MicroBooNE Lifetime Database Interface

Brandon Eberly

September 13, 2016

Introduction

- MicroBooNE would like to read electron lifetime calibrations from a database for use in larsoft modules
  - Currently use a fcl-configured parameter in DetectorPropertiesStandard

- Our lifetime database holds four parameters per interval of validity: two fit parameters and their uncertainties

- Want to timestamp the parameters using run number rather than use seconds from Unix epoch

- Needed to develop a lifetime database interface in larsoft on short notice so that lifetime results could be tested on calorimetry
  - DetectorPropertries does not implement any of its database hooks, so decided to ignore it for now

Database Interface Design

•Started with the design pattern used for other MicroBooNE database interfaces: detector pedestals, channel status, pmt gains

•Implemented entirely in uboonecode, so did not bother with the usual abstraction layers

•Single provider class, with an instance owned by a service class

•Provider class inherits from DatabaseRetrievalAlg in larevt – provides the database hooks in the same way as the other providers that MicroBooNE uses

•Provider class also uses a new lifetime object container class in uboonecode

•Feature branch: **eberly_dblifetimehack** in uboonecode and larevt

Container Class

```
#include "larevt/CalibrationDBI/IOVData/ChData.h"

namespace lariov {
  /**
     \class ElectronLifetime
  */
  class ElectronLifetimeContainer : public ChData {

    public:

      /// Constructor
      ElectronLifetimeContainer(unsigned int ch) : ChData(ch) {}

      /// Default destructor
      ~ElectronLifetimeContainer() {}

      float ExpOffset()       const { return fExpOffset; }
      float TimeConstant()    const { return fTimeConstant; }
      float ExpOffsetErr()    const { return fExpOffsetErr; }
      float TimeConstantErr() const { return fTimeConstantErr; }

      void SetExpOffset(float val)       { fExpOffset       = val; }
      void SetTimeConstant(float val)    { fTimeConstant    = val; }
      void SetExpOffsetErr(float val)    { fExpOffsetErr    = val; }
      void SetTimeConstantErr(float val) { fTimeConstantErr = val; }

    private:

      float fExpOffset;
      float fTimeConstant;
      float fExpOffsetErr;
      float fTimeConstantErr;

  }; // end class
} // end namespace lariov
```

Provider Class

- The implementation of Update() follows that of the other providers

```
class UbooneElectronLifetimeProvider : public DatabaseRetrievalAlg {

  public:

    /// Constructors
    UbooneElectronLifetimeProvider(const std::string& foldername,
                                   const std::string& url,
                             const std::string& tag="");

    UbooneElectronLifetimeProvider(fhicl::ParameterSet const& p);

    /// Reconfigure function called by fhicl constructor
    void Reconfigure(fhicl::ParameterSet const& p);

    /// Update Snapshot and inherited DBFolder if using database.  Return true if updated
    bool Update(DBTimeStamp_t ts);

    /// Retrieve lifetime information
    const ElectronLifetimeContainer& LifetimeContainer() const;
    float ExpOffset() const;
    float TimeConstant() const;
    float ExpOffsetErr() const;
    float TimeConstantErr() const;

    //hardcoded information about database folder - useful for debugging cross checks
    static constexpr unsigned int NCOLUMNS = 5;
    static constexpr const char* FIELD_NAMES[NCOLUMNS]
      = {"channel", "exponential_offset", "err_exponential_offset", "time_constant", "err_time_constant"};
    static constexpr const char* FIELD_TYPES[NCOLUMNS]
      = {"bigint", "real", "real", "real", "real"};

  private:

    DataSource::ds fDataSource;

    Snapshot<ElectronLifetimeContainer> fData;

    const unsigned int fLifetimeChannel = 0;
};
```

Service Class

- The constructor implementation passes fcl parameters to the provider and registers the PreProcessEvent call back

- Note the timestamp is the run number

```
  class UbooneElectronLifetimeService {

    public:

      UbooneElectronLifetimeService(fhicl::ParameterSet const& pset, art::ActivityRegistry& reg);
      ~UbooneElectronLifetimeService(){}

      void PreProcessEvent(const art::Event& evt) {
        fProvider.Update( (DBTimeStamp_t)evt.run() );
      }

      const UbooneElectronLifetimeProvider& GetProvider() const {
        return fProvider;
      }

    private:

      UbooneElectronLifetimeProvider fProvider;
  };
}//end namespace lariov

DECLARE_ART_SERVICE(lariov::UbooneElectronLifetimeService, LEGACY)
```

Time Stamp

- The underlying code for the database interface (DatabaseRetrievalAlg and DBFolder in larevt) assumes that the passed in timestamp is in nanoseconds from the Unix epoch
  - This is enforced through the TimeStampDecoder class, which converts the input DBTimeStamp_t to the string format used for database queries
  - Enforcement check: DBTimeStamp_t must be a 19 digit integer
- Had to relax this check in order to use run number as a timestamp (hence the larevt feature branch)
- Also fixed a bug in the IOVTimeStamp class caused by the assumption that all string timestamps will have a decimal point

TimeStamp Changes

- The change:

```
IOVTimeStamp TimeStampDecoder::DecodeTimeStamp(DBTimeStamp_t ts) {

  std::string time = std::to_string(ts);

  //microboone stores timestamp as ns from epoch, so there should be 19 digits.
  if (time.length() == 19) {
    //make timestamp conform to database precision
    time = time.substr(0, 10+kMAX_SUBSTAMP_LENGTH);

    //insert decimal point
    time.insert(10,".");

    //finish construction
    IOVTimeStamp tmp = IOVTimeStamp::GetFromString(time);
    return tmp;
  }
  else if (time.length() < kMAX_SUBSTAMP_LENGTH && ts!=0) {
    IOVTimeStamp tmp = IOVTimeStamp::GetFromString(time);
    return tmp;
  }
  else {
    std::string msg = "TimeStampDecoder: I do not know how to convert this timestamp: " + time;
    throw IOVDataError(msg);
  }
```

- The bug fix:

```
IOVTimeStamp IOVTimeStamp::GetFromString(const std::string& ts) {
  unsigned long stamp = std::stoul(ts.substr(0, ts.find_first_of(".")));
  unsigned long stamp;
  std::string substamp_str;
  if (ts.find_first_of(".") == std::string::npos) {
    stamp = std::stoul(ts);
    substamp_str = "0";
  }
  else {
    stamp = std::stoul(ts.substr(0, ts.find_first_of(".")));
    substamp_str = ts.substr(ts.find_first_of(".")+1);
  }

  std::string substamp_str = ts.substr(ts.find_first_of(".")+1);
  if (substamp_str.length() > kMAX_SUBSTAMP_LENGTH) {
    throw IOVDataError("SubStamp of an IOVTimeStamp cannot have more than six digits!");
  }
```

Long Term

- It is not my intention that this be the permanent solution for MicroBooNE
  - Though it would be nice to push the larevt feature branch to develop since it fixes a bug and should not break other experiment's code

- What I think the right model is:
  - DetectorPropertiesStandard: make the electron lifetime parameter a provider interface that has functions that return the lifetime and uncertainty
    - Default implementation can still be fcl-configurable for those without a DB

  - MicroBooNE implementation reads however many parameters we want out of the database, and knows how to calculate the lifetime and error from them

  - Implement the Update() function in DetectorPropertiesStandard to call Update() for each provider
    - This means the interface classes need an Update() – something that was removed from the design pattern of other interfaces

  - DetectorPropertiesServiceStandard needs to register a callback for Update()
    - Separate services required for NOvA and MicroBooNE-style databases?

Long Term

- Timestamping: Current way of doing this is unwieldy – decoder needs to intuit what kind of timestamp is being used from its length

- Possible remedies for the long term
  - DBTimeStamp_t changed to a type that also includes a string or status bit to indicate the timestamp type

  - Remove all timestamp decoding and force users to submit timestamps as ints or floats that match those used in the database