ACQGEN: Data acquisition and real time control with *MatLab*

A.T. Hofkamp

release-v1-RC3

1 Introduction

Real-time controller design is often done in an environment like *MatLab*. Designs developed here can be translated to C, giving a controller implemented in C that has sufficient performance. However, to actually control some physical device, a bridge between the hardware sensors and actuators, and the designed controller is required. Normally, this again has to be manually implemented in the C language.

The ACQGEN program offers another path. It generates (and compiles) a *MatLab-Mex* function that converts input and output data-value streams from sensors and actuators to streams of input and output *MatLab* matrices containing these values. Connection to the hardware sensors and actuators can be expressed in *MatLab*, making the connection to a physical device much simpler.

Since the ACQGEN program provides hardware sensors and actuators access at *MatLab* level, this program can also be used for other uses where such access is required. A common example is data acquisition, where input is sampled with or without controlling actuators. In this area, the ACQGEN program provides features such as generating commonly used output signals, filtering and triggering input signals, and pre-writing input streams.

MatLab and *Mex* are owned by Mathworks, *EtherCAT* is owned by Beckhoff.

2 Installing and usage

Installation is described in Section 6. Using ACQGEN to conqueror the real-time world only takes five simple steps.

1. Define desired functionality of the ACQGEN generated module in a configuration text file (see examples in Section 3 and the reference documentation in Section 4).
2. Generate and compile the *Mex*-C source file using ACQGEN.
3. Use the module in your *MatLab* data acquisition or control program. Section 5 explains the details of using it.
4. ???
5. Profit!

3 Examples

Below are a few example configuration files. First a simple input/output configuration without buffering by the generated module.

```
config(memsize = 32000,
    nic = "eth5", # Modify to the ethernet device connected to EtherCaT
    frequency = 1000, # hz
    buffered = false
);

# Matlab gets samples from ain[0] and din[1].2, the first analogue input, and
# bit 2 (3rd bit) of the second digital input.
input(ain[0], din[1].2);

# Matab provides analogue control data to aout[0] and aout[1].
aout[0], aout[1] = output();
```

The `config` sets up the general configuration, providing memory for IO (and buffers, if running in buffered mode). It also defines the ethernet port to use for connecting to *EtherCAT*, and the frequency of sampling/controlling. The `buffered = false` means the generated module runs in unbuffered mode, it does not do any buffering, output data is directly given to the *EtherCAT* master, and input samples are directly returned.

The `input` and `output` blocks represent the generated module, it 'takes' analogue and digital inputs, and 'provides' values for the analogue outputs.

In the unbuffered mode, the generated *Mex* module takes one input array consisting of one column with real values for the outputs. The module returns two output values. The first value is an error string (empty if no error happened), the second value is a real array with one column containing the data values retrieved from the *EtherCAT* master.

The application that uses the generated module should call the module at the indicated frequency. If the application calls the module earlier, it will wait until the right time has been reached.

While the above is often sufficient with unbuffered input/output modules, the generator can also handle functions, filters, and triggers here (as shown below).

The example in buffered mode is more elaborate. Here, actuators are not driven from the application, they are controlled by the generated module. The sensor input is used to return measurements around a particular point in the (periodic) input signal.

```
config(memsize = 32000,
    nic = "eth5", # Modify to the ethernet device connected to EtherCaT
    frequency = 1000, # Frequency in Hz
    buffered = true,
    default_input_buffersize = 100,
    default_input_buffercount = 2,
    default_prewrite = 0.2
);

# aout[0] provides 100Hz sine wave at sample frequency 1000Hz, between -4 and 4.
aout[0] = function(shape=sine, frequency=100, base=0.0, amplitude=4.0);
```

```
# Filter ain[0] by averaging the last 3 samples.
fout = filter(ain[0], weights=[0.33, 0.33, 0.33]);

# Trigger on upgoing flank of filter output, passing 1.3. Trigger output is
# high for 100 samples (= 100/1000 = 0.1 seconds).
tout = trigger(fout, policy=up, value=1.3, count=100);

# Save and return raw ain[0] when triggered by tout, with 20% prewriting.
input(ain[0], index = 1, enable=tout);
```

In buffered mode, a separate thread performs the data IO to the hardware. The generated module takes a cell array as input, which should contain real matrices indexed by the `index` parameters of the `output` blocks. (This example has none, so it should provide an empty cell array.) The module returns two values, an error string as the first output, and a cell array with matrices from the inputs, indexed on the value of the `index` parameter of the `input` blocks. In each matrix, a row represents a device, in the same order as listed in the `input` block, each column contains values from all sensors at a single point in time.

4 Specifying functionality

The configuration file uses two main concepts, *signals* and *blocks*. A block is a piece of functionality, a signal is a 'wire' to connect blocks to each other.

In addition, the program knows two modes of running, *unbuffered* mode and `buffered` mode. A configuration is specific for one of these modes. In the unbuffered mode, there is nothing between the controller and the hardware IO. The generated *Mex* module has one input and one output block, it takes a matrix with one column of output for the actuators, and returns one column of samples from the sensors (and an error message, if it occurred). The module must be called within each sample period, although it waits if it is called too early. In buffered mode, several input and output blocks may be used, and the generated *Mex* module exchanges matrices with several columns of data. That data is buffered inside the module, and actual hardware IO is handled by a separate thread, thus decoupling matrix IO streams from hardware sample IO streams.

4.1 Signals

A signal is written by exactly one block, and is used by one or more other blocks. Cycles in signals are not allowed, a signal cannot (directly or indirectly) attach to the block that writes the signal.

Each signal has a unique name. Signals that don't leave the computer (for example a signal that connects a filter block output with a trigger block input) use a single word as name. Any word will do as long as it is unique, although it is recommended to use a word that has a useful meaning about the purpose of the signal.

Signals connected to the hardware have names that point out exactly what it connects to. For example, the signal named `el1014[0]:din.0` is connected to the first *el1014* box. It reads the *din* port (digital input), bit number *0* (the lowest bit). If a port does not have bits, or you are not interested in single bits (that is, you want all of them), drop the `.0` part.

Currently known hardware IO boxes are `ebox`, `el1004`, `el1008`, `el1014`, `el1018`, `el1819`, `el2004`, `el2008`, `el2809`, `el3102`, `el3104`, `el4038`, `el4132`, `el4134`, `el5002`, `el5101`, and

`el5152`. Currently known port-types are `din` (digital input), `ain` (analogue input), `enc` (encoder/counter input), `dout` (digital output), `aout` (analogue output), and `pwm` (pulse width modulator). If a box has several of the same type of ports, an index number is appended, like `aout0` (first analogue output of the box), `aout1` (second analogue output of the box), etc. Only the digital input and digital output port types have bits that can be selected.

In case you don't know or don't care what box is being used for a signal to a port, it is also allowed to drop the box part. For example, the signal named '`enc[3]`' connects to encoder number three, the fourth available encoder input in the system (unlike *MatLab*, electronics uses 0-based numbering).

Note: At the time of writing, the *acqgen* program does not have knowledge about the port types of each type of hardware box. This may lead to failures in finding a source or a destination for hardware signals while using the generated *MatLab* module.

4.2 Blocks

The general notion of a block is a piece of functionality that takes signals and parameter values as input, and produces output to other signals, for example

```
filtered = filter(ain[0], weights=[0.1, 0.7]);
```

This `filter` block takes samples from input signal `ain[0]`, applies the filter function to the samples using the `weights` parameter value `[0.1, 0.7]`, and writes the results to the `filtered` signal. (Details about `filter` blocks are in Section 4.2.4.)

The general pattern of a block is

⟨output-signals⟩ = ⟨block-name⟩**(**⟨input-signals⟩ **,** ⟨parameter-values⟩**) ;**

Below, each type of block is discussed in more detail. The *config* block is a special case, it does not attach to signals, and configures the general properties of the acquisition or control program.

4.2.1 *config* block

The *config* block is obligatory. It does not participate in transfer of signal values, but it handles configuration of several properties of the generated module. The first part of the parameters of the *config* block are

Parameter	Type	Description
nic	string	Name of the ethernet port used for *EtherCAT*, for example `"eth3"`.
memsize	integer	Number of bytes memory available for storing data. Default value is `262144` bytes (256 KByte).
template	string	Name of the file to use as module source template. (Only useful for debugging or advanced experiments.)
buffered	boolean	Whether to generate a buffered acquisition module. Default value is `true`.
default_input_buffersize	integer	Default number of samples in a buffer in an *input* block.
default_input_buffercount	integer	Default number of buffers in an *input* block.
default_prewrite	real	Default *prewrite* value for an *input* block.
default_output_buffersize	integer	Default number of samples in a buffer in an *output* block.
default_output_buffercount	integer	Default number of buffers in an *output* block.
default_input_synchronous	bool	Global parameter whether to return a block input data on each call. Default value is `false`.
frequency	real	Number of IO operations per second.
delay_nano	integer	Number of nano-seconds between two IO operations.
delay_micro	integer	Number of micro-seconds between two IO operations.
delay_milli	integer	Number of milli-seconds between two IO operations.

The default buffer size, count, and prewrite values are convenience parameters, they can be overridden in each *input* and *output* block. In the current setup however, there is not much reason to do so, in particular for the buffer sizes. Everything runs at the same `delay_nano` rate, the generated module blocks until it has delivered all its output data, and picks up at most one input data block. Using different buffer sizes with different blocks thus complicates usage of the generated module.

The `default_input_synchronous` parameter is the global default that enforces a matrix with samples is returned for each input block on each call. Each *input* block may override this value. The `synchronous` parameter of the *input* block explains the use and effects in more detail.

The various speed parameters all set the same value, but use different base units. The *delay_nano* is the primitive used by the generated module, all other speed parameters are converted to it. Default speed is `10000000` nano-seconds, which is 100Hz.

In addition, the *config* block has a large number of parameters to tune how the software is built. They exist for advanced C code experiments and debugging purposes, and need not be specified otherwise.

Parameter	Type	Description
matlab_base	string	Base-path of *MatLab*, default value is "/usr/local/MATLAB/<newest>".
matlab_mex	string	Path to the *Mex* executable, default value is "<*matlab_base*>/bin/mex".
soem_base	string	Base path to the installed *SOEM* library.
soem_lib	string	Path to the library file of the *SOEM* library, default value is "<*soem_base*>/lib/libsoem.a".
soem_lflags	string	Link-flags for the *SOEM* library, default value is derived from "<*soem_lib*>".
soem_include	string	Path to the *SOEM* include directory, default value is "<*soem_base*>/soem/include".
compiler	string	Name of the C compiler executable, default value is "<*matlab_mex*>".
acqgen_dir	string	Path to the acqgen runtime source files.
acqgen_sources	string	Files to include in the acqgen runtime. If *acqgen_dir* is specified, the path is prefixed to each file.
cflags	string	C compilation flags, valid value depends on the used compiler. Default value is "-I<*acqgen_source*> -I<*soem_include*>".
lflags	string	C linking flags, valid value depends on the used compiler. Default value is "<*soem_lflags*> -lpthread -lm".

The `compiler` parameter defines the C compiler being used, which by default points at the newest *Mex* program that can be found. This can be tuned by parameter the `matlab_base` or `matlab_mex` parameters. The *SOEM* library being used is decided with the `soem_include` directory (for finding the .h files of *SOEM*), and the `soem_lib` path to the library file. Both use the `soem_base` parameter. By changing its value, you can use your own *SOEM* version. Last but not least, the runtime source code being compiled with the generated C file is given by the `acqgen_dir` and/or `acqgen_files` parameters. Setting them to your own C code files allows advanced C code experiments. Likely, you then also need to modify the template parameter.

4.2.2 *input* block

The *input* block takes one or more input signals, and does not produce any output signal. Instead it groups the signals into matrices, and produces a stream of matrices with data samples of the signals for the *MatLab* application. Each signal gets its own row (same order as specified in the block), each column has data values at a particular point in time for all signals.

Its parameters are

Parameter	Type	Description
index	integer	Cellarray index of the data output of the generated *MatLab* module in buffered mode.
buffersize	integer	Number of samples (number of columns) of one returned matrix. If not specified, the `default_input_buffersize` of the *Configure* block is used.
buffercount	integer	Number of buffers used for the input block. If not specified, the `default_input_buffercount` of the *Configure* block is used.
enable	signal	Signal controlling whether samples are stored.
prewrite	real	Number of samples to store before an up-flank of the `enable` signal is detected. If not specified, the `default_prewrite` of the *Configure* block is used.
synchronous	bool	Enable synchronous *MatLab* operation.

Internally in the module, samples are buffered (in buffered mode) up to `buffersize` samples for each signal, before being given to the application. (Often matrices are completely filled, but that need not always be the case). A filled buffer is copied in its entirety to a matrix which is then passed back to the user for further (batch) processing. Larger `buffersize` thus generally means larger batches of samples. The `buffercount` parameter controls the number of buffers used for one input. In buffered mode, at least one buffer is required (and two buffers are recommended). Even larger values of `buffercount` can be used to deal with larger variations in processing times.

The `enable` signal can be used to control storage of samples. If the attached signal is higher or equal than 0.5, storage is enabled, else it is disabled. Without `enable` signal, storage is always enabled. Although the signal may have any source, the usual approach is to read the output of a *trigger* block, which in turn uses a filtered hardware input signal. The *input* block does not provide control how many samples are taken, it follows the value of the enable signal (that is, configure the *trigger* block for controlling how many samples are taken).

The `prewrite` parameter defines how many samples are (attempted to be) written before the `enable` signal rises to or above 0.5. The parameter value may be a fraction of the buffer size (for example, a value of 0.2 means 20% of the buffer size), or it may be the number of samples to use (less than the size of the buffer). This is useful if the moment of the enable signal is very close to the part of the signal that is of interest. There is however no guarantee on the number of prewritten samples. If the `enable` signal drops below the 0.5 for a few cycles, there may be insufficient time to collect enough samples while prewriting.

The `synchronous` parameter enables synchronous exchange of *MatLab* matrices on each call, for this block. Enabling this parameter causes the *MatLab* function to wait until a matrix with sample data becomes available from this input block. The effect is that one *MatLab* call gives data to output blocks, and then always returns samples from this input block (and any other inputs that have the `synchronous` parameter enabled). However, an input block gives data *after* its buffer is filled. While waiting for that, the output blocks consume the provided data (assuming equal buffer sizes). As a result, to avoid starvation of the output, the next call to the generated module must be made within the sample period. Not enabling the `synchronous` parameter means some *MatLab* calls may not return sample data from input blocks. On the other hand, the *MatLab* call returns earlier in this case. This time can be used to compute the next matrix of output data, or perform post-processing on previously received samples.

4.2.3 *output* block

The *output* block does the reverse of the input block. It accepts matrices with sample values from the *MatLab* application, and outputs the samples to its output signals, one column at a

time. The block has no input signals. There is also no enable signal, data is always sent. The parameters of the output block are

Parameter	Type	Description
index	integer	Cellarray index of the data input of the *MatLab* module in buffered mode.
buffersize	integer	Number of samples (number of columns) of one buffer. If not specified, the `default_output_buffersize` of the *Configure* block is used.
buffercount	integer	Number of buffers used for the output block. If not specified, the `default_output_buffercount` of the *Configure* block is used.

Like in the input block, each output signal has a row (in the same order as specified). The `buffersize` defines the number of available columns in a buffer. It is allowed to provide less columns with data, as long as there is at least one column. Note however that supplying less columns than available wastes buffer space, and generally decreases time that is available for providing the next matrix with data.

The `buffercount` parameter has the same function as with the input block. It defines the number of buffers available for 'writing ahead'. In buffered mode, there must be at least one buffer (two buffers are recommended).

4.2.4 *filter* block

A *filter* block takes one input signal, and produces one output signal. The latter is computed from the weighted sum of the last N samples of the input signal. By using carefully computed weights, many different filters can be constructed.

The filter block has only one parameter, the weights of the samples.

Parameter	Type	Description
weights	list of reals	Multiplication factors applied to the last samples.

The first weight is applied to the most recent sample, the second weight to the second most recent sample, and so on. The number of samples used in the sum is equal to the number of weights of the `weights` parameter.

4.2.5 *trigger* block

The *trigger* block converts its single input signal to an output signal that is either 0 or 1. The normal state of the output is 0. When the input signal varies in the specified way, the trigger fires and the output changes to 1. It keeps that value for a given number of sample periods, and then reverts back to 0.

The parameters of the trigger block are

Parameter	Type	Description
policy	text	Direction of input signal change that is monitored while crossing the given value.
value	real	Input signal value being monitored.
interval	real	Length of the 1 output signal interval, in nano-seconds.
count	integer	Number of sample periods of the 1 output signal interval.

The trigger continuously compares the input signal against the reference value given by the `value` parameter. The `policy` parameter decides how the input signal must reach or cross the reference value. If the policy is '`up`', the trigger fires when the reference signal rises to or above the reference value. The '`down`' policy fires the trigger if the input signal decreases from at or

above, to below the reference signal. Finally, the '`value`' policy fires the trigger in both cases, the input signal has to pass the reference value in either direction.

Both the `interval` and the `count` parameters define the length of the 1 value of the output signal, but in different base units. Only one of them should be stated.

4.2.6 *function* block

A function block generates a known signal, and sends it to the output. Its primary use is to generate such a signal at an hardware output, without having to create and send your own samples.

Currently, three signal shapes are available, a sine, a block signal, and a triangle signal. Each of them is configurable to provide a wide variety of signals.

Sine function

The sine function swings back and forth around a center base signal, at a given amplitude, with a given period. Its parameters are

Parameter	Type	Description
shape	string	Fixed value `sine`
frequency	real	Length of the period, in Hz.
base	real	Value of the center value.
amplitude	real	Maximal deviation from the base value.
max_samples	integer	Maximum number of samples that may be computed.

The output of the sine function is pre-computed to improve performance, sampling the sine function on the global hardware sample period. To improve accuracy, the sine function may be sampled for several periods. The `max_samples` parameter control the upper limit on the number of samples that can be used to represent the sine function.

Block function

The block function periodically instantaneously switches between the high level and the low level output. A common use of this function is to produce a pulse signal.

The parameters of the block function are

Parameter	Type	Description
shape	string	Fixed value `block`
frequency	real	Length of the period, in Hz.
duty_cycle	real	Fraction of the period to emit the high level value.
low	real	Value to produce in low level output.
high	real	Value to produce in high level output.

Triangle function

The triangle function generates a signal that alternates between linearly increasing to the high level, and linearly decreasing to the low level. A common use of this function is to produce a saw tooth signal.

The parameters of the triangle function are

Parameter	Type	Description
shape	string	Fixed value `triangle`
frequency	real	Length of the period, in Hz.
peak_at	real	Fraction of the period where the signal is at its highest value.
low	real	Lowest possible value of the function.
high	real	Highest possible value of the function.

At the start and the end of a period, the signal is at the lowest value. At the `peak_at` fraction of the period, the signal is at its highest value.

5 Using the generated module

If you skipped Section 4 so far, you may want to go back and read the general description part first to learn about modes, signals, and the general idea behind blocks, in particular input and output blocks.

The generated module is used by simply calling it with matrices containing data values to send out. After the call the module returns samples that it received. The first call will take additional time to setup *EtherCAT*, but otherwise, the data is processed normally.

The signature of the function is in normal operation is

```
[errors, recvdata] = Function(senddata)
```

In unbuffered mode, `senddata` is a $N \times 1$ column matrix carrying one sample for each of the N output signals. The returned `error` is a string containing an error message if an error was found. The `recvdata` column matrix of size $M \times 1$, containing a sample value for all M input signals.

In buffered mode, the function signature changes to one cell matrix in and two cell matrices out with error messages and samples.

The `senddata` cell matrix contain matrices with more samples for an output block. For example, to send output data to an output defined as

```
aout[0], aout[1] = output(buffersize=5, buffercount=2, index=3);
```

a 2×5 matrix (5 samples for each output row) should be made, and assigned to cell index number 3, like

```
senddata{3} = [1, 2; 3, 4; 5, 6; 7, 8; 9, 0];
```

Less samples than the specified `buffersize` is allowed but not recommended.

During or after the `Function` call, the analogue outputs first get the first column values, then the second column values, etc. Note that the `buffercount` of an input or output block has no effect on the matrix format and size.

The receiving side works in the same way, except the data is returned in the `recvdata` cell array. Each row represents an input point, in the order listed with the `input` block. Each column contains data samples of all sensors at a single point in time. The `recvdata` cell matrix uses the `index` numbers provided in the `input` blocks.

The `errors` return value is a cell array with error messages that occurred before or during the function call. The index used in the cell array indicates which input or output block had the error, if the index is larger than any index block number, a block could not be decided. (That is, it means a general error has occurred, rather than in a specific input or output stream.)

A call to the generated module blocks until it has delivered all `senddata`. It also waits for a block of received samples from each *input* block that has its `synchronous` parameter set. Finally,

if available, it picks up one block of received samples from the *input* blocks that do not have the `synchronous` parameter set.

The important notion here is that output data and input data must be kept in balance. Doing too much at one side will starve or overrun the other side. This becomes more complicated if various buffer sizes are used.

5.1 Stopping data acquisition

In buffered mode, a separate thread is running, performing the actual input and output. This thread continues to run until you close *MatLab*, or you tell it to stop. The latter is done by calling the `Function` without any arguments. The module closes down, and returns nothing from the call.

6 Installation

The ACQGEN program is designed to run at a *Linux* system, such as *Ubuntu LTS 16.04*, but others may also work.

Software required for installing ACQGEN:

- The *SOEM EtherCAT* master software, source available from `https://github.com/OpenEtherCATsociety/SOEM` .
- CMake (available from the Ubuntu *cmake* package).
- gcc (available from the Ubuntu *gcc* package, includes pthreads library).
- make (available from the Ubuntu *make* package).
- Python3 (used Python 3.5,1, available from the *python3* package).
- The *MatLab* software (used Matlab R2015a but newer should work too), available locally from a ICT helpdesk.

The ACQGEN software depends on both *MatLab* and *SOEM EtherCAT*, so these need to be installed first. Matlab provides its own installation instructions, and installs by default in a directory in `/usr/local/MATLAB`. If you install it elsewhere, adjust the *MatLab* paths as needed. The *SOEM EtherCAT* software needs some special handling to make it useful for dynamic linking, this is explained in the next section.

6.1 Installing *SOEM EtherCAT* master software

The *SOEM* software comes as C source code, and must be compiled using the `-fPIC` flag. Steps to do this:

1. Download source from `https://github.com/OpenEtherCATsociety/SOEM`, as source download or by cloning the `git` repository. Here, revision `fb975cbc70bde723a178936ddb8c3b1c8c192c39` was used (committed at March 31, 2017), but newer likely works too.
2. Unpack the software to a source tree from the archive, if required. Sub-directory `SOEM` is assumed to hold all files.

3. The software uses *CMake* to generate its out-of-source build files. Make a build directory for it, by entering `mkdir build` from a terminal. Also, enter that directory using `cd build` .

4. To generate the build files for SOEM, use the following `CMake` command:

```
cmake -DCMAKE_INSTALL_PREFIX=/path/to/installed/soem \
      -DCMAKE_C_COMPILER:FILEPATH=gcc \
      -DCMAKE_C_FLAGS:STRING=-fPIC \
      ../SOEM
```

   The command takes three `-D` options, where the `/path/to/installed/soem` is the destination directory for the SOEM library (replace it with your own preferred destination). The fourth line `../SOEM` points to the SOEM source directory that was just downloaded and unpacked (path changes if you make the `build` directory elsewhere).

   The backslashes at the end of the lines are not part of the command, they denote that the command continues on the next line. The command can also be typed at a single line, without the backslashes.

5. If all is well, the above command finds all the required software for the SOEM software, and it generates a `Makefile` in the local (`build`) directory. Now build the software by typing `make`.

6. Finally, the created software should be installed by typing `make install`, which results in a populated `/path/to/installed/soem` directory.

7. To clean up, the `build` directory can be deleted:

```
cd ..   # Leave the 'build' directory.
rm -r build
```

6.2 Installing acqgen software

The ACQGEN software can be installed using the supplied `install.py` Python3 script. It comes with online help (type `./install.py --help`). The normal command-line to install is a command like:

```
./install.py --install-prefix=/path/to/installed/acqgen \
             --soem=base=/path/to/installed/soem
```

where `/path/to/installed/acqgen` is the destination of the ACQGEN software. Replace it with your own preferred location, by default it uses */usr/local*. The `/path/to/installed/soem` is the path where *SOEM* was installed previously (that is, the `CMAKE_INSTALL_PREFIX` value in the previous section).

The installation doesn't say anything. Afterwards, there should be a `bin/acqgen` Python script installed in the destination path of ACQGEN, and a Python code library in `share/acqgen`. The installer also allows to specify the path of each of these, in case your system uses a different file system layout.