

# ATLAS Muon MDT

---

## MdtJtagGen Reference Manual

Document Version: 1.0  
Document ID: ATLAS-TDAQ-2008-XXX  
Document Date: 26 March 2009  
Document Status: Release

---

### Abstract

This document is a reference manual for the *MdtJtagGen* library (DLL) providing JTAG bit string generation and return/reply bit string handling, for MDT chamber front-end electronics initialization and configuration.

### Institutes and Authors:

NIKHEF, Amsterdam: H.Boterenbrood

**Table 1** Document Change Record

<b>Title:</b> ATLAS TDAQ Muon MDT MdtJtagGen Reference Manual			
<b>ID:</b> ATLAS-TDAQ-2008-XXX			
<b>Version</b>	<b>Issue</b>	<b>Date</b>	<b>Comment</b>
1.0	2	21 March 2007	Draft version.
1.0	3	9 April 2007	Added methods <code>version()</code> , <code>nextStringWithLen()</code> and <code>handleReplyWithLen()</code> .
1.0	4	12 April 2007	List of sequence identifiers changed (see <code>startSequence()</code> ). Added methods <code>csmErrorStatus()</code> and <code>csmStatus()</code> .
1.0	5	12 April 2007	
1.0	6	17 April 2007	Added status bits to <code>csmStatus()</code> . List of sequence identifiers changes (see <code>startSequence()</code> ).
1.0	7	26 April 2007	Added methods <code>dbAccessTime()</code> and <code>dbOperationTime()</code> . (for timing monitoring purposes only). Added figure with CSM JTAG chain of devices.
1.0	8	11 June 2007	Added methods <code>asdThreshold()</code> and <code>setAsdThreshold()</code> .
1.0	9	22 August 2007	Added methods <code>csmDeviceMask()</code> , <code>deviceIdCode()</code> , <code>extendedCheck()</code> and <code>setExtendedCheck()</code> .
1.0	10	9 October 2007	Changed names of some sequences; added test sequences. Added bits to <code>replyStatus()</code> . Added <code>amtStatusError()</code> .
1.0	11	18 January 2008	Changed name of sequence <code>SEQ_SCAN_CHAIN</code> to <code>SEQ_SCAN_CHAIN_AMT</code> . Added sequence <code>SEQ_SCAN_CHAIN_ASD</code> .
1.0	12	24 January 2008	Added methods <code>setUseAmtPhase()</code> and <code>setAmtPhase()</code> , for debugging purposes (not available to PVSS). Added description of some error bits to <code>csmErrorStatus()</code> .
1.0	13	4 April 2008	Added method <code>setInitMode()</code> .
1.0	14	14 October 2008	Added bit <code>DB_ERR_ASD_OFFS_RANGE</code> to status word returned by <code>dbStatus()</code> . Renamed <code>dbStatus()</code> to <code>dbErrorStatus()</code> .
1.0	15	26 March 2009	Added additional options to <code>setInitMode()</code> . Added some references.

## Table of Contents

<b>1 INTRODUCTION</b>	<b>5</b>
1.1 PURPOSE OF THE DOCUMENT	7
1.2 GLOSSARY, ACRONYMS AND ABBREVIATIONS	7
1.3 REFERENCES	7
<b>2 MDTJTAGEN</b>	<b>8</b>
2.1 DESCRIPTION	8
2.2 INTERFACE (API)	8
2.2.1 amtPhase	12
2.2.2 amtPhaseErrors	12
2.2.3 amtStatusError	12
2.2.4 asdThreshold	13
2.2.5 csmDeviceMask	13
2.2.6 csmErrorStatus	14
2.2.7 csmStatus	15
2.2.8 csmVersion	16
2.2.9 csmVersionDate	16
2.2.10 dbDefs	17
2.2.11 dbErrorStatus	17
2.2.12 dbPars	17
2.2.13 deviceIdCode	18
2.2.14 errStr	19
2.2.15 extendedCheck	19
2.2.16 handleReply	20
2.2.17 handleReplyWithLen	21
2.2.18 mezzMask	21
2.2.19 nextString	22
2.2.20 nextStringWithLen	23
2.2.21 replyStatus	23
2.2.22 setAsdThreshold	24
2.2.23 setChamber	24
2.2.24 setDbase	25
2.2.25 setExtendedCheck	25
2.2.26 setInitMode	26

---

2.2.27 setMezzMask	26
2.2.28 startSequence	27
2.2.29 version	27
<b>3 EXAMPLE PVSS SCRIPT</b>	<b>28</b>
<b>4 EXAMPLE C++ CODE</b>	<b>31</b>

## 1 Introduction

The MDT front-end electronics consists of a CSM module and up to 18 mezzanine boards connected to it. The devices (or ICs) on all the boards on one MDT chamber are daisy-chained by means of a **JTAG port** present on each device, as shown in Figure 1. A JTAG port has the following pins: TMS (mode select), TCK (clock), TDI (serial data in) and TDO (serial data out).

The JTAG ports of each device are state machines, controlled by the TCK and TMS signals. The state diagram is shown in Figure 2.

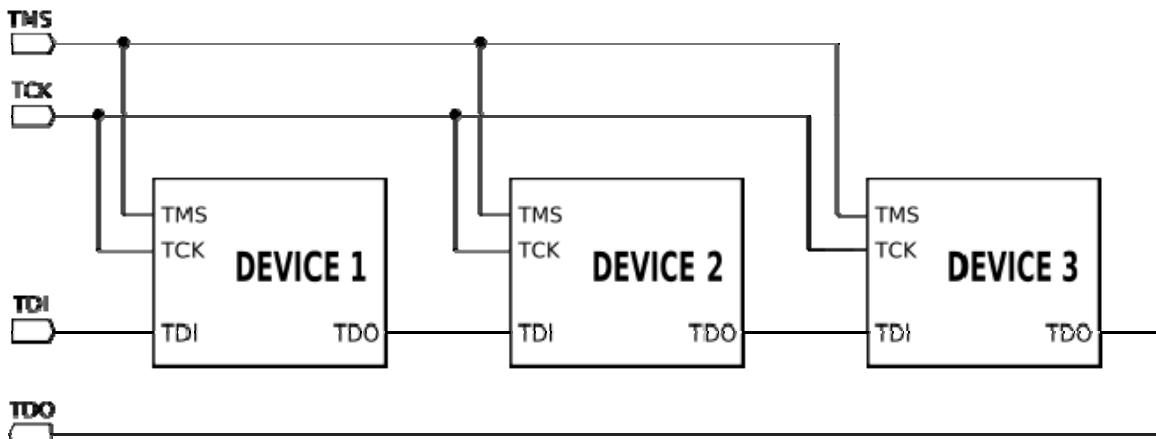


Figure 1. Example of a JTAG chain.

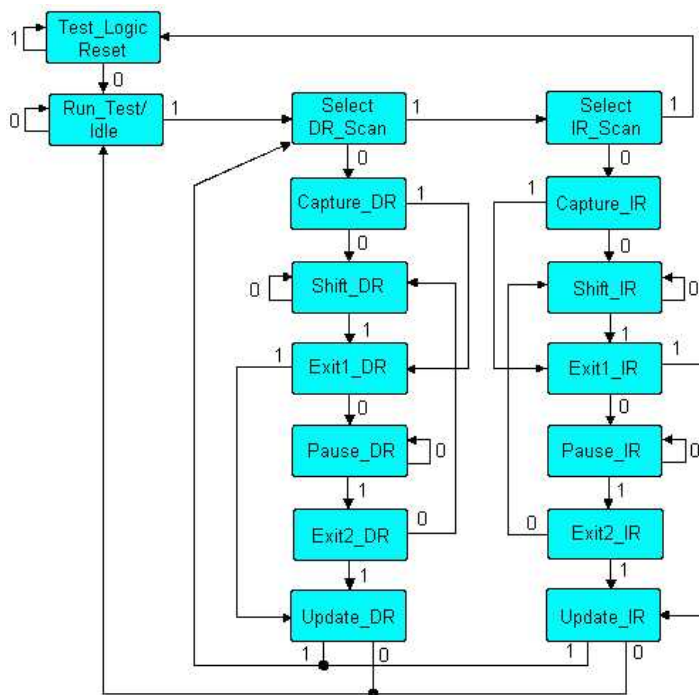
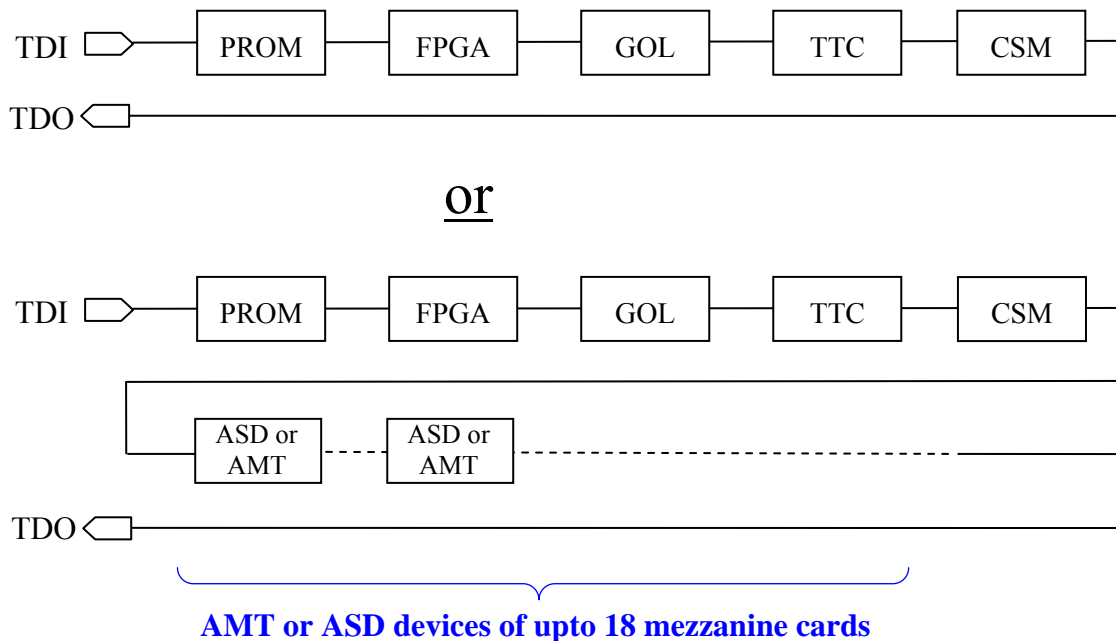


Figure 2. JTAG state transition diagram with '0' and '1' indicating the value of TMS during the TCK-controlled state transition.

While in state *Shift\_DR* or *Shift\_IR* (see Figure 2) bits are shifted serially into TDI and out of TDO. The TDO of a device may be connected to the TDI of the next device. Together the devices look like one long shift register. The total length depends on the number of bits present in each device between its TDI and TDO pins. This number may be different for each device. It depends also on whether the bits are shifted into the *Instruction Register* or into the *Data Register* of the JTAG interface. The contents of the *Instruction Register* selects a particular *Data Register*. Each type of device has its own set of instructions and set of data registers. Registers (Instruction and Data) may have any length. So depending on the 'instruction' in each device, the total length of the JTAG register chain varies.

The devices in the CSM JTAG chain are shown in Figure 3. Note that the ASD or AMT devices may be part of the chain or not. The number of AMT or ASD devices varies according to the number of mezzanine boards included in the chain, up to a maximum of 18 (it is configurable by means of an 18-bit mask setting in the CSM JTAG device).

The instruction register size for the AMT and ASD devices is 5, for the CSM device it is 6, for the TTC device it is 4, for the GOL device it is 4, for the FPGA device it is 6 and for the PROM device it is 16.



**Figure 3. JTAG devices in the CSM module. The devices on the mezzanines are included in the chain by a mask setting in the CSM device.**

Strings are always generated by MdtJtagGen in pairs: an **instruction string**, which on each device selects a certain data register, and a **data string**, which either contains the data for the selected register of each device, or contains enough (don't-care) bits to read out (i.e. shift out) the requested data from the devices. The JTAG protocol on the TMS and TCK lines controls the state machine (see Figure 2) and thus selects whether an instruction or a data string is currently being shifted through the chain.

In order to initialize and configure the CSM and mezzanine devices for data-taking in ATLAS, a sequence of bit strings must be uploaded to the JTAG port of the CSM module. For more detailed information see [ ].

The JTAG port of each CSM is connected to and operated by an MDT-DCS module (MDM) which is installed on each MDT chamber near the CSM, and in turn is connected to the ATLAS Detector Control System (DCS) by a CAN-bus.

Strings are generated on a host system that sends them via the CAN-bus to the MDM, which executes the JTAG protocol to upload the bit strings to the CSM. The MDM also collects the bits shifted out of the CSM's JTAG chain. These can be read out by the host system if required, or when explicitly requested by the library described in this manual.

## 1.1 Purpose of the document

This document is a reference manual for the **MdtJtagGen** class and library (DLL). It implements the JTAG bit string generation needed for the MDT chamber front-end electronics configuration and initialization procedure.

## 1.2 Glossary, acronyms and abbreviations

Provide definitions for all terms, acronyms and abbreviations used in this document. Any existing glossaries which are used should be referenced, to avoid repetition here.

<b>AMT</b>	ATLAS Muon TDC (Time-to-Digital-Converter), an IC for measuring the duration of the pulse-shaped signal from the ASD.
<b>ASD</b>	Amplifier-Shaper-Discriminator, an IC for processing the signals from the MDT wires.
<b>CSM</b>	Chamber Service Module, on-chamber electronics module, interfacing to the mezzanines (modules with ASD chips and an AMT chip, connecting directly to the MDT tube wires), and to the ATLAS Trigger, DAQ and DCS systems.
<b>DCS</b>	Detector Control System.
<b>JTAG</b>	Joint Test Action Group, IEEE 1149.1 standard describing test access ports for printed circuit boards, used also for testing sub-blocks of ICs, as a mechanism for debugging embedded systems, and –as described in this document– as a general-purpose access mechanism for on-chip configuration registers.
<b>MDT</b>	Monitored Drift Tube, a kind of muon chamber.

## 1.3 References

- 1 J.W. Chapman, *CSM-4 & Final CSM Design Manual*, ATL-M-ER-0004, EDMS Id 897581, [https://edms.cern.ch/file/897581/1/CSM\\_user\\_Manual.pdf](https://edms.cern.ch/file/897581/1/CSM_user_Manual.pdf)
- 2 Y. Arai *et al*, ATLAS Muon Drift Tube Electronics, 2008\_JINST\_3\_P09001, <http://www.iop.org/EJ/abstract/1748-0221/3/09/P09001>

## 2 MdtJtagGen

### 2.1 Description

MdtJtagGen is a C++ class that generates on request various sequences of strings needed for initialization, configuration and/or monitoring of the MDT front-end electronics.

The first implementation of MdtJtagGen derives from the string sequences generation code in a standalone MDT DAQ and configuration program called **gDaq**, version 3.9 (by Jeff Gregory), a program written in C and C++.

MdtJtagGen differs from *gDaq* in that it reads parameter settings from the ATLAS database, rather than from locally stored configuration files. Because MdtJtagGen is set up as a library, its inner workings are quite different from those in program *gDaq*.

MdtJtagGen has reasonable defaults for parameter settings, and thus may be used without having to access the ATLAS database containing the configuration parameters. In this case the only setting required is a bitmask indicating the mezzanines connected to the CSM.

Normally however MdtJtagGen is given the name of an MDT chamber, which it then uses to retrieve all parameters for the chamber from the database, to generate the strings of bits required to perform a specific operation (such as initialization or scanning for connected mezzanines).

The MdtJtagGen API has been extended such that it can be used as a so-called Control Extension (in the form of a *lib* or *DLL*) in the ATLAS SCADA system PVSS.

### 2.2 Interface (API)

Below a summary list of methods (or functions) available to users of MdtJtagGen is shown. If available, the corresponding name of the method if called from within PVSS scripts, is also shown, in [blue](#).

Note that in PVSS scripts the MdtJtagGen class is not instantiated, this is handled by PVSS internally, probably at the time PVSS loads the library. Multiple instantiations of the class is accomplished by defining a PVSS manager for each instantiation. In ATLAS there is a manager for each CAN-bus so that MDT initialization can take place on all CAN-buses in parallel.

Subsequent sub-sections describe the methods or functions in more detail. The functions are listed in alphabetical order, not in 'logical' order. For this see section 3 for an example PVSS script using MdtJtagGen.

- MdtJtagGen()  
(constructor for testing in C++ only).
- MdtJtagGen( BaseExternHdl \*nextHdl,  
PVSSulong funcCount,  
FunctionListRec fnList[] )  
(this constructor is implicitly called by PVSS).
- int amtPhaseRead( int i )  
int jtag\_amtPhase( int i )
- int amtPhaseErrors()  
int jtag\_amtPhaseErrors()
- int amtStatusError( int i )  
int jtag\_amtStatusError( int i )
- int asdThreshold( int asd, int mezz )  
int jtag\_asdThreshold( int asd, int mezz )
- void closeDbase()  
int jtag\_closeDbase()
- int csmDeviceMask()  
int jtag\_csmDeviceMask()
- int csmErrorStatus()  
int jtag\_csmErrorStatus()
- int csmState()
- int csmStatus()  
int jtag\_csmStatus()
- int csmVersion()  
int jtag\_csmVersion()
- int csmVersionDate()  
int jtag\_csmVersionDate()
- int dbAccessTime()  
int jtag\_dbAccessTime()
- int dbDefs()  
int jtag\_dbDefs()
- int dbErrorStatus()  
int jtag\_dbErrorStatus()
- int dbOperationTime()  
int jtag\_dbOperationTime()
- int dbPars()  
int jtag\_dbPars()
- int deviceIdCode( int dev )  
int jtag\_deviceIdCode( int dev )
- std::string errStr()  
string jtag\_errStr()

- `bool extendedCheck()`  
`int jtag_extendedCheck()`
- `bool handleReply( unsigned char *reply,`  
`int nbits )`  
`int jtag_handleReply( dyn_char reply,`  
`int nbits )`
- `bool handleReplyWithLen( unsigned char *reply )`  
`int jtag_handleReplyWithLen( dyn_char reply )`
- `int mezzMask()`  
`int jtag_mezzMask()`
- `bool nextString( unsigned char **instr,`  
`unsigned char **data,`  
`int *instr_len,`  
`int *data_len,`  
`bool *pause,`  
`bool *get_reply );`  
`int jtag_nextString( dyn_char &instr,`  
`dyn_char &data,`  
`int &instr_len,`  
`int &data_len,`  
`bool &pause,`  
`bool &get_reply )`
- `bool nextStringWithLen( unsigned char **instr,`  
`unsigned char **data,`  
`int *instr_len_bytes,`  
`int *data_len_bytes,`  
`bool *pause,`  
`bool *get_reply );`  
`int jtag_nextStringWithLen( dyn_char &instr,`  
`dyn_char &data,`  
`int &instr_len_bytes,`  
`int &data_len_bytes,`  
`bool &pause,`  
`bool &get_reply )`
- `int replyStatus()`  
`int jtag_replyStatus()`
- `void setAmtPhase( int i, int ph )`
- `void setUseAmtPhase( bool b)`
- `bool setAsdThreshold( int threshold )`  
`int jtag_setAsdThreshold( int threshold )`
- `bool setChamber( std::string chamber_name )`  
`int jtag_setChamber( string chamber_name )`

- `bool setdBase( std::string username,  
                  std::string password,  
                  std::string connectstring )`  
`int jtag_setDbase( string username,  
                  string password,  
                  string connectstring )`
- `bool setExtendedCheck( bool check )`  
`int jtag_setExtendedCheck( bool check )`
- `bool setInitMode( int mode )`  
`int jtag_setInitMode( int mode )`
- `bool setMezzMask( int mask )`  
`int jtag_setMezzMask( int mask )`
- `bool startSequence( int sequence_id )`  
`int jtag_startSequence( int sequence_id )`
- `void show( bool show_strings=true, bool show_defs=true )`  
(method for debug purposes in C++).
- `void ttcRxBcId()`
- `void ttcRxEvtId()`
- `void ttcRxStatus()`
- `int version()`  
`int jtag_version()`

### 2.2.1 amtPhase

#### SYNOPSIS

```
int jtag_amtPhase( int i );
```

#### PARAMETERS

*i* index of the mezzanine ( $0 \leq i \leq 17$ ).

#### DESCRIPTION

Returns the value of the ‘phase’ setting of the AMT on mezzanine *i* as an integer. This value is 4-bits significant.

This function should only be called after the CSM status has been read out (see function `startSequence`).

### 2.2.2 amtPhaseErrors

#### SYNOPSIS

```
int jtag_amtPhaseErrors();
```

#### PARAMETERS

None.

#### DESCRIPTION

Returns the status of the ‘phase error’ of all AMTs connected to the CSM as an integer, with 1 bit per AMT. This value is 18-bits significant.

This function should only be called after the CSM status has been read out (see function `startSequence`).

### 2.2.3 amtStatusError

#### SYNOPSIS

```
int jtag_amtStatusError( int i );
```

#### PARAMETERS

*i* index of the mezzanine ( $0 \leq i \leq 17$ ).

#### DESCRIPTION

Returns the value of the ‘hard error bits’ of the AMT on mezzanine *i* as an integer. This value is currently 9-bits significant.

This function should only be called after a CSM initialisation has been carried out (see function `startSequence`).

## 2.2.4 asdThreshold

### SYNOPSIS

```
int jtag_asdThreshold( int asd, int mezz );
```

### PARAMETERS

asd        number of the ASD chip on the mezzanine ( $0 \leq \text{asd} \leq 2$ ).

mezz      number of the mezzanine ( $0 \leq \text{mezz} \leq 17$ ).

### DESCRIPTION

Returns the current value of the ADC threshold setting of an ASD chip in the parameters of the current MDT chamber (or a default value if no chamber is selected).

The value is in milliVolts, and has a value between -254 and 256 mV (in steps of 2 mV).

## 2.2.5 csmDeviceMask

### SYNOPSIS

```
int jtag_csmDeviceMask();
```

### PARAMETERS

None.

### DESCRIPTION

Returns a bitmask with bits set, detailing the devices found on the CSM, after the ‘scan for devices’ sequences `SEQ_SCAN_CHAIN_AMT` or `SEQ_SCAN_CHAIN_ASD` has been completed (see function `startSequence`). The `SEQ_SCAN_CHAIN_AMT` sequence reads the so-called ‘ID code’ of the various devices that should or could be there, and decides on the basis of the ID code found whether the device appears in the ‘device mask’ returned by this function. To read the actual device ID code see function `deviceIdCode`.

The `SEQ_SCAN_CHAIN_ASD` sequence reads the ID code of some devices, but not of the ASD chips (there are actually 3 per mezzanine) which do not have an internal ID code. The ASD chips are detected by means of writing configuration data into them and reading this data back.

Currently the following bits are defined:

- Bits 0 – 17: AMT or ASD chip found (i.e. mezzanine found in this location and either AMT or ASD chips accessible)
- Bits 24 to 28: CSM, TTC, GOL, FPGA, PROM chip found, resp.

or:

```
#define CSM_DEVICE_AMT_OR_ASD_MASK 0x0003FFFF  
#define CSM_DEVICE_CSM             0x01000000  
#define CSM_DEVICE_TTC             0x02000000  
#define CSM_DEVICE_GOL             0x04000000  
#define CSM_DEVICE_FPGA            0x08000000  
#define CSM_DEVICE_PROM            0x10000000
```

So e.g. a CSM with all mezzanines connected should return device mask `0x1F03FFFF`, whether scanning for AMT or ASD.

## 2.2.6 csmErrorStatus

### SYNOPSIS

```
int jtag_csmErrorStatus();
```

### PARAMETERS

None.

### DESCRIPTION

Returns a bitmask detailing error conditions present on the CSM.

This function should only be called after the CSM status has been read out (see function `startSequence`).

Currently the following bits are defined:

```
#define CSM_ERR_GOL_NOT_READY          0x00000001
#define CSM_ERR_TTC_NOT_READY          0x00000002
#define CSM_ERR_LHC_CLK_NOT_LOCKED    0x00000004
#define CSM_ERR_GOL25_CLK_NOT_LOCKED  0x00000008
#define CSM_ERR_TTC_LOAD_ERROR         0x00000010
#define CSM_ERR_I2C_OP_FAILURE         0x00000020
#define CSM_ERR_TTC_I2C_COMP_ERR       0x00000040
#define CSM_ERR_CSM_ERROR               0x00000080
#define CSM_ERR_GOL_NOT_LOCKED         0x00000100
#define CSM_ERR_STATE_NOT_IDLE         0x00000200
#define CSM_ERR_SAMPLE_PHASE_ERR       0x00000400
#define CSM_ERR_CSM_CMD                 0x00000800
#define CSM_ERR_GOL25_MEZZ80           0x00001000
#define CSM_ERR_MEZZ_SPEED_PARS        0x00002000
```

The bit definitions above up to `CSM_ERR_GOL_NOT_LOCKED` correspond to inverted values of the ‘okay’ tickboxes present in the ‘Status Window’ of the **gDaq/gJtag** programs, labeled respectively:

- **GOL Ready**
- **TTC Ready**
- **Clock Lock**
- (`CSM_ERR_GOL25_CLK_NOT_LOCKED` *not present in gDaq*)
- **TTC Load**
- (`CSM_ERR_I2C_OP_FAILURE` *not present in gDaq: because already available as “I2C Check”?*)
- **I2C Ok**
- **No CSM Err**
- **GOL JTAG**

`CSM_ERR_STATE_NOT_IDLE` corresponds to a line-edit widget on the left side of the **gDaq/gJtag** Status Window, labeled **State**, which should have value **Idle** to be ‘okay’ after CSM initialization.

**gDaq/gJtag** tickbox **AMT Phase** does no longer represent an active error bit (info from Bob Ball).

There are 3 more ‘okay’ tickboxes in gDaq/gJtag, labeled “**CSM Ok**”, “**I2C Check**” and “**TTC/GOL Ok**”. These correspond to the 3 ‘status’ bits from the CSM, monitored by the MDM, which should all read zero when okay. MdtJtagGen does not deal with these bits: the user should read the bits directly from the MDM (The MDM may be configured to send these bits *on-change*, for immediate notification to the user).

CSM\_ERR\_CSM\_CMD means that the CSM command (4 bits) as part of a CSM Data Register JTAG bitstring was not read back correctly.

NB: status bit “**CSM Ok**” is the same as “**No CSM Err**”, and so is redundant to bit CSM\_ERR\_CSM\_ERROR.

NB: status bit “**I2C Check**” is redundant to bit CSM\_ERR\_I2C\_OP\_FAILURE.

NB: status bit “**TTC/GOL Ok**” is the *and* of “**GOL Ready**” and “**TTC Ready**”, and so is redundant to bits CSM\_ERR\_GOL\_NOT\_READY and CSM\_ERR\_TTC\_NOT\_READY.

## 2.2.7 csmStatus

### SYNOPSIS

```
int jtag_csmStatus();
```

### PARAMETERS

None.

### DESCRIPTION

Returns a bitmask detailing the status of the CSM.

This function should only be called after the CSM status has been read out (see function `startSequence`).

Currently the following bits are defined:

```
#define CSM_STAT_GOL_50MHZ           0x00000001
#define CSM_STAT_80MHZ_OPERATION     0x00000002
#define CSM_STAT_DAQ_ON              0x00000004

#define CSM_STAT_USE_DB_ASDOFFSETS   0x00010000
#define CSM_STAT_USE_DB_AMTPHASES    0x00020000
#define CSM_STAT_GOL50_AT_25        0x00040000
```

The first 3 bit definitions above correspond to the tickboxes present in the ‘Status Window’ of the **gDaq/gJtag** programs, labeled respectively:

- **GOL@50MHz**
- **80 MHz**
- **DAQ On**

CSM\_STAT\_USE\_DB\_ASDOFFSETS indicates whether the ASD offsets from the database are applied to the ASD parameters on the current chamber.

CSM\_STAT\_USE\_DB\_AMTPHASES indicates whether AMT phases are set from the values in the database (rather than determined dynamically) on the current chamber.

CSM\_STAT\_GOL50\_AT\_25 indicates whether the CSM’s 50 MHz GOL (if it has one) is forced to run at 25 MHz on the current chamber.

## 2.2.8 csmVersion

### SYNOPSIS

```
int jtag_csmVersion();
```

### PARAMETERS

None.

### DESCRIPTION

Returns a value representing the version number of the CSM firmware. This value is 12-bits significant, to be displayed as a hexadecimal number.

Current version numbers are 0x025 for a CSM-4 with GOL at 50MHz and 0x021 for a CSM-4 with GOL at 25 MHz.

This function can be called after the CSM status has been read out (see function `startSequence`).

## 2.2.9 csmVersionDate

### SYNOPSIS

```
int jtag_csmVersionDate();
```

### PARAMETERS

None.

### DESCRIPTION

Returns a value representing the calendar date associated with the CSM firmware. This value is 32-bits significant. It contains year, month, day and hour, as follows, with `d` the returned value (simply displaying `d` as a decimal value makes all numbers visible too):

```
year=d/1000000; month=(d%1000000)/10000; day=(d%10000)/100; hour=d%100;
```

This function can be called after the CSM status has been read out (see function `startSequence`).

## 2.2.10 dbDefs

### SYNOPSIS

```
int jtag_dbDefs();
```

### PARAMETERS

None.

### DESCRIPTION

Returns the number of JTAG definitions read from the database. May be called after the database has been opened, i.e. after a call to `setDbase()`. JTAG definitions describe parameters by name, size and location in the bit string sequences. Chamber parameter values are stored using these names.

## 2.2.11 dbErrorStatus

### SYNOPSIS

```
int jtag_dbErrorStatus();
```

### PARAMETERS

None.

### DESCRIPTION

Returns a bit pattern detailing the current (error) status of database operations.

Currently the following bits are defined:

```
#define DB_OKAY 0x00000000
#define DB_ERR_OPEN 0x00000001
#define DB_ERR_JTAGDEFS 0x00000002
#define DB_ERR_CHAMBERNAME 0x00000004
#define DB_ERR_NO_DATA 0x00000008
#define DB_ERR_PAR_READ 0x00000010
#define DB_ERR_PAR_UNKNOWN 0x00000020
#define DB_ERR_PAR_MEZZ 0x00000040
#define DB_ERR_DEV_UNHANDLED 0x00000080
#define DB_ERR_ASD_OFFSETS_RANGE 0x00000100
```

## 2.2.12 dbPars

### SYNOPSIS

```
int jtag_dbPars();
```

### PARAMETERS

None.

### DESCRIPTION

Returns the number of parameter settings for the current MDT chamber, read from the database. May be called after the parameters for a chamber have been read from the database, i.e. after a call to `setChamber()`.

## 2.2.13 deviceIdCode

### SYNOPSIS

```
int jtag_deviceIdCode( int dev );
```

### PARAMETERS

dev        index of the AMT-chip (mezzanine) ( $0 \leq dev \leq 17$ ), or CSM ( $dev = 24$ ) or TTC ( $dev = 25$ ) or GOL ( $dev = 26$ ) or FPGA ( $dev = 27$ ) or PROM ( $dev = 28$ ).

### DESCRIPTION

Returns the value of the 'ID code' found for the device during the the 'scan for devices' sequence SEQ\_SCAN\_CHAIN\_AMT or SEQ\_SCAN\_CHAIN\_ASD (see startSequence ).

Currently recognized 'valid' ID codes are:

- AMT        0x38B85031
- CSM        0x43534D37
- TTC        0x1545408F
- GOL        0x14535049
- FPGA       0x01038093
- PROM       0xF5057093

On the basis of the ID codes found the 'CSM device mask' is determined (see function csmDeviceMask).

## 2.2.14 errStr

### SYNOPSIS

```
string jtag_errStr();
```

### PARAMETERS

None.

### DESCRIPTION

Returns a string with a descriptive line of text, mostly concerning any error that occurred while using MdtJtagGen.

## 2.2.15 extendedCheck

### SYNOPSIS

```
int jtag_extendedCheck();
```

### PARAMETERS

None.

### DESCRIPTION

Returns the setting (either 0 or 1) of the ‘extended checking’ of JTAG reply strings. See also function `setExtendedCheck`.

## 2.2.16 handleReply

### SYNOPSIS

```
int jtag_handleReply( dyn_array reply, int nbits );
```

### PARAMETERS

`reply` byte array containing the data bit string read back from the CSM JTAG interface, but *NOT* including the string length in bits returned in the first 2 bytes of the reply from the MDM/CSM.

`nbits` number of bits in the `reply` byte array.

### DESCRIPTION

This function is called after the upload of the bit strings obtained from the previous call to `jtag_nextString()`, provided the returned `get_reply` parameter is `true`. See section 2.2.19.

This function either checks the ‘correctness’ of the bit string in `reply`, or extracts data from the string; what ‘correct’ is or what data is present in the string, depends on the contents of the pair of instruction/data strings that were uploaded to obtain this reply string (but this is handled internally by `MdtJtagGen`, since a call to `jtag_nextString()` should always precede a call to `jtag_handleReply()`). The number of bits in the reply data string should be equal to the number of data bits uploaded in the previous data string upload.

Returns a value of 1 for an ‘error-free’ string and 0 otherwise. If there is an error call `jtag_replyStatus()` for more information about the error.

Note that it will not be necessary to call `jtag_handleReply()` after each and every string pair upload (what exactly *is* necessary remains to be determined, but that issue will be internal to `MdtJtagGen`; the user simply responds to the value of `get_reply` returned by `jtag_nextString[withLen]()`. If e.g. the sequence is a status read-out (sequence identifier `SEQ_CSM_STATUS`, see section 0) then the retrieval of most of the reply strings from the MDM is required and `jtag_handleReply()` to be called.

**NB:** it is more efficient to use `jtag_handleReplyWithLen()`, see the next section.

## 2.2.17 handleReplyWithLen

### SYNOPSIS

```
int jtag_handleReplyWithLen( dyn_array reply );
```

### PARAMETERS

`reply` byte array containing the data bit string read back from the CSM JTAG interface, *including* the string length in bits returned in the first 2 bytes of the reply from the MDM/CSM (Most Significant Byte first).

### DESCRIPTION

This function is identical in functionality to `jtag_handleReply()`, except that parameter `reply` now is the literal byte array returned from the MDM: there is no need for the user to extract the string length in bits from the array, then to remove the 2 bytes containing this value from the array and finally to include the length as a separate parameter, as is the case for `jtag_handleReply()`.

## 2.2.18 mezzMask

### SYNOPSIS

```
int jtag_mezzMask();
```

### PARAMETERS

None.

### DESCRIPTION

Returns the bitmask representing the presence of mezzanines connected to the CSM. This value is 18-bits significant, to be displayed as a hexadecimal number. The value could have been set explicitly by the user by a call to `jtag_setMezzMask()`, or it originates from the database.

## 2.2.19 nextString

### SYNOPSIS

```
int jtag_nextString( dyn_array &instr,
                    dyn_array &data,
                    int      &instr_len,
                    int      &data_len,
                    bool     &pause,
                    bool     &get_reply );
```

### PARAMETERS

<code>instr</code>	byte array containing the instruction bit string.
<code>data</code>	byte array containing the data bit string.
<code>instr_len</code>	number of <u>bits</u> in the <code>instr</code> byte array.
<code>data_len</code>	number of <u>bits</u> in the <code>data</code> byte array.
<code>pause</code>	whether the user has to pause (1 s) after the string upload.
<code>get_reply</code>	whether the user should retrieve the CSM return/reply data bit string and call <code>handleString()</code> .

### DESCRIPTION

Returns the next instruction and data bit string pair in the sequence started by the previous call to `jtag_startSequence()`, for uploading to the CSM.

Note that the byte arrays containing the strings do not start with the 2 bytes containing the string length (in bits), as required by the MDM/CSM JTAG protocol.

Returns a value of 1 if it is not the last string pair, and 0 if it is.

Note that both string lengths may be zero: sometimes a string pair in a sequence is skipped for various reasons; the user must continue to call `jtag_nextString()` until a value of 0 is returned, independent of the string lengths returned.

When `pause` is true, it is recommended to wait for 1 second after the string pair upload, before further actions are taken.

When `get_reply` is true, the user is expected to retrieve the return/reply data string from the MDM/CSM and call `jtag_handleReply()`. It is not an error *not* to do this, but status or other returned information may be missed.

**NB:** it is more efficient to use `jtag_nextStringWithLen()`, see next section.

## 2.2.20 nextStringWithLen

### SYNOPSIS

```
int jtag_nextStringWithLen( dyn_array &instr,
                           dyn_array &data,
                           int      &instr_len_bytes,
                           int      &data_len_bytes,
                           bool      &pause,
                           bool      &get_reply );
```

### PARAMETERS

<code>instr</code>	byte array containing the instruction bit string and its length.
<code>data</code>	byte array containing the data bit string and its length.
<code>instr_len_bytes</code>	number of <u>bytes</u> in the <code>instr</code> byte array.
<code>data_len_bytes</code>	number of <u>bytes</u> in the <code>data</code> byte array.
<code>pause</code>	whether the user has to pause (1 s) after the string upload.
<code>get_reply</code>	whether the user should retrieve the CSM return/reply data bit string and call <code>handleReplyWithLen()</code> .

### DESCRIPTION

This function is similar in functionality to `jtag_nextString()`, except that the instruction and data bit strings are now preceded by their length in bits, contained in the first 2 bytes (MSB first), as required by the MDM/CSM JTAG protocol: there is no need for the user to add the string length in bits and calculate the array size in bytes himself before sending the bit strings to the MDM across the CAN-bus.

## 2.2.21 replyStatus

### SYNOPSIS

```
int jtag_replyStatus();
```

### PARAMETERS

None.

### DESCRIPTION

Returns a bit pattern detailing the type of error detected in the last reply string, including on which mezzanine, if appropriate. See `handleReply()`.

Currently the following bits are defined:

```
#define REPLY_OKAY                0x00000000
#define REPLY_DONE                0x00000001
#define REPLY_ERR_LEN             0x00000002
#define REPLY_ERR_CSM             0x00000004
#define REPLY_ERR_GOL             0x00000008
#define REPLY_ERR_AMT             0x00000010
#define REPLY_ERR_ASD             0x00000020
#define REPLY_ERR_AMT_PHASE_ERR  0x00000040
#define REPLY_ERR_AMT_PHASE      0x00000080
#define REPLY_ERR_AMT_STATUS     0x10000000
#define REPLY_ERR_UNKNOWN_PHASE 0x40000000
#define REPLY_ERR_MEZZMASK       0x03FFFF00
```

## 2.2.22 setAsdThreshold

### SYNOPSIS

```
int jtag_setAsdThreshold( int threshold );
```

### PARAMETERS

threshold ASD ADC threshold to set, in mV ( $-254 \leq \text{threshold} \leq 256$ , in increments of 2 mV)

### DESCRIPTION

This function may be used to set the threshold of all ASDs of all mezzanines of the current MDT chamber to the given value. The value must be between -254 and 256, and have an even value (odd values are rounded to the next even value nearest to zero).

The current ASD threshold values (read from the database, if a call to `jtag_setChamber()` was made earlier) are overwritten. Note also that ASD offsets are no longer applied (if they were enabled) after a call to `jtag_setAsdThreshold()`, until after the next call to `jtag_setChamber()`.

Current values for individual ASDs can be read using `jtag_asdThreshold()`.

Returns a value indicating whether a valid threshold was given (=1) or not (=0).

## 2.2.23 setChamber

### SYNOPSIS

```
int jtag_setChamber( string name );
```

### PARAMETERS

name name of the MDT chamber as a 6-char string, e.g. "BOL1A13".

### DESCRIPTION

This function is to be used after the database has been opened to retrieve the chamber's parameters from the database.

Returns a value indicating whether the operation was successful (=1) or not (=0). If not, call `jtag_dbErrorStatus()` and/or `jtag_errStr()` to get more information about any errors, e.g. wrong chamber name, no data for this chamber in the database, etc.

## 2.2.24 setDbase

### SYNOPSIS

```
int jtag_setDbase( string username,  
                  string password,  
                  string connectstring );
```

### PARAMETERS

username	name of the database user.
password	password for database access.
connectstring	string describing the database (name, protocol, etc) or containing an alias pointing to the description (i.e. an Oracle <i>tnsnames.ora</i> file entry).

### DESCRIPTION

This function is to be used to open the database with configuration parameters for the MDT chambers. After opening it reads the so-called ‘JTAG definitions’ from the database, a list of names describing the various bit sub-strings of the main JTAG data string used for CSM device configuration.

Returns a value indicating whether the operation was successful (=1) or not (=0). If not, call `jtag_dbErrorStatus()` and/or `jtag_errStr()` to get more information about any errors.

## 2.2.25 setExtendedCheck

### SYNOPSIS

```
int jtag_setExtendedCheck( bool check );
```

### PARAMETERS

check	false or true, to disable or enable the ‘extended check’ feature.
-------	---

### DESCRIPTION

This function can be used to enable a more extensive check of JTAG reply strings, when initializing a CSM. It results in getting more often a `get_reply` with a `true` value from `jtag_nextString[withLen]()` during a CSM initialization sequence.

Using this extended check will increase the time to initialize each CSM.

## 2.2.26 setInitMode

### SYNOPSIS

```
int jtag_setInitMode( int mode );
```

### PARAMETERS

mode      mode identifier.

### DESCRIPTION

This function may be used to modify some (ASD or AMT) parameters used in the subsequent MDT chamber initialisation sequence of the current MDT chamber.

Currently the following *modes* are defined:

```
#define INIT_MODE_NORMAL      0
#define INIT_MODE_CALIB_55    1
#define INIT_MODE_CALIB_AA    2
#define INIT_MODE_ASD0_ENA    3
#define INIT_MODE_ASD1_ENA    4
#define INIT_MODE_ASD2_ENA    5
```

Mode `INIT_MODE_NORMAL` sets ASD parameter *channel mask* to 0 and *capacitor select bits* to 0 (which are the defaults).

Mode `INIT_MODE_CALIB_55` sets ASD parameter *channel mask* to 0x55 and *capacitor select bits* to 4.

Mode `INIT_MODE_CALIB_AA` sets ASD parameter *channel mask* to 0xAA and *capacitor select bits* to 4.

Mode `INIT_MODE_ASD0_ENA` keeps all AMT's ASD #0 *channel enable* bits and disables all other channels (ASD #1 and ASD #2).

Mode `INIT_MODE_ASD1_ENA` keeps all AMT's ASD #1 *channel enable* bits and disables all other channels (ASD #0 and ASD #2).

Mode `INIT_MODE_ASD2_ENA` keeps all AMT's ASD #2 *channel enable* bits and disables all other channels (ASD #0 and ASD #1).

Returns a value indicating whether a valid mode identifier was given (=1) or not (=0).

## 2.2.27 setMezzMask

### SYNOPSIS

```
int jtag_setMezzMask( int mask );
```

### PARAMETERS

mask      bit mask denoting which mezzanines are to be included in the JTAG bitstring generation.

### DESCRIPTION

This function may be used to configure which mezzanines are present. Normally this information is taken from the database. If the database is not used, other device parameters should have reasonable values by default.

Don't use this function if parameters are taken from the database: the database should contain the proper mezzanine mask (which can be checked by calling `jtag_mezzMask()`).

Returns a value indicating whether a valid mask was given (=1) or not (=0).

## 2.2.28 startSequence

### SYNOPSIS

```
int jtag_startSequence( int sequence_id );
```

### PARAMETERS

`sequence_id` identifier of the requested sequence of JTAG bitstrings.

### DESCRIPTION

Prepares the string generator for generation of a particular sequence of JTAG bitstrings. The strings are obtained by repeated calls to `jtag_nextString()` until this returns 0.

Currently the following sequence identifiers are defined:

```
#define SEQ_NONE 0
#define SEQ_INIT_AND_START_CSM450 1
#define SEQ_INIT_AND_START_CSM4 2
#define SEQ_INIT_AND_SET_PHASE_CSM450 3
#define SEQ_INIT_AND_SET_PHASE_CSM4 4
#define SEQ_INIT_AND_START_CSM 5
#define SEQ_INIT_AND_SET_PHASE_CSM 6
#define SEQ_STATUS 7
#define SEQ_START_DAQ 8
#define SEQ_STOP_DAQ 9
#define SEQ_STARTDAQ_AND_CSMSTAT 10
#define SEQ_INFO 11
#define SEQ_SCAN_CHAIN_AMT 12
#define SEQ_SCAN_CHAIN_ASD 13
#define SEQ_TEST_1 14 // Undefined test seq
#define SEQ_TEST_2 15 // AMT status read
```

Returns a value indicating whether a valid identifier was given (=1) or not (=0).

Some sequences should be preceded by a reset to the CSM followed by a JTAG reset, including all `SEQ_INIT_XXX` sequences, as well as `SEQ_SCAN_CHAIN_AMT` and `SEQ_SCAN_CHAIN_ASD`.

## 2.2.29 version

### SYNOPSIS

```
int jtag_version();
```

### PARAMETERS

None.

### DESCRIPTION

Returns a value representing the version number of the MdtJtagGen code. The value is to be interpreted hexadecimal as `0xYYMMDDRR`, in which `YY`=year, `MM`=month, `DD`=day and `RR`=revision number.

### 3 Example PVSS Script

Below is shown a listing of a PVSS script, illustrating the use of the MdtJtagGen Control Extension: the database is opened, the parameters for a chamber are read from the database and a sequence to initialize the front-end electronics is generated.

```
#uses "MdtJtagGen.dll"

main()
{
    int      result;
    string   errstr;

    // Open the configuration database (and read the 'JTAG definitions')
    result = jtag_setDbase( "user", "password", "connection_string" );
    checkResult( result, "setDbase" );

    // Get chamber parameters
    result = jtag_setChamber( "BOL1A13" );
    checkResult( result, "setChamber" );

    jtag_closeDbase();

    // The number of 'JTAG definitions' from the database
    result = jtag_dbDefs();
    DebugN( "dbDefs=" + result );

    // The number of parameters of chamber "BOL1A13" from the database
    result = jtag_dbPars();
    DebugN( "dbPars=" + result );

    // The mezzanine mask of chamber "BOL1A13"
    result = jtag_mezzMask();
    string mm;
    sprintf( mm, "%08X", result );
    DebugN( "mezzMask=" + mm );

    // Start a bit string pair generation sequence
    result = jtag_startSequence( 1 );
    checkResult( result, "startSequence" );

    dyn_char instr, data, reply;
    int      instr_len, data_len, reply_len;
    bool     pause, get_reply;
    result = 1;

    // Continue until last string pair generated
    while( result )
    {
        string s, v;
        int    i;

        // Get the next string pair
        result = jtag_nextString( instr, data, instr_len, data_len,
                                pause, get_reply );
        checkResult( result, "nextString" );

        // Display the instruction bit string (as hex bytes)
        s = "";
        DebugN( "len=" + instr_len );
        for( i=1; i<=(instr_len+7)/8; ++i )
        {
            sprintf( v, "%02X ", instr[i] );
            s += v;
        }
        DebugN( s );
    }
}
```

```
// Display the data bit string (as hex bytes)
s = "";
DebugN( "len=" + data_len );
for( i=1; i<=(data_len+7)/8; ++i )
    {
        sprintf( v, "%02X ", data[i] );
        s += v;
    }
DebugN( s );

// Upload the strings to the MDM
.....
if( pause ) 'insert a delay (use 1 second)'

if( get_reply )
    {
        // Download the reply (data) string from the MDM, if required
        .....

        result = jtag_handleReply( reply, reply_len );
        checkResult( result, "handleReply" );

        if( result == 0 )
            {
                // Something is wrong
                result = jtag_replyStatus();
                sprintf( v, "%08X ", result );
                DebugN("replyStatus=" + v );

                // Show the bits detailing the problem(s)
                result = jtag_csmErrorStatus();
                sprintf( v, "%08X ", result );
                DebugN("csmErrorStatus=" + v );
            }
    }

// Show the bits detailing the CSM status
result = jtag_csmStatus();
sprintf( v, "%08X ", result );
DebugN("csmStatus=" + v );

// Start a bit string pair generation sequence to read the CSM status
result = jtag_startSequence( 11 );
checkResult( result, "startSequence" );

// Continue until last string pair generated
result = 1;
while( result )
    {
        string s, v;
        int i;

        // Get the next string pair
        result = jtag_nextString( instr, data, instr_len, data_len, pause );
        checkResult( result, "nextString" );

        // Upload the strings to the MDM
        .....
        if( pause ) 'insert a delay (use 1 second)'

        if( get_reply )
            {
                // Download the reply (data) string from the MDM (required!)
                .....

                result = jtag_handleReply( reply, reply_len );
                checkResult( result, "handleReply" );
            }
    }
}
```

```
        if( result == 0 )
        {
            // Something is wrong
            result = jtag_replyStatus();
            sprintf( v, "%08X", result );
            DebugN("replyStatus=" + v );
        }
    }

    // Now some other CSM status data should be available
    string str;

    result = jtag_csmVersion();
    sprintf( str, "%02X", result );
    DebugN( "csmVersion=" + str );

    result = jtag_csmVersionDate();
    sprintf( str, "%d", result );
    DebugN( "csmVersionDate=" + str );

    result = jtag_amtPhaseErrors();
    sprintf( str, "%08X", result );
    DebugN( "amtPhaseErrors=" + str );
}

void checkResult( int result, string source )
{
    if( result == 0 )
    {
        // Check for call error or execute error
        dyn_errClass err;
        err = getLastError();
        if( dynlen(err) > 0 )
            errorDialog( err );
        else
            DebugN( source + ": " + jtag_errStr() );
    }
    else
    {
        DebugN( source + ": OK" );
    }
}
```

## 4 Example C++ Code

Below is shown a listing of main C++ program, illustrating the use of the MdtJtagGen class.

```
#include <iostream>
using namespace std;

#include "MdtJtagGen.h"

int main( int argc, char **argv )
{
    MdtJtagGen mjg;

    std::string user = "user";
    std::string pass = "password";
    std::string conn = "connection_string";
    if( !mjg.setDbase(user, pass, conn) )
    {
        cout << "###Error opening dbase:" << endl;
        cout << mjg.errStr() << endl;
    }
    else
    {
        cout << conn << ": " << mjg.dbDefs() << " definitions" << endl;
    }

    std::string chamber = "BOL1A13";
    if( mjg.setChamber(chamber) == false )
    {
        cout << "###" << chamber
            << " " << mjg.errStr() << endl;
        cout << "###dbErrorStatus="
            << hex << setw(8) << setfill('0')
            << mjg.dbErrorStatus() << endl;
    }
    else
    {
        cout << chamber
            << ": " << mjg.dbPars() << " parameters" << endl;
    }

    cout << endl << "Mezz Mask: " << hex << mjg.mezzMask() << dec << endl;

    cout << endl << "SEQ_INIT_CSM450" << endl;

    unsigned char *instr=0, *data=0;
    unsigned char reply[1024];
    int          ilen, dlen, i;
    bool         pause, get_reply;

    mjg.startSequence( SEQ_INIT_AND_START_CSM450 );
    while( mjg.nextStringWithLen( &instr, &data, &ilen, &dlen, &pause, &get_reply ) )
    {
        cout << setfill(' ');
        cout << "ilen=" << setw(3) << ilen << ", "
            << "dlen=" << setw(4) << dlen << ", "
            << "pause=" << pause << endl;
    }
}
```

```
    if( ilen && instr )
    {
        cout << "  instr: " << setfill('0') << hex << uppercase;
        for( i=0; i<ilen; ++i )
            cout << setw(2) << instr[i] << ' ';
        cout << endl;
    }
    if( dlen && data )
    {
        cout << "  data: " << setfill('0') << hex << uppercase;
        for( i=0; i<dlen; ++i )
        {
            cout << setw(2) << instr[i] << ' ';
            if( (i & 0xF) == 0xF ) cout << endl << "          " ;
        }
        cout << endl;
    }

    // Create a dummy reply from the original data
    for( i=0; i<dlen; ++i ) reply[i] = data[i];

    cout << setfill('0') << hex << uppercase;
    if( mjpg.handleReplyWithLen( reply, dlen ) == false )
    {
        cout << "  reply: ERROR, " << setw(8)
            << mjpg.replyStatus() << endl;
    }
    else
    {
        cout << "  reply: OKAY" << endl;
    }
}
return 0;
}
```

This document has been prepared using the Short Note Template provided and approved by the ATLAS TDAQ and DCS Connect Forum. For more information, go to

<http://atlas-connect-forum.web.cern.ch/Atlas-connect-forum/>.

This template is based on the SDLT Single File Template that has been prepared by the IPT Group (Information, Process and Technology), IT Division, CERN (The European Laboratory for Particle Physics) and then converted to MS Word. For more information, go to <http://framemaker.cern.ch/>.