MUSIC

a

Multi-**U**ser **S**ystem

for

Instrument **C**ontrol

System Coordination Overview

1.0 Introduction

The **MUSIC** system design is based on the assumption that remote observing will be a routine part of Keck Observatory operations and that teams of researchers will often work together on a single observing run. In addition it is assumed that both of these situations will often arise concurrently; there will be a team of observers, and this team will not be gathered at one central location. In most data-acquisition systems there is a single point of control, a terminal or a workstation, where one observer enters commands to a computer and controls the operation of the instrument and where information about the operation of the instrument is displayed. As a result the other members of the observing team, whether local or remote, tend to wait around in a fairly non-productive manner while the one active observer does all of the work. This is especially true if the observers are geographically dispersed since it then becomes difficult for the remote team members to know what the active observer is doing.

MUSIC strives to provide real support for team activities by making it possible for observers to work in a more productive, coordinated manner. To accomplish this, the single point of control has been eliminated in favor a multi-user system in which many control points are possible. These control points can be located anywhere a terminal or workstation can be located, from the Nasmyth platform on the telescope to the observer's kitchen in California. The astronomer at each control point can run a program which supplies the user interface into the system. The particular user interface chosen will depend on the observer's needs and the capabilities of the control point hardware. **MUSIC** will support multiple users running different user interface programs concurrently, and these programs need not run on a single computer. To provide this level of flexibility **MUSIC** is composed of a collection of closely synchronized, co-operating processes which can be divided into three categories: 1) User interfaces, 2) Data-acquisition processes, and 3) System coordination processes. As described above there will be one or more user interface programs running while the system is in use. There may be several data-acquisition processes, and these processes will generally be designed such that each has primary responsibility for controlling or interacting with one or more closely related peripherals. For instance, there may be one process responsible for controlling the detectors and instrument and one process responsible for recording observations on tape or other archival storage media. Underlying all of these processes is the set of system coordination processes. The system coordination processes control the execution of the other processes in the system and help to manage the flow of information between them. It is the design of these coordination functions which make the multiple control point **MUSIC** system possible.

The remainder of this document provides an introduction to the system coordination processes. Most of the descriptions given in this document are based on programs already developed for the Lick Observatory data acquisition systems on Mt. Hamilton. Some of the programs, especially those labeled below as data acquisition processes, could have major differences between the Lick and Keck versions. However, such differences will have little impact on the discussion which follows.

2.0 System Coordination Processes

Figure 1 shows a typical set of processes which constitute the **MUSIC** system in a single cpu environment. Three types of activities must be coordinated for the **MUSIC** system to work: process start-up and shutdown, inter-process communications, and information storage and retrieval. Three programs have been developed, *runner*, *traffic*, and *infoman*, to control each of these activities. These are general purpose programs in the sense that they know nothing about the specific data acquisition system of which they are a part, and they do not depend on system specific resources other than TCP/IP-based Unix sockets. Therefore, they are applicable to a variety of systems and computers and in fact, they are already in use on Lick Observatory's data acquisition computers. The *traffic* controller and *runner* processes are started at boot-time and are always running. When no user interfaces are running these processes are idle, waiting for new connections on their TCP/IP network sockets.

At the request of the user interface processes, the *runner* program starts up the various data-acquisition processes and the *infoman* process, and it terminates them when there are no more user interface processes running. The *traffic* program is the central communications hub for the entire system. The lines which connect the various processes shown in Figure 1 represent inter-process TCP/IP Unix sockets. Except for the paths to the *runner* process which are used only at the initial start-up and final shutdown of each user interface, essentially all communications between processes takes place indirectly through the *traffic* controller. One of its most important functions is to receive information from a single process and to broadcast that information out to a selected set of other processes. It is this function which keeps multiple user interfaces updated with the latest instrument data. The third coordination process, *infoman*, is a simple data-base program and is responsible for maintaining most of the non-real-time information used by the **MUSIC** system. It responds to requests for information by other processes and it accepts information from other processes. Whenever *infoman* receives new information, it sends out a message which is broadcast by the *traffic* controller, and this message informs other processes of the change. If any process then determines that the changed information is needed, it can request *infoman* to send that information. In this way one user interface can be informed of changes introduced by an observer using another interface. Each of the three system coordination programs is described in detail below.

2.1 Runner

The *runner* process performs two primary functions, process start-up and process shutdown. It also provides an important secondary function by generating unique process names which are used later to make connections through the *traffic* controller. As noted earlier, the *traffic* controller and, *runner* processes are always running, waiting for new connections on their TCP/IP network sockets. When an observer starts up a user interface it establishes a TCP/IP connection to the *runner* process. It then sends requests to the *runner* process to start up the other parts of the **MUSIC** system as are appropriate (i.e. the 'dtake', 'fitstape', and *infoman* processes shown in figure 1). For each request to start a process, the user interface will receive a response indicating whether or not *runner* could successfully start the requested process. In the case of a success response *runner* also sends a unique name to be used in making connections through the *traffic* controller, and it adds the new process to its internal list of currently running processes. The unique name is also passed as one of the program arguments to the process being started so it can identify itself to the *traffic* controller with the appropriate name. The unique names are created by concatenating the hostname, the process name, and the process ID number.

If there were only one user interface the actions performed by *runner* could more simply be carried out directly by the user interface. The real advantages of the *runner* process are realized when multiple user interfaces are in use. When the second and subsequent user interfaces are started up they too make a connection to the *runner* process and they make the same requests as did the first user interface. But this time *runner* finds the requested processes in its list of currently running processes. So instead of starting up another copy it just returns a success message to the user interface for each requested process, and it records that another user interface has requested that process. As a result any number of user interfaces can be started in any order and as far as the user interfaces are concerned, they have all started up the **MUSIC** system in the same way.

As an observer closes down their own user interface the *runner* process detects the lost TCP/IP connection which the interface had originally established. In response to this lost connection *runner* will delete the record of that user interface from the list maintained for each requested process. When the last user interface is deleted from the list for a particular process, *runner* will send that process a Unix SIGTERM signal to terminate the process. Because of the way *runner* starts and stops processes, the entire **MUSIC** system is brought up when the first user interface is run and it remains up until the last user interface terminates.

Although figure 1 illustrates the processes for a single computer system, it is possible to distribute the **MUSIC** system over several computers. The user interfaces, for instance, can be run on any machine which can reach the mountain-top network. (To provide security observers will have to know the official password to run a remote user interface.) The data acquisition processes can be run on any machine which has a *runner* process to start them. Details on the use of *runner* are given in the Runner User's Guide, Part 2 of UCO/Lick Technical Report 55.

2.2 Traffic

The *traffic* controller process is the central communications hub for the **MUSIC** system. There is one *traffic* controller process, into which all other processes make TCP/IP connections. The user interfaces, the data acquisition processes, and the *infoman* process do not establish direct TCP/IP connections with each other nor do they communicate directly with each other. If they did there would be a large, complex, collection of TCP/IP interconnections. Such a large collection of TCP/IP connections adversely effects overall network performance and it adds extra complexity to each process which must deal with the connections. Instead, each process in the **MUSIC** system makes a single connection to the *traffic* controller.

Processes can send messages to other processes which have made connections to the *traffic* controller. The messages have a specific format consisting of a fixed length message header and a variable length message body. The header contains information on the type of message, the source and destination for the message, and the length of the message body. It also contains error checking/recovery data. The content of the message body depends on the type of message and the particular application for which the message is used. Details of the message format are given in the Traffic Controller User's Guide, Part 1 of UCO/Lick Technical Report 55.

As part of the initial connection procedure, each process identifies itself to the *traffic* controller with a unique process name. Remember that these names are assigned by the *runner* process to the data acquisition and *infoman* processes. The user interface processes create their own unique names from a combination of hostname, process name, and process ID. In response to the connection request, *traffic* assigns each process a small, positive number, which becomes that process's 'message address'. In order for one process (A) to send a message to another process (B) through the traffic controller, process A first sends the unique process name for B to *traffic* and asks for its message address. Then process A can construct messages addressed to B and send them off to *traffic* which in turn, after some error checking, forwards them to B.

This procedure is generally referred to as a connection from process A to process B through the *traffic* controller, but of course all it really means is that process A has obtained process B's message address. Note that while process A must have some knowledge of process B (its unique name to be specific), process B does not have to know of the existence of process A in order to receive messages from it. This is exactly the situation which arises with multiple user interfaces. As each user interface is started up it obtains the unique process names of the other **MUSIC** processes from *runner* and connections to those processes may be established through the *traffic* controller. The **MUSIC** data acquisition processes neither know about, nor care how many, user interfaces are running.

Although the data acquisition processes and *infoman* do not attempt to learn directly about all of the user interfaces running, they can nevertheless, send them messages in two ways. The first way is in direct response to a message received. Each message contains the message address of the sender of the message; this return address can be used to direct a reply to the original sender. *Infoman* uses this technique to respond to requests for information. The second technique of sending messages used by the data acquisition processes and *infoman* is to send out a 'broadcast' message, which is identical to a normal message except that the message address is set to -1. When the *traffic* controller receives a broadcast message it searches for a list of processes which have already told the *traffic* controller they want to receive that particular type of broadcast message, and *traffic* will send a copy of the message to each of those processes.

The display of remaining exposure time during an observation is one simple example of the use of broadcast messages. If a user interface provides for the continuous display of exposure time, for instance, it sends a message to the *traffic* controller requesting any remaining-exposure broadcast messages. The 'dtake' process, which monitors and manipulates the instrument CCD controller, issues a remaining-exposure broadcast message once per second during an observation. This single message is then replicated by the *traffic* controller and sent to all appropriate user interfaces, and the receiving user interfaces can update their displays. This same technique is used to distribute a variety of information on detector status, instrument status, tape drive status, and other real-time events which may be of interest to a variety of user interfaces. *Infoman* also uses it to broadcast notices of changes in the data it is managing.

Details on the use of the traffic controller and the format of messages is given in the Traffic Controller User's Guide, Part 1 of UCO/Lick Technical Report 55.

2.3 Infoman

The observer enters commands and setup data via the user interface. If there were only one user interface then that data could be stored, retrieved, and distributed to other processes directly by the user interface. But when there are multiple, simultaneous user interfaces running, the user interface can no longer be responsible for managing the information received from the observer since there is no longer a single, unique source for that information. For this reason the *infoman* process was developed. It is the single, centralized source for nearly all non-real-time data used in the **MUSIC** system. These data include items such as observation parameters, detector setups and limits, instrument setups, limits, and symbolic names, object names, the observer's name, observer comments, and observation number. The user interface processes and the data acquisition processes all obtain such data from *infoman* and they all send changes in such data to *infoman*. Although it does not have the capability yet, *infoman* may also serve as the central source for online help and error logging. It is not the source for dynamically changing real-time data such as the current positions of particular instrument elements or the current status of an exposure. This type of information is obtainable from the data acquisition processes. *Infoman* does not store or retrieve CCD images.

Although the information which *infoman* manages may be very instrument specific, *infoman* itself is general purpose. The data which are to be managed are described in a file, or set of files, known as the dictionary, and *infoman* reads the dictionary to obtain the definitions for the data it is to manage. Through use of the dictionary *infoman* can be configured to support a variety of data acquisition systems. The dictionary defines three categories of items: 1) the locations and characteristics of the data, 2) the data requests to which *infoman* should respond, and 3) the relationship between the data managed and the data requests.

The data managed by *infoman* ultimately reside in a set of disk data files, and these files can be classified as dynamic string files and static data files. In this context 'dynamic' and 'static' refer to the structure of the files, not the contents. The dynamic string files are simple ASCII text files. Each line in such a file has a leading word, an ASCII space, and a following character string; the leading word is basically a name for the string. As *infoman* encounters references to string files in the dictionary it reads the string files and makes a table of all string names and corresponding strings. Processes can then request strings by name, can delete strings, can modify existing strings, and can send *infoman* entirely new strings and string names. As *infoman's* table of strings changes the corresponding original disk files are updated. This simple string facility is used in **MUSIC** to assign names to various elements of the system like filter positions, CCDs, and instrument setups.

The static data files have a more structured format than do the dynamic string files. They consist of variable length lines of ASCII text, each line a FITS-like entry with a parameter name, an equal sign, a parameter value, and an optional comment introduced by the virgule (/). Each line is terminated by a newline character. The variable length and newline character are not FITS-card format but they do make the file easily readable. In addition, comment lines beginning with the virgule are also allowed. The parameter values may change, but these files are static in the sense that the number of parameters in the file is fixed and new parameters can not be introduced after the dictionary is completely processed. These files are suitable for storing setups, limits, and similar data.

For the static data files, *infoman* will be able to provide and accept information in two formats. In the first format only parameter values will be passed, and they will be passed in their 'natural' forms (numeric data will be in binary form, character data will be null-terminated, etc.). In the second format parameters will be passed in true 80-character FITS card-image format. When information requests are received by *infoman* for information in this format, the parameter names and comment fields found in the original disk data files will be used in constructing the 80-character FITS card responses.

Infoman tries to minimize its disk activity and thereby increase its responsiveness in two ways. First, after *infoman* has processed the dictionary, it will have cached in memory all of the data it is to manage. When requests for data are received the in-memory, cached data will be returned. Second, when updated data are received by *infoman* the names of the files containing the modified data are placed on an update queue. Only after a file has been on the update queue for 20 seconds will the file on disk be updated. This queueing mechanism allows processes to send new information to *infoman* in a series of transactions and, if that information resides in one file, the corresponding file will probably only be updated once. When *infoman* receives the Unix SIGTERM signal from *runner* upon final shutdown of the **MUSIC** system, it flushes all of the files on the queue before terminating.

One other feature of *infoman* is essential to the operation of the **MUSIC** system. As mentioned previously, whenever new data are received *infoman* sends out a broadcast type message to indicate that a change has taken place. The message includes the message address of the process sending the new data and information on what data have been changed. User interface processes can receive this broadcast message and can decide if the changed data are needed. If needed they can then individually request the data from *infoman*. Note that the changed data was not transmitted in the original change broadcast message since this would require every user interface to accept incoming *infoman* data in two ways, as a response to a direct request, and as part of a change broadcast message.

As an example, consider how the object name might be handled. Suppose several user interface processes are running and they are each displaying the object name for the next observation. One observer then decides to change the name and proceeds to do so. That user interface sends to *infoman* the object name just entered by the observer. In response to the change *infoman* sends out a broadcast message and all of the user interfaces receive this broadcast message. The user interface which sent in the original change recognizes its own message address in the broadcast and since it already has the new object name it takes no further action. But the other user interfaces recognize that they need the new object name, so they make the necessary request to *infoman*, and upon receipt of the new object name they proceed to update their own displays. Of course, any user interface not displaying the object name, as might be the case with a command driven interface, would also discard the change broadcast message and take no further action. In this way all user interfaces in the **MUSIC** system can be properly kept informed of the activity of other observers.

This change notification mechanism can be used to provide a simple means of automatic backup of the *infoman* database. Suppose **MUSIC** is running on machine A but that another machine, B, could be used if A were to break down. If the switch to B were needed it would be convenient if B were to have the same database as A. To accomplish this another copy of *infoman* could be run on B along with a program which connects, through the *traffic* controller, to *infoman* on both A and B. This process tells the *traffic* controller on A that it would like to receive the change broadcast messages from *infoman* on that machine. Whenever it receives such a broadcast from *infoman* on A it requests the changed information and then sends it on to *infoman* running on B. In this way the database on B is kept current with the one on A.

Infoman will provide one other system coordination function for the **MUSIC** system. Since there can be multiple user interfaces running and these interfaces may be geographically dispersed, it could be possible for two observers to attempt to modify the same data at the same time or for one observer to modify some data in a way which conflicts with the data entered by another observer. If such interactions are not controlled they could lead to confusion and lost observing time. For this reason the user interfaces will operate under an arbitration scheme in which some user interfaces (the 'active' ones) will be able to modify system data and others will not (the 'passive' ones). *Infoman* will serve as the central arbitrator in this scheme. The active user interfaces will be able to operate without restriction while the passive ones will be able to receive and display information but not modify it.

To make the arbitration as flexible as possible we plan three modes for deciding which interfaces are active and which are passive: 1) everyone is active, 2) release on request, and 3) release when done. In the first mode all user interfaces are active. This mode is appropriate when there is only one observer or when the observers can be certain they will not accidentally interfere with each other. In the release on request mode only one user interface is active at any one time, but an observer at any other user interface can request active status and it is immediately granted by *infoman*. The active user interface then becomes passive. In the most restrictive mode, release when done, any user interface can request to become active, but the observer at the currently active interface must first indicate to *infoman* that they are willing to relinquish active status. Any process can ask *infoman* to identify which user interface is currently active and *infoman* will issue a broadcast message whenever a change takes place. The user interfaces can then display this information to keep observers informed about who has active control.

Details on the use of *infoman* including details on the format and capabilities of the dictionary can be found in the Infoman User's Guide, UCO/Lick Technical Report 56.

3.0 Making it all Play

In this section we give a brief, step-by-step description of what actually happens when a user interface is started by an observer. This description is based on the preliminary use of the Lick **MUSIC** system, and while the operation of the Lick system is likely to differ in detail from the Keck system, one of our design goals is to make their overall behaviors very similar. Although the **MUSIC** system can be distributed across a number of networked computers, our first example (Sections 3.1 and 3.2) considers only the case in which all processes run on the same machine. The processes considered will be the ones shown in figure 1. Our second example, (Sections 3.3 and 3.4) will illustrate a system in which two computers are involved.

3.1 The dtakeservice File

There are basic data which must be available to the user interface, and to the other parts of the **MUSIC** system, in order to set the system up. These data include TCP/IP network connection information, and path names for executable files. All of this information is collected in a single configuration file called dtakeservice. The simplest form of this file consists of a series of text lines and each line contains two strings separated by at least one space or tab character. The first string on the line is effectively the name for the second string. Also, anything following a pound sign (#) is ignored. A simple C routine is available whose input parameter is the name (the first string) and whose output returns the second string. A simple dtakeservice file might look like:

```
runnerport     1990       # Port number for runner process.
trafficport    1996       # Port number for traffic controller.
# Pathnames for MUSIC processes:
dtake          /u/ccdev/bin/dtake
fitstape       /u/ccdev/bin/fitstape
infoman        /u/ccdev/bin/infoman
```

3.2 User Interface Startup

When a user interface is started it consults the dtakeservice file for the name 'trafficport' and obtains the TCP/IP port number for the *traffic* controller process. Using the port number it establishes a network connection to *traffic* and it identifies itself to the *traffic* controller as described in section 2.2. It next consults the dtakeservice file for the name 'runnerport' and obtains the TCP/IP port number for the *runner* process, and using that number, establishes a network connection to *runner*. Once these connections are made the user interface proceeds to ask *runner* to start *infoman*, dtake, and fitstape, in that order. Each request to start a process includes the name of the process as given in dtakeservice. It does not include the complete file name of the process's executable image. Instead, *runner* uses the passed process name to look up the file name of the executable in the dtakeservice file. This distinction between passing to *runner* the process name instead of the file name is important when the user interface is running on a remote machine. That remotely running process should not have to know file names on another machine.

One of the first things the processes started by *runner* do is to look up the port number for the *traffic* controller, to make the TCP/IP connection, and to identify themselves to *traffic* using the name *runner* passed to them as an argument. At the same time, the user interface is using the *runner*-provided names to establish connections through the *traffic* controller to those processes. (See section 2.2 for more details). Since the user interface and the started processes are running asynchronously, it may happen that the user interface tries to establish a *traffic* connection to a process before that process has had time to identify itself to the *traffic* controller. In this case the user interface will receive a response from the *traffic* controller indicating failure in setting up the connection, and as a result the user interface will pause for one second and try again. The attempt to establish connection is repeated several times before the user interface gives up and issues a fatal error message. We have found that it rarely takes more than two tries to make the connection on locally networked machines.

As soon as the user interface has started *infoman*, it establishes the message connection to *infoman* through *traffic* and sends *infoman* a message which tells it to initialize itself, and to do so for a particular version of the **MUSIC** system. *Infoman* then processes the configuration dictionary for that version of the system (this is described in detail in the Infoman User's Guide, UCO/Lick TR 56) and returns a message to the user interface confirming that it is now initialized for the requested version. For the moment, assume that two instruments will run different versions of the **MUSIC** system and that for some technical reasons they can not both be run at the same time on the same machine. Consider what happens if a second user interface starts up, but for the other version of the **MUSIC** system. When it asks *infoman* to initialize itself, *infoman* will find that it has already done so and it will return to the second user interface the same response it returned to the first. But this time the version requested by the user interface does not match the version *infoman* returned. By comparing the request with the response the user interface can verify that the proper version of the system is running. If they don't match the user interface informs the observer that another version of the system is already running and then aborts.

The dtake process is responsible for running the CCD controller and must routinely obtain information about how to set up the CCD and how to read it out. Therefore, it too must make a connection through the *traffic* controller to *infoman*, where such data are maintained. To do so it must first obtain the unique name by which *infoman* has identified itself to *traffic* and this piece of information is provided by *runner*. So dtake first makes a TCP/IP connection to *runner* and asks *runner* to start *infoman*, just like the user interface did. However, since the user interface started *infoman* first, *runner* will immediately return a success message to dtake, along with the unique name for *infoman*. Dtake then closes down its TCP/IP connection to *runner* and establishes the message connection to *infoman* through the *traffic* controller, using the returned name. Note that dtake does not need to keep its connection to *runner* open since *runner* will not terminate *infoman* until the last user interface terminates. Essentially the same sequence of operations is carried out by the fitstape process for it to make a message connection to *infoman*.

The initial startup of the **MUSIC** system is now completed. The user interface has established message connections through the *traffic* controller to dtake, fitstape, and *infoman*; and dtake and fitstape have also established connections to *infoman*. When subsequent user interfaces (for the same version) start up they run through the same steps and end up with connections to dtake, fitstape, and *infoman*.

3.3 Dtakeservice Revisited

The descriptions given above, though complex, provide a simplified description of the real situation. In the **MUSIC** system, various processes of the system might be running on different machines, so to support this possibility one additional structure is added to the dtakeservice file and several more data are added. The final file might look like:

```
# network section defines TCP/IP port numbers and
# host for traffic controller and infoman
section:network
runnerport    1990       # Port number for runner process.
trafficport   1996       # Port number for traffic controller.
# hires section describes host names for processes and
# path names for local processes
section:hires
traffichost   hires
infomanhost   hires
fitstapehost  hires
dtakehost     ccdbox
fitstape      /u/ccdev/bin/fitstape
infoman       /u/ccdev/bin/infoman
# ccdbox section describes host names for processes and
```

```
# path names for local processes
section:ccdbox
traffichost   hires
infomanhost   hires
dtake         /u/ccdev/bin/dtake
```

The file is now divided into named sections and several more entries are added to supply the hostnames of the computers on which the various processes are to run. This particular example is modeled after the Lick system in which the dtake process runs on a separate cpu. The C routine which accesses this file now takes two input parameters, the section name and the particular item name, and it returns the second string on the line with the matching item name in the specified section. For instance, for the (section,item) pair ("hires","dtakehost") the routine returns "ccdbox".

3.4 Distributed System Startup

This example is illustrated in figure 2. We assume that the user interfaces will be run on the host named 'hires' and that the dtake process is run on the host named ccdbox. Note that in figure 2 we show only one of many possible user interface processes. When a user interface is started by an observer it reads the dtakeservice file, accessing section 'network' to get the port numbers for *traffic* and *runner*. The sequence of startup events proceeds exactly the same as in the first example, except that the user interface uses its own host name as the section name and looks up the host names for the other **MUSIC** processes. In particular it looks for ("hires","traffichost"), ("hires","infomanhost"), ("hires","dtakehost") and ("hires","fitstapehost"). It then sets up the TCP/IP connections to *traffic* and to *runner* on hosts hires and ccdbox, and it asks the appropriate *runner* to start dtake, fitstape, and *infoman*. The *runner* processes, each using their own host name as the section name, look up the full path names for the requested processes, and they run those processes. To be specific, the *runner* on host ccdbox looks up ("ccdbox","dtake") and finds "/u/ccdev/bin/dtake" while the *runner* on hires looks up ("hires","fitstape") and ("hires","infoman"). When dtake, fitstape, and *infoman* are started, they look in the dtakeservice file to find the host name and port number of the *traffic* controller and they set up their TCP/IP connections.

Although the description given here is somewhat verbose, the general organization of the dtakeservice file is simple. The information on the basic system facilities, needed by processes on any host, are located in the 'network' section of the file. Information needed by processes on a particular host are given in the section of the file whose section name is the same as the host name. This organization permits one to construct a single dtakeservice file which is appropriate for many combinations of hosts, and this single file can be copied to all those hosts.

Various additional options can be implemented with the dtakeservice file. For instance, alternate host names could be supplied, such that if a user interface could not bring the system up on host A it would then try on host B. There could also be a section of dtakeservice named 'remote' which would be used by any user interface whose host name did not appear as a section name. This section would contain just the hostnames of processes and would look something like:

```
section:remote
traffichost   hires
infomanhost   hires
fitstapehost  hires
dtakehost     ccdbox
```

This would make it easy to copy the dtakeservice file to any remote host without having to make a unique section for that host.

The routine which reads the dtakeservice file looks for the file in a standard directory. However, this directory can be over-ridden by defining the Unix environment variable DTAKESERVICE with an alternate directory. This makes it easier to run a user interface on a remote machine whose directories may not match the default, and it makes it easier to use an alternate dtakeservice file.

4.0 Continued Development

This document describes the system coordination processes in various stages of development at Lick Observatory (as of December, 1989). For instance, the format and use of inter-process messages has been defined and in use for more than a year in all of the Lick CCD data acquisition systems. A functional version of the *traffic* controller has existed for about 6 months and the *runner* and *infoman* processes have been in operation for about 2 months. The complete **MUSIC** system is currently in use in the CCD development lab in Santa Cruz and, as soon as the active/passive user interface arbitratation mechanism (see section 2.3) is fully incorporated, **MUSIC** will become the standard system on Mt. Hamilton. Because the system is still under development, details of the system are likely to evolve away from the descriptions given here, and new features are likely to appear as we gain more experience with multi-user interactions and we identify additional requirements and useful features for such a system. But the **MUSIC** system is well beyond the preliminary design stage, and the entire system will be fully developed and tested long before any Keck instrument software is operational. It should provide a stable framework upon which to build that software.

The author would like to acknowledge the work of April Atwood, who wrote the first version of the *traffic* controller, and the many contributions by Robert Kibrick, who always knows what questions to ask to reveal flaws in the design.