TRIUMF

Python in midas

Ben Smith

Midas Workshop 2023

Discovery, accelerated

Why python?

- Requested by users at last workshop
- Initial concept was to replace hacky scripts
  - python scripts that called odbedit
  - midas file > mdump > parse text > python analysis
  - many more examples of calling midas command-line tools from Tcl/bash/csh/python/perl scripts

- Python is common (only?) language that students know
- Python interfaces nicely with C

Design goals

- Usability
  - Make it "pythonic"
    - E.g. error-checking via exceptions, not return codes
  - Make simple interfaces with sensible defaults
  - Give people the tools they need
- Maintainability
  - Don't be a burden on the core C/C++ code
  - Make it simple to add new features to python code

What is implemented

- Midas file reader
  - Pure-python implementation
- Midas client
  - ODB access, run transitions, event buffers, RPC, alarms, messages, history...
  - Python wrapper calls the C++ code
- Frontend framework
  - Periodic/polled equipment - same concepts as C/C++ frontends, just written in python

File reader

```
import midas.file_reader

my_file = midas.file_reader.MidasFile("/path/to/file.mid.lz4",
                                      use_numpy=True)

# Get ODB as a dict
odb = my_file.get_bor_odb_dump().data
run_number = odb["Runinfo"]["Run number"]

# Loop over events in file
for event in my_file:
  # Bank data is either numpy array or python tuple
  some_counter = event.banks["SCAN"].data[0]
```

Midas client

```
import midas.client

if __name__ == "__main__":
    client = midas.client.MidasClient("pytest")

    # Get data from ODB
    state = client.odb_get("/Runinfo/State")

    if state == midas.STATE_STOPPED:
      # Set data in ODB
      client.odb_set("/pyexample", {"an_int": 1, "a_dbl": 4.56})

    # Write message to midas log (set is_error=True for error)
    client.msg("Hello from python")
```

Full list of midas.client functions

- communicate
- connect_to_other_client
- create_alarm_class
- create_evaluated_alarm
- deregister_disconnect_callback
- deregister_event_request
- deregister_message_callback
- deregister_transition_callback
- disconnect
- disconnect_from_other_client
- get_message_facilities
- get_midas_version
- get_recent_messages
- get_triggered_alarms
- hist_get_data
- hist_get_events
- hist_get_recent_data
- hist_get_tags
- msg
- odb_delete
- odb_exists
- odb_get
- odb_get_link_destination
- odb_last_update_time
- odb_link
- odb_rename
- odb_set
- odb_stop_watching
- odb_watch
- open_event_buffer
- pause_run
- receive_event
- register_deferred_transition_callback
- register_disconnect_callback
- register_event_request
- register_jrpc_callback
- register_message_callback
- register_transition_callback
- reset_alarm
- resume_run
- send_event
- set_transition_sequence
- start_run
- stop_run
- trigger_internal_alarm

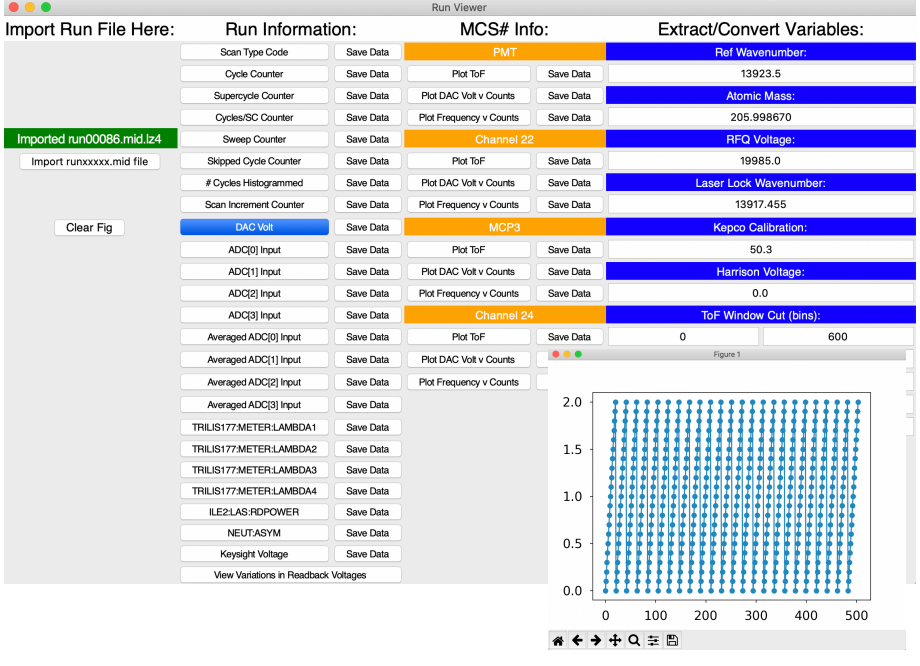
Example use - data analysis

- Lots of students only know python/numpy
- For several experiments I convert midas events into experiment-specific data structures

```
reader = pol_data.PolFile(file_path)

for pol_event in reader:
    print(pol_event.keysight_voltage)
```

- One also has a custom analysis GUI in python/Tk



Example use - frontend

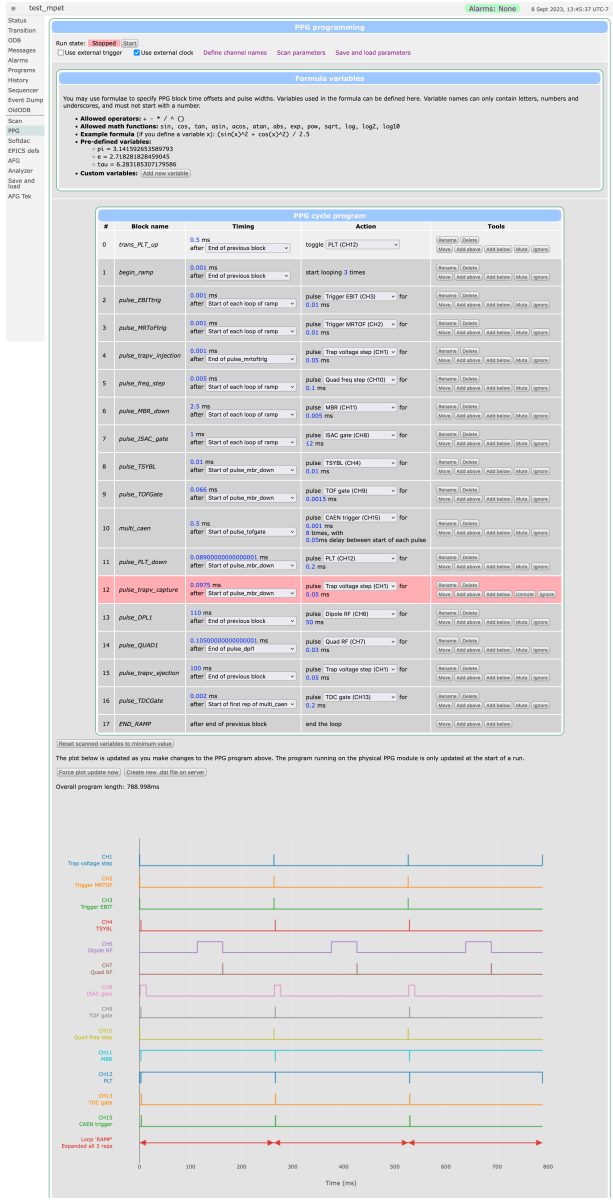
- Many slow control devices use LXI/vxi-11 protocol
  - E.g. Tektronix/LeCroy scopes/function generators
- Python has a vxi11 package available on PyPi
  - `pip install python-vxi11`, and you don't have to worry about the low-level details
- Frontend in python doesn't require much code to be written (and performance isn't important in this case)
- Could all be done in C++, but quicker for me to implement in python

Example use - complex configuration

- Many of my experiments do complex computations before configuring a device
  - E.g. computing RF frequencies and modulation based on human-understandable inputs
- Students often send me the calibration routines in python
  - May as well keep that code rather than re-writing!
- I write custom webpages that show the input and output
  - Use RPC and/or ODB hotlinks
  - Much nicer to report problems early, rather than waiting until user tries to start a run

Example use - PPG compiler

- PPG is a 32-channel timing sequencer used at TRIUMF. Accepts low-level bytecode.
- Users wanted a simple way to program it
- Python frontend/compiler and webpage!
  - Pulses/loops etc stored in ODB
  - Webpage issues RPC to get a visual display of the sequence
  - Bytecode generated at run start
- Compiler would have been much more tedious to write in C++



<technical details>

Advanced usage - API

- `odb_set` has parameters that let you do powerful things, especially when passing a dict
- Defaults are sensible (principle of least surprise), but options are there if you need them
- All the options are clearly documented in `client.py`
- Same idea for many of the functions

```
def odb_set(self, path, contents, create_if_needed=True,
            remove_unspecified_keys=True, resize_arrays=True,
            lengthen_strings=True, explicit_new_midas_type=None,
            update_structure_only=False):
    """
    Set the value of an ODB key, or an entire directory. You may pass in
    normal python values, lists and dicts and they will be converted to
    appropriate midas ODB key types (e.g. int becomes midas.TID_INT, bool
    becomes midas.TID_BOOL).

    Sensible defaults have been chosen for converting python types to the C
    types used internally in the midas ODB. However if you want more control
    over the ODB type, you may use the types defined in the ctypes library.
    For example, regular python integers become a midas.TID_INT, but you can
    use a `ctypes.c_uint32` to get a midas.TID_DWORD.

    If you are setting the content of a directory and care about the order
    in which the entries appear in that directory, `contents` should be a
    `collections.OrderedDict` rather than a basic python dict. See the note
    in `odb_get` for more about dictionary ordering.

    Args:
        * path (str) - The ODB entry to set. You may specify a single array
            index if desired (e.g. "/Path/To/My/Array[1]").
        * contents (int/float/string/tuple/list/dict etc) - The new value to set
        * create_if_needed (bool) - Automatically create the ODB entry
            (and parent directories) if needed.
        * remove_unspecified_keys (bool) - If `path` points to a directory
            and `contents` is a dict, remove any ODB keys within `path` that
            aren't present in `contents`. This means that the ODB will exactly
            match the dict you passed in. You may want to set this to False
            if you want to only update a few entries within a directory, and
            want to do so with only a single call to `odb_set()`.
        * resize_arrays (bool) - Automatically resize any ODB arrays to match
            the length of lists present in `contents`. Arrays will be both
            lengthened and shortened as needed.
        * lengthen_strings (bool) - Automatically increase the storage size
            of a TID_STRING entry in the ODB if it is not long enough to
            store the value specified. We will include enough space for a
            final null byte.
        *  explicit_new_midas_type (one of midas.TID_xxx) - If you're
            setting the value of a single ODB entry, you can explicitly
            specify the type to use when creating the ODB entry (if needed).
        * update_structure_only (bool) - If you want to add/remove entries
            in an ODB directory, but not change the value of any entries
            that already exist. Only makes sense if contents is a dict /
            `collections.OrderedDict`. Think of it like db_check_record from
            the C library.

    Raises:
        * KeyError if `create_if_needed` is False and the ODB entry does not
            already exist.
        * TypeError if there is a problem converting `contents` to the C
            type we must pass to the midas library (e.g. the ODB entry is
            a TID_STRING but you passed in a float).
        * ValueError if there is a non-type-related problem with `contents`
            (e.g. if `resize_arrays` is False and you provided a list that
            doesn't match the size of the existing ODB array).
        * midas.MidasError if there is some other midas issue.
    """
```

Advanced usage - ctypes

- ODB gives fine control over how data is stored
  - uint8_t, int16_t, float, double etc
- Python just has integer and float (really a double)
- If you care, can use `ctypes` library to specify exact data type you want (e.g. `ctypes.c_uint32`) and/or specify the midas data type (e.g. `midas.TID_UINT32`)

Implementation of midas.client - C side

- Python ctypes library comes as standard and can call C functions
- Midas is now compiled as C++, which is much harder to call from python
- I added a midas_c_compat.cxx/h file that provides a C-compatible wrapper of the functions I need
  - Generally trivial, some shenanigans for functions that populate `std::vector<std::string>` etc (`char***`)
  - All exposed functions use `extern "C"`

Implementation of midas.client - py side

- midas.MidasLib uses ctypes' ability to call C functions
- Automatically discovers all the functions in libmidas-c-compat.so
- Intercepts all the return values from C functions
  - If not 1 (SUCCESS), raises an Exception
  - Whitelist of functions that return other codes (e.g. al_reset_alarm can return "AL_RESET") or that don't return status codes at all

Process to expose a new function

- Write a wrapper in midas_c_compat.cxx/h (with conversion between C-compatible and C++ if needed)

```
INT c_al_reset_alarm(const char *alarm_name) {
   return al_reset_alarm(alarm_name);
}
```

- Write python function in midas.client (with conversion between python and ctypes if needed)

```
def reset_alarm(self, alarm_name):
    c_name = ctypes.create_string_buffer(bytes(alarm_name, "utf-8"))
    self.lib.c_al_reset_alarm(c_name)
```

- Edit MidasLib if this C function is "special" and doesn't return a status code

</technical details>

Future plans

- I recently added support for accessing the history system
  - Anything you did with mhist can be done in python
- I don't think there are any major midas features left that can't be accessed through python
- Keep emailing me with requests though!
  - There are always use cases that I haven't thought of (e.g. 40kHz of events through a python frontend, getting recent messages from the log, ...)
  - I'll try to implement/improve ASAP

Summary

- Lots of progress on using python in midas since the last workshop
- Should be easy to maintain as midas evolves
- You should be able to write scripts, frontends and data analysis code all in python
- I really like having custom webpages talking to python code via JRPC
- Let me know if there are any features or improvements you'd like to see!