# Hierarchical Data Acquisition (HDAQ) and Task Control (HTC)

## Definitions

### Hierarchical Task Control

1. A *Study* consists of multiple *Root* *Tasks* that may be carried out independently from each other, in an arbitrary order and at arbitrary time points, in a user-defined way.
2. A Root Task such as acquiring a stack of images may contain an arbitrary number of Subtasks grouped hierarchically at arbitrary nesting levels that form a *Task Tree* structure. The Root Task is the topmost Task node within the Task Tree.
3. Task Trees function independent of each other.
4. Parent Task nodes can have multiple child Task nodes or Subtasks. A child Task node may have only one parent Task node.
5. Adding a Child Task node to a Parent Task node makes the Child Task node share the same Root Task with the Parent node.
6. Once started, a parent Task is complete when all child Tasks are complete.
7. Tasks are carried out by *Task Controllers* (TCs) which model the actions of a software or hardware module during the execution of the Root Task.
8. TC names within a Task Tree are unique. TC names between Task Trees may be the same.
9. Only one TC is allowed per module/device and the TC may participate in only one Task Tree at a time during the execution of the Root Task. This promotes the reuse of TCs within different Root Tasks and decreases their total number required to carry out the Study.
10. The execution of a Task by a Task Controller is understood as an iteration of the Task.
11. Depending on the number of iterations, there are 3 kinds of Task Controllers: 1) Zero iterations – Null Task Controller (NTC), 2) Finite iterations – Finite Task Controller (FTC) and 3) Continuous iterations – Continuous Task Controller (CTC).
12. An NTC functions similarly to a FTC with a single iteration, except that it does not execute its iteration function.
13. A FTC once started executes its iterations a finite number of times after which the task completes or the iterations are manually stopped.
14. A CTC has an arbitrary number of iterations. For a CTC to complete it must be stopped manually or a certain stopping condition must be met after an arbitrary number of iterations.
15. The execution of a Task Tree can be started and stopped only from its Root Task Controller which relays these commands to its child TCs. The child TCs in turn relay commands further down to their child TCs, to the deepest nesting level of the Task Tree.
16. The execution state of child TCs is relayed back to their parent TC and thus affects the execution state of their parent.

### Task Controller Iterators

1. Each TC, including the Root Task TC, may have several TC *Iterators* (TCIs) that form a TC *Iterator Set* (TCIS) of the form {TCI1, TCI2,.., TCIN}. The elements in a TCIS are dimensions in an N-dimensional iterator space and the convention taken here is that elements in this set starting from left to right increase the dimensionality of this space.
2. A single TCI from a TCIS may have a name and keeps track of the total number of iterations, the current iteration index, and a TCI parameter set (TCIPS). This is summarized as TCI = ("iterator name"= name (optional if there are more than one TCIPs), "current iteration index/total number of iterations" = i/N, "parameter set" = {TCIP1, TCIP2, .., TCIPN}).
3. TCI names from a TCIS are unique within the set.
4. The total number of TC iterations is the product of the individual TCI total number of iterations.
5. A single TCIP from a TCIPS contains the name of the parameter that is iterated (such as frame number or z-stage position), the current “value” of the parameter which is a pointer (numeric, string, images, etc.) and the unit of measure if any. This can be summarized as TCIP = ("name" = name, "data type" = dtype, "value" = val, "unit" = unit).
6. The name of a TCI may be the same as the name of a TCIP.
7. Just as there are 3 different types of TCs: NTCs, FTCs and CTCs, there are 3 different kinds of TCIS elements: 1) null number of iterations – null task controller iterator (NTCI), 2) finite number of iterations – finite task controller iterator (FTCI), 3) indefinite number of iterations – continuous task controller iterator (CTCI).
8. A FTC finishes when all iteration indices from the TCIS are equal to the total number of required iterations for each index.

### Virtual Channels and Data Exchange

1. Data exchange between the TCs occurs through *Virtual Channels* (VCs).
2. VCs of a TC have unique names and each TC manages its own list of VCs.
3. Depending on the data flow direction, there are two types of VCs: *Sink* VCs and *Source* VCs.
4. Data exchange between TCs is done by connecting Sink and Source VCs.
5. A Source VC can be connected to multiple Sinks, however, a Sink may have only one Source connected to it.
6. A Sink and Source VC exchange data through *Data Packets* (DPs) by opening and closing a *Data Exchange Session* (DES).
7. Depending on whether a TC iteration needs data before starting or in order to complete, a DES may be opened and closed only once before or during an iteration and the iteration cannot start or complete without opening and closing an open DES. A DES is complete if the required amount of data between the Source and the Sink has been exchanged.
8. Data contained in a DP is shared by multiple Sink VCs and the DP is disposed if there are no more Sink VCs using it. This eliminates the need to make several copies of the data.

### Hardware Triggers and Data Exchange Dependencies

1. A *Hardware-Triggering* or *Triggered* (HWT) *Task Controller* (HWT-TC) generates or requires one or more digital pulses that are used to time precisely the start of a TC iteration.
2. A HWT-TC that generates a trigger is a HWT-TC Master whereas if it requires a trigger, it is a HWT-TC Slave.
3. A HWT-TC can be either a HWT-TC Master or a HWT-TC Slave and not both.
4. A HWT-TC Master can be connected to one or more HWT-TC Slaves.
5. A HWT-TC Master generates a HWT only when all its HWT-TC Slaves are ready to receive such a trigger.
6. A TC iteration that generates or requires a HWT is complete once the HWT is received or generated. Each iteration of a HWT TC must generate (HWT-TC Master) or require a HWT (HWT-TC Slave).
7. A connection between a HWT-TC Master and HWT-TC Slave is valid if given the structure of a Task Tree the total number of times the HWT-TC Master generates a trigger equals the total number of times the HWT-TC Slave will receive the trigger.
8. Within a Task Tree, the following type of HWT-TC connections can be established: 1) Master Parent and Slave Child (MPSC, Figure 1), 2) Master and Slave Siblings (MSS, Figure 2).

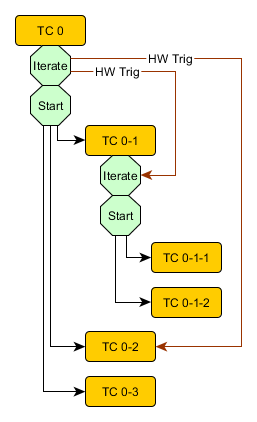


Figure 1. Master Parent Slave Child (MPSC) HW triggering in the case of an Iterate First type of Task Controller (TC). Starting TC 0 will call its iterate function pointer which will prepare it to send a HW trigger and switch to a TASK\_STATE\_RUNNING\_WAITING\_HWTRIG\_SLAVES after which it will send a TASK\_EVENT\_START to all its child TCs. Note that in this case, the iteration is not considered complete until the HW trigger can be generated. Child TCs 0-1 and 0-2 are Slave Hardware Triggered (HWT) TCs and for executing their iterations they require a HW trigger from TC 0. Therefore when they enter a TASK\_STATE\_RUNNING\_WAITING\_ITERATION, they will wait for the HW trigger to complete their iteration. TC 0-1 being an Iterate First type of TC as well, it will call its iteration function pointer first after which it will also send a TASK\_EVENT\_START to its child TCs 0-1-1 and 0-1-2. Note that also in this case, there is a choice whether to complete the iterate function call before or sometime after sending TASK\_EVENT\_START. To have a consistent number of triggers generated and received, for this example, establishing a HW trigger between a parent (TC 0) and child (TC 0-1 and TC 0-2) forces the number of iterations of TC 0-1 and 0-2 to be 1.

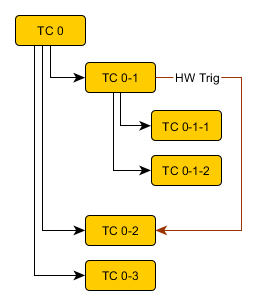


Figure 2. Master Child Slave Child (MSS) Task Controller HW Trigger.

1. In a MPSC HWT, one or more Slave Child TCs receive a HW trigger from the same parent TC.
2. In a MSS HWT, the Sibling TCs sending each other triggers share the same parent TC.
3. A Master or Slave HWT TC may have zero or more child TCs.
4. A Master HWT TC in a MPSC-type connection can be only of an Iterate First TC type, i.e. the provided iteration function pointer is called before sending a TASK\_EVENT\_START to all the child TCs of the parent Master HWT TC.

### Task Controller events and actions

1. A Task Controller is an extended state machine that executes user-provided callback functions when it receives various events depending on its current state. The following is a list of possible states that a TC may be in, events it can send and receive and function callbacks it can execute:

States

* **Unconfigured**
* **Configured**
* **Initial**
* **Idle**
* **Running**
* **Iteration Needs Data**
* **Completing Iteration**
* **Waiting For Armed HW Triggered Slaves**
* **Stopping**
* **Done**
* **Error**

Events

* **Configure**
* **Start**
* **Iterate**
* **Iteration Done**
* **Iteration Timeout**
* **Iterate Once**
* **Reset**
* **Stop**
* **Stop All**
* **Stop Continuous Task Controller**
* **Abort**
* **Subtask State Changed**
* **HW Triggered Slave Armed**
* **Data Received**
* **Custom Module Event**
* **Subtask Added To Parent Task**
* **Subtask Removed From Parent Task**

User provided Function Pointer Callbacks (FPCB)

* **Configure**
* **Iterate**
* **Abort Iteration**
* **Start**
* **Reset**
* **Done**
* **Stopped**
* **Dim/Undim User Interface**
* **Set UITC Mode**
* **Error**
* **Data Received**
* **Module Event**

1. The user may control the execution of a Task Tree only by interacting with its Root Task Controller which is also called as an User Interface Task Controller (UITC).
2. While a Task Controller can respond to any of the listed events, User Interface Task Controllers (UITCs) use only a subset of the possible events that can be generated only by pressing button controls. The events a UITC can respond to are:

* **Start**
* **Configure**
* **Stop**
* **Stop All**
* **Abort**

#### Stop

When a Task Controller receives a **Stop** event, before it stops and enters in an Idle or Done state:

* Completes its current iteration.
* Waits until all its Finite Iteration Child Task Controllers complete *all their iterations*.
* Waits until all its Continuous Iteration Child Task Controllers complete *their current iteration*.

In this case:

* Finite Iteration Child Task Controllers will be in a Done state having performed their required number of iterations.
* Continuous Iteration Child Task Controllers will be in an Idle state.
* The parent Task Controller that received the Stop event will be in an Idle state if it did not complete the required number of iterations or in a Done state otherwise.

#### Stop All

When a Task Controller receives a **Stop All** event, before it stops and enters in an Idle or Done state:

* Forwards the Stop All event to all its Child Task Controllers.
* Completes its current iteration.
* Waits until all its Child Task Controllers complete *their current iteration*.

In this case:

* Finite Iteration Child Task Controllers will be either in a Done or Idle state depending on whether they completed the required number of iterations.
* Continuous Iteration Child Task Controllers will be in an Idle state.
* The Parent Task Controller that received the Stop All event will be either in an Idle or Done state depending on whether it completed the required number of iterations.

#### Abort

When a Task Controller receives an **Abort** event, it interrupts its current iteration and:

* Forwards the Abort event to all its Child Task Controllers.
* Transitions to the Stopping state and waits until its Child Task Controllers Abort as well and enters either Idle, Done or Error state depending on whether interrupting an iteration caused an error for a Finite TC or Continuous TC or not.

In this case:

* TCs will be in an error state if aborting the iteration caused an error. If interrupting the iteration does not cause an error, the TC will be either in a Done or Idle state depending on its type.
* Finite TCs

## Hierarchical data format (HDF5)

### Introduction

Since data is acquired and indexed in a hierarchical way by the TCIS, it is convenient to also save it in a similar manner. To this end, using the time-tested industry standard HDF5 file format is the most logical approach:

http://www.hdfgroup.org

http://www.hdfgroup.org/HDF5/doc/UG/index.html

"*HDF5 is an open-source technology suite for managing diverse, complex, high-volume data in heterogeneous computing and storage environments. HDF5 includes: (1) a versatile self-describing data model that can represent very complex data objects and relationships, and a wide variety of metadata; (2) a completely portable binary file format with no limit on the number or size of data objects; (3) a software library with time and space optimization features for reading and writing HDF5 data; and (4) tools for managing, manipulating, viewing, and analyzing data in HDF5 files.*"

" *An HDF5 file is a data container, similar to a file system. Within it, user communities or software applications define their organization of data objects. The basic HDF5 data model is simple, yet extremely versatile in terms of the scope of data that it can store. It contains two primary objects: groups, which provide the organizing structures, and datasets, which are the basic storage structures. HDF5 groups and datasets may also have attributes attached, a third type of data object consisting of small textual or numeric metadata defined by user applications.*

*An HDF5 dataset is a uniform multidimensional array of elements. The elements might be common data types (for example, integers, floating-point numbers, text strings), n-dimensional memory chunks, or user-defined compound data structures consisting of floating-point vectors or an arbitrary bit-length encoding (for example, 97-bit floating-point number). An HDF5 group is similar to a directory, or folder, in a computer file system. An HDF5 group contains links to groups or datasets, together with supporting metadata. The organization of an HDF5 file is a directed graph structure in which groups and datasets are nodes, and links are edges. Although the term HDF implies a hierarchical structuring, its topology allows for other arrangements such as meshes or rings.*

*HDF5 is a completely portable file format with no limit on the number or size of data objects in the collection. During I/O operations, HDF5 automatically takes care of data-type differences, such as byte ordering and data-type size. Its software library runs on Linux, Windows, Mac, and most other operating systems and architectures, from laptops to massively parallel systems. HDF5 implements a high-level API with C, C++, Fortran 90, Python, and Java interfaces. It includes many tools for manipulating and viewing HDF5 data, and a wide variety of third-party applications and tools are available.*

*The design of the HDF5 software provides a rich set of integrated performance features that allow for access-time and storage-space optimizations. For example, it supports efficient extraction of subsets of data, multiscale representation of images, generic dimensionality of datasets, parallel I/O, tiling (2D), bricking (3D), chunking (nD), regional compression, and the flexible management of user metadata that is interoperable with XML. HDF5 transparently manages byte ordering in its detection of hardware. Its software extensibility allows users to insert custom software "filters" between secondary storage and memory; such filters allow for encryption, compression, or image processing. The HDF5 data model, file format, API, library, and tools are open source and distributed without charge.*"

### HDAQ mapping to HDF5 data storage model

1. There are three group types: Image, Waveform and Iterators. The group type is stored as a group attribute named "type".
2. An Image group contains an Image dataset and an Iterators group.
3. A Waveform group contains a Waveform dataset and an Iterators group.

#### Image dataset

An image is a 1D array.

#### Waveform dataset

#### Iterator dataset

## Case studies

### Imaging: Finite frame scan movie

### Imaging: Continuous frame scan movie

Until stopped by the user.

### Imaging: Z-stack acquisition

### Imaging: Mosaic z-stack acquisition

### Imaging: ROI scan

### Ephys: Continuous measurement and plotting

### Ephys: Finite measurement

### Ephys: Stimulation and measurement

### Imaging & Ephys: In parallel and decoupled measurement of Ephys and continuous frame scan movie

Ephys Measurement and Continuous Frame Scan Movie can be done in parallel by setting up two separate Task Trees.

### Imaging & Ephys: Synchronized finite frame scan movie and recording of Ephys

### Imaging & Ephys: Synchronized point recording of fluorescence and Ephys

### Imaging & Ephys: Two-photon glutamate uncaging and simultaneous point recording of fluorescence and Ephys

### Imaging & Ephys: Two-photon uncaging, light induced dendritic inhibition and recording of Ephys

# Data and HW-trigger dependencies ideas

So here are some scenarios for data exchange/hw triggering and some thoughts that can get you started. I'll go on using several cases that are encountered in our case first.

**1) Calibration of galvos:**

There are 3 task controllers in this case:   
  
**a)** The user interface task controller (UITC) that starts the calibration process and has 1 iteration.  
 **b)** The galvo calibration TC that sends and receives a waveform with each iteration, the total number of iterations is not known beforehand, but they are finite, i.e. the galvo calibration TC decides after each iteration whether to perform another one, for example if the galvo cannot keep up with the scan speed and switches off, then it performs another iteration with a lower scan speed.

**c)** The DAQmx TC that has two tasks set-up, one AI and one AO. Also in this case, to perform one iteration, the AO waveform must be received and then the AI waveform is sent. The AI and AO tasks are triggered internally to each other so that both the AI and AO samples are synchronous. This internal DAQmx tasks triggering is set up from the DAQmx manager panel. Since only if data is available to the AO task can it start, the AI task is triggered by the start of the AO task. This task controller is set-up to perform one iteration in total after which it gets to a done state.

Now, the way the DAQmx task controller from c) is executed as many times as the calibration TC from b) wants, is to make c) a child TC of b), and b) is a child TC of a).

When a TC has other child TCs attached to it, it has 3 ways in which it can carry out its provided iterate function:

    1) The iterate function is called and must complete before starting the child TCs.

    2) The iterate function is executed in parallel with the child TCs.

    3) The iterate function is called after all the child TCs finish their iterations and are in a Done state.

The iteration of a parent TC is complete only if both its iterate function has been called and is complete and also if all its child TC are done with their iterations.

The way of executing 1-3 is important because if this execution mode is not chosen carefully, the HW-trigger and data-exchange dependencies can cause a deadlock.

In this case, to make the data exchange between the galvo TC and the DAQmx TC possible, the galvo TC must execute in parallel with the DAQmx TC, thus having the 2) execution mode.  
  
-----------

So, what we have up to now is basically two modules, say Galvo and DAQmx that both have one Rx and one Tx channel. With each iteration data ping-pongs between them.

Below are some ideas about the behavior of the TC with data and HW dependencies:

* Source and Sink VChans can be in two states: open / closed.
* Since multiple Sink VChans can receive the same data from a Source VChan, a Source VChan is open only if all its Sinks are open.
* Sending data to a Source VChan can happen only while executing the iteration function of the TC. Data cannot be sent after the iteration function is complete.
* If one or more Source VChans are attached to a TC, its iteration function is not called until all attached Source VChans are open.
* If one or more Sink VChans are attached to a TC, its iteration function does not start until data from all Sink VChans is received.
* A Master HW-trigger generating TC generates this trigger by calling its iteration function.
* A Slave HW-triggered TC is armed by calling its iteration function.
* A Slave HW-triggered TC can have only one HW-triggering Master. A Master HW-triggering TC can have multiple HW-triggered Slaves.
* Sink and Source VChans used for data exchange that belong to a TC do not in general have to connect to the same sender/receiver. In other words a TC may have Sink VChans receiving data from different Source TC VChans and Source VChans sending data to other TC Sinks.

I'm thinking that we need to add two more states:

1) Waiting For Iteration Data

2) Waiting To Send Data

* For a TC with only Sink VChans  and no HW-trig dependencies, transition from Waiting For Iteration Data -> Iterating (calling the iteration function) can happen only if all Sink VChans received their data. Also, the transition from Running -> Waiting For Iteration Data opens the attached Source VChans.
* For a TC with only Source VChans and no HW-trig dependencies, transition from Waiting To Send Data -> Iterating (calling the iteration function) can happen only if all Source VChans are open. Once data has been sent (the iteration function completed), the sending TC updates the status of its Source VChans as closed.

The following combination of HW and Data dependecies are possible for a TC to perform its iteration function:

**a)** Purely data dependecies:

1. One or more Sink VChans.
2. One or more Source VChans.
3. One or more Sink & Source VChans.

**b)** Purely HW-trig dependecies:

1. Slave HW-trig
2. Master HW-trig

**c)** Mixed Data and HW-trig dependencies:

1. One or more Sink VChans + Slave HW-trig
2. One or moreSink VChans + Master HW-trig
3. One or more Source VChans + Slave HW-trig
4. One or more Source VChans + Master HW-trig
5. One or more Sink & Source VChans + Slave HW-trig
6. One ore more Sink & Source VChans + Master HW-trig

The uses of each case are below detailed:

**a1)** TC in Initial state, gets a Start event, switches to Running state and sends an Iterate event to self. In the Running state, if its iteration function must be called before the child TCs, the Iterate event causes a transition to the Waiting for Iteration Data and sends an event to the TC that owns the Source VChan connecting to its Sink that the data channel is open.

If the child TCs must start first, a Start event is sent to them and the parent TC stays in the Running state. After they complete, a transition to Waiting for Iteration Data occurs and just like before, the channels are open. Once data is received, the TC moves to Waiting for Iteration Completion.

**a2)** TC in Initial state, gets a Start event, sets the status of its Source VChans to closed, sends an Iterate event to self and switches to Running state. Again 3 possible execution modes. In this state, the TC owing the Source VChan can receive an event from the TC owing a Sink VChan connected to this Source, that the Source VChan is open, in which case it updates the status of this source. Also in the meantime, the Iterate event sent to itself is received in this Running state and if all Source VChans are open it executes its iteration function and goes to Waiting For Iteration Completion. If not all Source VChans are open, then it switches to Waiting to Send Data where it waits for channel open events.

I think if a TC is both sending and receiving data with each iteration, that this can be done by making use of the TSQ blocking calls. There are 3 possible send/receive scenarios:  
1) receive all, send all  
2) send all, receive all  
3) mixed send/receive  
  
The choice of 1-3 depends on the module function. This can all be achieved using the TSQ blocking method: for the sender, it doesn't place more elements in the queue until it is empty and the receiver, until it read all elements or encounters a null element that marks the end of the waveform. There's not even a need for extra states in this case.  
  
A similar issue is also when HW-trig must be combined with SW dependencies. The module's functionality decides in which order these dependencies must be satisfied. In this multi-dependencies scenario, we have to always specify the order in which they must be satisfied, but this is done at the moment of implementing the module's functionality.  
  
The way things are implemented now, is that if we have HW triggering, in the case of the Master HW triggering TC, the iteration function is called only after the slaves are armed. This means, that by default, the HW trig dependency must be satisfied first, and only after the iteration function is called, data can be exchanged. This is not the case for the Slave HW triggered TC. In this case we still have the freedom to choose the order in which these dependencies have to be satisfied.  
  
Somehow we should also have the freedom to choose the order of satisfying these dependencies for the Master HW-triggering TC. I have 2 possible ideas here:  
a) To take out the state where the master is waiting for armed slaves and treat the HW dependency in the same way as the software dependency by exchanging a data element and making use of the TSQ call blocking.  
b) To add multiple iteration function pointers that are called one after each other, each with its own set of dependencies. Each iteration function from this set is executed once per iteration.  
  
I think it would be better if we could solve this by method a). Treating HW-trig and software dependencies the same way and even simplifying the state machine. Like this, sending data that is HW trig dependant should be no problem. Also we have to consider that a master can trigger multiple slaves just like a source can send data to multiple sinks, both of which should be done of all sinks are empty.

# Implementation of HW-trigger and data-exchange dependencies

A TC has 3 execution modes:

1. Completing its iteration function before starting its child TCs.
2. Executing its iteration function in parallel to the execution of its child TCs.
3. Executing its iteration function after child TCs complete their iterations.

In each of these modes, the child TCs are executed in parallel in separate threads. If the TC iteration has HW-trigger and data-exchange dependencies, these can be satisfied by exchanging data between the various TCs while they are in the Completing Iteration State through thread-safe queues (TSQs) that form the link between a Source and Sink VChans. As it will be clarified below, both HW-trigger and data-exchange dependencies will be solved by exchanging data through TSQs and enforcing a certain order of events to take place.

The mechanism by which data can be exchanged between a Source and multiple Sink VChans is through calling CmtReadData such that if this function is called and the required data is not available, the function does not return from its call and the thread remains blocked.