# MCM Design Document
# Revision 2.0

**21 May 2001**
**Revised 5 June 2001**

**Cayci Suitt**
**Gene Wie**
**Salvador Ledezma**
**Jimar Garcia**

# 1. Introduction

The Motion Capture Music System (MCM) is a software system that will be developed by the MCM Project Group for the University of California, Irvine (UCI) Dance Department. Its purpose shall be to interpret three-dimensional (3D) motion data and generate music based on the motion.

The project sponsors are Ms. Lisa Naugle, Assistant Professor, and Mr. Christopher Dobrian, Associate Professor, in the Dance and Music Department, respectively, of the UCI Claire Trevor School of the Arts (SOTA). Mr. Dobrian is also a professor in the Information and Computer Science (ICS) Department at UCI.

Currently, the project sponsors work with a Vicon Motion Systems (Vicon) Vicon8 Optical Motion Capture (Vicon8) System located in the School of the Arts Music Media Center. The Vicon8 System resides in the motion capture studio. In this room, there are eight infrared cameras that capture motion data from a predefined area of the room. The capture subject will don 1 to 48 spheres and move about within a cylindrical portion of the 3D space in the room. A circle is marked off on the ground to represent the projection of the 3D cylindrical space. Each of the spheres represents a point in 3D space. The cameras capture the points in motion at a variable rate specified by the user, typically 30 frames per second. At the present time, the motion capture is not used as input for creating new information. It records motion data via the network of cameras, but does not allow the simultaneous generation of a music or visual accompaniment based on the motion.

The project sponsors would like to expand the capabilities of the existing system so that in can be used in new and innovative ways, such as music composition. The MCM will interpret the motion data captured by the network of infrared cameras, and generate music in Musical Instrument Digital Interface (MIDI) format in real-time. This will allow the subject's specific movements to generate different pitches, change the intensity of the music, or change the voice of the instrument. The mapping of specific movements to pitch, tone, volume, and instrument will be customizable and it will allow the composer to load and save different mapping configurations.

Key features of the MCM system include real-time interpretation and a Graphical User Interface (GUI). MCM will translate the 3D motion data based on a user-specified mapping. This translation will be outputted in MIDI data in real-time. The GUI will allow the user to specify a mapping. It will allow the user to customize, save, and load files, which will contain specific mappings of motion to music.

The requirements documents defined the functional and non-functional specifications for the MCM software system and its interaction with the existing Vicon8 System. It is included in this document by reference.

The purpose of this document is to describe the design of the MCM software system. It primarily describes the architectural style of the system, the modules that comprise the system, and how those modules interact with one another.

At a high level, MCM is a two-stage program and therefore is comprised of two separate executable programs, MCMMap and MCMTranslate.

**1.1. MCMMap** accepts data from the user in order to create a mapping of C3D motion data onto MIDI data. The data input program will consist of a GUI through which the user can input, save and/or load his translation choices.

**1.2. MCMTranslate** uses the mapping to perform the specified mapping in real-time. The translation program acquires the data from the network port that connects the Vicon Real-Time Workstation and the translated visual output via reader code. This recorded data is what is translated from the streaming C3D motion data format to MIDI music.

In addition to this introduction, this document details the design specifications within the context of the following sections:

2. Project Plan – The project schedule and resources will be addressed in this section. They have been updated from the plan presented in the Requirements document to reflect changes in the schedule.

3. Design Specification – This section will give a detailed overview of the system design, including an architectural overview and the module specification.

4. Integration Test Plan – In this section, the plan for integration testing will be presented. The Integration Test Plan and its implementation will ensure that the software modules defined in this document are able to interact.

5. Tracking and Control Mechanisms – This section will describe the tracking and control mechanisms implemented specifically for the design phase of the project.

6. Requirements Changes – This section will describe the requirements that have changed in light of the conversations for the sponsor and/or after conducting a technology review.

7. Minutes – Minutes of all the meetings held by the project team in support of the development of this document will be included in this section.

8. Glossary – Technical terms and acronyms presented in this document will be defined in the glossary.

9. Revisions – Revisions to this document will be listed in this section. They will include date of revision, the nature of the revision, the team member that made or requested the revision, and the sections of this document that are affected by the revision.

## 2. Project Plan

The following continuation of the project plan from the Requirements document has been slightly revised given the time frame and actual progress on the system.

| | | |
|---|---|---|
| 5-6 | Architectural Design | All |
| 6 | Produce Design Specifications | All |
| 6 | Develop capture/translation algorithms | All |
| 6 | Prepare Technical Presentation | Gene, Sal |
| 6-7 | Give Technical Presentation, Submit Report | All |
| 7 | Complete Design Specifications | All |
| 7-8 | Implement Mapping Functionality<br>?? User interface<br>?? Disk I/O | <br>Cayci<br>Cayci |
| 7-8 | Implement Translation Functionality<br>?? Vicon Motion Data Stream Reader<br>?? Translator Process<br>?? MIDI Output | <br>Jimar<br>All<br>Gene |
| 8 | Test Mapping Functionality | Cayci |
| 8 | Test Translation<br>?? Unit Testing<br>?? Integration Testing | <br>Gene, Jimar, Sal<br>All |
| 8-9 | Integration Testing (MAP+TRANSLATE) | All |
| 9 | Usability Testing | All |
| 9-10 | Performance Analysis | All |
| 10 | Final Demonstration | All |
| 11 | Maintenance, Revision | All |

# 3. Design Specification

## 3.1 Architectural Overview

The overall MCM system has no specific architectural style; it is simply comprised of two separate programs, MCMMap and MCMTranslate.

MCMMap is defined as a hierarchy of functions. It is composed of the functions that make up the GUI. It also reads and writes user-input to a file.

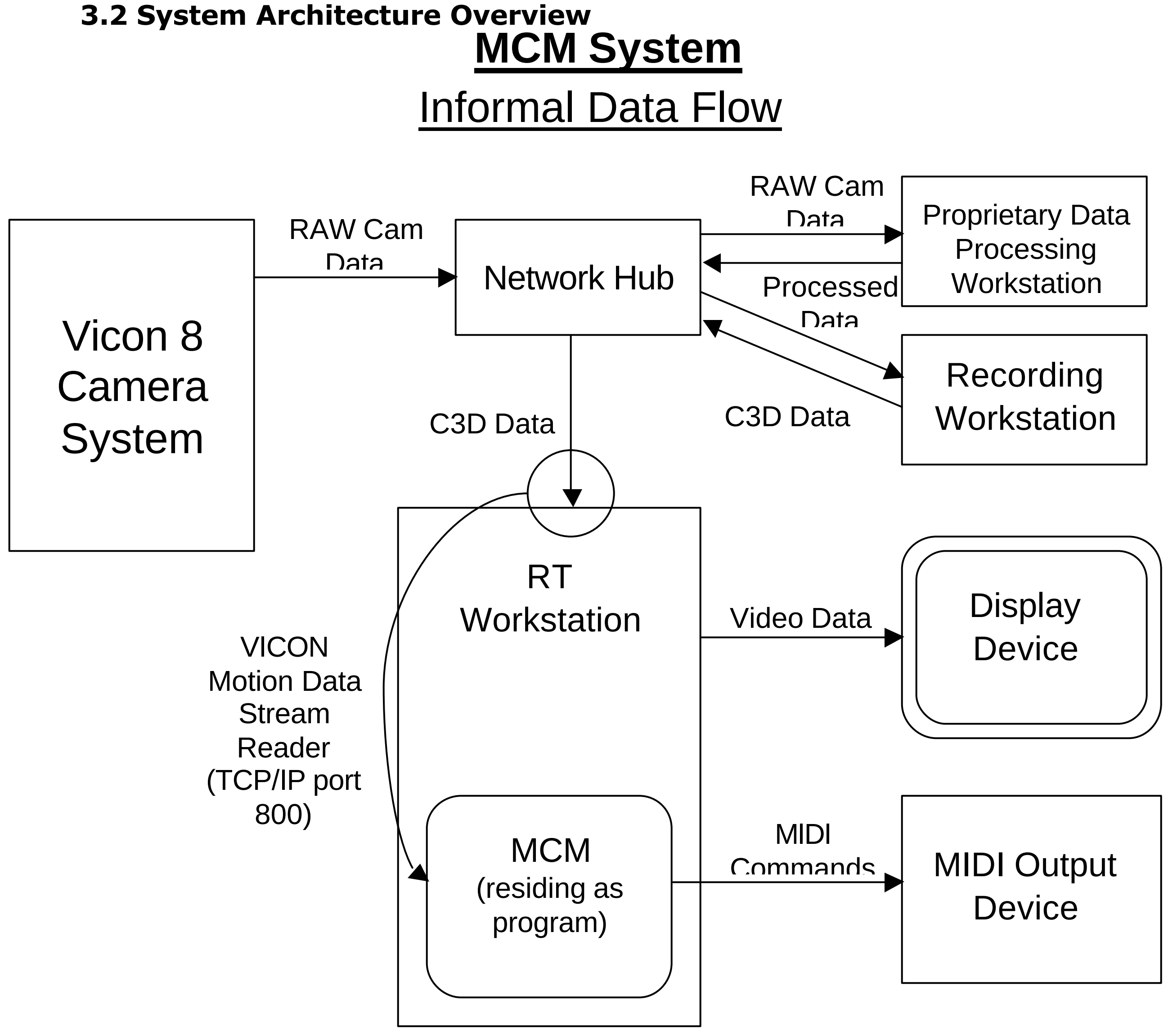
The architectural style for the Translation portion of MCM is Pipe and Filter. The MCMTranslate program/module adheres to this architectural style in that the data is collected, filtered, manipulated and then output again without any other program "knowing" the inner workings of this data translation.

The MCMTranslate needs the MCMMap to function in that it cannot filter data without a mapping, but the reverse is not true. A user is able to create mappings without ever using them to translate 3D motion data into MIDI.

The following table gives and overview of the modules that compose the MCM system, along with their completion status as of the first writing of this document:

| **Module** | **Developer** | **Coded?** | **Unit Tested?** | **Integration Tested?** |
|---|---|---|---|---|
| Map (main) | Cayci | Yes | Yes | No |
| Map::GUI | Cayci | Yes | Yes | Yes |
| Map::IO | Cayci | Yes | Yes | Yes |
| Translate (main) | Gene/Sal | No | No | No |
| Translate::Vicon | Jimar | Yes | No | No |
| Translate::MappingInfo | Sal | No | No | No |
| Translate::MIDI | Gene | Yes | No | No |
| Translate::Translator | J/G/S | No | No | No |

## 3.2 System Architecture Overview



The figure on this page provides a high-level overview of the data flow of the MCM System.

## 3.3 Subsystem Narrative

### 3.3.1 User Interface

The user interface of the Mapping portion of the MCM system is comprised of a GUI interface built in the Java programming language. It provides the user the ability to enter in information that correlates motion data to MIDI commands.

### 3.3.2 Input/Output (MAP)

The Mapping portion of the MCM system, having received user input correlating motion data to MIDI commands, can save or load a specified "mapping" by the user to a file on disk.

#### 3.3.3 Input/Output (TRANSLATE)

The Translation portion of the system, which runs as a standalone executable, loads in the saved mapping on disk into memory.

#### 3.3.4 Storage in Memory

The mapping information is stored in memory as a hash table to allow efficient insertions and searches.

#### 3.3.5 Vicon Motion Data Stream Reader

The Vicon-supplied motion data stream reading code reads the real-time motion data off the network; the data is sent in real time by the Vicon 8 camera system to its recording workstation and intercepted by this subsystem.

#### 3.3.6 Translator

The Translator Portion of the MCM system uses the mapping provided by the Map Portion of the MCM system to translate the motion data provided by the stream reader into corresponding MIDI commands. It is in here that the set of heuristics defined by the mapping is applied to the set of incoming motion data in order to transform it into an equivalent in sound data.

#### 3.3.7 MIDI

Improv 2.3.0 is a library of routines for MIDI command output that MCM uses to generate the sounds. The Translator sends the Improv-based MIDI module the relevant commands it has determined from the heuristics for the mapping to translation for immediate output of the sound to a General MIDI (GM) compatible device. Permission has been granted to use this code for the non-commercial purpose of academic research.

#### 3.3.8 Limitations

Limitations on the current design include:

**3.3.8.1** Full System for testing not present at UCI. Integration testing must be completed on an emulator provided by Vicon

**3.3.8.2** Dependency on external third-party code. Much of the network stream reading capability is defined in code provided by Vicon. In order for implementation to succeed, this code must not have any significant faults.

### 3.4 Module Specification

### 3.4.1 MCMMap

**Purpose:** This is the top-level module that represents the data input portion of the MCM. This module will collect data from the user and save it to a file so that MCMTranslate can use it. The main() will exist in this module and will be the first program the user runs.

**In interface:** none

*// DEFINE: An "in" interface is what others call*

**Out interface:** none

*// DEFINE: An "out" interface is what [the module]*
*// needs from others*

### 3.4.2 MCMMap Modular Design

**<u>MCMMap::GUI</u>**

**Purpose:** This is the graphical user interface used for collecting mapping choices from the user.

**In interface:** none

**Out interface:**

```
String[][] getData(File mapname)
void SendData(String[][] data, File mapping, int rows)
void RowData.addDuration(String d)
```

**<u>MCMMap::IO</u>**

**Purpose:** This class handles IO operations for MCMMap.

**In interface:**

```
String[][] getData(File mapname)
void SendData(String[][] data, File mapping, int rows)
void RowData.addDuration(String d)
```

**Out interface:** none

**<u>MCMMap::RowData</u>**

**Purpose:** This is the data structure format of the file that stores the user-specified mapping in the GUI.

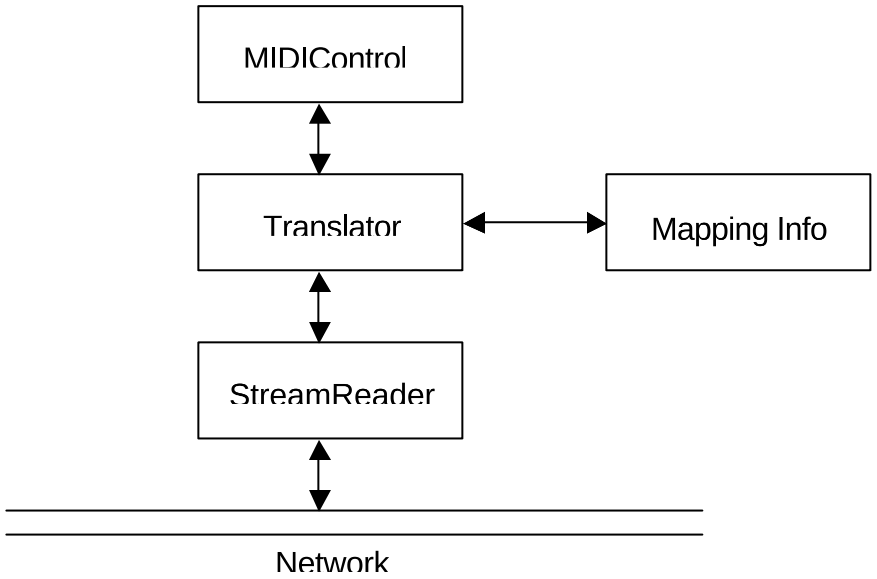
**In interface:**

```
void addDuration(String d)
```

**Out interface:** none

### 3.4.3 MCMTranslate Modular Design

**3.4.3.1 MCMTranslate** is comprised of four sub-components: Stream Reader, Translator, Mapping Info (I/O), and MIDI.



#### 3.4.3.2 Interfaces

Module: MCMTranslate::StreamReader

| **Function Name** | **Return Type** | **Parameters** | **Description** |
|---|---|---|---|
| StreamReader | None | None | Constructor |
| setIPAddress | void | char * *strIPAddy* | Sets internal variable that stores the IP address of the Vicon Real Time system |
| setHostName | void | char * *strHostName* | Sets the internal variable that stores the host name of the Vicon Real Time system |
| connect | bool | None | Connects with the Vicon Real Time system. This function will need threading so we can populate a local cache independently of being polled for reconstruction points. |
| cacheIsEmpty | bool | None | Returns TRUE if |

| | | | |
|---|---|---|---|
| | | | reconstruction point cache is empty. |
| getReconstructionPt | int * | None | Returns an integer array with three elements containing the x, y, and z coordinates of the queued reconstruction point. |
| isConnected | bool | None | Returns TRUE if still connected with the Vicon RT system. |
| queryDisconnect | void | None | Queries the StreamReader object that a disconnection request has been made. It will attempt to disconnect in the next network polling period. |
| getIPAddress | char * | None | Returns the stored IP address of the Vicon RT system. |
| getHostName | char * | None | Returns the stored host name of the Vicon RT system. |

Module: MCMTranslate::MappingInfo

| **Function Name** | **Return Type** | **Parameters** | **Description** |
|---|---|---|---|
| MappingInfo | None | char* *fileName* | Sets the internal variable that stores the absolute path of the mapping file. The size of the hast table is set and the file is open, parsed, and nodes are inserted into the hash table. |
| allocateHTable | void | None | Sets the size of the hash table to the default constant size. It also allocates an int array that tracks the number of items stored in each array slot and initializes it to 0. |
| hash | unsigned int | string *bodyId*, int *tableSize* | The hash() function hashes the bodyId and efficiently finds the location of an item to be inserted or searched into the hash table. It is a standard hash function for strings taken from Weiss, *Algorithms, Data Structures, and Problem Solving with C++*, Addison-Wesley, 1996, p. 611. |

| | | | |
|---|---|---|---|
| insert | void | string *bodyPart*, char *axis*, Node* *n* | Inserts a bodypart/axis combination into the hash table. The body part and axis are concatenated and together they form the key that is hashed. The node is then inserted into the front of the linked list at that location. The number of items at that location is incremented. |
| find | Node* | char* *bodyPart*, char *axis* | Given a bodypart/axis pair, the Node associated with the pair is returned. |
| minRangeMotion | int | char* *bodyPart*, char *axis*, char* *command* | Given a bodypart/axis/midi command triple, the associated minimum physical range limit is returned. |
| maxRangeMotion | int | char* *bodyPart*, char *axis*, char* *command* | Given a bodypart/axis/midi command triple, the associated maximum physical range limit is returned. |
| minRangeMidi | int | char* *bodyPart*, char *axis*, char* *command* | Given a bodypart/axis/midi command triple, the associated minimum Midi range limit is returned. |
| maxRangeMidi | int | char* *bodyPart*, char *axis*, char* *command* | Given a bodypart/axis/midi command triple, the associated maximum Midi range limit is returned. |
| commandType | string* | char* *bodyPart*, char *axis* | Finds and returns all the Midi commands associated with a given bodypart/axis pair. |
| getDuration | string* | char* *bodyPart*, char *axis*, char* *command* | Given a bodypart/axis/midi command triple, the associated duration is returned. If the command is not a note, then the value stored is the string "null". For this reason, the duration is stored internally as a string. |
| exists | bool | char* *bodyPart*, char *axis* | Searches and determines whether there is any Midi command associated with a given bodypart/axis pair. |
| numAtlocation | int | char* *bodyPart*, char *axis* | Determines the number of nodes associated with a given bodypart/axis pair. More specifically, it returns the number of nodes at a particular location in the hash table, which theoretically can be associated with different bodypart/axis pairs. |
| initialize | void | none | The file with the mapping |

|  |  |  |  |
|---|---|---|---|
|  |  |  | information is read and parsed. Nodes are composed of 9 parameters. Based on these parameters, the Nodes for bodypart/axis pairs are created and inserted into the hash table. |

Module: MCMTranslate::MIDI

| **Function Name** | **Return Type** | **Parameters** | **Description** |
|---|---|---|---|
| MIDI output for the MCM system is handled by Improv 2.3.0, a library of C++ classes primarily consisting of generalized MIDI I/O communication functions; the specific class MCM uses in the Improv 2.3.0 system for MIDI output is called **MidiOutput** and has the basic functions MCM will use to output the sound as follows (the use of this library allows MCM to directly send raw MIDI data to a General MIDI (GM) compatible device for immediate output (from Improv 2.3.0 MidiOutput Class Definition): |  |  |  |

int **cont**(int channel, int controller, int data);

Sends the *value* for the MIDI continuous *controller* command (0xb0) on the specified MIDI *channel*. Here is a list of continuous controllers.

?? *channel:* MIDI channel [0..15]

?? *controller:* MIDI continuous controller number [0..127]

?? *value:* MIDI data [0..127]

int **off**(int channel, int keynum, int releaseVelocity);

Sends a note off command using midi command 0x80 with the specified key number and release velocity. Note that the most common way of turning off a note is to send the note-on command (0x90) with an attack velocity of 0 (see the play function.

?? *channel:* MIDI channel [0..15]

?? *keyno:* MIDI key number [0..127] (middle C = 60)

?? *value:* release velocity [0..127]

int **pc**(int channel, int timbre);

Sends a MIDI patch change which changes the *timbre* on the specified *channel*. If you want to change to a timbre greater than 127, then check how your synthesizer does this. Usually, timbres are organized into banks, and you specify first which bank with the continuous controller #0.

int **play**(int channel, int keynum, int velocity);

Sends a note on or note off on the specified *channel*. If *velocity* parameter is missing, then a note off command is sent.

?? *channel:* MIDI channel [0..15]

- ?? *keyno:* MIDI key number [0..127] (middle C = 60)
- ?? *value:* attack velocity [0..127] (note off = 0)

int **pw**(int channel, int mostByte, int leastByte);

Sends pitch wheel information on the specified MIDI *channel*. For the three parameter version, the *mostByte* = coarse tuning values 0..127 which are the most significant 7 bits of a tuning value, and the fine tuning values (*leastByte*) 0..127 are the least significant 7 bits of a tuning value. For the two parameter version, tuningData is a 14 bit number.

int **pw**(int channel, int tuningData);

Sends pitch wheel information on the specified MIDI *channel*. For the three parameter version, the *mostByte* = coarse tuning values 0..127 which are the most significant 7 bits of a tuning value, and the fine tuning values (*leastByte*) 0..127 are the least significant 7 bits of a tuning value. For the two parameter version, tuningData is a 14 bit number.

int **pw**(int channel, double tuningData);

Converts a number in the range from -1.0 to +1.0 into a 14 byte number which is sent out with the pitch wheel command.

void **recordStart**(char *filename, int format);

Starts recording MIDI output to the file *filename* according to the specified *format* which defaults to ascii. MIDI output sent through the *send* command are recorded, which output send through the *rawsend* command are not recorded. If the file already exists, the file will be overwritten. The defined formats are:

- ?? **0** ascii format
- ?? **1** binary format
- ?? **2** Standard MIDI file format, type 0

Here is a description of the formats for recording MIDI output.

void **recordStop**(void);

Stops recording MIDI output to the file specified with recordStart.

void **reset**(void);

sends the MIDI command 0xFF which should force the MIDI devices on the other side of the MIDI cable (which is connected to the port of the MidiOutput object) into their power-on reset condition, clear running status, turn off any sounding notes, set Local Control on, and otherwise clean up the state of things.

int **send**(int command, int p1, int p2);

Sends MIDI data from the computer to a synthesizer. If not recording, then just calls *rawsend* . If recording, bytes are stored in an output buffer until the buffer is supposed to be flushed. Then the time since the previous flush is calculated and recorded, as well as all the bytes in the output buffer.

- ?? *byte:* an 8-bit MIDI value to be sent out.

?? *flush:* 0=store *byte* in output buffer, 1=send output buffer data to MIDI I/O as well as current byte.

int **send**(int command, int p1);

Sends MIDI data from the computer to a synthesizer. If not recording, then just calls *rawsend* . If recording, bytes are stored in an output buffer until the buffer is supposed to be flushed. Then the time since the previous flush is calculated and recorded, as well as all the bytes in the output buffer.

?? *byte:* an 8-bit MIDI value to be sent out.

?? *flush:* 0=store *byte* in output buffer, 1=send output buffer data to MIDI I/O as well as current byte.

int **send**(int command);

Sends MIDI data from the computer to a synthesizer. If not recording, then just calls *rawsend* . If recording, bytes are stored in an output buffer until the buffer is supposed to be flushed. Then the time since the previous flush is calculated and recorded, as well as all the bytes in the output buffer.

?? *byte:* an 8-bit MIDI value to be sent out.

?? *flush:* 0=store *byte* in output buffer, 1=send output buffer data to MIDI I/O as well as current byte.

void **silence**(int aChannel = -1);

silence MIDI data from the computer to a synthesizer. If channel == -1, then send note off commands on all channels, otherwise send only on specified channel.

void **sustain**(int channel, int status);

Turns on/off the continuous controller 0x40 (sustain). Turn on sustain with sustain(1). Turn off sustain with sustain(0). Same as off=*cont(channel, 0x40, 0)*, or on=*cont(channel, 0x40, 127)*.

?? *channel:* MIDI channel [0..15]

?? *status:* on/off switch, [0..1] 0=off, 1=on

Module: MCMTranslate::Translator

| **Function Name** | **Return Type** | **Parameters** | **Description** |
|---|---|---|---|
| Translator | None | None | Constructor |
| setRTHostName | void | char * *strHostName* | Sets the internal variable for the host name of the Vicon RT system. This variable will be used to initialize the StreamReader |

| | | | |
|---|---|---|---|
| | | | object. |
| setRTIPAddress | void | char * *strIPAddy* | Sets the internal variable for the IP address of the Vicon RT system. This variable will be used to initialize the StreamReader object. |
| setMappingFile | void | char * *strFilePath* | Sets the internal variable for the absolute path for the mapping information. This variable will be used to initialize the MappingInfo object. |
| initialize | bool | None | Initializes itself and all the objects it uses. It will initialize the StreamReader, MappingInfo object, and the MIDIController. This function will return TRUE if successful, FALSE otherwise. |
| run | int | None | Runs the translator, along with all subservient objects (ie. StreamReader). It returns and integer that represents a return code. A return code of 0 means that the function exited gracefully. |

# 4. Integration Test Plan

The purpose of the MCM system Integration Test Plan (ITP) is to ensure that the designs of the modules are compatible. After individual unit testing, the modules will be brought together and their interfaces and interactions with each other shall be tested.

<u>Data Sets and Testing Conditions:</u>

Our team made extensive use of .v (pre-recorded motion capture) files and .mcm (mapping information) files to complete the integration testing of our system. The .mcm files used in testing were simple mappings of 3-6 rows of mapping data, usually named a derivative of *test*.mcm. The final testing .mcm file, complete.mcm, is included with this document. Another testing file, yoyoyo.mcm, was used to test things related to the entire body part listing. The one .v file we had to use, hvdemo07.v, included the motion of a person walking in a circle.

The .v file was used by the Vicon Real-time Emulator to feed a stream of data to the StreamReader module over the network port 800. The StreamReader module then passed the motion data to the Translator module, which then passed it on to the MIDIOutput module. The IO module, with the help of the MCMMap module, created .mcm files, which were in turn read in by the MappingInfo module. The Translator module used the data structure created by the MappingInfo module.

The MCMMap and MCMTranslate modules were tested separately before they were brought together on the same machine. The main MCMMap module was integrated with the IO module by adding the information from the various GUI elements one at a time until the correct information on the whole was being written to the .mcm files. A similar yet reverse method was used to read the information from the .mcm file back into the GUI components for the file-open functionality. The interfaces between the various MCMTranslate modules were individually tested and integrated into the system as a whole. First, our team attached the StreamReader module to the Translate module assuring that data points could be read off the network and decoded via data structures. By running the StreamReader module and printing out the data points the connection between these two modules was verified. Next, the team included the MappingInfo module in the project and initialized the class with the test mapping (.mcm) file. Queries were made on the MappingInfo object containing data stored in the mapping file. Finally, the two connections were fused by obtaining data from the StreamReader object and using the information to make the proper queries on the MappingInfo object. This gave verification by creating a MIDI command that could be sent to the MIDI module.

We performed full integration tests by installing both executables on the same machine and using one program to call the other. We created a .mcm file using MCMMap, saved it, and then called MCMTranslate. The Real-time Emulator needed to have been running a .v file at this point to simulate a motion capture. The StreamReader module also needed to wait for the MappingInfo module to create the data structure before it could look for the stream data. We tested each data set multiple times to make sure that the same MIDI output was created for the same .v file and .mcm file pair.

We need to either move the system to the client machine or acquire more sample .v files from the client in order to do testing of simple motions and mappings.

<u>Possible Risks/Problems:</u>

The possible problems that may occur once we move the system over to the client machine involve the individual programs running as stand alone programs outside of their respective development environments. The team has had trouble running MCMMap.exe on a machine on which it was not already compiled and running in a development environment. We found that this was due to MCMMap requiring the Java Runtime libraries to be present on the host machine to run properly.

Unforeseen problems may also occur once the system must work with the Vicon hardware as we have as yet only tested it using the Real-time Emulator software.

Due to limitations in the Vicon API, MCMTranslate must be recompiled once the IP address of the machine the stream is coming from is known. The IP address of this machine must be hard-coded into our system.

<u>Test Matrix</u>

The following individual test cases were performed in support of the above-mentioned unit-testing and configuration tests. They supported the overall goal verifying the system design specifications. The tests conducted were:

| | |
|---|---|
| **Test Name:** | Map Program |
| **Purpose of Test:** | To verify proper running of MCMMap |
| **Module Interactions:** | Map::GUI, Map::IO |
| **Input Specification:** | Start program |
| **Output Specification:** | Program loads |
| **Test Environment Restrictions:** | Does not need MCMTranslate present |

This is the most simple of the testing procedures, and is provided to ensure that the method for the user to create mappings is available.

| | |
|---|---|
| **Test Name:** | GUI |
| **Purpose of Test:** | To verify functionality of GUI comment of MCMMap |
| **Module Interactions:** | Map::GUI |
| **Input Specification:** | User manipulation of all modifiable graphical elements and text fields. |
| **Output Specification:** | No error messages upon data entry |
| **Test Environment Restrictions:** | none |

There are numerous individual GUI elements in the user interface available. This includes the text fields used for range entry as well as the drop down menus that describe the available MIDI commands. Limitations on range as described in the requirements are implemented in the design and will feature in several iterations of this particular test.

| | |
|---|---|
| **Test Name:** | Map IO |
| **Purpose of Test:** | To verify functionality of IO component of MCMMap |
| **Module Interactions:** | Map::IO |
| **Input Specification:** | Save/Load file |
| **Output Specification:** | Save/Load file (corresponding) |
| **Test Environment Restrictions:** | none |

The Input/Output routines must write a properly delimited text file of the mapping information to disk.

| | |
|---|---|
| **Test Name:** | Translate Program |
| **Purpose of Test:** | To verify proper running of MCMTranslate |
| **Module Interactions:** | Translate::IO/MIDI/StreamReader/Translator |
| **Input Specification:** | Start program during capture |
| **Output Specification:** | Program executes during capture, generates sound |
| **Test Environment Restrictions:** | Can only be performed upon the completed testing and verification of functionality of the individual modules comprising this portion of the entire system. |

This test should be the final step in integration testing. The mapping element of MCM exists as its own independent component, able to be used without the presence of this portion. However, this particular test is the complete run of all the significant system components that allows the motion capture system users to generate sounds in real-time.

| | |
|---|---|
| **Test Name:** | Translate IO |
| **Purpose of Test:** | To verify functionality of IO component of MCMTranslate |
| **Module Interactions:** | Translate::IO/Translator |
| **Input Specification:** | Load mapping |
| **Output Specification:** | Build data structure of mapping description for use in translation |
| **Test Environment Restrictions:** | Must be tested concurrently with translator |

The Translator cannot perform any sort of conversion of motion data to sound data without the presence of the mapping, which serves as the "road map" for the translation process.

| | |
|---|---|
| **Test Name:** | Translate MIDI |
| **Purpose of Test:** | To verify proper operation of the MIDI output routines |
| **Module Interactions:** | Translate::MIDI/Translator |
| **Input Specification:** | MIDI commands |
| **Output Specification:** | Sounds or modification of current sound from output device |
| **Test Environment Restrictions:** | This test is performed in conjunction with a MIDI output device connected to the RT workstation. |

The MIDI output routines are provided by a third-party organization, which professes the completeness and usability of their code.

| | |
|---|---|
| **Test Name:** | Translator |
| **Purpose of Test:** | To verify satisfactory operation of the heuristics code in the translation portion of the system |
| **Module Interactions:** | Translate::IO/StreamReader |
| **Input Specification:** | Decoded motion data from StreamReader |
| **Output Specification:** | Corresponding MIDI command |
| **Test Environment Restrictions:** | none |

This test features an individual motion data element's information causing the generation of the mapping-defined equivalent MIDI command.

| | |
|---|---|
| **Test Name:** | StreamReader |
| **Purpose of Test:** | To verify network operation of the Vicon supplied motion data reader code |
| **Module Interactions:** | Translate::StreamReader |
| **Input Specification:** | RT Emulator or Vicon 8 Camera system motion output |
| **Output Specification:** | Readout of coordinates for each point |
| **Test Environment Restrictions:** | A capture (either simulated or real) must be being performed with the data moving on a closed local area network; either the Vicon 8 camera system or its emulator can be used. |

# 5. Tracking and Control Mechanisms

Since the system development for MCM is rather small in comparison to other software projects, the team as opted not to use conventional tracking and control mechanisms such as CVS or other code/update repositories.

The benefit of the separation of program elements into two parts, mapping and translation, and their subsequent breakdown into individual modules, allows each team member to cover a specific area of detail without saturating his/her workload.

The team website and account at **www.gts2k.com/~ics125/** provides a common area for storage and transmission of all documents, code, and testing information.

The following table provides a listing of how the MCM requirements, as specified in the Requirements Specification, are satisfied by the design presented in this document.  For further information, refer to the Requirements Specification Document.

| **IMPLEMENTATION MAP** | |
|---|---|
| **REQUIREMENT** | **DESIGN MODULE** |
| 3.2 Environmental Characteristics | MCM has been designed to run on the Vicon8 network in a standard Microsoft Windows NT/2000 operating system environment. |
| 3.3.1 Correctness | All |
| 3.3.2 Reliability | All |
| 3.3.3 Robustness | All |
| 3.3.4 Performance | MCMTranslate (All) |
| 3.3.5 User-friendliness | MCMMap::GUI |
| 3.3.6 Verifiability | All |
| 3.3.7 Maintainability | All |
| 3.3.8 Reparability | All |
| 3.3.10 Evolvability | All |
| 3.3.11 Reusability | All |
| 3.3.12 Portability | All |
| 3.3.13 Understandability | All |
| 3.3.14 Interoperability | All |
| 3.3.15 Productivity | MCMMap::GUI |
| 3.3.16 Scalability | All |
| 3.3.18 Visibility | All |
| 3.4.2 Transfer of Data | MCMTranslate::StreamReader |
| 3.4.3 Motion Data Mapping | MCMTranslate::Translator |
| 3.4.5 Data Range | MCMMap::MapData |
| 3.4.6 Pairing of Data Point Axis with MIDI Command | MCMMap:MapData |
| 3.4.7 MIDI Command Selection | MCMTranslate::MIDIControl |
| 3.4.8 Axis Mapping | MCMMap::MapData |
| 3.4.9 Multiple Mapping Configurations | MCMMap::IO |
| 3.4.10 Note Velocity | MCMMap::MapData, MCMTranslate::MIDIControl |
| 3.5 UI Model | MCMMap (All) |

| | |
|---|---|
| 3.8 File Format | MCMMap::IO,<br>MCMTranslate::Streamreader,<br>MCMTranslate::MappingInfo<br>MCMTranslate::Translator |

# 6. Modifications to Requirements

For reference and further information, the reader is referred to the Requirements Document Version 1.1. Changes to the document occurred primarily because of the ambiguity in the definition "linear mappings" as related to the translation of motion data to MIDI. Other changes were for clarification and there were also several cosmetic changes.

| **Date** | **Version** | **Status/ Action** | **Revision By** | **Comments** |
|---|---|---|---|---|
| 4/26/01 | 1.0 | Completed Requirements Specification | All | The Requirements were submitted and presented today. |
| 5/17/01 | 1.1 | Added more detail and clarification to the document. | All | ?? Made extensive modifications to Section 3.4 Domain Specific Rules..<br>?? The Use-Case Scenarios were moved to Section 3.7, after Section 3.5, which defines the components discussed in the Use-case scenarios.<br>?? Updated Section 6 Test Plan to include environmental requirements.<br>?? Added several terms to the glossary.<br>?? Updated the Section 9 Meeting Minutes to show the latest Requirements meetings. |

# 7. Meeting Minutes

**9.1 - Sunday, 6 May 2001**
Evening meeting of Sal, Jimar, and Gene to finish research, work on technical presentation slides, and begin construction of the design document.

**9.2 - Monday, 7 May 2001**
Short meeting with Professor Ebert at 12:30pm to discuss preparation for technical presentation and answer questions about topics to focus on. Sal, Jimar, and Gene present.

**9.3 - Tuesday, 8 May 2001**
Morning meeting before class to complete technical presentation slides. Sal, Jimar, and Gene present.

**9.4 - Wednesday, 9 May 2001**
Regular weekly meeting in the evening to go over required details for design document, proceed with individual module designs and separation of tasks. Sal, Jimar, and Gene present.

**9.5 - Saturday, 12 May 2001**
Planned afternoon meeting for Jimar and Gene to test Vicon StreamReader code with RT emulator missed because scheduling conflict.

**9.6 - Tuesday, 15 May 2001**
Morning meeting to coordinate slides for Design presentation. Sal, Jimar, and Gene present design concepts for MCM and related technologies in class.

**9.7 - Saturday, 19 May 2001**
"Online" meetings to coordinate final additions and changes to design document.

# 8. Glossary (Local to Design Phase)

API
Application Programming Interface – a set of functions that allows the developer to use the functionality of an existing application.  To use the application, the developer makes the appropriate function calls.  The API usually includes documentation that defines what the functions are, what the parameters are, and what the return values are.

C3D File Format
The C3D format stores 3D coordinate and numeric data for any movement measurement, usually used in recording Biomechanics experiments.

GUI
Graphical User Interface

ITP
Integration Test Plan

MIDI
Musical Instrument Digital Interface.  MIDI is a standard protocol that was agreed upon by major manufacturers of Electronic Musical Instruments.  It allows Keyboards, Synthesizers, Computers, Tape Decks and even Mixers & Stage Light Controllers to talk to each other.

RT Emulator
A software program that reconstructs in real-time a Vicon 8 camera system motion capture event.

RT Workstation
The Vicon company's high-powered PC running Windows NT/2000 with Vicon software that accepts the real-time capture stream on the network and outputs the current capture to a video display in real time.

TCP/IP
Networking communication protocols for transferring data from one computer to another.  The routable protocol is the standard for the Internet and it ensures reliable data transfer and congestion control.

## 9. Revisions

| Date | Version | Status/Action | Revision By | Comments |
|---|---|---|---|---|
| 5/21/01 | 1.0 | Completed Design Specification | All | The Design was completed and submitted and today. |
| 6/5/01 | 1.1 | Added more detail and clarification to the document. | All | ?? The Introduction was expanded to include the same introductory material presented in the Requirements document.<br>?? The module descriptions were changed to add more detailed information about their functionality and their interfaces.<br>?? Several cosmetic changes were made to make the document more presentable.<br>?? Several relevant terms were added to the glossary. |
| | | | | |
| | | | | |