MTConnect Adapter Implementation Manual

Version 0.7

Initial draft for review
Prepared for: AMT
Prepared by: William Sobel et al.
Date: May 29, 2008

Table of Contents

Revision History

Date	Description	Author	Version
3/24/08	Initial Iteration	Paul Wicks	0.1
4/4/08	Corrected and many edits	Will Sobel	0.2
4/9/08	Incorporated comments by Jennings	Will Sobel	0.3
4/10/08	Added example adapters and reordered sections.	Will Sobel	0.4
4/12/08	Completed examples	Will Sobel	0.5
4/20/08	Added sample data	Will Sobel	0.6
5/28/08	Changed date time format to ISO 8601	Will Sobel	0.7

1. Overview

This manual describes the methods used to implement an adapter so devices can achieve rapid compliance with the MTConnect standard. The objective is to wrap a device vendor's specific Application Programming Interface (API) in a standardized interface and then expose data in a standard format. The adapter provides a common method for gather data from devices, regardless of its type or origin. Adapters simply retrieve data from device and write to a socket when the data changes. The agents that consume this data handle all the HTTP communication. This spares the developer of device adapters from having to deal with the details of Web application architecture such as RESTful URLs, HTTP request headers, or XML.

Top obtain a copy of the MTConnect standard, go to the MTConnect.org website, sign in, and download the latest version of the standard.

1.1. Goals

The purpose of an adapter is to enable equipment to stream data to agents that power MTConnect enabled applications. The MTConnect adapter design pattern enables the separation of concerns that preserves modularity, reduces dependencies, and reduces level of effort. MTConnect adapters should be simple and easy to understand and enhance.

The purpose of this manual is to provide design guidelines, high-level technical specifications, and a few specific examples. We are providing a common framework with working code for a few specific APIs. The developer should be able to use our examples as a guide to creating their specific implementation, but it is doubtful they will be able to lift the examples out verbatim since each device is different.

We have constructed a framework that we hop will allow the developer to focus on gathering data from their specific device and setting the appropriate values in the devices data objects. The framework will take care of all socket communication, buffer management, and workflow. The overarching goal is to provide the resources to become MTConnect compliant as rapidly as possible and it is to this end we are providing the reference documentation, adapter framework, and agent.

1.2. Background

The MTConnect standard is intended to establish a common mechanism for collecting and exchanging sensor and telemetry data in manufacturing process control. Emphasis is placed on separation of concerns between layers of the software architecture. This enables the development of adapters to be relatively simple. In addition, very few assumptions are made regarding the type and behavior of equipment as well as the protocols used to communicate data. An adapter communicates with a device using its specific API, then places the resulting data into a stream of data that can be ingested by a separate agent, eliminating the need for the agent to have any knowledge or understanding of that device's API or communications protocols.

1.3. Caveats

The larger subject of using adapters to assemble an application is beyond the scope of this document. Since the adapter is not a requirement of the MTConnect standard, the use of the adapter and the architecture provided herein are only suggestions and may be used or not at the discretion of the developers.

It is important to understand that this is not a reference manual to document the MTConnect Standard. For information regarding the standard, refer to *The MTConnect Draft Standard*. This is the recommended architecture for quick MTConnect compliance. An implementation can disregard this manual and still be fully MTConnect compliant.

1.4. Terminology

Adapter	An optional software component that connects the Agent to the Device.
Agent	A process that implements the MTConnect specification, acting as an interface to the device.
Alarm	An event that requires attention and indicates a deviation from normal operation.
Application	A process or set of processes that access the MTConnect *Agent* to perform some task.
Attribute	A part of an element that provides additional information about that element. For example, the `name` element of the `Device` is given as `<Device name="mill-1">...</Device>`
Class	An encapsulation of data and functions.
Client	A process that is receiving information from this adapter.
Data Source	The machine, device, sensor, or controller that is supplying the data to the adapter using an API.
Datum	A single piece of data collected from the device.
Device	A piece of equipment or component capable of providing data.
Event	A change in state that occurs at a point in time. Note: An event does not occur at predefined frequencies.
Sample	A data point for continuous data items, that is, the value of a data item at a point-in-time.
Socket	When used concerning interprocess communication, it is a connection between two end-points (usually processes). Socket communication most often uses TCP/IP as the underlying protocol.

TCP/IP	TCP/IP is the most prevalent stream-based protocol for interprocess communication. It is based on the IP (Internet Protocol) stack and provides the flow-control and reliable transmission layer on top of the IP routing infrastructure.
UML	Unified Modeling Language. A standardized method of representing software design using diagrams.

2. Intended Use

An adapter is used to provide an application agent with a stream of data in a normalized format, regardless of differences in controller hardware and corresponding APIs. The bigger picture is that any application that seeks to collect and exchange sensor and telemetry data, to be simple and flexible, uses adapters to contain differences and avoid the larger concerns of the application.

There are two high-level uses that an adapter fulfills:

1. The adapter collects data from a controller, sensor, or data bus.
2. The adapter streams normalized data for external consumption.

There are, however, a number of prerequisites implied by these responsibilities:

1. To conserve computing resources, only changes in data should be collected from the sources.
2. To collect data, the adapter must call each of the device-specific APIs that correspond with the data elements specified in the MTConnect standard.
3. To collect data effectively, the adapter must select an appropriate sampling rate.
4. To stream data for external consumption, the adapter must provide a socket server for streaming data to MTConnect agents.
5. In the case that the data represents an event, such as an alarm, it is especially important to normalize the data – translate from its native form to the vocabulary defined in the MTConnect standard.

   Note: Units conversion is not the responsibility of the Adapter–the Agent will convert units for samples.

The following diagram illustrates the responsibilities of the MTConnect components:

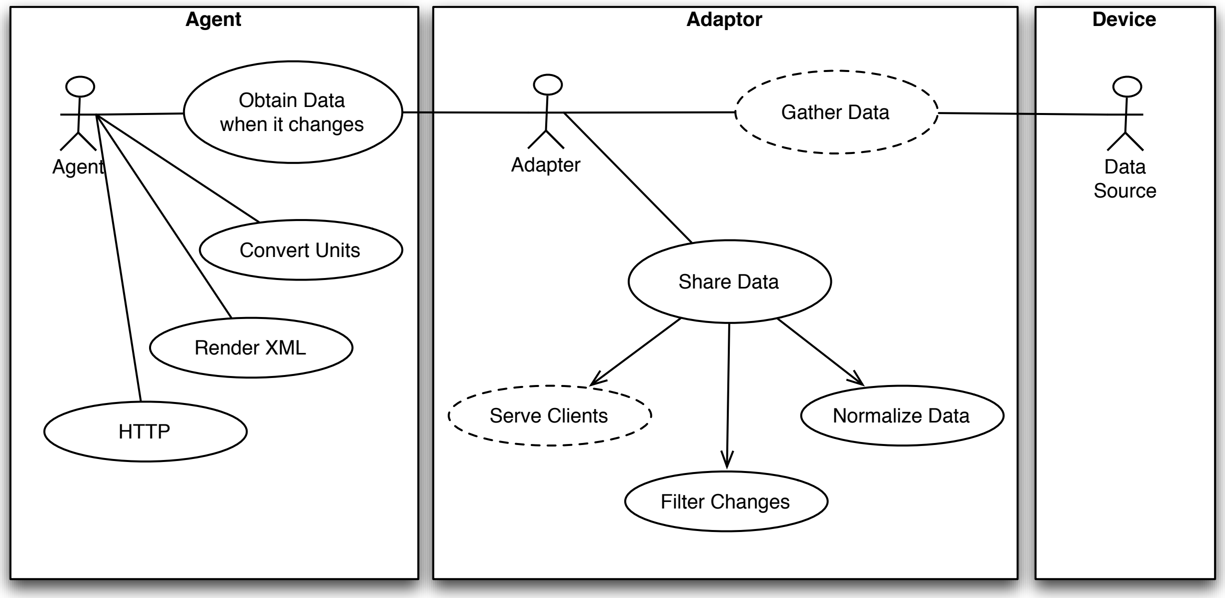

Figure 1. MTConnect Adapter Use Cases

Note: Actors (depicted as stick-figures) represent the objects in the solution domain, not users. The use cases (shown as ovals) may or may not correspond to discrete methods or even collaborating objects. A sample, for instance, could either be a data structure within the adapter or a separate object. This will be discussed next.

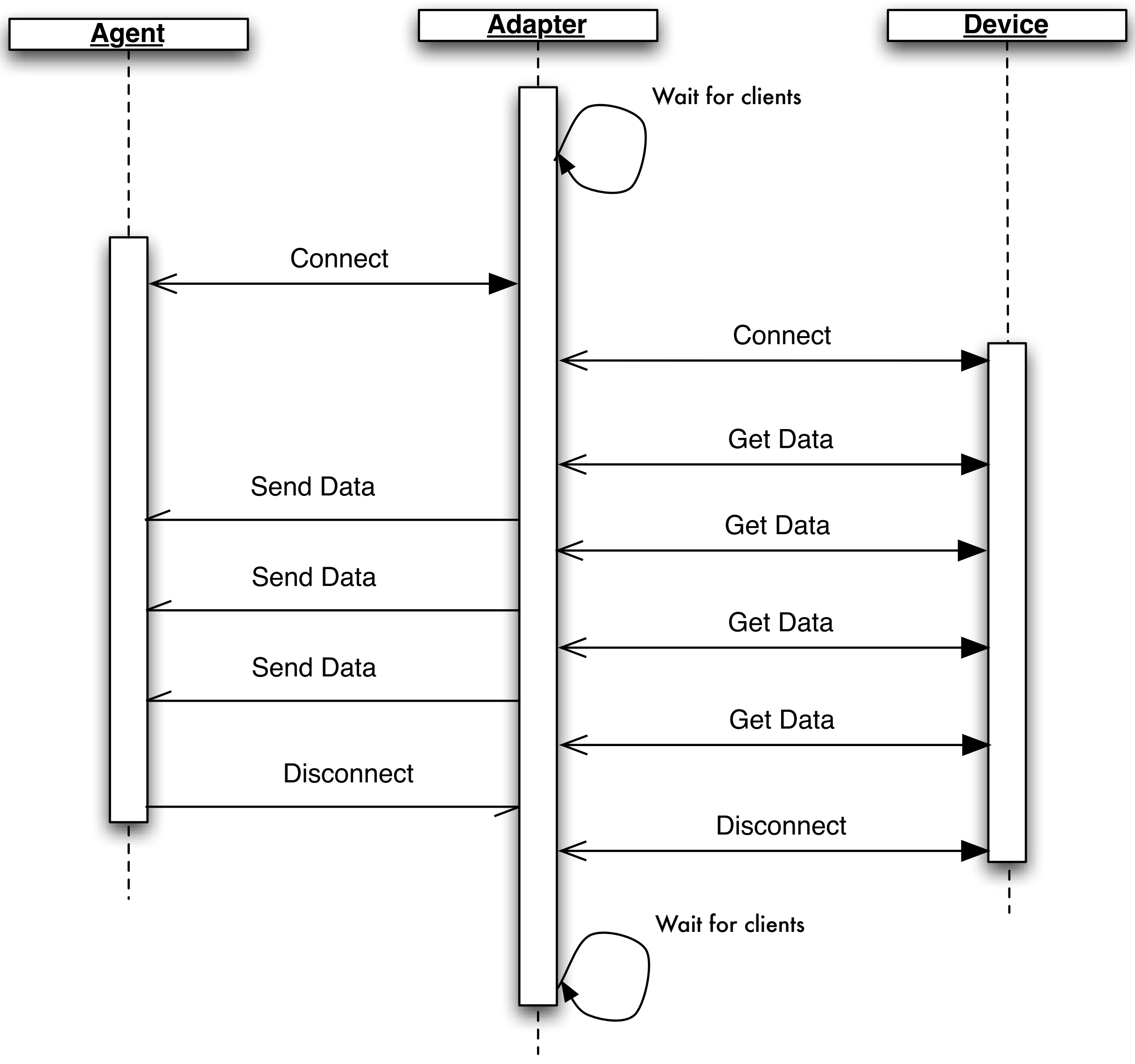

Figure 2. Data Flow

The data flow for the adapter is very simple. It looks for clients until one connects and then it connects to the device and begins streaming data back to the client. When the client disconnects from the agent the agent in turn, disconnects from the device. This will reduce the load on the device when the agent is not delivering data to a client.

3. Anatomy of an Adapter

An adapter reads data from a data source, for example, a controller and writes it to a socket. From the perspective of the agents that are its clients, it is read-only (an adapter cannot be commanded externally at this time). An adapter should manage data from one data source–an Agent can handle multiple data streams from multiple adapters.

3.1. System Libraries

Following the basic principles of modular software development, begin by including access to objects or libraries responsible for the following:

1. Socket Server
2. Time
3. File I/O, optionally network attached or virtualized
4. API for the particular device

3.2. Framework Classes and Relationships

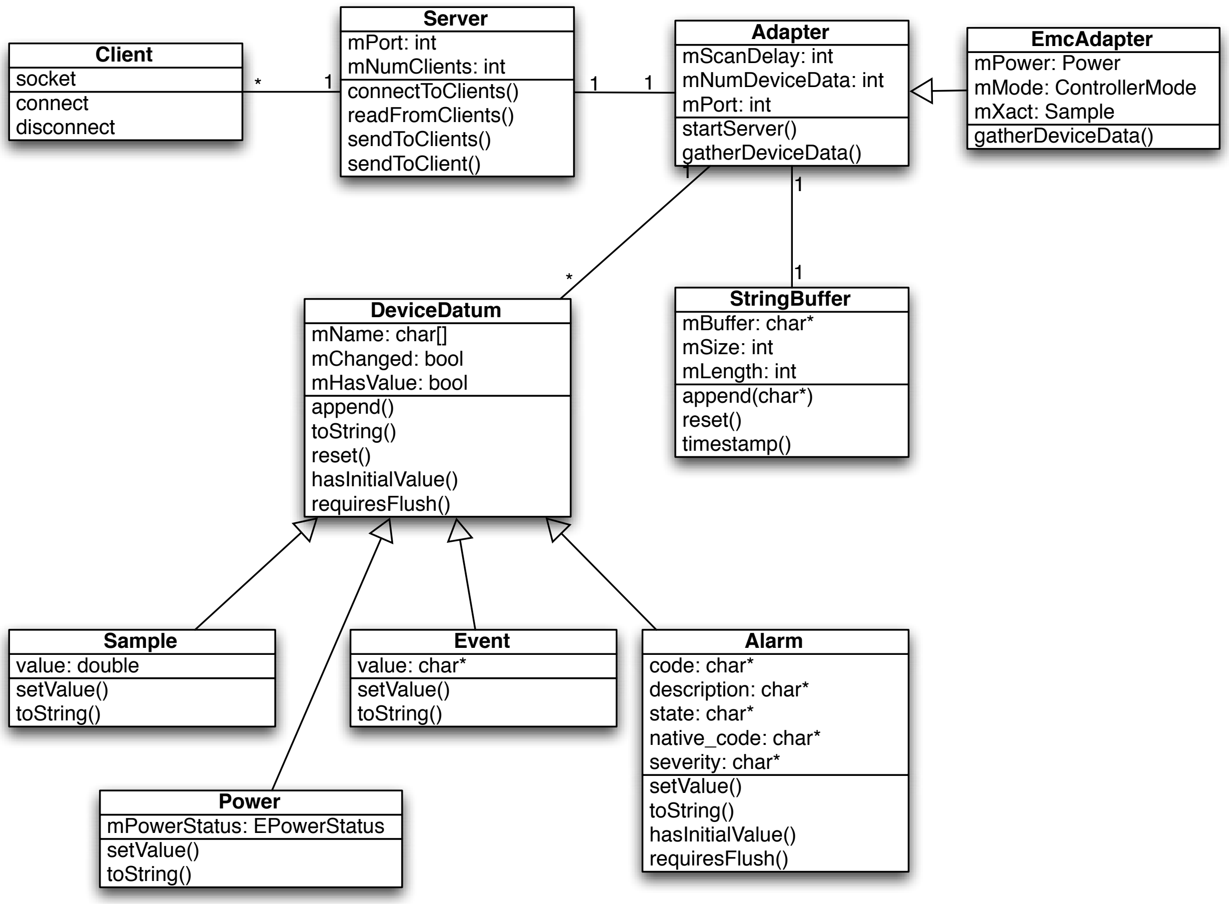

Figure 3. Object Model Diagram

3.3. Adapter Classes

There should be classes or modules responsible for the following:

1. `DeviceDatum`
   The `DeviceDatum` abstraction detects and records changes to values and sets the `mChanged` flag.

2. The `Adapter` Itself
   The adapter tracks clients, marshals the data, establishes a rate for data acquisition, creates a socket, polls for changes, writes a stream of data to the socket, and handles errors in communication with clients and the data source.

Data is obtained in the format dictated by the data source's API. This is converted into a normalized format for the MTConnect standard.

The following data elements are attributes of the `DeviceDatum` abstraction:

Variable	Purpose
`mName`	The name of the data value
`mChanged`	Whether the data has changed since the last recorded value

A `DeviceDatum` can have a sub-type for Samples:

Variable	Purpose
`mValue`	A floating point value for the sample

Or Events:

Variable	Purpose
`mValue`	A character value for the event. The value must conform to the MTConnect specification. Can also be a numeric value in the case of a line number

And Alarms:

Variable	Purpose
`mCode`	The code for the alarm according to the MTConnect standard
`mNativeCode`	The native code for the alarm from the API
`mDescription`	The native description of the alarm
`mSeverity`	The severity of the alarm
`mState`	The state of the alarm: `eINSTANT`, `eACTIVE` or `eCLEARED`

The complete vocabulary is defined in the *device_datum.hpp* file located in the `src` directory of the framework.

3.4. Activity

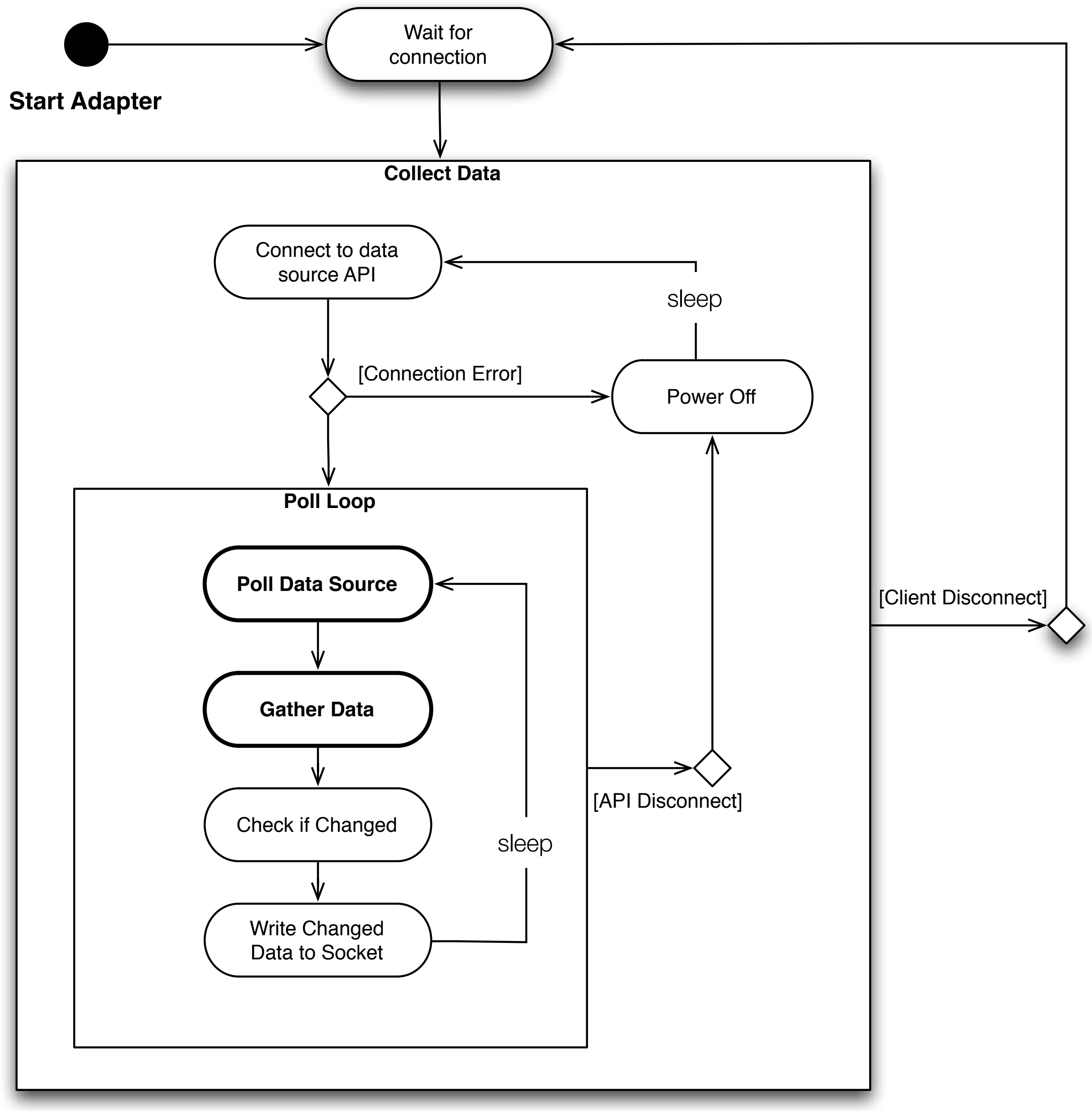

Figure 4. Adapter Activity Diagram

At the highest level, the responsibilities of an Adapter are: poll the data source, via its API, in order to set the value of the instance of the `DeviceDatum` object associated with the information using the type or vocabulary standardized in the MTConnect specification; to send that data, if it had changed, to the Adapter's list of currently active clients. Conditional processing occurs if the client fails to respond or the API disconnects.

Your responsibility is to provide the functionality that gathers data from the data source and set the values in the appropriate objects–these items are in **bold** in the diagram above. A toolkit has been provided to provide all the remaining requirements.

4. Construction

The following example code was written using C++. An MTConnect adapter may be implemented using any programming language capable of interfacing with the device controller. If additional examples in other languages are desired, we will consider providing alternative implementations if there is sufficient demand.

4.1. Coding Conventions

Variable	Purpose
`DeviceDatum`	Class names are camel case starting with an upper case methods.
`mNumItems`	Member variables always start with a lower case m and are camel case.
`aCode`	Attributes are prefixed with a lower case a.
`status`	Local variables start with lower case and camel case for words
`MAX_ITEMS`	Constants are in upper case
`ESeverity`	An enumeration type
`eWARNING`	A member of an enumeration

All file names will be written using underscores and with the extension `.hpp` and `.cpp` as in `emc_adapter.cpp`. In the following section we will provide examples of Adapters and where to add the device specific changes. The next section will contain a more in-depth discussion on the implementation of the framework.

5. Adapter Examples

Now that you understand the architecture of the adapter, this section will show you how to change just the necessary parts of the sample adapter code to implement your adapter. We will present two adapters, the minimal adapter that does not connect to an actual device and sets the power on and off. This is for illustrative purposes only and can for the basis of a simple adapter.

In the next section following the minimal adapter, we will present a full adapter using LinuxCNC. The techniques used in the LinuxCNC adapter can be used in developing your own adapter. Full source for these adapters as well as others are available from the MConnect website.

5.1. Minimal Adapter

We will start with a minimal adapter that does not connect to any device, but only cycles the power on and off. The purpose of this adapter is to show the minimal amount of code to achieve a working adapter.

5.1.1. Adapter Class Definition

fake_adapter.hpp

```
1.  class FakeAdapter : public Adapter
2.  {
3.  protected:
4.    /* Define all the data values here */
5.
6.    /* Events */
7.    Power mPower;
8.
9.  public:
10.   FakeAdapter(int aPort);
11.   ~FakeAdapter();
12.
13.   virtual void gatherDeviceData();
14. };
```

Line 7 declares our one datum we're collecting. The gatherDeviceData method, on line 13, is called once information is required from the adapter. This method MUST be provided by every adapter, otherwise the application will fail to compile.

5.1.2. Adapter Methods

fake_adapter.cpp

```
1.  FakeAdapter::FakeAdapter(int aPort)
2.    : Adapter(aPort),
3.      mPower("power")
4.  {
5.    addDatum(mPower);
6.  }
```

```
7.
8.  FakeAdapter::~FakeAdapter()
9.  {
10. }
11.
12. void FakeAdapter::gatherDeviceData()
13. {
14.   if (!mPower.setValue(Power::eON))
15.     mPower.setValue(Power::eOFF);
16.   sleep(5);
17. }
```

Every datum must be constructed and given a name (line 3). In the constructor it must be added to the list of adapters using the addDatum method (line 5). This is required for all adapters and all the data you wish to collect.

The gatherDeviceData method (line 12) sets the power to on. If the power is already on it will return false (the value did not change), so we turn it off. Then we sleep 5 seconds. This will cause the power to cycle on/off every five seconds. This adapter is provided as a simple test exercise the framework.

5.1.3. Main Program Entry Point

```
18. #include "internal.hpp"
19. #include "fake_adapter.hpp"
20. #include "server.hpp"
21. #include "string_buffer.hpp"
22.
23. int main(int aArgc, char *aArgv[])
24. {
25.   int port = 7878;
26.   if (aArgc > 1)
27.     port = atoi(aArgv[1]);
28.
29.   /* Construct the adapter and start the server */
30.   FakeAdapter adapter(port);
31.   adapter.startServer();
32.
33.   return 0;
34. }
```

All applications have an entry point, and in C++ the main function servers this purpose. In this simplest form the adapter has a single optional argument that specifies the port number to bind to. This is optional and will default to 7878 if not given.

On line 30, we construct the adapter as a local object. The next line (31) is the main loop of the application and never returns. This will start the socket server and poll the device at a fixed rate. The data will begin to be set once a client has connected to the adapter.

To implement a simple device that only knows its power status, one must only replace the gather data method with the appropriate call to your API and set the power data on or off. This will provide minimal compliance with MTConnect.

5.2. EMC2 Adapter (LinuxCNC)

EMC2 is a free CNC controller that can be obtained from www.linuxcnc.org. It runs on ubuntu and provides a good test platform for data collection. Since the code is freely available, we will present the implementation using this device simulating a three axis mill. We will be collecting the following infomation: Alarms, Power status, execution state, line number in the current program, the name of the program being run, and the controller's mode (auto, MDI, manual).

We will also collect the positions of the X, Y, and Z axes for both actual and comanded positions. And lastly we will collect the spindle speed and the path feedrate. This is a standard simple set of data items collected from a device.

5.2.1. Adapter Class Definition

emc_adapter.hpp

```
class EmcAdapter : public Adapter
{
protected:
  /* Define all the data values here */

  /* Events */
  Alarm mAlarm;
  Power mPower;
  Execution mExecution;
  IntEvent mLine;
  Event mProgram;
  ControllerMode mMode;

  /* Samples */
  /* Linear Axis */
  Sample mXact;
  Sample mYact;
  Sample mZact;

  Sample mXcom;
  Sample mYcom;
  Sample mZcom;

  /* Spindle */
  Sample mSpindleSpeed;

  /* Path Feedrate */
  Sample mPathFeedrate;

30. protected:
31.   EMC_STAT mEmcStatus;
32.   NML *mEmcErrorBuffer;
33.   char mErrorString[LINELEN];
34.   RCS_STAT_CHANNEL *mEmcStatusBuffer;
35.   bool mConnected;
36.
37.   char mNmlFile[1024];
38.
39. protected:
40.   bool connect();
41.   void disconnect();
42.   void actual();
43.   void commanded();
44.   void spindle();
45.   void feedrate();
46.   void program();
47.   void machine();
48.   void execution();
49.   void alarms();
50.
51. public:
52.   EmcAdapter(int aPort, const char *aNmlFile);
53.   ~EmcAdapter();
54.
55.   virtual void gatherDeviceData();
56. };
```

Lines 6-28 define the `DeviceData` objects we use to hold the data and detect changes. The instance variables on lines 31-37 declare the API buffers that are used to collect information from the device. The internal methods from lines 40-49 each collect certain pieces of information from the device. Finally we have the same public interface as the simple fake adapter, except we require one configuration file (line 52) that is used to initialize the API.

5.2.2. Adapter Constructor

emc_adapter.cpp

```
1.  EmcAdapter::EmcAdapter(int aPort, const char *aNmlFile)
2.    : Adapter(aPort),
3.      mAlarm("alarm"), mPower("power"), mExecution("execution"),
       mLine("line"),
4.      mXact("Xact"), mYact("Yact"), mZact("Zact"),
5.      mXcom("Xcom"), mYcom("Ycom"), mZcom("Zcom"),
6.      mSpindleSpeed("spindle_speed"), mPathFeedrate("path_feedrate"),
7.      mProgram("program"), mMode("mode")
8.  {
9.    addDatum(mAlarm);
10.   addDatum(mPower);
11.   addDatum(mExecution);
12.   addDatum(mLine);
13.   addDatum(mXact);
14.   addDatum(mYact);
15.   addDatum(mZact);
16.   addDatum(mXcom);
17.   addDatum(mYcom);
18.   addDatum(mZcom);
19.   addDatum(mSpindleSpeed);
20.   addDatum(mPathFeedrate);
21.   addDatum(mProgram);
22.   addDatum(mMode);
23.
24.   mErrorString[0] = 0;
25.   mEmcErrorBuffer = mEmcStatusBuffer = 0;
26.   strcpy(mNmlFile, aNmlFile);
27.   mConnected = false;
28. }
```

We follow the same process of constructing all the `DeviceDatum` objects and giving them names on lines 3-7. The objects are then added to the adapters list of objects on lines 9-27. Currently we support up to 128 individual pieces of data that can be support. If this is insufficient, the constant `MAX_DEVICE_DATA = 128;` in *adapter.hpp* can be increased and the framework rebuilt. The API specific instance variables are initialized and the configuration file is set on lines 24-27.

5.2.3. Adapter Data Gathering

```
void EmcAdapter::gatherDeviceData()
{
  if (!mConnected)
  {
    if (!connect())
      sleep(5);
  }
  else
  {
    if (!mEmcStatusBuffer->valid())
    {
      disconnect();
      return;
    }

    if(mEmcStatusBuffer->peek() == EMC_STAT_TYPE) {
      memcpy(&mEmcStatus, mEmcStatusBuffer->get_address(),
   sizeof(EMC_STAT));
      actual();
      commanded();
      spindle();
      program();
      machine();
      execution();
      alarms();
    }
  }
27. }
```

The `gatherDeviceData()` is the main entry point for all subclasses of the Adapter. The flow is as follows: check if we are currently connected, if not, try to connect. If we cannot connect, sleep for five-seconds and then retry (lines 3-7). If we are connected then make sure the connection is valid and if it is not, disconnect (lines 10-14).

The codes from lines 16-25 constitute the core of the adapter. Line 17 does the actual data gathering by copying the status buffer from the devices shared memory to local application memory. Lines 18-24 gather specific pieces of information.

5.2.4. Adapter Connecting to Device

```
bool EmcAdapter::connect()
{
  int retval = 0;
  if (mEmcStatusBuffer == 0)
  {
    mEmcStatusBuffer = new RCS_STAT_CHANNEL(emcFormat, "emcStatus", "adapter", mNmlFile);
    if (! mEmcStatusBuffer->valid() || EMC_STAT_TYPE != mEmcStatusBuffer->peek())
    {
      rcs_print_error("emcStatus buffer not available\n");
      delete mEmcStatusBuffer;

      mEmcStatusBuffer = 0;
      mPower.setValue(Power::eOFF);
      return false;
    }
  }

  if (mEmcErrorBuffer == 0)
  {
    mEmcErrorBuffer = new NML(nmlErrorFormat, "emcError", "adapter", mNmlFile);
    if (!mEmcErrorBuffer->valid())
    {
      rcs_print_error("emcError buffer not available\n");
      delete mEmcErrorBuffer;
      mEmcErrorBuffer = 0;
      mPower.setValue(Power::eOFF);
      return false;
    }
  }
  mConnected = true;
  return true;
}
```

The previous code connects to the status and error interface of EMC2. There is an additional interface for commands, but since we are read-only, we don't use it. The connect method returns true if the connection is successful, false otherwise.

5.2.5. Adapter Disconnecting from the Device

```
void EmcAdapter::disconnect()
{
  if (mConnected)
  {
    if (mEmcErrorBuffer)
      delete mEmcErrorBuffer;
    mEmcErrorBuffer = 0;

    if (mEmcStatusBuffer)
      delete mEmcStatusBuffer;
    mEmcStatusBuffer = 0;
    mConnected = false;
  }
}
```

The disconnect method safely disconnects from the controller and frees the buffers.

5.2.6. Getting Actual Positions of the Axes

```
void EmcAdapter::actual()
{
  EmcPose &pose = mEmcStatus.motion.traj.actualPosition;
  mXact.setValue(pose.tran.x);
  mYact.setValue(pose.tran.y);
  mZact.setValue(pose.tran.z);
}
```

Once we are connected we can gather the data from the controller. Here is a simple method to get the actual positions and set the values for the X, Y, and Z axes. We don't have to check for changes since the underlying `DeviceDatum` object will do it for us.

5.2.7. Getting the Machine State

```
void EmcAdapter::machine()
{
  if (mEmcStatus.task.state == EMC_TASK_STATE_ON)
  {
    if (mPower.setValue(Power::eON))
    {
      Alarm warn("alarm");
      warn.setValue(Alarm::eMESSAGE, "ON", Alarm::eINFO, Alarm::eINSTANT,
   "Power ON");
      sendDatum(&warn);
    }
  }
  else
  {
    if (mPower.setValue(Power::eOFF))
    {
      Alarm warn("alarm");
      warn.setValue(Alarm:: eMESSAGE, "OFF", Alarm:: eINFO,
   Alarm::eINSTANT, "Power OFF");
18.       sendDatum(&warn);
19.     }
20.   }
21. }
```

This method is used to determine if the machine is on or off. We have added an additional Alarm datum to illustrate sending a one-off datum. Lines 7-9 and 16-18 demonstrate how to use the `sendDatum` method to deliver flush the current buffer and send the alarm. This technique can be used to send any datum immediately.

5.2.8. Getting the Execution State of the Controller

```
void EmcAdapter::execution()
{
  if (mEmcStatus.task.mode == EMC_TASK_MODE_MANUAL)
    mMode.setValue(ControllerMode::eMANUAL);
  else if (mEmcStatus.task.mode == EMC_TASK_MODE_AUTO)
    mMode.setValue(ControllerMode::eAUTOMATIC);
  else if (mEmcStatus.task.mode == EMC_TASK_MODE_MDI)
    mMode.setValue(ControllerMode::eMANUAL_DATA_INPUT);

  if (mEmcStatus.task.interpState == EMC_TASK_INTERP_PAUSED)
    mExecution.setValue(Execution::ePAUSED);
  else if (mEmcStatus.task.interpState == EMC_TASK_INTERP_IDLE)
    mExecution.setValue(Execution::eIDLE);
  else if (mEmcStatus.task.interpState == EMC_TASK_INTERP_WAITING)
    mExecution.setValue(Execution::eEXECUTING);
}
```

The above codes determine the controller mode and the execution state. All controlled vocabularies, like `Execution` and `ControllerMode`, that have a limited set of possible values use an enumeration to prevent invalid values being sent.

5.2.9. Adapter Destructor

```
EmcAdapter::~EmcAdapter()
{
  disconnect();
}
```

The destructor should disconnect from the API and properly clean up all resources being used. We use the same `disconnect()` method to perform cleanup as we did when we detected the the API was no longer valid.

The rest of the data collection follows the same pattern. The rest of the source for the emc adapter is in the emc subdirectory. We will be providing more example adapters as we implement additional vendor APIs and have rights to redistribute.

6. Framework Internals

6.1. Overview

6.1.1. Device Data

We provide an abstraction for `DeviceData` to enforce a minimum structure and set of behaviors for all values we would like to report on. Subclasses of the `DeviceDatum` object detects changes to values and sets the changed flag. All current supported events and samples are provided in the supplied framework.

6.1.1.1. Device Datum Class

```
class DeviceDatum {
protected:
  char mName[NAME_LEN];
  bool mChanged;
  bool mHasValue;

public:
  DeviceDatum(const char *aName);
  virtual ~DeviceDatum();

  bool changed() { return mChanged; }
  void reset() { mChanged = false; }

  char *getName() { return mName; }
  virtual char *toString(char *aBuffer, int aMaxLen) = 0;
  virtual bool append(StringBuffer &aBuffer);
  virtual bool hasInitialValue();
  virtual bool requiresFlush();
};
```

The `DeviceDatum` only requires the constructor and the `toString` method be defined. Since the purpose of the `DeviceDatum` object is to manage a value, all subclasses define a `setValue` method and a typed `mValue`.

6.1.1.2. Event Class

```
class Event : public DeviceDatum
{
protected:
  char mValue[EVENT_VALUE_LEN];

public:
  Event(const char *aName);
  bool setValue(const char *aValue);
  virtual char *toString(char *aBuffer, int aMaxLen);
};
```

The `Event` class defines a datum that has a text value. All events, except `Alarm`, are subclassed from Event. There are special classes for most `Event` types that require a controlled set of values.

6.1.1.3. Sample Class

```
class Sample : public DeviceDatum
{
protected:
  double mValue;

public:
  Sample(const char *aName);
  bool setValue(double aValue);
  virtual char *toString(char *aBuffer, int aMaxLen);
};
```

Similarly, a `Sample` has a double value. This class is used for all samples; for example: position, speed, feedrate.

6.1.2. Tracking Changes

Every time a value is set, it is compared with the previous value. If it has changed then the value is set and the `mChanged` flag is set. This will flag the datum to be sent after all the data is gathered.

```
bool Sample::setValue(double aValue)
{
  if (aValue != mValue)
  {
      mChanged = true;
      mValue = aValue;
  }
  mHasValue = true;
  return mChanged;
}
```

6.1.3. Main data gather loops

Clients will connect to the adapter using an agreed upon port number, currently defaulting to 7878. The adapter is capable of allowing multiple clients to connect at the same time. The adapter does not attempt to connect to the device until at least once client has connected. If you test the adapter, it will appear idle (only polling for connections). This ensures that there will be no additional overhead when using the adapter and the agent is not connected.

```
void Adapter::startServer()
{
  mServer = new Server(mPort);

  /* Process forever... */
  while (1)
  {
```

```
8.      /* Check if we have any new clients */
9.      Client **clients = mServer->connectToClients();
10.     if (clients != 0)
11.     {
12.       for (int i = 0; clients[i] != 0; i++)
13.       {
14.         /* If there are any new clients, send them the initial values for
      all the
15.          * data values */
16.         sendInitialData(clients[i]);
17.       }
18.     }
19.
20.     /* Read and discard all data from the clients */
21.     mServer->readFromClients();
22.
23.     /* Don't bother getting data if we don't have anyone to read it */
24.     if (mServer->numClients() > 0)
25.     {
26.       mBuffer.timestamp();
27.       gatherDeviceData();
28.       sendChangedData();
29.       mBuffer.reset();
30.     }
31.
32.     sleepMs(mScanDelay);
33.   }
34. }
```

We check for new client connections on line 9 and then send the initial values if a new client has connected. We only gather data from the clients if any are present. The data distribution process is as follows: We timestamp the buffer; we gather data; we send the changed data; and then we repeat.

Line 27 is the code the adapter write must supply for every device. The method is responsible for initializing the connection, if it is not already connected, and making the appropriate API calls to retrieve the data.

The sendChangedData method will automatically send any data that remains in the data buffer. If no data is present, it will do nothing. The gatherDeviceData method can also send one-off datum objects. One example is Alarms where multiple instances need to be sent within a single cycle. A one-off send will force the existing data in the buffer to be sent and the reset the buffer. There is an example of this behavior in the following adapter example code.

7. Sample Data

7.1. Samples and Events

```
1.  2008-04-20T18:28:18.687603|power|ON|execution|EXECUTING|Xact|1.3549591303|
    Yact|0.2879030704|Zact|-0.1000000015|Xcom|1.3561842845|Ycom|0.2824387292|
    Zcom|-0.1000000000|spindle_speed|3400.0000000000|program|/home/mtconnect/
    emc2/nc_files/examples/spiral.ngc|mode|AUTOMATIC
2.  2008-04-20 18:28:18.797576|Xact|1.3640016317|Yact|0.2413364202|Xcom|
    1.3647230322|Ycom|0.2353807124
3.  2008-04-20 18:28:18.907572|Xact|1.3692923784|Yact|0.1976564378|Xcom|
    1.3700138636|Ycom|0.1916564378
4.  2008-04-20T18:30:20.927574|Zact|0.7356040478|Zcom|0.7536040307
5.  2008-04-20T18:30:21.037569|Zact|0.8676040173|Zcom|0.8856040307
6.  2008-04-20T18:30:21.147571|Zact|0.9815723896|Zcom|0.9896790496
7.  2008-04-20T18:30:21.307639|execution|IDLE|Zact|1.0000000000|Zcom|
    1.0000000000|spindle_speed|0.0000000000
```

The prior samples include both samples and events. As can be seen on the first line the initial state of the device is provided for all data items that are being monitored. As each value changes, line is emitted with the new key–value pairs. All key value pairs on the same line share the same timestamp.

The format of the sample and event data is as follows:

```
<time_stamp>|<data_item_1>|<value_1>|<data_item_2>|<value_2>|…
```

The timestamp must be provided in ISO 8601 format:

```
    YYYY-MM-DDThh:mm:ss.ffff
```

Where YYYY is the year with century, MM is a two digit month number, DD is the two digit day of the month, hh is the hours using 24 hour clock, mm is the minutes, ss is the seconds, and ffff is the fractional seconds. The fractional seconds can be between 2 to six digits depending on what is available.

7.2. Alarms

Each alarm must be placed on a separate line. Since alarms do not consist of just a single key value pair, the additional information will be parsed separately. Messages should also be encoded as alarms of type `MESSAGE` with a severity of `INFO` and `INSTANT` state.

```
1.  2008-04-20T18:34:51.780917|alarm|MESSAGE|OFF|INFO|INSTANT|Power OFF
2.  2008-04-20T18:34:51.780917|alarm|ESTOP|ESTOP|CRITICAL|ACTIVE|ESTOP Pressed
3.  2008-04-20T18:34:53.314707|alarm|ESTOP|ESTOP|CRITICAL|CLEARED|ESTOP Reset
4.  2008-04-20T18:34:54.437511|alarm|MESSAGE|ON|INFO|INSTANT|Power ON
```

The format of an alarm is as follows:

```
<time_stamp>|<data_item>|<code>|<nativeCode>|<severity>|<state>|<description>
```

The first item must match the name or the source of a data item of type `ALARM`. The next value is the `code` for the alarm followed by the `nativeCode`. The severity comes next and must match the MTConnect classification as does the next parameter, `state`. The last piece of information is the description that will continue until the end of the line. See the MTConnect Draft Standard for more on the Alarm vocabulary.

8. Summary

Regardless of the language you use to realize your MTConnectAdapter, you should now have a broad overview of the parts of an adapter, its collaborators, the data structures and methods of each object, and the process flow necessary to obtain and pass on Samples. Your data source API and its implementation may differ, but broad structure and division of responsibilities should be roughly similar to the design presented in section 3.

This is intended to be a living document, subject to continuous improvement and error correction. Use the MTConnect Website to provide feedback, at http://mtconnect.org.