RM0327
Reference manual

Multi-Target Trace API version 1.0

Overview

Multi-Target Trace (MTT) is an application instrumentation library that provides a consistent way to embed instrumentation into a software application, and selectively transmit the data to a remote client for analysis. The API is portable and independent of the underlying transmission mechanisms.

Contents

1 **Introduction** 4

2 **Using the API** 6

2.1 Examples 7

3 **API version definition** 11

4 **API function return values** 12

5 **Trace initialization** 13

5.1 Enumerations 13

5.2 Data structure types 14

5.3 Functions 14

6 **Trace components** 18

6.1 Macros 18

6.2 Data structure types 18

6.3 Functions 19

7 **Trace levels and filtering** 23

7.1 Trace levels 23

7.2 Filtering 24

7.3 User defined level 24

7.4 Functions 25

8 **Trace generation** 30

8.1 Macros 30

8.2 Enumerations 32

8.3 Functions 33

9 Trace monitoring 37
9.1 Data structure types 37
9.2 Functions 37
10 Revision history 39
Index of functions 40
Index 41

1 Introduction

Implementations of the MTT API are available as libraries for STLinux (both kernel and user spaces) and OS21. It supports the following cores:

- ARM Cortex-A9
- ST40
- STxP70
- ST231

The MTT library provides an abstraction layer between the user application and the actual trace mechanism. It makes no assumptions about how the trace data is made available on remote workstations.

The MTT API contains four main function groups:

- initialization
- trace component management
- trace filtering
- data trace

Figure 1. MTT library block diagram

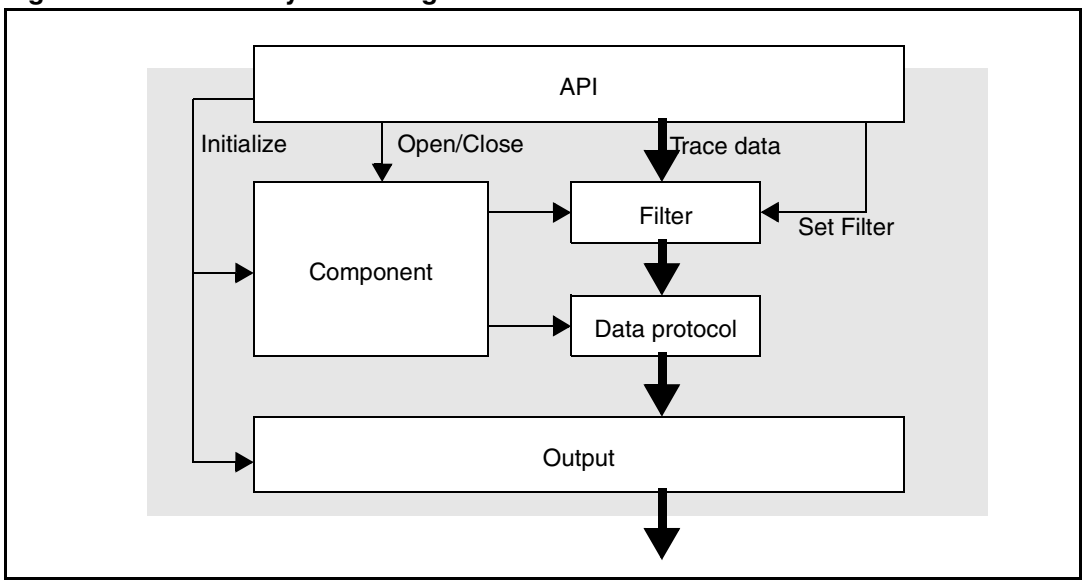
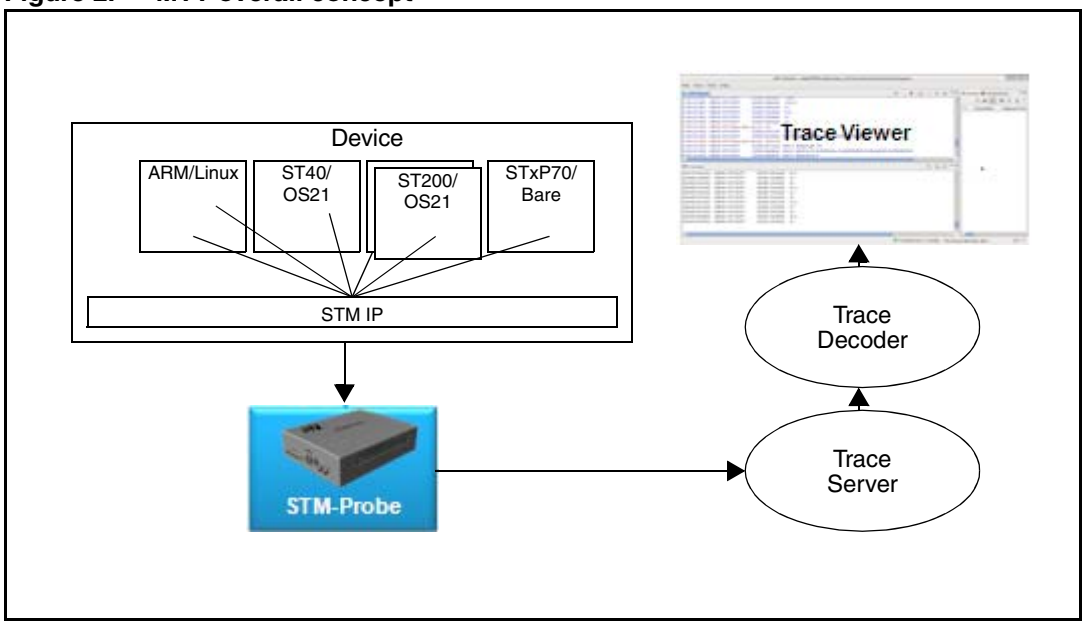


Figure 1 illustrates how these functionalities are managed in the library.

- *Initialize* creates the necessary setting up and allocation of resources to enable the library to generate data on the output media
- *Component* provides the ability to group trace messages according to user-defined categories.
- *Filtering* provides the ability to enable or disable the generation of trace messages according to trace levels.
- *Data protocol* ensures that the data are formatted in a manner that is consistent across all API implementations.
- *Output* sends the trace log, formatted according to the MTT data protocol, to the output media supported by the API implementation. The typical output media is STM port, but in Linux it can be files or over a socket.

Figure 2 shows the MTT overall trace concept, with transport over STM IP and a trace viewer client.

Figure 2. MTT overall concept



2 Using the API

The Multi-Target Trace API is defined in the header file `mtt.h`. All API users must include this header file in their source code. The following considerations must be taken into account when including this header file.

- The MTT API extensively utilizes standard fixed size integer types as defined by ISO. These types can be found in the system library header named `<stdint.h>`. In case this file is not provided with the platform or the build environment, suitable definitions for the required types need to be created.
- The MTT API uses the Boolean type `bool` as defined by ISO. This type can be found in the system library header named `<stdbool.h>`. In case this file is not provided with the platform or the build environment, suitable definitions for the required types must be created. In most cases it should be possible to find definitions of these platform-specific types on the internet.

The basic data types are defined in `mtt_types.h`, which also includes `<stdint.h>`.

The following naming conventions are used for defining various data structures and function calls of the MTT API:

- all defines begin with the prefix `MTT_`
- all data structures begin with the prefix `mtt_`
- all data structures used by the API functions end in the suffix `_t` (this stands for "type")
- all API function names begin with the prefix `mtt_`

`MTT_` and `mtt_` stand for Multi-Target Trace.

The MTT API is intended to work with any trace transport mechanism.

- One supported trace mechanism is the STM IP, which has a limited number of channels.
- Another supported trace mechanism is trace in a file. This does not have the same limitation as STM IP.
- The MTT API is independent of the trace transport mechanism. As a consequence, no output device needs to be supplied by the user of the API.
- It is the user's responsibility, at trace session configuration time, to select the appropriate mechanism for trace generation.

2.1 Examples

This section provides several simple examples of MTT API usage.

Figure 3 shows an example for tracing a Linux kernel module. The example includes:

- calls to `mtt_trace()` to trace a set of named signals
- a call to `mtt_print()` to demonstrate printf-like logging

Figure 3. Linux kernel module example

```
#include "mtt.h"
...
static mtt_comp_handle_t mtt_handle;
mtt_return_t rv;

rv = mtt_open( MTT_COMP_ID_ANY, DEVICE_NAME, &mtt_handle);
if (rv != MTT_ERR_NONE) processMTTerror(rv);



/*
 * log the number of processed irqs
 * don’t check here if log was successful in order to reduce intrusiveness
 **/
mtt_trace(mtt_handle, MTT_LEVEL_INFO, MTT_TRACEITEM_UINT32, &nb_irqs, “nb_irqs");
...
/*
 * trace when gold value 'val' is released by the driver
 * don’t check here if log was successful in order to reduce intrusiveness
 **/
mtt_trace(mtt_handle, MTT_LEVEL_INFO, MTT_TRACEITEM_UINT32, &val, “gold_value");
...
/*
 * Printf like
 * don’t check here if log was successful in order to reduce intrusiveness
 */
mtt_print(mtt_handle, MTT_LEVEL_API_RETURN, "fake_alsa_init");
...
/*
 * No longer need to trace data for this component
 **/
rv = mtt_close(mtt_handle);
if (rv != MTT_ERR_NONE) processMTTerror(rv);
...
```

Figure 4 shows an example of setting up tracing for a user-mode application.

Figure 4. Linux user-mode example

```
#include "mtt.h"

static mtt_comp_handle_t mtt_handle;
static mtt_init_param_t *init_param;
mtt_return_t rv;

/* register to the MTT API as a client, and setup the required features
*/
init_param.attr = MTT_INIT_LOCAL_TIME | MTT_INIT_CONTEXT;
rv = mtt_initialize (&init_param);
if (rv != MTT_ERR_NONE) processMTTerror(rv);
...

/* open, ask for a component handle for this thread
*/
rv = mtt_open( MTT_COMP_ID_ANY, “my_application”, &mtt_handle);
if (rv != MTT_ERR_NONE) processMTTerror(rv);
...

/* release the component, unregister with the trace infrastructure
*/
rv = mtt_close(mtt_handle);
if (rv != MTT_ERR_NONE) processMTTerror(rv);
...
```

Both kernel space tracing and user space tracing use the same API.

Figure 5 shows how to log a vector of integers together with a brief text explanation “FrameError”.

Figure 5. Logging a vector example

```
{
  unsigned int tracevector[2];
  tracevector[0] = frame_id;                /* the frame where an error has occured */
  tracevector[1] = (unsigned int) deviation; /* Deviation regarding expected value */

   /* Send this vector as an error */
      mtt_trace(mtt_handle, MTT_LEVEL_ERROR,
           MTT_TYPE(MTT_TYPE_UINT32,MTT_TYPEP_VECTOR,sizeof(int),2),
           (void *)(tracevector), "FrameError");
}
```

Figure 6 shows an example that uses trace levels.

Figure 6. Trace levels example

```
{
/* Enable traces for all components on my core */
uint32_t * prev_level;
mtt_set_core_filter ( MTT_LEVEL_ALL, &prev_level);

/* Instantiate 2 trace components */
mtt_comp_handle_t  comp_handle_in;
mtt_comp_handle_t  comp_handle_out;
mtt_open (MTT_COMP_ID_ANY, "data_in",  &comp_handle_in);
mtt_open (MTT_COMP_ID_ANY, "data_out", &comp_handle_out);

/* Set trace level to Error for 1st component */
/* Set trace level to Info for 2nd component */
mtt_set_component_filter (comp_handle_in, MTT_LEVEL_ERROR, &prev_level)
mtt_set_component_filter (comp_handle_out, MTT_LEVEL_INFO, &prev_level);


/* Trace ... */
mtt_trace (comp_handle_in,  MTT_LEVEL_ERROR, type_info, data, "mydata:in");
mtt_trace (comp_handle_out, MTT_LEVEL_INFO,  type_info, data, "mydata:out");

/* ... */

/* Disable all traces for the core: no trace */
mtt_set_core_filter ( MTT_LEVEL_NONE, &prev_level);


/* ... */

/* Enable all traces for the core */
/* Disable all traces for 1st component, Enable all traces for 2nd component */
/* Only 2nd component traces are generated */
mtt_set_core_filter ( MTT_LEVEL_ALL, &prev_level);
mtt_set_component_filter (comp_handle_in, MTT_LEVEL_NONE, &prev_level)
mtt_set_component_filter (comp_handle_out, MTT_LEVEL_ALL, &prev_level);

/* ... */
}
```

3 API version definition

The version number of the API is specified by the defines listed in *Table 1*.

Table 1. API version definition

Define	Description
MTT_API_VER_MAJOR	The major revision number of this API
MTT_API_VER_MINOR	The minor revision number of this API
MTT_API_VERSION	Combination of major and minor version so that it can be used as an integer
MTT_API_VER_STRING	API version number, formatted as a string
MTT_API_VER_STRING_LEN	The length of MTT_API_VER_STRING

The data structures of the MTT API use these constant values to determine the version number of the API.

The API version is sent as an implicit trace to inform the viewer tools of the implemented API version. If the versions of the API header file mtt.h and the implementation do not correspond, then a defect must be submitted to the API library provider.

4 API function return values

All MTT functions return a value that is a member of the `mtt_return_t` enumeration.

The calling function must evaluate the return value of an API function call in order to monitor the success or failure of that function call. If the function returns an error, then the calling function must carry out an appropriate action, as defined by the return values.

Table 2. Members of the mtt_return_t enumeration

Return value	Description
MTT_ERR_NONE	No error
MTT_ERR_OTHER	An error that is not listed here has occurred
MTT_ERR_FN_UNIMPLEMENTED	The called function has not yet been implemented
MTT_ERR_USAGE	MTT API used incorrectly
MTT_ERR_PARAM	Invalid parameter passed to function
MTT_ERR_CONNECTION	Connection failed
MTT_ERR_ALREADY_CNF	API already configured
MTT_ERR_TIMEOUT	Communication timeout occurred
MTT_ERR_FERRROR	File access error
MTT_ERR_SOCKERR	Socket error (cannot connect)
MTT_ERR_ALREADY_OPENED	Trace component was already open

5 Trace initialization

The data structures, enumerations and variables required for configuring the MTT API are described in this section.

The API includes functions for MTT API initialization and closure. Their purpose is to set up the interaction between a trace tool and the target to be traced, and to clean up all existing connections before closure.

Note: The API initialization functions cannot be called from an interrupt.

5.1 Enumerations

The API uses the enumeration `mtt_init_attr_bit_t` for coding a bit mask for the `impl_attr` field in a structure of type `mtt_impl_version_info`. *Table 3* describes the use for each of the elements in this enumeration.

Table 3. Members of the mtt_attr_bit_t enumeration

Enumerator	Description
`MTT_INIT_LOCAL_TIME`	Configure Trace records to be dispatched with local time. When `MTT_INIT_LOCAL_TIME` is set, the API implementation adds a field encoding the “local time” in the output trace records. The “local time” (also known as the “software time stamp”) is an optional field of the MTT data protocol. When requested, local time is inserted by the API implementation and therefore is related to the environment the API is used: – on Linux, it is typically given in ns – on OS21, it is given as a number of ticks STM trace capture has a system time-stamping support that is independent from `MTT_INIT_LOCAL_TIME`. Therefore the software time stamp is not required in this case, and consequently instrumentation intrusiveness and traced data are smaller.
`MTT_INIT_DROP_RECORDS`	Discard the trace record if it cannot be sent. If `MTT_INIT_DROP_RECORDS` is not set in the initialization parameters, the default condition is to wait until output port access is available.
`MTT_INIT_LOG_LEVEL`	Log trace level in the trace record. When set, the trace level is logged in the trace record. The trace level is an optional field of the MTT data protocol.
`MTT_INIT_CONTEXT`	Trace records despatched with execution context. When `MTT_INIT_CONTEXT` is set, an optional field, encoding the context from where the call to the API trace is performed, is inserted in the trace record. The context is an optional field of the MTT data protocol. In the case of the Linux implementation, this is the PID of the process that has issued the trace.

5.2 Data structure types

Table 4 lists the data structure types provided for configuring the API.

Table 4. Configuration data structure types

Type	Description
`mtt_init_param_t`	Structure type for the MTT API configuration. Used by *mtt_initialize on page 15*,
`mtt_impl_version_info_t`	Structure type for the implemented version information. Returned by *mtt_get_capabilities on page 17*.
`mtt_api_caps_t`	Structure type for the API's capabilities. Returned by *mtt_get_capabilities on page 17*.

5.3 Functions

The API defines configuration and initialization functions. See *Table 5*.

Table 5. Trace initialization functions

Function	Description
`mtt_initialize()`	Use this function to initialize the MTT API. See *mtt_initialize on page 15*.
`mtt_unitialize()`	Use this function to clear up when the API is no longer needed. See *mtt_uninitialize on page 16*.
`mtt_get_capabilities()`	Use this function to ascertain the APIs capabilities. See *mtt_get_capabilities on page 17*.

mtt_initialize API initialization function

Description: Use this function to initialize the MTT API internal state. It is implicitly called on the first call to `mtt_open()` with a parameter of NULL in order to initialize the API with the default settings.

This function may be called more than once. Subsequent calls may change the `attr` field of the `mtt_init_param_t` structure that is passed as a parameter to `mtt_initialize()`.

The MTT API is intended to work with any trace transport mechanism, which means that the API has no output device specification. The implementation of `mtt_initialize()` invokes a transport mechanism constructor and therefore supports whatever transport mechanisms that are available for a given core.

Definition:

```
mtt_return_t mtt_initialize (
   const mtt_init_param_t * init_param )
```

Arguments:

`init_param` — This parameter takes as input a structure of type `mtt_init_param_t` that contains the configuration parameters to apply.

If the value is NULL, `mtt_initialize` initializes the API with the default settings.

Returns: A return code from `mtt_return_t` (see *Chapter 4 on page 12*). This function returns one of the following error codes:

- `MTT_ERR_NONE` if the initialization was successful
- `MTT_ERR_OTHER` if the target implementation is incompatible
- `MTT_ERR_PARAM` if an inconsistent parameter setting was passed as input
- `MTT_ERR_CONNECTION` if the API cannot connect to trace output

Comments: This function uses a structure with the type `mtt_init_param_t` to set the parameters of the API configuration. *Table 6* provides details of the fields in this structure.

Table 6. Fields of mtt_init_param_t

Field name	Description
`attr`	Bitmask encoding the attributes available, as specified by the enumeration `mtt_init_attr_bit_t`. See *Section 5.1 on page 13*.
`name`	An ASCII string that provides a name for this initialization, If not used, this can be NULL. The maximum size of `name` is 64 characters.
`reserved`	Reserved for future development.

`mtt_initialize()` attempts to generate a control trace containing the API version. However, as tracing is not necessarily enabled at the point when `mtt_initialize()` is called (or it is possibly in flight-controller recording mode) this initial control trace may not be sent or saved by the trace system. This means that the API version may need to be sent again.

When using trace over STM, `mtt_initialize()` does not configure the trace port. The intention is that the configuration is performed using JTAG (ST Target Pack) or by using a configuration utility on the master CPU.

Linux kernel client code does not require a call to this function.

mtt_uninitialize — Uninitialize the API

Description: When the API is no longer required, call this function in order to carry out the essential cleaning up.

This function performs some clean-up functionality for connections. Do not make any API calls after this function has been called.

Definition: `mtt_return_t mtt_uninitialize ( void )`

Arguments: None.

Returns: A return code from `mtt_return_t`. See *Chapter 4 on page 12*. This function returns the following error code:

– `MTT_ERR_NONE` if the uninitialization was successful

mtt_get_capabilities — Retrieve data regarding API capabilities

Description: This function returns the full mask of options that are supported by the MTT API implementation and are available to be passed to `mtt_initialize()` when the API was first initialized.

Definition: `mtt_return_t mtt_get_capabilities ( mtt_api_caps_t * api_caps )`

Arguments:

`api_caps`	The function fills in the structure pointed to by this argument with the API implementation capabilities and details of the implemented version.

Returns: A return code from `mtt_return_t` (see *Chapter 4 on page 12*). This function returns one of the following error codes:

- `MTT_ERR_NONE` if the retrieval was successful
- `MTT_ERR_OTHER` if the capabilities could not be retrieved
- `MTT_ERR_FN_UNIMPLEMENTED` if this function is not implemented

Comments: This function uses a structure with the type `mtt_api_caps_t` to return information describing the capabilities of the API implementation. This structure is an output parameter of `mtt_get_capabilities()`. *Table 7* provides details of the fields in this structure.

Table 7. Fields of mtt_api_caps_t

Field name	Description
`impl_info`	A structure of type `mtt_impl_version_info_t` that contains details of the implemented version of the API.
`impl_attr`	A bitmask that encodes a set of attributes, as specified by the `mtt_init_attr_bit_t` enumeration. See *Section 5.1 on page 13*.
`reserved`	Reserved for future development.

The type `mtt_api_caps_t` uses a structure with the type `mtt_impl_version_info_t` as a container for information about the implemented version. *Table 8* provides details of the fields in this structure.

Table 8. Fields of mtt_impl_version_info_t

Field name	Description
`version`	Container for `MTT_API_VERSION`.
`string`	Container for `MTT_API_VER_STRING`.

This function can be invoked before `mtt_initialize()` to check the API's capabilities. For example, if `api_caps->impl_attr` returns a bitmask `MTT_INIT_LOCAL_TIME | MTT_INIT_DROP_RECORDS | MTT_INIT_CONTEXT` then the application can pass any of these flags to `mtt_initialize()` with confidence that these features are supported.

6 Trace components

The developer may group together a number of elements of application functionality into a single *component*. That component can then be treated as a single entity for tracing purposes. For example, all the functions and interrupts related to a driver can be considered a component, or all the functions from a specific library.

The MTT API can trace different software components independently of one another. This means that the trace data can be filtered to show only the trace output of specific components (see *Chapter 7 on page 23* for information about filtering).

6.1 Macros

The MTT API provides a number of macros to help handle component IDs. See *Table 9*.

Table 9. Trace component defines

Define	Description
`MTT_COMP_ID_ANY`	If set, tell the API to assign any free component ID to an `mtt_open()` action.
`MTT_COMP_ID_MASK`	This macro defines the maximum value for a user-defined component ID.
`MTT_COMP_ID`	This is a helper macro to help constructing custom component IDs. This macro accepts a single parameter, which is a user-defined ID. This must be a positive integer that is smaller than `MTT_COMP_ID_MASK`.

6.2 Data structure types

The API defines a data structure type `mtt_comp_handle_t`. See *Table 10*.

Table 10. Trace component data structure type

Type	Description
`mtt_comp_handle_t`	This is an opaque container for trace connection handles.

6.3 Functions

The API defines several functions for interacting with trace components. See *Table 11*.

Table 11. Trace component functions

Function	Description
`mtt_open()`	Open a connection for a given trace component. See *mtt_open on page 20* for more information.
`mtt_get_component_handle()`	Retrieve a component handle from its identifier. See *mtt_get_component_handle on page 21* for more information.
`mtt_close()`	Close a connection. See *mtt_close on page 22* for more information.

mtt_open — Open a connection to a log trace for a given component

Description: Use this function to open a specific trace connection. This must be done by a software component to get a handle before it invokes any of the trace functions.

Definition:

```
mtt_return_t mtt_open (
   const uint32_t comp_id,
   const char * comp_name,
   mtt_comp_handle_t * comp_handle )
```

Arguments:

`comp_id`	The identifier for the software component that is attempting to open tracing. If can either be `MTT_COMP_ID_ANY` or the value returned by the `MTT_COMP_ID` macro. See *Section 6.1 on page 18*.
`comp_name`	The name of the component (an ASCII string). If the component is anonymous, then pass `NULL`.
`comp_handle`	The function returns a unique handle for this connection into the structure pointed to by this parameter. Other functions in the API use this handle to identify the component that they are tracing.

Returns: A return code from `mtt_return_t` (see *Chapter 4 on page 12*). This function returns one of the following error codes:

- `MTT_ERR_NONE` if the operation was successful
- `MTT_ERR_PARAM` if the `comp_id` parameter is ambiguous
- `MTT_ERR_CONNECTION` if opening the core connection failed (this may be due to insufficient resources being available)
- `MTT_ERR_ALREADY_OPENED` if the connection is already open

Comments: This function carries out the following actions.

- Registers a new software component, as specified by `comp_id` and `comp_name`.
- If STM transport is being used, the function allocates the next available free channel for this component. (If all the available exclusive channels have been allocated, the component is allocated instead to one of the channels that the API has reserved for shared use, which it may have to share with other components.)
- Returns a pointer to a filled-in structure of type `mtt_comp_handle_t`. All subsequent trace calls involving this component must use this handle.

The user does not indicate whether to allocate an exclusive or shared channel. This means that the components are allocated to exclusive channel on a “first-come, first served” basis, until all the exclusive channels are taken. Any additional component IDs are then allocated to one of the channels that have been reserved for shared use.

The number of channels that are available as exclusive depends upon the core and the configuration of the SoC.

Unless the parameter `comp_id` has the value `MMT_COMP_ID_ANY`, `comp_id` must be unique. If `comp_id` has already been used in a previous call to `mtt_open()`, then the function returns the error `MTT_ERR_ALREADY_OPENED`.

mtt_get_component_handle

Retrieve a component handle from its identifier

Description: Use this function to retrieve the handle associated with a given component. Use this function in cases where the application may not have direct access to the handle created by `mtt_open()`, such as when in an interrupt handler.

Definition:

```
mtt_return_t mtt_get_component_handle (
   const uint32_t comp_id,
   mtt_comp_handle_t * comp_handle )
```

Arguments:

`comp_id`	The component identifier.
`comp_handle`	The function fills in the structure pointed to by this parameter with the component handle.

Returns: A return code from `mtt_return_t` (see *Chapter 4 on page 12*). This function returns one of the following error codes:

- `MTT_ERR_NONE` if the retrieval was successful
- `MTT_ERR_OTHER` if there is no component corresponding with the given identifier

Comments: This function returns an error if called with `MTT_COMP_ID_ANY`. It also returns an error if `comp_id` has not yet been used in a call to `mtt_open`.

mtt_close — Close a trace component

Description: Use this function to release a component already opened with `mtt_open()` and associated resources.

Definition: `mtt_return_t mtt_close ( const mtt_comp_handle_t comp_handle )`

Arguments:

`comp_handle`	The handle associated with the connection to close.

Returns: A return code from `mtt_return_t` (see *Chapter 4 on page 12*). This function returns one of the following error codes:

- `MTT_ERR_NONE` if successful
- `MTT_ERR_OTHER` if closing the core connection failed

7 Trace levels and filtering

The MTT API includes a mechanism for filtering traces with respect to their trace level. This gives the trace tool the ability to collect only the trace messages that are of interest in any given circumstances.

7.1 Trace levels

The MTT API provides an enumeration called `mtt_trace_level_t` to define the trace levels for a given trace record. A component is initialized by `mtt_open()` with a filter level set to `MTT_LEVEL_ALL`.

When a trace function is invoked, the core filter level and the component filter level are both taken into account when determining if the trace should be generated. The core filter level is checked first; if that allows the trace to pass, then the component filter level is also checked.

Table 12 describes the use for each of the elements in this enumeration.

Table 12. Members of the mtt_trace_level_t enumeration

Enumerator	Description
`MTT_LEVEL_NONE`	Disable all traces
`MTT_LEVEL_ERROR`	Enable trace type level “Error”
`MTT_LEVEL_WARNING`	Enable trace type level “Warn”
`MTT_LEVEL_INFO`	Enable trace type level “Info”
`MTT_LEVEL_USERTAG`	Low boundary for user defined levels
`MTT_LEVEL_ALL`	Enable all traces

The trace write MTT functions have a `level` parameter. This `level` value is a bit mask that identifies the type of message; for example: “error”.

The `level` value is part of the trace message. The trace viewer on the workstation can use the `level` to decorate a chart display. The `level` value in the collected trace data can also be used to search and filter for a subset of records.

7.2 Filtering

The scope of trace filtering can be set as follows:

- per core (that is, for all components of a core)
- by component (allowing the filter level to be tuned for specific components)

Every call to write a trace record contains a trace level parameter that defines permutations of trace level (“error”, “information”, “warning”, and so forth) as a bit mask. A specific bit in the level parameter indicates the nature of the message. Typically, this will be a single bit, but the user can combine multiple attributes, if necessary; for example, “information” and “user defined value 1” can be combined to distinguish a particular user-defined set of information messages from all others.

The filter operates by dispatching a trace record only if the *same* bit in its own trace level bitmask is also set in both the core trace filter mask and the component trace filter mask.

The purpose of the core trace filter mask is to filter the trace stream from a specific core. For example, it can be configured to dispatch just the “error” level messages from a given core and suppress all other levels.

Note: The core trace filter mask is set by default to enable all levels of trace message.

The purpose of the component trace filter mask is to filter the trace stream from a specific component. For example, it can be configured to dispatch all the trace records associated with “Component A” but none at all from any other component.

Table 12 shows the trace levels that are defined by default. It is possible to define customized trace levels in addition to those listed in this table.

Use the function `mtt_set_core_filter()` to set the overall trace level at runtime. To disable all tracing, call `mtt_set_core_filter()` with a `new_level` parameter of `0x00`.

Use the function `mtt_set_component_filter()` to change the trace level for a specific component at runtime. Therefore, to suppress the collection of “Info” records for a given component, call `mtt_set_component_filter()` with the trace level bit `MTT_LEVEL_INFO` unset in the mask passed at the `new_level` parameter.

A trace record’s `level` value is not an absolute value. It represents the type of message ("warning", "user_value_10" and so forth) and the filter expression identifies which types of message to pass or suppress. For example, it could be set to suppress all but the "error" level bit value.

7.3 User defined level

The API provides a macro called `MTT_LEVEL_USER` to specify a user defined level of up to 7 bits to identify custom payload types in the trace stream. For example, a developer may wish to trace the contents of an aggregate value (such as a structure). A structure can be traced as an array of integers. By defining a user defined level specifically for this structure, the developer can search the trace data using the level field as a key, and then convert the data fields of any trace records that match this key from the array of integers to a more suitable aggregate type.

7.4 Functions

The API defines several functions for trace filtering at the core level. See *Table 13*.

Table 13. Core level filtering functions

Function	Description
`mtt_set_core_filter()`	Set the filter level for all components. See *mtt_set_core_filter on page 26* for more information.
`mtt_get_core_filter()`	Get the current filter level for all components. See *mtt_get_core_filter on page 27* for more information.

The API defines several functions for trace filtering at the component level. See *Table 14*.

Table 14. Component level filtering functions

Function	Description
`mtt_set_component_filter()`	Set the filter level for a specified component. See *mtt_set_component_filter on page 28* for more information.
`mtt_get_component_filter()`	Get the current filter level for a specified component. See *mtt_get_component_filter on page 29* for more information.

mtt_set_core_filter — Set the filtering level for all components

Description: Use this function to enable or disable tracing at one or more trace levels at runtime for all components. The trace levels are set or unset by passing a bitmask created by logically ORing a number of `mtt_trace_level_t` enumeration elements.

After this function is called, the API emits all traces (for any component) that have a trace level set in the `new_level` filter value.

Under normal circumstances, the user should not change the core level filter using `mtt_set_core_filter()`, as this can lead to confusion over the traces that are filtered out at the component level and those that are filtered out at the core level. This function is provided for use at the SoC level to enable or disable trace from specific cores.

Definition:

```
mtt_return_t mtt_set_core_filter (
   const uint32_t new_level,
   uint32_t * prev_level )
```

Arguments:

`new_level`	The new filter level to apply. This is a bitmask created by ORing `mtt_trace_level_t` enumeration elements.
`prev_level`	The function returns the previous filter level in the variable pointed to by this parameter, provided as a bitmask of `mtt_trace_level_t` enumeration elements.

Returns: A return code from `mtt_return_t` (see *Chapter 4 on page 12*). This function returns the error code `MTT_ERR_NONE` if successful.

mtt_get_core_filter — Get the trace filter level for the core

Description: Get the current trace filter level for the core.

Definition:

```
mtt_return_t mtt_get_core_filter (
   uint32_t * level )
```

Arguments:

`level`	The function returns the current filter level in the variable pointed to by this parameter, provided as a bitmask of `mtt_trace_level_t` enumeration elements.

Returns: A return code from `mtt_return_t` (see *Chapter 4 on page 12*). This function returns the error code `MTT_ERR_NONE` if successful.

mtt_set_component_filter — Set the filtering level for a component

Description: Use this function to enable or disable tracing at one or more trace levels at runtime for a specified component. The trace levels are set or unset by passing a bitmask created by logically ORing a number of `mtt_trace_level_t` enumeration elements.

After this function is called, the API emits all traces for the component referenced by `comp_handle` that have a trace with any bits set at the `new_level` filter value and also set in the core level filter. (See *mtt_set_core_filter on page 26*.)

Definition:

```
mtt_return_t mtt_set_component_filter (
   const mtt_comp_handle_t comp_handle,
   const uint32_t new_level,
   uint32_t * prev_level )
```

Arguments:

`comp_handle`	The handle of the component, as previously provided by `mtt_open()`.
`new_level`	New filter level, created by ORing a number of `mtt_trace_level_t` enumeration elements.
`prev_level`	The function returns the previous filter level in the variable pointed to by this parameter, provided as a bitmask of `mtt_trace_level_t` enumeration elements.

Returns: A return code from `mtt_return_t` (see *Chapter 4 on page 12*). This function returns the error code `MTT_ERR_NONE` if successful.

mtt_get_component_filter — Get the filtering level for a component

Description: Get the current level of filtering for a given component.

Definition:

```
mtt_return_t mtt_get_core_filter (
   const mtt_comp_handle_t comp_handle,
   uint32_t * level )
```

Arguments:

`comp_handle`	The handle of the component, as previously provided by `mtt_open()`.
`level`	The function returns the current filter level in the variable pointed to by this parameter, provided as a bitmask of `mtt_trace_level_t` enumeration elements.

Returns: A return code from `mtt_return_t` (see *Chapter 4 on page 12*). This function returns the error code `MTT_ERR_NONE` if successful.

8 Trace generation

The API provides three functions for generating trace messages. These are described in *Section 8.3 on page 33*.

In addition, the API also provides a number of macros and enumerations for use with the `mtt_trace()` function. These are described in section *Section 8.1* and *Section 8.2 on page 32* respectively.

8.1 Macros

The `mtt_trace()` function has a parameter called `type_info` which expects the data to be formatted in a specific way. To assist with this formatting, the MTT API provides a number of helper macros for generating the `type_info` parameter in the correct format. These macros are listed in *Table 15*.

Table 15. Trace macros

Define	Description
`MTT_TYPE`	General purpose macro for generating a `type_info` parameter. It can generate a `type_info` parameter for scalar, vector, hex data or a string. The maximum size of the data is 64 KBytes.(1) This macro accepts four parameters, which are described in detail in *Table 16*.
`MTT_TRACEITEM_STRING`	Macro for generating a `type_info` parameter for a string. The string is assumed to be ASCII. This macro accepts a single parameter `size` which is the number of characters in the string (including the terminal NUL).
`MTT_TRACEITEM_BLOB`	Macro for generating a `type_info` parameter for a binary large object (BLOB). This macro accepts a single parameter `size` which is the size of the BLOB in bytes. The maximum size is 64 KBytes.(1)
`MTT_TRACEITEM_DUMP`	Macro for generating a `type_info` parameter for a 32-bit hex dump. This macro accepts a single parameter `size` which is the size of the dump in 32-bit words. The maximum size is 16 K words.
`MTT_TRACEVECTOR_UINT8`	Macro for generating a `type_info` parameter for a vector of unsigned bytes.
`MTT_TRACEVECTOR_INT8`	Macro for generating a `type_info` parameter for a vector of signed bytes.
`MTT_TRACEVECTOR_UINT16`	Macro for generating a `type_info` parameter for a vector of unsigned shorts.
`MTT_TRACEVECTOR_INT16`	Macro for generating a `type_info` parameter for a vector of signed shorts.
`MTT_TRACEVECTOR_UINT32`	Macro for generating a `type_info` parameter for a vector of unsigned 32-bit integers.

Table 15. Trace macros (continued)

Define	Description
MTT_TRACEVECTOR_INT32	Macro for generating a type_info parameter for a vector of signed 32-bit integers.
MTT_TRACEVECTOR_UINT64	Macro for generating a type_info parameter for a vector of unsigned 64-bit integers.
MTT_TRACEVECTOR_INT64	Macro for generating a type_info parameter for a vector of signed 64-bit integers.
MTT_TRACEVECTOR_FLOAT	Macro for generating a type_info parameter for a vector of floating point variables.
MTT_TRACEVECTOR_DOUBLE	Macro for generating a type_info parameter for a vector of double floating point variables.

1. For data greater than 64_KBytes, use mtt_data_raw().

All of the MTT_TRACEITEM_* macros accept a single parameter size, which is the size of the data item (string, BLOB or hex dump) that is being passed to mtt_trace().

All of the MTT_TRACEVECTOR_* macros accept a single parameter size, which is the number of variables of the given data type in the vector being passed to the macro. To pass a scalar data item using one of these macros, set size to 1.

The macro MTT_TYPE accepts several parameters. These are described in *Table 16*.

Table 16. Parameters to MTT_TYPE

Parameter name	Direction	Description
scalar_type	in	The data type contained in the vector, as defined in mtt_type_scalar_t (see *Section 8.2*).
type_param	in	One of the following: – MTT_TYPEP_SCALAR – MTT_TYPEP_VECTOR – MTT_TYPEP_STRING – MTT_TYPEP_SHOWHEX These are defined in mtt_type_param_t (see *Section 8.2*).
element_size	in	The size of the scalar_type argument (in bytes). This value must be consistent with the selected type passed in the scalar_type parameter. (See *Table 17* for information on the correct values.)
nb_elements	in	The number of elements in the vector. If the data is a scalar, then this parameter must equal 1.

8.2 Enumerations

The MTT API defines several enumerations for use with the macros documented in *Section 8.1 on page 30*.

mtt_type_scalar_t

Use this enumeration to define the type of data that is to be encoded within the trace. This is to ensure that the remote tool interprets the incoming data correctly. The enumeration has been defined to address all basic data types.

Use a member of the `mtt_type_scalar_t` enumerator to fill in the `scalar_type` parameter of the `MTT_TYPE` macro. The members of this enumerator are listed in *Table 17*.

Table 17. Members of the mtt_type_scalar_t enumeration

Enumerator	Size of data (in bytes)(1)	Description
`MTT_TYPE_UINT8`	1	Data to be interpreted as an unsigned 8-bit integer
`MTT_TYPE_INT8`	1	Data to be interpreted as a signed 8-bit integer
`MTT_TYPE_UINT16`	2	Data to be interpreted as an unsigned 16-bit integer
`MTT_TYPE_INT16`	2	Data to be interpreted as a signed 16-bit integer
`MTT_TYPE_UINT32`	4	Data to be interpreted as an unsigned 32-bit integer
`MTT_TYPE_INT32`	4	Data to be interpreted as a signed 32-bit integer
`MTT_TYPE_UINT64`	8	Data to be interpreted as an unsigned 64-bit integer
`MTT_TYPE_INT64`	8	Data to be interpreted as a signed 64-bit integer
`MTT_TYPE_FLOAT`	4	Data to be interpreted as a 32-bit floating point (float)
`MTT_TYPE_DOUBLE`	8	Data to be interpreted as a 64-bit floating point (double)
`MTT_TYPE_BOOL`	1	Data to be interpreted as a boolean. It can only accept two values, `TRUE` or `FALSE`.

1. This value is required when filling in the the `element_size` parameter of `MTT_TYPE`.

mtt_type_param_t

Use this enumerator to fill in the `type_param` parameter of the `MTT_TYPE` macro.

The members of this enumerator are listed in *Table 18*.

Table 18. Members of the mtt_type_param_t enumeration

Enumerator	Description
`MTT_TYPEP_SCALAR`	Specify that the trace data is an atomic value.
`MTT_TYPEP_VECTOR`	Specify that the trace data is a vector (consisting of two or more values)
`MTT_TYPEP_STRING`	Specify that the trace data is an ASCII string.
`MTT_TYPEP_SHOWHEX`	Specify that the data must be displayed using a HEX representation. This is the type to choose for pointers. This enumerator is only applicable when used in combination with the `MTT_TYPE_UINT_*` and `MTT_TYPE_INT_*` enumerators of the `mtt_type_scalar_t` enumeration.

8.3 Functions

The API defines several functions for generating trace data. See *Table 19*.

Table 19. Trace data functions

Function	Description
`mtt_print()`	Provides basic `printf()` style functionality. See *mtt_print on page 35* for more information.
`mtt_trace()`	Generate a formatted trace record. See *mtt_trace on page 34* for more information.
`mtt_data_raw()`	Write binary data directly into trace system. See *mtt_data_raw on page 36* for more information.

mtt_trace — Formatted trace

Description: Use this function to generate a formatted trace record. The data inserted into the trace record can either be scalar (a single data item, such as an integer), a vector (such as an array of data items) or a string.

Definition:

```
mtt_return_t mtt_trace (
   const mtt_comp_handle_t component,
   const mtt_trace_level_t level,
   const uint32_t type_info,
   const void * data,
   const char * hint )
```

Arguments:

`component`	The component handle for the component that this trace record is tracing. The handle was provided by `mtt_open()`.
`level`	The trace level, which is a bitmask generated by logically ORing members of the enumeration `mtt_trace_level_t`. The trace record is emitted only if `level` contains bits that are also set in both the core filter and component filter. See *Section 7.1 on page 23* for information about trace levels.
`type_info`	A key built using an `MTT_TYPE` macro (or a derived macro) to specify the type of data (scalar or vector) and the size of the data. See *Section 8.1 on page 30*.
`data`	An opaque pointer to the data to include in the trace record.
`hint`	This is a short (12 bytes) string used as a human-readable identifier for the tracepoint. Pass `NULL` if this feature is not needed. If `hint` is not used, then traces are identified in the form <*routine_name*>+<*offset*>.

Returns: A return code from `mtt_return_t` (see *Chapter 4 on page 12*). This function returns the error code `MTT_ERR_NONE` if successful.

Comments: The combination of `hint` and `component` must be unique. The trace decoder tools can use the `hint` parameter to perform search requests and provide outlines for that trace in a similar way to using the `level` value in the message.

mtt_print "printf-like" function

Description: A function that provides "printf" (or "printk") functionality.

Definition:

```
mtt_return_t mtt_print (
   const mtt_comp_handle_t component,
   const mtt_trace_level_t level,
   const char * format_string,...)
```

Arguments:

`component`	The component handle of this trace, as previously provided by `mtt_open()`.
`level`	The trace level, as defined by `mtt_trace_level_t`. The trace record is emitted only if `level` contains bits that are also set in both the core filter and component filter. See *Section 7.1 on page 23*.
`format_string`	Message format, provided as a "printf" style format string.

Returns: A return code from `mtt_return_t` (see *Chapter 4 on page 12*). This function returns the error code `MTT_ERR_NONE` if successful.

mtt_data_raw — Raw data tracing

Description: Use this function to write binary data directly into the trace system without the need of a trace handle. Consequently, this function does not need any prior invocation of the `mtt_open()` function. It can also be called before `mtt_initialize()`, if necessary.

Definition:

```
mtt_return_t mtt_data_raw (
   const uint32_t componentid,
   const uint32_t level,
   const size_t length,
   const void * data )
```

Arguments:

`componentid`	The identifier for the software component that is attempting to open tracing. It can either be `MTT_COMP_ID_ANY` or the value returned by the `MTT_COMP_ID` macro. See *Section 6.1 on page 18*.
`level`	The trace level, which is a bitmask generated by logically ORing members of the enumeration `mtt_trace_level_t`. See *Section 7.1 on page 23*.
`length`	The length of the data to be sent.
`data`	The binary data to send to the output port.

Returns: A return code from `mtt_return_t` (see *Chapter 4 on page 12*). This function returns the error code `MTT_ERR_NONE` if successful.

Comments: This function is designed to be used in situations before the kernel is started. It is unsafe to use this function after the kernel has started as it does not honour any locks on resources.

The records generated by this function are not run-time filtered. The `componentid` and `level` parameters are included for the information of the trace viewer only.

When an STM port is used, this function logs the data using a predefined channel specifically reserved for this purpose.

There is no size limit for the data to be logged, so this function can be used for exceptionally large data BLOBs.

Warning: **STMicroelectronics do not recommend the use of this function as a default tracing function. Use `mtt_trace()` (or one of the macros derived from it) instead.**

9 Trace monitoring

The API provides functions that return important information and statistics relating to components.

9.1 Data structure types

The MTT API defines a data structure types for trace monitoring. See *Table 20*.

Table 20. Trace monitoring data structure types

Type	Description
`mtt_comp_info_t`	Structure for describing component information and statistics. Used by *mtt_get_component_info on page 38*.

9.2 Functions

The API defines a function for monitoring trace data. See *Table 21*.

Table 21. Trace data monitoring functions

Function	Description
`mtt_get_component_info()`	Provides information regarding a component. See *mtt_get_component_info on page 38* for more information.

mtt_get_component_info — Provide information about a component

Description: Given the handle to a component, this function fills in a structure of type `mtt_comp_info_t` with information and statistics about that component.

Definition:

```
mtt_return_t mtt_get_component_info(
   const mtt_comp_handle_t comp_handle,
   mtt_comp_info_t * comp_info)
```

Arguments:

`comp_handle`	The handle of the component, as previously provided by `mtt_open()` or `mtt_get_component_handle()`.
`comp_info`	The function fills in the `mtt_comp_info_t` structure that is pointed to by this parameter.

Returns: A return code for `mtt_return_t` (see *Chapter 4 on page 12*). This function returns the error code `MTT_ERR_NONE` if successful.

Comments: This function uses a structure with the type `mtt_comp_info_t` to return information relating to the component referenced by `comp_handle`. *Table 22* provides details of the fields in this structure.

Table 22. Fields of mtt_comp_info_t

Field name	Description
`handle`	The unique handle of the component. This is a pointer to a structure defined by `mtt_comp_handle_t`.
`port`	The trace port on which this component is logging traces. (If STM trace is being used, this is the channel number.)
`id`	The component ID.
`level`	A bitmask that provides the trace level for this component.
`name`	The component name, which is provided as an ASCII string. (The length is truncated to 64 bytes if the name is longer.)
`stat[4]`	An array of four `uint64` variables for statistics relating to this component. `stat[0]` : The total number of times that trace functions have been invoked for this component. `stat[1]` : The total number of times that trace records have been dispatched. `stat[2]` : Reserved for future use. `stat[3]` : Reserved for future use.

10 Revision history

Table 23. Document revision history

Date	Revision	Changes
16-Oct-2012	1	Initial release.

Index of functions

mtt_close 22
mtt_data_raw 36
mtt_get_capabilities 17
mtt_get_component_filter 29
mtt_get_component_handle 21
mtt_get_component_info. 38
mtt_get_core_filter 27
mtt_initialize 15
mtt_open 20
mtt_print 35
mtt_set_component_filter 28
mtt_set_core_filter. 26
mtt_trace 34
mtt_uninitialize 16

Index

A

ARM Cortex-A9 4

B

Boolean type 6

C

component trace filter mask 24
core trace filter mask 24

M

mtt.h 6, 11
mtt_comp_handle_t 20
mtt_impl_version_info_t 17
mtt_init_attr_bit_t 13
mtt_init_param_t 15
mtt_return_t 12
mtt_trace_level_t 23
MTT_TYPE 30-31
mtt_type_param_t 33
mtt_type_scalar_t 32

R

revision number 11

S

ST231 4
ST40 4
stdbool.h 6
stdint.h 6
STM IP 5-6, 20
STxP70 4

T

trace component 4, 18
trace filtering 4, 24
trace level 23