# ASAM MCD-3 MC

One of the major tasks of ECU development is the calibration and test of control strategies, i.e. tuning of parameters and the recording of internal variables during the runtime of the ECU. These tasks can be done via various busses, bus protocols or proprietary plug-on devices between the ECU and the calibration system. Those interfaces are technology-dependent and vendor-specific. In order to provide uniform access to an ECU, calibration tools typically have an MC-server that other tools can use to connect to an ECU without having to deal with different interface technologies. The main objective of the ASAM MCD-3 MC standard is to specify the functions of an MC-server and to provide a remote control interface for client applications to the MC-server via an object-oriented API. The functions of an MC-server are primarily to provide measurement and calibration services to clients. The standard allows that any client application, such as test automation systems or automated calibration systems, can connect via the MC-server to an ECU and carry out typical measurement and calibration tasks. Several client applications can be connected to one MC-server and have access to one ECU in parallel.
To be able to access data on an ECU, the MC-server reads an A2L data description file (according to ASAM MCD-2 MC), which contains a description of available calibration parameters (CHARACTERISTICS) and measurement variables (MEASUREMENTS). The MC-server then makes services available to access this data. Methods for read- and write-access to the calibration parameters of various types such as scalars, arrays, strings and look-up tables can be used. Measurement access methods are available via Collector, Watcher and Recorder services. The Collector acquires the values of measurement variables or calibration parameters with a common rate over a defined period of time (continuous data acquisition) and sends the data to the client applications at the same time. The Recorder is used to manage high bandwidth measurements, when synchronous data transfer to the client is not possible. The Recorder stores the data locally and makes it available to clients at a later time. A Watcher is a service which continuously monitors measurement variables or calibration parameters and triggers events if a predefined condition is met. Multiple Watchers can be defined to monitor multiple objects at the same time. The Watcher may be used to start and stop Collectors or Recorders.
The standard is used for calibration and measurement purposes in development, testing and production of ECUs. ASAM MCD-3 MC currently coexists with the older ASAM ASAP3 standard, which is dependent on specific interfaces (RS232, Ethernet) and is still widely used. The ASAM MCD-3 MC API is specified in an object-oriented but technology-independent UML model and mapped to DCOM. This configuration allows easily addition of new programming language mappings to the standard without having to change the core of the standard.

**Datasheet**

| | |
|---|---|
| **Title** | Application Programming Interface for Measurement and Calibration Server |
| **Category** | AE |
| **Current Version** | 3.0.0 |
| **Release Date** | 23.09.2011 |
| **Download** | ASAM MCD-3 MC V3.0.0 |
| **Application Areas** | • Test stand automation<br>• Automated calibration<br>• Data logging |
| **Specification Content** | • Client-server API<br>• Technology references for COM-IDL<br>• MC-server architecture |
| **File Formats** | - |

# History

ASAM MCD-3 MC started life in the early 1990s as a company-specific specification from AVL List that defines a serial interface protocol based upon RS232 for test stand automation systems. The specification was used as the foundation to create the first public standard under the name ASAP3. In subsequent years, the standard was further developed and TCP/IP was added as a second supported protocol.

By the end of the 1990s, ASAP3 was given to ASAM and renamed to ASAM MCD-3. The members of ASAM created new strategic guidelines for further developing the standard. The standard was promoted to a higher level, now describing a hardware-independent software interface rather than a hardware-dependent protocol. The drafts of the two specifications ASAM MCD-3 D and ASAM MCD-3 MC have been merged into one standard. Data and functions were described via object-oriented models. The new standard supported significantly more use-cases for test stand automation and automated calibration than before. A library implementation from Vector Informatik was used as the template for the first version of this fundamental re-design of the standard. The standard was released as version 1.0 in 2003. The standard was further matured and revised with the help of prototype implementations and cross-testing events.

With version 2.0 and 2.1, the Watcher- and Recorder-services have been added. Since the requirements for the MC part and the diagnostics part of the standard increasingly diverged, ASAM decided to split the standard into two independent standards. Since version 3.0, the name of the standard is now ASAM MCD-3 MC. This version improves the initialization-time of MC-servers and the connection and release of client application.

Main contributors to the standard are AVL List GmbH, Robert Bosch GmbH, BMW AG, Continental Automotive GmbH, D2T, Daimler AG, dSPACE GmbH, ETAS GmbH, HORIBA GmbH, imc meßsysteme GmbH, M&K Mess- und Kommunikationstechnik GmbH, Porsche AG, Vector Informatik GmbH and Visu-IT! GmbH.

# Motivation

The main motivation for ASAM MCD-3 MC and its predecessors is to provide standardized and abstracted access to ECU calibration parameters and measurement data. The standard uses calibration tools for this purpose, which have direct access to such data. The standard adds a server module to calibration tools with a specified software interface. The interface completely decouples client applications from hardware-, bus-, protocol- or vendor-specific properties of subsequent components of the tool chain.

Since the MC-server is implemented as a software interface, the server interface is directly available in programming languages and scripts that can be written by end users. MC-servers according to this standard provide ECU data in their physical representation format, i.e. 50km/h instead of 0x3C. This has the advantage that programs for test automation or automated calibration are independent from the ECU software. They can be easily ported to different ECUs and different tools.

# Application Areas

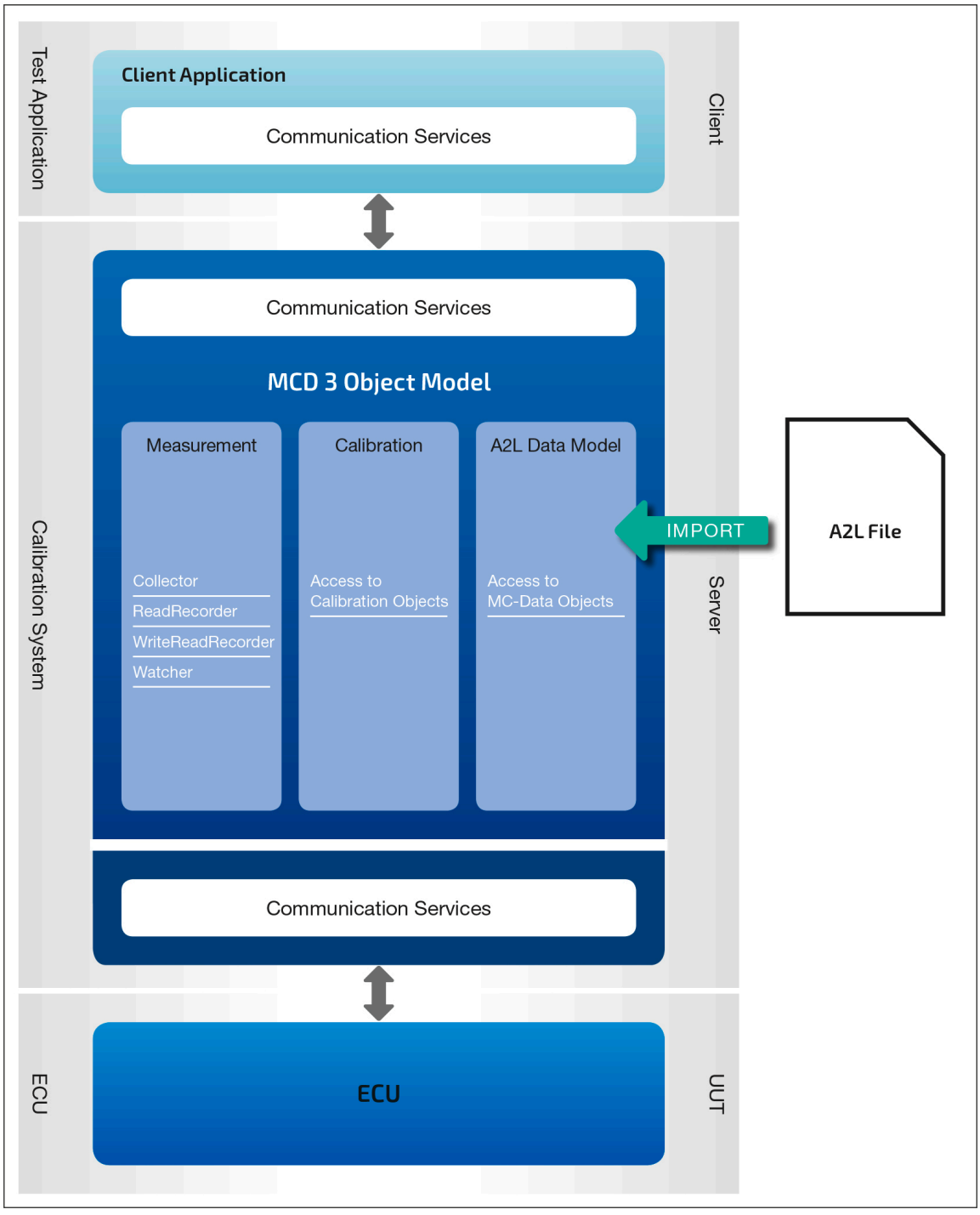
The primary application area is test stand automation. The execution of tests for ECUs, vehicle subcomponents, engines and complete vehicles is mostly automated in today's test labs. Tool setup, configuration, programming and debugging could consume a significant portion of the available time for testing and the utilization of expensive test stands. ASAM MCD-3 MC is one provision to significantly reduce this time.

Some further areas of use for the standard is in HIL testing (although now increasingly replaced by ASAM XIL) and automated calibration. The latter is actually an advanced use-case for test stand automation. This use-case is still not often seen in the industry, since many calibration tasks are deemed to be too complex for automated execution. However, the standard provides the necessary features to support this application area.

# Technical Content

## Architecture of an MC-Server

The MC-Server is part of a calibration system or modular vehicle communication system. Such systems have access to measurement variables and calibration parameters on an ECU. The task of the MC-Server is to collect data from the ECU while it is running and to make the data available to client applications via a programmatic interface. The functions of data collection and the interface to client applications are standardized through ASAM MCD-3 MC as an object-oriented API. To carry out the functions, the server consist of three functional blocks:




**Functional Architecture of an MC-Server**

- Database: classes to access the meta data of measurement and calibration objects
- Measurement: classes to acquire measurements
- Calibration: classes to set and acquire calibration parameters

Specifically, the measurement classes can be further subdivided into classes that implement specific functions:

- Collector: data logging and synchronous transfer to clients
- Recorder: data-logging and asynchronous transfer to clients
- Watcher: triggering of data logging

Collectors and Recorders acquire data according to a data acquisition list. A data acquisition list is defined in ASAM MCD-3 MC via a "collection". In general, collections in the sense of this standard are objects of the same kind that shall be logically grouped together. In this respect, a data acquisition list is considered to be a collection.

The data is acquired through "logical links". The standard defines a logical link as one physical communication line to one ECU that uses a specific interface and protocol. Logical links contain all Collectors defined for the ECU and all calibration objects available on the ECU.

Data bases, logical links, Recorders and Watchers are put together in a "project". All references to data objects within one project can be resolved. The MC-server may hold the definition of several projects, but can only work on one active project at the same time. Consequently, all clients connected to one MC-server work on one specific project.

Clients and the MC-Server do not necessarily have to run on the same computer. They may communicate via a remote interface with each other, e.g. via COM/DCOM. ASAM MCD-3 MC provides a COM-IDL file for implementing such an interface.

The following chapters describe the primary functions of an MC-server and the classes that are available to carry out those functions: provide access to calibration objects and to collect measurement data from an ECU. Furthermore, one chapter covers the access to the MC-server database.

## Access to Calibration Objects

ASAM MCD-3 MC provides a comprehensive API to retrieve meta data about calibration objects and to remotely manage calibration objects. The API supports the full range of calibration objects in an ECU:

- Scalar: single value
- Value block: array
- ASCII: sting
- Curve: 1D look-up table
- Map: 2D look-up table
- Cuboid: 3D look-up table
- 4D look-up table
- 5D look-up table

For the first use-case, - retrieve meta data about calibration objects -, the standard defines one class for generic read-access methods applied to all calibration objects.

| Class | Function |
| --- | --- |

| | |
|---|---|
| *MCDDbCharacteristic* | ▪ Get name, descriptions, memory address, da identifier, calibration limits, format string, ma change, reference to computation method ar attribute of the calibration object |

## This class is actually a parent class for specific calibration object API classes, as listed in the next table. Their API provides additional read-access methods on top of MCDDbCharacteristic, which is specific to the calibration objects.

| Class | Additional Function |
|---|---|
| *MCDDbScalarCharacteristic* | ▪ none |
| *MCDDbValueBlockCharacteristic* | ▪ Get the x-and y-dimensions of the array |
| *MCDDbAsciiCharacteristic* | ▪ Get the maximum number of characters of th |
| *MCDDbCurveCharacteristic* | ▪ Access to the axis |
| *MCDDbMapCharacteristic* | ▪ Access to the axes |
| *MCDDbCube3DCharacteristic* | ▪ Access to the axes |
| *MCDDbCube4DCharacteristic* | ▪ Access to the axes |
| *MCDDbCube5DCharacteristic* | ▪ Access to the axes |

## Axes of look-up table calibration objects have their own access classes.

| Class | Function |
|---|---|

| | |
|---|---|
| *MCDDbAxisDescription* | • Get name, descriptions, axis type, data type, calibration limits, format string, maximum allc points, reference to axis points, reference to working point of the table, reference to comp read-only attribute |
| *MCDDbAxisPoints* | • Get the names and number of axes, find axe |
| *MCDDbAxisPts* | • Get name, descriptions, memory address, di calibration limits, maximum allowed number maximum allowed value change, reference t working point of the table, reference to comp read-only attribute of the calibration object |

## For the second use-case, - remotely manage calibration objects -, the standard defines API classes that allow to read and write the values of calibration objects in an ECU. Furthermore, classes exist to handle concurrent access to the objects and to add or remove them from collections.

The following table lists one API class, which provides generic access methods to calibration objects.

| Class | Function |
|---|---|
| *MCDCharacteristic* | • Get name and descriptions, handle concurre calibration object from multiple clients, e.g. lc access to the object |

## This class is actually a parent class for specific calibration object API classes, as listed in the next table. Their API provides additional read- and write access methods on top of MCDCharacteristic, which is specific to the calibration objects.

| Class | Calibration Object | Access to ... |
|---|---|---|
| *MCDScalarCharacteristic* | Scalar | • Value |

| | | |
|---|---|---|
| *MCDValueBlockCharactersistic* | Value block | • Array values |
| *MCDASCIICharactersitic* | ASCII | • String array values |
| *MCDCurveCharacteristic* | Curve | • X-axis array<br>• Curve table array |
| *MCDMapCharacteristic* | Map | • X-axis array<br>• Y-axis array<br>• Map table array |
| *MCDCube3DCharacteristic* | Cuboid | • X-axis array<br>• Y-axis array<br>• Z-axis array<br>• Cuboid table array |
| *MCDCube4DCharacteristic* | 4D look-up table | • X-axis array<br>• Y-axis array<br>• Z-axis array<br>• W-axis array<br>• 4D look-up table ar |
| *MCDCube5DCharacteristic* | 5D look-up table | • X-axis array<br>• Y-axis array<br>• Z-axis array<br>• W-axis array<br>• V-axis array<br>• 5D look-up table ar |

# Those calibration object API classes use secondary API classes for the read- and write-access to table values as per the next table.

| Class | Access to ... |
|---|---|
| *MCDVectorCharacteristic* | • Curve table array |
| *MCDMatrixCharacteristic* | • Map table array |

| | |
|---|---|
| *MCDMatrix3DCharacteristic* | • Cuboid table array |
| *MCDMatrix4DCharacteristic* | • 4D look-up table array |
| *MCDMatrix5DCharacteristic* | • 5D look-up table array |

## The standard defines some more API classes on this level for read-only access to table values as per the next table.

| Class | Read-only access to ... |
|---|---|
| *MCDValueCurve* | • Curve table array |
| *MCDValueMap* | • Map table array |
| *MCDValueCube3D* | • Cuboid table array |
| *MCDValueCube4D* | • 4D look-up table array |
| *MCDValueCube5D* | • 5D look-up table array |

## Those secondary-level API classes use third-level generic API classes for read-only access to array values of various dimensions, as per the next table.

| Class | Read-only access to ... |
|---|---|
| *MCDValueArray* | • Values of a generic 1D array |
| *MCDValueMatrix* | • Values of a generic 2D array |
| *MCDValueMatrix3D* | • Values of a 3D array |
| *MCDValueMatrix4D* | • Values of a 4D array |

| | |
|---|---|
| *MCDValueMatrix5D* | ▪ Values of a 5D array |

**Read access can start and end at specific array index positions, so that only a sub-array is returned. The same applies to write access. Besides reading and writing of the actual parameter values, the API classes also have methods to read the array's dimensions, i.e. number of elements for each dimension.**

## Access to Measurement Objects

ASAM MCD-3 MC provides a comprehensive API to retrieve meta data about measurement objects and to remotely manage the logging of measurement objects.

For the first use-case, - retrieve meta data about measurement objects -, the standard defines one class for generic read-access methods applied to all measurement objects.

| Class | Function |
|---|---|
| *MCDDbMeasurement* | ▪ Get name, descriptions, default sample rate, array size, data type, display identifier, forma to computation method |

**For the second use-case, - remotely manage the logging of measurement objects -, the standard defines two objects, which are described in the next chapters.**

- Collector: data logging to a ring buffer and synchronous transfer of the data to clients
- Recorder: data-logging to files and asynchronous transfer of the data to clients

## Collector

Collectors log the values of measurement and calibration objects from the ECU with a common rate in a ring buffer and makes the data available to clients every time one sample of a measurement has been finished. The use of Collectors is suitable, when the bandwidth between the MC-system and the clients is big enough to transfer the data while the acquisition is continuously running. The collection of data is started and stopped by Watchers, which may include manual triggering from a client.

Collectors log data according to a data acquisition list and some further acquisition parameters that has been set up by their client. The data acquisition list consist of the names of measurement and calibration objects (i.e. MCDDatatypeShortName). All objects on the data acquisition list need to be tied to the same logical link. In case of calibration objects, the list further defines the sub-objects, e.g. axes or tables, and which cell value or a range of cell values shall be acquired. The data

acquisition list and sample rate must be set up by the client before the Collector can be activated. All further acquisition parameters can be optionally set by the client, as they have default values. Those parameters are:

- Time stamp: If set to "true", then a relative time value is added to each sample. The zero-point of all time values is the first activation of any Collector, Watcher or WriteReadRecorder during the life-time of the MC-server. This provides a common time base across all acquisitions, including acquisitions with multiple clients.
- Rate: Defines the data acquisition sample rate.
- Down-sampling: Defines that only each $n^{th}$ value shall be stored.
- Start delay: Defines the time between activation of the Collector and start of sending data to the client. A positive value indicates a delay between activation and start. A negative value indicates that data shall be sent to the client that has been captured before the activation point-of-time.
- Stop delay: Defines the time between deactivation of the Collector and stop of sending data to the client. A positive value indicates a delay between deactivation and stop. A negative value indicates that the stop shall occur before the deactivation point-of-time.
- No of samples: Number of samples in the buffer when the Collector informs about that a measurement is ready to be sent (onCollectorResultReady event).
- Buffer size: Number of samples that the buffer can store.

The Collector is addressable by clients in its initial state eOS_CREATED. A client application can add the names of measurement and calibration objects to the acquisition list during this state. The trigger condition may be set and the ring buffer can be configured. In the state eOS_CONFIGURED, the MC-server checks that data acquisition is possible with the given acquisition list. Once activated, the Collector transitions into the state of eOS_ACTIVATED and starts to continually write the data to its ring buffer. The trigger condition is monitored.

The Collector reaches the state eOS_STARTED_PENDING, once the trigger condition evaluates to "true" and a positive start time delay was defined for the data acquisition. The Collector now waits until this time elapsed. Data transfer to the client starts in the state of eOS_STARTED. Every time the buffer has been filled with a complete measurement sample, the Collector signals to a client via an onCollectorResultReady event, that a new sample is available. Clients have to actively pull this data from the MC-server. If a client does not pull the data in time, a buffer overflow could occur. The Collector would report that to the client with a buffer overflow event (onCollectorError).

During the started state, the Collector monitors the stop trigger condition. When the stop condition is met, the Collector moves to the eOS_ACTIVATED_PENDING state and continues to transfer data until a positive stop time delay has elapsed. The Collector then transitions back into the eOS_ACTIVATED state. In case the logical link becomes offline during data acquisition, the Collector automatically falls back into the state eOS_CONFIGURED. Every state transition of a Collector is reported via events to clients.

ASAM MCD-3 MC defines several classes for setting up and managing Collectors and their data acquisition lists.

| Class | Function |
| --- | --- |
| *MCDCollectors* | • Get the names and number of Collector obje index or name, add, deactivate or remove C |
| *MCDCollector* | • Get name and descriptions of Collector, activ configuration, check the configuration, config acquisition, stop acquisition, deactivate, get s times, get and set start and stop watcher obj manage the buffer, transfer results to client, state |

| | |
|---|---|
| *MCDCollectedObjects* | • Get the names and number of collected obje objects by index or name, add scalar, value l remove collected objects |
| *MCDCollectedObject* | • Get name and descriptions of collected objec object's description, get the measurement or |
| *MCDScalarCollectorDescription* | • Get the representation type of a scalar |
| *MCDValueBlockCollectorDescription* | • Get the representation type of a value block |
| *MCDCurveCollectorDescription* | • Get the representation type of a curve |
| *MCDMapCollectorDescription* | • Get the representation type of a map |
| *MCDCollectorEventHandler* | • Process Collector events on creation, configu pending, start, activate pending, lock, unlock ready, modification of the data acquisition lis error |
| *MCDBuffer* | • Get and set buffer size, sample rate, down s stamping flag, get error and filling level |

## A Collector has a data acquisition list (accessible via the MCDCollectedObjects class), which contains references to measurement and calibration objects. Their meta data can be access via the classes of the following table.

| Class | Function |
|---|---|
| *MCDDbCharacteristics* | Get list of calibration object names, get number o the list, find calibration objects in the collection by |
| *MCDDbMeasurements* | Get list of measurement object names, get numb objects in the list, find measurement objects in th name |

## The collected data is stored in a data structure that is accessible via the MCDResults parent class and further

**derived classes. MCDResults in the context of this standard means the acquisition of one sample (i.e. one line of the buffer). The acquired data may originate from multiple ECUs. An MCDResult object may contain multiple MCDResponse objects. MCDResponse in the context of this standard means the data of one sample from one ECU. An MCDResponse object contains MCDResponseParameter objects. MCDResponseParameter objects in the context of this standard are used to describe the data structure of the collected data. At the bottom of the data structure, MCDResponseParameter references to a value of the class MCDValue.**

| Class | Function |
| --- | --- |
| *MCDResults* | • Get number of measurement samples, get m by index, get error |
| *MCDResult* | • Get the responses available for this measure always exactly one response |
| *MCDResponses* | • Get the response by index |
| *MCDResponse* | • Get name and descriptions of the measurem parameters |
| *MCDResponseParameters* | • Get number and names of response parame response by index or name |
| *MCDResponseParameter* | • Get name and descriptions of the parameter parameters, get data type, unit, valid numbe radix, value and value range information |
| *MCDValue* | • Data type dependent get and set methods fo values, get data type, get length of strings, g clear value |

# The standard defines two representation types, in which the data is stored and transferred to the clients:

- eRT_ECU: raw, internal implementation value, as stored in the ECU
- eRT_PHYSICAL: physical, human-readable value, converted from the ECU value via a computation method

Data of the type eRT_PHYSICAL is always stored as a double-precision floating-point data type (A_FLOAT64). Data of the type eRT_ECU has the same data type as stored in the ECU. The standard defines the following data types:

- eA_BOOLEAN: boolean
- eA_FLOAT32: floating-point 32-bit
- eA_FLOAT64: floating-point 64-bit
- eA_UINT8: unsigned integer 8-bit
- eA_UINT16: unsigned integer 16-bit
- eA_UINT32: unsigned integer 32-bit
- eA_UINT64: unsigned integer 64-bit
- eA_INT8: signed integer 8-bit
- eA_INT16: signed integer 16-bit
- eA_INT32: signed integer 32-bit
- eA_INT64: signed integer 64-bit
- eA_ASCIISTRING: ASCII string
- eA_UNICODE2STRING: Unicode zero-terminated string according to ISO-10646 UCS-2

## Recorder

Recorders acquire data the same way as Collectors, but make the data in a different way available to clients. In fact, the data acquisition of a Recorder is internally set up via a Collector object. However, instead of transferring the buffer content immediately to the client after the acquisition, the content is actually written to one or multiple files. The Recorder API then allows to transfer the data from the files asynchronously to clients while the measurement is still running, or to transfer the data at a later time after the acquisition has finished. Since the logged data is available in files, it is also possible to transfer the data through other means than the ASAM MCD-3 MC API. Recorders are typically used, when the bandwidth between the MC-system and clients is not sufficient enough for synchronous data transfer.

ASAM MCD-3 MC defines two types of  Recorders:

- Write-Read Recorder via the class MCDWriteReadRecorder:
  Acquires data, writes data to files, transfers data to client, is started and stopped by a Watcher.
- Read-Only Recorder via the class MCDReadRecorder:
  Transfers data to client.

Only the Write-Read Recorder is capable of carrying out a data acquisition. The Read-Only Recorder is used for transferring data captured by the Write-Read Recorder to clients at a later point-of-time, typically when the measurement has finished.

The Write-Read Recorder acquires the data the same way as the Collector with two exceptions. The Write-Read Recorder has the additional eOS_PAUSED state, which can temporarily pause the data acquisition via the pause method, and restart data acquisition via the resume method. Furthermore, the Write-Read Recorder can acquire data from different logical links, unlike Collectors whose data must originate from one logical link.

The standard does not stipulate specific file formats for storing the measured data. The standard allows to optionally define the filename and -scheme, path and number of files to be used to store the measured data. A new file will be created during

each transition from eOS_ACTIVATED to  eOS_STARTED. Each transition from eOS_STARTED to eOS_ACTIVATED or eOS_CONFIGURED closes the current file. Files with the same name will be overwritten by the MC-server. Consequently, by using a suitable filename-scheme, the Write-Read Recorder can be set up to store data in a file-based ring-buffer scheme.

The standard defines several classes for setting up and managing Recorders.

| Class | Function |
| --- | --- |
| *MCDRecorders* | • Get list of Recorder names, get number of R create Recorders, find Recorders by index o available file formats, remove Recorders |
| *MCDWriteReadRecorder* | • Get name and descriptions of Recorder, acti configuration, check the configuration, start a acquisition, deactivate, write data to file, get times, get absolute time of first sample, get a watcher objects, get current state, transfer re Collector objects for the Recorder, get file na errors, get lock state |
| *MCDWriteReadRecorderCollectors* | • Get list of Collector names, get number of C create Collectors, find Collectors by index or Collectors |
| *MCDWriteReadRecorderCollector* | • Get name and descriptions of Collector, trans get and set sample rate and down sampling get number of samples, get list of collected c |
| *MCDReadRecorder* | • Get name and descriptions of Recorder, get the Recorder, get absolute time of first samp format, get representation type of the data |
| *MCDReadRecorderCollectors* | • Get list of Collector names, get number of C Collectors by index or name |
| *MCDReadRecorderCollector* | • Get name and descriptions of Collector, get Recorder, transfer results to client, get numb |
| *MCDReadCollectedObjects* | • Get list of names of collected objects, get nu list, find collected objects by index or name |

| | |
|---|---|
| *MCDRateInfo* | ▪ Get name and descriptions of sample rate ot scaling factor of one sample step, get the val |
| *MCDWriteReadRecorderEventHandler* | ▪ Process Collector events on creation, configu pending, start, activate pending, lock, unlock ready, modification of the data acquisition lis error |

# The acquired data is stored in files in the same structure as described for Collectors. Unlike in the case of Collectors, each sample must have a time stamp.

## Watcher

Watchers allow to set up event-triggered data logging on the MC-server.  A watcher is a server-sided object, which continuously monitors other objects, such as values from measurement or calibration objects, and triggers an event once a pre-defined condition is met. The event can be used to start and stop Collectors and Recorders, to pause Recorders, or may be used by other event handlers.

A watcher has exactly one trigger condition with up to two trigger source objects. The sources can be the value of a measurement object, the value of a calibration object (a cell value in case of an array), an event, the absolute or relative time. The trigger source objects have to be defined in the MC-server's database and may be available on different logical links. A source can also be another trigger condition, which effectively allows to construct complex trigger conditions that involve more than two source objects. The trigger conditions are expressed with simple operators, such as bit operators, logical or relational operators, and edge-detection operators. Furthermore, a trigger condition may checks, whether a specific offset or gradient thresholds has been exceeded.

A watcher object has three internal states. In the eWS_CREATED state, the watcher can be configured, i.e. it is created or removed and the trigger condition can be set. In the eWS_INACTIVE state, the watcher is linked to Collectors, Recorders or other watcher objects. From this state, the watcher can be activated and would then transition into the state eWS_ACTIVE. While active, the watcher continuously monitors the source objects of the trigger condition. The watcher will trigger a "Fire" event, once the trigger condition evaluates to "true". The watcher then either remains in the state eWS_ACTIVE and continues to monitor the sources, or transitions back into the state eWS_INACTIVE depending on its AutoDeactivate configuration flag. API methods allow to manually move the watcher between the active and inactive state at any time. A watcher can also be activated by another watcher, which allows to set up watcher cascades.

The standard defines watcher API classes to setup and manage the watchers as described in the following table.

| Class | Function |
|---|---|
| MCDGlobalEventTriggerSource | ▪ Defines an event trigger source from a globa |
| MCDSystemEventTriggerSource | ▪ Defines an event trigger source from a syste |

| | |
|---|---|
| MCDLogicalLinkEventTriggerSource | • Defines an event trigger source from a logica |
| MCDConstantValueTriggerSource | • Defines a trigger source object, which always value set during configuration of the watcher |
| MCDScalarValueTriggerSource | • Defines a scalar trigger source object |
| MCDVectorValueTriggerSource | • Defines a vector trigger source object |
| MCDMatrixValueTriggerSource | • Defines a matrix trigger source object |
| MCDRelativeTimeValueTriggerSource | • Defines a trigger source object, which returns activation of the watcher |
| MCDConstantTimeTriggerSource | • Defines a trigger source object, which always time set during configuration of the watcher |
| MCDAbsoluteTimeTriggerSource | • Defines an absolute time trigger source obje |
| MCDEmptyTrigger | • Defines a trigger, which always evaluates to |

# Triggers use operator classes to evaluate the trigger condition as per the next table.

| Class | Function |
|---|---|
| *MCDUnaryTrigger* | • Operator applied to one trigger source |
| *MCDBinaryTrigger* | • Operator applied to two trigger sources |
| *MCDGradientTrigger* | • Calculates the difference between the currer from a specific time period in the past of the evaluates to "true", if the difference exceeds |

| | |
|---|---|
| *MCDOffsetTrigger* | • Calculates the difference between the currer object and its value during activation of the w to "true", if the difference exceeds a specifie |
| *MCDEmptyTrigger* | • Operator always evaluates to "false" |

# Further API classes provide methods for the creation and administration of watchers.

| Class | Function |
|---|---|
| *MCDWatchers* | • Add, find and remove watchers from the syst |
| *MCDWatcher* | • Activate, deactivate, change the configuratio check the configuration and unconditionally f activation watcher, set up event handlers, as configuration and current state data |
| *MCDTriggerFactory* | • Create trigger operators |
| *MCDTriggerSourceFactory* | • Create trigger sources |
| *MCDWatcherEventHandler* | • Returns "true", if a specific watcher was activ deactivated, has fired, has errored-out or has unlocked |

## Database

The measureable variables and calibration parameters of an ECU are described in A2L-files according to ASAM MCD-2 MC. Calibration tools read this file and internally store their content. The ASAM MCD-3 MC API provides programmatic read-only access for clients to this data. Those API classes all have the characters "Db" at the fourth and fifth position in the class name. The objects provide access to nearly the complete content of what is typically described in an A2L-file. The preceding chapters already described the access classes to measureable variables and calibration parameters:

- MCDDbMeasurement
- MCDDbCharacteristic and its derived classes:
  - MCDDbScalarCharacteristic
  - MCDDbValueBlockCharacteristic
  - MCDDbAsciiCharacteristic
  - MCDDbCurveCharacteristic
  - MCDDbMapCharacteristic
  - MCDDbCube3DCharacteristic

- MCDDbCube4DCharacteristic
- MCDDbCube5DCharacteristic

Those classes contain the meta data about the name of the measurement or calibration object, descriptions, data type, display identifier, format and limits. Specifically for measurement variables, the respective class further provides the default sample rate, accuracy, resolution and array size. For calibration parameters, the respective class provides the address in memory, extended limits, max value change in one calibration step, the read-only flag and further meta data specific to the calibration parameter type.

Every measurement and calibration object has a reference to a computation method object MCDDbCompuMethod. This object describes, how a value is transformed from its internal representation type (eRT_ECU) to a physical representation type (eRT_PHYSICAL). The conversion is typically done through a linear or rational function, for which this class provides the coefficients. This object has further references to conversion tables (MCDDbCompuTab, MCDDbCompuVTab or MCDDbCompuVTabRange) and units (MCDDbUnit). The latter has a reference to the physical dimension (MCDDbPhysicalDimension). The last two classes allow to convert values of the same physical dimensions to different units.

Further classes are available to access other objects of the data base that describe general properties of the ECU SW architecture: MCDDbFunction, MCDDbGroup and MCDDbModPar.

# Relation to Other Standards

ASAM MCD-3 MC is the successor of ASAM ASAP3. Both standards are not compatible to each other, though. The two standards just cover broadly the same application areas and use-cases.

ASAM MCD-3 MC uses ASAM MCD-2 MC. An MC-server according ASAM MCD-3 MC is able to parse and import data from A2L-files as specified by ASAM MCD-2 MC. This data is available via classes of the MC-server API.

# Industry Adaption

From a technical point-of-view, ASAM MCD-3 MC is a modern standard that covers a wide array of use-cases for MC-servers. Despite its sophistication, ASAM MCD-3 MC has never been more successful in the industry than its early predecessor ASAM ASAP3. The level of sophistication and the attempt to cover many use-cases resulted in a complex API, which is costly in implementation and complex in use. The forced integration of D-servers to the standard throughout its development history contributed to the complex API. The D-server has been separated from the standard meanwhile, but the complexity of the API remained essentially the same. Furthermore, the standard favors an implementation via DCOM, whose configuration and administration is considerably more complex compared to ASAP3-based systems. Since ASAP3 covers the most important use-cases, is easier to use and is very stable in operation, there is rarely justification to switch to ASAM MCD-3 MC. Although version 3.0 of the standard was released in 2011, most tool vendors still use version 2.2 implementation in their tools today.

# List of Deliverables

The standard includes the following deliverables:

- Programmer's Guide
- COM-IDL Technology Reference Mapping Rules
- IDL file
- API Reference
- UML Model