Celoxica
pure acceleration

Generic Market ACcelerator GMAC

API Reference Guide

Celoxica

pure acceleration

About This Document

This document describes the GMAC API reference guide.

Copyright Information

Neither the whole nor any part of the information contained in, or the product described in, this document may be adapted or reproduced in any material form except with the prior written permission of the copyright holder.

The product described in this document is subject to continuous development and improvement. All particulars of the product and its use contained in this document are given by Celoxica Limited in good faith. However, all warranties implied or express, including but not limited to implied warranties of merchantability, or fitness for purpose, are excluded.

This document is intended only to assist the reader in the use of the product. Celoxica Limited shall not be liable for any loss or damage arising from the use of any information in this document, or any incorrect use of the product. The information contained herein is subject to change without notice and is for general guidance only.

Sales	sales@celoxica.com
Customer Support	support@celoxica.com
Website	http://www.celoxica.com

UK Head Office
Celoxica Limited
34 Porchester Road
London
W2 6ES, UK
Phone: +44 (0) 20 7313 3180

US Head Office
Celoxica Inc.
275 Madison Avenue, 6th floor
New York, NY
10016, USA
Phone: +1 (0) 212 880 2075

US Chicago Office
Celoxica Inc.
141 W Jackson Blvd, Suite 2350
Chicago, IL
60604, USA
Phone: +1 (0) 312 893 1204

Content

Revisions

Revision	Date	Description of Changes
R2012-8.3	16 JAN 2013	Release R2012-8.3 - Updated Snapshot Support section - Added a note to GMACFilteringInstrumentRequired()
R2012-8.1	20 DEC 2012	Release R2012-8.1 - Removed Terminology section (refer to GMAC Product Overview)
R2012-6.3	25 SEP 2012	Release R2012-7.0 - Updated DMA queue definition in section Terminology
R2012-6.0	20 AUG 2012	Release R2012-6.0 - Added hardware filtering guide section - Removed software filtering section - Added GMAC_SNAPSHOT_MESSAGES_LIMIT_REACHED to GMACSnapshotStatus enumeration
R2012-5.0	04 JUL 2012	New template
<= 3.18		Older versions

1. Core Functionality

1.1 Introduction

This document should be read in conjunction with:

- GMAC Product Overview
- GMAC Configuration Guide
- GMAC V1 and V3 Messages Templates

1.2 Description

1.2.1 Overview

The GMAC API is a dynamic library which accesses multiple feeds from the FPGA via the PCI Express bus and delivers them to the user application in a normalized format. The library logically separates data into channels for processing. The channels are then directed into multiple DMA queues which deliver the normalized data directly to a predetermined CPU. This enables the use of multiple CPUs when processing the normalized data produced by GMAC.

1.2.2 Usage

The market-dependent configuration information (which multicast addresses are directed to which channels, etc.) is stored in the GMAC configuration file. The user application calls `GMACOpen()`, which accesses the configuration file, initializes the card and the connections.

The channels are mapped to multiple DMA queues according to the workload.

The user should write the processing application with multithreading in mind, to ensure high CPU scalability in the future.

Each thread (CPU) can process one DMA queue. A DMA queue is initialized by `GMACQueueOpen()`.

To read data from the queues call `GMACQueueRead()`. The data returned is normalized, for more information on data processing, refer to section *Processing Data*.

GMAC also provides a mechanism to send control messages (commands) to specific threads (CPUs) by writing into the control queue. The library then ensures the data is read by the appropriate thread (CPU). This can be done by calling `GMACControlQueueWrite()`.

For performance benchmarking, GMAC is collecting various statistics. For more information, refer to section *Statistics*.

GMAC enables the user to use hardware assisted latency measurement in 8 ns resolution. For more information, refer to section *Latency*.

1.2.3 Error Handling

Most of the GMAC functions return status code, which may indicate an error or simply the state of operation. However, to allow the user to have centralized error handling, the GMAC API calls the error handler before returning the error status code. Users can implement and register their own error handler using `GMACRegisterErrorHandler()`.

The error handler receives an error status code, the function name and a string description. The error handler can then modify which status code is returned by the failing function.

Note:

> If an error occurs in a nested function within GMAC, the error handler may be invoked multiple times.

1.3 Functions

1.3.1 GMACOpen()

Description Opens all markets specified in the configuration file

Prototype

```
GMACStatus GMACOpen(GMACHandle **GMACH, const
Config *GMACConfigPtr)
```

Arguments	`GMACH`	GMAC handle
	`GMACConfigPtr`	Pointer to the market configuration

1.3.2 GMACClose()

Description Closes all markets specified in the configuration file

Prototype

```
GMACStatus GMACClose(GMACHandle *GMACH)
```

Arguments	`GMACH`	GMAC handle

1.3.3 GMACGetNativeOrderID()

Description Returns the native coding of an order ID. It is the user's responsibility to provide the buffer.

Celoxica strongly recommends the use of transcoded order IDs. Using this function to recover native order IDs will affect performance.

Prototype

```
GMACStatus GMACGetNativeOrderID(GMACHandle *GMACH,
int MarketID, uint64_t OrderID, char* Buffer, int
Length)
```

Arguments	`GMACH`	GMAC handle
	`MarketID`	Market ID
	`OrderID`	Actual normalized order ID
	`Buffer`	Text buffer to hold the native order ID, null terminated when returned
	`Length`	Length of the buffer supplied. Recommended size is 32 bytes.

1.3.4 GMACIsRecovery()

Description Returns the GMAC general recovery enable flag

Prototype

```
int GMACIsRecovery(GMACHandle *GMACH)
```

Arguments	`GMACH`	GMAC handle

Returns	1	Recovery is enabled (GMAC level)
	0	Recovery is not enabled (GMAC level)

1.3.5 GMACIsRecoveryMulticast()

Description Returns the multicast recovery enable flag

Prototype

```
int GMACIsRecoveryMulticast(GMACHandle *GMACH,
GMACMulticastID MulticastID)
```

Arguments	`GMACH`	GMAC handle
	`MulticastID`	Multicast ID

Returns 1 Recovery is enabled for the specified multicast ID (overrides the GMAC level)

0 Recovery is not enabled for the specified multicast ID (overrides the

GMAC level)

1.3.6 GMACNextExpected()

Description Returns next expected sequence number for the multicast

Prototype

```
GMACStatus GMACNextExpected(GMACHandle *GMACH,
GMACMulticastID MulticastID, uint64_t *Sequence)
```

Arguments

`GMACH`	GMAC handle
`MulticastID`	Multicast ID
`Sequence`	Next expected sequence

1.3.7 GMACChannelGetTranscodingType()

Description Returns encoding type for the channel.

Encoding type can be:

- 1 Consolidated book messages
- 0 Deep book messages

Prototype

```
GMACStatus GMACChannelGetTranscodingType(GMACHandle
*GMACH, GMACChannelID ChannelID, int *Enc)
```

Arguments

`GMACH`	GMAC handle
`ChannelID`	Channel ID
`Enc`	Pointer to where result is stored

1.3.8 GMACGetLocalChannelID()

Description Returns Channel number from `MulticastID`.

The `MulticastID` returned in **GMACPacketHeader** is globally unique, and generated internally by GMAC. Call this function to get the `LocalChannelID` from the `MulticastID`. The `LocalChannelID` is unique for multicasts of the same type, within the same market. It is the id that is present for each channel in the GMAC configuration file. Moreover, it is used to group complementary channels (of different types).

For instance, a regular 'updates' channel and a snapshot channel would have the same `LocalChannelID` if they are for the same market segment. The same two channels would have different `MulticastID`, generated by GMAC and thus not known before runtime.

Prototype

```
GMACStatus GMACGetLocalChannelID(GMACHandle *GMACH,
GMACMulticastID MulticastID, int *LocalChannelID)
```

Arguments

`GMACH`	GMAC handle
`MulticastID`	Multicast ID
`LocalChannelID`	Pointer to where channel number is stored

1.3.9 GMACGetLocalChannelIDEx()

Description Returns local channel ID and arbitration parameters

Prototype

```
GMACStatus  GMACGetLocalChannelIDEx(GMACHandle
*GMACH, GMACMulticastID MulticastID,
GMACMulticastID *BaseMulticastID, int
*LocalChannelID, char *LocalChannelChar, int
*ArbitrationMode, const char **ArbitrationModeDesc)
```

Arguments

`GMACH`	GMAC handle	
`MulticastID`	Multicast ID	
`BaseMulticastID`	The multicast id where this multicast is redirected to (-1 when not assigned)	
`LocalChannelID`	Pointer to where ’channel number’ is stored	
`LocalChannelChar`	‘A’	Line A
	‘B’	Line B
	‘’	No arbitration
`ArbitrationMode`	0	Arbitration off
	1	Redundancy
	2	Lag

1.3.10 GMACCorrectPrice()

Description Returns corrected prices in fixed point precision

Prototype

```
int64_t GMACCorrectPrice(GMACPriceFactor
*PriceFactor, int64_t Price)
```

Arguments

`PriceFactor`	Price factor
`Price`	Price from the GMAC message

Returns Price with GMAC_DECIMAL_PLACES fixed point precision

1.3.11 GMACCorrectPriceFloat()

Description Returns corrected prices in floating point precision

Prototype

```
double GMACCorrectPriceFloat(GMACPriceFactor
*PriceFactor, int64_t Price)
```

Arguments

`PriceFactor`	Price factor
`Price`	Price from the GMAC message

Returns Price with floating point precision

1.3.12 GMACGetFirmwareVersion()

Description Returns the firmware version

Prototype

```
GMACStatus GMACGetFirmwareVersion(int BoardNum, int
*Revision, int *VersionMajor, int *VersionMinor,
char Name[5])
```

Arguments

`BoardNum`	Board number
`Revision`	Pointer to the location where the revision number should be stored if the call is successful
`VersionMajor`	Pointer to the location where the version number should be stored if the call is successful
`VersionMinor`	Pointer to the location where the version number should be stored if the call is successful
`Name`	The bitfile name

1.3.13 GMACGetMaxBoardNum()

Description Returns the number of boards installed

Prototype

```
GMACStatus GMACGetMaxBoardNum(int *MaxBoardNum)
```

Arguments

`MaxBoardNum`	Pointer to store the total number of boards available

1.3.14 GMACGetMulticastMappingTable()

Description Returns the multicast mapping table for a specified market

Prototype

```
GMACStatus GMACGetMulticastMappingTable(GMACHandle
*H, unsigned MarketID, GMACMulticastMapping
**MulticastMappingPtr)
```

Arguments

`GMACH`	GMAC handle
`MarketID`	Market ID
`MulticastMappingPtr`	Address of the multicast mapping table

1.3.15 GMACMulticastIsRefresh()

Description Returns refresh status for a Multicast

Prototype

```
int GMACMulticastIsRefresh(GMACHandle *GMACH,
GMACMulticastID MulticastID)
```

Arguments

`GMACH`	GMAC handle
`MulticastID`	Multicast ID

Returns

1	Enabled
0	Not enabled

1.3.16 GMACQueryMarketHours()

Description Returns the market status based on the opening hours and last timestamp received

Prototype

```
GMACStatus GMACQueryMarketHours(GMACHandle *GMACH,
int MarketID, int *MarketOpenPtr, uint64_t
ExchangeTimestamp)
```

Arguments

`GMACH`	GMAC handle
`MarketID`	Market ID
`MarketOpenPtr`	Pointer to the open indicator set by the function
`ExchangeTimestamp`	Exchange timestamp to examine

1.3.17 GMACRegisterErrorHandler()

Description Registers an error handler.

Error handler is called before some status codes are returned. It allows user to change the error code being returned, as well as react on it.

For performance reasons, the error handler is not called when one of following status codes is returned:

- GMAC_STATUS_OK
- GMAC_STATUS_NODATA

Prototype `void GMACRegisterErrorHandler(GMACHandle *GMACH, GMACErrorHandler Handler)`

Arguments

`GMACH`	GMAC handle
`Handler`	The new error handler

1.3.18 GMACDefaultErrorHandler()

Description Default error handler that is called if an error occurs.

It can be overwritten using function `GMACRegisterErrorHandler()`.

Prototype `GMACStatus GMACDefaultErrorHandler(GMACHandle *GMACH, GMACStatus StatusCode, const char* Function, char *Msg)`

Arguments

`GMACH`	GMAC handle
`StatusCode`	Status code
`Function`	The function where the error happened
`Msg`	Specific message

1.3.19 GMACQueueRegisterErrorHandlerForADE()

Description Registers an error handler to the underlying ADE library. This error handler is called before the status codes are returned and allows the user to change the error code being returned, as well as react on it.

Prototype `GMACStatus GMACQueueRegisterErrorHandlerForADE(GMACHandle *GMACH, int Queue, ADEErrorHandler Handler)`

Arguments

`GMACH`	GMAC handle
`Queue`	Queue number
`Handler`	The new error handler

1.3.20 GMACChannelEnable()

Description Enables or disables the channel

Prototype `GMACStatus GMACChannelEnable(GMACHandle *GMACH, int MarketID, int LocalChannelID, int Enable)`

Arguments

`GMACH`	GMAC handle
`MarketID`	Market ID
`LocalChannelID`	Pointer to where channel number is stored
`Enable`	To enable or disable the channel

1.3.21 GMACChannelEnableEx()

Description Enables or disables the channel.

Extended function with additional IGMP membership parameter.

Prototype `GMACStatus GMACChannelEnableEx(GMACHandle *GMACH, int MarketID, int LocalChannelID, int Enable, int Subscribe)`

Arguments

`GMACH`	GMAC handle
`MarketID`	Market ID
`LocalChannelID`	Pointer to where channel number is stored
`Enable`	To enable or disable the channel
`Subscribe`	To enable or disable IGMP membership. If set to non-zero, GMAC will initiate or drop the IGMP subscription according to the `Enable` parameter. If set to zero, the IGMP subscription will always remain active and GMAC will simply either pass or discard the data.

1.3.22 GMACChannelEnableByType()

Description Enables or disables the Channel specifying channel type

Prototype `GMACStatus GMACChannelEnableByType(GMACHandle *GMACH, int MarketID, int LocalChannelID, GMACMulticastType MulticastType, int Enable);`

Arguments

`GMACH`	GMAC handle
`MarketID`	Market ID
`MulticastType`	Channel type
`LocalChannelID`	Pointer to where channel number is stored
`Enable`	To enable or disable the channel

1.3.23 GMACChannelEnableByTypeEx()

Description Enables or disables the Channel specifying channel type.

Extended function with additional IGMP membership parameter.

Prototype `GMACStatus GMACChannelEnableByTypeEx(GMACHandle *GMACH, int MarketID, int LocalChannelID, GMACMulticastType MulticastType, int Enable, int Subscribe);`

Arguments

`GMACH`	GMAC handle
`MarketID`	Market ID
`MulticastType`	Channel type
`LocalChannelID`	Pointer to where channel number is stored
`Enable`	To enable or disable the channel
`Subscribe`	To enable or disable IGMP membership. If set to non-zero, GMAC will initiate or drop the IGMP subscription according to the `Enable` parameter. If set to zero, the IGMP subscription will always remain active and GMAC will simply either pass or discard the data.

1.3.24 GMACSymbolSuffixGenericToNative()

Description Converts generic symbol suffix into market specific suffix if present

Prototype

```
GMACStatus
GMACSymbolSuffixGenericToNative(GMACHandle *GMACH,
unsigned MarketID, const char * GenericSymbol, char
* Dst)
```

Arguments	`GMACH`	GMAC handle
	`MarketID`	Market ID
	`GenericSymbol`	Generic symbol to be converted
	`Dst`	Pointer to the character array where the resulting string should be stored

1.3.25 GMACSymbolSuffixNativeToGeneric()

Description Converts market specific symbol suffix into generic suffix if present

Prototype

```
GMACStatus
GMACSymbolSuffixNativeToGeneric(GMACHandle *GMACH,
unsigned MarketID, const char * NativeSymbol, char
* Dst)
```

Arguments	`GMACH`	GMAC handle
	`MarketID`	Market ID
	`NativeSymbol`	Market specific symbol to be converted
	`Dst`	Pointer to the character array where the resulting string should be stored

1.3.26 GMACStatusToString()

Description Transforms the GMAC status to a string representation

Prototype

```
const char* GMACStatusToString(GMACStatus GMACS)
```

Arguments	`GMACS`	Status code

Returns String representation of the status

1.3.27 GMACMulticastTypeToString()

Description Transforms the multicast type to a string representation

Prototype

```
const char*
GMACMulticastTypeToString(GMACMulticastType Type)
```

Arguments	`GMACH`	GMAC handle
	`MulticastID`	Multicast querying from
	`Sequence`	Next expected sequence

Returns String representation of the multicast type

1.3.28 GMACIdentify()

Description Outputs the information about library and boards

Prototype

```
GMACStatus GMACIdentify(FILE *f)
```

Arguments	`f`	File pointer to where the information should be written

2. Hardware Filtering

2.1 Description

This feature consists in filtering market data according to symbol name and index, so that only market data for specific symbols are processed, with the filtering taking place in the accelerator card. This leads to improved overall performance since 'non-interesting' data are dropped by the card, so are never normalized or passed to the GMAC software-side.

Packet sequence numbers are changed if the card drops some data. Hardware filtering impact on the sequence numbering is illustrated by the following example:

Assuming the following three packets are received, hardware filtering being disabled:

#1	4	5	6	Packet Sequence = 4	Next Sequence = 7	Messages = 3

#2	7	8	9	10	Packet Sequence = 7	Next Sequence = 11	Messages = 4

#3	11	12	13	Packet Sequence = 11	Next Sequence = 14	Messages = 3

Hardware filtering being enabled, if message #8 is dropped by the card, the second packet will be re-numbered as follows:

#2	7		8	9	Packet Sequence = 7	Next Sequence = 11	Messages = 4

Hardware filtering being enabled, if message #7 is dropped by the card, the second packet will be re-numbered as follows:

#2		7	8	9	Packet Sequence = 7	Next Sequence = 11	Messages = 4

The first message in a packet is given the same sequence number as the original packet.

Hardware filtering being enabled, if messages #7 to #10 are dropped by the card (the second packet is fully dropped), the third packet is re-numbered as follows:

#3	11	12	13	Packet Sequence = 7	Next Sequence = 14	Messages = 3

Users can therefore easily determine whether packets have been dropped somewhere other than by the hardware filtering module by monitoring the packet sequence and packet next sequence.

2.2 Functions

2.2.1 GMACFilteringEnable()

Description

This function enables hardware filtering for the market providing the instruments list to be filtered has been set using `GMACFilteringInstrumentRequired()`.

Once filtering is enabled, it is not allowed to add extra instruments to the list of filtered instruments.

Prototype

```
GMACStatus GMACFilteringEnable(GMACHandle *GMACH,
int MarketID)
```

Arguments

`GMACH`	GMAC handle
`MarketID`	Market ID

2.2.2 GMACFilteringInstrumentRequired()

Description

This function is used to add symbol names or symbol indexes to the filtering list for the specified market.

Note:

Only symbol IDs may be passed to the method for plugin CME2 i.e. if a symbol name is passed, the filtering will not work for that symbol.

Prototype

```
GMACStatus
GMACFilteringInstrumentRequired(GMACHandle *GMACH,
int MarketID, const char* InstrumentID);
```

Arguments

GMACH	GMAC handle
MarketID	Market ID
InstrumentID	Instrument identifier

2.2.3 GMACFilteringSupportedMode()

Description This function returns the filtering mode for a specified market.

If filtering is supported, it also provides whether a symbol index or symbol name has to be provided.

Filtering has to be enabled in the configuration file using `<hardware-filtering>`.

The function returns:

- GMAC_STATUS_OK if hardware filtering is supported for the specified market
- GMAC_STATUS_NOT_IMPLEMENTED if hardware filtering is not supported for the specified market

Prototype

```
GMACStatus GMACFilteringSupportedMode(GMACHandle
*GMACH, int MarketID, GMACFilteringMode
*FilteringMode);
```

Arguments

GMACH	GMAC handle
MarketID	Market ID
FilteringMode	Filtering mode

2.3 Hardware Filtering Guide

1. The user enables the hardware filtering module in the configuration file using `config.gmac.hardware-filtering`.
2. Then the user enables hardware filtering on a per-market basis in the configuration file using `config.gmac.markets.market.hardware-filtering`.

   Hardware filtering must be enabled globally using `config.gmac.hardware-filtering` to be enabled or disabled at the market level.
3. The user checks whether hardware filtering is supported for a specified market and whether hardware filtering uses a symbol index or symbol name for that market using `GMACFilteringSupportedMode()`.
4. Assuming hardware filtering is supported for a specified market, the users sets the instrument filtering lists using `GMACFilteringInstrumentRequired()`. The function must be called for each instrument to be filtered.
5. The user enables hardware filtering using `GMACFilteringEnable()`.

3. Processing Data

3.1 Description

3.1.1 Overview

The GMAC provides data in a normalized format across all supported feeds. This enables the user to write code processing GMAC messages and listen to all markets supported in GMAC.

GMAC is hiding complexities of maintaining connections, gap detection, recovery mechanisms and various differences between markets.

GMAC is stateless (except sequence checking); each packet received is normalized into one or multiple GMAC messages. Any cross referencing between data (price factor, etc.) in different packets must be done in the layers after GMAC.

On top of that, GMAC provides an easy way to write multithreaded code. Ensuring that data belonging to the same channel is never passed to two different threads at the same time. This allows users to have lockless, high speed code.

3.1.2 Data Classification

GMAC is working with data on the multicast level. Each multicast stream represents a channel. For each channel there is a gap detection and recovery mechanism. Furthermore, each channel belongs to a Market (or exchange), the market implies rules for the channel, how the data should be normalized and how the recovery is handled.

As the channels represents logically separated data, these can be processed in parallel. GMAC architecture ensures that data from the same channel is not passed to two different queues (threads) to hide locking complexities from the user. However, cross-referencing data between channels should be protected by locks or other mechanisms on the user side.

3.2 Functions

3.2.1 GMACQueueOpen()

Description Opens a new reading queue

Prototype `GMACStatus GMACQueueOpen(GMACHandle *GMACH, int Queue)`

Arguments		
	`GMACH`	GMAC handle
	`Queue`	Queue number

3.2.2 GMACQueueClose()

Description Closes a reading queue

Prototype `GMACStatus GMACQueueClose(GMACHandle *GMACH, int Queue)`

Arguments		
	`GMACH`	GMAC handle
	`Queue`	Queue number

3.2.3 GMACInitDynamicMcDesc()

Description Initialises **GMACDynamicMcDesc** structure

Prototype `void GMACInitDynamicMcDesc(GMACDynamicMcDesc *McDesc)`

Arguments		
	`McDesc`	Address of multicast descriptor structure to initialize

3.2.4 GMACOpenMulticast()

Description Opens a multicast.

Enables multicasts to be dynamically opened and closed. May be called after the GMAC Queue has been opened.

These functions should be used with care as otherwise they could confuse the code which normalizes to generic GMAC template the message stream for this market.

Prototype

```
GMACStatus  GMACOpenMulticast(GMACHandle *GMACH,
int MarketIndex, int LocMid, int Queue,
GMACDynamicMcDesc *McDesc, char *Interface, char
*AdditionalInfo, GMACChannelID DesiredChannelID,
GMACMulticastID *MID)
```

Arguments

`GMACH`	GMAC handle
`MarketIndex`	Multicast ID
`LocMid`	A number to group multicasts for processing
`Queue`	Queue number
`McDesc`	Address of multicast descriptor structure. Can be Null to use defaults.
`Interface`	Interface to use, e.g. "ac0"
`AdditionalInfo`	Generally the IP and port address, e.g. "1.2.3.4:5"
`ChannelID`	Channel (as distinct from Multicast), used to logically group multicasts after GMAC. May be specified as 0xFFFFFFFF when it will be assigned automatically.
`MID`	Address to write the new Multicast ID

3.2.5 GMACCloseMulticast()

Description Closes a multicast.

Enables multicasts to be dynamically closed. No errors are returned, the multicast will not be open.

These functions should be used with care as otherwise they could confuse the code which normalizes to generic GMAC template the message stream for this market.

Prototype

```
void GMACCloseMulticast(GMACHandle *GMACH,
GMACMulticastID MulticastID)
```

Arguments

`GMACH`	GMAC handle
`MulticastID`	Multicast ID

3.2.6 GMACQueueRead()

Description Returns data on the specific hardware queue.

This function polls data from the hardware queue. It checks channel/queue relations to prevent multiple CPUs processing the same part of the book. It is guaranteed that no other CPU polling on a different hardware queue is processing data from the same channel until `GMACQueueRelease()` is called.

The function calls the market specific GMAC plug-in and returns the data in the Celoxica normalized format. It provides sequence integrity checking and recovery. The returned data are guaranteed to be in the correct order without gaps.

The Data pointer is set to the first packet in the reorder list if the return code is GMAC_STATUS_RECOVERY_BUFFERING or GMAC_STATUS_RECOVERY_FAILED. This data should not be processed as market data since it will be returned again with GMAC_STATUS_OK. The user should not call `GMACQueueRelease()` until GMAC_STATUS_OK is returned.

If the return code is GMAC_STATUS_OK, the user must call `GMACQueueRelease()`, after processing the passed data.

GMAC_STATUS_CONTROL is returned with the Data pointer provided by user in call `GMACControlQueueWrite()`. In this case the mechanism to free the data is the user's responsibility.

GMAC_STATUS_NO_CHANNELS is returned when there are no inputs open on a GMAC/ADE queue. This can occur if there are no inputs (multicasts) on this queue.

GMAC_STATUS_RECONNECT is returned if GMAC attempts to reconnect to a socket channel.

GMAC_STATUS_ADE_ERROR is returned if the maximum number of retries has been reached (set by config node `gmac.markets.market.socket-max-retries`) or the retry time frame has been exceeded (set by config node `gmac.markets.market.socket-max-retries-us`) for a socket channel.

GMAC_STATUS_ADE_ERROR is the default return status if no reconnection parameters are configured or the channel is not a socket.

Note:

> In cases where GMAC needs to dynamically open/close multicasts this may be a 'soft' error. Handling is up to the caller.

If a duplicate packet is found and consequently dropped in the reordering buffer, GMAC_STATUS_NODATA is returned, and **GMACStats** .PacketsDroppedAsDuplicates counter is incremented.

The function is non-blocking.

Prototype `GMACStatus GMACQueueRead(GMACHandle *GMACH, int Queue, char **Data, GMACChannelID *ChannelID)`

Arguments		
	`GMACH`	GMAC handle
	`Queue`	The queue number
	`Data`	Pointer to where the returned data are to be stored
	`ChannelID`	Channel the data were received from

3.2.7 GMACQueueRelease()

Description Releases data returned from `GMACQueueRead()` for cases where `GMACQueueRead()` status code was GMAC_STATUS_OK.

Prototype `void GMACQueueRelease(GMACHandle *GMACH, char *Data)`

Arguments		
	`GMACH`	GMAC handle
	`Data`	Pointer returned by `GMACQueueRead()`

3.2.8 GMACControlQueueWrite()

Description Writes a control message.

Writes a message to the control queue, the message will be read by the thread responsible for the specified channel. The data will then be read by `GMACQueueRead()` call with highest priority.

This function allows the user to send control messages to the CPUs responsible for specific channels.

Function `GMACQueueRead()` will return GMAC_STATUS_CONTROL and supplied data pointer will be given to the user. Any data manipulation alloc/free must be done by the user. `GMACQueueRelease()` must not be called.

Warning: The behavior after supplying NULL pointer is undefined.

Prototype `GMACStatus GMACControlQueueWrite(GMACHandle *GMACH, char *Data, GMACChannelID ChannelID)`

Arguments

	`GMACH`	GMAC handle
	`Data`	Pointer to a control message
	`ChannelID`	The channel the control message applies to

3.2.9 GMACControlQueueWriteMarket()

Description Writes a control message.

Sends the message on all channels the market has, using `GMACControlQueueWrite()` function.

Each channel receives the same copy of the data (same supplied pointer). User should ensure, that data are freed only after read certain times.

Prototype `GMACStatus GMACControlQueueWriteMarket(GMACHandle *GMACH, char *Data, int MarketID)`

Arguments

	`GMACH`	GMAC handle
	`Data`	Pointer to a control message
	`MarketID`	Market ID the messages will be send on

3.2.10 GMACMulticastGetMarketID()

Description Returns the market ID for a specific multicast.

Prototype `int GMACMulticastGetMarketID(GMACHandle *GMACH, GMACMulticastID MulticastID, const char **NamePtr)`

Arguments

	`GMACH`	GMAC handle
	`MulticastID`	Multicast ID
	`NamePtr`	Returns the pointer to the market name

Returns Market ID or -1 if there is no market registered on specified multicast

3.2.11 GMACMulticastGetMarketIDFromChannel()

Description Returns the market ID for a specific channel

Prototype `int GMACMulticastGetMarketIDFromChannel(GMACHandle *GMACH, int ChannelID, const char **NamePtr)`

Arguments

	`GMACH`	GMAC handle
	`ChannelID`	Channel ID
	`NamePtr`	Returns the pointer to the market name

Returns Market ID or -1 if Channel ID invalid or there is no market registered for the Channel ID

3.2.12 GMACGetMarket()

Description Returns the name of market for supplied Market ID

Prototype

```
GMACStatus GMACGetMarket(GMACHandle *GMACH, char
**MarketName, uint8_t MarketID)
```

Arguments	`GMACH`	GMAC handle
	`MarketName`	Pointer to where pointer to the name is stored
	`MarketID`	Market ID

3.2.13 GMACGetMarketNames()

Description Returns the list of market names

Prototype

```
GMACStatus GMACGetMarketnames(GMACHandle *GMACH,
GMACMarketNames MarketNames)
```

Arguments	`GMACH`	GMAC handle
	`MarketNames`	Pointer to the market names structure

3.2.14 GMACGetQueueWithChannelID()

Description Returns the queue for the specified channel ID

Prototype

```
GMACStatus GMACGetQueueWithChannelID(GMACHandle
*GMACH, GMACChannelID ChannelID, int *Queue)
```

Arguments	`GMACH`	GMAC handle
	`ChannelID`	Channel ID
	`Queue`	Returns the pointer to the queue

3.2.15 GMACGetQueueWithMarketID()

Description Returns the queue for the specified market ID

Prototype

```
GMACStatus GMACGetQueueWithMarketID(GMACHandle
*GMACH, int MarketID, int *Queue)
```

Arguments	`GMACH`	GMAC handle
	`MarketID`	Market ID
	`Queue`	Returns the pointer to the queue

3.2.16 GMACMarketGetTranscodingType()

Description Returns the transcoding type

Prototype

```
GMACStatus GMACMarketGetTranscodingType(GMACHandle
*GMACH, uint8_t MarketID, int *Enc)
```

Arguments	`GMACH`	GMAC handle
	`MarketID`	Market ID
	`Enc`	Pointer to where function returns the transcoding type

3.2.17 GMACDefaultPriceFactor()

Description Returns the default price factor for the market specified.

Returns -1 for feeds without default price factor; in that case the price factor needs to be determined from a data referential message.

Prototype `GMACStatus GMACDefaultPriceFactor(GMACHandle *GMACH, GMACChannelID ChannelID, GMACPriceFactor *PriceFactor)`

Arguments		
	`GMACH`	GMAC handle
	`ChannelID`	Channel ID
	`PriceFactor`	Pointer to where the price factor is returned

3.2.18 GMACDefaultCurrency()

Description Returns the default currency for the market specified

Prototype `GMACStatus GMACDefaultCurrency(GMACHandle *GMACH, GMACChannelID ChannelID, char **Currency)`

Arguments		
	`GMACH`	GMAC handle
	`ChannelID`	Channel ID
	`Currency`	Pointer to where the currency is returned

3.2.19 GMACDefaultRoundLotSize()

Description Returns the default round lot size for the market specified.

Returns -1 if the round lot size is unknown and needs to be determined from a data referential message. Feed that do not use round lots will return a round lot size of 1 here

Prototype `GMACStatus GMACDefaultRoundLotSize(GMACHandle *GMACH, GMACChannelID ChannelID, uint64_t *RoundLotSize)`

Arguments		
	`GMACH`	GMAC handle
	`ChannelID`	Channel ID
	`RoundLotSize`	Pointer to where the round lot size is returned

3.2.20 GMACGetDefaultStatus()

Description Returns the default status values for Market, Sector, Segment, Channel and Instrument.

A client can use these as the initial values until they are known. Values for Sector, Segment and Channel are ’Open’ where these levels are not used.

Prototype `GMACStatus GMACGetDefaultStatus(GMACHandle *GMACH, int MarketID, GMACDefaultMarketStatus *DefStatus)`

Arguments		
	`GMACH`	GMAC handle
	`MarketID`	Market ID
	`DefStatus`	Pointer to where the status values are returned

3.2.21 GMACGetADEProperty()

Description Returns a plugin-specific property from the underlying ADE channel.

Prototype `GMACStatus GMACGetADEProperty(GMACHandle *GMACH, GMACMulticastID MulticastID, int Property, void`

```
**Result)
```

Arguments

`GMACH`	GMAC handle
`MulticastID`	Multicast ID
`Property`	Property index - ADE plugin specific
`Result`	Pointer to where the data are returned

Example We want to obtain the timestamps recorded in a pcap file that we are reading in via the file plugin.

```
struct timeval *tv;
GMACGetADEProperty(..., 0, (void**)&tv);
```

Here we pass 0 as property and a pointer to `struct timeval *` as Result. The function will return the local timestamp of the last read packet, as recorded in the PCAP file.

3.2.22 GMACMulticastSync()

Description Sets the expected sequence number for specific multicast.

This function allows the user to sync on specific sequence number, to take advantage of snapshot messages for example. If there are any data buffered in GMAC until this sequence number, those will be dropped. Any data received with sequence number lower than this one will be dropped as well.

This command will be processed by CPU currently responsible for the specific channel asynchronously.

If the user tries to sync on sequence number lower than the next expected one (means GMAC doesn't have the required data anymore), GMAC tries to recover the missing messages with respect of re-request timeout. The data in the reorder buffer are aging and it can easily happen that the gap is too old to be re-requested.

To avoid synchronizing on deprecated sequence numbers, it is recommended to use this function in combination with `GMACMulticastBuffer()`.

Prototype

```
GMACStatus GMACMulticastSync(GMACHandle *GMACH,
GMACMulticastID MulticastID, uint64_t Sequence)
```

Arguments

`GMACH`	GMAC handle
`MulticastID`	Multicast ID
`Sequence`	Sequence to sync to

3.2.23 GMACMulticastSetTaint()

Description Sets the taint flag for a multicast

Prototype

```
GMACStatus GMACMulticastSetTaint(GMACHandle *GMACH,
GMACMulticastID MulticastID, int Taint)
```

Arguments

`GMACH`	GMAC handle
`MulticastID`	Multicast ID
`Taint`	The taint can bet set or reset

3.2.24 GMACMulticastBuffer()

Description Starts buffering data on specific multicast.

This function helps to synchronize onto specific sequence number with the `GMACMulticastSync()` function.

The function call will cause all data on the specific channel to be buffered.

Each time a new packet is buffered, `GMACQueueRead()` will return status GMAC_STATUS_BUFFERING with a pointer to the first buffered packet data, which allows user to get sequence number of the first buffered packet. `GMACQueueRelease()` should not be called.

To disable buffering and start receiving data, user should use the `GMACMulticastSync()` function.

Buffered data will be stored in reorder/user buffer pool, the size can be set in the configuration file via node `config.gmac.bufferpool`.

Note:

> The bufferpool is shared between all channels on a single core.

If there is no space in the buffer pool, the data will be dropped and recovered later using recovery mechanisms. Using buffering will increase the latency and affect the performance so it only should be used for synchronization reasons.

Prototype `GMACStatus GMACMulticastBuffer(GMACHandle *GMACH, GMACMulticastID MulticastID)`

Arguments

`GMACH`	GMAC handle
`MulticastID`	Multicast ID

3.2.25 GMACTranslateMarketToEISIN()

Description Translates market local name to EISIN.

Prototype `GMACStatus GMACTranslateMarketToEISIN(GMACHandle *GMACH, GMACChannelID ChannelID, GMACLocalName Src, GMACEISIN Dst)`

Arguments

`GMACH`	GMAC handle
`ChannelID`	Channel ID
`Src`	Market local name
`Dst`	Pointer to where the EISIN will be copied (must be allocated by caller)

3.2.26 GMACTranslateMarketToEISINEx()

Description Translates the market local name to EISIN if this feature is supported for the specific feed.

Prototype `GMACStatus GMACTranslateMarketToEISINEx(GMACHandle *GMACH, int MarketID, GMACLocalName Src, GMACEISIN Dst)`

Arguments

`GMACH`	GMAC handle
`MarketID`	Market ID
`Src`	Market local name
`Dst`	Pointer to where the EISIN will be copied (must be allocated by caller)

3.2.27 GMACTranslateEISINToMarket()

Description Translates EISIN to market local name if this feature is supported for the specific feed.

Prototype `GMACStatus GMACTranslateEISINToMarket(GMACHandle *GMACH, GMACChannelID ChannelID, GMACEISIN Src, GMACLocalName Dst)`

Arguments		
	`GMACH`	GMAC handle
	`ChannelID`	Channel ID
	`Src`	EISIN
	`Dst`	Pointer to where the market local name will be copied (must be allocated by caller)

3.2.28 GMACTranslateEISINToMarketEx()

Description Translates the market local name to EISIN if this feature is supported for the specific feed.

Prototype `GMACStatus GMACTranslateEISINToMarketEx(GMACHandle *GMACH, int MarketID, GMACEISIN Src, GMACLocalName Dst)`

Arguments		
	`GMACH`	GMAC handle
	`MarketID`	Market ID
	`Src`	EISIN
	`Dst`	Pointer to where the market local name will be copied (must be allocated by caller)

3.2.29 GMACPrintMessages()

Description Outputs textual representation of Celoxica normalized messages into the supplied stream.

Prototype `void GMACPrintMessages(FILE *f, char *Data)`

Arguments		
	`f`	Stream to print to
	`Data`	Pointer to where Celoxica messages are

3.2.30 GMACPrintMessagesEx()

Description Outputs textual representation of Celoxica normalized messages into the supplied stream.

Prototype `void GMACPrintMessagesEx(FILE *f, char *Data, int(*IsPrint)(GMACMessageHeader *Header))`

Arguments		
	`f`	Stream to print to
	`Data`	Pointer to where Celoxica messages are
	`IsPrint`	Callback function to determine whether to print the message or not

3.3 Processing Data Guide

1. The user creates a queue for each processor he wishes to use. The queue is initialized using `GMACQueueOpen()`.
2. When a queue is initialized, the user should call `GMACQueueRead()` in a loop, to poll the data.
3. The user is required to call `GMACQueueRelease()` when he has finished using the data.

4. The user closes a queue using `GMACQueueClose()`.

GMAC provides some other information about the data as well, for instance:

- `GMACMulticastGetMarketIDFromChannel()` to check which market the channel belongs to
- `GMACDefaultPriceFactor()` to get the default price factor used by some feeds
- `GMACDefaultCurrency()` to get the default currency used by some feeds

GMAC aims to be able to assign same global name to instruments of different markets which represents the same product. Functions `GMACTranslateMarketToEISIN()` and `GMACTranslateEISINToMarket()`, should translate market local name to EISIN (Extended ISIN code) and other way around. This feature support depends on the feed.

User may require doing some processing on data belonging to specific channel: to avoid locking, GMAC provides a feature which allows to send command to thread responsible for the channel. This ensures the processing will be performed on the thread processing the channel, so no race conditions will be involved.

To send command use the control queue `GMACControlQueueWrite()`. It allows to pass any pointer, it will be picked by `GMACQueueRead()` in the target thread.

It may be necessary to synchronize the channel on particular sequence number, not known in advance. GMAC provides buffering feature with subsequent synchronization. Use the calls: `GMACMulticastBuffer()` and `GMACMulticastSync()`. This is required e.g. on EUREX feed.

4. Statistics

4.1 Description

GMAC is gathering statistics about processed data.

Pointer to structure where statistics are stored can be retrieved by calling `GMACStatsMulticast()` and `GMACStatsQueueGlobal()`.

4.2 Functions

4.2.1 GMACStatsMulticast()

Description Returns pointer to live statistics of a particular multicast

Prototype

```
GMACStatus GMACStatsMulticast(GMACHandle *H,
GMACMulticastID MulticastID, GMACStats **Stats)
```

Arguments		
	GMACH	GMAC handle
	MulticastID	Multicast ID
	Stats	Pointer to where statistics will be passed

4.2.2 GMACStatsChannel()

Description Returns pointer to live statistics of a particular channel

Prototype

```
GMACStatus GMACStatsChannel(GMACHandle *GMACH, int
MarketID, int LocalChannelID, GMACStats **Stats)
```

Arguments		
	GMACH	GMAC handle
	MarketID	Market ID
	LocalChannelID	Channel ID local to the market, the ID specified in the configuration file
	Stats	Pointer to where statistics will be passed

4.2.3 GMACStatsQueueGlobal()

Description Returns global statistics.

The user can get a copy of the global statistics. It is useful for monitoring all the multicast channels.

If the user detects packet drops, the user should check `CRCErrors` count returned by `GMACStatsQueueGlobal()`. A count greater than zero indicates a likely network infrastructure fault such as bad cabling, bad SFPs or a faulty switch.

The statistics are always representing the current state.

Prototype

```
GMACStatus GMACStatsQueueGlobal(GMACHandle *H, int
Queue, GMACStats **Stats, uint64_t *PacketsMissed,
uint64_t *CRCErrors)
```

Arguments		
	GMACH	GMAC handle
	Queue	The queue number
	Stats	Pointer to where statistics will be passed
	PacketsMissed	Packets dropped by the accelerator card, because software wasn't keeping up
	CRCErrors	Packets dropped by the accelerator card, because of underlying protocol errors

4.2.4 GMACStatsMulticastCount()

Description Returns number of open multicasts

Prototype

```
void GMACStatsMulticastCount(GMACHandle *H, int
*Count)
```

Arguments

	`GMACH`	GMAC handle
	`Count`	Multicast count

4.2.5 GMACStatsMarketMulticastCount()

Description Returns number of open multicasts for specific market

Prototype

```
GMACStatus GMACStatsMarketMulticastCount(GMACHandle
*H, int MarketID, int *Count)
```

Arguments

	`GMACH`	GMAC handle
	`MarketID`	Market ID
	`Count`	Multicast count

4.2.6 GMACStatsMarketChannelCount()

Description Returns number of open channels for specific market

Prototype

```
GMACStatus GMACStatsMarketChannelCount(GMACHandle
*H, int MarketID, int *Count)
```

Arguments

	`GMACH`	GMAC handle
	`Count`	Channel count

4.2.7 GMACStatsMarketCount()

Description Returns number of markets

Prototype

```
void GMACStatsMarketCount(GMACHandle *H, int
*Count)
```

Arguments

	`GMACH`	GMAC handle
	`Count`	Market count

4.2.8 GMACStatsGetCredit()

Description Queries the status of the DMA queue.

This is a debug function that can be used to query the amount of free space in the DMA queue. It can be useful during integration development, to ascertain whether the user code is processing incoming packets quickly enough and thus not falling behind and causing the DMA queue to fill up. This function call will affect latency and should therefore not be used in performance threads.

This function is only available for the "lldt" plugin (i.e. for data streams emanating from accelerator cards).

Prototype

```
GMACStatus GMACStatsGetCredit(GMACHandle *H, char
*Plugin, int Queue, int BoardIndex, uint64_t
*CreditAvail, uint64_t *CreditMax)
```

Arguments	`GMACH`	GMAC handle
	`Queue`	Queue number
	`BoardIndex`	Board index number. If set to -1, the first board on the queue is selected
	`CreditAvail`	Available DMA credit in bytes
	`CreditMax`	Maximum amount of credit for the given DMA buffer; i.e. the size of the DMA buffer

4.2.9 GMACStatsMulticastPrint()

Description Prints statistics of a multicast

Prototype

```
void GMACStatsMulticastPrint(GMACHandle *H, int
Queue, GMACMulticastID MulticastID, FILE *f)
```

Arguments	`GMACH`	GMAC handle
	`MulticastID`	MulticastID
	`f`	Stream to print to

4.2.10 GMACStatsAllMulticastPrint()

Description Prints all channels using supplied function

Prototype

```
void GMACStatsAllMulticastPrint(GMACHandle *H, int
Queue, FILE *f, void (*Fnc)( GMACHandle *H, int
Queue, GMACChannelID ChannelID, FILE *f))
```

Arguments	`GMACH`	GMAC handle
	`Str`	String to print
	`f`	Stream to print to

4.2.11 GMACStatsAllMulticastPrintSqn()

Description Prints all channels for the next expected sequence

Prototype

```
Void GMACStatsAllMulticastPrintSqn(GMACHandle *H,
FILE *f, void (*Fnc)( GMACHandle *H, char *Str,
FILE *f))
```

Arguments	`GMACH`	GMAC handle
	`f`	Stream to print to

4.2.12 GMACStatsMulticastPrintStr()

Description Prints statistics

Prototype

```
void GMACStatsMulticastPrintStr(GMACHandle *H, char
*Str, FILE *f)
```

Arguments	`GMACH`	GMAC handle
	`Str`	String to print
	`f`	Stream to print to

4.2.13 GMACStatsPrintSqn()

Description Prints statistics for the next expected sequence

Prototype `void GMACStatsPrintSqn(GMACHandle *H, FILE *f)`

Arguments	`GMACH`	GMAC handle
	`f`	Stream to print to

4.2.14 GMACStatsPrint()

Description Prints statistics

Prototype `void GMACStatsPrint(GMACHandle *H, int Queue, FILE *f)`

Arguments	`GMACH`	GMAC handle
	`f`	Stream to print to

4.2.15 GMACStatsDump()

Description Dumps the structure

Prototype `void GMACStatsDump(GMACStats *Stats, FILE *f)`

Arguments	`Stats`	Pointer to where statistics will be passed
	`f`	Stream to print to

5. Latency Measurement

5.1 Description

All GMAC messages contain a hardware timestamp which is applied as the last byte arrived on the accelerator's card Ethernet port. This timestamp is propagated in **GMACPacketHeader** structure. The user application can query the latency on messages at any time via two latency API functions:

- **Real-time latency function**: `GMACLatencyRealTime()` returns the number of timestamps and the total latency as reported by the accelerator card every 0.5 seconds, for the last 0.5 second window;
- **Total latency function**: `GMACLatencyTotal()` returns the sum of all latencies since the last reset, or application start. This type of request requires a register read from the accelerator card, which blocks access to it for a certain time and should therefore should be used sparingly.

The following figure illustrates how latency is computed.

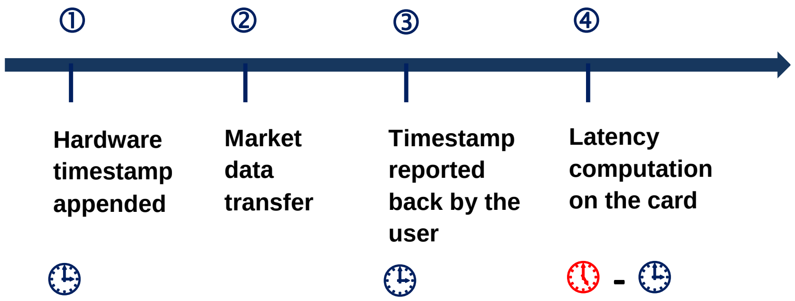
①
②
③
④
Hardware timestamp appended
Market data transfer
Timestamp reported back by the user
Latency computation on the card
-

Figure 1 – Latency

① Current hardware timestamp is appended to the packet when the end of the packet is received.

② Market data is transferred to the GMAC software side.

③ Timestamp is reported back to the card. This must be done by the user via `GMACLatencyTick()`.

④ Reported timestamp is evaluated against the current hardware timestamp.

Latency measurement is very light-weight and should not influence performance. Note that the computed latency also includes the time it takes to transfer the timestamp back to the hardware. Therefore the real latency is actually slightly smaller than what's reported.

5.2 Functions

5.2.1 GMACLatencyRealTime()

Description Returns the latency metrics, as reported by the hardware to the GMAC software-side every 0.5 seconds. The values it returns are for the most recent measured 0.5-second window. Calling the function multiple times within a 0.5-second window will therefore result in the same values being returned. This function is fast to return, so can be called in performance-sensitive code.

Note:

> The metrics returned depend on the user reporting latency ticks back to the hardware using `GMACLatencyTick()`.

Prototype

```
GMACStatus GMACLatencyRealtime(GMACHandle *GMACH,
int Queue, char *Plugin, uint64_t *TSCount,
uint64_t *Latency)
```

Arguments

Arguments	`GMACH`	GMAC handle
	`Queue`	Queue number
	`Plugin`	Name of the ADE plugin

`TSCount`	Number of reported timestamps
`LatencyTotal`	Total of all reported latencies in ns

5.2.2 GMACLatencyTotal()

Description

Returns the latency metrics for an entire GMAC session: i.e. since the last reset or since the application start. The function accesses registers on the accelerator card, which is an expensive operation. It should therefore not be called excessively, and it should not be called from performance-sensitive code. The function can be called safely from any thread, but the caller should ensure that GMAC does not close during the call (again because it accesses registers on the card).

Note:

> The metrics returned depend on the user reporting latency ticks back to the hardware using `GMACLatencyTick()`.

Prototype

```
GMACStatus GMACLatencyTotal(GMACHandle *GMACH, int
Queue, char *Plugin, uint64_t *TSCount, uint64_t
*LatencyTotal, int Reset)
```

Arguments

`GMACH`	GMAC handle
`Queue`	Queue number
`Plugin`	Name of the ADE plugin
`TSCount`	Number of reported timestamps
`LatencyTotal`	Total of all reported latencies in ns
`Reset`	If set to non-zero, the counters will be reset by this read

5.2.3 GMACLatencyTick()

Description

Reports a timestamp back to the accelerator card. The accelerator card can use this to provide latency metrics, see `GMACLatencyRealTime()` and `GMACLatencyTotal()`.

Prototype

```
void GMACLatencyTick(GMACHandle *GMACH,
GMACMulticastID MulticastID, uint32_t Timestamp)
```

Arguments

`GMACH`	GMAC handle
`MulticastID`	Multicast ID of the received data
`Timestamp`	Received timestamp from **GMACPacketHeader**

6. Snapshot Support

There are three levels of snapshot support in GMAC:

Level 0 - L0 Manual snapshot support for non-request based feeds handled by using the following functions:

- `GMACChannelEnable()`
- `GMACChannelEnableByType()`
- `GMACMulticastSync()`
- `GMACMulticastBuffer()`

In addition the user needs to process all related GMAC messages manually.

The level L0 is usually supported for feeds where snapshot channels are coming in cycles.

Level 1 - L1 Manual snapshot support for request based feeds handled by using the following functions:

- `GMACMulticastSync()`
- `GMACMulticastBuffer()`
- `GMACMulticastRefreshRequest()`
- `GMACMulticastRefreshComplete()`

In addition the user needs to process all related GMAC messages manually.

The level L1 is usually supported for TCP based feeds.

Level 2 - L2 Semi-automatic snapshot support handled by using the following functions:

- `GMACMulticastSync()`
- `GMACMulticastBuffer()`
- `GMACMulticastSnapshotRequest()`
- `GMACMulticastSnapshotProcess()`

The user needs to process all related GMAC messages manually. However the GMAC messages to be processed are selected by `GMACMulticastSnapshotProcess()`, so the user does not need to monitor himself. This level also hides some of the processing complexity as it provides the indication of snapshot request completion, handles error states, manages timeouts and provides the synchronisation points.

Refer to the Market Data-Market Support Matrix documentation for more details on the available snapshot level per plugin.

The second level is only supported by GMAC for selected markets. The user will get an error if he tries to use this level on the non-supported market.

Note that there are markets where the level L1 support is the only available snapshot support. It is planned to migrate all the snapshot support under the level L2.

6.1 Description

The level L2 snapshot support is based on the request/process basis per live multicast. The request is always associated with some multicast. Also there can be only one pending request per multicast.

The live multicast is a multicast which is of type GMAC_MULTICAST_TYPE_LIVE. It usually contains just incremental live data. This does not mean that the live multicast cannot contain snapshot data.

The user is required to put a snapshot request by using `GMACMulticastSnapshotRequest()` on the live multicast and then make the data from the GMAC queue available to `GMACMulticastSnapshotProcess()`.

On some markets there are multiple live multicasts. If possible GMAC tries to make the snapshot requests independent on different live multicasts on the same market. However this is not always possible; therefore the user must be aware that the request on the live multicast might be rejected in the beginning if there is already a pending request from the past on some other live multicast on the same market. For instance on the markets ICE and LIFFE XDP it is not possible to have simultaneous request on different live multicasts.

The timeout mechanism of the snapshot request is co-operated internally using the scheduler thread. Therefore it is secured that the timeout occurs also if no data comes from the network at all.

6.2 Level 1 Snapshot Support Functions

6.2.1 GMACMulticastRefreshRequest()

Description Enables a refresh channel

Prototype `GMACStatus GMACMulticastRefreshRequest(GMACHandle *GMACH, GMACRefreshHandle *GRH)`

Arguments	`GMACH`	GMAC handle
	`GRH`	GMAC refresh handle

6.2.2 GMACMulticastRefreshComplete()

Description Disables a refresh channel

Prototype `GMACStatus GMACMulticastRefreshComplete(GMACHandle *GMACH, GMACRefreshHandle *GRH)`

Arguments	`GMACH`	GMAC handle
	`GRH`	GMAC refresh handle

6.3 Level 2 Snapshot Support Functions

6.3.1 GMACMulticastSnapshotRequest()

Description When the user is interested to receive a snapshot for certain live multicast he can emit such a snapshot request with this function. The LiveSequence number is the last packet sequence number seen on the live multicast at the time of the request. The snapshot request is completed when the `GMACMulticastSnapshotProcess()` function returns CycleComplete flag set.

Prototype `GMACStatus GMACMulticastSnapshotRequest(GMACHandle *GMACH, GMACMulticastID MulticastID, GMACLocalName Symbol, uint64_t LiveSequence, uint32_t TimeoutSec)`

Arguments	`GMACH`	GMAC handle
	`MulticastID`	The live multicast ID for which a snapshot is needed
	`Symbol`	The name of the instrument of interest, or “” (meaning ’all instruments’). Note that currently per-instrument snapshots are not supported so “” (or GMAC_SNAPSHOT_ALL_SYMBOLS) must be used here.
	`LiveSequence`	The last packet sequence number seen on the multicast in question. The snapshot mechanism uses this to ensure that it only processes fresh updates and discards stale ones.
	`TimeoutSec`	Timeout in seconds, after which the snapshot mechanism should give up waiting for snapshot data. This is useful for feeds where snapshots are requested over TCP but received over UDP, where a failure to receive requested data may not be accompanied by a useful error/rejection code. This timeout should be chosen carefully depending on

the market; for example snapshot refresh updates for lightly-traded instruments on ICE can take 2minutes.

6.3.2 GMACMulticastSnapshotProcess()

Description When the user has at least one pending snapshot request he can start processing the received data from `GMACQueueRead()` with this function. If there is no pending snapshot this function has no effect.

When the `GMACMulticastSnapshotProcess()` returns the status GMAC_STATUS_OK the user needs to process the `SnapshotData` structure returned. First of all the user needs to process all the messages in the Messages[]. After that the user needs to check the CycleComplete flag. If the flag is set the user can determine the result of the snapshot request in the CycleStatus field.

Prototype

```
GMACStatus GMACMulticastSnapshotProcess(GMACHandle
*GMACH, GMACPacketHeader *Header, GMACSnapshotData
**SnapshotData)
```

Arguments

`GMACH`	GMAC handle
`Header`	GMAC packet header
`SnapshotData`	Returned snapshot data

6.4 Snapshot Request Guide

1. The live multicast where the user wants to emit the refresh has to be identified. There is a need to receive at least one GMAC packet on that multicast.

   The Header is the GMACPacketHeader associated with this packet:

   LiveMID = Header->MulticastID;

   Then the user needs to begin the buffering of the data on that multicast:

   GMACMulticastBuffer(GMACH, LiveMID);

2. The user emits the snapshot request:

   GMACMulticastSnapshotRequest(GMACH, LiveMID, GMAC_SNAPSHOT_ALL_SYMBOLS, /* LiveSequence */ Header->Sequence, /* Snapshot request timeout in seconds */ 300);

   For instance the common timeout for ICE market is ~ 2 minutes.

3. The data received from `GMACQueueRead()` is always passed to the process function:

   GMACMulticastSnapshotProcess(GMACH, ReceivedHeader, &SnapshotData);

   The status code is returned.

4. When the status code is set to GMAC_STATUS_OK the user examines the SnapshotData structure. First of all the snapshot messages selected by the processing function has to be processed. Afterwards the CycleComplete flag has to be checked for request completion. Once the request is completed the user determines the resulting state of the snapshot request in the CycleStatus field.

5. Once the request is complete and successful the user gets SyncSequence. This sequence number is used to synchronise the live multicast i.e. un-buffer the multicast from step 1. All the packets older than the synchronisation point are automatically dropped by GMAC.

   GMACMulticastSync(GMACH, LiveMID, SnapshotData->SyncSequence);

6. The request is completed.

7. Format

See the header files for more details.

7.1 Variables

The following table provides the values of some GMAC variables:

Variable	Value	Description
GMAC_DECIMAL_PLACES	**8**	Number of decimals places
GMAC_UNIT_MAX_LENGTH	**10**	Units in which the quantity is described
GMAC_INSTRUMENT_MAX_LENGTH	**70**	Maximum length of the instrument identifier
GMAC_UNDERLYINGID_MAX_LENGTH	**70**	Maximum length of the underlying
SEQUENCE_UNDEFINED	**~((uint64_t)0)**	Undefined start sequence number
SEQUENCE_RESET	**~((uint64_t)0)-1**	Reset sequence number
GMAC_LOCAL_NAME_LENGTH	**32**	Market local instrument name maximum length
GMAC_EISIN_LENGTH	**36**	Extended ISIN length
GMAC_MAX_EXTENSION_STRING		String extension maximum length
GMAC_PRICE_UNDEFINED	**~((uint64_t)0)**	Undefined price value
GMAC_SIZE_UNDEFINED	**~((uint32_t)0)**	Undefined size value
GMAC_SNAPSHOT_ALL_SYMBOLS	**“ “**	Snapshot is requested for all instruments
GMAC_MAX_CURRENCY_LENGTH	**4**	Maximum currency length

7.2 Enumerations

The following table provides the possible values for some GMAC lists:

Enumeration	Values	Description
GMACTradingStatus	**TSTATUS_UNKNOWN**	Unknown or invalid
	TSTATUS_CLOSE	Closed
	TSTATUS_SUSPEND	Suspended or halted
	TSTATUS_QUOTE	Quotation only
	TSTATUS_EXTEND	Extended market hours
	TSTATUS_OPEN	Open
GMACMulticastType	**GMAC_MULTICAST_TYPE_UNKNOWN**	Unknown
	GMAC_MULTICAST_TYPE_LIVE	Live
	GMAC_MULTICAST_TYPE_SNAPSHOT	Snapshot
	GMAC_MULTICAST_TYPE_DATAREF	Referential data
	GMAC_MULTICAST_TYPE_REREQUEST	Data re-request
GMACStatus	**GMAC_STATUS_OK**	Success
	GMAC_STATUS_INIT_ERROR	Initialization error
	GMAC_STATUS_CONTROL	Control data return
	GMAC_STATUS_BUFFERING	Data are being buffered on user request. Data pointer set to last packet received.
	GMAC_STATUS_BUFFERING_ERROR	User requested buffering failed, did not fit
	GMAC_STATUS_RECOVERY_BUFFERING	GMAC is buffering data within the recovery mechanism to fill the gap
	GMAC_STATUS_PARAM_ERROR	Parameter error
	GMAC_STATUS_CONFIG_ERROR	Configuration file error
	GMAC_STATUS_PARTIALLY_INITIALIZED	Partially initialized
	GMAC_STATUS_NOT_IMPLEMENTED	Not implemented

	GMAC_STATUS_CONNECTION_ERROR	Connection error
	GMAC_STATUS_PAYLOAD_ERROR	Payload error
	GMAC_STATUS_RUNTIME_ERROR	Runtime error
	GMAC_STATUS_RELEASE_FIRST	GMAC is waiting for `GMACQueueRelease()` to be called
	GMAC_STATUS_RECOVERY_FAILDE	Recovery failed on the channel specified in the packet header
	GMAC_STATUS_NOT_FOUND	Market name translation is not possible
	GMAC_STATUS_BUFFER_FULL	Buffer pool for the queue reached the limit in the configuration file. All data is discarded in the reorder/recovery buffers and all corresponding channel expected sequence numbers are set to UNDEFINED.
	GMAC_STATUS_RECONNECT	Returned by GMAC when the TCP connection has been lost. Also informing user that a new connection is in asynchronous opening state. When the connection becomes available the GMAC will start to decode and provide market data. If the GMAC is unable to successfully create a new connection within the timeout the GMAC will return this code again.
	GMAC_STATUS_NO_CHANNELS	ADE reports no multicasts in queue for `GMACQueueRead()`
	GMAC_STATUS_NODATA	No data
	GMAC_STATUS_ADE_ERROR	ADE error
GMACSnapshotStatus	**GMAC_SNAPSHOT_OK**	Success
	GMAC_SNAPSHOT_TIMEOUT	Snapshot has timed out
	GMAC_SNAPSHOT_NOT_FOUND	The requested instrument was not found
	GMAC_SNAPSHOT_CYCLE_GAP	There was a gap in the snapshot message cycle
	GMAC_SNAPSHOT_MESSAGES_LIMIT_REACHED	Internal limit of messages reached
GMACFilteringMode	**GMAC_FILTERING_NOT_SUPPORTED**	Filtering is not supported on the market
	GMAC_FILTERING_NOT_ENABLED	Filtering is not enabled
	GMAC_FILTERING_SYMBOL	Filtering is enabled and symbol names need to be provided
	GMAC_FILTERING_INDEX	Filtering is enabled and indexes need to be provided
ADEArbType	**ADE_ARB_NONE**	No arbitration
	ADE_ARB_REDUNDANCY	Redundancy arbitration

ADE_ARB_LAG	Lag arbitration

7.3 Data Structures

7.3.1 Packet Header

Description The packet header is prefixed to the data returned by `GMACQueueRead()`. It contains information about data as follows:

- Messages and Sequence can be used to detect gaps in the incoming data

Note:

> GMAC generates data without gaps if the recovery feature is enabled

- TranscodedMessages and/or Length can be used to determine how many normalized messages there are
- MulticastID gives information about which multicast the data come from

Heartbeats can be identified as follows:

- Messages is always set to 0
- And if TranscodedMessages is set to 1, the first (and only) message will not be a Multicast Timeout message (type GMAC_TYPE_MULTICAST_TIMEOUT)

Structure **GMACPacketHeader**

The structure description is as follows:

Field	Type	Description
MulticastID	GMACMulticastID	Multicast ID
OrigMulticastID	GMACMulticastID	Original Multicast ID. Set when packet was redirected from another Multicast ID. Sometimes used for recovery.
HWTimestamp	unit32_t	Time stamp from HW clock, taken when the last byte arrives from the wire
HWTimestampValid	unit8_t	Set to 1 when HWTimestamp is valid, otherwise set to 0.

Field	Type	Description
		Some messages do not have HWTimestamp, e.g. when the message was received from the recovery server.
Live	unit8_t	Flags whether a packet originates from a live interface and has been delivered directly and without buffering. This flag would be set to 0 for buffered packets (e.g. packets waiting for a retransmission or a snapshot/refresh cycle), or for packets received as a response to a retransmission or snapshot/refresh request.
Tainted	unit8_t	Set to 1 when the GMAC core passes a gap to the user. Remains set until the multicast is closed.
TranscodingType	unit8_t	Encoding type used for this feed, see `GMACChannelGetTranscodingType()`
Messages	unit32_t	Original number of messages in the packet (for next packet sequence checking)
TranscodedMessages	unit32_t	Number of normalized messages
Length	unit32_t	Total length of the data including this header
Sequence	unit64_t	Original sequence number of the first message in the packet
NextSequence	unit64_t	The next expected sequence number
SWTimestampIn	unit64_t	The time the message is read from the queue
SWTimestampOut	unit64_t	The time when GMAC finished processing the message
RawExchangeChannelID	unit64_t	Actual identifier for this multicast provided by the exchange. Initialized with -1.

7.3.2 Refresh Handle

Description This handle represents a refresh channel request.

Note:

> The MulticastID is for the live incremental channel associated with the snapshot/refresh channel and NOT the refresh channel itself.

Structure **GMACRefreshHandle**

The structure description is as follows:

Field	Type	Description
MulticastID	GMACMulticastID	Multicast ID
InstrumentID	GMACLocalName	Instrument name in the format used by the feed. It can be a segment or a sector name as well. If less than GMAC_LOCAL_NAME_LENGTH characters, C-standard zero byte is used at the end (null terminated string).

7.3.3 Default Market Status

Description Default status values for Market, Sector, Segment, Channel and Instrument. A client can use these as the initial values until they are known. Values for Sector, Segment and Channel are 'Open' where these levels are not used.

It is returned by `GMACGetDefaultStatus()`.

Structure **GMACDefaultMarketStatus**

The structure description is as follows:

Field	Type	Description
MarketStatus	GMACTradingStatus	Default (initial) status for the Market.
SectorStatus	GMACTradingStatus	Default (initial) status for each Sector.
SegmentStatus	GMACTradingStatus	Default (initial) status for each Segment.
ChannelStatus	GMACTradingStatus	Default (initial) status for each Channel.
InstrumentStatus	GMACTradingStatus	Default (initial) status for each Instrument.

7.3.4 Dynamic Multicast Descriptor

Description The GMAC Dynamic Multicast Descriptor is used to hold several values for a multicast to be opened dynamically (after the GMAC Queue has been opened).

It is used by `GMACOpenMulticast()`.

It is initialized to default values by `GMACInitDynamicMcDesc()`.

Structure **GMACDynamicMcDesc**

The structure description is as follows:

Field	Type	Description
OriginalChar	char	Arbitrage ‘A’ Line A ‘B’ Line B ‘-‘ Default
OriginalArbMode	ADEArbType	Arbitrage mode
Type	GMACMulticastType	Multicast type Default is GMAC_MULTICAST_TYPE_LIVE
Plugin	char	ADE plugin name Default is “lldt”
StartSeq	uint64_t	Initial sequence number Default is SEQUENCE_UNDEFINED

7.3.5 Snapshot Data

Description This structure encapsulates GMAC packet data to return matched snapshot data.

It is used by `GMACMulticastSnapshotProcess()`.

Structure **GMACSnapshotData**

The structure description is as follows:

Field	Type	Description
Header	GMACPacketHeader	Pointer to the GMAC packet header passed to the `GMACMulticastSnapshotProcess()`
MessageCount	int	Size of the Messages[] array returned in this structure
CycleComplete	int	When set the user request has been completed. The resulting status of the user request has to be determined from the field CycleStatus.
SyncSequence	uint64_t	It is valid only when the CycleComplete field is set and the CycleStatus is set to GMAC_SNAPSHOT_OK. This is a packet synchronization sequence number for the live multicast. Please note that for some feeds this is in fact a minimum synchronisation sequence number across all the snapshot messages associated with the request. This number can be directly used in the `GMACMulticastSync()`. However the user needs to monitor GMACMessageSnapshotUpdateV3.LastMsgSeqNum for the instrument synchronisation. This number might be the same or different. On some markets this number varies across the instrument list for the snapshot request. Note that the SyncSequence sequence number is always higher than or equal to LiveSequence + 1 in the user snapshot request.
CycleStatus	GMACSnapshotStatus	GMAC_SNAPSHOT_OK The snapshot request has been successful GMAC_SNAPSHOT_TIMEOUT The user defined timeout has expired GMAC_SNAPSHOT_CYCLE_GAP There was a gap detected in the snapshot cycle GMAC_SNAPSHOT_NOT_FOUND The specified LiveSequence sequence number in the user request is too new (high), i.e. the snapshot message just received has the synchronisation sequence number lower than or equal to the LiveSequence number. The user is expected to repeat the request with the same LiveSequence number. GMAC_SNAPSHOT_MESSAGES_LIMIT_REACHED The number of messages in the snapshot exceeds the size reserved for Messages[]. In such a case, CycleComplete will be set, but the user will need to consider the snapshot as incomplete as overflowed messages won't be part of Messages[].
***Messages[]**	GMACMessageHeader	The array of pointers to all the messages associated with the user request. Currently there is no support for per-instrument requests. Therefore the Messages[] are always pointing to all the messages in the GMAC packet. This array has an internal hard limit.

7.3.6 Multicast Mapping

Description It is returned by `GMACGetMulticastMappingTable()`.

Structure **GMACMulticastMapping**

The structure description is as follows:

Field	Type	Description
MappingTableSize	uint32_t	Size of the mapping table in bytes
Count	uint32_t	Number of nodes populated
MappingTable[]	GMACMulticastMappingNode	Table where the multicast map will be stored

7.3.7 Multicast Mapping Node

Description This is the format of the mapping table returned by `GMACGetMulticastMappingTable()`.

Structure **GMACMulticastMappingNode**

The structure description is as follows:

Field	Type	Description
MulticastID	GMACMulticastID	Multicast ID
ChannelID	GMACChannelID	Channel ID
IP	uint32_t	IP address of the multicast ID
Port	uint16_t	Port for the multicast ID
Plugin	char [32]	The Plugin used by the multicast ID

7.3.8 Stats

Description It is returned by `GMACStatsMulticast()`, `GMACStatsChannel()`, `GMACStatsQueueGlobal()` and `GMACStatsDump()`.

Structure **GMACStats**

The structure description is as follows:

Field	Type	Description
MulticastID	GMACMulticastID	Multicast ID
RedirectedToID	GMACMulticastID	Set to ~0 for no redirection. Otherwise the destination multicast ID
Type	GMACMulticastType	Multicast type
Buffered	int	Set to 1 when the multicast is currently being buffered. Otherwise set to ~0.
Disabled	int	Set to 1 when the multicast is disabled. Otherwise set to ~0.
ChannelID	GMACChannelID	Channel ID
LocalChannelID	int	Channel ID number private for GMAC plug-in
LocalChannelChar	char	Set to ’A’ or ’B’ when arbitration is enabled. Otherwise ’ ’ or ’-’
ADEPlugin	const char	Name of plugin, points back to Multicast
Tainted	int	Set to non-zero value when the GMAC detected any gap or problem in the data (this flag has unspecified behavior).
FilteringModuleInUse	int	Set to non-zero when filtering is enabled in the configuration file and is supported by the plugin
FilteringEnabled	int	Set to non-zero when filtering is enabled for the channel and is running
PacketsReceivedTotal	uint64_t	Number of packets received by GMAC on its inputs
PacketsReceivedADE	uint64_t	Number of packets received by GMAC from ADE

Field	Type	Description
PacketsReceivedTimeout	uint64_t	Number of packets generated by GMAC as timeouts
PacketsReceivedRedirected	uint64_t	Number of packets redirected to a different multicast
PacketsDelivered	uint64_t	Number of packets delivered by GMAC to user.
PacketsInReorderQueue	uint64_t	Number of packets stored currently in GMAC reorder queue
PacketsReceivedControl	uint64_t	Number of packets received from the control queue
GapsDetected	uint64_t	Number of gaps GMAC detected after the timeout for reordering expired
GapsPassed	uint64_t	Number of gaps GMAC passed to the user
PacketsDroppedPayloadError	uint64_t	Number of packets which couldn't be normalized
PacketsDroppedInTranscoder	uint64_t	Number of packets dropped by the GMAC plug-in (probably because the sequence number would confuse).
PacketsDroppedSimulated	uint64_t	Number of packets GMAC dropped intentionally on user request
PacketsDroppedAsDuplicates	uint64_t	Number of packets received with lower sequence number than expected
PacketsDroppedInReorderQueue	uint64_t	Number of packets dropped from the reorder queue (reset, buffer full, sync, duplicate, strange sequence)
ExpectedSequenceNumber	uint64_t	Expected sequence number
***RawPacketStats**	ADEStats	Statistics gathered by low level input library

7.3.9 ADE Statistics

Description ADE staticstics

Structure **ADEStats**

The structure description is as follows:

Field	Type	Description
BytesRead	uint64_t	Total bytes read on the channel

Field	Type	Description
PacketsRead	uint64_t	Total packets read ion the channel
BytesWritten	uint64_t	Total bytes written on the channel
PacketsWritten	uint64_t	Total packets written on the channel

7.4 Argument Types

The following table provides the argument types description:

Enumeration	Description
GMACHandle	Opaque structure
GMACMulticastID	uint32_t
GMACChannelID	uint32_t
GMACPriceFactor	Opaque structure
GMACErrorHandler	`GMACStatus (*GMACErrorHandler)(GMACHandle *GMACH, GMACStatus StatusCode, const char *Function, char *Msg)`
ADEErrorHandler	Opaque structure
Config	See the Configuration Parser API Reference Guide and the GMAC Configuration Guide.
GMACLocalName	char GMACLocalName [GMAC_LOCAL_NAME_LENGTH]
	Instrument name in the format used by the feed.
	If less than GMAC_LOCAL_NAME_LENGTH characters C-standard zero byte is used at the end (null terminated string)
GMACEISIN	char GMACEISIN [GMAC_EISIN_LENGTH]
	The GMAC "Extended ISIN" is a Celoxica-specific code and it provides a global naming scheme for instruments across all the supported markets. The codes are prefixed by 4 characters. The aim is to have a single type of identifier for the instruments, even if there is no real ISIN code.
	The prefixes available are: • **ISIN** real ISIN code follows • **SEDO** follows 5 spaces and SEDOL code • **MARC** ARCA Equities instruments • **MNAO** ARCA Options instruments • **MBAT** BATS instruments • **MBIT** LSE Borsa Italiana instruments • **MCHI** Chi-X instruments

	• **MCME** CME instruments • **MDIR** Direct Edge instruments • **MICE** ICE instruments • **MISE** ISE instruments • **MITC** ITCH instruments • **MLIF** Liffe XDP instruments • **MLSE** LSE instruments • **MNYS** NYSE instruments • **MOPR** OPRA instruments • **MPHL** PHLX instruments • **MTSX** TSX instruments • **MUQD** UQDF instruments • **MUTD** UTDF instruments
GMACMessageHeader	See GMAC V1 Messages Specifications and GMAC V3 Messages Specifications
GMACMarketNames	Char GMACMarketNames [MAX_MARKETS]

8. Shipped Examples

Every GMAC software release is shipped with an API Prototype example in /Examples/GMAC/example/ showing most of GMAC's functionality. 'Example' aims to be easy to understand and to illustrate the use of the GMAC API, and so is written with simplicity rather than speed in mind. Hence it uses operations that can degrade performance and that would therefore normally be avoided in production code, for instance locking and printing to the screen. It is possible to dynamically switch multiple modules on and off by changing the `AddModules()` function in example.c as appropriate.

8.1 Features

The example has a number of modules that can be switched on with command line options. These correspond to the distinct functional elements available in the API. To select the modules to enable, use –e followed by the appropriate module name:

1. '**-e dump**' enables the Dump module.
2. '**-e vwap**' enables the Vwap module.
3. '**-e snapshot**' enables the Snapshot module.
4. '**-e tob**' enables the Tob module.
5. '**-e sequencecheck**' enables the SequenceCheck module.
6. '**-e stats**' enables the Stats module.
7. '**-e index**' enables the Index module.
8. '**-e hwfiltering**' enables the HWFiltering module.

'Example' illustrates the use of multiple functional elements in GMAC including:

1. Basic initialization and reading of queue(s), error handling: example.c and error_handler.c
2. Printing normalized messages in text format: dump.c
3. Listing through messages and semantically using the data: vwap.c, tob.c and index.c
4. Following trades and computing VWAP (Volume-Weighted Average Price): vwap.c
5. Building an order book from GMAC messages: tob.c
6. Using snapshot refresh channels for mid-day start: index.c
7. Printing the symbol-to-index mapping extracted from the messages: index.c
8. Printing the statistics on the fly: example.c and statistics.c
9. Detection and reporting of gaps: statistics.c and sequence_check.c
10. Examining latency information on the fly: example.c
11. Hardware filtering: hwfiltering.c

8.2 Building the Example

To build the example:

1. If you haven't done so already, run 'copy_celoxica_examples' to copy the examples to your local directory
2. cd ~/celoxica-examples/GMAC/example/
3. To build the example simply run 'make'.

   This will build the example for all the feed adapters supported by your GMAC library.

8.3 Options

The example must be run with a set of command-line options, the minimum being '**-f <config file>**'.

Example configuration files for the supported markets are stored in directory ~/celoxica-examples/GMAC.

Refer to the ./example -h to get the available options.

The options for each module must follow the module name as shown:

./example [some general options] -e <module_name> [some module options] -e <module_name> [some module options] …