Keck Adaptive Optics Note #515

How to query the TRS from IDL

Erik Allaert, Jimmy Johnson, Erik Johansson and Marcos van Dam

Version 2

2007-Oct-26

1 Introduction

With a lot of the NGWFC telemetry data stored in a relational database, of course the question comes on how to conveniently extract these data and process them with one of the popular (image) processing tools used at Keck, i.e. IDL.

IDL features several techniques to interact with code not written in the IDL language. One of them is to write an *IDL System routine* in C-language, using the standard IDL API foreseen for that purpose. This code can then be packed into a *Dynamically Loadable Module* (DLM) for IDL, i.e. a shared library that can be loaded by the IDL executable at runtime. An IDL system routine provides full integration into IDL and allows direct access to IDL user variables and other data structures. A comprehensive description of these techniques, their advantages, the API etc can be found in the *IDL External Development Guide* – see section 8.1.

At the back-end (from the IDL user's point of view) of this system routine there is the TRS database, running on a dedicated server which is part of the NGWFC. This is a PostgreSQL database, allowing the retrieval of data with standard SQL queries. This can be done interactively with the help of some Postgres utilities, or in C-code using the Postgres API. This API is used internally by the TRS IDL system routine to obtain data from the TRS.

The philosophy followed for the implementation of the TRS-IDL interface has been to make only a minimal number of commands available at the IDL level. The benefit is a greater flexibility for this interface and a simpler implementation – it won't necessarily have to be modified if more data have to be returned in a different order – but of course it also implies that at the IDL level some additional parameters will have to be provided and dealt with. There is no application knowledge built into the interface. This has to be developed in the IDL routines using the interface, adjusting the options and arguments as needed.

2 Requirements

The following are the requirements to be fulfilled by the TRS-IDL interface:

R1. It must be possible to extract (i.e. read) any data item stored in the TRS database
R2. Data is returned in an IDL structure. Each field in the query will be a member of the structure and have the same field name. Each structure field is an array of basic types, where the array depth equals the number of rows returned.
R3. The primary selection criterion will be the timestamp of the database record (as this is also how the TRS database is organized)
R4. It must be possible to extract the data corresponding to a single specific time
R5. It must be possible to extract chronologically consecutive data starting from a given time (i.e. data-timestamp greater than a specific value)
R6. It must be possible to extract chronologically consecutive data up to a specific time (i.e. data-timestamp smaller than a specific value)
R7. data will be returned in the correct byte-order, i.e. eventual byte-swapping, due to different endianness of the machines involved, will be transparent to the IDL user.
R8. detector data will be returned in unscrambles order, i.e. eventual pixel re-ordering, due to the way the CCD was read-out and data got stored in the TRS database, will be transparent to the IDL user.

3 Build and release instructions

3.1 Build requirements

On the workstation where the IDL Dynamically Loadable Module for querying the TRS will be installed, the following software must be available:

- SUN Solaris 8 or 9
- GNU C-compiler 2.91.66 or 2.95.2
- GNU make 3.74 or 3.78.1
- IDL 6.0 installed under `/sol/apps/idl_60/idl_6.0`
- PostgreSQL 8.1.3, installed under `/opt/csw/postgresql/`

The installation *might* work with other (versions of the) tools than the ones listed, but these are the only ones with which the installation has been tested. If IDL and/or PostgreSQL are installed under different directories, you will have to edit the `Makefile` to make these changes – and hope for the best.

3.2 Procedure

1. Retrieve the `trsClient` sources from CVS
   (Section to be completed after Erik Johansson defines the NGWFC software directory structure).
2. From the directory containing the `trsClient` sources, issue the following command:
   ```
   gmake all
   ```
   This will compile the sources and generate the shared library `trsClient.so`[1].
3. Install `trsClient.so` into a directory which is searched during dynamic loading, when IDL runs. The list of these directories is (for Solaris and Linux) contained in the environment variable `LD_LIBRARY_PATH`, i.e. either you install `trsClient.so` into one of these directories, or you modify `LD_LIBRARY_PATH` to include the path where this shared library will reside. The LD_LIBRARY_PATH is set in the .cshrc file on k1aoserver and k2aoserver..
   (Section to be completed after Erik Johansson defines the NGWFC software directory structure.)
4. Install the IDL module description file `trsClient.dlm` into a directory which is searched by IDL at start-up. The list of these directories is contained in the environment variable `IDL_DLM_PATH`[2], i.e. either you install `trsClient.dlm` into one of these directories, or you modify `IDL_DLM_PATH` to include the path where this module description file will reside. The IDL_DLM_PATH is set in the .cshrc file. The IDL variable is !dlm_path.
   (Section to be completed after Erik Johansson defines the NGWFC software directory structure.)

[1] Currently only the 32-bit version of the library is generated. This is why IDL needs to start-up in 32-bit mode, and the PostgreSQL libraries needed/accessed are the 32-bit ones. Although the Makefile contains provisions for generating the 64-bit version, this has not been tested.
[2] IDL will actually first search the current working directory for DLMs, before looking into the directories listed in IDL_`DLM_PATH`.

4 Usage

4.1 Runtime requirements

On the workstation where the IDL Dynamically Loadable Module for querying the TRS will be used, the following software must be available:

- SUN Solaris 8 or 9
- IDL 6.0
- PostgreSQL 8.1.3

The installation *might* work with other (versions of the) tools than the ones listed, but these are the only ones with which the DLM has been tested.

Before starting IDL, ensure also that the following conditions are met (this is done in the .cshrc file):

The environment variables `LD_LIBRARY_PATH` and IDL_DLM_PATH are set appropriately (see section 6.4 for technical details). In csh syntax this becomes e.g.[3]:

```
setenv LD_LIBRARY_PATH ${LD_LIBRARY_PATH}:/opt/csw/postgresql/lib
setenv IDL_DLM_STARTUP '<IDL_DEFAULT>':/home/k1obsao/trsclient
```

The IDL startup script, pointed to by the environment variable `IDL_STARTUP`, should include the TRSHOST setting if you don't want to deal with that later on:

```
setenv IDL_STARTUP ~/startup.pro
```

In `~/startup.pro`, include the lines

```
TRSDEBUG=1 and

TRSHOST = '128.171.95.122' or whatever the IP address for the TRS
host is
```

IDL must start up in 32-bit mode:

```
k1aoserver userX: 11 > idl
starting IDL 6 ...
IDL Version 6.0, Solaris (sunos sparc m32). (c) 2003, Research Systems,
Inc.
...
```

The following subsections assume that the above conditions are met, and that IDL is up and running.

4.2 Querying the TRS database

To query the TRS-database, the interface needs to know where this PostgreSQL server is. This can be indicated via an optional keyword (see below) in the function call, or by setting an IDL variable as follows:

```
IDL> TRSHOST=hostname
```

with *`hostname`* a string containing either the hostname (must be resolvable by the name-server used by the IDL host) or the IP address of the TRS workstation to be queried. When the `trsClient.so` shared library is loaded, this variable will be defaulted to 'localhost' unless it is already defined (e.g. in the IDL startup script).

The `TRSHOST` setting can be modified anytime, and all subsequent queries will use the updated setting. After this variable has been set, the fields of the various tables in the TRS database can be queried typically as follows:

[3] Assuming that the IDL and trsClient shared libraries are installed in directories already included in `LD_LIBRARY_PATH`; otherwise, also these directories will have to be appended to `LD_LIBRARY_PATH`.

`IDL> y=trsquery(timestamp, samples, tablename [,fieldname [, fieldname]])`
whereby the parameters are as follows:

- *timestamp*: either an integer (as stored internally by TRS), or a string containing a timestamp in ISO-format ("YYYY-MM-DDThh:mm:ss.ffffff"). If it is set to -1 or the string 'now', the query will return the most recent data. If a specific timestamp is given, the data returned will be by default starting from that timestamp. See also below for the *condition* keyword.
- *samples*: a positive integer, limiting the number of elements returned by the query
- *tablename*: a string with the name of the table, as known by the TRS database. This is either 'dfb', 'ffb', 'strap', 'configuration' or 'nonrtc'.
- *fieldname*: a string with the name of the field, as known by the TRS database. This is e.g. (for the 'dfb' table) 'rawpixel' or 'timestamp'. There can be up to 100 fieldnames passed as arguments. If none is given, all fields in the table will be returned.

The result will be stored in the IDL variable y, as an anonymous structure. Data are always returned in chronological order (i.e. higher index means more recent data).

Example:

```
IDL> TRSHOST='128.171.96.51'
IDL> y=trsquery(-1, 20, "ffb", "residualrms", "ttcommands", "timestamp")
% Loaded DLM: TRSCLIENT.
IDL> help,y,/str
** Structure <98748>, 3 tags, length=400, data length=400, refs=1:
   RESIDUALRMS     FLOAT     Array[20, 1]
   TTCOMMANDS      FLOAT     Array[2, 20]
   TIMESTAMP       ULONG64   Array[20]
IDL> print,y.ttcommands[1,15]
```

Note that the byte-array data in the 'ffb', 'dfb' and 'strap' tables are stored internally in the TRS in little endian order (as they get stored as a byte stream read-out from an Intel CPU) – the TRS does not know or care that e.g. the 'rawpixel' data are 16-bit unsigned integers. Consequently, Postgres will not automatically do the byte-swapping as necessary when it receives a query for these data. This is instead done by the `trsQuery()` software.

The following IDL optional keywords are also supported within the `trsquery()` function:

- *close*: flag indicating whether the database connection should be closed upon termination of the query. By default, this connection will be left open until IDL exits. By default, this connection will be left open until IDL exits.
- *condition*: boolean operator on the timestamp, permitting to select ascending or descending from the timestamp specified. Defaults to '>' (data more recent than timestamp given). Can be set to '>', '>=', '<', '<=' or '='.
- *trshost*: hostname or IP-address of the PostgreSQL server; this value will also be stored into the IDL variable `TRSHOST`, i.e. subsequent calls to trsquery() without this keyword will use this same value.
- *sqlstmt*: the presence of this keyword flags that the first argument of `trsquery()` is to be sent directly to the TRS server as an SQL statement. This provides quite some flexibility to `trsquery()`, allowing it to retrieve basically anything which is stored in the TRS database.

Example 1:

```
IDL> y=trsquery("select * from ffb where timestamp < 1000", /sqlstmt,
trshost="kuhio")
IDL>
```

If you want the most recent data (instead of data from/to a specific time), you can also obtain them via an SQL statement similar to:

```
IDL> y=trsquery("select * from ffb order by timestamp desc limit 50",
/sqlstmt)
IDL>
```

Note that the above query will give the data in reverse chronological order (i.e. array index 0 corresponds to the most recent data). Additional SQL operations are needed to put it in ascending chronological order:

```
IDL> y=trsquery("select * from (select * from ffb order by timestamp desc
limit 50) as dummy order by timestamp asc", /sqlstmt)
IDL>
```

Example 2: configuration paramenters

```
IDL> y=trsquery("select * from get_conf(65571, ts2conf(1000))", /sqlstmt)
IDL>
```

The above example will return the value of the configuration parameter with ID 65571, at the time corresponding to a timestamp with value 1000. This result will be in a structure containing the `conf_id`, `par_id` resp. `par_val`. If only the value itself is wanted, it can be obtained as follows:

```
IDL> y=trsquery("select par_val from get_conf(65571, ts2conf(1000))",
/sqlstmt)
IDL>
```

If all configuration parameters for a particular timestamp can be obtained by an SQL statement like the following one:

```
IDL> y=trsquery("select * from get_conf_all(ts2conf(1000))", /sqlstmt)
IDL>
```

4.3 Timestamp conversion

The TRS database has timestamps stored as 64-bit unsigned integers. The value of such a timestamp is the number of hundreds of nanoseconds since the start of the calendar year. The `trsClient.so` shared library includes an IDL function to convert this integer to a human-readable format, and vice versa. It can be called as follows:

```
IDL> y=trstimestamp(timestamp)
```

whereby *`timestamp`* is either an integer (as stored internally by TRS), or a string containing a timestamp in ISO-format ("YYYY-MM-DDThh:mm:ss.ffffff"). In the former case, an ISO-format timestamp will be returned, in the latter an integer as dealt with by TRS. The variable *`timestamp`* can be either a scalar or an array, and the return value will be of the same type as the argument.

Example:

```
IDL> y=trsquery(-1, 5, "ffb", "residualrms", "ttcommands", "timestamp")
IDL> print,y.timestamp[0:1]
       198276949637254       198276949646743
IDL> z=trstimestamp(y.timestamp[0:1])
IDL> help,z
Z               STRING    = Array[2]
IDL> print,z
2006-08-18T11:41:34.9637254 2006-08-18T11:41:34.9646743
IDL> x=trstimestamp(z)
```

```
IDL> help,x
X               ULONG64   = Array[2]
IDL> print,x
       198276949637254       198276949646743
IDL> y=trstimestamp(123.45)
% TRSTIMESTAMP: timestamp of wrong type
% Execution halted at: $MAIN$
IDL> x=trstimestamp(-2)
% TRSTIMESTAMP: timestamp out of range
% Execution halted at: $MAIN$
IDL> IDL> y=trstimestamp("not a timestamp")
% TRSTIMESTAMP: not an ISO timestamp
% Execution halted at: $MAIN$
IDL>
```

5 Error handling in IDL

If the execution of `trsquery` or `trstimestamp` has problems, these commands may return an error to IDL. Such errors can be handled within IDL applications in the standard way, with the help of the `CATCH` and `ON_ERROR` commands. The following sample code shows how this can be done:

```
; outer procedure
PRO proc1
print, 'begin proc1'

retry = 0
; establish error handler. When errors occur, errorVar will have the
; index of the error. It is initialized to zero.
catch, errorVar
if errorVar ne 0 then begin
    ; course of action could be made dependent on kind of error,
    ; by looking at !ERR_STRING. This is not done here.
    retry = retry + 1
    if retry gt 3 then begin
        print, '3 successive errors - that is enough'
        ; cancel the error handling - we won't need it anymore
        catch, /cancel
        ; the call to proc2 is automatically repeated by IDL
        ; after catching the error. As we don't want that after
        ; our 3 retries, we explicitly jump to the point beyond that.
        ; (Alternatively, we could just as well 'return'.)
        goto, myLabel
    endif
endif

; the next call will return an error, which we'll catch - see
; above. After catching the error, IDL will re-execute this command,
; unless the error-handler jumps to another place.
proc2

myLabel: print, 'end  proc1'
end

;=====================================================
; inner procedure
PRO proc2
print, 'begin proc2'

; on error within this proc, return to the caller of this proc.
; this is the recommended approach for debugged procs.
ON_ERROR, 2

; with an IP address that does not correspond to TRS we'll get an error
y=trsquery("select * from dfb", /sqlstmt, trshost='192.168.1.2')

; should never get here, due to the error above.
print, 'end  proc2'
end

;=====================================================
print, 'main will now call proc1, which calls proc2'
proc1
print, 'main has finished calling a chain of procs'

end
```

A detailed explanation of IDL error handling can be found in the IDL documentation (e.g. the "*Building IDL Applications*" manual, the chapter on "*Controlling Errors*")

For the `trsquery()` function, the following conditions will lead to an IDL error:

- the IDL variable TRSHOST has been unset
- too few or too many arguments
- failure to connect to the TRS database within a timeout of 10 seconds
- failure to reset a TRS database connection after it went bad
- failure of the Postgres query
- the first argument is not a proper timestamp, nor an SQL statement
- the second argument (sample count) is not of the proper type – it must be an integer
- one of the fieldnames passed does not exist in the TRS database
- `IDL_MakeStruct`, `IDL_MakeTempStruct` or `IDL_StructTagInfoByIndex` failure (should not happen)
- the query did not yield any data

For the `trstimestamp() function`, the following conditions will lead to an IDL error

- too few or too many arguments
- the argument passed is a structure instead of a scalar or array
- the argument is not a proper timestamp, or is out or range

6 Internals

6.1 Files

The trsClient software contains the following files under the `src/` subdirectory:

- `Makefile`
- `trsClient.c`: has the initialization code executed when the `trsClient.so` shared library is first loaded, i.e. the function `IDL_Load()`
- `trsClient.h`: include file, used by `trsClient.c`, `trsQuery.c` and `trsTimestamp.c`
- `trsClient.dlm`: the IDL module definition file
- `trsQuery.c`: contains the code for the IDL function `trsquery()`, plus all related auxiliary functions (see below).
- `trsTimestamp.c`: contains the code for the IDL function `trstimestamp()`.
- `convertTimestamp.c`: contains the auxiliary functions `iso2trs()` and `trs2iso()`, to convert a single timestamp from ISO format to a 64-bit unsigned integer, and vice versa.

The files contained under the `test/` subdirectory are listed in chapter 7.

6.2 Calling sequence

The internal calling sequence is as follows, (within square brackets means logical operation):

```
TRSQuery
      PrepareStatement
      [ExecuteSqlStatement]
      ProcessResults
            CreateFieldInfoFromResults
            GetColumnMetaData
            ConvertToIDLVar
      IDL_MakeStruct
      CopyIDLData
      [PackResults]
      FreeAllocatedMem

TRSTimestamp
      [CreateTempArray]
      [ExtractInputData]
      iso2trs/trs2iso
```

6.3 TRSDEBUG

The IDL variable `TRSDEBUG` is used to control the printing of debugging messages during the execution of the IDL function `trsQuery()` and `trsTimestamp()`. It can be set to an integer number. The higher the number, the more verbose these debugging messages become. If `TRSDEBUG` is not set, no debugging messages will appear. See also the `trsDebug` macro declared inside `trsClient.h`, and the `debugPrint()` function defined inside `trsClient.c`.

A debug-message will be printed by IDL in the following format:

```
% COMMAND: timestamp [requiredLevelToShow] file:lineNr: message
```

Example:

```
% TRSTIMESTAMP: 2006-08-25T16:45:58 [7] convertTimestamp.c:149:
End   trs2iso, timestamp = '2006-01-01T00:02:03.4567890'
```

6.4 IDL environment variables

In order for IDL to find the trsClient shared library and its dependent PostgreSQL share library, the environment variables `LD_LIBRARY_PATH` and `IDL_DLM_PATH` need to be set:

- The environment variable `LD_LIBRARY_PATH` must include the directory where the 32-bit versions of IDL 6.0 (default: `/sol/apps/idl_60/idl_6.0),` the PostgreSQL libraries (default: `/opt/csw/postgresql/lib`) and the `trsClient.so` shared library reside:

  ```
  kuhio userX: 9 > echo $LD_LIBRARY_PATH
  /sol/apps1/dv/lib:/usr/lib:/usr/dt/lib:/sol/apps/lang:/usr/NX/lib:/usr/la
  ng:/usr/openwin/lib:/usr/local/lib:/opt/csw/postgresql/lib
  kuhio userX: 10 >
  ```

  `Note` that `LD_LIBRARY_PATH` could also be (re)set by the IDL start-up script, so in any case also check from within IDL:

  ```
  IDL> $ echo $LD_LIBRARY_PATH
  ```

  If this does not give the expected result, correct it and restart IDL.

- The module description file `trsClient.dlm` must either be in the current working directory or in one of the directories given by the environment variable `IDL_DLM_PATH`. In other words, also the setting of the environment variable `IDL_DLM_PATH` needs to be verified:

  ```
  kuhio userX: 10 > echo $IDL_DLM_PATH
  <IDL_DEFAULT>:/usr/local/idl_60/idl_6.0/external/dlm
  kuhio userX: 11 >
  ```

  Note that the `<IDL_DEFAULT>` component in the example above is resolved by IDL when it starts up.

  Same as for `LD_LIBRARY_PATH`, also check from within IDL:

  ```
  IDL> $ echo $IDL_DLM_PATH
  ```

  If this does not give the expected result, correct it and restart IDL.

6.5 Data re-mapping

Some of the data stored in the TRS database appear to be 'scrambled', i.e. they are not recorded in the order one would logically expect. This is the case for e.g. the `rawpixel` data, which are for efficiency reasons stored in the order as the pixels are read out from the CCD. For processing with IDL, these arrays are first re-mapped after they are read from the TRS database. This is done within the trsClient software, completely transparent to the IDL user.

The vectors used for remapping these TRS data are stored in the TRS database, and read-out when the trsClient starts up – modifications to these data after the trsClient has started will not be taken into account by the `trsquery()` command till the IDL session is restarted.

Use the "insertMapping" utility to re-insert these data into the TRS 'datatype' table, as follows:

```
insertMapping trsHost arrayName ccdName binning subaperture file
```

with

- `trsHost`: IP address or (resolvable) hostname of the TRS host
- `arrayName`: fieldname of the TRS table (ffb , dfb, strap, configuration) to which the subsequent data apply

- `ccdName`: name of the CCD to which the subsequent data apply
- `binning`: binning factor to which the subsequent data apply (0 means apply to any binning)
- `subaperture`: size of the subaperture (0 means apply to any subaperture)
- `file`: name of the file containing the remapping vector. This is a file with binary data, 4-byte integer per element, big endian order

Examples:

```
insertMapping 128.171.96.51 rawpixel CCD39 1 4 pixMap_CCD39_bin1_subap4
insertMapping 128.171.96.51 offsetcentroid CCD39 0 0 centRemap
```

As can be seen from above, the remapping vector can be configuration dependent, i.e. it can depend on the binning and subaperture size – as is the case for `rawpixel` data from `dfb`. When such `rawpixel` data are requested, the proper binning and subaperture size need to be determined first in order to know which remapping vector to use. They are retrieved from the TRS database via the `conf_id` applicable to the data set. Note however that this `conf_id` is cached within `trsQuery`, i.e. subsequent queries of rawpixel arrays will not retrieve the binning and subaperture size from the database again if the `conf_id` has not changed.

6.6 Data types

The TRS database contains an additional configuration table (‘`datatypes`’) which tells what kind of data-type is stored in the various arrays in the dfb, ffb, strap and configuration tables. This implies that any modification of the TRS database structure of any of the latter tables needs to be reflected by corresponding modifications inside this additional configuration table.

Use the “insertDataTypes” utility to re-insert these data into the TRS ‘`datatypes`’ table, as follows:

```
insertDataTypes trsHost file
```

with

- `trsHost`: IP address or (resolvable) hostname of the TRS host
- `file`: name of the file containing the data type definitions This is a text file, in the following format:
    - A line starting with '# ' is a comment line and will be ignored.
    - Blank lines will also be ignored.
    - For each field in the union of the dfb, ffb, configuration and strap tables there is a row with 5 columns:
        1. the fieldname
        2. a boolean indicating if the field is an array or scalar
        3. the IDL data-type of a single element, as a text-string
        4. the base-size in bytes of a single element of this field
        5. the endianness of the data element (-1 for network order, 0 for little endian, 1 for big endian)
    - These columns must be separated by at least 1 space, tab or comma.

Examples:

```
insertDataTypes 128.171.96.51 datatypes
```

6.7 Caveats

1. Currently the `trsquery()` IDL function creates the connection to the TRS database on the first `trsquery()` call, and leaves it by default open until IDL exits (via a registered exit handler). An unexpected termination (e.g. via "`kill -9`") will not clean-up the database connection on the TRS server side, and if repeated this server may in the end run out of connection resources.
2. The *`samples`* parameter of the `trsQuery()` call (see section 4.2) does not currently have an upper limit. This may lead to requests of huge amounts of data, with the corresponding consequences on the time this takes to execute, the load on the network and other system resources like memory.
3. The TRS configuration table does not really include the binning factor. What it does contain is the actual number of detector pixels after binning. To derive the binning factor, one needs the dimension of the CCD – but that number isn't stored either, nor is there a CCD identifier (which would allow deriving the CCD dimension). The current work-around is to hardcode some of these numbers and use a compilation flag for the trsClient code to "select" between detectors; for that reason the trsClient library can currently only work for one type of detector. Fixing this requires first the addition of the referred parameters in the configuration table.
4. The `insertMapping` and `insertDataTypes` utilities (see sections 6.5 and 6.6) should be run from the TRS host as part of the procedure to (re)construct the TRS database. This implies that these hosts need access to the original mapping-files stored under $KROOT on a different machine. Neither of this is the case, i.e. the TRS hosts do not yet have access to the authoritative versions of the mapping-files, and the procedure to build the TRS database does not yet include the running of these utilities. They currently have to be invoked by hand from another host (e.g. kuhio).
5. A call to `trsquery()` containing a direct SQL statement will return the data as stored in the PostgreSQL database, with byte-swap for endianness but without unscrambling. This is because `trsquery` passes such SQL query on without parsing it, i.e. it does not know what kind of data have been requested (and in particular not if the `conf_id` is part of these data, which is needed to determine the proper remapping vector). For that reason, the use of a direct SQL statement in a `trsquery()` call should only be used for engineering purposes.
6. In order to retrieve the binning and subaperture size corresponding to a particular `conf_id`, `trsQuery()` queries the TRS database restricting the search to the `par_id`s corresponding to these parameters. The numeric values of these `par_id`s are actually defined as macros in the file `wifPublic.h`, but this file can in its current state not be included for compilation on Solaris as it contains definitions of structures that are for VxWorks only. Therefore, these 2 particular macros (`SET_SUBAPERTURE_SIZE` and `SET_RAW_PIXEL_NR`) are simply copied over from `wifPublic.h` into `trsClient.h`.
7. The real-time data stored in the TRS do not lend themselves very well for testing; it is very hard for an automated test procedure to judge if e.g. the byte-swapping and/or data-remapping functions work OK if the values cannot be predicted – or even worse, if you don't know in advance how many (if any) values are stored in the database. For that reason, it would be very helpful to have either extra tables or extra fields in existing tables with fixed test-data, to be used for regression tests.

7 Regression tests

The `trsClient` module includes under the `test/` subdirectory a test-suite that permits to run an automated test exercising the `trsClient` shared library. It will compare the output sent during these tests to `stdout` with a reference file, and report if the test passed or failed, based on that comparison.

The files dealing with these tests, and their function, are the following ones:

- `trsTest.list`: file describing the different tests, one line per test. Lines starting with a pound-sign are considered comment-lines. A non-comment line describing a test must have 3 columns:
  1. test number; must be a unique integer number, not necessarily in order or consecutive.
  2. name to give to this test. This will be the base-name for the `.grep`, `.sed`, `.out` and all other test-related input/output files
  3. command to execute; can have any amount of arguments.
- `trsTest`: shell script executing the actual test. It has the following steps:
  1. Process the arguments (-v = verbose, -g = generate reference file, -h = print usage string and stop, remaining arguments = list of test numbers to execute [default = all tests]).
  2. Prepare the test
     - set up the generic environment variables (PATH, LD_LIBRARY_PATH)
     - set up the environment variables applicable to all tests in this test-suite (IDL_DLM_PATH) by source-ing the file `trsTest.env`
  3. For each of the tests listed in `trsTest.list`, do the following:
     - Skip to next test if this test does not need to be executed.
     - Set up the test-specific environment variables (e.g. IDL_STARTUP) by source-ing the file <testName>`.env`
     - Run the test, issuing the command as defined in `trsTest.list`
     - Do the filtering of the output:
       - Apply the generic `trsTest.grep` reverse grep filter
       - Apply the test-specific `<testName>.grep` reverse grep filter
       - Apply the generic `trsTest.sed` filter
       - Apply the test-specific `<testName>.sed` filter
     - Compare the filtered output with the reference file `ref/<testName>.ref`

  Tests are executed in the order as listed in `trsTest.list`, independent from both the test-number they have been assigned and the list of test-numbers passed as an argument to `trsTest`.
- `Makefile`: has a '`test`' target which creates the IDL script `trsQuery.pro`, by inserting the setting for `TRSHOST` and merging the file `trsQuery.idl`. The setting for `TRSHOST` is either picked up from the environment variable of that name, or from

the argument '`TRSHOST=<myIpAddress>`' passed to `gmake`.
The second relevant target in this `Makefile` is the '`clean`' target, which will remove all generated output files.

- `trsQuery.idl`: file with IDL commands, not including the TRSHOST setting, used to "make" the file `trsQuery.pro`. The latter file is needed for the execution of the trsQuery test.
- `trsTimestamp.pro`: IDL test script for the trsTimestamp test.
- `trsTest.env`: environment variables that need to be set-up for all tests in the test-suite. Each test can also have its own specific setting for any environment variable, e.g. there is a `trsQuery.env` and a `trsTimestamp.env` as well. These specific settings are sourced after the general one.
- `trsTest.grep`: general test-suite reverse grep filter for test-output, i.e. a list of patterns (1 per line) that tells which lines have to be filtered out from the original output produced by running IDL. Each test can also have its own specific reverse grep filter, i.e. there can be a `trsQuery.grep` and a `trsTimestamp.grep` as well. The specific filters are applied after the general one.
- `trsTest.sed`: sed editing commands to filter the output produced by running IDL (*after* it was processed with '`grep –f trsTest.grep`'). This file is the general test-suite sed filter, applied to all tests. Each test can also have its own specific sed filter, i.e. there is a `trsQuery.sed` and a `trsTimestamp.sed` as well. The specific filters are applied after the general one.
- `ref/<testName>.ref`: reference output files, with which the (filtered) output files will be compared
- `out/<testName>.out`: original output after running the tests
- `out/<testName>.out_filtered`: the output files processed by the generic and specific reverse grep filters, and then by the sed filters
- `out/<testName>.diff`: output from executing '`diff <testName>.out_filtered <testName>.ref`'

8 Documentation

8.1 IDL External Development Guide

The IDL External Development Guide is from Research Systems Inc./ITT Visual Information Solutions. It can be obtained directly from http://www.rsinc.com/idl/pdfs/edg.pdf. Alternatively, you can start from http://www.rsinc.com/idl/pdfs/onlguide.pdf; this latter PDF document contains links to plenty other IDL documents, including the External Development Guide. Remark however that this RSInc website contains only docs for IDL version 6.1. The up-to-date website for IDL (http://www.ittvis.com/) unfortunately does not seem to contain anymore links to downloadable manuals.

If you intend to edit one of the TRS-IDL system routines, and are new to DLMs, you should first of all read the chapters named "*IDL Internals: Types*", "*IDL Internals: Variables*", "*IDL Internals: Keyword Processing*" and "*Adding System Routines*"[4]. Chances are that with this background you will not have to read the rest of this manual.

8.2 PostgreSQL 8.1.0 Documentation

Another relevant part in the TRS-IDL system routine is the interface to the TRS database, which is a PostgreSQL database. You can easily get to the online PostgreSQL documentation via the link http://www.postgresql.org/docs/. In there you should look in particular to the chapter labeled "*libpq – C library*" (chapter 28 for version 8.1.0).

[4] The numbers of these chapters depend on the version of this guide. For version 6.1, these numbers are (in the order as listed) 11, 12, 13 and 21. For version 6.3 they are 5, 6, 7 and 15.

9 TRS database lay-out

```
           List of relations
 Schema |     Name      | Type  | Owner
--------+---------------+-------+-------
 public | configuration | table | trs
 public | dfb           | table | trs
 public | ffb           | table | trs
 public | nonrtc        | table | trs
 public | strap         | table | trs

     Table "public.configuration"
 Column  |  Type   |
---------+---------+-----------
 conf_id | integer |
 par_id  | integer |
 par_val | bytea   |

Indexes:
    "configuration_conf_id" btree (conf_id)
    "configuration_par_id" btree (par_id)

          Table "public.dfb"
    Column    |   Type   |
--------------+----------+-----------
 buffertype   | smallint |
 recordlength | smallint |
 timestamp    | bigint   |
 framecounter | integer  |
 rawpixel     | bytea    | int16
 rawpixelsize | smallint |

Indexes:
    "dfb_index" btree ("timestamp")


            Table "public.ffb"
      Column       |   Type   | Modifiers
-------------------+----------+-----------
 buffertype        | smallint |
 recordlength      | smallint |
 timestamp         | bigint   |
 framecounter      | integer  |
 subapintensity    | bytea    | float32
 offsetcentroid    | bytea    | float32
 residualwavefront | bytea    | float32
 dmcommand         | bytea    | float32
 dmcommandsclipped | bytea    | int32
 residualrms       | bytea    | float32
 conf_id           | integer  |
 ttcommands        | bytea    | float32
 ttstraingauge     | bytea    | float32
 ttcommandsclipped | bytea    | int32

Indexes:
```

```
    "ffb_index" btree ("timestamp")

           Table "public.strap"
       Column       |  Type   | Modifiers
--------------------+---------+-----------
 timestamp          | bigint  |
 framecounter       | integer |
 conf_id            | integer |
 apdcounts          | bytea   | float32
 dttcentroids       | bytea   | float32
 dttcommands        | bytea   | float32
 dttstraingauge     | bytea   | float32
 dttcommandsclipped | bytea   | int32
Indexes:
    "strap_index" btree ("timestamp")
```