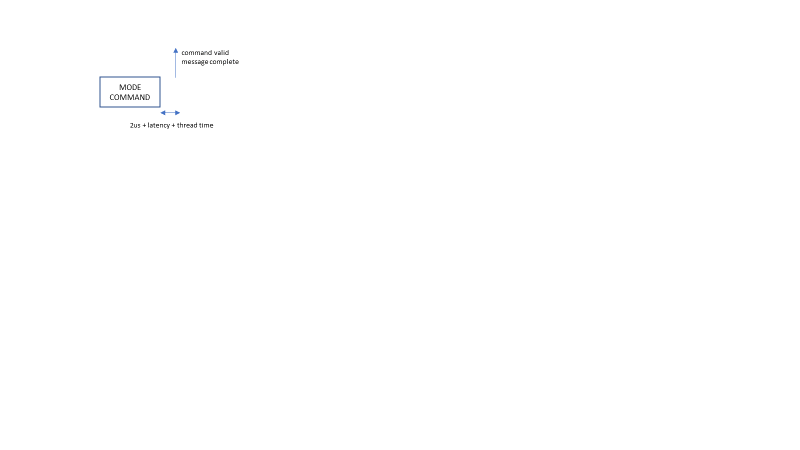


Figure : Mode Command (No Data) Broadcast Timing Diagram

#### Mode Command With Data Receive Broadcast

The timing for a broadcast mode command with data is shown below. Note that the diagramed interrupts only occur if enabled.

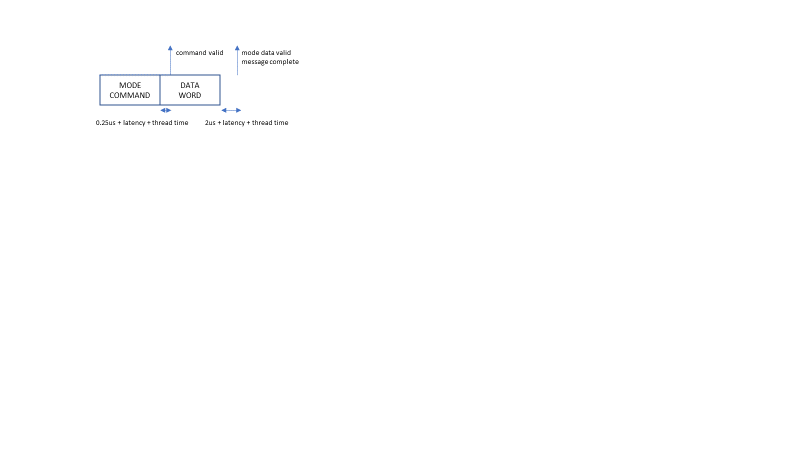


Figure : Mode Command (Data Receive) Broadcast Timing Diagram

### Bus Controller

A bus controller provides interrupt capability for items it may receive. Thus the command valid interrupt is not supported. Nor are interrupts generated for data transmitted by the BC, only received. Likely only the message complete, error and bus idle interrupts need be enabled. Timing diagrams for each of the 10 possible message formats are found in the below sections. 1553 words transmitted by the BC are marked in blue.

#### BC -> RT

The timing for a BC to RT transfer is shown below. Both the multi-data word and single data word cases are shown as the interrupt timing and associated status differs between the two cases. Note that the diagramed interrupts only occur if enabled.

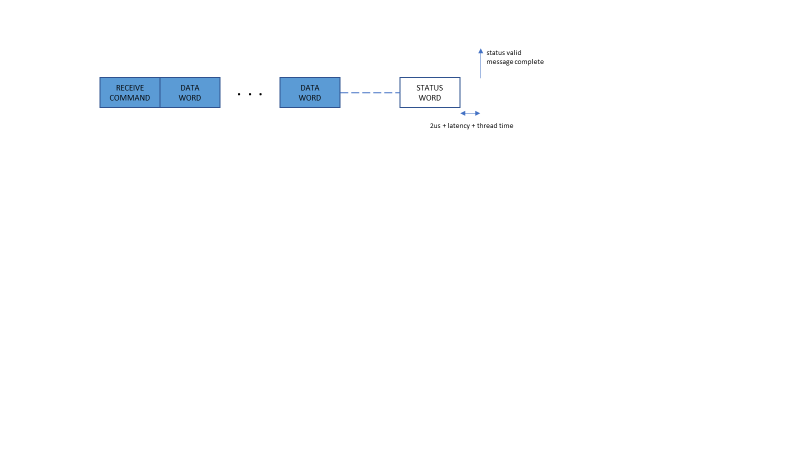


Figure : BC -> RT (Receive) Message Timing Diagram

#### RT -> BC

The timing for an RT to BC transfer is shown below. Note that the diagramed interrupts only occur if enabled.

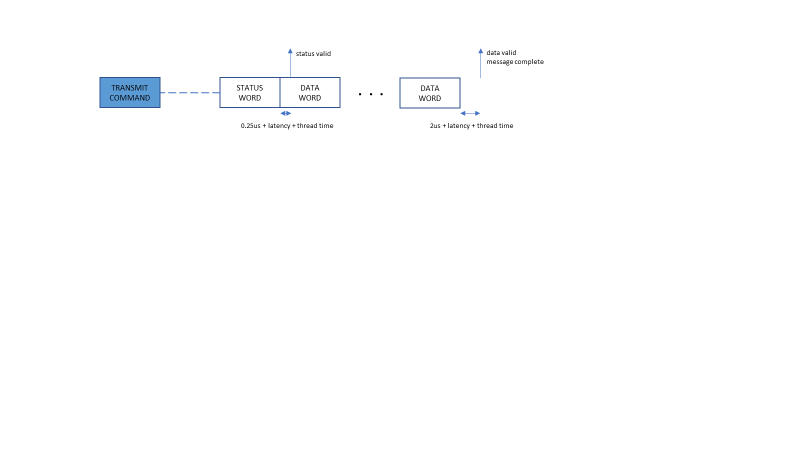


Figure : RT -> BC (Transmit) Message Timing Diagram

#### RT -> RT

The timing for an RT to RT transfer is shown below. Note that the diagramed interrupts only occur if enabled.

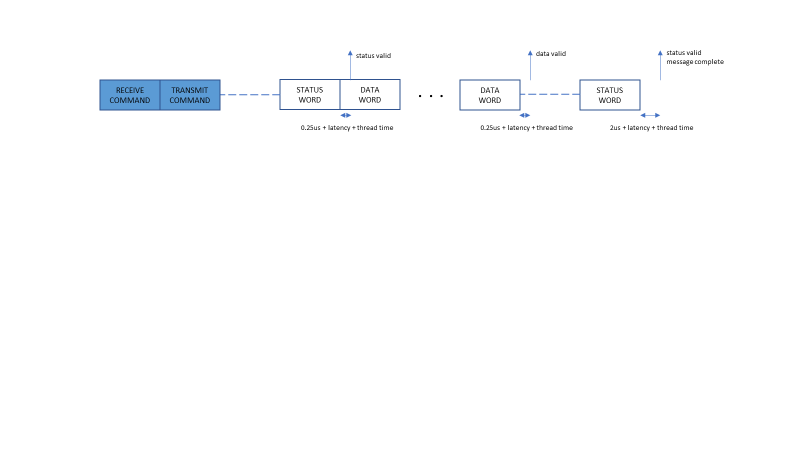


Figure : RT -> RT Message Timing Diagram

#### Mode Command Without Data

The timing for a mode command with no data is shown below. Note that the diagramed interrupts only occur if enabled.

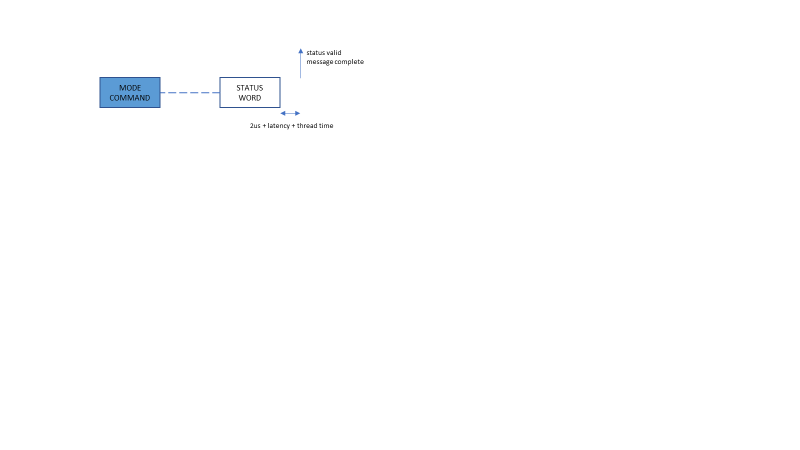


Figure : Mode Command (Without Data) Timing Diagram

#### Mode Command With Data Transmit

The timing for a mode command with a transmit data word is shown below. Note that the diagramed interrupts only occur if enabled.

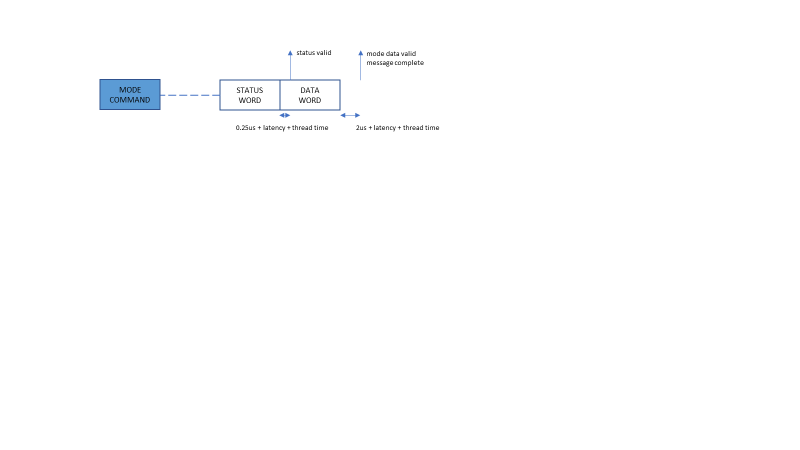


Figure : Mode Command (Data Transmit) Timing Diagram

#### Mode Command With Data Receive

The timing for a mode command with a receive data word is shown below. Note that the diagramed interrupts only occur if enabled.

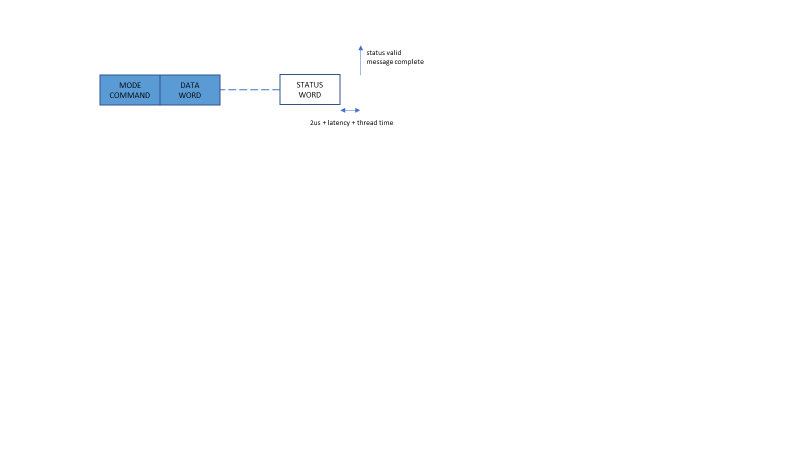


Figure : Mode Command (Data Receive) Timing Diagram

#### BC -> RT Broadcast

The timing for a BC to RT broadcast transfer is shown below. Both the multi-data word and single data word cases are shown as the interrupt timing and associated status differs between the two cases. Note that the diagramed interrupts only occur if enabled.

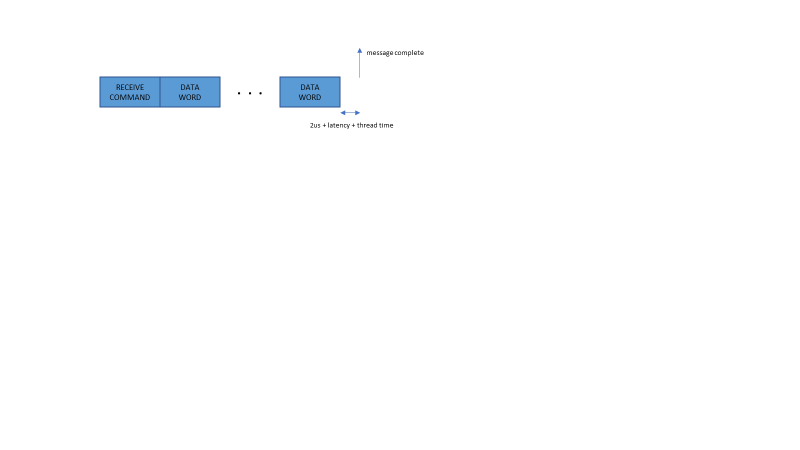


Figure : BC -> RT (Receive) Broadcast Message Timing Diagram

#### RT -> RT Broadcast

The timing for an RT to RT broadcast transfer is shown below. Note that the diagramed interrupts only occur if enabled.

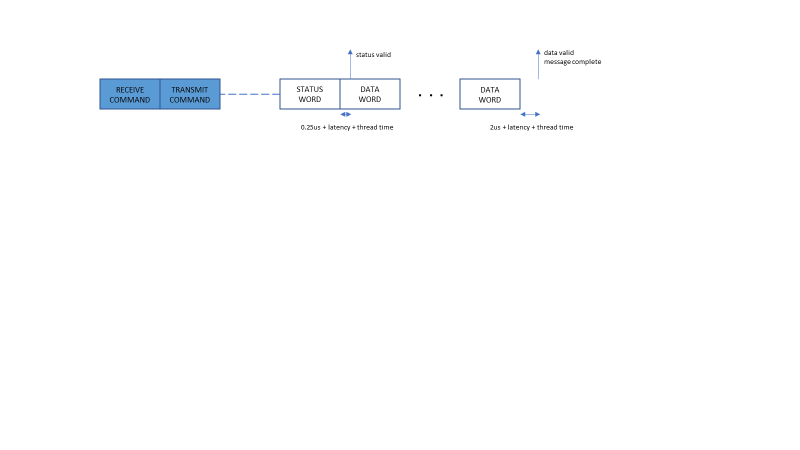


Figure : RT -> RT Broadcast Message Timing Diagram

#### Mode Command Without Data Broadcast

The timing for a broadcast mode command with no data is shown below. Note that the diagramed interrupt only occurs if enabled.



Figure : Mode Command (No Data) Broadcast Timing Diagram

#### Mode Command With Data Receive Broadcast

The timing for a broadcast mode command with data is shown below. Note that the diagramed interrupts only occur if enabled.



Figure : Mode Command (Data Receive) Broadcast Timing Diagram

# Testing Artifacts

The software testing makes use of the eTPU2+ Development Tool and System Development Tool from ASH WARE. There are two types of tests. The main type consists of standalone testing the eTPU software. The second type is the system testing – these tests exercise and validate the functioning of the host API software that is part of this package. All tests are packaged up in regression-style command line batch scripts – the whole thing can be executed by running ‘Test.bat’

## eTPU Tests

There are 4 sets of eTPU simulation tests. One tests the basic RX and TX signal decoding/encoding. The second tests the MT protocol processing, the third the RT protocol processing, and the fourth the BC terminal. For the MT and RT tests, the script command files act like a BC, generating messages on the simulated 1553 bus. In the RT testing, there is also a simulated MT instance listening to the 1553 bus, in order to verify that the RT instance under test responds properly when it is supposed to. Last, for the BC testing a responding RT and passive MT in the simulation in order to ensure the BC deals properly with issuing messages. All possible message types are tested, as well as all fault detection.

### Code Coverage

eTPU2+ Development Tools allows the collection of code coverage information. This capability is used to verify that cumulatively, the simulation tests execute 100% of the driver C code. Additionally, it is verified that 100% branch coverage is achieved. This is accomplished at the eTPU instruction (opcode) level – every branch instruction must be executed with both true and false conditions. The coverage script command tests generate coverage reports.

### Testing-Only Code

The eTPU software build includes a source code file called ‘Tester.c’. The eTPU function defined in that code is used in the eTPU standalone testing to verify proper functioning in corner cases, by allowing a test to create certain kinds of delays/latency situations.

## System Tests

There is a system test for an MT driver instance, an RT instance, and a BC. Note that the RT test also includes an MT instance on eTPU-B to track the RT responses. The BC test includes both an RT and MT on eTPU-B to assist in the testing process. Each is located in their own test sub-directory, and have the same overall structure. The eTPU script command file associated with the Mt/RT tests acts like a BC on the bus, generating messages to test the driver software. For the BC testing, scripting is only used for generating errors. The test code is all located in the source file main.c for each test case. The structure is as follows:

* Initialize eTPU module using the eTPU software build auto-generated output data.
* Initialize the MT/RT/BC instance under test.
* Enter a processing loop that polls for interrupts that invokes the main 1553 interrupt handling code, and then any user-specified sub-handlers for specific interrupt types. For the BC test, the processing loop triggers a new message test every 200us.
* The processing loop also checks for the end of a test, and when detected, verifies that the test completed as expected.