Joint Tactical Radio System Standard
MHAL on Chip Bus
Application Program Interface

Version: 1.1.5
26 June 2013

Revision History

Version	Description	Last Modified Date
1.0.5	Preparation for public release **ICWG Approved**	29-Jun-2010
1.1 <Draft>	-Added MOCB RF Coordinator (RFC) Extension -Update Introduction, references, and Abbreviations in MOCB API	22-Apr-2011
1.1 <Final Draft>	No further changes	01-Jun-2011
1.1.1<Draft>	Delete section E.3.2 Scanning Function Delete Figure 35:MOCB Data Bus Structure Delete Figure 36: MOCB Waveform Initiator Functional Block Diagram Delete Figure 37: Tx Enabled Rx Enabled Timing Diagram Delete Table 7 - MOCB RFC Framework Update Figure 34: MOCB Waveform Initiator Context Diagram Update Table 7 MOCB Initiator Module I/O Update text throughout Section E. MOCB RF Chain Coordinator Extension Update E.3.1:Transmit Power Control Function: MOCBRFC_TPC Update E.3.2:Rx Gain Function: *MOCBRFC_RXGAIN*	03-Aug-2011
1.1.1<Final Draft>	Update Section 3.1.1 Parameters Add additional clarification to Figure 10 Remove contents from section 3.2.1.1 Module I/O Delete text from section 3.1 Transmit Power Control Function Delete section E.3.2.3 Sample Interface	29-Aug-2011
1.1.1	**ICWG Approved**	31-Aug-2011
1.1.2<Draft>	Add “nEntries” parameter to section C.3.1.3, C.3.1.4,C.3.1.7, C.3.1.8, C.5.4.4, C.5.4.5	16-Nov-2011
1.1.2<Final Draft>	Misc. Redline changes	10-Jan-2012
1.1.2	Misc. Redline changes **ICWG Approved**	24-Jan-2012
1.1.3	Misc. Redline changes **ICWG Approved**	18-Apr-2012

Version	Description	Last Modified Date
1.1.4<Draft>	Change “read” operation to “mocbRead” in Section C.3.1.1 Change “write” operation to “mocbWrite” in Section C.3.1.5	10-Jul-2012
1.1.4<Final Draft>	No further changes	7-Aug-2012
1.1.4	No further changes **ICWG Approved**	14-Aug-2012
1.1.5	Preparation for public release **ICWG Approved**	26-Jun-2013

Table of Contents

Table of Contents

Lists of Figures

List of Tables

A. MOCB

A.1 INTRODUCTION

This API provides information to the software developer to utilize the *Modem Hardware Abstraction Layer on Chip Bus* (*MOCB*) interfaces in waveform target configurations.

The MOCB API provides parallel interfaces between the JTR channel modem interfaces from the application software. The MOCB API supports communications between application components hosted on General Purpose Processors (GPPs), Modem Digital Signal Processors (DSPs) and/or Modem Field Programmable Gate Arrays (FPGAs).

For the purposes of this API, the following applies to processor naming conventions:

- A GPP represents a CORBA capable processor (this could be a DSP that supports CORBA).
- A DSP represents a C capable processor, but does not provide CORBA capability.
- An FPGA represents a HDL capable processor, again without CORBA capability.

From one MHAL Computational Element (CE) (i.e. GPP, FPGA, or DSP) (see *MHAL API* [1]), it is possible to access other CE(s) using the interfaces defined in each MOCB API extension. The *MOCB FPGA* represents the parallel address/data bus interface and is the Waveform/User interface to a memory map. The *MOCB FPGA* consists of an FPGA entity library linked into a waveform build. The JTR set interfaces are unique, but the interfaces exposed to waveform components defined in section D *MOCB FPGA API Extension*. The *MOCB GPP* is the CORBA-based SCA *CF::Device* [2] interface and defined in section B MOCB GPP API Extension. The *MOCB DSP* is a library of standardized components linked into the waveform code at build time. The external interfaces and transport are JTR set defined, but the exposed interfaces to DSP waveform components defined in section C MOCB DSP API Extension. Additional capabilities for RF control; specifically Power Control (PC), Scanning, and Receiver Gain are defined in section E MOCB RF Chain Coordinator (RFC) API Extension.

The MOCB API does not specify the number of CEs a JTR platform provide. The MOCB API does not specify the platform specific transport, implementation or hardware architecture. For example, the MOCB API is byte oriented and abstracts away the details of the hardware architecture, like a physical interface's transfer dimensions (8/16/32/64 bits).

The MOCB API does specify the parallel interfaces of different CEs for communication between the waveform and hardware.

The MOCB GPP/DSP API (at this time) only supports the basic functionality of the MOCB FPGA API Extension. APIs for extended features such as "Locked Transfers" or "Priority Transfers will be added as required.

A.1.1 Overview

This document contains as follows:

a. Section A.1, *Introduction*, of this document contains the introductory material regarding the Overview, Service Layer description, and Referenced Documents of this document.

b. Section A.2, *Services*, provides summary of service uses.

c. Section A.3, *Interface Definitions*

d. Section A.5, *Data Types and Exceptions*

e. Appendix A.A – *Abbreviations and Acronyms*

f. Appendix A.B – *Performance Specification*

A.1.2 Service Layer Description

Not applicable

A.1.3 Referenced Documents

The following documents of the exact issue shown form a part of this specification to the extent specified herein.

A.1.3.1 Government Documents

The following documents are part of this specification as specified herein.

A.1.3.1.1 Specifications

A.1.3.1.1.1 Federal Specifications

None

A.1.3.1.1.2 Military Specifications

None

A.1.3.1.1.3 Other Government Agency Documents

[1] JTRS Standard, “JTRS Standard MHAL API,” JTNC, Version 2.13.2
[2] JTRS Standard, “Software Communications Architecture (SCA),” JPEO, Version 2.2.2.
[3] JTRS Standard, “JTRS Standard CORBA Types,” JPEO, Version 1.0.2
[4] JTRS Standard, “Software Communications Architecture (SCA),” JPEO, Version 4.0.

A.2 SERVICES

A.2.1 General Assumptions

- MOCB software operations provide feedback to the caller via error codes.
- MOCB software requires the Waveform provided source or destination buffer memory to be aligned matching the platform.
- MOCB software requires the Waveform provided semaphore string "name" (handle for DSP) to represent a previously registered named semaphore.
- MOCB software is byte addressable.
- MOCB SW addresses data on byte boundaries regardless of the data width.

A.2.2 Logical Destination (LD) Assumptions

- The LD is mapped to an address in the FPGA physical memory (e.g. LD_1, LD_2, LD_3)
- No size is associated with the LD to address mapping. Size is defined by the User/Waveform at the time of the call (read, write, etc.). The platform could choose to restrict/bound the size at that time.
- LDs can be mapped to any location in memory. Allowing LDs to overlap and possibly appear as subsets to another LD.

In **Figure 1**, for example, the assumptions would imply that when a "read" (**pink below**) is executed for LD_1, other LDs (LD_2, LD_3) could be mapped into the memory space being read. Likewise LD_2 could later be accessed (**orange below**) for only a subset of the previous "read" operation of LD_1.

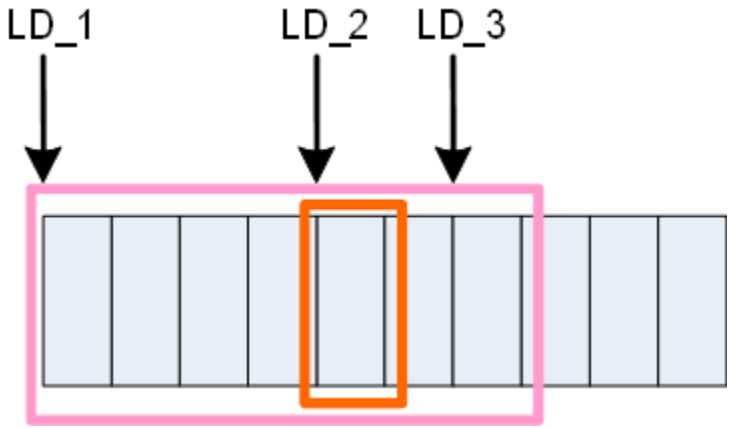

Figure 1 – LD Example

The MOCB addressing is performed with an LD and an offset within the specified LD as shown in **Figure 2**. The offset permits granular access to registers within a memory range assigned to a logical

destination. **Figure 2** shows that an offset of 0 (zero) would point to the beginning of the logical destination. Offsets n and m in the figure are address offsets within the logical destination X (LD_X). The shaded area is the waveform memory map within the terminal memory map accessible by this operation.

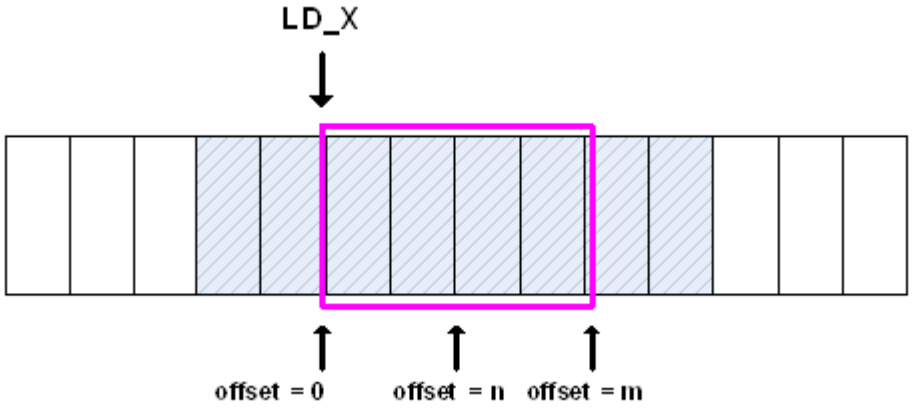

Figure 2 – LD Offset Example

A.3 SERVICE PRIMITIVES AND ATTRIBUTES

None

A.4 INTERFACE DEFINITIONS

None

A.5 DATA TYPES AND EXCEPTIONS

None

APPENDIX A.A – ABBREVIATIONS AND ACRONYMS

API	Application Program Interface
CE	Computational Element
CF	Core Framework
CLK	Clock
CORBA	Common Object Request Broker Architecture
DMA	Direct Memory Access
DSP	Digital Signal Processor
EN	Enable
EOM	End of Message
FIFO	First In First Out
FPGA	Field Programmable Gate Array
Fx	Function
GPIO	General Purpose Input/Output
GPP	General Purpose Processor
HDL	Hardware Description Language
HW	Hardware
I/O	Input/Output
ICD	Interface Control Document
ICWG	Interface Control Working Group
IDL	Interface Definition Language
IU	In-Use (bit)
JPEO	Joint Program Executive Office
JTNC	Joint Tactical Networking Center
JTR	Joint Tactical Radio
JTRS	Joint Tactical Radio System
LD	Logical Destination
LSB	Least Significant Byte
MHz	Megahertz
MOCB	MHAL on Chip Bus
MSB	Most Significant Byte
N/A	Not Applicable
PPS	Pulse Per Second
RAM	Random Access Memory
RF	Radio Frequency
RFC	Radio Frequency Chain
Rx	Receive
SCA	Software Communications Architecture
SW	Software
Tid	Transaction Identification Number
Tx	Transmit
UML	Unified Modeling Language
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit
WF	Waveform

APPENDIX A.B – PERFORMANCE SPECIFICATION

Not applicable

B. MOCB GPP API EXTENSION

B.1 INTRODUCTION

The *MOCB GPP* API Extension extends the *MOCB* base API (see section A) and supports methods and attributes that are specific to the General Purpose Processor (GPP) Modem Hardware (HW) device represented. This API Extension provides the ability to synchronously and asynchronously read/write/modify data to and from a service user/provider's shared memory. This API Extension also provides memory mapped interfaces that provide a read (pull) capability between components. This API Extension also includes event lines to signal to the application software that data is ready. For the purposes of this API the following applies to processor naming conventions

- A GPP represents a CORBA capable processor (this could be a DSP that supports CORBA).

This API Extension provides information to the software developer to utilize the *MOCB GPP* interfaces in the Waveform target configurations.

All accesses to shared memory via the *MOCB GPP* and the *MOCB DSP* are atomic to prevent data corruption (this is the responsibility of the platform). A single API operation of (read/write/modify) is defined as an access.

Note: A platform may have a 64-bit long data type (vs. 32bits) which should be considered during waveform porting

B.1.1 Overview

This document contains as follows:

a. Section B.1, *Introduction*, of this document contains the introductory material regarding the overview, and Service Layer description.

b. Section B.2, *Services*, provides summary of service interface uses, interface for each device component, port connections, and sequence diagrams.

c. Section B.2, Services specifies the operations provided by the *MOCB GPP*.

d. Section B.4, *IDL*

e. Section B.5, *UML*

f. Appendix B.A – *Abbreviations and Acronyms*

g. Appendix B.B – *Performance Specification*

B.1.2 Service Layer Description

B.1.2.1 MOCB Port Connections

Figure 3 shows the port connections for the *MOCB*.

Note: All port names are for reference only.

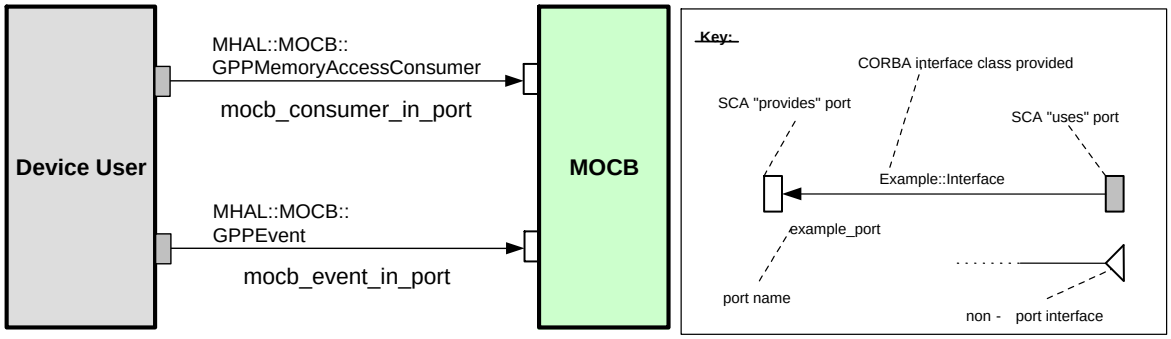

Figure 3 – MOCB Port Diagram

***MOCB* Provides Ports Definitions**

mocb_consumer_in_port is provided by the MOCB to synchronously and asynchronously read/write/modify data through operations available.

mocb_event_in_port is provided by the MOCB to manage events.

***MOCB* Uses Ports Definitions**

None

B.1.3 Modes of Service

Not applicable

B.1.4 Service States

B.1.4.1 MOCB State Diagram

The *MOCB* state model is illustrated in **Figure 4**. *MOCB* states ensure that received operations are executed only when the *MOCB* is in the proper state. The five states of the *MOCB* are as follow:

- CONSTRUCTED - The state transitioned to upon successful creation.
- INITIALIZED - The state transitioned to upon successful initialization.
- ENABLED - The state transitioned to upon successful start.
- DISABLED - The state transitioned to upon successful stop.
- RELEASED - The state transitioned to upon successful release.

The *MOCB* transitions between states in response to the initialize, start, stop and releaseObject operations.

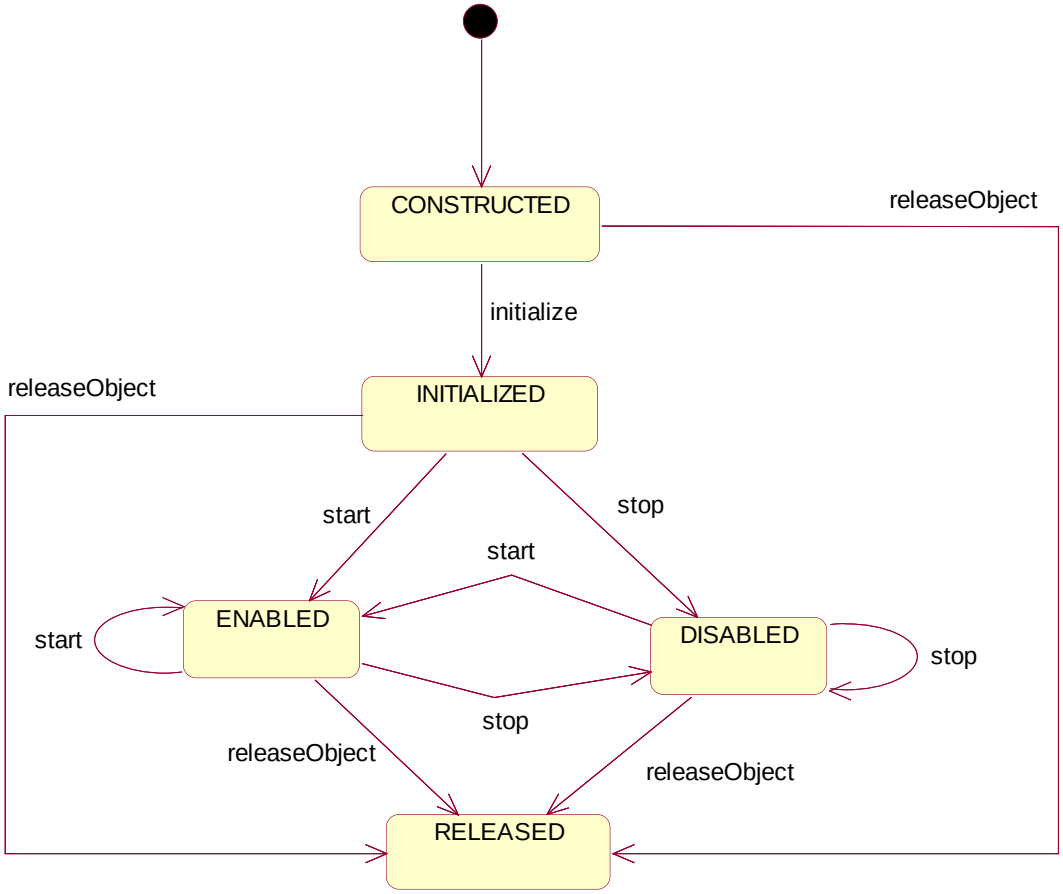

Figure 4 – MOCB State Diagram

B.1.5 Referenced Documents

The following documents of the exact issue shown form a part of this specification to the extent specified herein.

B.1.5.1 Government Documents

The following documents are part of this specification as specified herein.

B.1.5.1.1 Specifications

B.1.5.1.1.1 Federal Specifications

None

B.1.5.1.1.2 Military Specifications

None

B.1.5.1.1.3 Other Government Agency Documents

See section A.1.3.1.1.3.

B.2 SERVICES

The MOCB CORBA-compliant API separates platform interfaces from waveform interfaces.

B.2.1 Provide Services

The *MOCB* provide service consists of the following service ports, interfaces, and primitives, which can be called by other client components.

Table 1 – MOCB Provide Service Interface

Service Group (Port Name)	Service (Interface Provided)	Primitives (Provided)	Parameter Name or Return Value	Valid Range
mocb_consumer_in_port	MHAL::MOCB::GPPMemoryAccessConsumer	read	See section B.3.1.1	See section B.3.1.1
		readWait	See section B.3.1.2	See section B.3.1.2
		multiReadWait	See section B.3.1.3	See section B.3.1.3
		multiLDReadWait	See section B.3.1.4	See section B.3.1.4
		write	See section B.3.1.5	See section B.3.1.5
		writeWait	See section B.3.1.6	See section B.3.1.6
		multiWriteWait	See section B.3.1.7	See section B.3.1.7
		multiLDWriteWait	See section B.3.1.8	See section B.3.1.8
		modify	See section B.3.1.9	See section B.3.1.9
		modifyWait	See section B.3.1.10	See section B.3.1.10

Service Group (Port Name)	Service (Interface Provided)	Primitives (Provided)	Parameter Name or Return Value	Valid Range
		configLDMap	See section B.3.1.11	See section B.3.1.11
mocb_event_in_port	MHAL::MOCB::GPPEvent	registerSemaphore	See section B.3.2	See section B.3.2
		unregisterSemaphore	See section B.3.2.2	See section B.3.2.2
		registerEventMux	See section B.3.2.3	See section B.3.2.3

B.2.2 Use Services

None

B.2.3 Interface Modules

B.2.3.1 MHAL::MOCB

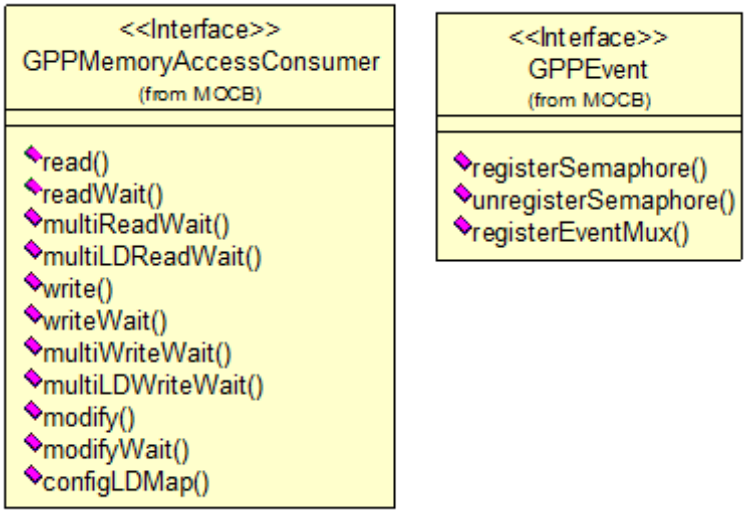

Figure 5 – MOCB Interface Class Diagram

B.2.3.1.1 GPPMemoryAccessConsumer Interface Description

The interface of the *GPPMemoryAccessConsumer* is depicted in **Figure 5**. It provides the ability to synchronously and asynchronously read/write/modify data to and from a service user/provider's shared memory.

B.2.3.1.2 GPPEvent Interface Description

The interface design of the *GPPEvent* illustrated in **Figure 5** provides the ability to manage events.

B.2.4 Sequence Diagrams

None

B.3 SERVICE PRIMITIVES AND ATTRIBUTES

To enhance the readability of this API document and to avoid duplication of data, the type definitions of all structured types (i.e., structures, typedefs, exceptions, and enumerations) used by the Service Primitives and Attributes have been co-located in section B.5 UML.

B.3.1 MHAL::MOCB::GPPMemoryAccessConsumer

B.3.1.1 *read* Operation

This operation provides the ability to read data from shared memory. This operation is non-blocking and returns the data via an MHAL message (**Table 2**) to the provided "callbackLD". This utilizes the *MHALPacketConsumer* interface defined in the MHAL GPP API Extension [1].

To read 32 bits, 4 octets will be used. For example, for a terminal with a 32-bit bus, the 4 octets will be used in one transaction, for a terminal with a 16-bit bus, 2 transactions will occur each using 2 octets.

Note: The callback message (10 bytes) consists of an MHAL header with the callbackLD, a payload of 6 bytes representing the ErrorCodes for this operation and the data read.

Table 2 – MOCB GPP Extension: Read Operation Callback

<MHAL Header >		<Payload >	
LD	Length		
callbackLD $_{16}$	**nByte + 10** $_{16}$	*ErrorCode* $_{32}$	*Read Data* $_{nByte}$

B.3.1.1.1 Synopsis

```
oneway void read (
    in unsigned short LD,
    in unsigned long offset,
    in unsigned short nByte,
    in unsigned short callbackLD
);
```

B.3.1.1.2 Parameters

Parameter Name	Description	Type	Units	Valid Range
LD	The logical destination for the message.	unsigned short	Logical Destination ID	0 – 32767
offset	Address offset from base address assigned to a logical destination (LD).	unsigned long	Offset from Logical Destination	Not Specified
nByte	The number of bytes to be read.	unsigned short	N/A	0 – 65525

Parameter Name	Description	Type	Units	Valid Range
callbackLD	The logical destination for the callback message.	unsigned short	Logical Destination ID	0 – 32767

B.3.1.1.3 State

ENABLED CF::Device::operationalState.

B.3.1.1.4 New State

This operation does not cause a state change.

B.3.1.1.5 Return Value

None

B.3.1.1.6 Originator

Service Provider

B.3.1.1.7 Exceptions

None

B.3.1.2 *readWait* Operation

This is a blocking operation and permits reading data from the application memory map. The user/waveform is responsible for destroying the sequence allocated during the readWait() call by the MOCB.

Note: Zero for both sec and nsec would indicate to wait indefinitely.

B.3.1.2.1 Synopsis

```
ErrorCodes readWait (
   in unsigned long sec,
   in unsigned long nsec,
   in unsigned short LD,
   in unsigned long offset,
   in unsigned short nByte,
   out JTRS::OctetSequence buf
);
```

B.3.1.2.2 Parameters

Parameter Name	Description	Type	Units	Valid Range
sec	Integer seconds of time to wait.	unsigned long	Seconds	0 to 2147483647
nsec	Nanoseconds of time to wait.	unsigned long	Nanoseconds	0 to 999999999
LD	Logical destination of the message.	unsigned short	Logical Destination ID	0 – 32767
offset	Address offset from base address assigned to a logical destination (LD).	unsigned long	Offset from Logical Destination	Not Specified
nByte	Number of bytes to read.	unsigned short	N/A	0 – 65531
buf	The returned data.	JTRS::OctetSequence (See JTRS CORBA Types [3])	N/A	N/A

B.3.1.2.3 State

ENABLED CF::Device::operationalState.

B.3.1.2.4 New State

This operation does not cause a state change.

B.3.1.2.5 Return Value

Description	Type	Units	Valid Range
The error code representing the status of the operation's completion	ErrorCodes	N/A	See section B.5.2.2

B.3.1.2.6 Originator

Service Provider

B.3.1.2.7 Exceptions

None

B.3.1.3 *multiReadWait* Operation

This operation provides the ability for the application to read data from offsets within the same logical destination

B.3.1.3.1 Synopsis

```
ErrorCodes multiReadWait(
   in unsigned long sec,
   in unsigned long nsec,
   in unsigned short LD,
   in MultiRead addrList,
   out JTRS::OctetSequence buf
);
```

B.3.1.3.2 Parameters

Parameter Name	Description	Type	Units	Valid Range
sec	Integer seconds of time to wait.	unsigned long	Seconds	0 to 2147483647
nsec	Nanoseconds of time to wait.	unsigned long	Nanoseconds	0 to 999999999
LD	Logical destination of the message.	unsigned short	Logical Destination ID	0 – 32767
addrList	Sequence of offsets, and number of bytes to read.	MultiRead (See section B.5.1.1)	N/A	N/A
buf	Returned data.	JTRS::OctetSequence (See JTRS CORBA Types [3])	N/A	N/A

B.3.1.3.3 State

ENABLED CF::Device::operationalState.

B.3.1.3.4 New State

This operation does not cause a state change.

B.3.1.3.5 Return Value

Description	Type	Units	Valid Range
The error code representing the status of the operation's completion	ErrorCodes	N/A	See section B.5.2.2

B.3.1.3.6 Originator

Service Provider.

B.3.1.3.7 Exceptions

None

B.3.1.4 *multiLDReadWait* Operation

This operation provides the ability for the application to read data from offsets from multiple logical destinations.

B.3.1.4.1 Synopsis

```
ErrorCodes multiLDReadWait (
   in unsigned long sec,
   in unsigned long nsec,
   in MultiLDRead addrList,
   out JTRS::OctetSequence buf
);
```

B.3.1.4.2 Parameters

Parameter Name	Description	Type	Units	Valid Range
sec	Integer seconds to wait.	unsigned long	Seconds	0 to 2147483647
nsec	Nanoseconds of time to wait.	unsigned long	Nanoseconds	0 to 999999999
addrList	A sequence of LDs, offsets, and number of bytes to read.	MultiLDRead (See section B.5.1.2)	N/A	N/A
buf	The data read.	JTRS::OctetSequence (See JTRS CORBA Types [3])	N/A	N/A

B.3.1.4.3 State

ENABLED CF::Device::operationalState.

B.3.1.4.4 New State

This operation does not cause a state change.

B.3.1.4.5 Return Value

Description	Type	Units	Valid Range
The error code representing the status of the operation's completion	ErrorCodes	N/A	See section B.5.2.2

B.3.1.4.6 Originator

Service Provider.

B.3.1.4.7 Exceptions

None

B.3.1.5 *write* Operation

This operation writes data to shared memory. The operation is non-blocking and returns a confirmation via an MHAL message (**Table 3**) to the provided "callbackLD". This utilizes the *MHALPacketConsumer* interface defined in the MHAL GPP API Extension [1].

Four octets are used to write 32 bits. As an example, a terminal with a 32-bit bus will transmit 4 octets one transaction, whereas a terminal with a 16-bit bus will execute 2 transactions with 2 octets each.

A "non-blocking" write() with a "callbackLD" of "NOCALLBACK" disables the confirmation callback for that instance. "NOCALLBACK" is defined as a symbolic LD reference just like "RFCHAIN" and is assigned a value by the platform.

Note: The callback message (8 bytes) consists of an MHAL header with the callbackLD and a payload of 4 bytes containing the ErrorCodes for this operation.

Table 3 – MOCB GPP Extension: Write Operation Callback

<MHAL Header >		<Payload >
LD	Length	
callbackLD 16	**8** 16	*ErrorCode* 32

B.3.1.5.1 Synopsis

```
oneway void write (
   in unsigned short LD,
   in unsigned long offset,
   in JTRS::OctetSequence buf,
   in unsigned short callbackLD
);
```

B.3.1.5.2 Parameters

Parameter Name	Description	Type	Units	Valid Range
LD	Logical destination for the message	unsigned short	Logical Destination ID	0 – 32767
offset	Address offset from base address assigned to a logical destination (LD).	unsigned long	Offset from Logical Destination	Not Specified

Parameter Name	Description	Type	Units	Valid Range
buf	The data to be written. Note: The length of the sequence is used as the size of the transaction.	JTRS::OctetSequence (See JTRS CORBA Types [3])	N/A	N/A
callbackLD	The logical destination for the callback message	unsigned short	Logical Destination ID	0 – 32767

B.3.1.5.3 State

ENABLED CF::Device::operationalState.

B.3.1.5.4 New State

This operation does not cause a state change.

B.3.1.5.5 Return Value

None

B.3.1.5.6 Originator

Service Provider

B.3.1.5.7 Exceptions

None

B.3.1.6 *writeWait* Operation

This operation is blocking and provides the ability to write data to shared memory.

Note: Zero for both sec and nsec indicates to wait indefinitely.

B.3.1.6.1 Synopsis

```
ErrorCodes writeWait (
   in unsigned long sec,
   in unsigned long nsec,
   in unsigned short LD,
   in unsigned long offset,
   in JTRS::OctetSequence buf
);
```

B.3.1.6.2 Parameters

Parameter Name	Description	Type	Units	Valid Range
sec	Integer seconds of time to wait.	unsigned long	Seconds	0 to 2147483647
nsec	Nanoseconds of time to wait.	unsigned long	Nanoseconds	0 to 999999999
LD	The logical destination for the message	unsigned short	Logical Destination ID	0 – 32767
offset	Address offset from base address assigned to a logical destination (LD).	unsigned long	Offset from Logical Destination	Not Specified
buf	The data to be written. Note: The length of the sequence is used as the size of the transaction.	JTRS::OctetSequence (See JTRS CORBA Types [3])	N/A	N/A

B.3.1.6.3 State

ENABLED CF::Device::operationalState.

B.3.1.6.4 New State

This operation does not cause a state change.

B.3.1.6.5 Return Value

Description	Type	Units	Valid Range
The error code representing the status of the operation's completion	ErrorCodes	N/A	See section B.5.2.2

B.3.1.6.6 Originator

Service Provider

B.3.1.6.7 Exceptions

None

B.3.1.7 *multiWriteWait* Operation

This operation provides the ability for the application to write data to offsets within same logical destination.

B.3.1.7.1 Synopsis

ErrorCodes multiWriteWait (
in unsigned long sec,
in unsigned long nsec,
in unsigned short LD,
in MultiWrite addrValPairs
);

B.3.1.7.2 Parameters

Parameter Name	Description	Type	Units	Valid Range
sec	Integer seconds of time to wait.	unsigned long	Seconds	0 to 2147483647
nsec	Nanoseconds of time to wait.	unsigned long	Nanoseconds	0 to 999999999
LD	The logical destination for the message.	unsigned short	Logical Destination ID	0 – 32767
addrValPairs	A sequence of offsets, and the buffers of data to be written.	MultiWrite (See section B.5.1.3)	N/A	N/A

B.3.1.7.3 State

ENABLED CF::Device::operationalState.

B.3.1.7.4 New State

This operation does not cause a state change.

B.3.1.7.5 Return Value

Description	Type	Units	Valid Range
The error code representing the status of the operation's completion	ErrorCodes	N/A	See section B.5.2.2

B.3.1.7.6 Originator

Service Provider.

B.3.1.7.7 Exceptions

None

B.3.1.8 *multiLDWriteWait* Operation

This operation provides the ability for the application to write data to offsets within multiple logical destinations.

B.3.1.8.1 Synopsis

ErrorCodes multiLDWriteWait (
in unsigned long sec,
in unsigned long nsec,
in MultiLDWrite addrValPairs
);

B.3.1.8.2 Parameters

Parameter Name	Description	Type	Units	Valid Range
sec	Integer seconds of time to wait.	unsigned long	Seconds	0 to 2147483647
nsec	Nanoseconds of time to wait.	unsigned long	Nanoseconds	0 to 999999999
addrValPairs	A sequence of LDs, offsets, and the buffers of data to be written.	MultiLDWrite (See section B.5.4.4)	N/A	N/A

B.3.1.8.3 State

ENABLED CF::Device::operationalState.

B.3.1.8.4 New State

This operation does not cause a state change.

B.3.1.8.5 Return Value

Description	Type	Units	Valid Range
The error code representing the status of the operation's completion	ErrorCodes	N/A	See section B.5.2.2

B.3.1.8.6 Originator

Service Provider

B.3.1.8.7 Exceptions

None

B.3.1.9 *modify* Operation

This operation modifies data in shared memory. The operation is non-blocking and returns a confirmation via an MHAL message to the provided "callbackLD". This utilizes the *MHALPacketConsumer* interface defined in the MHAL GPP API Extension [1].

Four octets are used to write 32 bits. As an example, a terminal with a 32-bit bus will transmit 4 octets one transaction, whereas a terminal with a 16-bit bus will execute 2 transactions with 2 octets each.

A modify() with a "callbackLD" of NOCALLBACK disables the confirmation callback for that instance. NOCALLBACK is defined as a symbolic LD reference just like RFCHAIN and is assigned a value by the platform.

Note: The callback message (8 bytes) consists of an MHAL header with the callbackLD and a payload of 4 bytes representing the ErrorCodes for this operation.

Table 4 – MOCB GPP Extension: Modify Operation Callback

<MHAL Header >		<Payload >
LD	Length	
callbackLD 16	**8** 16	*ErrorCode* 32

B.3.1.9.1 Synopsis

```
oneway void modify (
   in unsigned short LD,
   in unsigned long offset,
   in JTRS::OctetSequence buf,
   in BitOp bitOperation,
   in unsigned short callbackLD
);
```

B.3.1.9.2 Parameters

Parameter Name	Description	Type	Units	Valid Range
LD	The logical destination for the message	unsigned short	Logical Destination ID	0 – 32767
offset	Address offset from base address assigned to a logical destination (LD).	unsigned long	Offset from Logical Destination	Not Specified

Parameter Name	Description	Type	Units	Valid Range
buf	The data to be modified. Note: The length of the sequence is used as the size of the transaction.	JTRS::OctetSequence (See JTRS CORBA Types [3])	N/A	N/A
bitOperation	The bitwise operation to be performed.	BitOp	N/A	See section B.5.2.3
callbackLD	The logical destination for the callback message	unsigned short	Logical Destination ID	0 – 32767

B.3.1.9.3 State

ENABLED CF::Device::operationalState.

B.3.1.9.4 New State

This operation does not cause a state change.

B.3.1.9.5 Return Value

None

B.3.1.9.6 Originator

Service Provider

B.3.1.9.7 Exceptions

None

B.3.1.10 *modifyWait* Operation

This operation is blocking and provides the ability to modify data in shared memory.

Note: Zero for both sec and nsec indicate to wait indefinitely.

B.3.1.10.1 Synopsis

```
ErrorCodes modifyWait (
   in unsigned long sec,
   in unsigned long nsec,
   in unsigned short LD,
   in unsigned long offset,
   in JTRS::OctetSequence buf,
   in BitOp bitOperation
);
```

B.3.1.10.2 Parameters

Parameter Name	Description	Type	Units	Valid Range
sec	Integer seconds of time to wait.	unsigned long	Seconds	0 to 2147483647
nsec	Nanoseconds of time to wait.	unsigned long	Nanoseconds	0 to 999999999
LD	The logical destination for the message	unsigned short	Logical Destination ID	0 – 32767
offset	Address offset from base address assigned to a logical destination (LD).	unsigned long	Offset from Logical Destination	Not Specified
buf	The data to be operated on. Note: The length of the sequence is used as the size of the transaction.	JTRS::OctetSequence (See JTRS CORBA Types [3])	N/A	N/A
bitOperation	The bitwise operation to be performed.	BitOp	N/A	See section B.5.2.3

B.3.1.10.3 State

ENABLED CF::Device::operationalState.

B.3.1.10.4 New State

This operation does not cause a state change.

B.3.1.10.5 Return Value

Description	Type	Units	Valid Range
The error code representing the status of the operation's completion	ErrorCodes	N/A	See section B.5.2.2

B.3.1.10.6 Originator

Service Provider

B.3.1.10.7 Exceptions

None

B.3.1.11 *configLDMap* Operation

This operation maps logical destinations to a starting address in the waveform memory map. The terminal software has knowledge of which portion of its memory map are for the waveform.

The "addressWidth" is used to determine the space reserved for the address component of the LD & Address pair in the payload parameter.

B.3.1.11.1 Synopsis

```
ErrorCodes configLDMap (
   in Map configMap
);
```

B.3.1.11.2 Parameters

Parameter Name	Description	Type	Units	Valid Range
configMap	Mapping of LDs to addresses	Map	N/A	See section B.5.1.1

B.3.1.11.3 State

ENABLED CF::Device::operationalState.

B.3.1.11.4 New State

This operation does not cause a state change.

B.3.1.11.5 Return Value

Description	Type	Units	Valid Range
The error code representing the status of the operation's completion	ErrorCodes	N/A	See section B.5.2.2

B.3.1.11.6 Originator

Service Provider

B.3.1.11.7 Exceptions

None

B.3.2 MHAL::MOCB::GPPEvent

B.3.2.1 *registerSemaphore* Operation

This operation registers a semaphore managed (created and destroyed) by the waveform software. The waveform calls this operation for each relevant subEvent represented by a bit in the event mux register**.

The semaphore is posted by the MOCB software when the specified MOCB event occurs. This event is known by the waveform software and the waveform firmware (e.g., an FPGA event line). The waveform hardware event line(s) is(are) mapped to terminal specific hardware (e.g., a GPIO line(s)) during waveform porting. The terminal software has knowledge of which terminal lines are connected to waveform hardware lines, and upon assertion of that discrete, the semaphore is posted.

The MOCB software provides event triggering to waveform software on a per bit basis within the event mux register.

The waveform firmware is responsible to clear all the subEvent bit(s) in the mux register after detecting the MOCB software reading the mux register (if more synchronization controls are required, the waveform may implement an independent clear/feedback register that can be written by waveform software via the MOCB *write*() or *writeWait*() operations)

MOCB software will post all subEvents present when the mux register is read after the MOCB event line is signaled.

** Note: The MOCB firmware provides a minimum of (1…n) event line(s) for waveform use. The waveform firmware provides one event mux register(1..n bytes) for each MOCB event line.

B.3.2.1.1 Synopsis

```
ErrorCodes registerSemaphore (
   in unsigned short eventId,
   in unsigned short subEventId,
   in string name
);
```

B.3.2.1.2 Parameters

Parameter Name	Description	Type	Units	Valid Range
eventId	The event Identifier associated with the semaphore. This represents the event provided to waveform software from the MOCB FPGA interface.	unsigned short	N/A	0 – 32767

Parameter Name	Description	Type	Units	Valid Range
subEventId	Associated with the bit position in the waveform provided mux register.	unsigned short	N/A	0 – 32767
name	Name of the semaphore created by the application	string	N/A	N/A

B.3.2.1.3 State

ENABLED CF::Device::operationalState.

B.3.2.1.4 New State

This operation does not cause a state change.

B.3.2.1.5 Return Value

Description	Type	Units	Valid Range
The error code representing the status of the operation's completion	ErrorCodes	N/A	See section B.5.2.2

B.3.2.1.6 Originator

Service Provider

B.3.2.1.7 Exceptions

None

B.3.2.2 *unregisterSemaphore* Operation

Un-registers a previously registered semaphore upon a MOCB event. The only parameter needed to un-register a semaphore is the name.

B.3.2.2.1 Synopsis

ErrorCodes unregisterSemaphore (
in string name
);

B.3.2.2.2 Parameters

Parameter Name	Description	Type	Units	Valid Range
name	Name of the semaphore created by the application	string	N/A	N/A

B.3.2.2.3 State

ENABLED CF::Device::operationalState.

B.3.2.2.4 New State

This operation does not cause a state change.

B.3.2.2.5 Return Value

Description	Type	Units	Valid Range
The error code representing the status of the operation's completion	ErrorCodes	N/A	See section B.5.2.2

B.3.2.2.6 Originator

Service Provider

B.3.2.2.7 Exceptions

None

B.3.2.3 *registerEventMux* Operation

This operation assigns an event to a mux register of subEvents the waveform will use. The waveform calls this operation on each MOCB event line.

B.3.2.3.1 Synopsis

ErrorCodes registerEventMux(
in unsigned short eventId,
in unsigned short LD,
in unsigned long offset,
in unsigned short nByte
);

B.3.2.3.2 Parameters

Parameter Name	Description	Type	Units	Valid Range
eventId	The event Identifier that represents the event discrete provided to waveform software from the MOCB FPGA interface (i.e. MOCB firmware)	unsigned short	N/A	0 – 32767
LD	The logical destination for the message. This is the location in the waveform memory space where the platform reads "nbytes" of event mux register when "eventId" occurs	unsigned short	Logical Destination ID	0 – 32767
offset	Address offset from base address assigned to a logical destination (LD).	unsigned long	Offset from Logical Destination	Not Specified
nByte	The number of bytes to be read.	unsigned short	N/A	0 – 65525

B.3.2.3.3 State

ENABLED CF::Device::operationalState.

B.3.2.3.4 New State

This operation does not cause a state change.

B.3.2.3.5 Return Value

Description	Type	Units	Valid Range
The error code representing the status of the operation's completion	ErrorCodes	N/A	See section B.5.2.2

B.3.2.3.6 Originator

Service Provider

B.3.2.3.7 Exceptions

None

B.4 IDL

B.4.1 MOCB Device IDL

```
/*
** MocbDevice.idl
*/

#ifndef __MOCBDEVICE_DEFINED
#define __MOCBDEVICE_DEFINED

#ifndef __JTRSCORBATYPES_DEFINED
#include "JtrsCorbaTypes.idl"
#endif

module MHAL {

        module MOCB {

            struct MultiLDReadEntry {
                unsigned short LD;
                unsigned long offset;
                unsigned short nByte;
            };

            typedef sequence<MultiLDReadEntry> MultiLDRead;

            struct MultiReadEntry {
                unsigned long offset;
                unsigned short nByte;
            };

            typedef sequence<MultiReadEntry> MultiRead;

            struct MultiLDWriteEntry {
                unsigned short LD;
                unsigned long offset;
                JTRS::OctetSequence buf;
            };

            typedef sequence<MultiLDWriteEntry> MultiLDWrite;

            struct MultiWriteEntry {
                unsigned long offset;
                JTRS::OctetSequence buf;
            };

            typedef sequence<MultiWriteEntry> MultiWrite;

            typedef JTRS::ExtEnum AddressIndexType;

            const AddressIndexType CONSTANT = 0;
            const AddressIndexType INCREMENT= 1;
            const AddressIndexType DECREMENT = 2;

            typedef short ErrorCodes;
```

```
        const ErrorCodes SUCCESSFUL = 0;
        const ErrorCodes INV_MEM_ACCESS = -1;
        const ErrorCodes INV_LD_OR_ADDR = -2;
        const ErrorCodes SRC_MEM_NOT_ALIGNED = -3;
        const ErrorCodes DST_MEM_NOT_ALIGNED = -4;
        const ErrorCodes INV_EVENT_ID = -5;
        const ErrorCodes INV_SEMAPHORE_NAME = -6;
        const ErrorCodes INV_TIME = -7;
        const ErrorCodes INV_SUBEVENT = -8;
        const ErrorCodes TIMER_EXPIRED = -9;
        const ErrorCodes INV_SIZE = -10;

        typedef JTRS::ExtEnum BitOp;

        const BitOp AND = 0;
        const BitOp OR = 1;
        const BitOp XOR = 2;
        const BitOp NAND = 3;
        const BitOp NOR = 4;

        struct MapEntry {
            unsigned short LD;  // the logical destination
            unsigned long address;  // the address this LD maps to
            AddressIndexType indexType;  // the index type for this
    address
        };

        typedef sequence<MapEntry> Map;

        interface GPPMemoryAccessConsumer {

            oneway void read (
                in unsigned short LD,
                in unsigned long offset,
                in unsigned short nByte,
                in unsigned short callbackLD
            );

            ErrorCodes readWait (
                in unsigned long sec,
                in unsigned long nsec,
                in unsigned short LD,
                in unsigned long offset,
                in unsigned short nByte,
                out JTRS::OctetSequence buf
            );

            ErrorCodes multiReadWait (
                in unsigned long sec,
                in unsigned long nsec,
                in unsigned short LD,
                in MultiRead addrList,
                out JTRS::OctetSequence buf
             );

            ErrorCodes multiLDReadWait (
                in unsigned long sec,
                in unsigned long nsec,
                in MultiLDRead addrList,
                out JTRS::OctetSequence buf
             );
```

```
        oneway void write (
            in unsigned short LD,
            in unsigned long offset,
            in JTRS::OctetSequence buf,
            in unsigned short callbackLD
        );

        ErrorCodes writeWait (
            in unsigned long sec,
            in unsigned long nsec,
            in unsigned short LD,
            in unsigned long offset,
            in JTRS::OctetSequence buf
        );

        ErrorCodes multiWriteWait (
            in unsigned long sec,
            in unsigned long nsec,
            in unsigned short LD,
            in MultiWrite addrValPairs
        );

        ErrorCodes multiLDWriteWait (
            in unsigned long sec,
            in unsigned long nsec,
            in MultiLDWrite addrValPairs
        );

        oneway void modify (
            in unsigned short LD,
            in unsigned long offset,
            in JTRS::OctetSequence buf,
            in BitOp bitOperation,
            in unsigned short callbackLD
        );

        ErrorCodes modifyWait (
            in unsigned long sec,
            in unsigned long nsec,
            in unsigned short LD,
            in unsigned long offset,
            in JTRS::OctetSequence buf,
            in BitOp bitOperation
        );

        ErrorCodes configLDMap (
            in Map configMap
        );

    };

    interface GPPEvent {

        ErrorCodes registerSemaphore (
            in unsigned short eventId,
            in unsigned short subEventId,
            in string name
        );

        ErrorCodes unregisterSemaphore (
```

```
                    in string name
                );

                ErrorCodes registerEventMux(
                    in unsigned short eventId,
                    in unsigned short LD,
                    in unsigned long offset,
                    in unsigned short nByte
                );

            };
        };
};

#endif
```

B.5 UML

This section contains the Device component UML diagram and the definitions of all data types referenced (directly or indirectly) by section Service Primitives and Attributes.

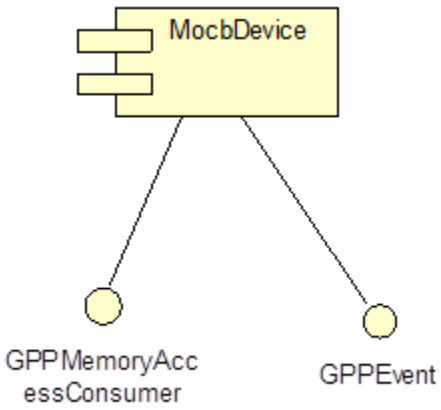

Figure 6 – MOCB Component Diagram

B.5.1 Data Types

B.5.1.1 MultiRead

This sequence type definition provides pairings of offsets, and number of bytes to read (see section B.5.4.1).

typedef sequence<MultiReadEntry> MultiRead;

B.5.1.2 MultiLDRead

This sequence type definition provides pairings of LDs, offsets, and number of bytes to read (see section B.5.4.2).

typedef sequence<MultiLDReadEntry> MultiLDRead;

B.5.1.3 MultiWrite

This sequence type definition provides pairings of offsets, and buffers of data to be written (see section B.5.4.3).

typedef sequence<MultiWriteEntry> MultiWrite;

B.5.1.4 MultiLDWrite

This sequence type definition provides pairings of LDs, offsets, and buffers of data to be written (see section B.5.4.4).

typedef sequence<MultiLDWriteEntry> MultiLDWrite;

B.5.1.5 Map

This sequence type definition provides a list of the LD/address mappings (see section B.5.4.5).

typedef sequence<MapEntry> Map;

B.5.2 Enumerations

B.5.2.1 AddressIndexType

This enumeration definition enumerates the address indexing that can be performed.

typedef JTRS::ExtEnum AddressIndexType;

const AddressIndexType CONSTANT = 0;
const AddressIndexType INCREMENT= 1;
const AddressIndexType DECREMENT = 2;

JTRS::ExtEnum	Element	Value	Description
AddressIndexType	CONSTANT	0	Constant address indexing
	INCREMENT	1	Increment address indexing
	DECREMENT	2	Decrement address indexing

B.5.2.2 ErrorCodes

This type definition is a JTRS extension enumeration (see JTRS CORBA Types [3]). It enumerates the error codes supported by the MOCB.

typedef short ErrorCodes;

const ErrorCodes SUCCESSFUL = 0;
const ErrorCodes INV_MEM_ACCESS = -1;
const ErrorCodes INV_LD_OR_ADDR = -2;
const ErrorCodes SRC_MEM_NOT_ALIGNED = -3;
const ErrorCodes DST_MEM_NOT_ALIGNED = -4;
const ErrorCodes INV_EVENT_ID = -5;
const ErrorCodes INV_SEMAPHORE_NAME = -6;
const ErrorCodes INV_TIME = -7;
const ErrorCodes INV_SUBEVENT = -8;
const ErrorCodes TIMER_EXPIRED = -9;
const ErrorCodes INV_SIZE = -10;

short	Element	Value	Description
ErrorCodes	SUCCESSFUL	0	Successful. The transfer was successfully queue/executed
	INV_MEM_ACCESS	-1	Invalid Memory Access. The memory location addressed was outside the platform's valid range.
	INV_LD_OR_ADDR	-2	Invalid LD / Address. The LD does not have a valid address map entry.

short	Element	Value	Description
	SRC_MEM_NOT_ALIGNED	-3	Source Memory Not Aligned. The (source) location where the data is originally stored, is not located on the processors defined address boundary for an efficient memory transfer. i.e. An error would occur if addressing the second byte in a 32-bit word, as the start of a word transfer.
	DST_MEM_NOT_ALIGNED	-4	Destination Memory Not Aligned. The location where the data will be stored is not located on that processors defined addressable boundary for an efficient memory transfer. i.e. An error would occur if addressing the second byte in a 32-bit word, as the start of a sequence of word transfer.
	INV_EVENT_ID	-5	Invalid EventID. The event ID does not correspond to a valid MOCB event line.
	INV_SEMAPHORE_NAME	-6	Invalid Semaphore Name. The event string has not been mapped to a subEventId.
	INV_TIME	-7	Invalid Time (i.e. sec or nsec). The time specified is outside the valid range.
	INV_SUBEVENT	-8	Invalid SubEventID. The subEventId is outside the bounds of the mux register size.
	TIMER_EXPIRED	-9	Timer expired. The time specified expired prior to completion of the operation.
	INV_SIZE	-10	Invalid size. The number of bytes requested exceeds the limit that can be returned via the MHAL callback.

B.5.2.3 BitOp

This type definition is a JTRS extension enumeration (see JTRS CORBA Types [3]). It enumerates the bit-wise operations supported by the MOCB.

typedef JTRS::ExtEnum BitOp;

const BitOp AND = 0;
const BitOp OR = 1;
const BitOp XOR = 2;
const BitOp NAND = 3;
const BitOp NOR = 4;

JTRS::ExtEnum	Element	Value	Description
BitOp	AND	0	Logical AND
	OR	1	Logical OR
	XOR	2	Logical XOR
	NAND	3	Logical NAND
	NOR	4	Logical NOR

B.5.3 Exceptions

None

B.5.4 Structures

B.5.4.1 MultiReadEntry

This structure defines the pairing between an offset and number of bytes to read.

struct MultiReadEntry {
unsigned long offset;
unsigned short nByte;
};

Struct	Attributes	Type	Valid Range	Description
MultiReadEntry	offset	unsigned long	Not Specified	Address offset from base address assigned to a logical destination (LD).
	nByte	unsigned short	0 – 65531	The number of bytes to read.

B.5.4.2 MultiLDReadEntry

This structure defines the pairing between an offset and number of bytes to read.

struct MultiLDReadEntry {
unsigned short LD;
unsigned long offset;

unsigned short nByte;
};

Struct	Attributes	Type	Valid Range	Description
MultiLDReadEntry	LD	unsigned short	0 – 32767	The logical destination for the message.
	offset	unsigned long	Not Specified	Address offset from base address assigned to a logical destination (LD).
	nByte	unsigned short	0 – 65531	The number of bytes to read.

B.5.4.3 MultiWriteEntry

This structure defines the pairing between an offset and buffer of data to be written.

struct MultiWriteEntry {
unsigned long offset;
JTRS::OctetSequence buf;
};

Struct	Attributes	Type	Valid Range	Description
MultiWriteEntry	offset	unsigned long	Not Specified	Address offset from base address assigned to a logical destination (LD).
	buf	JTRS::OctetSequence (See JTRS CORBA Types [3])	N/A	The data to be written.

B.5.4.4 MultiLDWriteEntry

This structure defines the pairing between an LD, offset and buffer of data to be written.

struct MultiLDWriteEntry {
unsigned short LD;
unsigned long offset;
JTRS::OctetSequence buf;
};

Struct	Attributes	Type	Valid Range	Description
MultiLDWriteEntry	LD	unsigned short	0 – 32767	The logical destination for the message.

Struct	Attributes	Type	Valid Range	Description
	offset	unsigned long	Not Specified	Address offset from base address assigned to a logical destination (LD).
	buf	JTRS::OctetSequence (See JTRS CORBA Types [3])	N/A	The data to be written.

B.5.4.5 MapEntry

This structure defines the mapping between a logical destination and address in memory.

```
struct MapEntry {
   unsigned short LD;  // the logical destination
   unsigned long address;  // the address this LD maps to
   AddressIndexType indexType;  // the index type for this address
};
```

Struct	Attributes	Type	Valid Range	Description
MapEntry	LD	unsigned short	0 – 32767	The logical destination for the message.
	address	unsigned long	Not Specified	The address the LD maps to.
	indexType	AddressIndexType	See section B.5.2.1	The index type for this address.

APPENDIX B.A – ABBREVIATIONS AND ACRONYMS

See section Appendix A.A.

APPENDIX B.B – PERFORMANCE SPECIFICATION

Table 5 provides a template for the generic performance specification for the *MOCB GPP* API Extension documented in the waveform or user using the interface. This performance specification corresponds to the port diagram in **Figure 3**.

Table 5 – MOCB Performance Specification

Specification	Description	Units	Value
Worst Case Command Execution Time for mocb_consumer_in_port	*	*	*

Note: (*) These values should be filled in by individual developers.

C. MOCB DSP API EXTENSION

C.1 INTRODUCTION

The *MOCB DSP* API Extension extends the *MHAL DSP* API Extension [1] and consists of a collection of C function specifications that provides the ability to synchronously (i.e. blocking) and asynchronously (i.e. non-blocking) read/write/modify data to and from a service user/provider's shared memory and control platform defined events. For the purposes of this API, the following applies to processor naming conventions:

- A DSP represents a C capable processor that does not provide CORBA capability.

The service user includes the C function specifications in the service users DSP code. Calls to the *MOCB DSP* functions are made from the service users DSP source code.

All accesses to shared memory via the *MOCB GPP* and the *MOCB DSP* are atomic to prevent data corruption (this would be assumed as a platform responsibility). An access is defined as a single API operation (read/write/modify).

Note: A platform may have a 64-bit long data type (vs. 32bits) which should be considered during waveform porting

C.1.1 Overview

This document contains as follows:

a. Section C.1, *Introduction*, of this document contains the introductory material regarding the Overview.

b. Section C.2, *Services*, provides summary of service uses.

c. Section C.3, *Service Primitives and Attributes*, specifies the functions that are provided by the *MOCB DSP*.

d. Section C.4, *Interface Definitions*

e. Section C.5, *Data Types and Exceptions*, specifies the data types that are provided by the *MOCB DSP*.

f. Appendix C.A – *Abbreviations and Acronyms*

g. Appendix C.B – *Performance Specification*

C.1.2 Service Layer Description

Not applicable

C.1.3 Referenced Documents

The following documents of the exact issue shown form a part of this specification to the extent specified herein.

C.1.3.1.1 Government Documents

The following documents are part of this specification as specified herein.

C.1.3.1.1.1 Specifications

C.1.3.1.1.1.1 Federal Specifications

None

C.1.3.1.1.1.2 Military Specifications

None

C.1.3.1.1.1.3 Other Government Agency Documents

See section A.1.3.1.1.3.

C.2 SERVICES

C.2.1 Interface Modules

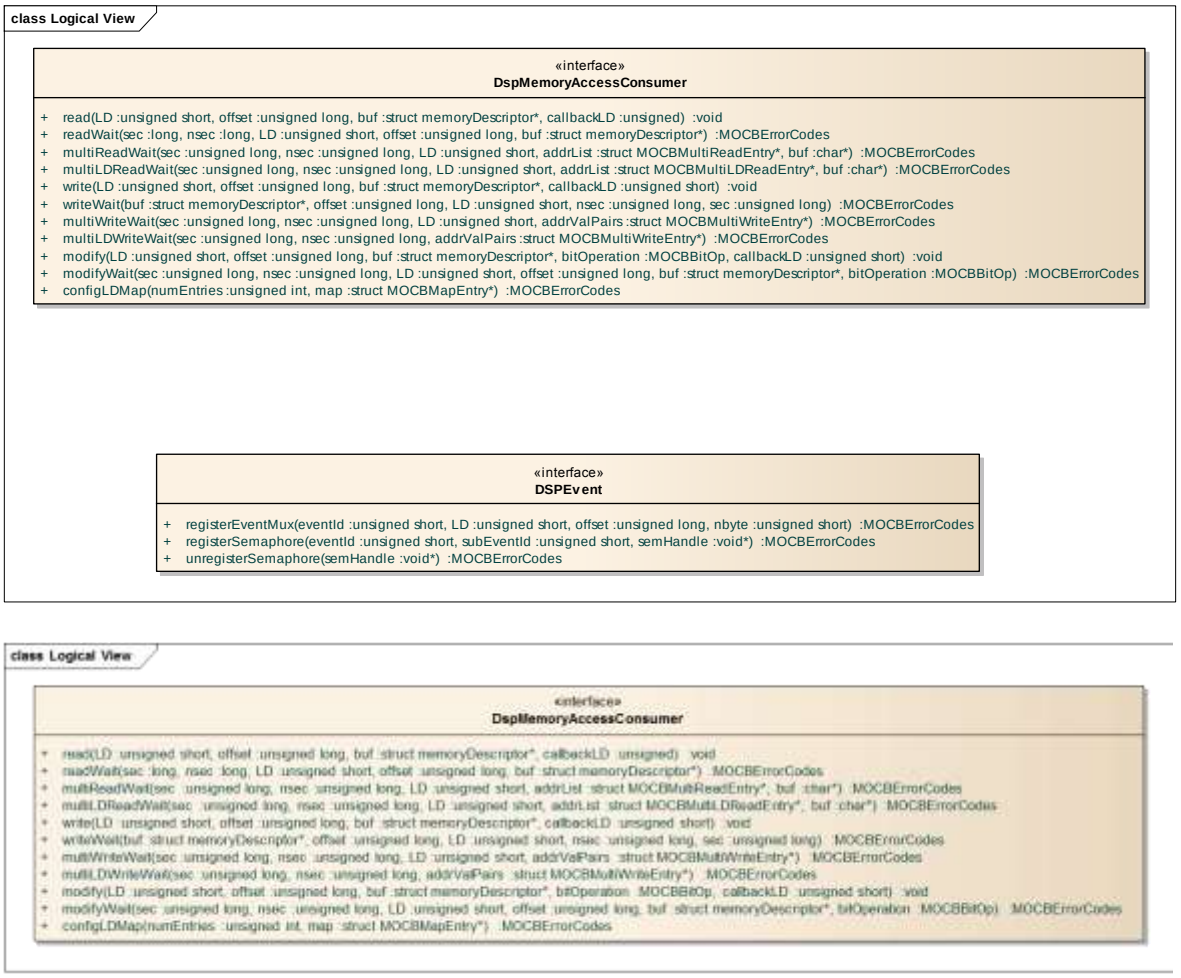

Figure 7 – MOCB DSP Interface Diagram

Note: The GPP counterpart to these interfaces is available in section B. MOCB GPP API Extension.

C.2.1.1 MOCB DSP Memory Access Consumer Interface Description

The interface design of the *DSPMemoryAccessConsumer* is shown in **Figure 7**. It provides the ability to synchronously (i.e. blocking) and asynchronously (i.e. non-blocking) read/write/modify data to and from a service user/provider's shared memory.

C.2.1.1.1 MOCB DSP Event Interface Description

The interface design of the *DSPEvent* is shown in **Figure 5**. It provides the ability to manage events.

C.2.2 Sequence Diagrams

None

C.3 SERVICE PRIMITIVES AND ATTRIBUTES

To enhance the readability of this API document and to avoid duplication of data, the type definitions of all structured types (i.e., structures, typedefs, exceptions, macros) used by the Service Primitives and Attributes have been co-located in section C.5 Data Types and Exceptions.

C.3.1 DSPMemoryAccessConsumer

C.3.1.1 *mocbRead* Operation

This operation reads data from memory. The operation is non-blocking and returns a completion status event via an MHAL message provided in the "callbackLD" parameter. This utilizes the existing MHAL Communication Routing interface defined in the MHAL DSP API Extension [1].

Callback MHAL messages are created by the user/waveform and provided by reference in the "callbackLD" parameter. The user/waveform is responsible for the persistence/management of the memory allocation.

C.3.1.1.1 Synopsis

```
void mocbRead (
   unsigned short LD,
   unsigned long offset,
   struct MOCBMemoryDescriptor* buf,
   unsigned short callbackLD
);
```

C.3.1.1.2 Parameters

Parameter Name	Description	Type	Units	Valid Range
LD	The logical destination for the message.	unsigned short	Logical Destination ID	0 – 32767
offset	Address offset from base address assigned to a logical destination (LD).	unsigned long	Offset from Logical Destination	Not Specified
buf	The structure indicating the number of bytes and buffer to store the data that is read. The buffer is pre-allocated by the caller of this operation.	struct MOCBMemoryDescriptor *	N/A	See section C.5.4.1

Parameter Name	Description	Type	Units	Valid Range
callbackLD	The logical destination for the callback message.	unsigned short	Logical Destination ID	0 – 32767

C.3.1.1.3 State

ENABLED CF::Device::operationalState.

C.3.1.1.4 New State

This operation does not cause a state change.

C.3.1.1.5 Return Value

None

C.3.1.1.6 Originator

Service Provider

C.3.1.1.7 Exceptions

None

C.3.1.2 *readWait* Operation

This operation is blocking and provides the ability to read data from memory.

C.3.1.2.1 Synopsis

MOCBErrorCodes readWait (
unsigned long sec,
unsigned long nsec,
unsigned short LD,
unsigned long offset,
struct MOCBMemoryDescriptor buf*
);

C.3.1.2.2 Parameters

Parameter Name	Description	Type	Units	Valid Range
sec	Integer seconds of time to wait.	unsigned long	Seconds	0 to 2147483647
nsec	Nanoseconds of time to wait.	unsigned long	Nanoseconds	0 to 999999999
LD	The logical destination for the message	unsigned short	Logical Destination ID	0 – 32767
offset	Address offset from base address assigned to a logical destination (LD).	unsigned long	Offset from Logical Destination	Not Specified
buf	The structure indicating the number of bytes and buffer to store the data that is read. The buffer is pre-allocated by the caller of this operation.	struct MOCBMemoryDescriptor *	N/A	See section C.5.4.1

C.3.1.2.3 State

ENABLED CF::Device::operationalState.

C.3.1.2.4 New State

This operation does not cause a state change.

C.3.1.2.5 Return Value

Description	Type	Units	Valid Range
The error code representing the status of the operation's completion	MOCBErrorCodes	N/A	See section C.5.2.2

C.3.1.2.6 Originator

Service Provider

C.3.1.2.7 Exceptions

None

C.3.1.3 *multiReadWait* Operation

This operation provides the ability for the application to read data from offsets within the same logical destination

C.3.1.3.1 Synopsis

MOCBErrorCodes multiReadWait (
unsigned long sec,
unsigned long nsec,
unsigned short LD,
struct MOCBMultiReadEntry addrList,*
unsigned short nEntries,
char buf*
);

C.3.1.3.2 Parameters

Parameter Name	Description	Type	Units	Valid Range
sec	Integer seconds of time to wait.	unsigned long	Seconds	0 to 2147483647
nsec	Nanoseconds of time to wait.	unsigned long	Nanoseconds	0 to 999999999
LD	The logical destination for the message.	unsigned short	Logical Destination ID	0 – 32767
addrList	A sequence of offsets, and number of bytes to read.	struct MOCBMultiReadEntry*	N/A	See section C.5.4.2
nEntries	Number of entries contained in the addrList struct	unsigned short	N/A	0 – 65535
buf	The data read.	char*	N/A	N/A

C.3.1.3.3 State

ENABLED CF::Device::operationalState.

C.3.1.3.4 New State

This operation does not cause a state change.

C.3.1.3.5 Return Value

Description	Type	Units	Valid Range
The error code representing the status of the operation's completion	MOCBErrorCodes	N/A	See section C.5.2.2

C.3.1.3.6 Originator

Service Provider.

C.3.1.3.7 Exceptions

None

C.3.1.4 *multiLDReadWait* Operation

This operation provides the ability for the application to read data from offsets from multiple logical destinations.

C.3.1.4.1 Synopsis

```
MOCBErrorCodes multiLDReadWait (
    unsigned long sec,
    unsigned long nsec,
    struct MOCBMultiLDReadEntry* addrList,
   unsigned short nEntries,
    char* buf
);
```

C.3.1.4.2 Parameters

Parameter Name	Description	Type	Units	Valid Range
sec	Integer seconds of time to wait.	unsigned long	Seconds	0 to 2147483647
nsec	Nanoseconds of time to wait.	unsigned long	Nanoseconds	0 to 999999999
addrList	A sequence of LDs, offsets, and number of bytes to read.	struct MOCBMultiLDReadEntry*	N/A	See section C.5.4.3
nEntries	Number of entries contained in the addrList struct	unsigned short	N/A	0 – 65535
buf	The data read.	char*	N/A	N/A

C.3.1.4.3 State

ENABLED CF::Device::operationalState.

C.3.1.4.4 New State

This operation does not cause a state change.

C.3.1.4.5 Return Value

Description	Type	Units	Valid Range
The error code representing the status of the operation's completion	MOCBErrorCodes	N/A	See section C.5.2.2

C.3.1.4.6 Originator

Service Provider.

C.3.1.4.7 Exceptions

None

C.3.1.5 *mocbWrite* Operation

This operation writes data to memory. The operation is non-blocking and returns a write confirmation via an MHAL message provided in the "callbackLD" parameter. This utilizes the existing MHAL Communication Routing interface defined in the MHAL DSP API Extension [1].

If the provided "callbackLD" is zero, a completion message is not sent when finished with the "non-blocking" transfer.

A "non-blocking" mocb*Write*() with a "callbackLD" of "NOCALLBACK" disables the confirmation callback for that instance. "NOCALLBACK" is defined as a symbolic LD reference just like "RFCHAIN" and is assigned a value by the platform.

Callback MHAL messages are created by the user/waveform and provided by reference in the "callbackLD" parameter. The user/waveform is responsible for the persistence/management of the memory allocation.

C.3.1.5.1 Synopsis

```
void mocbWrite (
   unsigned short LD,
   unsigned long offset,
   struct MOCBMemoryDescriptor* buf,
   unsigned short callbackLD
);
```

C.3.1.5.2 Parameters

Parameter Name	Description	Type	Units	Valid Range
LD	The logical destination for the message	unsigned short	Logical Destination ID	0 – 32767
offset	Address offset from base address assigned to a logical destination (LD).	unsigned long	Offset from Logical Destination	Not Specified
buf	The structure indicating the number of bytes and buffer to be written.	struct MOCBMemoryDescriptor *	N/A	See section C.5.4.1
callbackLD	The logical destination for the callback message	unsigned short	Logical Destination ID	0 – 32767

C.3.1.5.3 State

ENABLED CF::Device::operationalState.

C.3.1.5.4 New State

This operation does not cause a state change.

C.3.1.5.5 Return Value

None

C.3.1.5.6 Originator

Service Provider

C.3.1.5.7 Exceptions

None

C.3.1.6 *writeWait* Operation

This operation is blocking and provides the ability to write data to memory.

C.3.1.6.1 Synopsis

MOCBErrorCodes writeWait (
unsigned long sec,
unsigned long nsec,
unsigned short LD,
unsigned long offset,
struct MOCBMemoryDescriptor buf*
);

C.3.1.6.2 Parameters

Parameter Name	Description	Type	Units	Valid Range
sec	Integer seconds of time to wait.	unsigned long	Seconds	0 to 2147483647
nsec	Nanoseconds of time to wait.	unsigned long	Nanoseconds	0 to 999999999
LD	The logical destination for the message	unsigned short	Logical Destination ID	0 – 32767
offset	Address offset from base address assigned to a logical destination (LD).	unsigned long	Offset from Logical Destination	Not Specified
buf	The structure indicating the number of bytes and buffer to be written.	struct MOCBMemoryDescriptor *	N/A	See section C.5.4.1

C.3.1.6.3 State

ENABLED CF::Device::operationalState.

C.3.1.6.4 New State

This operation does not cause a state change.

C.3.1.6.5 Return Value

Description	Type	Units	Valid Range
The error code representing the status of the operation's completion	MOCBErrorCodes	N/A	See section C.5.2.2

C.3.1.6.6 Originator

Service Provider

C.3.1.6.7 Exceptions

None

C.3.1.7 *multiWriteWait* Operation

This operation provides the ability for the application to write data to offsets within same logical destination.

C.3.1.7.1 Synopsis

MOCBErrorCodes multiWriteWait (
unsigned long sec,
unsigned long nsec,
unsigned short LD,
struct MOCBMultiWriteEntry addrValPairs,*
unsigned short nEntries
);

C.3.1.7.2 Parameters

Parameter Name	Description	Type	Units	Valid Range
sec	Integer seconds of time to wait.	unsigned long	Seconds	0 to 2147483647
nsec	Nanoseconds of time to wait.	unsigned long	Nanoseconds	0 to 999999999
LD	The logical destination for the message.	unsigned short	Logical Destination ID	0 – 32767
addrValPairs	A sequence of offsets, and the buffers of data to be written.	struct MOCBMultiWriteEntry*	N/A	See section C.5.4.4
nEntries	Number of entries contained in the addrValPairs struct	unsigned short	N/A	0 – 65535

C.3.1.7.3 State

ENABLED CF::Device::operationalState.

C.3.1.7.4 New State

This operation does not cause a state change.

C.3.1.7.5 Return Value

Description	Type	Units	Valid Range
The error code representing the status of the operation's completion	MOCBErrorCodes	N/A	See section C.5.2.2

C.3.1.7.6 Originator

Service Provider.

C.3.1.7.7 Exceptions

None

C.3.1.8 *multiLDWriteWait* Operation

This operation provides the ability for the application to write data to offsets within multiple logical destinations.

C.3.1.8.1 Synopsis

MOCBErrorCodes multiLDWriteWait (
unsigned long sec,
unsigned long nsec,
struct MOCBMultiLDWriteEntry addrValPairs,*
unsigned short nEntries
);

C.3.1.8.2 Parameters

Parameter Name	Description	Type	Units	Valid Range
sec	Integer seconds of time to wait.	unsigned long	Seconds	0 to 2147483647
nsec	Nanoseconds of time to wait.	unsigned long	Nanoseconds	0 to 999999999
addrValPairs	A sequence of LDs, offsets, and the buffers of data to be written.	struct MOCBMultiLDWriteEntry*	N/A	See section C.5.4.5
nEntries	Number of entries contained in the addrValPairs struct	unsigned short	N/A	0 – 65535

C.3.1.8.3 State

ENABLED CF::Device::operationalState.

C.3.1.8.4 New State

This operation does not cause a state change.

C.3.1.8.5 Return Value

Description	Type	Units	Valid Range
The error code representing the status of the operation's completion	MOCBErrorCodes	N/A	See section C.5.2.2

C.3.1.8.6 Originator

Service Provider.

C.3.1.8.7 Exceptions

None

C.3.1.9 *modify* Operation

This operation modifies data in memory. The operation is non-blocking and returns a modify confirmation via an MHAL message provided in the "callbackLD" parameter. This utilizes the existing MHAL Communication Routing interface defined in the MHAL DSP API Extension [1].

If the provided "callbackLD" is zero, a completion message is not sent when finished with the transfer.

A *modify*() with a "callbackLD" of NOCALLBACK disables the confirmation callback for that instance. NOCALLBACK is defined as a symbolic LD reference just like RFCHAIN and is assigned a value by the platform.

Callback MHAL messages are created by the user/waveform and provided by reference in the "callbackLD" parameter. The user/waveform is responsible for the persistence/management of the memory allocation.

The "buf" value (at a bit level) is either "&", "|", "XOR", "!&", or "!|" to the existing memory located at LD, based on the "bitOperation".

C.3.1.9.1 Synopsis

```
void modify (
   unsigned short LD,
   unsigned long offset,
   struct MOCBMemoryDescriptor* buf,
   MOCBBitOp bitOperation,
   unsigned short callbackLD
);
```

C.3.1.9.2 Parameters

Parameter Name	Description	Type	Units	Valid Range
LD	The logical destination for the message	unsigned short	Logical Destination ID	0 – 32767
offset	Address offset from base address assigned to a logical destination (LD).	unsigned long	Offset from Logical Destination	Not Specified
buf	The structure indicating the number of bytes and buffer to modify.	MOCBMemoryDescriptor *	N/A	See section C.5.4.1
bitOperation	The operation to perform against "buf" and the memory at LD.	MOCBBitOp	N/A	See section C.5.2.3

Parameter Name	Description	Type	Units	Valid Range
LD	The logical destination for the message	unsigned short	Logical Destination ID	0 – 32767

C.3.1.9.3 State

ENABLED CF::Device::operationalState.

C.3.1.9.4 New State

This operation does not cause a state change.

C.3.1.9.5 Return Value

None

C.3.1.9.6 Originator

Service Provider

C.3.1.9.7 Exceptions

None

C.3.1.10 *modifyWait* Operation

This operation is blocking and provides the ability to modify data in memory.

C.3.1.10.1 Synopsis

MOCBErrorCodes modifyWait (
unsigned long sec,
unsigned long nsec,
unsigned short LD,
unsigned long offset,
struct MOCBMemoryDescriptor buf,*
MOCBBitOp bitOperation
);

C.3.1.10.2 Parameters

Parameter Name	Description	Type	Units	Valid Range
sec	Integer seconds of time to wait.	unsigned long	Seconds	0 to 2147483647
nsec	Nanoseconds of time to wait.	unsigned long	Nanoseconds	0 to 999999999
LD	The logical destination for the message	unsigned short	Logical Destination ID	0 – 32767
offset	Address offset from base address assigned to a logical destination (LD).	unsigned long	Offset from Logical Destination	Not Specified
buf	The structure indicating the number of bytes to be modified and the location of memory to modify.	MOCBMemoryDescriptor*	N/A	See section C.5.4.1
bitOperation	The operation to perform against “buf” and the memory at LD.	MOCBBitOp	N/A	See section C.5.2.3

C.3.1.10.3 State

ENABLED CF::Device::operationalState.

C.3.1.10.4 New State

This operation does not cause a state change.

C.3.1.10.5 Return Value

Description	Type	Units	Valid Range
The error code representing the status of the operation's completion	MOCBErrorCodes	N/A	See section C.5.2.2

C.3.1.10.6 Originator

Service Provider

C.3.1.10.7 Exceptions

None

C.3.1.11 *configLDMap* Operation

This operation maps logical destinations to a starting address in the waveform memory map. The terminal software has knowledge of which portion of its memory map are for the waveform.

C.3.1.11.1 Synopsis

MOCBErrorCodes configLDMap (
unsigned int numEntries,
struct MOCBMapEntry configMap*
);

C.3.1.11.2 Parameters

Parameter Name	Description	Type	Units	Valid Range
numEntries	Number of entries in the map	unsigned int	N/A	Not Specified
configMap	Mapping of LDs to addresses	struct MOCBMapEntry*	N/A	See section C.5.4.2

C.3.1.11.3 State

ENABLED CF::Device::operationalState.

C.3.1.11.4 New State

This operation does not cause a state change.

C.3.1.11.5 Return Value

Description	Type	Units	Valid Range
The error code representing the status of the operation's completion	MOCBErrorCodes	N/A	See section C.5.2.2

C.3.1.11.6 Originator

Service Provider

C.3.1.11.7 Exceptions

None

C.3.2 DSPEvent

C.3.2.1 *registerSemaphore* Operation

This operation registers a semaphore managed (created and destroyed) by the waveform software. The waveform calls this operation for each relevant subEvent represented by a bit in the event mux register**.

The semaphore is posted by the MOCB software when the specified MOCB event occurs. This event is known by the waveform software and the waveform firmware (e.g., an FPGA event line). The waveform hardware event line(s) is(are) mapped to terminal specific hardware (e.g., a GPIO line(s)) during waveform porting. The terminal software has knowledge of which terminal lines are connected to waveform hardware lines, and upon assertion of that discrete, the semaphore is posted.

The MOCB software provides event triggering to waveform software on a per bit basis within the event mux register.

The waveform firmware is responsible to clear all the subEvent bit(s) in the mux register after detecting the MOCB software reading the mux register (if more synchronization controls are required, the waveform may implement an independent clear/feedback register that can be written by waveform software via the MOCB *write*() or *writeWait*() operations)

MOCB software will post all subEvents present when the mux register is read after the MOCB event line is signaled.

** Note: The MOCB firmware provides a minimum of (1…n) event line(s) for waveform use. The waveform firmware provides one event mux register(1..n bytes) for each MOCB event line.

C.3.2.1.1 Synopsis

```
MOCBErrorCodes registerSemaphore (
   unsigned short eventId,
   unsigned short subEventId,
   void* semHandle
);
```

C.3.2.1.2 Parameters

Parameter Name	Description	Type	Units	Valid Range
eventId	The event Identifier to tie the semaphore to. This represents the event provided to waveform software from the MOCB FPGA interface.	unsigned short	N/A	0 – 32767

Parameter Name	Description	Type	Units	Valid Range
subEventId	Bit position in the waveform provided mux register	unsigned short	N/A	0 – 32767
semHandle	Name of the semaphore created by the application	void*	N/A	N/A

C.3.2.1.3 State

ENABLED CF::Device::operationalState.

C.3.2.1.4 New State

This operation does not cause a state change.

C.3.2.1.5 Return Value

Description	Type	Units	Valid Range
The error code representing the status of the operation's completion	MOCBErrorCodes	N/A	See section C.5.2.2

C.3.2.1.6 Originator

Service Provider

C.3.2.1.7 Exceptions

None

C.3.2.2 *unregisterSemaphore* Operation

Un-registers a previously registered semaphore registered upon a MOCB event. The only parameter needed to un-register a semaphore is the name.

C.3.2.2.1 Synopsis

MOCBErrorCodes unregisterSemaphore (void semHandle);*

C.3.2.2.2 Parameters

Parameter Name	Description	Type	Units	Valid Range
semHandle	Name of the semaphore created by the application	void*	N/A	N/A

C.3.2.2.3 State

ENABLED CF::Device::operationalState.

C.3.2.2.4 New State

This operation does not cause a state change.

C.3.2.2.5 Return Value

Description	Type	Units	Valid Range
The error code representing the status of the operation's completion	MOCBErrorCodes	N/A	See section C.5.2.2

C.3.2.2.6 Originator

Service Provider

C.3.2.2.7 Exceptions

None

C.3.2.3 *registerEventMux* Operation

This operation assigns an event to a mux register of subEvents the waveform will use. The waveform calls this operation on each MOCB event line.

C.3.2.3.1 Synopsis

MOCBErrorCodes registerEventMux (
unsigned short eventId,
unsigned short LD,
unsigned long offset,
unsigned short nByte
);

C.3.2.3.2 Parameters

Parameter Name	Description	Type	Units	Valid Range
eventId	The event Identifier associated with the discrete line.	unsigned short	N/A	0 – 32767
LD	The logical destination for the message. Location in the waveform memory space where the platform reads "nbytes" of event mux register when "eventId" occurs	unsigned short	Logical Destination ID	0 – 32767
offset	Address offset from base address assigned to a logical destination (LD).	unsigned long	Offset from Logical Destination	Not Specified
nByte	The number of bytes to be read.	unsigned short	N/A	0 – 65525

C.3.2.3.3 State

ENABLED CF::Device::operationalState.

C.3.2.3.4 New State

This operation does not cause a state change.

C.3.2.3.5 Return Value

Description	Type	Units	Valid Range
The error code representing the status of the operation's completion	MOCBErrorCodes	N/A	See section C.5.2.2

C.3.2.3.6 Originator

Service Provider

C.3.2.3.7 Exceptions

None

C.4 INTERFACE DEFINITIONS

None

C.5 DATA TYPES AND EXCEPTIONS

C.5.1 Data Types

None

C.5.2 Macros

C.5.2.1 MOCBAddressIndexType

This enumeration definition enumerates the address indexing that can be performed.

typedef unsigned short MOCBAddressIndexType;
const MOCBAddressIndexType CONSTANT = 0;
const MOCBAddressIndexType INCREMENT = 1;
const MOCBAddressIndexType DECREMENT = 2

unsigned short	Element	Value	Description
MOCBAddressIndexType	CONSTANT	0	Constant address indexing
	INCREMENT	1	Increment address indexing
	DECREMENT	2	Decrement address indexing

C.5.2.2 MOCBErrorCodes

This type definition is a JTRS extension enumeration (see JTRS CORBA Types [3]). It enumerates the error codes supported by the MOCB.

typedef short MOCBErrorCodes;
const MOCBErrorCodes SUCCESSFUL = 0;
const MOCBErrorCodes INV_MEM_ACCESS = -1;
const MOCBErrorCodes INV_LD_OR_ADDR = -2;
const MOCBErrorCodes SRC_MEM_NOT_ALIGNED = -3;
const MOCBErrorCodes DST_MEM_NOT_ALIGNED = -4;
const MOCBErrorCodes INV_EVENT_ID = -5;
const MOCBErrorCodes INV_SEMAPHORE_NAME = -6;
const MOCBErrorCodes INV_TIME = -7;
const MOCBErrorCodes INV_SUBEVENT = -8;
const MOCBErrorCodes TIMER_EXPIRED = -9;
const MOCBErrorCodes INV_SIZE = -10;

short	Element	Value	Description
MOCBErrorCodes	SUCCESSFUL	0	Successful. The transfer was successfully queue/executed
	INV_MEM_ACCESS	-1	Invalid Memory Access. The memory location addressed was outside the platform's valid range.
	INV_LD_OR_ADDR	-2	Invalid LD / Address. The LD does not have a valid address map entry.
	SRC_MEM_NOT_ALIGNED	-3	Source Memory Not Aligned. The (source) location where the data is originally stored, is not located on the processors defined address boundary for an efficient memory transfer. i.e. An error would occur if addressing the second byte in a 32-bit word, as the start of a word transfer.
	DST_MEM_NOT_ALIGNED	-4	Destination Memory Not Aligned. The location where the data will be stored is not located on that processors defined addressable boundary for an efficient memory transfer. i.e. An error would occur if addressing the second byte in a 32-bit word, as the start of a sequence of word transfer.
	INV_EVENT_ID	-5	Invalid EventID. The event ID does not correspond to a valid MOCB event line.
	INV_SEMAPHORE_NAME	-6	Invalid Semaphore Name. The event string has not been mapped to a subEventId.
	INV_TIME	-7	Invalid Time (i.e. sec or nsec). The time specified is outside the valid range.
	INV_SUBEVENT	-8	Invalid SubEventID The subEventId is outside the bounds of the mux register size.
	TIMER_EXPIRED	-9	Timer expired. The time specified expired prior to completion of the operation.

short	Element	Value	Description
	INV_SIZE	-10	Invalid size. The number of bytes requested exceeds the limit that can be returned via the MHAL callback.

C.5.2.3 MOCBBitOp

This enumeration definition is a JTRS extension enumeration (see *JTRS CORBA Types* [3]). It enumerates the bitwise operations that can be performed.

typedef unsigned short MOCBBitOp;
const MOCBBitOp AND = 0;
const MOCBBitOp OR = 1;
const MOCBBitOp XOR = 2;
const MOCBBitOp NAND = 3;
const MOCBBitOp NOR = 4;

unsigned short	Element	Value	Description
MOCBBitOp	AND	0	A bitwise “and”.
	OR	1	A bitwise “or”.
	XOR	2	A bitwise “xor”.
	NAND	3	A bitwise “nand”.
	NOR	4	A bitwise “nor”.

C.5.3 Exceptions

Not applicable

C.5.4 Structures

C.5.4.1 MOCBMemoryDescriptor

This structure defines the byte size "of" and pointer "to" a memory allocation.

```
struct MOCBMemoryDescriptor {
   unsigned short nByte;
   char* bufAddr;
};
```

Struct	Attributes	Type	Valid Range	Description
MOCBMemoryDescriptor	nByte	unsigned short	0 – 65531	Set by the user/waveform and used by MOCB
	bufAddr	char*	N/A	Allocated by the user/waveform and used by MOCB

C.5.4.2 MOCBMultiReadEntry

This structure defines the pairing between an offset and number of bytes to read.

```
struct MOCBMultiReadEntry {
   unsigned long offset;
   unsigned short nByte;
};
```

Struct	Attributes	Type	Valid Range	Description
MOCBMultiReadEntry	offset	unsigned long	Not Specified	Address offset from base address assigned to a logical destination (LD).
	nByte	unsigned short	0 – 65531	The number of bytes to read.

C.5.4.3 MOCBMultiLDReadEntry

This structure defines the pairing between an offset and number of bytes to read.

```
struct MOCBMultiLDReadEntry {
   unsigned short LD;
   unsigned long offset;
   unsigned short nByte;
};
```

Struct	Attributes	Type	Valid Range	Description
MOCBMultiLDReadEntry	LD	unsigned short	0 – 32767	The logical destination for the message.
	offset	unsigned long	Not Specified	Address offset from base address assigned to a logical destination (LD).
	nByte	unsigned short	0 – 65531	The number of bytes to read.

C.5.4.4 MOCBMultiWriteEntry

This structure defines the pairing between an offset and buffer of data to be written.

```
struct MOCBMultiWriteEntry {
   unsigned long offset;
   char* buf;
  unsigned short nByte;
};
```

Struct	Attributes	Type	Valid Range	Description
MOCBMultiWriteEntry	offset	unsigned long	Not Specified	Address offset from base address assigned to a logical destination (LD).
	buf	char*	N/A	The data to be written.
	nByte	unsigned short	0 – 65535	The number of bytes to write.

C.5.4.5 MOCBMultiLDWriteEntry

This structure defines the pairing between an LD, offset and buffer of data to be written.

```
struct MOCBMultiLDWriteEntry {
   unsigned short LD;
   unsigned long offset;
   char* buf;
   unsigned short nByte;
};
```

Struct	Attributes	Type	Valid Range	Description
MOCBMultiLDWriteEntry	LD	unsigned short	0 – 32767	The logical destination for the message.
	offset	unsigned long	Not Specified	Address offset from base address assigned to a logical
	buf	char*	N/A	The data to be written.
	nByte	unsigned short	0 – 65535	The number of bytes to write.

C.5.4.6 MOCBMapEntry

This structure defines the mapping between a logical destination and address in memory.

```
struct MOCBMapEntry {
   unsigned short LD;  // the logical destination
   unsigned long address;  // the address this LD maps to
  MOCBAddressIndexType indexType;  // the index type for this address
};
```

Struct	Attributes	Type	Valid Range	Description
MOCBMapEntry	LD	unsigned short	0 – 32767	The logical destination for the message.
	address	unsigned long	Not Specified	The address the LD maps to.
	indexType	MOCBAddressIndexType	See section C.5.2.1	The index type for this address.

APPENDIX C.A – ABBREVIATIONS AND ACRONYMS

See section Appendix A.A.

APPENDIX C.B – PERFORMANCE SPECIFICATION

Not applicable

D. MOCB FPGA API EXTENSION

D.1 INTRODUCTION

The *MOCB FPGA* API consists of a collection signals and busses between an initiator and a target that provide services to route MHAL communications. For the purposes of this API, the following applies to processor naming conventions:

- An FPGA represents a HDL capable processor, again without CORBA capability.

At build time, an MOCB FPGA interface precompiled core or VHDL is compiled together with the waveform HDL to form a single loadable FPGA image for the target platform. An HDL entity description defines MOCB interface signals available to the waveform FPGA developer.

The MOCB interface was created to utilize memory-mapped interfaces for a read (pull) capability between Waveform Components. This allows Components, as shown in **Figure 8**, to achieve better real-time behaviors with less isolation. For example if we assume CE #1 is a DSP and CE #2 is an FPGA, this interface gives the DSP capability to write/read data to/from Component B or C. Likewise, in the FPGA, Component B could use the interface internally to read/write data in Component C.

Note: In **Figure 8**, the MOCB is assumed to be part of the orange "MHAL" blocks.

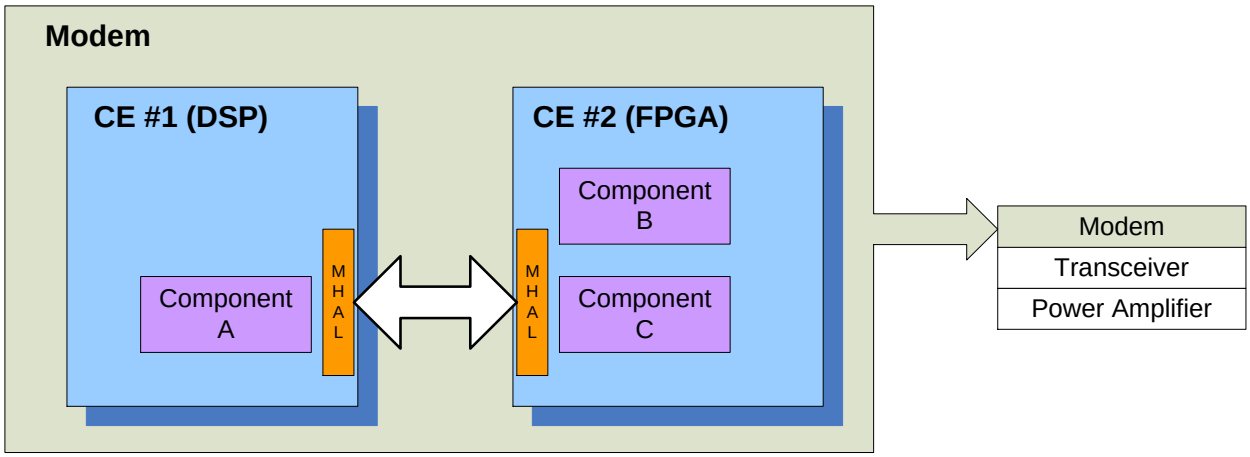

Figure 8 – Waveform Component Allocation Example

Another feature incorporated into the interface is event lines. For example in Figure **8**, Component C might signal "data needs to be read" to Component A. As a result "data is read" from the FPGA by the DSP. The platform designer determines the number and location of the Event lines.

The MOCB interface, illustrated in Figure 9, consists of two interface categories and three categories of signals.

The interfaces are:

- Address / Data bus interface

- Event interface

The categories of signals are:

- Required basic interface signals
- Optional Basic Signals
- Extended interface signals

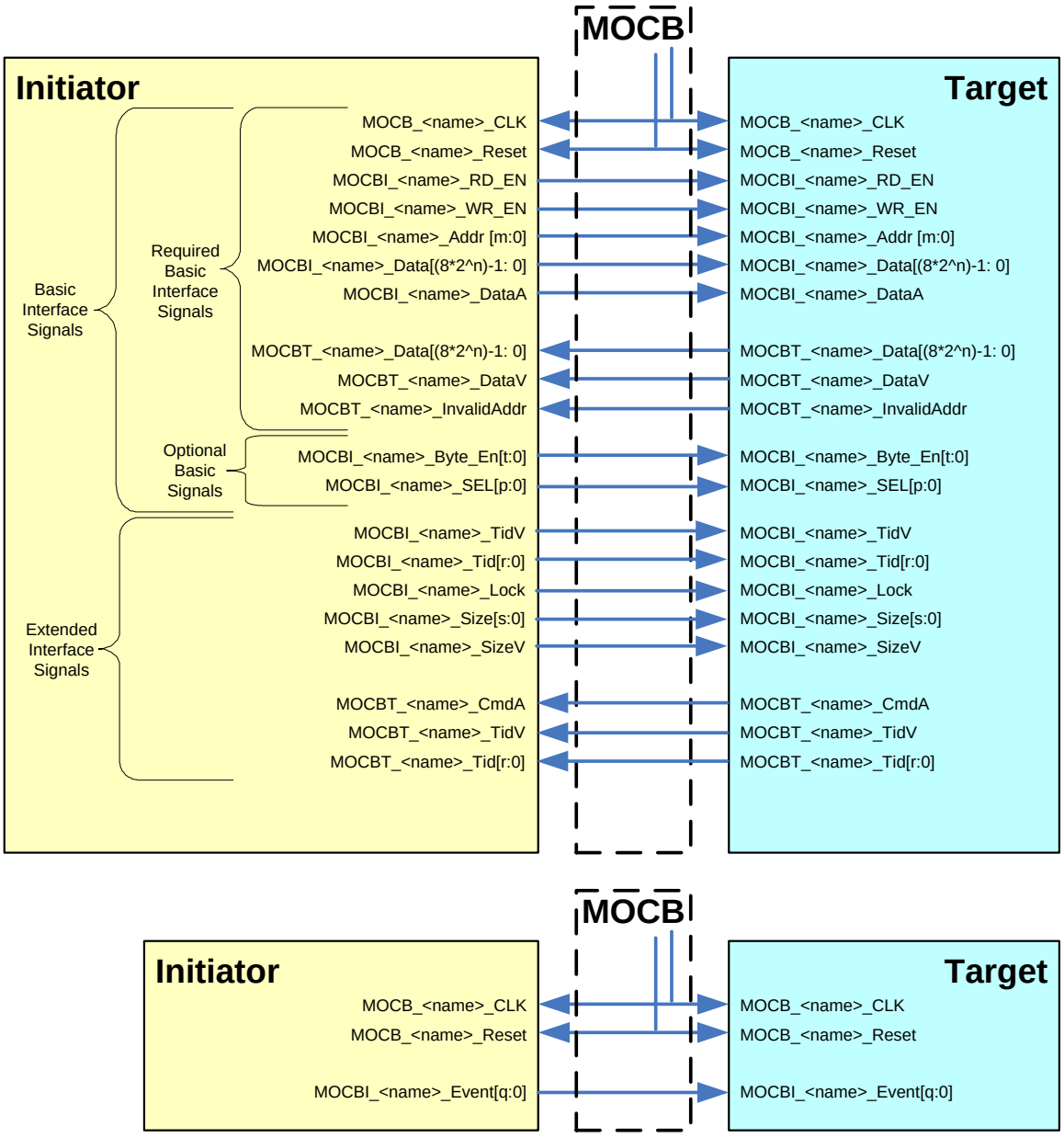

Figure 9 – MOCB Bus Interface

The address data signals are the basic set of signals required to provide data access to/from the FPGA waveform. They are intended for processor communication, radio control interface, external memory interface, inter-chip communication etc. The platform provides a single access point for address / data per FPGA per waveform per hardware interface. If the FPGA has multiple hardware interfaces intended for waveform use, the platform will abstract each interface to a unique separate MOCB access point. If

the waveform spans multiple FPGAs, each FPGA will provide an access point per hardware interface on the respective FPGA acceptable by the waveform.

The Event interface provides a means to signal events or occurrences. The initiator sources the event signals to the target. The Event initiator may reside in either the platform or waveform logic.

D.1.1 Overview

This document contains as follows:

a. Section D.1, *Introduction*, of this document contains the introductory material regarding the Overview, and Service Layer description.

b. Section D.2, *Services*

c. Section D.3, *Service Primitives and Attributes*

d. Section D.4, *Definitions*

e. Section D.5, *Data Types and Exceptions*

f. Appendix D.A – *Abbreviations and Acronyms*

g. Appendix D.B – *Performance Specification*

h. Appendix D.C – *Clock Specification*

D.1.2 Service Layer Description

D.1.2.1 MOCB FPGA Signals

D.1.2.1.1 FPGA Signals Naming Convention

The MOCB interface provides for a standardized naming convention for the benefit of readability, commonality and the identification/differentiation from other bus signals that may exist within the FPGA.

- All MOCB signals begin with a MOCB (I,T) identifier prefix with “I” for initiator or a “T” for target to identify the driver of each signal.

The standard provides for an optional user defined field to allow the user to uniquely identify the individual MOCB bus from other MOCB interfaces that may reside within the FPGA. The final required field is the functional identifier of the signal.

Example
MOCBI_WF_A_Data = MOCB data bus driven by an initiator to waveform A
MOCBT_WF_A_Data = MOCB data bus driven by a target from waveform A

D.1.2.1.2 MOCB FPGA Best Design Practice Implementation

In order to ensure design portability and compatibility of independently developed cores, the MOCB interface conform to some basic best practice design rules. This will ensure that logic independently

developed by different development teams will contain the same basic design practices and thus avoid common interfacing compatibility issues.

- MOCB Interface be synchronous to the MOCB clock.
- MOCB Interface has an asynchronously applied, synchronously released reset signal used to reset target and initiator logic to a default state.
- All MOCB Interface signaling be active high.
- All MOCB Interface signals are driven from a MOCB clocked register.
- MOCB Interface be little endian.
- MOCB Interface is used to abstract all external HW interfaces, including but not limited to processor interfaces, radio frequency control, external RAM memory, and FLASH.
- MOCB Interface address signals represent a byte address regardless of the data width.

Note1: The MOCB standard does not define the source of the MOCB clocks It may be a platform interface clock or a waveform clock (see Appendix D.C – Clock Specification). For implementations where the waveform requires a different clock than the platform provides, interconnect translation logic would need to be created to transfer between the initiator and target interface.

Note2: The MOCB does not preclude the use of logic locks or source code for the platform.

Note3: The MOCB does not specify the location of the bus adaptation and it may be in the waveform or platform.

D.1.2.1.3 Basic MOCB FPGA Signals

The basic MOCB interface signals are the basic set of signals required by the platform to adhere to the MOCB interface protocol. The actual set of signals implemented will be dependent on the capabilities of the hardware platform or needs of the waveform. For instance, if the hardware platform provides a push packet interface, it may not be practical to provide a read enable on the MOCB interface. The signals provided to the MOCB interface are left to the platform and waveform developers to determine. The Basic MOCB Interface satisfies the following requirements:

- MOCB Interface uses the Basic Interface Signals (Input Data, Output Data, Address, Clock, Read/Write enables, Data Valid, and Event lines) based on the configuration of the hardware platform.
- MOCB compliant systems utilize an interconnect fabric to translate the initiator address width to the target address width.
- MOCB Interface address width is adjusted to account for differences between the platform data width and the waveform data width. (For example, drop the least significant address bit when converting a 16-bit platform data bus to a 32-bit waveform data bus.)
- MOCB compliant systems utilize an interconnect fabric to translate the platform data width to the waveform data width. (For example, logic is needed to take two 16-bit platform accesses and form a 32-bit waveform access)
- MOCB Interface limits the data bus width to 8 times a power of 2 (e.g. 8 bits, 16 bits, 32 bits, etc.).
- MOCB Interface supports Data Accept (DataA) signal for flow control.
- MOCB Interface supports single-cycle writes for a single word or back-to-back burst transfers as illustrated in the timing diagrams in section D.1.2.3**.**

- MOCB Interface supports single-cycle reads for a single word or back-to-back burst transfers as illustrated in the timing diagrams in section D.1.2.3**.**
- MOCB Interface provides one or more event lines whose functionality will be user defined but synchronous to MOCB clock (Event).
- MOCB Event Interface Initiator provides a single cycle pulse, synchronous to MOCB clock, to indicate an active event
- MOCB Interface supports variable latency reads ("DataV").
- MOCB Interface Targets always provides a DataV signals coinciding with the return of valid data.
- MOCB Interface Initiators utilizes the DataV as an indication of Valid Data return**.**

Figure 10 illustrates the smallest subset of signals provided by the platform to the waveform to provide communication to/from the outside software and hardware components. It is not required that the waveform utilizes the entire set of signals, even though the signals are required to be provided by the platform. If the waveform has no functional use for a particular set of signals, it may drop the signals from its interface. The platform must still provide the complete set of basic signals for use by future waveforms. If the waveform and platform required functions and features do not align, a translation layer is required.

The MOCB Clock is the clock for all signals associated with a particular Initiator/Target(s) pair. The source of the MOCB clock system dependent. If the required clock of the Initiator and Target do not align, a translation layer may be required as part of the porting tasks,

The MOCB Reset is an asynchronously set, synchronously released reset. It is synchronous to the MOCB CLK.

The MOCB Read/Write lines are active high signals that indicate if the Initiator intends to perform a data transfer. The read write lines are also used to as part of the arbitration protocol. When using the extended command accept feature (CmdA), the read / write signals serve as a request lines. The Initiator would present the address and active read or write line. This indicates to the Target the Initiator intends perform a transfer. The target will indicate the command has been accepted by driving the Command Accept signal active. Section D.1.2.5.6 illustrates an example transfer.

The MOCB address bus is the address presented to the Target by the Initiator.

The MOCB Data Bus is defined as the bus used to synchronously pass data between the Initiator and the Target. It is limited to 8 times the power of '2' in width. If the target and Initiator bus width do not align, a translation layer may be constructed to adapt the target to the Initiator if neither the target nor the initiator provide bus adaptation.

The MOCB protocol provides the designer the capability to throttle or hold off return data from the Target by utilizing the Data Accept (DataA). This permits stalling the return of data from the Target. In the configuration where there is a single initiator attached to many targets and the targets have a variable latency read return, it may be necessary to stall one of the targets data return to avoid data bus collisions when reading across multiple targets in a burst or contiguous read. The data return accept function should be used in conjunction with the command accept function. When the initiator stalls the return data pipeline, the target should also stall the command pipeline to keep from over running the target with commands it cannot service. The intent is that the target merely stalls its command and data pipeline rather than forced to store commands internally until the data can be returned to the initiator. The

pipeline itself becomes the storage elements to store in flight commands and return data. IF the designer chooses to implement the Data Accept without use of the Command accept, the target will be forced to buffer return data until the target can accept the data.

The MOCB Event lines provide a means of signaling an event or occurrence to or from the waveform. The Event lines pass both into the waveform as well as from the waveform. Pulse Per Second (PPS) was an example of an event that could be passed into the waveform. A read request is an example of an event line that would be passed from the waveform. The Event lines are segregated into a separate interface class. They may have (but not required to have) their unique CLK and reset. The direction on the event lines follows the standard nomenclature of the interface. The Initiator defines the source of a transaction. The Target defines the destination of the transaction. In the case of data transfer across the data interface, the Initiator originates the transfer. The target is the logic acted upon for the transaction. It does not reflect the actual flow of data, just the source of the transfer command. In the case of the event lines, the event originates from the Initiator and is passed into the Target. The events are not polled. In the case of events from the waveform, the EVENT Initiator would reside in the waveform. The platform would provide a Target to receive the events.

The DataV signal was created to satisfy two basic needs; variable latency returns within a platform and performance variations between different platforms.

Variable latency returns encompasses many scenarios. They include reads across asynchronous boundaries, non-deterministic interfaces or shared bus access where transfers may be stalled. When applied to the variable return latency, the feature allows for the simplification of logic within the waveform and platform as well as increasing performance. There would be no need to buffer data and hold until all data is returned before forwarding to the Initiator based on a set latency. In cases where a set latency were required, it would have to be greater than the maximum latency variation of the target. Forcing a set latency adds complexity and hinders performance. For waveforms that require a set latency, accommodations would need to be made as part of the porting effort to support the needs of the waveform. However, in cases where variations in read returns posed no issue to the waveform design, there would be advantage to the waveform as well as platform in the form of complexity/resource reduction and performance increases.

A secondary advantage of the DataV is to accommodate differences in platform performance. For example, if a RAM access takes 3 cycles on platform X and a similar RAM access takes 4 cycles on platform Y, a waveform, ported on both platforms may not have to change logic to accommodate the variations from platform to platform. The waveform may simply trigger off the DataV.

All Targets must provide a DataV. However, it is left to the logic designer to decide if they want to take advantage for the DataV in the initiator logic. A designer may choose to use the DataV or the Target Data Valid Latency constant to customize the logic

While the DataV feature allows for the possibility of variable data returns, it does not mandate that data returns will be variable. For example, in the case where the RAM is attached to the FPGA for the sole use of the waveform, the returns would be deterministic. All reads would require the same number of CLK cycles. The Initiator may choose to use a DataV or the latency number specified in the package file (section D.1.2.3.5.6) though a generic variable. In both cases, the waveform logic could assume data returned in a deterministic time frame.

The MOCB Invalid Address signal is used by the target to notify the Initiator that the requested address was invalid. Because the MOCB protocol relies on the DataV to complete a read transfer, an invalid

address hit may cause a lock up condition. A target may not provide a DataV return when reading from an invalid address space. Without an indication from the target indicating a transfer is complete or invalid, the Initiator will hang waiting for a the DataV. The Invalid address signal provides the designer a means of terminating invalid address transfer without indicating valid data has been returned. It may also be used in error reporting logic.

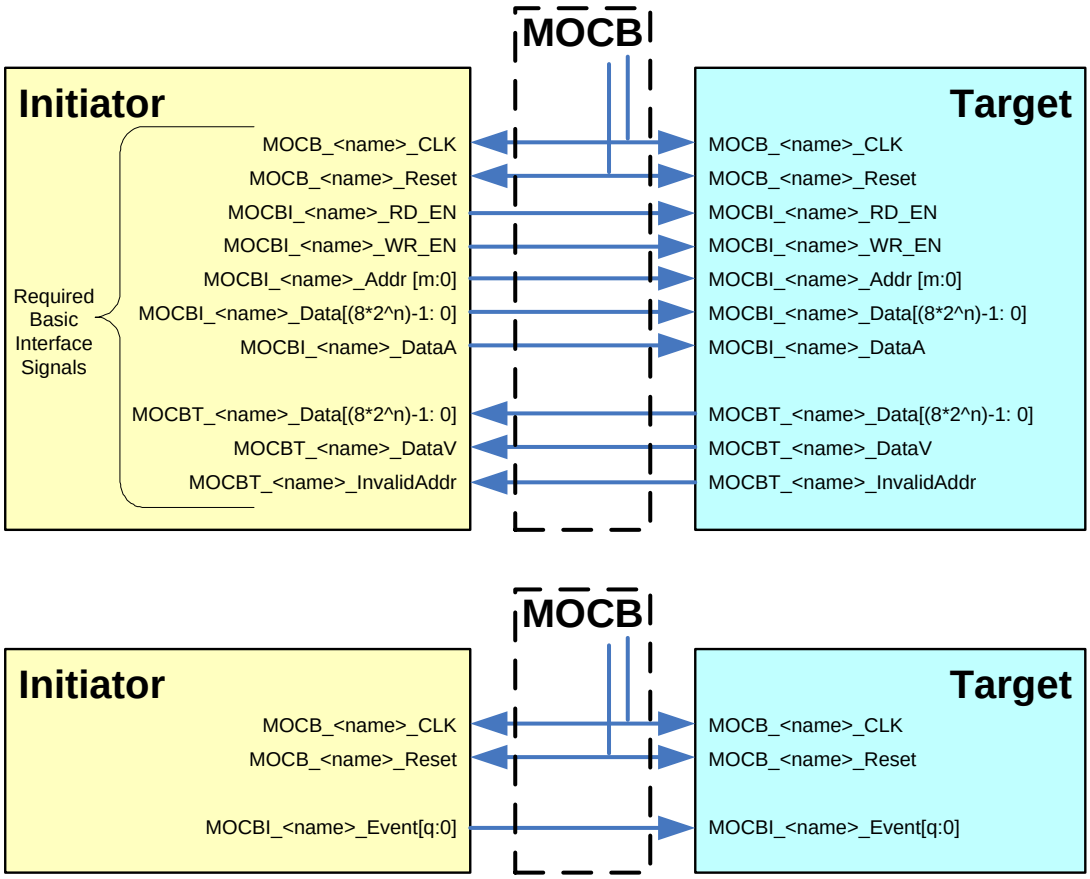

Figure 10 – MOCB **Required Basic Bus Interface Signals

**Note: The platform must provide all required signals, however a waveform is not required to utilize all required signals.

Common Signals

MOCB_<name>_CLK	MOCB clock – Data and control signals are captured or driven on the rising edge of the MOCB Clock
MOCB_<name>_Reset	Synchronous Released Reset

Initiator Driven Signals

MOCBI_<name>_RD_EN	Read Enable signal used to identify the transfer and a data fetch
MOCBI_<name>_WR_EN	Write Enable signal used to identify the transfer and data deposit
MOCBI_<name>_Addr [m:0]	Address bus (m down to 0) user to address the data transfer

MOCBI_<name>_Data[(8*2^n)-1: 0]	Data bus – Data Bus used to transfer data to the target. The Data bus width is required to 8 times a power of 2 (for example: 8 bits, 16 bits, 32 bits…).
MOCBI_<name>_DataA	Return Data Accepted by Target
MOCBI_<name>_Event[q:0]	User defined event lines used to signal a CE of an event

Target Driven Signals

MOCBT_<name>_Data[(8*2^n)-1: 0]	Data Bus used to transfer data to the Initiator. The Data bus width is required to be 8 times a power of 2 (for example: 8 bits, 16 bits, 32 bits…).
MOCBT_<name>_DataV	Data Valid - Indicates when returned data is valid for each clock cycle on the bus. Data Valid must be active for each valid word of the return data on each cycle of the bus.
MOCBT_<name>_InvalidAddr	Invalid address signal.

D.1.2.1.4 Basic MOCB FPGA Signals (Optional)

The optional MOCB interface signals provide the designer additional signals in order to optimize the interface for the needs of the platform and waveform. The MOCB Optional Basic Interface satisfies the following requirements:

- MOCB Interface ***may*** utilize the byte enable signal to specify specific bytes within a data word.
- MOCB Interface ***may*** support an address select line.

Figure 11 illustrates the optional basic signals defined by the MOCB protocol. The platform designer is free to choose to implement these signals or not. The intent of the signals is to provide the designer with some of the common features available on hardware components and thus reduce the logic required for abstraction.

The MOCB select line will allow the user to take advantage of memory select/chip select lines available on many common DSP and GPP processors. The designer may also choose to utilize the MOCB select to handle an overlapping address mapping.

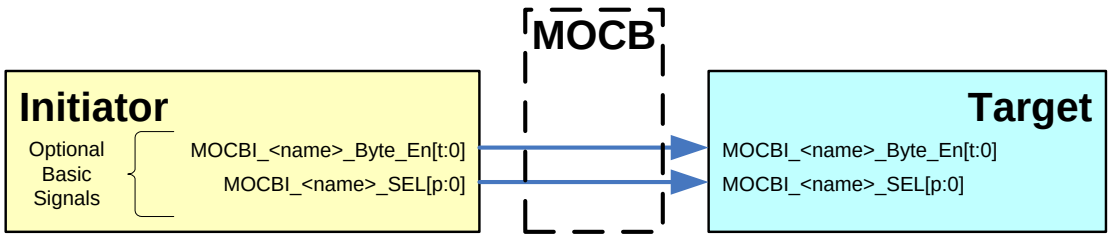

Figure 11 – Optional Basic Signals

Initiator Driven Signals

MOCBI_<name>_Byte_En[t:0]	Byte Enable
MOCBI_<name>_SEL[p:0]	Address memory select

D.1.2.1.5 Extended MOCB PFGA Signals

The Extended MOCB interface signals provide the platform greater flexibility for complex architectures, supports advanced flow Control and enable the waveform(s) to gain ownership of the bus through arbitration. The MOCB Extended Interface satisfies the following requirements:

- MOCB Interface ***may*** use the Extended Interface Command Accept (CmdA) Signal for flow control.
- MOCB Interface ***may*** use the Extended Interface Command Accept (CmdA) Signal for access control.
- MOCB Interface ***may*** use the Extended Interface Command Accept (CmdA) function for flow control when implementing the Interface Data Accept (DataA) function.
- MOCB Interface ***may*** use the Extended Interface Transaction ID (Tid) and Transaction Valid (TidV) Signals for out of order read responses.
- MOCB Interface ***may*** use the Extended Interface Lock Signal for locked transfers.
- MOCB Interface ***may*** use the Extended Interface Size Signal to provide the target with the transfer size in advance.
- MOCB Interface ***may*** define separate master and slave interfaces, such that any given interface is either a master or a slave, but not both.
- MOCB Interface ***may*** support multi-master bus control ("Command Accept").
- MOCB Interface ***may*** support interrupted or stalled transactions. ("Command Accept / DataA").
- MOCB Interface ***may*** support locked transactions to provide bus exclusivity to a given user. ("lock").
- MOCB Interface ***may*** support out-of-order read data by using the Transaction ID (Tid) signals.

Figure 12 illustrate the extended optional signals available to the developer within the MOCB protocol. These signals provide flexibility and scalability for high performance, multi-master advanced control platforms. Utilizing various combinations of these signals, the MOCB interface will be able to support data transfer throttling, bus arbitration, critical transfer locks, read modify write locks, out of band transfer size notification, and out of order data responses. When porting a waveform to an individual platform, there may be instances where the supported extended features or required features of a platform and waveform do not align. If the platform and waveform have not provided the needed functionality or configurability, a translation layer may be required to provide a bridge between the platform and waveform logic. The creation of the translation logic would be part of the porting exercise. D.1.2.4 Translation Layer illustrates an example of a logic layer required to mate a MOCB waveform with a MOCB platform with incompatible configurations.

The MOCB extended interface bus protocol provides the designer the capability of utilizing the Command Accept signal (CmdA) for both bus arbitration and data throttling (flow Control). The Target can use the signal to grant access to the bus as well as throttle the data transfer. The intent of this feature is to provide the capability of allowing an Initiator to request access to a bus and allow the Target, with support of an arbiter, to grant access. The feature is also useful for throttling the flow of data between interfaces' with different bandwidths. The Command Accept feature can be used to throttle data flow between the initiator and the target interfaces by holding off or stalling commands.

The MOCB extended interface bus protocol provides the designer the capability to throttle or holding off the return data from the Target by utilizing the Data Accept (DataA). The data accept allows the

designer to stall the return of data from the Target. In the configuration where there is a single initiator attached to many targets and the targets have a variable latency read return, it may be necessary to stall one of the targets data return to avoid data bus collisions when reading across multiple targets in a burst or contiguous read. The data return accept function should be used in conjunction with the command accept function. When the initiator stalls the return data pipeline, the target should also stall the command pipeline to keep from over running the target with commands it cannot service. The intent is that the target merely stalls its command and data pipeline rather than being forced to store commands internally until the data can be returned to the initiator. The pipeline itself becomes the storage elements to store in flight commands and return data. If the designer chooses to implement the data Accept without use of the Command accept, the target will be forced to buffer return data until the target can accept the data.

The MOCB extended interface bus protocol provides the designer with the option to incorporate a LOCK feature. The Lock signal indicates to the Target arbiter that the Initiator needs to retain ownership of the bus during a transfer as well as between transfers. This prevents the Target from granting access to a higher priority transfer during the current transfer or from giving ownership of the bus to another requester between transfers. Once a series of transfers starts with the lock signal set, the target arbiter must maintained continued ownership of the bus to that initiator.

The MOCB extended interface bus protocol provides the designer with the option to utilize the SIZE field of the interface. The intent of this field is to advertise the size of the transfer in advance to allow Direct Memory Access (DMA) to be set up to handle the transfer or to form a data packet header for a push packet interface. This reduces the need to buffer transfer to determine the size.

The MOCB extended interface bus protocol provides the designer with the ability to "TAG" read transfers with a unique transfer identification number (Tid). The intent of this feature is to provide the application the ability to support out of order transfers. In some platform applications, it is difficult to guarantee multiple read transfers to multi sinks will be returned in the order they were requested. This is especially true for high-speed push packet interfaces such as Ethernet. Providing Transaction IDs enables the application to tag a transfer with a unique ID that will be returned with the data and used to inform the Initiator what transaction the data is associated with.

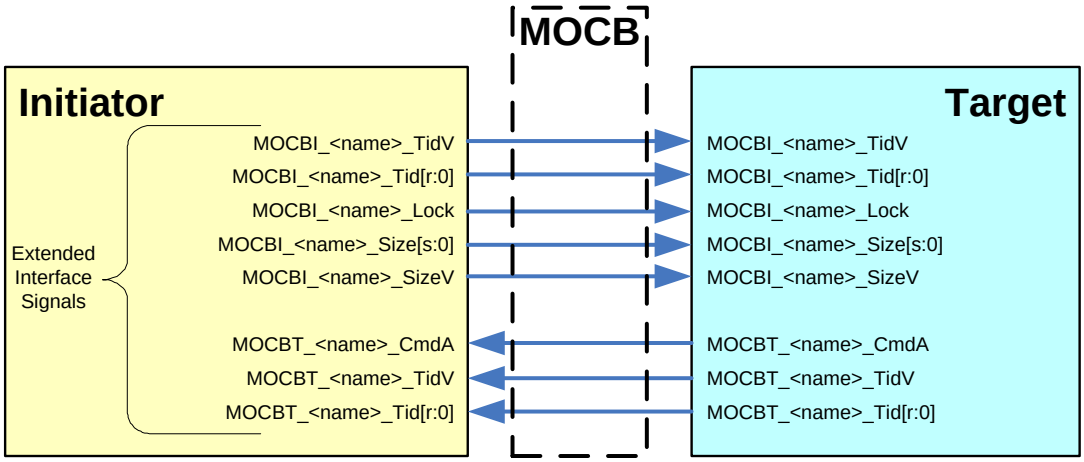

Figure 12 – MOCB Extended Interface Signals

Initiator Driven Signals

MOCBI_<name>_TidV	Transaction ID Valid
MOCBI_<name>_Tid[r:0]	Transaction ID
MOCBI_<name>_Lock	Transfer Locked
MOCBI_<name>_Size[s:0]	Transfer Size
MOCBI_<name>_SizeV	Transfer Size Valid

Target Driven Signals

MOCBT_<name>_CmdA	Command Accept
MOCBT_<name>_TidV	Return Transaction ID Valid
MOCBT_<name>_Tid[r:0]	Return Transaction ID

D.1.2.2 Data and Control Flow

The MOCB interface is broken down into two primary functional components, the Initiator and the Target. This section will illustrate the necessary flow to operate as one or the other of these primary functions.

The flow chart is depicted using a standardized symbol flow system to aid in the decision and process of performing proper bus protocol to meet the MOCB requirements. See **Figure 13** for the list of standard symbol definitions.

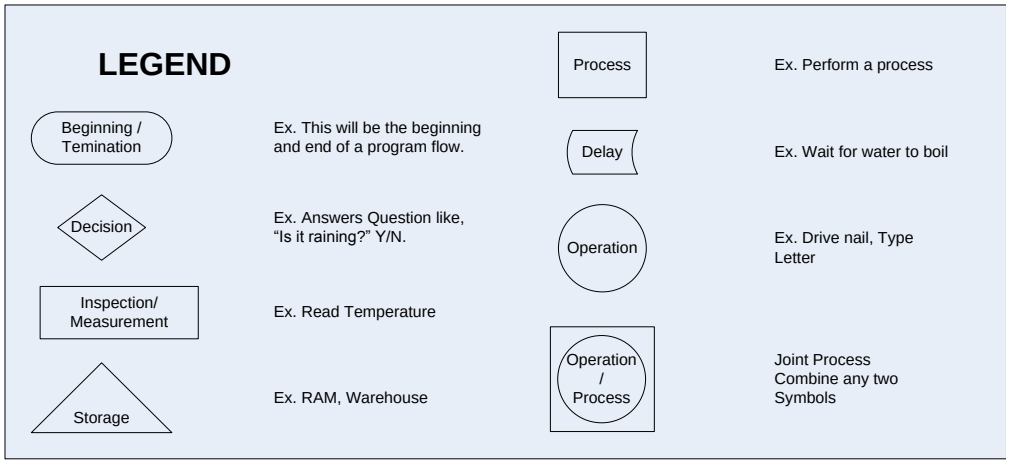

Figure 13 – Basic Flowchart Standard Definitions

Since the interface has both basic and enhanced features, a super-set is illustrated to demonstrate the most complex to the simplest implementation. The optional or enhanced control and data flow features are identified using dashed lines of connection, isolating them from the basic flow features.

Many aspects for the logic necessary to perform Data, Address, and Tid assignments are not captured and should be implementer specific.

In the diagrams, note where interface signaling and data is passed out of the flow of the Target and into the Initiator and vice versa. This is done with the use of arrows showing the origin or destination for a particular signal or data.

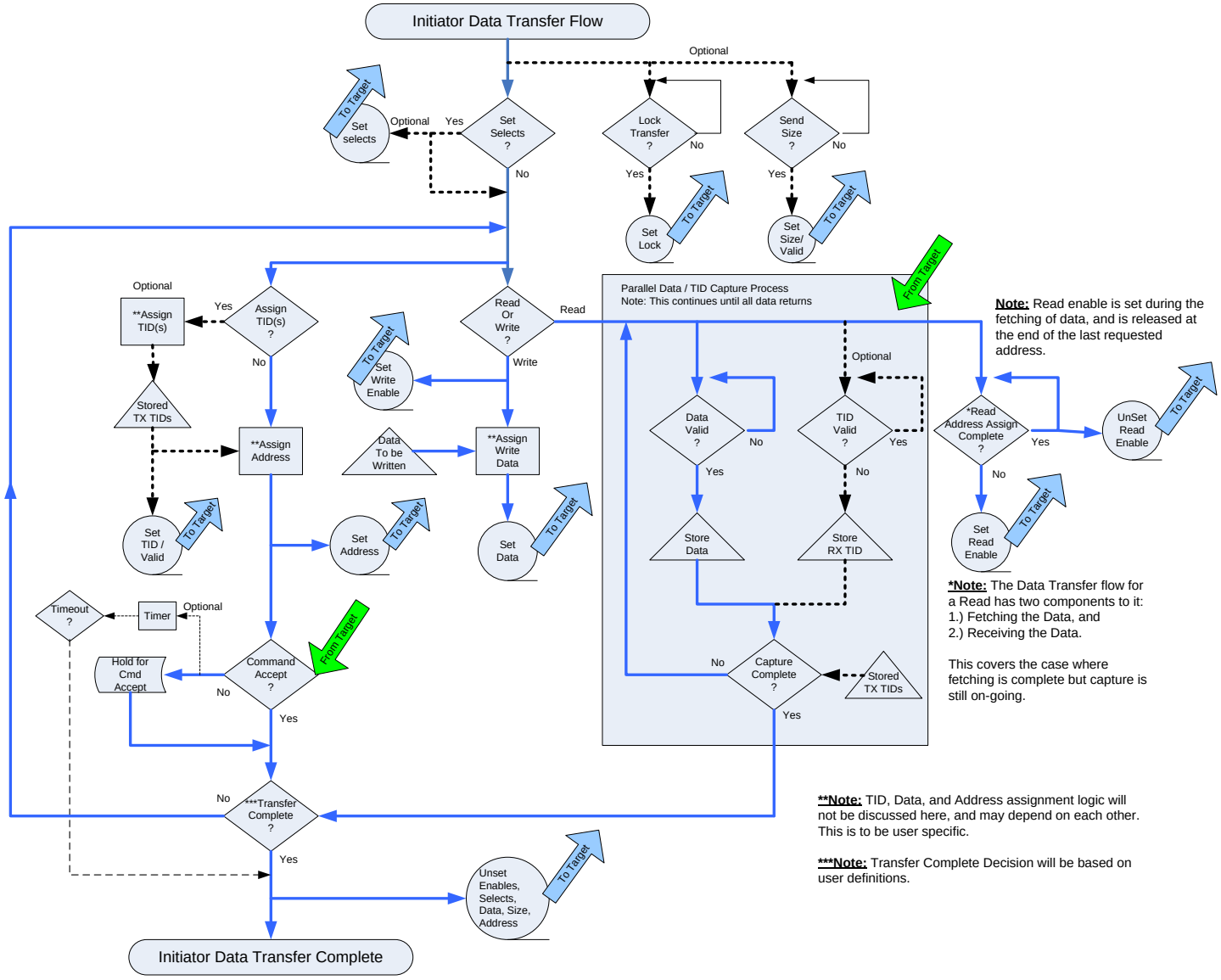

Figure 14 – Initiator Data and Control Transfer Flow Diagram

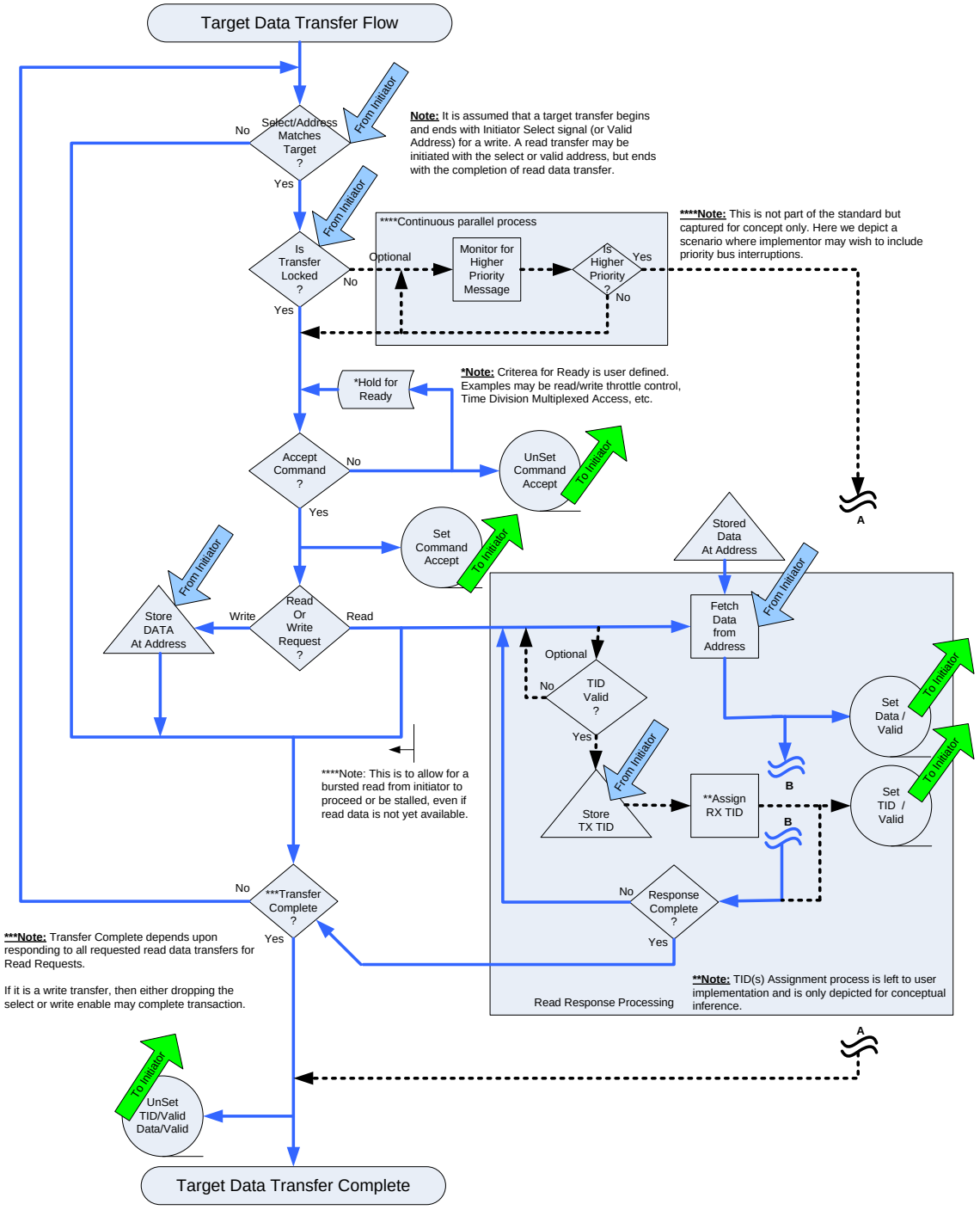

Figure 15 – Target Data and Control Transfer Flow Diagram

D.1.2.3 MOCB Configuration Package

MOCB compliant platforms and waveforms will each define and provide a package file, platform_pkg.vhd and waveform_pkg.vhd respectively, with MOCB defined constants. These are two separate package files. Each package file will define the appropriate values for the respective Targets and Initiators in the waveform and platform. The packages files consist of a common set of constants

used to customize the logic or reflect the configuration and requirements of the logic. The use of a common set of constants across all waveform and platform applications will aid in porting from platform to platform.

The determination of which are editable and which are not is left to the individual developers; waveform and platform respectively. The designation of each can be commented or segregated in the package file by the respective designers. In other words, the designer can create two sections in the package file. One section that clearly states the following constants are allowed to be edited as well as what the acceptable values are for each constant and another section clearly states the following are not to be edited.

D.1.2.3.1 MOCB Memory Map Configuration

The **c_<name>_pStartAddr** and the **c_<name>_pMemsize** constants identify the platform memory map space in bytes. The **c_<name>_wStartAddr** and the **c_<name>_wMemsize** constants identify the waveform memory map space in bytes. The memory allocation for the platform and waveform within a single FPGA consist of a single continuous block of memory for the platform as well as the waveform. Waveform memory allocations will not be interleaved with the platform memory allocation. Platforms having multiple FPGA resources may choose to organize memory allocations based on physical boundaries. For platforms allocating memory blocks based on physical boundaries, a starting address and memsize constant should be declared for each memory space.

For platform with multiple FPGA resources, the memory mapping may be organized as a single continuous block of memory for the platform and the waveform or each FPGA may be considered to have a unique platform and waveform memory map allocation. This is to allow waveform and platform memory allocations to be organized based on functional divisions (platform or waveform) or physical resource boundaries. Within each physical FPGA boundary, the waveform memory allocations will not be interleaved with the platform memory allocation. Examples are provided in **Figure 16** and **Figure 17**.

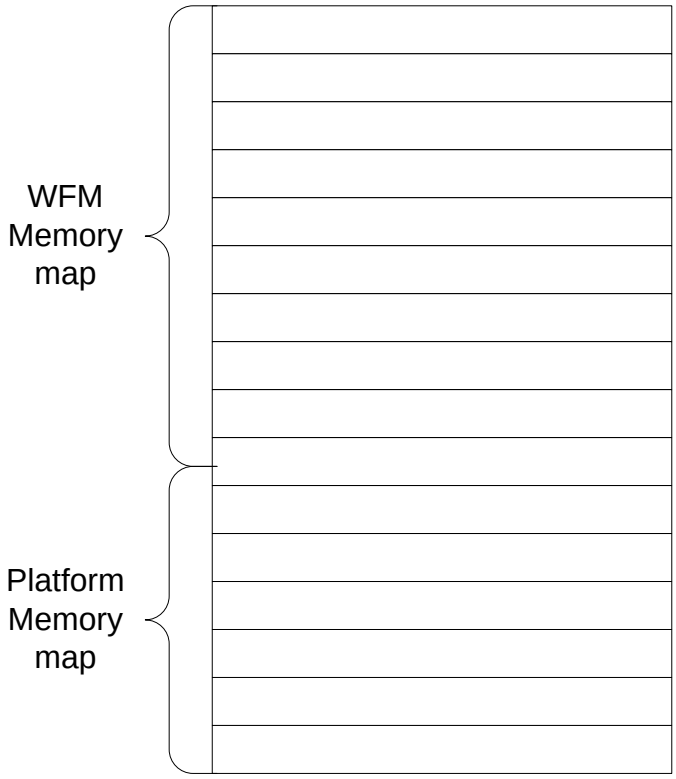

Figure 16 – Single FPGA Memory Map

Figure 16 illustrates the memory organization of a single FPGA. The platform and waveform are both allocated a continuous block of memory.

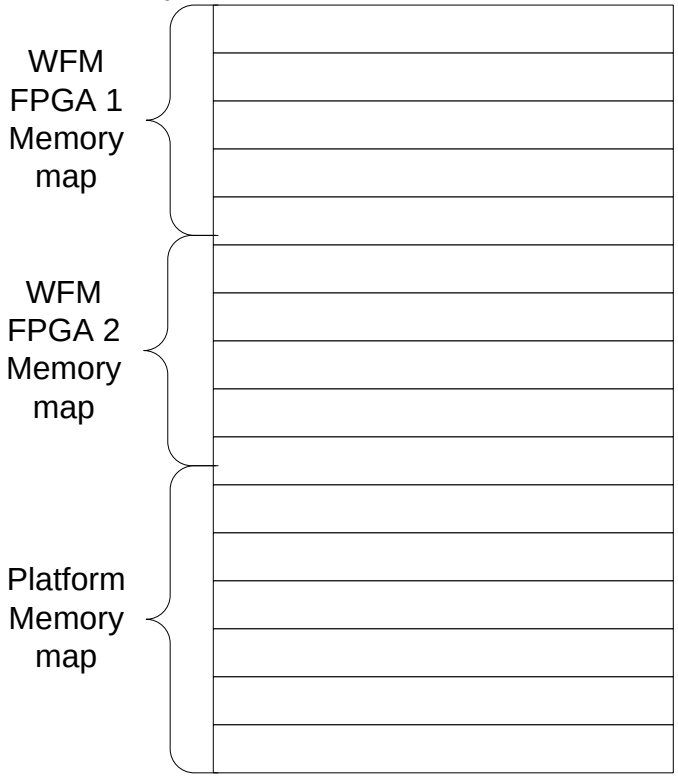

Figure 17 – Multiple FPGA Single Platform Memory Allocation

Figure 17 illustrates a possible memory organization of a two FPGA platform. The platform and waveform are both allocated a continuous block of memory. The waveform memory is divided into two continuous blocks. One block is reserved for waveform space in each FPGA. The waveform memory map is not interleaved between physical FPGA resources. Additionally, the platform memory map is not interleaved in the waveform memory map. This allows the platform to perform only course memory map decode. The arrangement may be useful for HW configurations that have a processor interface provided to one FPGA and an interconnect bus provided between FPGAs. Waveform starting address and sizes would need to be specified for each memory block in the package files.

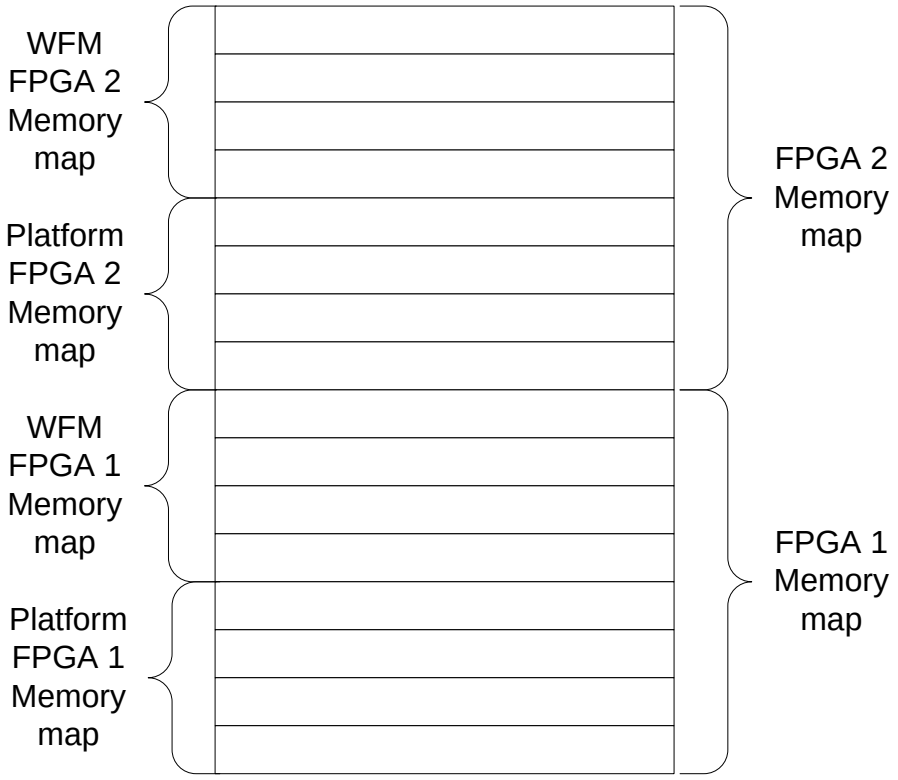

Figure 18 – Multiple FPGA Split Platform Memory Allocation

Figure 18 illustrates a possible memory organization of a two FPGA platform. The platform and waveform for each FPGA are organized into continuous blocks of memory. Neither the waveform nor the platform memory map is interleaved between physical FPGA resources. Additionally, the waveform and platform memory map are not interleaved within each physical FPGA. This allows the platform to perform only course memory map decode. The arrangement may be useful for HW configurations that have a processor interface(s) dedicated to each FPGA resource. All transactions for a particular FPGA resource can be routed or selected by the processor based on physical boundaries. Platform and waveform starting address and sizes would need to be specified for each memory block in the package files.

Platform Package Constant

c_<name>_pStartAddr	Identifies the starting address of the platform memory map starting address
c_<name>_pMemsize	Identifies the size of the platform memory space in bytes

Waveform Package Constant

c_<name>_wStartAddr (natural)	Identifies the starting address of the each waveform memory map starting address
c_<name>_wMemsize (natural)	Identifies the size of the waveform memory space in bytes for each memory allocation

D.1.2.3.2 Initiator Configurations

The Initiator configuration consists of a set of constants used to convey the physical configuration, the functional capabilities, and the requirements of an Initiator. The platform and waveform package files would need to create a set of constants for each Initiator it supplies.

D.1.2.3.2.1 Bus Configuration

Each package file will provide bus configuration for each Initiator provided. The configuration values will be used to configure the logic if allowed or to build a transition layer for porting between Initiators and Targets with incompatible configurations.

D.1.2.3.2.1.1 Initiator Event Bus Configuration

The Initiator event constant is used to specify number of event signals the Initiator can receive. If the Target requires more event lines than the Initiator provides, a translation layer would need to be constructed to provide the number of event lines required by the Target.

c_<name>_iEvents (natural)	Number of Initiator accepted Event Lines

D.1.2.3.2.2 Clock Crossing Configuration

Each package file will provide clock crossing configuration information for each Initiator provided. The configuration values will be used to configure the logic, if allowed, or to build a transition layer for porting between Initiators and Targets with incompatible configurations.

D.1.2.3.2.2.1 Clock Crossing provided

The Initiator clock crossing constant is used to specify if the Initiator provides clock crossing logic to match between Initiator and Target. This would be necessary if the Initiator was designed to operate with an interface clock and the Target is required to operate off a separate waveform clock. To aid in porting, the Initiator may choose to implement a clock crossing for the waveform. If the Initiator does not provide clock crossing, the constant could be used to provide the platform configuration to a configurable Target or a translation layer to provide the crossing.

c_<name>_iClkCross (boolean)	Initiator Clk Crossing Provided

D.1.2.3.2.2.2 Clock Crossing Elastic Buffer

A common source clock crossing technique is to utilize an elastic buffer. If the Initiator does provide an elastic buffer clock crossing, the elastic buffer depth may be used to specify the depth of the FIFO used for the elastic buffer. The depth may need to change from waveform to waveform port to accommodate varying clock frequencies between the Initiator and the Target. If the Initiator does not provide clock crossing logic, the constant is not applicable to the Initiator configuration and maybe set to zero.

c_<name>_iElasticDepth (natural)	Depth of elastic buffer

D.1.2.3.2.2.3 Initiator MOCB Clock Period

The Initiator MOCB clock Period constant is used to specify the period the Initiator operates in picoseconds. This information may be used to calculate the required buffer size when crossing domains with an elastic buffer.

iMClkPeriod_num and iMClkPeriod_dem specify the resolution of the MOCB clock period in picoseconds. The definition involves specifying two separate values, the numerator and the denominator. The result of dividing the numerator by the denominator should correspond to the MOCB clock period in picoseconds. For example, a numerator of 1,000,000 and a denominator of 60 define a 60 MHz MOCB Clock with a period of 16,666.666…. picoseconds. Expressing the resolution as a fraction minimizes clock rounding errors for frequencies that cannot be expressed as a simple integer.

c_<name>_iMClkPeriod_num (natural) Initiator MOCB CLK Period numerator in picoseconds
c_<name>_iMClkPeriod_den (natural) Initiator MOCB CLK Period denominator in picoseconds

Note: Most PLL's use the numerator and denominator values as the way to define the clock being generated.

D.1.2.3.2.3 Initiator Supported Extended Feature

The Initiator supported extended features constants are provided to identity what feature the Initiator requires or supports.

D.1.2.3.2.3.1 Initiator Transaction IDs Support

The MOCB extended interface bus protocol provides the designer with the ability to "TAG" read transfers with a unique transfer identification number (Tid). The intent of this feature is to provide the application the ability to support out of order transfers. Transaction IDs are used by the waveform on platforms that cannot guarantee data is returned in the order it was requested. Waveforms initially developed for platforms that could not guarantee the order of returned data may require the TIDS functionality even on platforms that can guarantee the order. The Initiator TIDs support constant indicates if the Initiator requires transaction ID support. If the Initiator requires TIDS support and the Target do not provide it, a translation layer would need to be added to loop back the TIDS to the Initiator. The TID support constant could be used by the Target or a translation layer to configure the logic to provide the Initiator the needed TIDs support.

c_<name>_iTids (boolean) Initiator Requires TIDS Support

D.1.2.3.2.3.2 Initiator Maximum Transaction Transfer Depth

The Initiator max transfer depth constant conveys to the Target or the translation logic the maximum length of transfers. The depth may be required if the Target does not support TIDS and needs to loop the TID number back to the Target upon a data return. It may also be used if the Initiator does not support the size field required by the target. The translation logic may use the Max transfer depth to size the buffer required to determine the size of individual transfers.

c_<name>_iXferMax (natural) Maximum Length of Transfers

D.1.2.3.2.3.3 Initiator Transfer Size

The MOCB extended interface bus protocol provides the designer with the option to utilize the SIZE field of the interface. The intent of this field is to advertise the size of the transfer in advance to allow a DMA to be set up to handle the transfer or to form a data packet header for a push packet interface. This reduces the need for the Target to buffer the entire transfer in order to determine the size.
The Initiator size field indicates if the Initiator supports the Size feature of the standard. If the Initiator supports the size, but the Target does not require it, no translation is necessary.

c_<name>_iSize (boolean) Initiator Provides Size field

D.1.2.3.2.3.4 Initiator Command Accept Support

The MOCB extended interface bus protocol provides the designer the capability of utilizing the Command Accept signal for both bus arbitration and data throttling (flow Control). The Target can use the signal to grant access to the bus as well as throttle the data transfer.
The "command accept" support constant indicates if the Initiator requires support of the command accept. If the Initiator requires the command accept function and the Target does not support it, the constant can be used by the Target or a translation layer to drive the signal to a constant logic '1'.

c_<name>_iCmdA (boolean) Initiator Requires Command accept

D.1.2.3.2.3.5 Initiator Data Accept Support

The MOCB extended interface bus protocol provides the designer the capability to throttle or hold off the return data from the Target by utilizing the Data Accept (DataA). The data accept is implemented to allow the designer to stall the return of data from the Target.
The Data accept support constant indicates if the Initiator requires support for the "Data Accept" feature. If the Initiator requires the "Data Accept" function and the Target do not support it, the constant can be used by the Target or a translation layer to buffer data transfers until the Initiator can accept them. The max transaction transfer depth could be used in conjunction to determine the size of the buffer.

c_<name>_iDataA (boolean) Initiator Utilizes Data Accept

D.1.2.3.2.3.6 Initiator Lock Support

The MOCB extended interface bus protocol provides the designer with the option to incorporate a LOCK feature. The Lock signal indicates to the Target arbiter that the Initiator needs to retain ownership of the bus during a transfer as well as between transfers.
The Data accept support constant indicates if the Initiator requires support of the transfer lock. If the Initiator requires the transfer lock function and the Target do not support it, translation layer logic may be needed to provide the lock function.

c_<name>_iLock (boolean) Initiator Provides Lock Support

D.1.2.3.3 Target Configuration

Each package file will provide bus configuration for each Target provided. The configuration values will be used to configure the logic, if allowed, or to build a transition layer for porting between Initiators and Targets with incompatible configurations.

D.1.2.3.3.1 Target Event Bus Configuration

The Target event constant is used to specify number of event signals the Target provides. If the initiator does not support the required number of event lines, a translation layer would need to be constructed to provide the number of event lines required by the Target.

c_<name>_tEvents (natural) Number of Target provided Event Lines

D.1.2.3.4 Target Clock Crossing Configuration

Each package file will provide clock crossing configuration information for each Target provided. The configuration values will be used to configure the logic, if allowed, or to build a transition layer for porting between Initiators and Targets with incompatible configurations.

D.1.2.3.4.1 Clock Crossing provided

The Target clock crossing constant is used to specify if the Target provides clock crossing logic to match between Initiator and Target. This would be necessary if the Initiator was designed to operate with an interface clock and the Target is required to operate off of a separate waveform clock. To aid in porting, the Target may choose to use implement a clock crossing for the waveform. If the Target does not provide clock crossing, the constant could be used to provide the target configuration to a configurable Initiator or a translation layer to provide the crossing.

c_<name>_tClkCross (boolean) Target Clk Crossing Provided

D.1.2.3.4.2 Target Clock Crossing Elastic Buffer

A common source clock crossing technique is to utilize an elastic buffer. If the Target does provide an elastic buffer clock crossing, the elastic buffer depth may be used to specify the depth of the FIFO used for the elastic buffer. The depth may need to change from platform to platform port to accommodate varying clock frequencies between the Initiator and the Target. If the Target does not provide clock crossing logic, the constant is not applicable to the Target configuration and maybe set to zero.

c_<name>_tElasticDepth (natural) Target Depth of elastic buffer depth

D.1.2.3.4.3 Target MOCB Clock Period

The Target MOCB clock Period constant is used to specify the period the Target operates in picoseconds. This information may be used to calculate the required buffer size when crossing domains with an elastic buffer.

tMClkPeriod_num and tMClkPeriod_dem specify the resolution of the MOCB clock period in picoseconds. The definition involves specifying two separate values, the numerator and the denominator. The result of dividing the numerator by the denominator should correspond to the MOCB clock period in picoseconds. For example, a numerator of 1,000,000 and a denominator of 60 define a 60 MHz MOCB Clock with a period of 16,666.666…. picoseconds. Expressing the resolution as a fraction minimizes clock rounding errors for frequencies that cannot be expressed as a simple integer.

c_<name>_tMClkPeriod_num (natural) Target MOCB CLK Period numerator in picoseconds
c_<name>_tMClkPeriod_den (natural) Target MOCB CLK Period denominator in picoseconds

D.1.2.3.5 Target Supported Extended Feature

The Target supported extended features constants are provided to identity what feature the Target requires or supports.

D.1.2.3.5.1 Target Transaction IDs Support

The MOCB extended interface bus protocol provides the designer with the ability to "TAG" read transfers with a unique transfer identification number (Tid). The intent of this feature is to provide the application the ability to support out of order transfers. Transaction IDs are used by the waveform on platforms that cannot guarantee data is returned in the order it was requested. Waveforms initially developed for platforms that could not guarantee the order of returned data may require the TIDS functionality even on platforms that can guarantee the order. The Target TIDs support constant indicates if the Target supports transaction ID's. If the Initiator requires TIDS support and the Target do not provide it, a translation layer would need to be added to loop back the TIDS to the Initiator. The TID support constant could be used by the Initiator or a translation layer to configure the logic to provide the Initiator the needed TIDs support.

c_<name>_tTids (boolean) Target Requires TIDS Support

D.1.2.3.5.2 Target Transfer Size

The MOCB extended interface bus protocol provides the designer with the option to utilize the SIZE field of the interface. The intent of this field is to advertise the size of the transfer in advance to allow a DMA to be set up to handle the transfer or to form a data packet header for a push packet interface. This reduces the need to buffer transfer to determine the size.
The Target size field indicates if the Target requires the Size feature of the standard. If the Initiator supports the size, but the Target does not require it, no translation is necessary. In cases where the target requires a "Size" field, such as abstracting a serial push packet interface and the Initiator does not support it, a translation layer would be required to buffer the transfer and determine the size. The **c_<name>_iXferMax** constant may be used to size the buffer.

c_<name>_tSize (boolean) Target utilizes Size field

D.1.2.3.5.3 Target Command Accept Support

The MOCB extended interface bus protocol provides the designer the capability of utilizing the Command Accept signal for both bus arbitration and data throttling (flow Control). The Target can use the signal to grant access to the bus as well as throttle the data transfer.
The "Command Accept" support constant indicates if the Target supports the "Command Accept". If the Initiator requires the command accept function and the Target does not support it, the constant can be used by the Target or a translation layer to drive the signal to a constant logic '1'.

c_<name>_tCmdA (boolean) Target Supports Command accept

D.1.2.3.5.4 Target Data Accept Support

The MOCB extended interface bus protocol provides the designer the capability to throttle or hold off the return data from the Target by utilizing the Data Accept (DataA). The data accept is implemented to allow the designer to stall the return of data from the Target.
The Data accept support constant indicates if the Target supports the data accept feature. If the Initiator requires the "data accept" function and the Target does not support it, the constant can be used by the

Initiator or a translation layer to buffer data transfers until the Initiator can accept them. The max transaction transfer depth could be used in conjunction to determine the size of the buffer.

c_<name>_tDataA (boolean) Target Provides Data Accept support

D.1.2.3.5.5 Target Lock Support

The MOCB extended interface bus protocol provides the designer with the option to incorporate a LOCK feature. The Lock signal indicates to the Target arbiter that the Initiator needs to retain ownership of the bus during a transfer as well as between transfers.
The Data accept support constant indicates if the Target provides support of the transfer lock. If the Initiator requires the transfer lock function and the Target do not support it, translation layer logic may be needed to provide the lock function.

constant c_<name>_tLock Initiator Provides Lock Support

D.1.2.3.5.6 Target Data Valid Latency

The Target Data Valid Latency constant is used to convey to the designers the typical latency associated with a particular interface. This may be required when interfacing to an external RAM where internal buffering or pre-fetching may be necessary if the latency is too high. The latency count would be based on the MOCB target CLK frequency.

c_<name>_tDataVLat Target Data Valid Cycle Latency

D.1.2.4 Translation Layer

When porting a waveform to an individual platform, there may be instances where the supported features or required features of a platform and waveform do not align. If the platform and waveform have not provided the needed functionality or configurability, a translation layer may be required to provide a bridge between the platform and waveform logic. The creation of the translation logic is considered part of the porting exercise. The translation layer may also be provided by the platform for the waveform development team to aid in future porting, but it is not required.

Figure 19 illustrates an example of a translation layer created between a platform MOCB interface and a waveform MOCB interface.

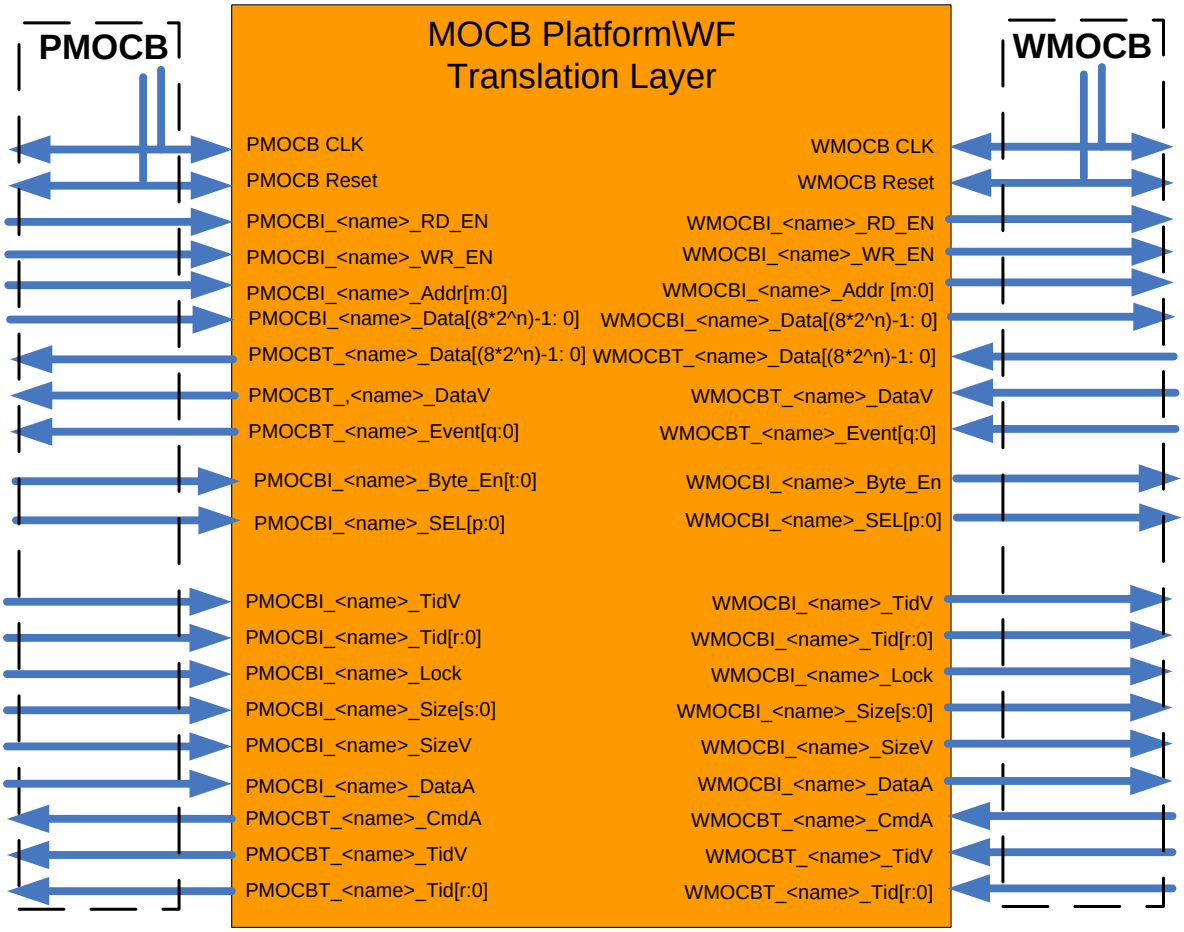

Figure 19 – Example MOCB Interconnect Translation Layer

D.1.2.5 MOCB FPGA Timing

The timing diagrams are intended to show the relationship between various signals and features. They are not intended to encompass every possible bus scenario or configuration. The MOCB implementation is not limited to those illustrated in this document. The MOCB protocol is limited by the signals relationships spelled out in this document and timing diagrams.

D.1.2.5.1 Basic burst Write, no flow control, No Tids, No Lock, No Size

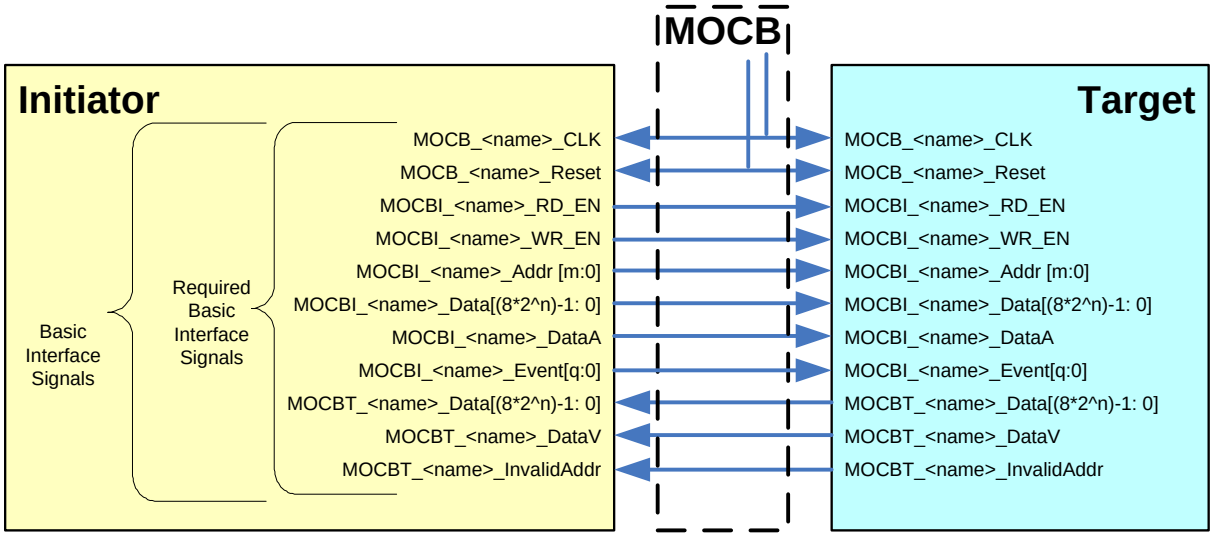

Figure 20 – Basic Burst Write

The timing diagram in **Figure 21** illustrates a simple Burst Write transfer from an Initiator block to a Target block.

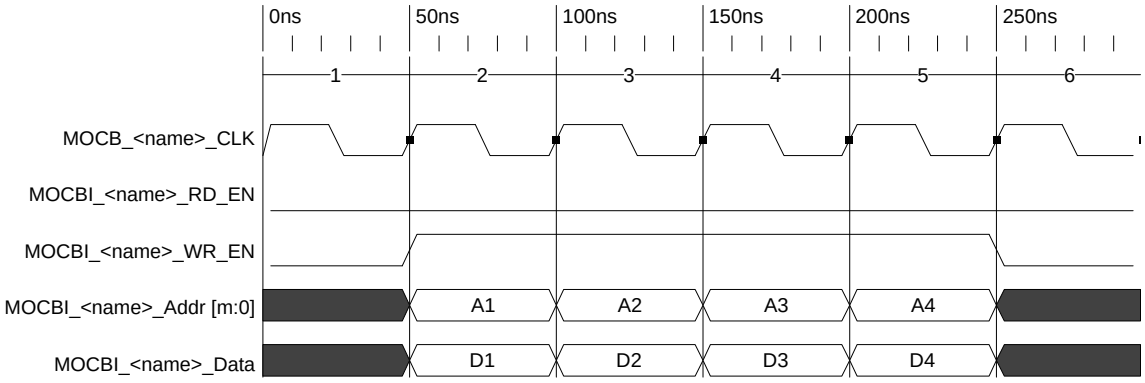

Figure 21 – Basic Burst Write

Sequences

1) On cycle 2, the Initiator starts the transfer by setting the write enable (WR_EN) and presenting valid address (A1) and data (D1). Since this implementation does not incorporate any flow control handshaking, the Initiator can assume the Address, data and control signals are captured on the next rising edge clock.
2) The Target will capture the address, data and control signals on each clock cycle and use them internally to perform the write.
3) The Initiator presents a new transfer on each subsequent clock cycle until the transfer is complete.

D.1.2.5.2 Basic burst Write with Size, no flow control, No Tids, No Lock

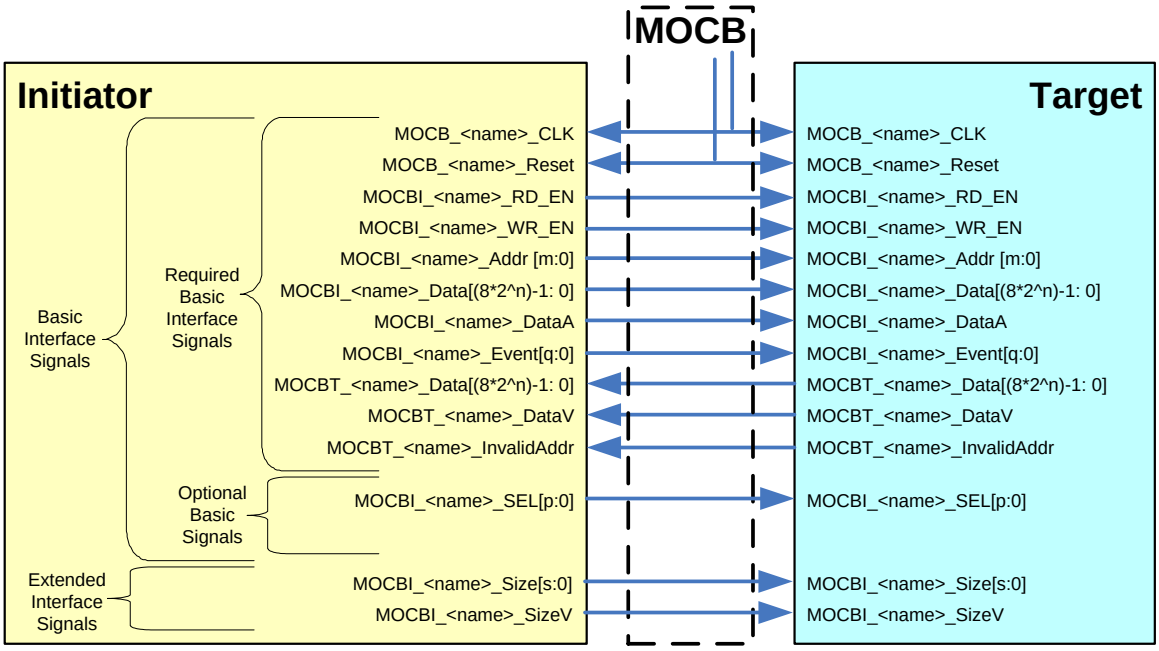

Figure 22 – Basic Burst Write w/Size

The timing diagram in **Figure 23** illustrates a simple Burst Write transfer from an Initiator block to a Target block using the transfer size field (Size/SizeV). The intent of this field is to advertise the size of the transfer in advance to allow a DMA to be set up to handle the transfer or to form a data packet for a push packet interface. This reduces the need to buffer transfer to determine the size.

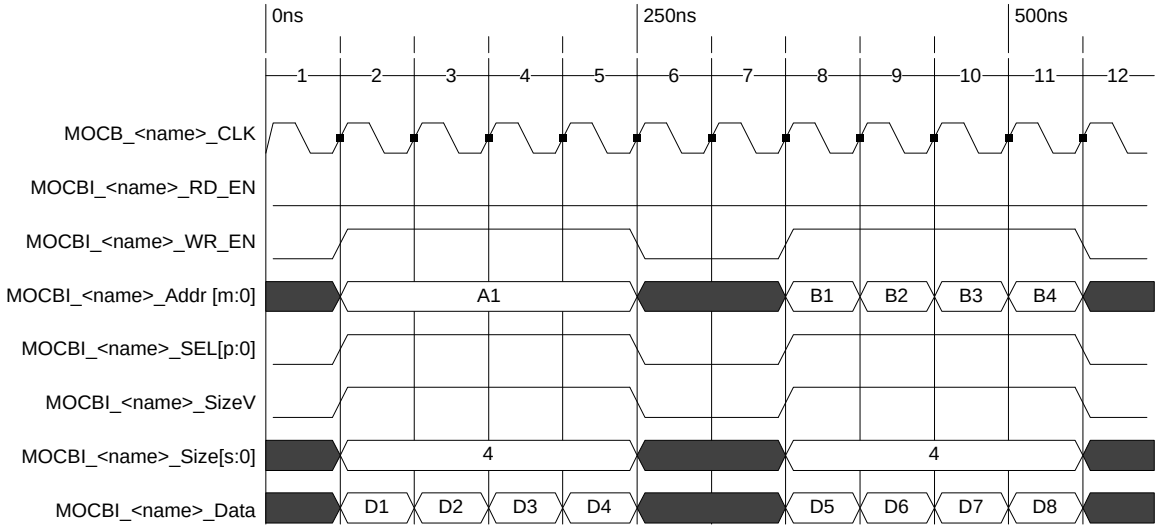

Figure 23 – Basic Burst Write w/Size

Sequences

1) On cycle 2, the Initiator starts the transfer by setting the write enable (WR_EN) and presenting valid address (A1), data (D1), and transfer size (Size). Since this implementation does not

incorporate any access control handshaking, the Initiator can assume the Address, data and control signals are captured on the next rising edge clock.

2) The Target will capture the address, data and control signals on each clock cycle and use them internally to perform the write. The Target may use the Size field to form a packet header or allocate space in memory
3) The Initiator presents a new command on each subsequent clock cycle until the entire transfer is complete. For this example, the Initiator is writing to single address 4 times. This could represent a write to a FIFO or a packet based interfaces memory mapped to Address A1.
4) On cycle 8, the Initiator begins a second burst transfer. The Initiator starts the transfer by setting the write enable (WR_EN) and presenting valid address (A1), data (D1), and transfer size (Size).
5) The Target will capture the address, data and control signals and use them internally to perform the write.
6) The Initiator presents a new transfer on each subsequent clock cycle until the transfer is complete. For this example, the Initiator is writing to a different address on each cycle.

D.1.2.5.3 Basic Read with no flow control, No Tids, No Lock

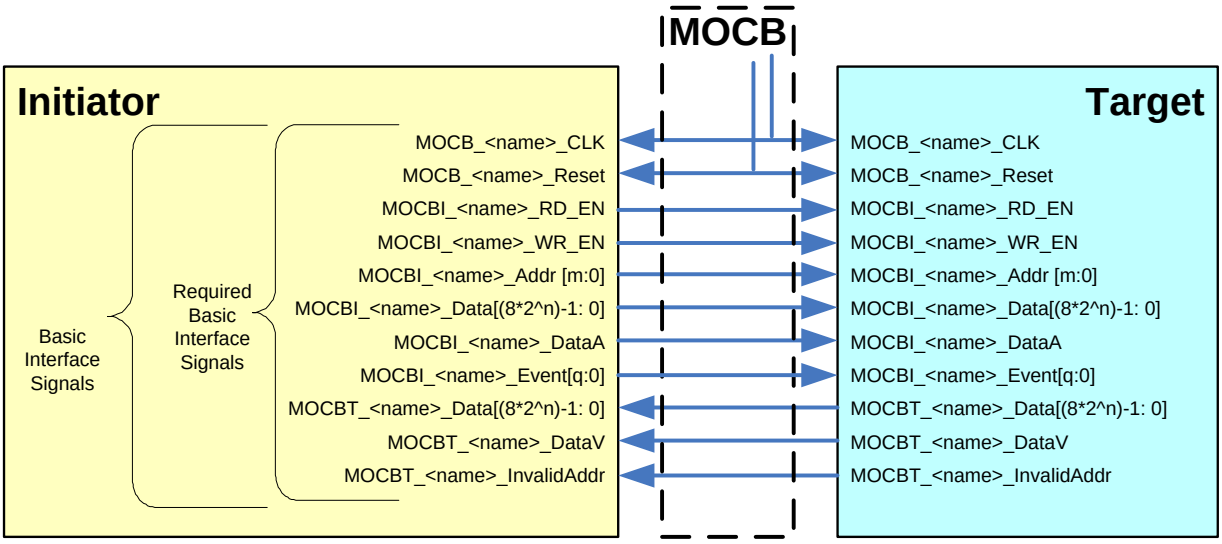

Figure 24 – Basic Read

The timing diagram in **Figure 25** is a simple Read transfer from an Initiator block to a Target block. This example illustrates a pipelined burst read with delayed data return. It is meant to illustrate a pipelined read with two different pipeline depths. The Data D3 and D4 take an extra cycle to return. The standard handles this with the use of a data valid line to indicate when data is valid on the bus. It is assumed in the example that data is being returned in order from a single Target and no backpressure is needed to hold off the data return. The target would be responsible for avoiding internal collisions and ordering the returned data. The data accept signal is driven active though the entire transfer. This is common for initiators with no requirements to throttle the return of data from the target. For implementation where there are no requirements to throttle the flow of data from the target to the initiator, the DataA signals is driven to a logical active state (‘1’) continuously.

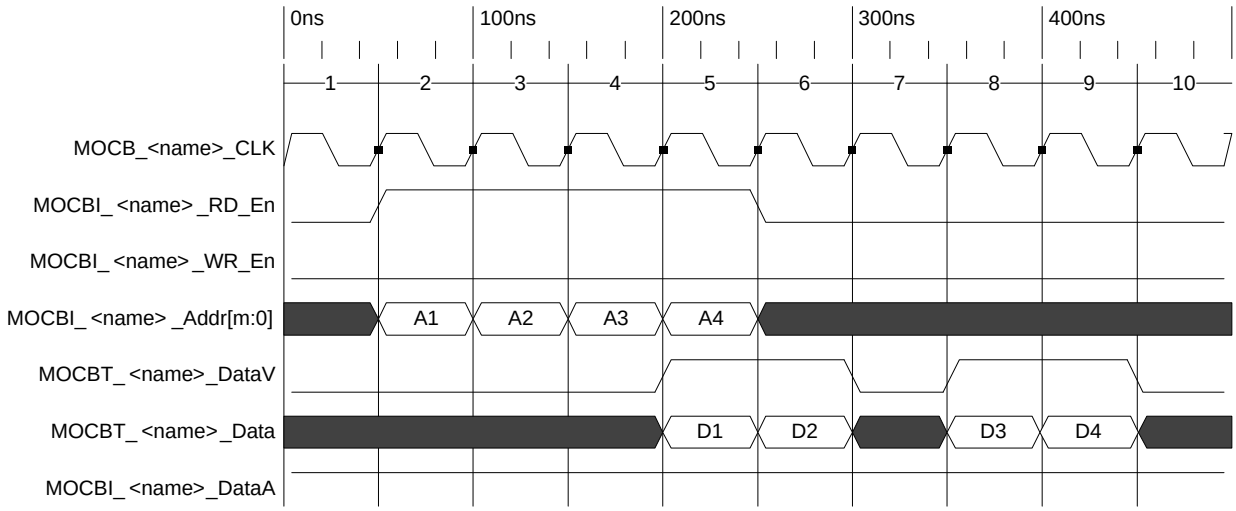

Figure 25 – Basic Read

Sequences

1) On cycle 2, the Initiator starts the transfer by setting the read enable (RD_EN) and presenting valid address (A1), Since this implementation does not incorporate any flow control handshaking, the Initiator can assume the address and control signals are captured on the next rising edge clock.
2) The Target will capture the address and control signals on each clock and use them internally to perform the read.
3) The Initiator presents a new transfer on each subsequent clock cycle until the transfer is complete.
4) On cycle 5, the Target begins to return the data from the read. The data is returned in the order it was requested. The data is qualified valid by the data valid line (DataV).
5) The Data accept (DataA) is driven high through the entire transfer indicating the Initiator is capable of receiving data presented by the Target.
6) On cycle 7, the Target invalidates the DataV signals indicating to the Initiator there is no valid Data on the bus.
7) On Cycle 8, the Target resumes transfer of the requested read data by presenting the data and data valid.

D.1.2.5.4 Basic Read with Return Data Flow Control, No Tids, No Lock

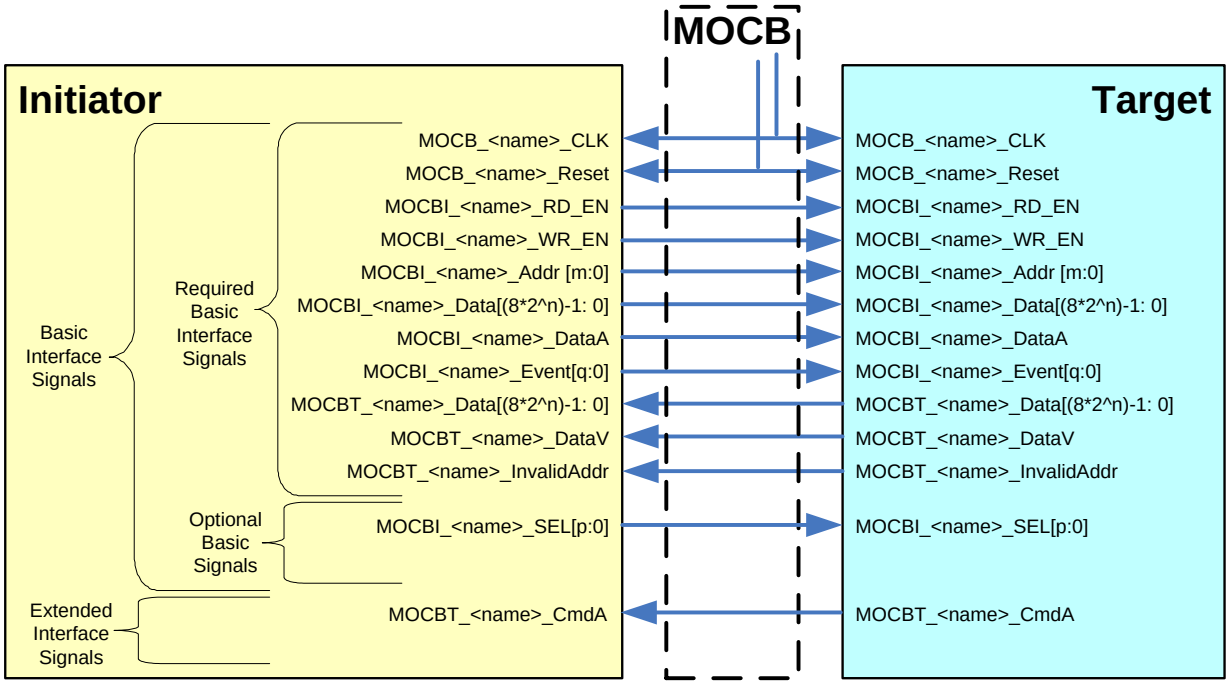

Figure 26 – Basic Read with Data Accept

The timing diagram in **Figure 27** is a basic Read transfer from a single Initiator block to two different Target blocks. The standard handles this with the use of a data valid (DataV), data accept (DataA), and command accept (CmdA) signals to indicate when data is valid on the target data bus, data has been accepted by the Initiator, and a command can be accepted by the Target. The data return accept function must be used in conjunction with the command accept function. When the initiator stalls the return data pipeline, the target must stall the command pipeline to keep from over running the target with commands it cannot service. The intent is that the target merely stalls its command and data pipeline rather than be forced to store commands internally until the data can be returned to the initiator. The pipeline itself becomes the storage elements and can be used to store in flight commands. This example illustrates a scenario where the target blocks have different pipelined depths resulting in a data return collision. The Data D1, D2 and D3 from Target 1 take an extra cycle to return. When a data return is stalled, the data from target 2 and the command from the initiator for target 2 are stalled until target 1 has completed its data return and released its data valid. Data from Target 2 is then accepted by the initiator by driving the data accept for target 2 active. The target then releases the command accept allowing the transfer to complete. In this example, it is assumed that the data is returned in order eliminating the need to incorporated Tids.

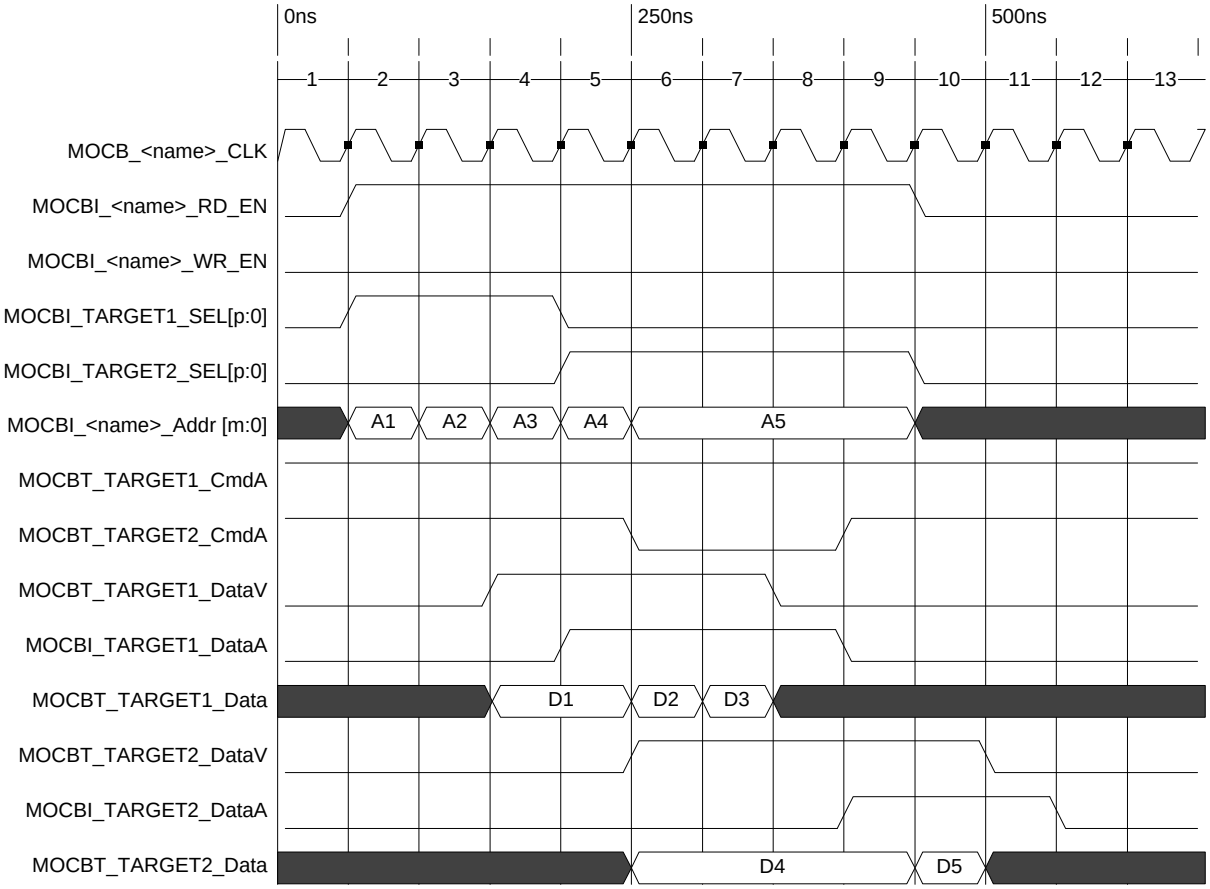

Figure 27 – Basic Read with Return Data Flow Control

Sequences

1) On cycle 2, the Initiator starts the transfer to target 1 by setting the read enable (RD_EN) and presenting valid address (A1), and the Address select for target1. Since this implementation utilizes the command accept for flow control and does not incorporate any arbitration handshaking, the address and control signals are captured on the next rising edge clock.
2) The Initiator presents a new transfer on each subsequent clock cycle until the transfer to target 1 is complete.
3) On cycle 5, the Initiator starts the transfer to target 2 by setting the read enable (RD_EN) and presenting valid address (A4), and the Address select for target 2. Since this implementation utilizes the command Accept for flow control and does not incorporate any arbitration handshaking, the address and control signals are captured on the next rising edge clock.
4) The Initiator presents a new transfer on each subsequent clock cycle until the transfer is complete.
5) The Targets will capture the address and control signals on each clock and use them internally to perform the read.
6) On cycle 4, the Target 1 presents return data from the read command. The data is returned in the order it was requested. The data is qualified valid by the data valid line (DataV). The data is held until the data accept line is active indicating that data has been captured from the bus.
7) On cycle 5, the data accept signal for target 1is set indicating to the target that the data has been captured.
8) On cycle 6, the Target 2 presents return the data from the read. The data is qualified valid by the data valid line (DataV). The data is held on the bus because the data accept (DataA) for

target 2 is inactive indicating to the Target that the data was not captured by the initiator. Data will be held until the initiator drives the data accept signal. The target must stall the return and command pipeline.

9) Cycle 6, the Target stalls the Initiator commands by driving the command accept inactive. This effectively stalls the entire round trip pipeline until the initiator is able to accept the pending return data.
10) On cycle 8, target 1 releases its data valid indicating that there is no valid data on the bus to capture.
11) Cycle 9, the initiator release the data accept for target 1 and sets the data accept for target 2 indicating that the pending data for target 2 has been accepted.
12) Cycle 9, the target will drive the command accept active indicating that the current command has been captured.
13) On cycle 11, target 2 releases its data valid indicating that its data return transfer is complete.

D.1.2.5.5 Write Command with Command Flow control, No Tids, No Lock

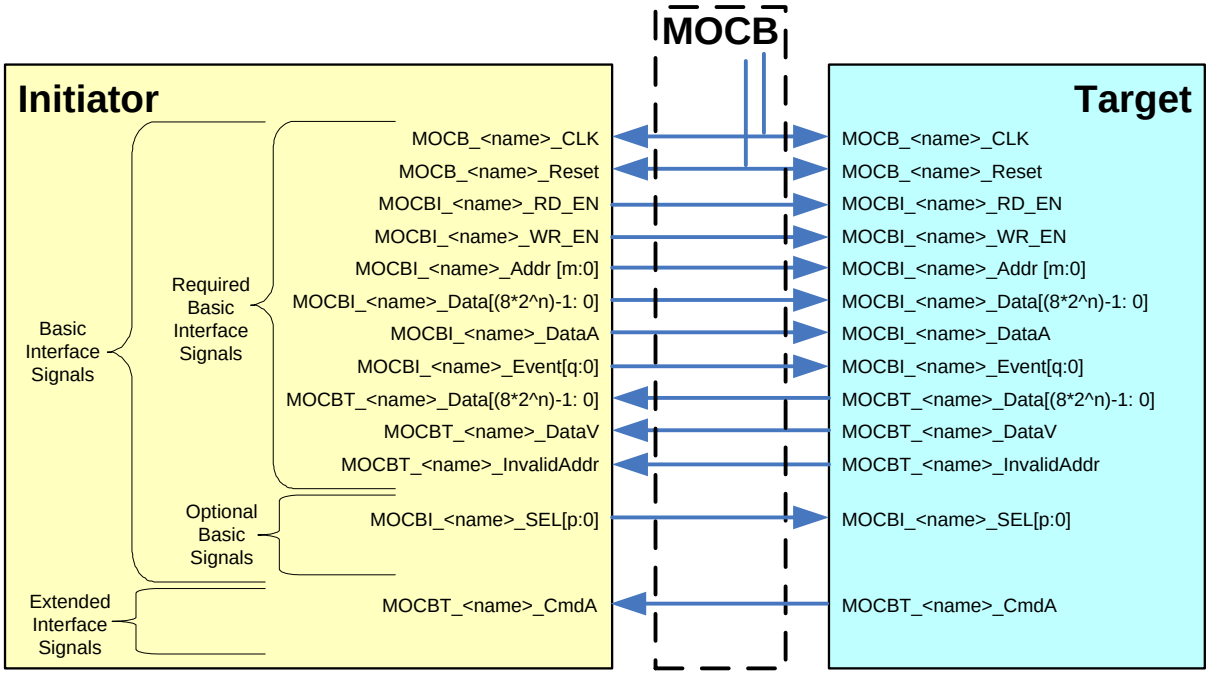

Figure 28 – Write Command with Command Flow Control

The timing diagram in **Figure 28** illustrates a Write transfer from an Initiator block to a Target block utilizing Command only flow control. In this example, the Target is using the command flow control signal (CmdA) to throttle the data transfer. This example is only throttling data from the Initiator and the Target. This scenario may be common when attaching to a push packet interface. The intent of this feature in this example is to throttle the flow of data between the core and an interface of lower bandwidth. This feature is also intended to provide the capability of allowing an Initiator to request access to bus and allow the Target, with support of an arbiter, to grant access. That feature is illustrated in later examples.

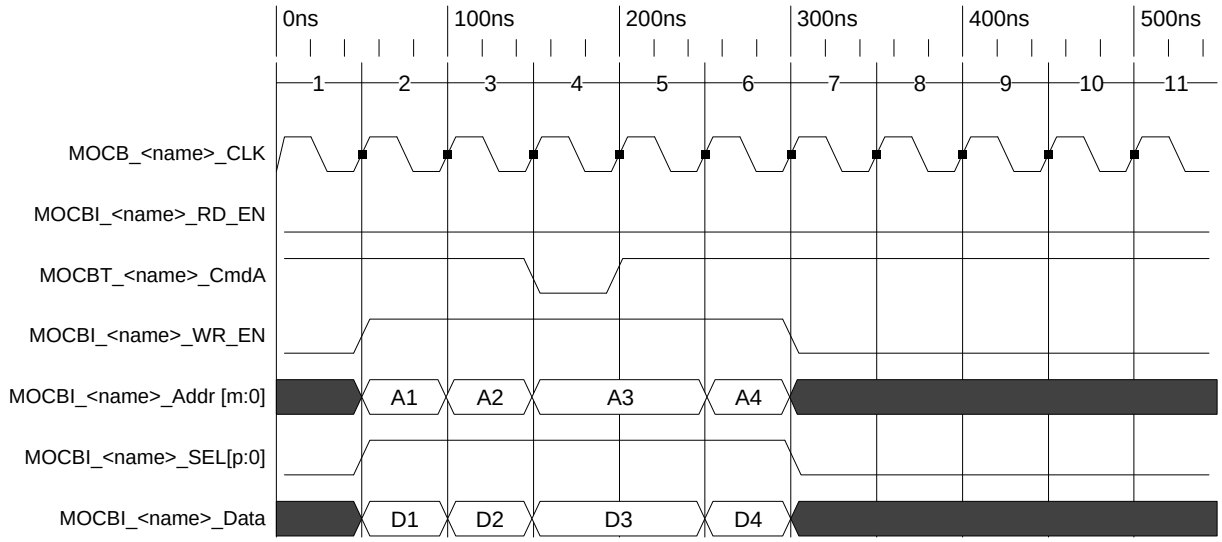

Figure 29 – Write Command with Flow Control

Sequences

1) On cycle 2, the Initiator starts the transfer by setting the write enable (WR_EN) and presenting valid address (A1) and address select (SEL). Since this implementation utilizes the command flow control handshaking for data throttling only (as indicated by the CmdA line being active before the write enable is set), the Initiator can assume the address and control signals are captured on the next rising edge clock.
2) The Target will capture the address, data, and control signals on each clock cycle and use them internally to perform the write.
3) The Initiator presents a new transfer on each subsequent clock cycle until the transfer is complete
4) On cycle 4, the Target drives the command accept (CmdA) inactive indicating that it did not accept the current transfer on the bus. The Initiator holds the current command until the Target accepts the command.
5) Cycle 5 the Target drives the CmdA active indicating the current command has been accepted.
6) On Cycle 7, the Initiator releases the valid write enable indicating that the current transfer is complete.

D.1.2.5.6 Read /Write Command with Access control and Lock, No Tids

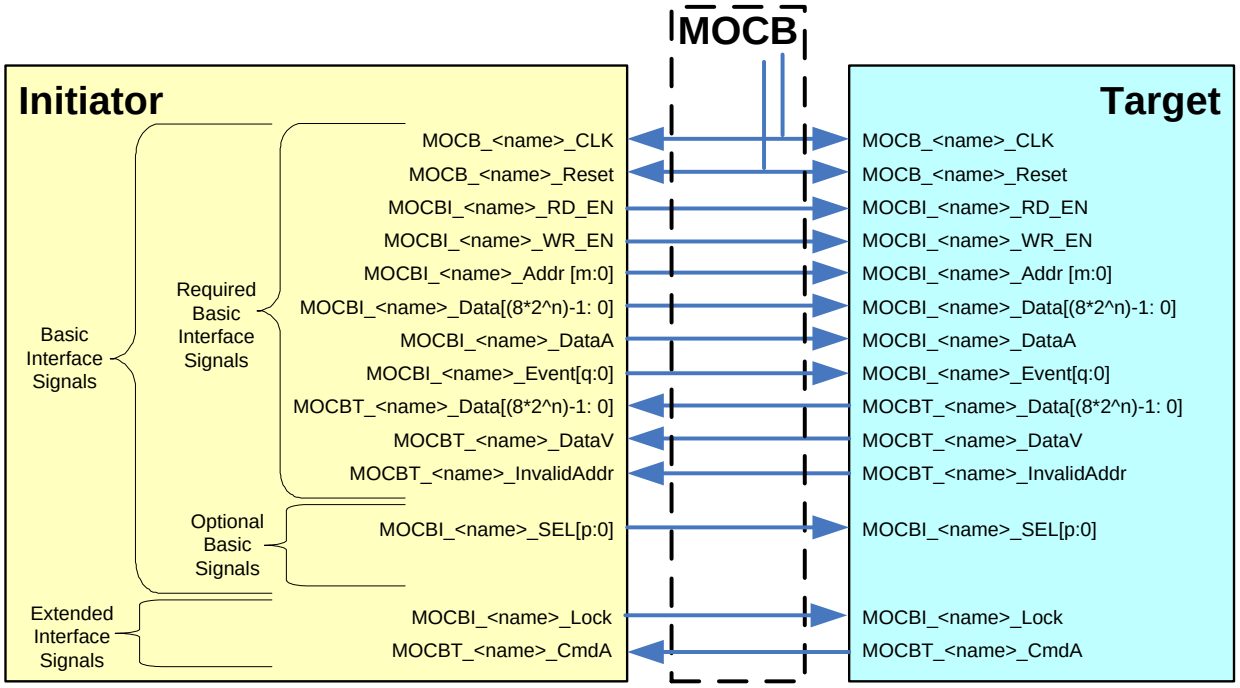

Figure 30 – Read/Write with Access Control and Lock

The timing diagram in **Figure 31** illustrates a Read transfer followed by a Write from an Initiator block to a single Target block utilizing Access control. In this example, the Target is using the Access control signal (CmdA) to grant access only. The intent of this feature is to provide the capability of allowing an Initiator to request access to a bus and allow the Target, with support of an arbiter, to grant access. The command accept is also utilized to throttle the flow of data across target interfaces with different bandwidths as illustrated in the flow control examples. This implementation does not implement flow control and therefore does not implement the data accept signal. This example does utilize a Lock signal to indicate to the Target arbiter that the Initiator needs to retain ownership of the bus during a transfer as well as between transfers. This prevents the Target from granting access to a higher priority transfer during the current transfer or from giving ownership of the bus to another requester between transfers. In the example the Initiator performs 2 generic reads (A1, A2) followed by a read modify write (A3, A4). The initiator sets the lock at the beginning of A3. The arbiter does not accept the A3 transfer and grants access to a higher priority transfer before returning access to the Initiator. Since the transfer had not begun when the lock was set, the Target was free to remove access even though the lock was set. Once the first transfer is accepted from the initiator when the lock is set, the Target is prohibited from removing access and granting access to a higher priority requester. The data accept signal is driven active though the entire transfer. This is common for initiators with no requirements to throttle the return of data from the target. For implementation where there are no requirements to throttle the flow of data from the target to the initiator, the DataA signals is driven to a logical active state (‘1’) continuously.

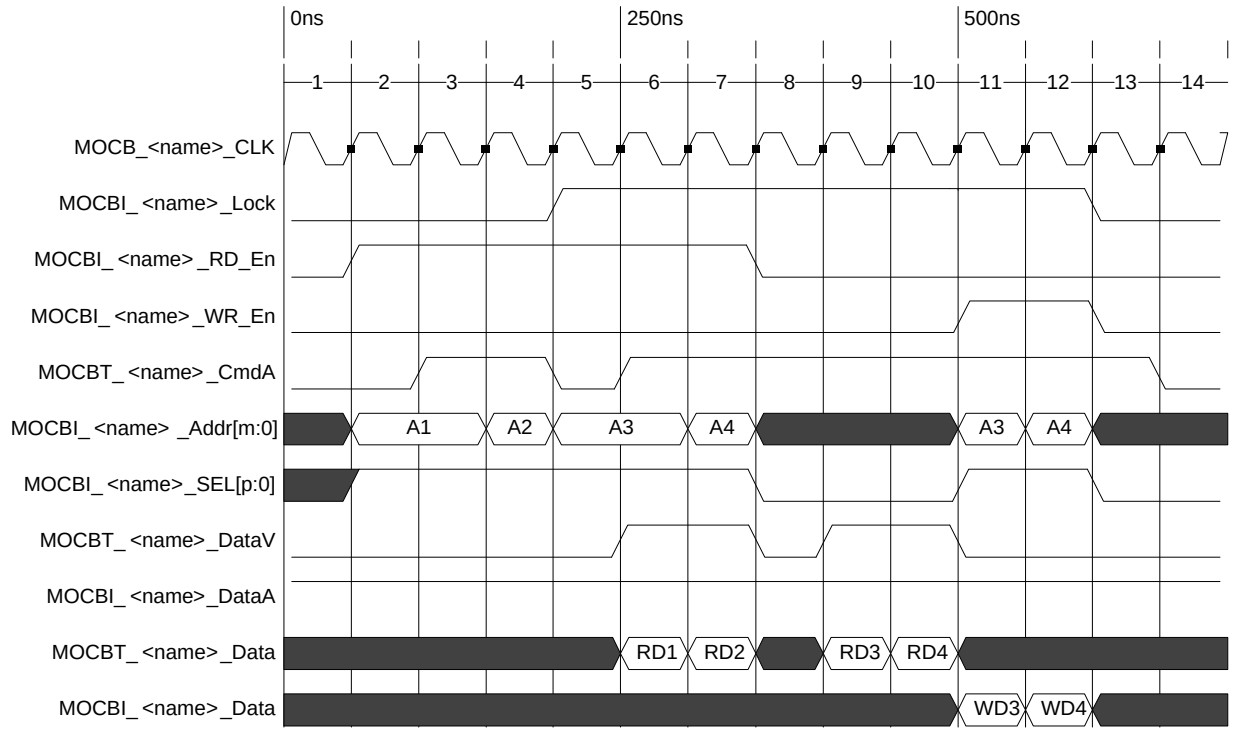

Figure 31 – Read/Write with Access Control and Lock

Sequences

1) On cycle 2, the Initiator requests a transfer by setting the read enable (RD_EN), valid address (A1), and address select (SEL). These signals are held active until the Target indicates the command has been accepted
2) On cycle 3, the Target signals the Initiator that the command has been accepted driving the command accept signal (CmdA) valid. The Target will capture the address and control signals on each clock cycle and use them internally to perform the read.
3) The Initiator presents a new transfer on each subsequent clock cycle until the transfer in complete
4) On cycle 5, the Target drives the command accept (CmdA) inactive indicating that it did not accept the current transfer on the bus. The Initiator holds the current command until the Target accepts the command.
5) On cycle 5, the Initiator requests a lock transfer by setting the “LOCK” signal. By setting the LOCK signal, the Initiator is signaling the Target that this will be a locked transfer. This indicates to the Target that the initiator requires maintained ownership of the bus until the lock is released.
6) Cycle 6 the Target drives the CmdA active indicating the current command has been accepted.
7) On cycle 6, the Target begins to return the data from the read. The data is returned in the order it was requested. The data is qualified valid by the data valid line (DataV).
8) The Data accept (DataA) is driven high through the entire transfer indicating the Initiator is capable of receiving data presented by the Target.
9) On cycle 8, the Target invalidates the DataV signals indicating to the Initiator there is no valid Data on the bus.
10) On Cycle 9, the Target resumes transfer of the requested read data by presenting the data and data valid

11) The Initiator completes first transfer block on Cycle 8 and the drives the read enable and write enable signals invalid, but maintains the Lock signal. This indicates to the Target that the Initiator has not completed all intended transfers and requires continued ownership of the bus. The Target maintains the CmdA active indicating that bus ownership is granted and more transfers can be accepted.
12) On cycle 11, the Initiator starts the Write transfer by setting the write enable (WR_EN) and presenting valid address (A3), address select (SEL) and data (WD3). Since the CmdA line is already active, the Initiator can assume the Address, data and control signals are captured on the next rising edge.
13) The Target will capture the address, data and control signals and use them internally to perform the write.
14) The Initiator presents a new transfer on each subsequent clock cycle until the transfer is complete.
15) The Target will capture the transfer on each clock cycle and perform the write.
16) Cycle 13, the transfer is complete and the Initiator releases the write enable and Lock, indicating to the Target that it is done with all current transfers and the bus can be granted to another user.
17) Cycle 14, the Target removes the CmdA line indicating that the Initiator no longer has ownership of the bus.

D.1.2.5.7 Write/Read Command with Access control and Tids, No Lock

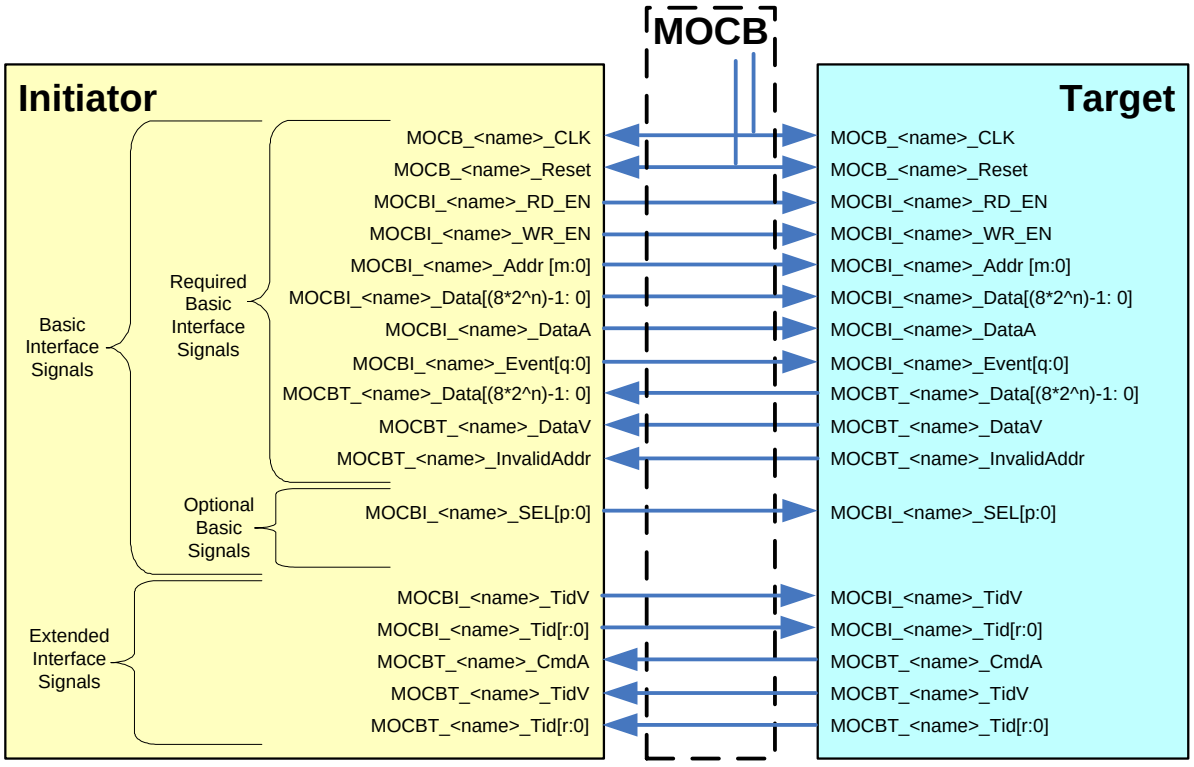

Figure 32 – Write/Read with Access Control and Tid

The timing diagram in **Figure 33** illustrates a Write transfer followed by a Read from an Initiator block to a Target block utilizing Access control and transaction Identification (Tids). In this example, it is assumed that there is a single Target and that the Target is using the access control signals to grant access to the bus. The intent of this feature is to provide the capability of allowing an Initiator to request access to a bus and allow the Target, with support of an arbiter, to grant access. The command accept is also

utilized to throttle the flow of data across target interfaces with different bandwidths as illustrated in the flow control examples. This implementation does not implement flow control and therefore does not implement the data accept signal. This example also utilizes a Transaction ID (Tids) signals to provide the application the ability to support out of order transfers. In some platform applications, it is difficult to guarantee multiple read transfers to multi sinks, even when flowed through a single Target interface, will be returned in the order they were requested. This is especially true for high-speed push packet interfaces such as Ethernet. Providing Transaction IDs enables the application to tag a transfer with a unique ID that will be returned with the data and used to inform the Initiator what transaction the data is associated with. The data accept signal is driven active though the entire transfer. This is common for initiators with no requirements to throttle the return of data from the target. For implementation where there are no requirements to throttle the flow of data from the target to the initiator, the DataA signals is driven to a logical active state (‘1’) continuously.

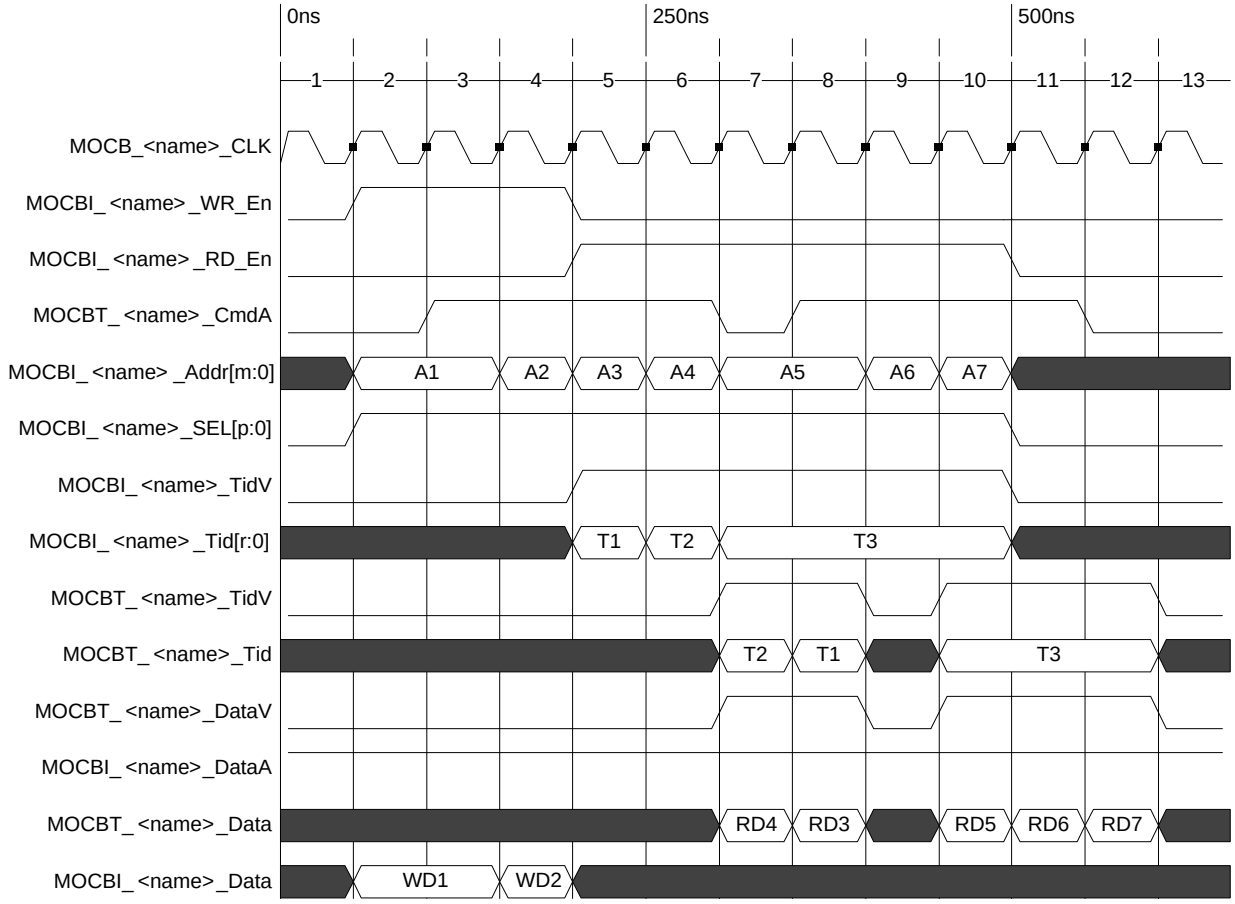

Figure 33 – Write/Read with Access Control and Tids

Sequences

1) On cycle 2, the Initiator requests a transfer by setting the write enable (WR_EN) as well as presenting valid address (A1) and data (WD1). These signals are held active until the Target indicates the command has been accepted.
2) On cycle 3, the Target signals the Initiator that the command has been accepted driving the command accept signal (CmdA) valid.
3) The Target will capture the address, data, and control signals on each clock and use them internally to perform the write.

4) The Initiator presents a new transfer on each subsequent clock cycle until the transfer is complete.
5) On cycle 5, the Initiator starts a read transfer by driving the write enable (WR_EN) invalid and the read enable valid (RD_EN). In this example, the Initiator is tagging the transfer with a transfer ID (Tid). The Tid field is set as well as the TidV to qualify the transfer as a tagged transfer. It is assumed that read transfer T1 may be returned out of order with respect to the other read transfer IDs.
6) Cycle 6, the Initiator tags the transfer with a new unique Transfer ID (T2). It is assumed that read transfer T2 may be returned out of order with respect to the other read transfer IDs.
7) Cycle 7, the Initiator tags the transfer with a new unique Transfer ID. It is assumed that read transfer(s) T3 may returned out of order with respect to the other read transfer IDs. T1 and T2. The three read transfers within the T3 transfer ID must be returned in order with respect to each other.
8) Cycle 7 the Target drives the command accept (CmdA) inactive indicating that it did not accept the current transfer on the bus. The Initiator holds the current command until the Target accepts the command. This may be done to temporarily grant access to a higher priority transfer.
9) Cycle 7, the Target begins to return the data from the read. The data is returned with a transfer ID (Tid) and a TidV qualifying the transfer as from a tagged transfer. The data is qualified valid by the data valid line (DataV). On this example, the T2 transfer is returned out of order with respect to the T1 transfer.
10) The Data accept (DataA) is driven high through the entire transfer indicating the Initiator is capable of receiving data presented by the Target.
11) Cycle 8, the Target returns the T1 data from the read command. The data is returned with a transfer ID (Tid) and a TidV qualifying the transfer as from a tagged transfer. The data is qualified valid by the data valid line (DataV). On this example, the T1 transfer is returned out of order with respect to the T2 transfer.
12) Cycle 8 the Target drives the CmdA active indicating the current command has been accepted.
13) Cycle 9, 10 the Initiator drives the remaining read commands. The Tid field is held constant during for each transfer. This indicates to the Target that the T3 transfers must maintain order with respect to the A5, A6, and A7 transfers, but may be out of order with respect to transfer T1 and T2.
14) Cycle 9, the Target invalidates the DataV signals indicating to the Initiator there is no valid Data on the bus.
15) On Cycle 10-12, the Target resumes transfer of the requested read data by presenting the data (Data), data valid (DataV), Transaction ID (Tid), and Transaction ID Valid (TidV)

D.1.3 Referenced Documents

The following documents of the exact issue shown form a part of this specification to the extent specified herein.

D.1.3.1 Government Documents

The following documents are part of this specification as specified herein.

D.1.3.1.1 Specifications

D.1.3.1.1.1 Federal Specifications

None

D.1.3.1.1.2 Military Specifications

None

D.1.3.1.1.3 Other Government Agency Documents

See section A.1.3.1.1.3.

D.2 SERVICES

Not applicable

D.3 SERVICE PRIMITIVES AND ATTRIBUTES

Not applicable

D.4 DEFINITIONS

D.4.1 Entity Definitions

D.4.1.1 Target Entity Description

```
--------------------------------------------------------------------------------
-- MHAL ON CHIP BUS (MOCB) Target Entity Description
--------------------------------------------------------------------------------
entity TARGET is

  generic (
    -- WIDTH OF ADDRESS BUS
    G_ADDR_BUS_WIDTH       : natural;

    -- WIDTH OF DATA BUS
    G_DATA_BUS_WIDTH       : natural;

    -- NUMBER OF EVENT SIGNALS
    G_EVENT_WIDTH          : natural;

    -- WIDTH OF ADDRESS SELECT BUS
    G_ADDR_SELECT_WIDTH    : natural;

    -- WIDTH OF TRANSFER SIZE BUS
    G_TRANSFER_SIZE_WIDTH  : natural;

    -- WIDTH OF TRANSACTION ID BUS
    G_TID_WIDTH            : natural);

  port (

--------------------------------------------------------------------------------
--COMMON SIGNALS
--------------------------------------------------------------------------------

-- MOCB CLOCK
MOCB_CLK              : in  std_logic;

-- SYNCHRONOUS RESET
MOCB_RESET            : in  std_logic;

--------------------------------------------------------------------------------
--BASIC REQUIRED INTERFACE SIGNALS
--------------------------------------------------------------------------------

-----------------------------
-- INITIATOR DRIVEN SIGNALS
-----------------------------
-- READ ENABLE
MOCBI_<NAME>_RD_EN  : in  std_logic;

-- WRITE ENABLE
MOCBI_<NAME>_WR_EN   : in  std_logic;

--ADDRESS BUS
MOCBI_<NAME>_ADDR    : in  std_logic_vector(G_ADDR_BUS_WIDTH-1 downto 0);

--DATA BUS
MOCBI_<NAME>_DATA    : in  std_logic_vector(G_DATA_BUS_WIDTH-1 downto 0);

-----------------------------
--TARGET DRIVEN SIGNALS
-----------------------------
--DATA BUS
MOCBT_<NAME>_DATA    : out std_logic_vector(G_DATA_BUS_WIDTH-1 downto 0);

-- DATA BUS VALID
```

```
MOCBT_<NAME>_DATAV       : out std_logic;

-- USER DEFINED EVENT LINES
MOCBT_<NAME>_EVENT       : out std_logic_vector(G_EVENT_WIDTH-1 downto 0);

-------------------------------------------------------------------------------
-- OPTIOINAL BASIC INTERFACE SIGNALS
-------------------------------------------------------------------------------
-- BYTE ENABLE
MOCBI_<NAME>_BYTE_EN     : in  std_logic;

-- ADDRESS MEMORY SELECT
MOCBI_<NAME>_SEL         : in  std_logic_vector(G_ADDR_SEL_WIDTH-1 downto 0);

-------------------------------------------------------------------------------
-- EXTENDED INTERFACE SIGNALS
-------------------------------------------------------------------------------
----------------------------
-- INITIATOR DRIVEN SIGNALS
----------------------------
-- DATA ACCEPT
MOCBI_<NAME>_DATAA       : in  std_logic;

-- TRANSACTION ID VALID
MOCBI_<NAME>_TIDV        : in  std_logic;

--   TRANSACTION ID
MOCBI_<NAME>_TID         : in  std_logic_vector(G_TID_WIDTH-1 downto 0);


-- TRANSFER LOCKED
MOCBI_<NAME>_LOCK        : in  std_logic;

-- TRANSFER SIZE
MOCBI_<NAME>_SIZE        : in  std_logic_vector(G_TRANSFER_SIZE_WIDTH-1 downto 0);

----------------------------
--TARGET DRIVEN SIGNALS
----------------------------
-- TRANSFER SIZE VALID
MOCBI_<NAME>_SIZEV       : in  std_logic;

-- COMMAND ACCEPT
MOCBT_<NAME>_CMDA        : out std_logic;

-- RETURN TRANSACTION ID VALID
MOCBT_<NAME>_TIDV        : out std_logic;

-- RETURN TRANSACTION ID
MOCBT_<NAME>_TID         : out std_logic_vector(G_TID_WIDTH-1 downto 0)

    );

end entity TARGET;
```

D.4.1.2 Initiator Entity Description

```
-------------------------------------------------------------------------------
-- MHAL ON CHIP BUS (MOCB) Initiator Entity Description
-------------------------------------------------------------------------------
entity INITIATOR is

  generic (
-- WIDTH OF ADDRESS BUS
G_ADDR_BUS_WIDTH       : natural;

-- WIDTH OF ADDRESS BUS
G_DATA_BUS_WIDTH       : natural;

-- NUMBER OF EVENT SIGNALS
G_EVENT_WIDTH          : natural;

-- WIDTH OF ADDRESS SELECT BUS
G_ADDR_SELECT_WIDTH    : natural;
```

```
-- WIDTH OF ADDRESS SELECT BUS
G_TRANSFER_SIZE_WIDTH  : natural;

-- WIDTH OF TRANSACTION ID BUS
G_Tid_WIDTH            : natural);

  port (
-------------------------------------------------------------------------------
--COMMON SIGNALS
-------------------------------------------------------------------------------
-- MOCB CLOCK
MOCB_CLK              : in  std_logic;

-- SYNCHRONOUS RESET
MOCB_RESET            : in  std_logic;

-------------------------------------------------------------------------------
--BASIC REQUIRED INTERFACE SIGNALS
-------------------------------------------------------------------------------
----------------------------
--TARGET DRIVEN SIGNALS
----------------------------
-- DATA BUS
MOCBT_<NAME>_DATA    : in  std_logic_vector(G_DATA_BUS_WIDTH-1 downto 0);

-- DATA VALID
MOCBT_<NAME>_DATAV   : in  std_logic;

-- USER DEFINED EVENT LINES
MOCBT_<NAME>_EVENT   : in  std_logic_vector(G_EVENT_WIDTH-1 downto 0);
----------------------------
-- INITIATOR DRIVEN SIGNALS
----------------------------
-- READ ENABLE
MOCBI_<NAME>_RD_EN   : out std_logic;

-- WRITE ENABLE
MOCBI_<NAME>_WR_EN   : out std_logic;

-- ADDRESS BUS
MOCBI_<NAME>_ADDR    : out std_logic_vector(G_ADDR_BUS_WIDTH-1 downto 0);

-- DATA BUS
MOCBI_<NAME>_DATA    : out std_logic_vector(G_DATA_BUS_WIDTH-1 downto 0);

-------------------------------------------------------------------------------
-- OPTIOINAL BASIC INTERFACE SIGNALS
-------------------------------------------------------------------------------
-- BYTE ENABLE
MOCBI_<NAME>_BYTE_EN : out std_logic;

-- ADDRESS MEMORY SELECT
MOCBI_<NAME>_SEL     : out std_logic_vector(G_ADDR_SELECT_WIDTH-1 downto 0);

-------------------------------------------------------------------------------
-- EXTENDED INTERFACE SIGNALS
-------------------------------------------------------------------------------
----------------------------
--TARGET DRIVEN SIGNALS
----------------------------
-- COMMAND ACCEPT
MOCBT_<NAME>_CMDA    : in  std_logic;

-- RETURN TRANSACTION ID VALID
MOCBT_<NAME>_TidV    : in  std_logic;

-- RETURN TRANSACTION ID
MOCBT_<NAME>_Tid     : in  std_logic_vector(G_Tid_WIDTH-1 downto 0);

----------------------------
-- INITIATOR DRIVEN SIGNALS
----------------------------
-- DATA ACCEPT
MOCBI_<NAME>_DATAA   : out  std_logic;
```

```
-- RETURN TRANSACTION ID VALID
MOCBI_<NAME>_TidV    : out std_logic;

-- RETURN TRANSACTION ID
MOCBI_<NAME>_Tid     : out std_logic_vector(G_Tid_WIDTH-1 downto 0);

-- TRANSFER LOCKED
MOCBI_<NAME>_LOCK    : out std_logic;

-- TRANSFER SIZE
MOCBI_<NAME>_SIZE    : out std_logic_vector(G_TRANSFER_SIZE_WIDTH-1 downto 0);

-- TRANSFER SIZE VALID
MOCBI_<NAME>_SIZEV   : out std_logic
    );

end entity INITIATOR;
```

D.4.2 Package definitions

D.4.2.1 Platform Description

```
package platform_pkg is

-------------------------------------------------------------------------------

-- MOCB Platform Configuration

-------------------------------------------------------------------------------

  -- Memory Mapping information

  constant c_<name>_pStartAddr    : std_logic_vector(AddrWidth-1 downto 0) := hex value;  --
Starting address of platform memory map

  constant c_<name>_pMemsize      : natural  := value;  -- memory allocation of platform in bytes

  -------------------------------------------------------------------------------

  -- Initiator Configurations

  -------------------------------------------------------------------------------

  -- Bus Size Configuration

  constant c_<name>_iDataAdapt    : boolean  := value;  -- Initiator Data Bus Adaption Provided

  constant c_<name>_iDataWidth    : natural  := value;  -- Initiator Data Bus Width

  constant c_<name>_iAddrWidth    : natural  := value;  -- Initiator Address Bus Width

  constant c_<name>_iEvents       : natural  := value;  -- Number of Initiator accepted Event
Lines

  -- CLK Crossing Configuration

  constant c_<name>_iClkCross       : boolean  := value;  -- Initiator Clk Crossing Provided

  constant c_<name>_iElasticDepth   : natural  := value;  -- Depth of elastic buffer

  constant c_<name>_iMClkPeriod_num : natural  := value;  -- Initiator MOCB CLK Period Numerator
in ps

  constant c_<name>_iMClkPeriod_den : natural  := value;  -- Initiator MOCB CLK Period
Denominator in ps

  -- Supported extended feature
```

```
  constant c_<name>_iTids           : boolean  := value;  -- Initiator Provides TIDS Support
  constant c_<name>_iTidsl          : natural  := value;  -- Maximum Length of TIDS Transfer
  constant c_<name>_iSize           : boolean  := value;  -- Initiator Provides Size field
  constant c_<name>_iCmdA           : boolean  := value;  -- Initiator Utilizes Command accept
  constant c_<name>_iDataA          : boolean  := value;  -- Initiator Utilizes Data Accept
  constant c_<name>_iLock           : boolean  := value;  -- Initiator Provides Lock Support
  -- Interface Characteristics

  constant c_<name>_iXferMax        : natural  := value;  -- Maximum Length of transfers

  ------------------------------------------------------------------------------
  -- Target Configuration
  ------------------------------------------------------------------------------
  -- Bus Size Configuration

  constant c_<name>_tDataAdapt      : boolean  := value;  -- Target Data Bus Adaption Provided
  constant c_<name>_tDataWidth      : natural  := value;  -- Target Data Bus Width
  constant c_<name>_tAddrWidth      : natural  := value;  -- Target Address Bus Width
  constant c_<name>_tEvents         : natural  := value;  -- Number of Target Provided Event
Lines
  -- CLK Crossing Configuration
  constant c_<name>_tClkCross       : boolean  := value;  -- Target Clk Crossing Provided
  constant c_<name>_tElasticBuf     : boolean  := value;  -- Elastic Buffer Used as Crossing
Technique
  constant c_<name>_tElasticDepth   : natural  := value;  -- Depth of Elastic Buffer
  constant c_<name>_tMClkPeriod_num : natural  := value;  -- Target MOCB CLK Period Numerator in
ps
  constant c_<name>_tMClkPeriod_den : natural  := value;  -- Target MOCB CLK Period Denominator
in ps
  -- Supported extended feature
  constant c_<name>_tTids           : boolean  := value;  -- Target Requires TIDS Support
  constant c_<name>_tSize           : boolean  := value;  -- Target Utilizes Size Field
  constant c_<name>_tCmdA           : boolean  := value;  -- Target Provides Command Accept
  constant c_<name>_tDataA          : boolean  := value;  -- Target Utilizes Data Accept
  constant c_<name>_tLock           : boolean  := value;  -- Target Provides Lock Support
  -- Interface Characteristic
  constant c_<name>_tDataVLat       : natural  := value;  -- Target Data Valid Cycle Latency
end package platform_pkg;
```

D.4.2.2 Waveform Description

```
package waveform_pkg is

--------------------------------------------------------------------------------

-- MOCB Waveform Configuration

--------------------------------------------------------------------------------

  -- Memory Mapping information

  constant c_<name>_wStartAddr      : std_logic_vector(AddrWidth-1 downto 0) := hex value;  --
Starting Address of Platform Memory Map

  constant c_<name>_wMemsize        : natural;  -- Memory Allocation of Platform in Bytes

  --------------------------------------------------------------------------------

  -- Initiator Configurations

  --------------------------------------------------------------------------------

  -- Bus Size Configuration

  constant c_<name>_iDataAdapt      : boolean  := value;  -- Initiator Data Bus Adaption Provided

  constant c_<name>_iDataWidth      : natural  := value;  -- Initiator Data Bus Width

  constant c_<name>_iAddrWidth      : natural  := value;  -- Initiator Address Bus Width

  constant c_<name>_iEvents         : natural  := value;  -- Number of Initiator accepted Event
Lines

  -- CLK Crossing Configuration

  constant c_<name>_iClkCross       : boolean  := value;  -- Initiator Clk Crossing Provided

  constant c_<name>_iElasticDepth   : natural  := value;  -- Depth of elastic buffer

  constant c_<name>_iMClkPeriod_num : natural  := value;  -- Initiator MOCB CLK Period Numerator
in ps

  constant c_<name>_iMClkPeriod_den : natural  := value;  -- Initiator MOCB CLK Period
Denominator in ps

  -- Supported extended feat

  constant c_<name>_iTids           : boolean  := value;  -- Initiator Provides TIDS Support

  constant c_<name>_iTidsl          : natural  := value;  -- Maximum Length of TIDS Transfer

  constant c_<name>_iSize           : boolean  := value;  -- Initiator Provides Size field

  constant c_<name>_iCmdA           : boolean  := value;  -- Initiator Utilizes Command accept

  constant c_<name>_iDataA          : boolean  := value;  -- Initiator Utilizes Data Accept

  constant c_<name>_iLock           : boolean  := value;  -- Initiator Provides Lock Support

  -- Interface Characteristics

  constant c_<name>_iXferMax        : natural  := value;  -- Maximum Length of transfers

  --------------------------------------------------------------------------------

  -- Target Configuration

  --------------------------------------------------------------------------------

  -- Bus Size Configuration
```

```
  constant c_<name>_tDataAdapt      : boolean  := value;  -- Target Data Bus Adaption Provided
  constant c_<name>_tDataWidth      : natural  := value;  -- Target Data Bus Width
  constant c_<name>_tAddrWidth      : natural  := value;  -- Target Address Bus Width
  constant c_<name>_tEvents         : natural  := value;  -- Number of Target Provided Event
Lines
  -- CLK Crossing Configuration
  constant c_<name>_tClkCross       : boolean  := value;  -- Target Clk Crossing Provided
  constant c_<name>_tElasticBuf     : boolean  := value;  -- Elastic Buffer Used as Crossing
Technique
  constant c_<name>_tElasticDepth   : natural  := value;  -- Depth of Elastic Buffer
  constant c_<name>_tMClkPeriod_num : natural  := value;  -- Target MOCB CLK Period Numerator in
ps
  constant c_<name>_tMClkPeriod_den : natural  := value;  -- Target MOCB CLK Period Denominator
in ps
  -- Supported extended features
  constant c_<name>_tTids           : boolean  := value;  -- Target Requires TIDS Support
  constant c_<name>_tSize           : boolean  := value;  -- Target Utilizes Size Field
  constant c_<name>_tCmdA           : boolean  := value;  -- Target Provides Command Accept
  constant c_<name>_tDataA          : boolean  := value;  -- Target Utilizes Data Accept
  constant c_<name>_tLock           : boolean  := value;  -- Target Provides Lock Support
  -- Interface Characteristics
  constant c_<name>_tDataVLat       : natural  := value;  -- Target Data Valid Cycle Latency


end package waveform_pkg;
```

D.5 DATA TYPES AND EXCEPTIONS

None

APPENDIX D.A – ABBREVIATIONS AND ACRONYMS

See section Appendix A.A.

APPENDIX D.B – PERFORMANCE SPECIFICATION

Not applicable

APPENDIX D.C – CLOCK SPECIFICATION

Table 6 provides the clock specification for the *MOCB*. This information will be provided by the JTRS Product Line developer.

Table 6 – MOCB Clock Specification

Specification	Location Description
MOCB Clock Frequency	*
MOCB Clock Location	*

Note: (*) These values should be filled in by JTRS Product Line developers.

E. MOCB RF CHAIN COORDINATOR (RFC) API EXTENSION

E.1 INTRODUCTION

This extension utilizes the Address/Data bus and Event interfaces defined in Section D.1 for specifying memory offsets and events, which provide for coordinated control of a JTRS Communication Channel's RF resources. MOCB provides the ability to read and write memory-mapped interfaces resident in either software (GPP/DSP) or hardware (FPGA), and supports event interfaces. This extension builds upon that base to provide specific RF related Parameters (FPGA registers or software memory locations) and Events (specific interrupt type signals within the FPGA) to support multiple RF capabilities.

This extension bears some resemblance to the MHAL RF Chain Coordinator Extension by providing RF control functions. However, this extension uses the MOCB paradigm of Logical Destinations (LDs) and Offsets to implement RF control functions, specifically Transmit Power Control (TPC) and Receiver Gain, using either the existing MOCB GPP/DSP operations or the MOCB FPGA interface. Note that additional functions can be added by defining additional LDs, parameters and event lists.

Each function is required to define a new Logical Destination (LD) that maps the beginning of the register/memory map for that particular function. All parameters appear at offsets within that LD.

E.1.1 Overview

This document contains as follows:

a. Section *E.1, Introduction,* of this document contains the introductory material regarding the Overview.

b. Section E.2, *Services* specifies the generic functions provided by the MOCB RFC.

c. Section *E.3, Service Primitives and Attributes*

d. Section E.4, *Interface Definitions*

e. Section *E.5, Data types and Exceptions*

f. Appendix E.A – *Abbreviations and Acronyms*

g. Appendix E.B – *Performance Specification*

E.2 SERVICES

This extension relies on pre-defined registers/memory-mapped locations that have a constant offset from LDs as specified in this document. Each Function consists of the following:

- Address offset to memory (register)
- Parameter name
- Parameter symbol
- Parameter access type
- Parameter size
- Description

Every parameter has a name associated with it and carries the "RFC_" prefix. The memory offsets are predefined integer constants relative to the LD. Each offset has a predefined parameter symbol that would typically be implemented via #defines in a C header file or as a constant in an HDL language, such as a VHDL package file. The memory offsets are byte addressable per the MOCB standard, however the parameter size may be 8, 16, 32, or 64 bits.

Each parameter has a specific access type that defines the access of the memory location from the context of the MOCB-Initiator (Device User). The MOCB-Initiator may be any type of CE (GPP, DSP, or FPGA). The parameter memory location addresses are constant offsets defined at compile time by the MOCB device. The memory locations can be read-only (RO) in which the MOCB-Target (Device Provider) writes the value to memory; the MOCB-Initiator may read the memory but cannot write to it and the Target cannot read the register. The memory locations can be write-only (WO) in which the MOCB-Initiator writes the value to memory; the MOCB-Target may read the memory but cannot write to it, and the Initiator cannot read the memory. The remaining type is read-write (RW) in which both the MOCB-Initiator and Target can read or write the memory without any constraints. The benefit of providing multiple access types is to protect against invalid data accesses and in some cases may provide a more streamlined implementation saving on resources.

E.2.1 I/F Modules

E.2.1.1.1 Module I/O

None

E.2.1.1.2 Module Design

The function of this interface is to transfer sample data from the Initiator to the Target, so its characteristics are hard coded and not configurable. Data bus width is num_bits, in multiples of eight bits. Address bus width is 0 (no address bus), and the clock frequency is synchronous to the sample rate. If a MOCB Target implements an address bus, it can be tied to ground at the inputs of the Target.

E.3 SERVICE PRIMITIVES AND ATTRIBUTES

E.3.1 Transmit Power Control Function: MOCBRFC_TPC

Modern waveforms have complex power management architectures and require precise control of the RF power. Some of these waveforms are derived from the 3G UMTS wireless standard and part of that standard defines a fairly complex transmit power control function, which serves multiple purposes, primarily bandwidth and User Equipment (UE) power conservation. Such waveforms require the transmit power to be configured at the right moment in time and at the correct accuracy. There are also terminal constraints, such as requirements for pre-distortion of the streaming data to linearize the Power Amplifier (PA) response, gain compensation to mitigate the compression effect when operating near saturation, and the need to dynamically change the PA bias in order to minimize battery drain. To satisfy such requirements, waveforms may use the MOCB RF Chain Coordinator function described in this section. The LD is defined as MOCBRFC_TPC. The terminal defines where this LD resides. This function specifies a table of parameters mapped to address offsets within the LD.

There is one Sync event defined for this function that synchronizes the register contents with a specific time in the waveform process. Different waveforms may require different time values for the Sync Event, depending on the time required by the terminal, while still providing a measure of optimization for the waveform. Timing and threshold relationships, such as timing data writes and data reads to the Sync event, will be specified as part of the porting process. There are multiple parameters in the API representing data from the WF to the Terminal that is available after the Sync event and is to be applied at a subsequent data boundary.

Since data and events for this function generally flow from the waveform to the terminal, the waveform will be the MOCB Initiator and the terminal will be the MOCB Target.

E.3.1.1 Parameters

Offset	Name	Symbol	Type	Description	Size
0x00	RFC_AvailPower	AVAILPWR	RO	Available power from terminal (dBm unsigned)	2
0x02	RFC_Status	STATUS	WO	Frame Status indicators	2
0x04	RFC_TPC_BasePower	TPCBASEPWR	WO	Power delta (dBm signed)	2
0x06	RFC_TPC_PwrRatio	TPCPWRRAT	WO	Digital power ratio to control signal (dB unsigned)	2
0x08	RFC_PercentNotch	PCTNOTCH	WO	Percent of the signal bandwidth notched	2
0x0A	RFC_WSD	WSD	WO	Waveform-specific data	2

E.3.1.2 Events

Vector Bit	Name	Type	Description	Size
0	RFC_Sync	Event	An event that occurs prior to power settings configuration and state change	N/A

E.3.2 Rx Gain Function: MOCBRFC_RXGAIN

This function provides the terminal receiver gain to the waveform when the external/terminal gain changes so that the waveform can calculate the received absolute power level. This is necessary to configure the user services appropriately and change various waveform settings to obtain the best performance.

The extension defines the general parameters for this function. The LD constant for this function is MOCBRFC_RXGAIN. The terminal will define where this LD will reside. In addition to the Gain parameter, a RFC_GainReduced boolean indicates whether the reported Gain is attenuated more than a fixed threshold level. Timing and threshold relationships, including the frequency/timing of gain reporting and the fixed threshold level of the RFC_GainReduced boolean, will be specified as part of the porting process.

E.3.2.1 Parameters

Offset	Name	Symbol	Type	Description	Size
0x00	RFC_Gain	RXGAIN	RO	Measured Receiver Gain – Nominal value with no attenuation. Units in dBm/lsb^2.	2
0x02	RFC_GainReduced	RXGAIN RDCT	RO	Boolean – Indicates TRUE (0x1) if the RXGAIN value is attenuated beyond a fixed threshold level from nominal, otherwise FALSE (0x0).	1
0x03	RFC_Reserved	RSVD	n/a	(future use)	1

E.3.2.2 Events

Vector Bit	Name	Type	Description	Size
0	RFC_GainUpdated	Event	An event indicating that the gain value has been updated.	N/A

E.4 INTERFACE DEFINITIONS

None

E.5 DATA TYPES AND EXCEPTIONS

Not applicable

APPENDIX E.A – ABBREVIATIONS AND ACRONYMS

See section Appendix A.A.

APPENDIX E.B – PERFORMANCE SPECIFICATION

Not applicable