ToolDAQ

Hi, I believe manuals should be short, to the point and with lots of examples. The idea is to get you up and running as quickly as possible and then expand into the subtleties later.

If you have questions I can be reached here b.richards@qmul.ac.uk.

So off we go.

Basic quick start
Sec 1. Basic Concept
Sec 2. Installation
Sec 3. Tools
Sec 4. Variables and the DataModel
Sec 5. Simple Example

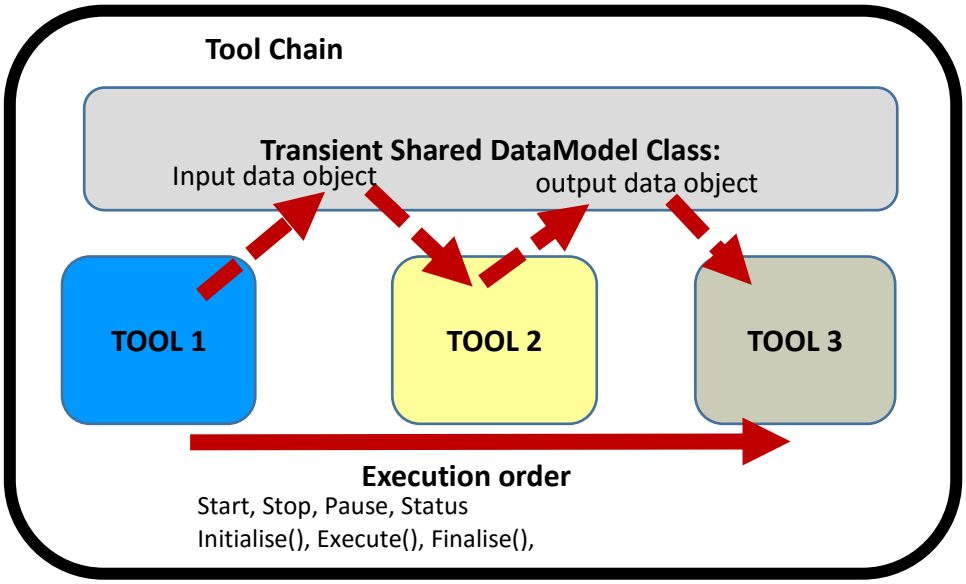
Detail
Sec 6. ToolChain and main
Sec 7. Logging
Sec 8. Root
Sec 9. Network
Sec 10. Parallelization

Sec 1. Basic Concept

ToolDAQ is a pure C++ framework for building DAQ systems built on the principle of the sequential running of modular Tools and transient data storage. It has built in network communications, dynamic service discovery, logging, execution and variable handling.

So if you followed that good, if not don't worry. ToolDAQ was built out of the frustration that things could be done simpler, easier, faster and better. With that in mind there are only three concepts to take away from this section and they are:

1. **What is a ToolChain**
2. **What is a Tool**
3. **What is the DataModel**



1. **What is a ToolChain**

   Put simply a **ToolChain** is just a class that holds **Tools**. It is also responsible for running those tools one by one from start to end sequentially. First it **Initialises** each tool, then it **Executes** each tool (as many times as needed), before finally **Finalising** each tool.

2. **What is a Tool**

   A **Tool** is just a class that does something. It can be anything from reading data from hardware, to processing that data, to writing that data to disk. The idea is to make your DAQ program as modular as possible by splitting the operations into **Tools** which you then place in a **ToolChain**. Tools are just classes that contain your code to do anything you like. They must however contain an **Initialise**, **Execute** and **Finalise** function.

3. **What is the DataModel**

   The **DataModel** is probably the most difficult concept in this section but it's actually quite simple. **Tools** cannot directly interact with each other and don't share memory and variables. So we have a transient data class called the **DataModel.** Think of it as a data storage vault to keep anything used by multiple **Tools**. An example Tool A reads data from some hardware and Tool B saves it to disk. To do this Tool A writes the data to the **DataModel** and Tool B reads it from the data model and writes it to disk.

And that's all you need to know to get started. To create your own DAQ software all you need to do is create your own **Tools** to interact with hardware, network, disk, databases or whatever you like and fill the **DataModel** with the variables you need.

Sec 2. Installation

The ToolDAQ repository has a few branches:

- The **master** branch holds the core framework code
- The **Application** branch is what you should download to build your own application
- The **Example** branch contains an example application.

The following guide is for the **Application** branch (but should work on all branches).

First get the code by running

```
#git clone https://github.com/ToolDAQ/ToolDAQFramework.git -b Application
```

The prerequisites for building ToolDAQ are bellow so please make sure they are installed with the package manager of your choice e.g.

```
#yum install make gcc-c++ gcc binutils libX11-devel libXpm-devel libXft-devel
libXext-devel git
```

The **Application** branch contains a template to build your own application from scratch. As such it only contains the code specific your application all the core ToolDAQ code must now be installed by running:

```
#./GetToolDAQ.sh
```

This will install all the core code and its prerequisites (zmq and boost) to a ToolDAQ folder. Once installed it will make both the core code and your application code.

Next you will need to source the locally installed prerequisites by running the following form the application install

```
#source Setup.sh
```

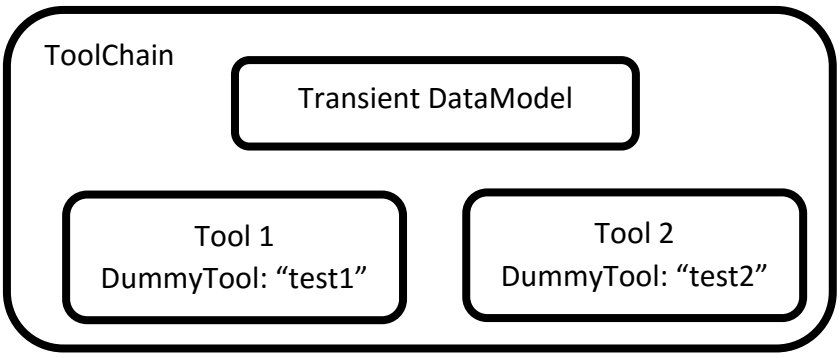
Now try and run the code with:

```
#./main
```

If all worked well you should see some output like that shown below.

(Detail: Feel free to skip:

So the default **ToolChain** contains two instances of the same tool called DummyTool, one called "test1" and one called "test2".



The Application when executed just sequentially adds the tools to the toolchain and then tells the Tools to Initialise, then Execute and then Finalise. The DummyTool itself just prints to screen the words "test1" on initialise and "test2" on Execute as can be seen in the output below.

)

```
[1]: UUID = 7e511e61-649a-4bb1-bf7c-03db68e66387
************************************************************
**** Tool chain created ****
************************************************************

[1]: Adding Tool="test1" tool chain
[1]: Tool="test1" added successfully

[1]: Adding Tool="test2" tool chain
[1]: Tool="test2" added successfully

[1]: ************************************************************
**** Initialising tools in toolchain ****
************************************************************

[2]: Initialising test1
[1]: test 1
[2]: test1 initialised successfully

[2]: Initialising test2
[1]: test 1
[2]: test2 initialised successfully

[1]:
**** Tool chain initialised ****
************************************************************

[1]: ************************************************************
**** Executing toolchain 1 times ****
************************************************************

[3]: ************************************************************
**** Executing tools in toolchain ****
************************************************************

[4]: Executing test1
[2]: test 2
[4]: test1 executed  successfully

[4]: Executing test2
[2]: test 2
[4]: test2 executed  successfully

[3]: **** Tool chain executed ****
************************************************************

[1]: ************************************************************
**** Executed toolchain 1 times ****
************************************************************

[1]: ************************************************************
**** Finalising tools in toolchain ****
************************************************************

[2]: Finalising test1
[2]: test1 Finalised successfully

[2]: Finalising test2
[2]: test2 Finalised successfully

[1]: **** Toolchain Finalised ****
************************************************************
```

Sec3. Tools

Great so now you have a working version of a ToolDAQ application now you want to make it do stuff right. Well then you will need to make your own **Tool.**

Steps for making and adding your own tool:

1. **Create a Tool**
2. **Add that Tool to the ToolChain**
3. **Compile and Run the application**

1. Create a Tool

So this couldn't be simpler all you need do is go to the UserTools directory and run the script newTool.sh followed by the name of our new tool eg.

```
# cd UserTools
# ./MakeTool MyFirstTool
```

(Detail: Feel free to skip

> So to make a Tools you just need to create a class that inherits from an abstract base class called Tool. This forces implementation of the Initialise Execute and Finalise methods required. The script also adds the tool to a factory class for ease of integration dynamically at run time but this is not strictly necessary if u wanted to create your own main. But I suggest you read the ToolChain section before you head down this route

)

2**. Add that Tool to the ToolChain.**

This somewhat relies on you knowing how ToolChains work which is a latter section but to get you up and running for now, just edit the ASCI config file (configfiles/ToolsConfig) That lists the Tools to add to the ToolChain.

Each line is an instruction to add a Tool to the ToolChain and contains three words.

ToolType ToolName ConfigFile

The Tool type is what we called our Tool when we used the newTool.sh script. The Tool name can be anything we like (note u can have multiple instances of the same tool if you like) and the config file is the path to an input ASCI file containing any input variables you would like to use in that Tool.

So for our Tool we need to add

MyFirstTool MyFirstTool1 Null

(you can see a description of using a non Null config file in the variables section)

3. **Compile and Run the application**

This is pretty easy navigate to the application directory and run make clean, make and then run the executable.

E.g

```
# make clean
# make
#./main
```

Well done. If all gone well you will now see in the messages that print put your tool is running in the tool chain and is initialised executed and finalised along with the dummy tools.

(if not check your ToolsConfig file as the ToolType is case sensitive and must be exactly the same as your created tool)

So that's fine but now you want to make your Tool Do something useful.

Time for some examples.

Example 1: Hello world

All coding examples start with a hello world.

Open up your Tools implementation file in your editor of choice eg.

$ emacs UserTools/MyFirstTool.cpp

From here you should see a constructor and 3 functions, Initialise, Execute and Finalise. You will possibly be unsurprised to hear that these are the functions run when the main executable tells you it is initialising executing and finalising your Tool.

(**Details: can skip if you're in a hurry**

> The idea is that Initialisation is use for your Tools setup (eg. Opening sockets/databases, loading variables from config files, initialising hardware settings etc). Execute is where you run whatever the tool is made to do (note: execute function was designed to do a small amount multiple times rather than long computation or blocking e.g the read of one event or one trigger from hardware). The Finalise method is mean for clean up and disconnection of your tool and resources.

)

So we just want to print hello world so we can add a print out line in the execute function ion to do just that.
Eg

```
std::cout<<"hello world"<<std::endl;
```

That should do it. Save the file rerun the make clean, make and then run the main and you should see the print out when your tool Executes.

Sec 4. Variables and the DataModel

Ok so you now have the ability to add Tools and hopefully understand where to put your code to do stuff. Next we need to talk about variables and the data model so you can start to build useful applications.

When storing and using variables in a Tool you need to consider scope (when and what needs to access it). The scope will determine the best place to put your variable. You're free to write your tool however you like in whatever style you like so part of this may fall into best practice but scope is always worth considering. In the framework of ToolDAQ it's important as Tools do not share memory and can only interact through the DataModel. So here is my suggestion.

There are three basic scopes:

1. Function specific scope
2. Tool specific scope
3. ToolChain scope

1. If you only want a variable to be used inside the execute function, go ahead and make it in the execute fiction as normal.

2. If you want to use a variable in many functions within the Tool class, e.g. initialise it in the initialise function and use it in the execute function. Then put its definition in the Tools header like a normal class

3. If you want multiple tools to have access to the variable. (Eg like data taken from a card in one tool and written to disk in another, or configuration variables passed between Tools) then put its definition in the DataModel header (DataModel/DataModel.h) and access it inside you tools by m_data->VariableName.

Config files and initialisation variables

When Creating Tools sometimes you will want to pass Tool specific configuration settings to the Tool.

This can be done manually by the user as you would in any other code implementation, or using ToolDAQ's own facility for this. You will recall in the Tool section that when adding a Tool to the ToolsConfig file the third word is the path to a configuration file to pass to the Tool e.g.

MyFirstTool MyFirstTool1 configfiles/MyFirstToolConfig

MyFirstToolConfig must have a specific format. It's an ASCI file where by each line defines a variable. The first word of each line is the variable name and the second word is the value e.g.

```
#myfirsttool config file
Var1 hello
Var2 6 #comments
#comments
Var3 38.53
```

Note variables can be any type just written in plain text (bools must be 1 or 0) and comments can be added after a '#' symbol.

This file will automatically be read by the Tool on initialisation using a universal storage class known as a **Store**.

These variables can then be accessed within your tool in the following way.

```
std::string a;
Int b;
float c;

m_variables.Get("Var1", a);
m_variables.Get("Var2", b);
m_variables.Get("Var3", c);
```

The get function is templated where the first argument is the name of the variable in the config file and the second is the variable to assign it to.

(Detail: Store Class

> The universal store class whilst useful is not an efficient way to store data so should not be used for large event based data. It does contain a Set function as well as a Print which can be useful for listing all the variables inside the Store.
>
> An instance of the Store class actually exists in the DataModel as well which can be used to pass variables (again inefficiently) between tools. It can be useful for counters and things of this nature.

)

Sec 5. Simple Example

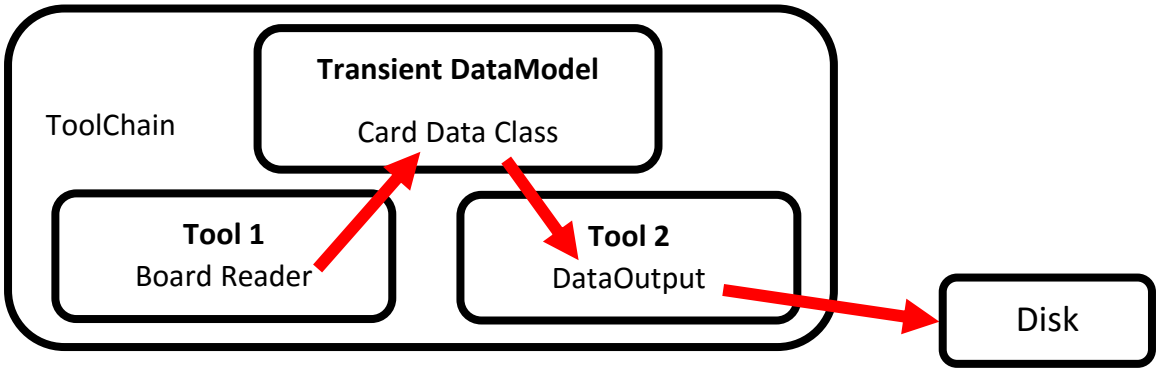
Ok so let's see how a full example of a DAQ implementation.

(You find the code for this example by downloading and installing the Example branch of ToolDAQ following the steps in Sec 2). Make sure you also have all the root prerequisits

```
gcc-c++ gcc binutils libX11-devel libXpm-devel libXft-devel libXext-devel
```

For this simple example we will make a DAQ application that simply simulates reading data from a front end board and then saves that to disk. To do this we will create two Tools and a data class. One Tool will simulate the board output and write it to the data class in the

DataModel and one Tool to write out the data to disk. The ToolDAQ diagram for this would like this:



Detailed Description

1) **Board Reader Tool:** This tool simulates a simple ADC/TDC output namely a single Time and Energy value for each channel in the card.
    1. The Initialise method just stores three values form a config file to local variables (random seed, Card ID and no. of channels)
    2. The Execute method creates an instance of the CardData class and fills it with a random number energy and random time stamp etc. This is them pushed back into the data model.

2) **CardData Class:** This class stores the output of a Card so contains the Card Id and a vector of energy and time stamps (one per channel). It also contains a serialisation function for use with boost serialisation (not necessary).

3) **DataOutput Tool:** In this example I have created two data output tools (RootWriter and BinaryWriter). The names should make their functions obvious but for clarity the RootWiter creates a Root file of the output and the BinaryWriter creates a simple binary file output using boost serialise (note: the boost file in this simple form will be less space efficient and root automatically reduces the size by compressing similar entries).

    The operation of both tools are very similar.

    1. The Initialise method reads in a config file variable to set the output file path and name. It then creates and opens a file in this location. In the case of the root version it sets up the necessary tree and branches (note: it points to a local card data instance as the memory address must be static for filling. Also the DataModel class contains functions for holding trees so I have utilised those as an example. These could therefore be used and filled by multiple tools and written out by a single tool at the end of the chain).

    2. The Execute method simple fills the output stream be it a direct ofstream for the binary writer or in the root writer's case copying it to

the local card data and then filling a ttree. Once done card data class in the Data model is cleared for the next loop

3. The Finalise method writes the output to disk and clears the necessary data objects.

And that's it you have a fully functioning example. Let's briefly see how to customise and operate the program.

- The default operation is to have a single BoardReader class that has 4 channels. This single board is read from 100 times and saved to a root file "out.root".
- To change the number of channels per board simply edit the configfiles/BoardReaderConfig file and change the channels variable. You can also adjust the seed and card id here.
- To switch the output root file location you can edit the configfiles/RootWriterConfig and adjust the outpat variable (for the BinaryWritter edit the configfiles/BianryWriterConfig instead)
- To switch to the binary writer output instead you need to change tools inside the configfiles/ToolsConfig file. Simply remove the '#' from the BinaryWriter and add one to the RootWriter.
- If you want to add more boards you can once again edit the configfiles/ToolsConfig and add more BoardReaders eg.

```
BoardReader BoardReader1 configfiles/BoardReaderConfig1
BoardReader BoardReader2 configfiles/BoardReaderConfig2
BoardReader BoardReader3 configfiles/BoardReaderConfig3
BoardReader BoardReader4 configfiles/BoardReaderConfig4
```

In the above I have reference a different config file for each BoardReader, this is not strictly necessary but as the CardID is read in from there it's useful to label each with their own ID so you can see that they are all creating their own output

- Finally you can change the operation of the ToolChain by editing the configfiles/ToolChainConfig file
    1) Change the "verbose" from 0 to 9 to change the level of printouts
    2) You can change the number of reads per execution of the application by changing the "Inline" variable.
    3) You can switch to interactive by setting "Interactive" to 1 and "Inline" to 0. Then follow the onscreen instructions.
    4) You can try remote operation by setting "Remote" to 1 and "Inline" and "Interactive" to 0. To do this start you application in the background with "./main &" and then launch the "./RemoteControl" application and follow the on screen instructions.

Note: all of the above customisations are all runtime changes and don't need any recompilation no matter how many cards and channels you add or how you choose to control the application.

Sec 6. ToolChain and the main

Ok now we come to the ToolChain, on the surface it's simple but there is a lot of things under the hood. We will start with what it does and how to use it to create your own main. Then we will talk about the configuration and then some deeper detail for those interested.

So the ToolChain is just a container for Tools. Rather than manually initialising executing and finalising tools we put them into a single container and then tell it to run the initialise execute and finalise methods on all the tools inside.

The best way to explain it is with an example. You may have noticed that in the src folder there is a single solitary file (main.cpp). This is the main of our application and its intentionally short and simple (in fact most of the lines are commented out).

If we want to produce the simplest possible main with ToolDAQ it would look like this.

```
#include "ToolChain.h"
#include "DummyTool.h"

int main(int argc, char* argv[]){

 ToolChain tools();

 DummyTool dummytool;

 tools.Add("DummyTool",&dummytool,"configfiles/DummyToolConfig");

 tools.Initialise();
 tools.Execute();
 tools.Finalise();


  return 0;

}
```

It Should be self-explanatory but in case it isn't, first we create a ToolChain called tools. Then we create a DummyTool called dummytool. We then add this tool to the ToolChain. Once added we Initialise, Execute and Finalise the ToolChain which calls the Initialise Execute and Finalise functions of each Tool.

Very simple right?...

Ok now if we wanted to take hypothetically 100 events we could call "tools.Execute();" 100 times before "tools.Finalise();" but we can also do that by passing the variable 100 to the Execute function. E.g.

```
tools.Initialise();
tools.Execute(100);
tools.Finalise();
```

This is known as **Inline** operation of ToolDAQ. There are two other running modes **Interactive** and **Remote.**

In **Interactive** execution will start a prompt where you can Initialise, Execute and Finalise the Tools manually (good for debugging). You can also call Start Stop Pause Unpause etc for continuous operation. E.g:

Start: calls Initialise then Execute in a Loop:
Stop: calls Finalise
Pause and Unpause: halt and continue the Execute loop

```
#include "ToolChain.h"
#include "DummyTool.h"

int main(int argc, char* argv[]){

 ToolChain tools();

 DummyTool dummytool;

 tools.Add("DummyTool",&dummytool,"configfiles/DummyToolConfig");

 tools.Interactive()


  return 0;

}
```

Remote operation works the same way, The same Initialise….Start….etc. functions as before can be sent from a remote machine rather than the current one. To use remote mode we just exchange "tools.Interactive()" for "tools.Remote()" in the above code.

Hopefully that's quite simple and we will return to how to operate the code in remote mode in a later section.

However whilst this is a good instructional example it has a flaw for operation, namely you have to recompile every time you want to change Tools and Running mode etc. What if we want to do it on the fly or remotely?

Well, you have already seen that Tools can be added from a config file so why not make the operation modes and everything word dynamically via a config file? Well that's what I did. So your main shrinks to just this.

```
#include <string>
#include "ToolChain.h"
#include "DummyTool.h"

int main(int argc, char* argv[]){

  std::string conffile;
  if (argc==1)conffile="configfiles/ToolChainConfig";
  ToolChain tools(conffile);

  return 0;

}
```

Which you will notice is exactly what's in src/main.cpp. But now you need a config file to run it so let's look at the format of that.

The ToolChainConfig file

So let's look at a ToolChainConfig file:

```
#ToolChain dynamic setup file

##### Runtime Parameters #####
verbose 9
error_level 0 # 0= do not exit, 1= exit on unhandeled errors only, 2= exit on unhandeled errors and handeled errors
attempt_recover 1
remote_port 24004

###### Logging #####
log_mode Interactive # Interactive=cout , Remote= remote logging system "serservice_name Remote_Logging" , Local = local file log;
log_local_path ./log
log_service LogStore
log_port 24010

###### Service discovery #####
service_discovery_address 239.192.1.1
service_discovery_port 5000
service_name main_service2
service_publish_sec 5
service_kick_sec 60

##### Tools To Add #####
Tools_File configfiles/ToolsConfig

##### Run Type #####
Inline 1
Interactive 0
Remote 0
```

Each variable of the file is in a section so let's look at them one at a time

```
##### Tools To Add #####
Tools_File configfiles/ToolsConfig
```

This is the path the config file which lists the Tools to add to our ToolChain the form of this file can be seen in the Tools section.

```
##### Run Type #####
Inline 1
Interactive 0
Remote 0
```

The Run Type should be self-explanatory if you have read the previous part about creating a main, but it allows you to choose which of the three run types you want using a bool 1 and 0. Node for Inline operation you can use an integer to define how many executions you want. Eg for 100 events you could put "Inline 100".

To Be Continued…..

Sec 7. Logging

As always you're free to write your own logs for your Tools separately but there is a centralised log available for all output.

Technically if you use std::cout your already using it. As ToolDAQ hijacks the standard out stream and redirects it through its own buffer class. This means that everything printed to screen can be redirected to a file instead or over the network to a computer tasked with storing the logs for all systems running ToolDAQ. Either if these three options are available by changing the configfiles/TooChainConfig config file.

There are a lot of variables in here and they are discussed in detail in the ToolChain section but for now your

To Be Continued…..