

MRC II State Data Design Documentation

Summary

The key data structure of the real-time system is the **State Data Table**. This is a 2D table that is indexed by “data id” and “time”. This table should contain all persistent (global) data in the real-time system. The “time” indexing provides a snapshot of the history of the real-time system and can be used for data collection as well as for debugging (i.e., to satisfy the customer requirement for a “flight data recorder” type of functionality). It will also solve the mutual exclusion problem between the real-time and non-real-time parts of the system (similar to a double-buffering technique). The State Data Table is a fixed size, so the time entries are implemented as a circular buffer.

It is assumed that all state data has a single writer and potentially multiple readers. Most of the state data will be written by the real-time process (Trajectory Control Loop), but some user-defined state data may be written by an external process via the non-real-time API. All state data can be read by the real-time and non-real-time processes.

This document first presents the some basic types (mrcObject, mrcStateDataId and mrcTimeIndex). Then, it presents some sample state data accesses, followed by the class definitions.

MRC Base Object (mrcObject)

We first define an mrcObject abstract base class that provides serialization support. All state data classes must derive from mrcObject. Other (non-state) classes may derive from it as well. This class serves a similar purpose to the CObject base class provided by the Microsoft Foundation Class (MFC).

```
class mrcObject
{
public:
    mrcObject();
    virtual ~mrcObject();

    // Methods for serializing data
    virtual void Serialize(mrcBuffer& buf) const = 0;
    virtual void DeSerialize(const mrcBuffer& buf) = 0;

    // Methods for streaming data
    virtual void StrOut(ostream& out) const = 0;
    virtual void StrIn(istream& in) = 0;

    // Friend functions
    friend ostream& operator << (ostream& out,
        const mrcObject& data); // calls StrOut
```

```

        friend istream& operator >> (istream& in,
                                     mrcObject& data); // calls StrIn
};

```

Note that the StrOut and StrIn virtual functions are declared for formatting and parsing data. The base class declares two friend functions that overload operators << and >> for stream output and input, respectively. Derived classes will stream properly if they define the appropriate StrOut and StrIn virtual functions (this is referred to as the “Virtual Friend Function” idiom).

In addition, this class provides Serialize and DeSerialize virtual functions. The StrOut and StrIn functions are intended to produce human-readable (text) output. This can be used for serialization/deserialization, but the Serialize/DeSerialize functions were added for additional flexibility (e.g., so that we can serialize/deserialize in binary formats). These functions take a reference to an mrcBuffer object (think of this as a byte array). We could overload operators << and >> for the mrcBuffer class, using the “Virtual Friend Function” idiom once again.

State Data ID (mrcStateDataId)

Each state data item shall be assigned a unique ID. For “built-in” state data, this is provided by an enum:

```

enum MRC_STATE_DATA_ID {
    CMD_POS_J,           // commanded joint position
    POS1_J,              // joint pos from primary sensor
    POS2_J,              // joint pos from redundant sensor (if present)
    ...
    NUM_STATE_DATA
};

```

User-defined state data can be assigned numbers starting at NUM_STATE_DATA when they are registered (see *MRC II Software Extensibility Design Document*).

This enum could be a public member of a class for name scope restriction. We also define an mrcStateDataId data type:

```

typedef int mrcStateDataId;

```

State Time Index (mrcTimeIndex)

The time indexing of state data is provided by the mrcTimeIndex class. Each iteration of the Trajectory Control Loop increments a tick counter, which is stored as an unsigned long (mrcTimeTicks). An mrcTimeIndex object contains the following information:

- The index into the set of circular buffers corresponding to the time
- The tick value corresponding to the time

The tick value is stored in the `mrcTimeIndex` class to act as a data validity check. This is used to ensure that the circular buffer has not wrapped around and overwritten the index entry between the time that the `mrcTimeIndex` object was created and the time it was used.

```
typedef unsigned long mrcTimeTicks;

class mrcTimeIndex : public mrcObject {

    int index; // array index
    mrcTimeTicks ticks; // corresponding State Time (for checking)
    int len; // array length (could instead be a global constant)

public:
    mrcTimeIndex() : index(0), ticks(0), len(0) { }
    mrcTimeIndex(int _index, mrcTimeTicks _ticks, int _len) :
        index(_index), ticks(_ticks), len(_len) { }
    virtual ~mrcTimeIndex() { }

    int Index() const { return index; }
    mrcTimeTicks Ticks() const { return ticks; }

    mrcTimeIndex& operator --()
        { ticks--; index--; if(index < 0) index=len-1; return *this; }
    mrcTimeIndex operator --(int)
        { mrcTimeIndex tmp = *this; --(*this); return tmp; }

    mrcTimeIndex& operator --(int n)
        { ticks -= n; index = (index-n)%len; return *this; }
    mrcTimeIndex operator -(int n) const
        { mrcTimeIndex tmp = *this; return (tmp -= n); }

    bool operator ==(const mrcTimeIndex& right) const
        { return (index == right.index)&&(ticks == right.ticks); }
    bool operator !=(const mrcTimeIndex& right) const
        { return !(*this == right); }

    // Methods for serializing data
    virtual void Serialize(mrcBuffer& buf) const;
    virtual void DeSerialize(const mrcBuffer& buf);

    // Methods for streaming data
    virtual void StrOut(ostream& out) const;
    virtual void StrIn(istream& in);
};
```

For improved efficiency, the mod operator (%) can be replaced by (&) if `len` is a power is two. For example, `(index&0x00ff)` performs the mod operation if `len` is 256. The `++`, `+=` and `+` operators have been omitted because we do not want to allow future times.

Note that we overload many of the same operators for `mrcTimeIndex` as we would for a random access iterator (e.g., that would be used with an STL vector). The fundamental difference is that we are using array indexing (e.g., `value[i]`) instead of pointers (e.g., `*value--`) for reasons that will be presented later.

State Data Access from Local/Remote Applications

We assume that the robot system API is provided by an `mrcRobot` class, and that there will be `mrcRobotLocal` and `mrcRobotRemote` derived classes (we may not need an `mrcRobotRemote` class, depending on how we handle the distributed computing).

As an example, we show how the commanded joint position would be retrieved from the local or remote application. For simplicity, we assume that we are only interested in retrieving the latest value. These functions will be generalized to include a “time” (`mrcTimeIndex`) parameter. This will allow the application to retrieve older data and to retrieve a time-synchronized data set (for example, to guarantee that the commanded joint position and the measured joint position were sampled at the same time).

```
// Get the commanded joint position (local)
int mrcRobotLocal::GetCmdPosJ(mrcJointPos& jpos)
{
    mrcStateTable* state = mrcStateTable::Instance();
    mrcTimeIndex time = state->GetReadIndex();
    return state->Read(CMD_POS_J, time, jpos);
}
```

The above code assumes that we have an `mrcStateTable` Singleton object that contains `GetReadIndex` and `Read` member functions. Although not evident above, it is also assumed that `mrcJointPos` is derived from `mrcObject`.

Following is an implementation of the remote version of this function. This implementation assumes that messages will be sent via a network object “nw” that has `Send` and `Recv` member functions.

```
// Get the commanded joint position (remote)
int mrcRobotRemote::GetCmdPosJ(mrcJointPos& jpos)
{
    // nw, cmd and reply to be declared
    cmd.code = MRC_STATE_QUERY;
    cmd.id = CMD_POS_J;
    nw->Send(cmd);
    nw->Recv(reply);
    int rc = reply.retcode;
    if (rc == MRC_OK)
        rc = jpos.DeSerialize(reply.data);
    return rc;
}
```

The remote version of this function requires a servicing function in the local object. Note, however, that all remote state queries can be handled by the same code in this function.

```
int mrcRobotLocal::ServiceRemoteCmds(void)
{
    // nw, cmd and reply to be declared
```

```

nw->Recv(cmd);
if (cmd.code == MRC_STATE_QUERY) {
    mrcStateTable* state = mrcStateTable::Instance();
    mrcTimeIndex time = state->GetReadIndex();
    reply.retcode = state->Serialize(cmd.id, time, reply.data);
}
else {
    // process other types of commands
}
nw->Send(reply);
}

```

State Data Access from Real Time Loop

The Real Time Loop reads from and writes to the State Data Table. It first calls `mrcStateTable::StartRT` to indicate that all subsequent data accesses (until the call to `mrcStateTable::EndRT`) will be made from the real-time process. The call to `EndRT` also makes the most recent set of data available to the non-real-time process.

```

void rt_loop()
{
    mrcStateTable* state = mrcStateTable::Instance();

    state->StartRT();
    mrcTimeIndex time = state->GetReadIndex();

    // Read the encoder position and store in State Table
    state->Write(POS1_J, read_encoders());

    // Compute the next joint goal and store in State Table
    // (Simple example: next joint goal obtained by adding delta)
    state->Read(CMD_POS_J, time-1, cmd_pos);
    cmd_pos += delta;
    state->Write(CMD_POS_J, cmd_pos);

    state->EndRT();
}

```

State Table Class (mrcStateTable)

The above examples illustrated the interface requirements for the State Data Table, so we are ready to describe its design. For now, we assume that we have a set of “state data array” objects, each of which is derived from an `mrcStateDataArray` abstract base class. For example, we will have an array of “commanded joint position” objects (`CMD_POS_J`) and an array of “measured primary joint position” objects (`POS1_J`).

The time history is provided by implementing a (producer/consumer) circular buffer in each of these arrays. We desire the following two properties:

1. All state data arrays should be the same length (e.g., `HISTORY_LEN`).
2. The producer and consumer “pointers” (indices) should be the same across all buffers.

Note that the STL vector class does not provide any benefits for the state data arrays because it does not support a circular buffer and furthermore, even if a circular vector and iterator were developed, one iterator object would not work across all state data objects. This exemplifies a fundamental difference between accessing an array by index (e.g., `value[i]`) and by pointer (e.g., `*value++`). In the first case, the index (`i`) can be used for many different data arrays, whereas in the second case a different pointer is needed for each array. Because STL iterators are a generalization of pointers, they are not well suited for this application. The STL vector class is useful, however, to maintain the list of state data arrays, especially because we may wish to change the array size at runtime to include user-defined state data.

Here is the `mrcStateTable` class, which is a standard Singleton object. Implementations of the member functions shall be presented later.

```
class mrcStateTable {
    vector<mrcStateDataArray*> statevec;

    vector<bool> rt_valid;
    mrcTimeTicks ticks[HISTORY_LEN];

    bool rt_active; // whether real-time loop is currently active
    int rt_index; // index used by real-time loop
    int bg_index; // index used by background

    static mrcStateTable* _instance;

protected:
    mrcStateTable() : statevec(NUM_STATE_DATA),
                    rt_valid(NUM_STATE_DATA) { }

public:
    enum { HISTORY_LEN = 256 }

    ~mrcStateTable();

    static mrcStateTable* Instance();

    // Called during system initialization
    int AddElement(mrcStateDataId id, const mrcStateDataArray& prot);

    // Gets handle for reading data
    mrcTimeIndex GetReadIndex(void) const;

    // Verifies that data is still valid
    bool ValidateReadIndex(const mrcTimeIndex& time) const;

    // Read specified data
    bool Read(mrcStateDataId id, const mrcTimeIndex& time,
             mrcObject& data) const;
};
```

```

// Write specified data
bool Write(mrcStateDataId id, const mrcObject& data);

// Serialize specified data
bool Serialize(mrcStateDataId id, const mrcTimeIndex& time,
              mrcBuffer& buffer) const;

// Called at start of real-time loop
void StartRT(void) { rt_active = true; }

// Advance circular buffer (called during initialization and
// at end of each real-time loop)
void Advance(void);

// Called at end of real-time loop
void EndRT(void) { Advance(); rt_active = false; }
};

```

Note that the Read member function takes a time parameter but the Write member function does not for the following reasons:

- To preserve the “flight data recorder” feature, we should not allow the code to modify the past.
- To ensure mutual exclusion between the real-time and non-real-time code, we must restrict modification of data.

The `rt_valid` data member will be used to track whether each state data object has been updated – it could be part of the `mrcStateDataArray` object instead of being part of this class.

State Data Array Base Class (mrcStateDataArray)

We now define the abstract base class for state data arrays.

```

class mrcStateDataArray
{
protected:
    mrcStateDataArray();

public:
    virtual ~mrcStateDataArray();

    // Overloaded subscript operator
    virtual mrcObject& operator[](int num) = 0;
    virtual const mrcObject& operator[](int num) const = 0;

    // Create the array of data
    virtual mrcStateDataArray* Create(int size) const = 0;

    // Copy data from one index to another

```

```

virtual void Copy(int to, int from) = 0;

// Get data from array
virtual bool Get(int index, mrcObject& data) const = 0;

// Set data in array
virtual bool Set(int index, const mrcObject& data) = 0;

};

```

State Data Template Array Class (mrcStateTempArray)

Individual state data classes can be created from an instance of the following template, where Data represents the type of data used by the particular state element. It is assumed that Data is derived from mrcObject. This template class is derived from the mrcStateData Array abstract base class.

```

template <class Data >
class mrcStateTempArray : public mrcStateDataArray {

protected:

    Data* data;

public:

    mrcStateTempArray() : data(0) { }
    mrcStateTempArray(int size) : data(new Data[size]) { }
    virtual ~mrcStateTempArray() { delete [] data; }

    // Create the array of data
    virtual mrcStateDataArray* Create(int size) const
        { return new mrcStateTempArray<Data>(size); }

    // Overloaded subscript operator
    virtual mrcObject& operator[](int num)
        { return data[num]; }
    virtual const mrcObject& operator[](int num) const
        { return data[num]; }

    // Copy data from one index to another
    virtual void Copy(int to, int from)
        { data[to] = data[from]; }

    // Get data from array
    virtual bool Get(int index, mrcObject& obj) const;

    // Set data in array
    virtual bool Set(int index, const mrcObject& obj);

};

```

The array is declared as a Data* instead of a vector<Data> (e.g., using the STL vector class) because it will be used for a circular buffer, as described earlier.

The Get and Set member functions deserve special mention because they must overcome a limitation of C++ – namely, that it does not fully support containers of heterogeneous objects. In particular, we expect the “obj” parameter to be of type `Data&` (the derived class) rather than `mrcObject&` (the base class). This can be handled using C++ Run Time Type Information (RTTI) features such as `dynamic_cast` and `typeid`. Following is an implementation of these functions using `dynamic_cast`:

```
template <class Data>
bool mrcStateTempArray<Data>::Get(int index, mrcObject& obj) const
{
    Data* ptr = dynamic_cast<Data*>(&obj);
    if (ptr) {
        *ptr = data[index];
        return true;
    }
    return false;
}

template <class Data>
bool mrcStateTempArray<Data>::Set(int index, const mrcObject& obj)
{
    const Data* ptr = dynamic_cast<const Data*>(&obj);
    if (ptr) {
        data[index] = *ptr;
        return true;
    }
    return false;
}
```

The use of `dynamic_cast` in a real-time system is a little troubling because it is not guaranteed to be a constant time operation. In general, `dynamic_cast` has to traverse the derivation tree of `obj` until it finds the target object (`Data`). In our case, unless there is a programming error this should be constant time because `obj` should already be the same type as the target.

If the compiler does not support RTTI (or if we like taking risks to obtain efficiency), we can replace the `dynamic_cast` with a C style cast. Moving even further into the C realm, we can use a `void*` to the object and eliminate the requirement for deriving all state data from `mrcObject` (of course, we would still need a solution for serialization).

Alternatives to RTTI

We can maintain an object-oriented design and avoid using RTTI, but the design is much less clean. One solution that was investigated is to make each state data array object a Singleton object that can be accessed either directly or via the State Data Table. For example, the `mrcRobotLocal::GetCmdPosJ` function presented earlier would use the statement:

```
return CmdPos1J::Instance()->Read(time, jpos);
```

instead of:

```
return state->Read(CMD_POS_J, time, jpos);
```

In this implementation, the Get and Set member functions would be part of the mrcStateTempArray template class instead of the mrcStateDataArray base class, so they would be declared to accept a Data& instead of an mrcObject&:

```
virtual bool Get(int index, Data& obj) const;  
virtual bool Set(int index, const Data& obj);
```

This implementation was rejected for the following two reasons:

1. The design is less clean when the state data can be accessed in more than one way.
2. This implementation requires a more complex interaction between the classes because the mrcStateTable object handles the circular buffer indexing (time) but the various mrcStateTempArray objects handle the data manipulation.

State Table Initialization

The state table is initialized by calling its AddElement member function for each state data array. The AddElement function calls mrcStateDataArray::Create. For example, the following code would be used to initialize the state table to contain CMD_POS_J and POS1_J state arrays:

```
MrcStateTable* state = mrcStateTable::Instance();  
state->AddElement(CMD_POS_J, mrcStateTempArray<mrcJointPos>());  
state->AddElement(POS1_J, mrcStateTempArray<mrcJointPos>());  
// Add other state elements here  
state->Advance();
```

Following is a simple implementation of AddElement that can handle user-defined state data:

```
int mrcStateTable::AddElement(mrcStateDataId id,  
    const mrcStateDataArray& prototype)  
{  
    if ((id < 0) || (id >= statevec.size())) {  
        // adding user-defined data  
        statevec.push_back(prototype.Create(HISTORY_LEN));  
        rt_valid.push_back(true);  
        id = statevec.size()-1;  
    }  
    else {  
        if (statevec[id]) // could treat as an error  
            delete [] statevec[id];  
        statevec[id] = prototype.Create(HISTORY_LEN);  
        rt_valid[id] = true;  
    }  
    return id;  
}
```

State Table Implementation

The GetReadIndex function can be implemented as follows. Note that mutual exclusion is handled in a simplistic (but effective) manner. The GetReadIndex function returns `rt_index` if called from the real-time loop; otherwise, it returns `bg_index`. We do not need an explicit critical section because we assume that we can read `bg_index` atomically (i.e., in a single machine instruction that cannot be interrupted during execution).

```
mrcTimeIndex mrcStateTable::GetReadIndex(void) const
{
    if (rt_active)
        return mrcTimeIndex(rt_index,ticks[rt_index],HISTORY_LEN);
    else {
        int tmp = bg_index; // assume atomic operation
        return mrcTimeIndex(tmp, ticks[tmp], HISTORY_LEN);
    }
}
```

The ValidateReadIndex function verifies that the data corresponding to the specified time is still valid (i.e., it has not been overwritten in the circular buffer). This is done by checking that the ticks value maintained by the State Table (`ticks[time.Index()]`) still matches the ticks value maintained in the `mrcTimeIndex` object (`time.Ticks()`).

```
bool mrcStateTable::ValidateReadIndex(const mrcTimeIndex& time) const
{
    return (ticks[time.Index()] == time.Ticks());
}
```

The Read function can be implemented as follows. Note that there are minor, but important, implementation differences when accessing data from the real-time loop vs. the non-real-time process. In the first case, we check the validity of the data before we read it – this saves us the overhead of copying invalid data. When accessing data from the non-real-time process, it is important to check the data validity after reading it to ensure that we read valid data (if we checked the data validity first, the real-time loop could interrupt and overwrite the data between the validity check and the read).

```
bool mrcStateTable::Read(mrcStateDataId id, const mrcTimeIndex& time,
    mrcObject& data) const
{
    if (rt_active) {
        // if we are trying to read the latest data before it is
        // available, return false
        if ((time.Index() == rt_index) && !rt_valid[id])
            return false;
        bool rc = ValidateReadIndex(time);
        if (rc)
            statevec[id]->Get(time.Index(), data);
        return rc;
    }
    else {
```

```

        statevec[id]->Get(time.Index(), data);
        return ValidateReadIndex(time);
    }
}

```

The Write function can be implemented as follows. We only write to one index (rt_index), which makes mutual exclusion a little easier. On the other hand, by supporting external state data updates we make mutual exclusion a little harder because the non-real-time software must also be able to write to the State Table. The following code does not specify the implementation details for the critical section – some mechanism (such as disabling interrupts) must be used.

```

bool mrcStateTable::Write(mrcStateDataId id, const mrcObject& data)
{
    bool rc;
    if (rt_active) {
        rc = statevec[id]->Set(rt_index, data);
        if (rc)
            rt_valid[id] = true;
    }
    else {
        // Start critical section
        rc = statevec[id]->Set(rt_index, data);
        if (rc)
            rt_valid[id] = true;
        // End critical section
    }
    return rc;
}

```

The Serialize member function just calls the Serialize function for the specified state object. This is very straightforward because Serialize is a member of the mrcObject base class. We could implement a mrcStateTable::DeSerialize function if necessary – there does not seem to be any need for it at this time.

```

bool mrcStateTable::Serialize(mrcStateDataId id,
    const mrcTimeIndex& time, mrcBuffer& buffer) const
{
    (*statevec[id])[time.Index()]>Serialize(buffer);
    return ValidateReadIndex(time);
}

```

The Advance member function loops through the state table and ensures that all data entries are valid when advancing the index. It also increments the ticks value.

```

void mrcStateTable::Advance(void)
{
    int i;
    bg_index = rt_index;
    rt_index = (rt_index+1)%HISTORY_LEN;
    ticks[rt_index] = ticks[bg_index]+1;

    for (i = 0; i < statevec.size(); i++) {

```

```

        if (rt_valid[i])
            rt_valid[i] = false;
        else {
            // Handle "missing" data. One option is to just copy
            statevec[i]->Copy(rt_index, bg_index);
        }
    }
}

```

Design Options and Future Enhancements

We could add an “attributes” field for each state data element that would specify certain parameters, such as the following:

- Whether the data is written by the real-time loop (the usual case) or by an external process via the API. This information could be used to streamline mutual exclusion code.
- How to handle missing data (i.e., data that is not updated before the real-time loop exits). We could, for example, silently copy the data from the previous loop or we could raise an error.

The “attributes” field could be part of `mrcStateTable` or `mrcStateDataArray` (or its derived classes).

The `rt_valid` entry is currently part of `mrcStateTable`. It could be moved to `mrcStateDataArray` or one of its derived classes (e.g., `mrcStateTempArray`). Furthermore, it could be changed to an integral type and track the number of “missed” updates. This could allow a small number of “missed” updates to be silently handled but ensure that an error is raised before the data gets too stale.

Many member functions currently return true or false to indicate success or failure. It could be better to return an error code from a global error list.

The `mrcStateDataArray` class could include an `mrcStateDataId` parameter, passed in the constructor. We could then eliminate the extra (and potentially inconsistent) `mrcStateDataId` parameter in some function calls, such as `mrcStateTable::AddElement`.

The `mrcStateDataArray::Create` function is a variation of the Prototype pattern. Given a prototype array object of any given size (including 0), the Create function creates a new array object of the specified size. Currently, the elements of the new array object are initialized using the default constructor – we do not “clone” the prototype object. One alternative would be to initialize the new array using elements from the prototype array – for example, if the prototype array has one element, each element of the new array could be initialized to value of the prototype array element.