555-7001-316

Meridian ACCESS
Developer's Guide

Product release 12 Standard 1.0 January 1998

NORTEL
NORTHERN TELECOM

P0875934

Meridian ACCESS

Developer's Guide

Publication number: 555-7001-316
Product release: 12
Document release: Standard 1.0
Date: January 1998

Publication history

January 1998

Guide released as Standard 1.0. No technical changes were necessary for Release 12. This edition makes all previous editions obsolete.

August 1995

Guide released as Standard 1.0. This version documents release 10.0 of Meridian Mail and release 2.0 of Meridian ACCESS. This edition makes all previous editions obsolete.

March 1994

Guide released as Standard 1.0. This version documents Release 9 of Meridian Mail. This edition makes all previous editions obsolete.

Contents

Figures

Tables

Procedures

About this guide

This guide describes Release 2.0 of Meridian ACCESS for the applications developer. It provides a brief introduction to the Meridian Mail system and to the Meridian ACCESS package.

Related documentation

This guide introduces the developer to the implementation of voice and telephony applications with Meridian ACCESS, and is not intended to provide detailed information on Meridian Mail, Meridian ACCESS-related tools on Meridian Mail, or the actual programming functions provided by Meridian ACCESS. For these details and other related information, see the following documents.

Meridian Mail General Description (NTP 555–7001–100)
Offers general information on the Meridian Mail architecture and voice messaging application. Describes Meridian Mail features and voice services.

Meridian Mail Site and Installation Planning
(NTP 555–70x1–200**)**
Assists in selecting and planning the Meridian Mail hardware installation site.

> ***Note:*** The "x" in the NTP numbers represents the applicable hardware platform. "X" can be "1" for Options; "4" for Modular Option; "5" for Modular Option GP; or "6 "for Modular Option EC.

Meridian Mail System Administration Tools (NTP 555–7001–305)
Describes tools available to the Meridian Mail system administrator, including Meridian ACCESS-related tools.

Meridian ACCESS Configuration Guide (NTP 555–7001–315)
Outlines Meridian Mail and PBX configuration.

Meridian ACCESS Application Programming Interface (API) Reference Manual (NTP 555–7001–317)
Provides full details on all of the Meridian ACCESS functions and options.

Meridian ACCESS Voice Prompt Editor User's Guide (NTP 555–7001–318)
Details the creation, maintenance and use of voice segment files by the Meridian ACCESS developer.

About the developer

The Meridian ACCESS developer is responsible for providing custom applications and should have basic knowledge of "C" programming, real-time applications, and the UNIX operating system.

These areas are not described in this guide, and suitable training is recommended before you begin developing applications.

In this guide

This guide provides information on how to use the Meridian ACCESS product to provide custom applications. It contains the following chapters:

Meridian ACCESS system overview
Describes the Meridian ACCESS "system" (which includes the Private Branch Exchange, Meridian Mail, and Meridian ACCESS) and introduces the major concepts involved in applications.

Installing Meridian ACCESS
Outlines all of the requirements that must be met before Meridian ACCESS can be installed, describes the installation procedure, and explains how to test the installation by executing a sample program.

Meridian ACCESS program development
Describes the Application Programming Interface (API) and API functions and provides event and call handling information.

Development guidelines
Provides guidelines for user interface, application design, and engineering, as well as tips on programming applications.

System configuration
Provides setup information for the user's environment and describes the tools available for use with Meridian ACCESS.

Engineering Guidelines
Provides information on engineering consideration for those developing ACCESS applications.

Telephony
Provides information on call states and events generated.

Appendix A: Sample program listings
Provides listings for the sample programs discussed in the document.

Chapter 1: Meridian ACCESS system overview

Meridian ACCESS allows customers or third parties to develop and maintain their own telephone-based voice applications. The most common application is an Interactive Voice Response (IVR) system where a telephone caller places a call, and the system provides information or places orders for the caller based on the caller's input on a tone-generating telephone.

Meridian ACCESS uses the UNIX development system as the application engine. UNIX is also capable of communicating with almost every other computer currently used, thus serving as an excellent bridge between the telephone network and pre-existing applications or databases.

By providing a link from a UNIX workstation to Meridian Mail and a telephone switching system (Private Branch Exchange or PBX), Meridian ACCESS gives the applications developer the full range of voice and telephony functions that a digital voice processing system and a telephone switching system can offer. The workstation does not need special voice and telephone interface cards or large hard disks for voice storage; the PBX and Meridian Mail provide all of these resources in a reliable, high-quality system. To use these resources, the application simply invokes routines in a supplied C-language software library.

Note: This guide refers to Release 2.0 of Meridian ACCESS.

Types of applications

Incoming call (inbound) applications

Applications that provide a service to callers who dial in are inbound applications. Callers dial from either an internal telephone (an extension on the PBX) or external telephone (a payphone or home telephone, for example) to the service, which is often an IVR service.

Outgoing call (outbound) applications

Applications that call internal or external telephone numbers are outbound applications. The application requests Meridian Mail to initiate an outgoing call and provides a service (IVR or call processing function) to a customer.

Administrative applications

Applications which do not take incoming calls or place outgoing calls are generally administrative in nature. Popular examples are electronic mail notification and faxing. Meridian ACCESS can be used to send summaries of voice messages to a host computer and can receive notification of text messages (and turn on Message Waiting Indication at a telephone set).

Desktop messaging applications

Meridian Mail supports Desktop Messaging applications such as VISIT Messenger. For such applications, ACCESS exports the Meridian Mail Voice Messaging capabilities to the desktop, such as support for displaying and maintaining lists of voice messages and for voice messaging functions like play, compose, and send.

Meridian ACCESS system concepts

This chapter describes how the PBX, Meridian Mail, and Meridian ACCESS function together to provide customized voice service applications.

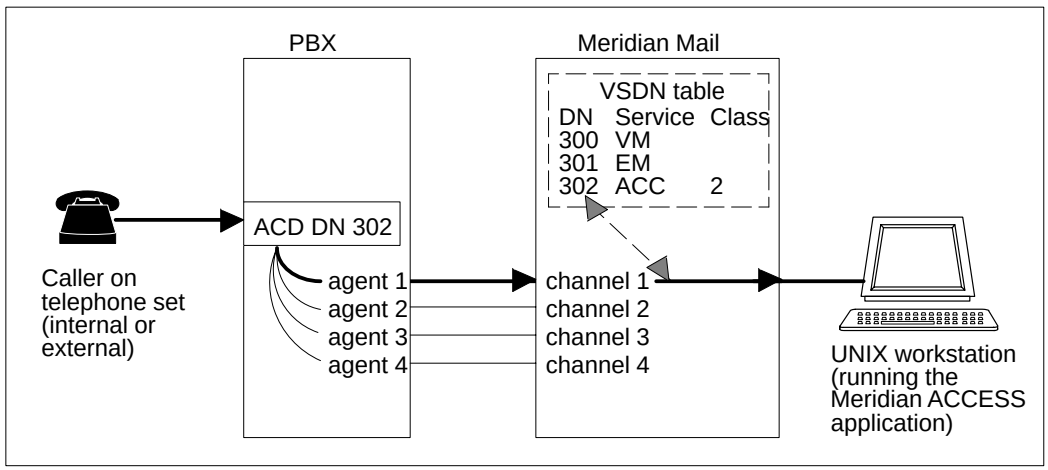
In order to plan and configure a Meridian ACCESS system, you must understand some basic concepts. Figure 1-1 illustrates the components of a system.

Information on PBX and Meridian Mail setup can be found in the Meridian ACCESS *Configuration Guide* (NTP 555-7001-315).

Private Branch Exchange

The Private Branch Exchange (PBX) provides the telephone interface to Meridian ACCESS voice services. For M1 systems, the Automatic Call Distribution (ACD) feature allows a number of “telephones” connected to the PBX (known as agent positions) to share equally in answering incoming calls. Incoming calls to an ACD Directory Number (DN) are placed in an ACD queue and presented to the available agents on a first-in, first-out basis. Other switches use similar queuing schemes.

Figure 1-1
Meridian ACCESS application overview in an M1 configuration



In Meridian 1 systems, Meridian Mail uses ACD to receive calls from users who have dialed a voice service telephone number or directory number known as the VSDN, which is also the ACD DN of an ACD queue. Calls are distributed to agent positions which correspond to voice channels on Meridian Mail. Inbound applications handle calls that originate outside the PBX. A call arrives on a trunk that terminates on an ACD queue.

Meridian Mail is usually configured so that all incoming calls share all available agents/channels. Therefore, calls are evenly distributed. Shared channel configuration involves one “primary” ACD DN queue that can direct calls to all configured agents. Some voice services use ACD DN queues that do not have agents assigned to them. These “virtual” ACD DN queues forward incoming calls for certain DNs to the primary ACD DN queue using the agents assigned to the primary DN.

In some cases, however, you may decide to use a dedicated channel configuration in order to allocate channels to a specific ACCESS application. For example, any incoming calls for the specified application would funnel into a primary ACD queue for DN 3311. Four agents would comprise the queue and correspond to four voice channels on Meridian Mail.

More on shared versus dedicated channels

The decision to use a shared or dedicated channel configuration depends on the specific requirements of the application. Often the application developer will specify which configuration to use. The shared channel method uses channels more efficiently in terms of channel utilization; however, the channel generates more Meridian ACCESS link traffic and Meridian Mail system load. The dedicated channel method reduces some application overhead, and is the best method for systems using a single application and no other voice services.

Generally, the configuration for shared incoming channels is performed on the PBX, and the configuration for shared outgoing channels is performed on Meridian Mail (as described below).

Meridian Mail

Meridian Mail provides all basic voice service capability to the Meridian ACCESS system. It stores all voice recordings (on a hard disk) and gives callers access to features like voice messaging (which in itself has a long list of available features), and voice menus and announcements. These features can be customized to meet the needs of a wide range of users. Digital-compression capability enables Meridian Mail to convert sound from analog to digital form (in a manageable file size) so that recorded messages can be transferred easily between mailboxes and across computer links.

Call processing

Incoming calls to a voice service (such as a Meridian ACCESS application) arrive on Meridian Mail channels according to ACD configuration as described above. Although a call may have been forwarded to another ACD DN and then reassigned to an agent position, it still retains information about the originally dialed DN.

Meridian Mail interprets this DN according to the Voice Service-DN (VSDN) table. The VSDN table lists all voice service DNs and type-of-service information on each DN. For more information on call processing and the VSDN table refer to Chapter 7 “Telephony.”

VSDN table

Every voice service has a DN associated with it; when this DN is dialed, the call is passed to Meridian Mail. Meridian Mail starts up the appropriate voice service by looking at the VSDN table entry for that DN.

The VSDN table entry for a Meridian ACCESS application contains three pieces of information: the DN, service type (ACCESS), and class. A service type of ACC indicates to Meridian Mail that the call should be passed to Meridian ACCESS. Every Meridian ACCESS application has a unique class number; the class indicates which application should be run. If the originally dialed DN corresponds to a Meridian ACCESS application, it will start the application for that class number.

Channel Allocation Table

The Channel Allocation Table (CAT) contains entries for each voice channel on Meridian Mail and matches these channels to ACD agents on the PBX. This table enables you to dedicate channels to a particular service on Meridian Mail or to a particular ACCESS class. As well, you can make the channels available to all services or ACCESS classes.

When you dedicate channels in the CAT to a service or an ACCESS class, Meridian Mail cannot allocate those channels to any other service or ACCESS class, nor can the service or the class use any other channel. If you do not dedicate particular channels on the PBX (using a separate ACD queue as described earlier), any service can use the channels on incoming calls. The CAT only controls the resources allocated by Meridian Mail. For more information about CAT, see the *Meridian ACCESS Configuration Guide* (NTP 7001-555-315).

Meridian Mail Channels

There are three types of channels in Meridian Mail: Multimedia ports, Full Voice ports, and Basic Voice ports. If there is a mix of ports on Meridian Mail, Meridian ACCESS will only be able to use basic voice ports that are configured as either ALL or ACC in the CAT. A Basic service will only be able to use a Full Voice port if there are no Basic Voice ports in service, and a Multimedia port can only be used if there are no Full and Basic voice ports in service.

Mailboxes

Most Meridian ACCESS applications require a Meridian Mail account (or mailbox) to store voice files. A single mailbox can be shared by a number of applications and must be shared if the applications use the same voice files. It may be useful, however, to have different applications use different mailboxes.

Mailboxes can be customized in a number of ways to suit a Meridian ACCESS application; space requirements for voice files must be taken into account, message waiting indication can be enabled or disabled (if there is no telephone number associated with the mailbox), and message retention information can be modified (sent messages can be retained or deleted automatically, read messages can be retained for a designated period of time).

Meridian ACCESS only supports eight-digit mailboxes. Consequently, you should not use more than eight-digit mailboxes in your application.

However, to maintain Meridian Mail compatibility with 18-digit mailboxes, a Meridian ACCESS application will be able to compose and send messages to a user with an 18-digit mailbox. Mailbox information that is returned in an application programming interface (API) will also be correct up to 18-digits.

Cabinets and files

For a Meridian ACCESS application to play or record voice, the application must access voice files. These voice files reside in Meridian Mail cabinets (which are equivalent to directories). There is a unique cabinet associated with each Meridian Mail account (mailbox). Each subscriber to Meridian Mail has an account and a cabinet.

To obtain access to an account’s cabinet, the application must log in to the account using the account number and corresponding password. This provides simple but effective security, and also allows applications to act as users of the Meridian Mail voice messaging system. Multiple Meridian ACCESS sessions can log in to the same account, thus sharing all the voice files stored in the cabinet.

Meridian ACCESS provides a variety of voice storage formats. There are three kinds of voice files: the simple voice file, the voice message file, and the voice segment file.

All voice files have names. Meridian ACCESS supports the association of a text string with the file allowing the subject of the file or other useful information to be stored with it.

Simple voice files

The simple voice file is used to store a single piece or segment of voice. The segment can be of any length, from a single tone to a conversation. The simple voice file can be recorded and played back but cannot be delivered to other users via the Meridian Mail voice messaging system. The simple voice file is the most efficient way of storing a single segment of voice.

An application can skip forward and backwards while playing a simple voice file. If concatenation of voice files is required, then the simple voice file type is not recommended in most cases. Each concatenated file must be opened and played separately which may result in an unacceptable delay between hearing the end of one voice file and the start of the next one. The voice segment file type is more suitable and recommended for this purpose although an application cannot skip forward or backwards within a list of segments or within individual segments.

Voice message files

The voice message file is like the simple voice file except that it can be addressed and submitted to the Meridian Mail voice messaging system for delivery. Since it provides for the additional addressing data required to send a message, a voice message file is slightly larger than a simple voice file with the same voice content.

Voice segment files

A voice segment file is a single file containing zero or more voice segments. A voice segment (which is typically less than a minute in length) consists of recorded voice, a name, a title, and a text field usually used to store the script of the voice. Voice segment files are primarily used when increased performance is needed in the playback of prompts, and are usually concatenated to produce full-length prompts.

If you require concatenation of a number of prompts in quick succession, as in dates, numbers, or amounts, then a voice segment file is the best voice file type to use. Voice segment files may also be useful for increasing performance even when concatenation is not required since they eliminate the requirement of opening multiple simple voice files.

A Voice Prompt Editor (VPE) is provided with Meridian ACCESS. This editor allows a user to create and modify voice segment files, and to record individual segments and edit corresponding text fields. VPE enables a user to associate text, such as titles and scripts, with the recorded voice of a voice segment. For further details on the use and capabilities of the VPE, consult the *Voice Prompt Editor User's Guide* (NTP 555-7001-318).

Call Billing

The Call Billing feature provides the ability to bill an outgoing call to a particular customer. Meridian ACCESS uses the Customer Billing DN if it is specified for billing purposes. If the multi-customer option is installed, there will be one Customer Billing DN for each customer. The billing DN can be found in the Voice Messaging Options screen in the "Voice Administration" chapter of the *System Administration Guide* (555–70x1–30x).

Operational Measurements

Meridian Mail's AdminPlus Enable option provides Operational Measurements (OM) file transfer from Meridian Mail to a PC running the Meridian Mail Reporter (MMR) application. The MMR data link supports Local Billing. The local billing file contains all of the local OM billing data such as all Logon, Call Answering, and Express Messaging sessions that occurred.

The AdminPlus Enable option also will transfer ACCESS OM data. The LOCBIL file request will contain all ACCESS sessions that were collected. Data for only one ACCESS class can be downloaded at a time. For more information, see the *Meridian Mail Reporter User Guide* (P0847870)

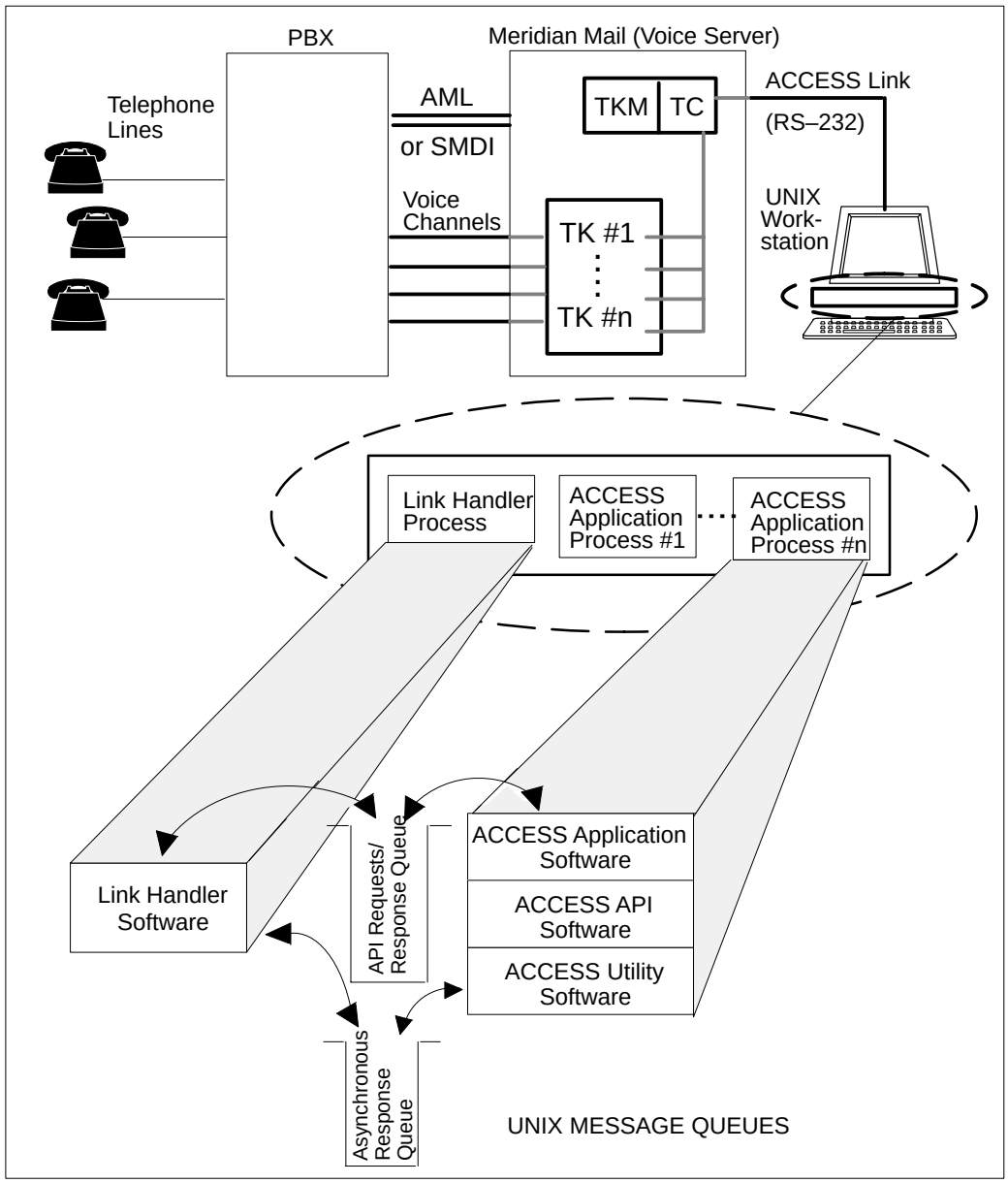
Agent Whisper

The Agent Whisper feature will allow a prompt to be played to an agent while a call is currently on hold. This is achieved through the use of the m_AddOnCall API.

VMUIF Mailboxes

VMUIF is set on a customer basis, and it is possible to add a VSDN entry for Meridian ACCESS in a VMUIF customer. However, VMUIF mailboxes are not supported through Meridian ACCESS.

Figure 1-2
Meridian ACCESS software architecture



Meridian ACCESS

Applications

Meridian ACCESS applications can be developed to meet a wide variety of requirements; an application can receive or place telephone calls, play prompts, receive input in the form of digitone keypresses (which can be interpreted as commands or data), transfer calls, record messages, and use Meridian Mail services. All of these functions can be built into a voice service that is tailored to meet special requirements.

Each different Meridian ACCESS application on a system can be treated as a separate voice service by assigning a unique class number to the application. As discussed earlier, the VSDN table on Meridian Mail uses the service type of ACC and this class number (which may be predetermined by the application developer or designed to be configurable by an administrator) to call up the appropriate application.

A typical application program would start by registering with the Meridian ACCESS system. It would then acquire a Meridian ACCESS session, and then log on to a voice storage account on the Meridian Mail system. Once these steps are completed, all the other Meridian Mail functions supported by Meridian ACCESS can be performed.

There is a one-to-one mapping between voice channels and active application channels.

Link Handler

Meridian ACCESS communicates with Meridian Mail over the Meridian ACCESS link which is controlled by the Link Handler. The Link Handler sends and receives packets on the link and ensures that Meridian Mail messages are delivered to the appropriate application process using message queues. The Link Handler supports 19.2 kbyte/s. There must be one Link Handler running for each ACCESS link configured.

Using multiple links involves using the default link or a specific link. You specify the link that you want to use by using an Application Programming Interface (API) command or by setting an application environment variable. All links must be defined in the lh.config file. For more information about the Link Handler, its lh.config file, multiple links, and the API command, see Chapter 5 of this guide.

Message queues

UNIX message queues are the mechanism used to pass packet information to or from,the Link Handler (lh) and the application processes. There are two message queues that are created and maintained by the Meridian ACCESS system. One message is used for API requests and responses, and the other message is used solely for handling asynchronous events from Meridian Mail to an application.

The unique number that identifies a UNIX message queue is called a “key” and is derived from the Link Handler’s “base key” unless overridden. If you explicitly define a key’s value in the lh.config file, the lh uses that value to create the queue; if you do not define a key’s value in the lh.config file, the lh derives a key from the base key to create a queue. For more information about the Link Handler and its configuration file, see Chapter 5 of this guide.

The Application Programming Interface

The application program, written in C, uses the Meridian ACCESS Application Programming Interface (API) for all voice and call processing functions, completely isolating the actual voice and telephony systems from the program. The application program can also deal with system calls or other software libraries for accessing local or remote data.

The Meridian ACCESS API is the library used by the application. The functions contained within the library handle the details of Meridian ACCESS protocols for the application program. These functions issue voice and call processing requests to the Meridian Mail system and inform the application program of relevant voice and call processing events. API functions are the building blocks of an application.

The following types of functions are provided by the Meridian ACCESS API library:

- local functions
- Link Handler functions
- resource management functions
- telephony functions
- file access functions
- voice operations functions

- messaging functions
- voice segment file functions
- user administration functions
- event handling functions
- high-level functions

API functions are discussed in more detail in Chapter 3 of this guide.

Chapter 2: Installing Meridian ACCESS

This chapter contains all of the information required to install the Meridian ACCESS software, and provides references to further information related to installation activities. The following items are discussed:

Pre-installation information
Describes the system requirements that must be met before Meridian ACCESS installation can be completed, including Meridian Mail, the PBX, and the Motorola workstation.

Installing from distribution tape
Details the actual software installation procedure, and illustrates the Meridian ACCESS software directory structure.

Post-installation information
Describes installation activities that are performed once the Meridian ACCESS software is installed. These include

- Reference information for Meridian Mail and PBX setup (see the *Meridian ACCESS Configuration Guide* (NTP 555-7001-315) for more information regarding the PBX setup)
- Reference information for tuning the UNIX kernel (see Chapter 5, “System configuration”)
- Identifying the link ports
- Running the diagnostic tool to verify the software installation
- Making and executing prepsample to verify the functionality of your Meridian ACCESS system
- Running the Voice Prompt Editor, if necessary

Pre-installation information

Before configuring your system for a new Meridian ACCESS application, ensure that the system meets the requirements listed below.

Meridian Mail

Your Meridian Mail system must

- be fully installed according to the *Meridian Mail Installation and Maintenance Procedures* (NTP 555-70x1-210) for your system
- have the Meridian ACCESS feature enabled (see the *Meridian Mail System Options Guide* (NTP 555-7001-215) for details)
- have sufficient voice ports (channels) installed
- have sufficient storage (disk space) for voice prompts and messages
- have a dataport available for Meridian ACCESS

  ACCESS supports basic transmit, receive, and ground pins (pins 2, 3, and 7). For Card Option systems, all dataports on the RS–232 Service Module (RSM) card are configured as Data Terminal Equipment (DTE). For Modular Option Enhanced Capacity (ModOpEC) systems, all dataports are configured as DTE on both the utility pack and the Single Board Computer (SBC) card. For Options NT/XT (hardware in the Cantilever/CenterMount Peripheral Equipment shelf packaging), Options ST/RT (hardware in the SL-1 Tier In-Skins packaging), Modular Option (ModOp), and Modular Option General Purpose (ModOpGP) systems, all dataports are configured as DTE on both the RSM card and the SBC card.

  On Message Services Module (MSM) systems, you require the NTGX06AB transition module for an ACCESS port on a SPN node. The NTGX06AB has serial ports configured as DTE.

  When connecting to the Motorola Delta series workstation, serial ports on the 332XT I/O card are configurable as DTE or Data Communications Equipment (DCE). The Motorola also has two, nine-pin SBC ports configured as DTE. When connecting DTE to DCE devices, you use a straight-through, 25-pin serial cable. You will require a null modem for a DCE-to-DCE or a DTE-to-DTE configuration.

PBX

Your Private Branch Exchange (PBX) must

- meet the requirements for Meridian Mail as specified in *Meridian Mail Site and Installation Planning* (NTP 555-70x1-200)
- be fully configured for Meridian Mail according to *Meridian Mail Installation and Maintenance Procedures* (NTP 555-70x1-210)
- have all voice ports configured
- have sufficient incoming and outgoing trunks and line cards installed and configured

UNIX workstation

The workstation must

- be a Motorola Delta Series V/68 workstation running UNIX 5.3 Release 3 version 6.2 or Release 3 version 7.1
- be connected to one or more dataports on Meridian Mail

Installing from distribution tape

The installation program provided with each Meridian ACCESS software release will perform the following tasks:

- Make an “m1access” group ID on the target system, if one does not already exist.
- Make an “m1access” user ID on the target system, if one does not already exist. This will be created as “NOLOGIN” account.
- Create the necessary sub-directories in the “m1access” home directory.
- Install all Meridian ACCESS files in their appropriate sub-directories, and give them appropriate permissions.

The following Meridian ACCESS components will be installed on the target system:

- API function library
- Include files
- Link Handler files
- Diagnostic tool executable

- Voice Prompt Editor executable
- Sample program preparation executable
- Sample programs

Procedure

The procedure below describes the steps involved in installing Meridian ACCESS onto your system from tape. All steps should be followed in the order in which they appear below.

Procedure 2-1
Installing Meridian ACCESS

1 Log on to your system as “root”.

2 Use the UNIX “tar” command to extract all files from the release tape into some temporary directory (/tmp, for example).

Your command may look something like this:

tar xvf /dev/tape

Note: Software release files should NOT be placed in the root (“/”) directory, or the “m1access” user home directory, if one exists.

3 Verify that you have files called “install” and “m1_access” in your local directory, and that “install” is executable.

4 Type **./install** on the UNIX command line, and press <Return>.

If you have previously created an “m1access” user and group, go to step 9.

If not, you will be asked if you wish to create one now.

Your system does not currently have an ‘m1access’ group id.
Do you want to make an ‘m1access’ group now [y/n]:”

If you do not wish to make an “m1access” group now, enter **n**.
Installation of the Meridian ACCESS software release will terminate.

5 Type **y**, and press <Return>.

The install program will ask if you want to use the default group ID (the default is the lowest available number).

Use the default group id for the ‘m1access’ group? [default =]? [y/n]:

If you want to use some number other than the default, enter **n**. The following prompt will appear:

Enter group ID:

6 Enter the number you wish to assign to the “m1access” group.

If an “m1access” user ID does not exist on your system, you will be asked if you want to create one now.

The user account created will be a “NOLOGIN” account (i.e., nobody can login as “m1access”). It may be changed to a login account after the installation is complete by editing the “/etc/passwd” file.

Your system does not currently have an ‘m1access’ user id.
Do you want to make an ‘m1access’ user now? [y/n]:

If you do not want to make an “m1access” user now, enter **n**. Installation of the Meridian ACCESS software release will terminate.

As with the group ID above, you will have the option of using the default user ID:

Use the default use id for the ‘m1access’ user? [default =] [y/n]:

or assigning some other number:

Enter user id:

7 Enter the user ID you wish to assign to “m1access”.

You will now be asked to enter the “m1access” user home directory.

This is where all Meridian ACCESS software release files will be placed on your system.

Enter home directory for ‘m1access’ user (Hit return to choose default home directory = /usr/m1access):

8 Enter the full pathname of the directory to be used as the “m1access” home directory.

You will now be asked to verify your selection.

Home directory chosen is [*your selected directory*]? [y/n]

If you enter **n**, you will be prompted again for a home directory for “m1access” (see step 7).

If you enter **y**, you will be prompted to confirm the installation path.

Do you want to install ACCESS under Home directory [your *selected directory*]? [y/n]

9 Enter **y** to continue, and press <Return>.

If you enter **n**, you will be prompted again for a home directory for "m1access" (see step 7).

10 Wait while the install program installs the Meridian ACCESS software files.

If a set of release files already exist, you will be prompted with

Overwrite existing Meridian ACCESS files [y/n]:

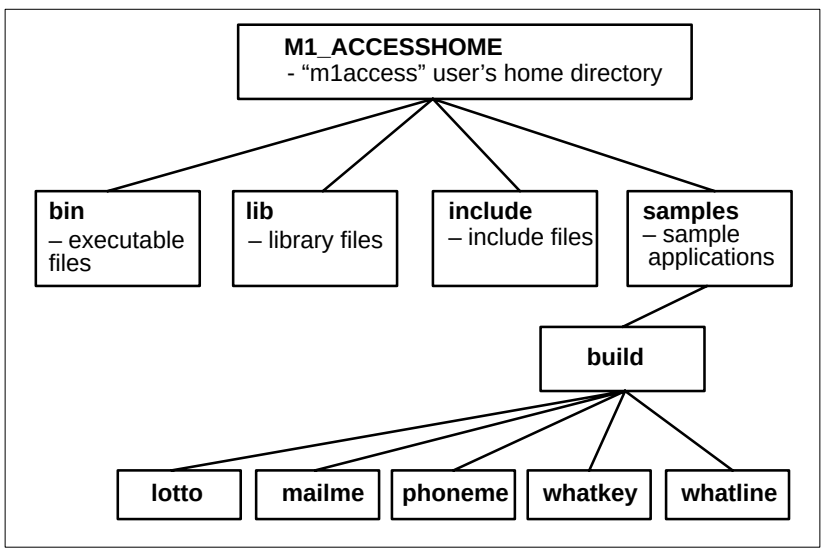
If you enter **y** the existing files will be overwritten, otherwise the installation program will terminate.

Meridian ACCESS is now fully installed on your system.

Meridian ACCESS directory structure

The following directory structure will be installed in the "m1access" user's home directory (from a Meridian ACCESS software release tape).

Figure 2-1
Meridian ACCESS directory structure



Each of the "samples/build" subdirectories contain C source, specific include, and "make" files for the Meridian ACCESS sample program with the same name.

Post-installation information

Meridian Mail and PBX setup

Configuration information for Meridian Mail and the PBX varies widely depending on the needs of each applications. For information on this subject, refer to the *Meridian Mail ACCESS Configuration Guide* (NTP 555-7001-315).

UNIX kernel tuning

Some applications place heavy demands on your system resources, and you may find that your UNIX kernel requires slight modifications to accommodate these demands. See Chapter 5, “System configuration,” for details.

Identifying the link port

The lh.config file identifies the ports on your Motorola Delta V/68 workstation that will be connected to the ports on Meridian Mail. This file is located within the $M1_ACCESSHOME/bin directory and is exclusively used by the Link Handler. The Link Handler uses this file to open the physical port device and configure its RS-232 characteristics (see Chapter 5, “System configuration,” for details). It is essential that the lh.config file be modified for the Link Handler to function properly.

Procedure 2-2
Identifying the link port

1. Log on to your system as “root”.
2. Change your working directory to the access bin directory by typing:

   cd $M1_ACCESSHOME/bin
3. Edit lh.config using a text editor.

   If you are using the vi editor, for example, type in the following command:

   vi ./lh.config
4. Change the line **Device=/dev/tty** to reflect your designated link port (for example, **Device=/dev/tty15**).
   For systems configured with multiple links, use the following convention:
   Device1=/dev/tty15
   Device2=/dev/tty16

   The number that you assign must be between 1 and 8 and must be passed to the link handler as an argument when it is started.

5 Save your changes.

6 Include a tty device file for your designated link port in the /dev directory. Ensure that the permissions for the tty device file allow read and write access for the Link Handler process. If a tty device file does not exist for your link port, use the UNIX command “portconfig” to create one and the “chmod” command to set the proper permissions. Consult a UNIX manual if you need more details on how to use these commands.

7 Check the /etc/inittab file to make sure there is no “getty” process running against your designated port.

If there is an entry against the designated port in the /etc/inittab file, remove it.

8 Ensure that the cables between your Motorola Delta V/68 workstation port and the dataports on Meridian Mail are securely connected.

9 Execute the Link Handler process by typing the following command:

./lh > /dev/console &

For more information (see Chapter 5, System Configuration).

Executing the Meridian ACCESS diagnostic tool

The Meridian ACCESS diagnostic tool checks whether certain critical software components are functional and, if not, attempts to determine the cause of any problems found.

Procedure 2-3
Executing the Meridian ACCESS diagnostic tool

1 Log on to your system as “root”.

2 Change your working directory to the access bin directory by typing the following:

cd $M1_ACCESSHOME/bin

3 Execute the Access Diagnostic tool by typing **./accdiag** and pressing <Return>.

The tool will display a status screen of the critical Meridian ACCESS components.

If the diagnostic tool finds any of these components non-functional, it will diagnose the problem and will recommend any corrective action. By default, accdiag will only look for link number 1. To view information on another link start accdiag and pass in the link number, for example, ./accdiag -n 2

4 Exit the diagnostic tool using the appropriate function key.

The installer should not attempt to continue with the remaining post-installation activities unless all critical Meridian ACCESS components are functional.

Making and executing prepsample

"Prepsample" creates the voice files that are needed by the sample programs, PHONEME, MAILME, WHATKEY, WHATLINE and LOTTO. These sample program are described in the chapters that follow to emphasize key Meridian ACCESS concepts.

Executing "prepsample" will perform a series of functions confirming that major components of Meridian ACCESS are working in conjunction with Meridian Mail and the PBX.

To run "prepsample" you must have the following items:

- Meridian Mail account

  Note: It is possible to use an existing user's account, but an account dedicated to Meridian ACCESS development is recommended.

- a free Meridian ACCESS voice channel
- a touch-tone telephone set connected to your PBX

Procedure 2-4
Making and executing prepsample

1 Log on to your system as "root".

2 Change your working directory to the access bin directory by typing the following command:

cd $M1_ACCESSHOME/bin

3 Create the sample voice files by typing **./prepsample** and pressing <Return>.

"Prepsample" will prompt you to enter the DN of your telephone set, and MM account/password where the sample voice files will reside.

4 Enter the DN, account number, and password when prompted.

"Prepsample" will register with Meridian ACCESS, acquire a voice channel, and logon to designated MM account. At this point, you will have the option of creating all or any specific sample program voice file.

When creating a voice file you will be prompted to record a number of voice segments using your digital telephone set. The only exception to this are the voice segments for the "lotto" sample program. Only the text for these voices segments is created, and the actual voice must be recorded using the "vpe" (Voice Prompt Editor). "Prepsample" can be executed as many times as necessary.

Running the Voice Prompt Editor

If you selected the option to create the voice files for "lotto" in prepsample, you will need the "vpe" to complete the voice segment recordings (see the *Voice Prompt Editor User's Guide* (NTP 555-7001-318)).

Before you can execute "vpe" you must ensure that your UNIX shell's environment TERM variable is set to a VT220 compatible type terminal. See the *Voice Prompt Editor User's Guide* (NTP 555-7001-318).

Chapter 3: Meridian ACCESS program development

General format of an API function

All Meridian ACCESS API functions begin with “m_”, return TRUE or FALSE to indicate whether the function was able to complete the desired operation, and return a status code “rc” indicating the nature of any unexpected states or failures during execution.

General format of an application

As mentioned earlier, there are a few API functions that all Meridian ACCESS applications must execute in order to perform various functions. As a result all voice processing applications are based upon the following framework:

Process initialization

This includes registration with the Link Handler and acquisition of a Meridian Mail voice session (channel). It may also include event handler installation, logging on to a Meridian Mail account, manipulation of various voice file (excluding playing and recording of prompts), and other local processing such as checking the version of the Meridian Mail server.

Call establishment

This involves placing an outgoing call or waiting for an incoming call.

Call processing

This may involve some interaction with caller or called party through DTMF, eventually leading to call management (for example, call transfer), and playing or recording of voice prompts.

Process shutdown

This includes closing files, logging off, release of the Meridian Mail voice session, and deregistration from the Link Handler.

Table 3-1
Meridian ACCESS resources

Resource	Acquired by	Released by	Prerequisite for
ACCESS session	Register	Deregister	All ACCESS commands
MM session/Voice Channel	Acquire or AOIC + incoming call	Release	All MM commands (except ENS commands)
MM file system	Log on	Log off	All MM commands using files
Active call	Incoming call + answer/make call	Disconnect	All DTMF, record, play commands
ENS	Acquire ENS	Release ENS	ENS notification
Note: This table illustrates hierarchy involved in acquiring various Meridian 1 ACCESS resources.			

Registering

All Meridian ACCESS application processes must register with the Link Handler before they can communicate with the Meridian Mail server. This registration identifies the process to the Link Handler so that events (events will be discussed later in this document), and responses to commands may be directed to the correct process. The m_Register() function establishes a connection between a process and the Meridian ACCESS Link Handler.

Acquiring a Voice Session (Channel)

Just as application processes must register with the Meridian ACCESS Link Handler, processes that utilize Meridian Mail resources must acquire a voice session, or channel from Meridian Mail. Once a process is acquired it may execute most Meridian ACCESS API functions that do not involve playing or recording voice prompts (these require that a call be established) and may begin to use Meridian Mail resources.

A voice session (channel) will be considered idle if it is inactive (that is, the process does not communicate with Meridian Mail) for a period of one minute, and Meridian Mail will terminate the session. When this occurs, the associated process will be sent a SessionEnd event. (See section on event handling.)

There are two functions that may be used to acquire a voice session with Meridian Mail: m_Acquire() and m_AcquireOnIncomingCall().

m_Acquire

m_Acquire dedicates a voice channel for Meridian ACCESS, since ACCESS may require channels to be configured to ACC in the Channel Allocation Table (CAT). The only Meridian Mail service that may use such a channel is Meridian ACCESS. This acquisition method proves useful when system resources allow channels to be dedicated for an ACCESS application. As well, this acquisition method is necessary for applications not receiving incoming calls (for example,, outdialing or system administrative Meridian ACCESS applications).

m_Acquire allows an application process to acquire a voice channel based upon its class. Since classes may be set on a specific voice channel basis, the application process can recognize which voice channel it has acquired.

m_AcquireOnIncomingCall

m_AcquireOnIncomingCall (AOIC) allows voice channels to be used for services in addition to Meridian ACCESS, since channels may be configured with an ALL service type in the CAT. AOIC should be used for Meridian ACCESS applications that wait for incoming calls (for example, IVR applications). These applications do not always need to perform Meridian Mail operations, which require a session and a voice channel, until after a call has arrived.

The service type used depends upon the service selected in the Voice Service Directory Number (VSDN) table. A Directory Number (DN) is associated with every voice service. When a DN is dialed, the call passes to Meridian Mail. A service type of ACC indicates to Meridian Mail that the call should be passed to Meridian ACCESS. Incoming calls to a Meridian ACCESS-based application arrive on Meridian Mail channels according to Automatic Call Distribution (ACD) configuration. ACD retains information about the originally dialed DN. Meridian Mail interprets this DN according to the VSDN table. The VSDN table lists the DN, service type (ACCESS), and class. Every Meridian ACCESS-based application has a unique class number, which indicates the application that should be notified to handle the call. If the originally dialed DN corresponds to a Meridian ACCESS-based application, that application will be notified and, therefore, handle the call.

AOIC allows an application to wait for calls. Furthermore, AOIC allows channels to be used for services besides ACCESS.

Basic error handling

The function value returned by calling a Meridian ACCESS function should be checked by the calling process. This value should be TRUE if the function executed successfully. If the function returns FALSE, then the “rc” error code parameter should be examined to determine why. Depending upon the reason for the failure, the function may want to be retried. If it is determined that the function should not be retried (or the retry results in a subsequent failure), it should be decided whether the error was serious enough to result in bringing down the call session and whether the Meridian ACCESS session should be brought down as well.

Problems that affect individual call sessions are those where the caller and the application are somehow related (for example, a unique mailbox may be opened for that caller), or spurious in nature. If it is determined that the call session should be brought down, then a prompt informing the caller of this should be played if possible. The problem should be resolved (if possible) before servicing additional callers. A call session problem may escalate to an Meridian ACCESS session problem if the problem is observed on the next call.

A method of indicating hardware problems (for example, a bad Meridian Mail voice channel card) should be built into the application. This information should be conveyed to the application system administrator.

Meridian ACCESS session problems are those where there are problems with the fundamental operation of Meridian ACCESS (for example, cannot log into any mailbox) and are not call session related. These problems are usually addressed by determining the application state machine the Meridian ACCESS session is in, performing the corrective actions required to bring it to the previous state (for example, releasing the Meridian ACCESS session if already acquired but cannot log into any mailbox), and proceeding from the previous application state (acquiring, for this example).

Every state in the application process should have code that allows a controlled traversal (usually by performing the complement of the action performed in the previous state, that is, releasing if the previous state was to perform an acquire) to the previous application state. This way the application process should effectively be able to correct itself from any correctable problems.

If the corrective actions are not performed in a controlled manner (that is, the application process is killed prior to releasing the voice channel), the application process may have to wait up to three minutes for the Meridian Mail voice channel to time out since the voice channel is faithfully standing by to serve its application process not knowing that the application has been killed. Restarting the application process in this case still results in a three minute period in which the voice channel cannot be acquired, since the voice channel only knows about the application process by the internal task ID that gets passed to it by the Link Handler. The application process is assigned a different task ID when it re-registers with the Link Handler and therefore appears as another process to that voice channel.

The link handler's status should be verified as being sound prior to performing any harsh actions with the application process. The Link Handler may have to be killed and restarted.

The final actions (if the regression approach described for the application process above does not correct the problem) is to have the application process kill itself and be respawned. Having the system perform very robust error recovery may be able to resolve problems when the application system administrator is not around.

If a problem is not correctable, it should be conveyed to the application system administrator and logged in a log file to assist with developer diagnosis. All the information available about the problem should be logged. It is important to control what is written into the log file, since it should not be swamped with the same error reports and thus possibly have information describing the source of the problem written over if the log file is circular in nature.

The PHONEME sample program

The PHONEME sample program illustrates the basic format of a Meridian ACCESS application, and how most Meridian ACCESS APIs should be used. PHONEME also demonstrates how applications can place calls, manipulate voice files, and play voice prompts. For each execution, it performs the following ten steps to place a call, open a voice file, and play a voice prompt. A complete listing is in Appendix A of this guide.

1 Prompts the user (at the terminal) for the account number and password for the account with the voice file, and for the extension number to be called.

2 Registers with the Meridian ACCESS system. This establishes local resources for the PHONEME application to perform communications with the Link Handler. In order for communication to take place, a message queue must be assigned to this application (determined by the system administrator).

3 Acquires a Meridian ACCESS session and channel. This step requests immediate use of a nondedicated Meridian ACCESS channel on the Meridian Mail system.

4 Logs on to the specified account with the password supplied. This provides access to the voice files in the account.

5 Opens the PHONEME voice file.

6 Establishes the voice connection between the Meridian Mail system and the specified extension. This function will fail if the extension is busy, the extension number is invalid, or the extension remains unanswered for longer than the specified time (MAX_TIME).

7 Starts playback of the PHONEME voice file. The file contains what was recorded with the PREPSAMPLE program.

8 Waits for a keypress on the terminal's keyboard. This keypress allows the user to listen to the voice file before the application goes on.

9 Releases the Meridian ACCESS session. This closes any open files (that is, the PHONEME file), logs off the Meridian Mail server, and disconnects the phone. This also frees the Meridian ACCESS channel for use by another application.

10 Deregisters from the Meridian ACCESS system, freeing the logical connection for use by another Meridian ACCESS application.

Compiling PHONEME

The UNIX make utility should be used on the makefile provided in the samples/build/phoneme directory to compile and link phoneme.c into an executable file called phoneme (that is, execute "make" in the samples/build/phoneme directory). Your machine must have the UNIX development system installed with the appropriate libraries and include files. The makefile for phoneme requires that you define a shell environment variable called M1_ACCESSHOME. This variable directs the make utility where you installed the ACCESS libraries and include files (see "Setting up a user's environment" in Chapter 5).

For example, from your UNIX shell you would type

```
M1_ACCESSHOME=/usr/m1access
export M1_ACCESSHOME
make
```

All the sample programs provided on the Meridian ACCESS release tape may be compiled in the manner described above.

Running PHONEME

PHONEME is a very simple Meridian ACCESS application. While limited in its functionality, it serves well as an introduction to Meridian ACCESS application programming. Subsequent sections introduce the remaining concepts necessary to implement an ACCESS application.

To run PHONEME

1 Run the PHONEME executable file created by the make utility from a UNIX shell.

Note: Before running PHONEME, the PREPSAMPLE program must be run, since PHONEME uses a voice file created by PREPSAMPLE.*PHONEME will prompt you for an account number, password, and extension number.*

2 Use the same account number and password used with the PREPSAMPLE program,

3 Once the number is entered, PHONEME calls that number. When the phone is answered, PHONEME plays the voice prompt. PHONEME then waits for a key press, after which the connection is dropped. The PHONEME application then terminates the session and returns to the shell.

MAILME sample program

MAILME is another sample program that illustrates the use of Meridian ACCESS API functions. The MAILME sample program performs many of the same actions as the PHONEME sample program. In addition, MAILME demonstrates how applications can provide voice messaging features.

For each execution it performs the following steps to place a call, open a voice file, and play a voice prompt, and record and send a voice message. A complete listing is in Appendix A of this guide.

1 Prompts the user (at the terminal) for the account number and password for the account with the voice file, the extension number to be called, and the account number to send a message to.

2 Registers with the Meridian ACCESS system.

3 Acquires a Meridian ACCESS session and channel.

4 Logs on to the specified account with the password supplied.

5 Opens the MAILME voice file.

6 Establishes the voice connection between the Meridian Mail system and the specified extension.

7 Plays a welcome prompt from the MAILME voice segment file.

8 Waits for a keypress on the terminal's keyboard. This keypress signals that the user is ready to begin recording the message to be sent to the specific recipient.

9 Closes the MAILME voice file.

10 Creates a voice message file called CustomerMsg. The recorded message will be placed in this file. Notice that the m_CreateFile function automatically opens the file for writing.

11 Adds the recipients mailbox number to the address section of the voice message file CustomerMsg.

12 Sets the subject field of the voice message file to be the sender's Meridian Mail account number.

13 Begins recording of the message to be sent. Recording will not stop until the user hits a key on the terminal keyboard.

14 Terminates and sends the recorded message (using m_SendMsg()).

15 Releases the Meridian ACCESS session. This closes any open files (that is, the PHONEME file), logs off the Meridian Mail server, and disconnects the phone. This also frees the Meridian ACCESS channel for use by another application.

16 Deregisters from the Meridian ACCESS system, freeing the logical connection for use by another Meridian ACCESS application.

Event handling

While the PHONEME and MAILME examples present important concepts and introduce you to Meridian ACCESS, they do not deal with the concepts involved in making an application handle events such as an incoming call or user input. The ability to handle these asynchronous events is mandatory for all applications which answer calls or accept user input.

Applications must be able to respond appropriately to the full variety of asynchronous events that may occur. For example, when a call arrives at a Meridian ACCESS channel, the application must answer it. When the user presses a digit key, the application must read the digit value and respond. When the user hangs up, the application must prepare itself for the next call.

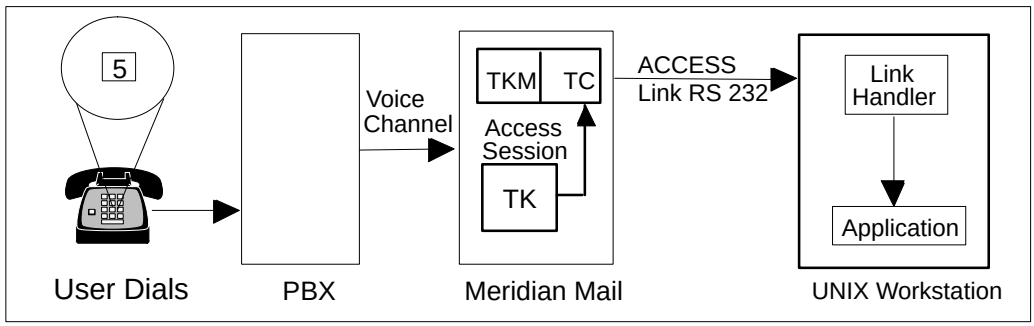
The events which a Meridian ACCESS application may receive are the following:

- Incoming call (call is ringing for a Meridian ACCESS application)
- Call progress (current call has changed state)
- Digit received (user pressed a key)
- Play end (message has finished playing)
- Record end (message prematurely finished recording)
- Error (system error has occurred)
- Block record warning (temporary storage on Meridian Mail is full)
- New message arrival (a new voice message has arrived in the current mailbox)
- Session disconnect (session has been released by the system)

- Timeout (warning of a pending disconnect sent)
- ENS event inbox status

An extensive list of events is included in the Meridian ACCESS *API Reference Manual* (NTP 555-7001-317).

Figure 3-1
Event signaling process



A user presses a digit key on the telephone keypad. The keypress is detected by the Meridian ACCESS session. The Meridian ACCESS session passes the keypress on, as an event, through the Link Handler to the application.

An application which plays a prompt, for example, will request that the prompt be played, and wait for an event. If a Play end event occurs first, then the application knows that the prompt was played to completion. If a Digit received event occurs, then the user has pressed a key while the prompt was playing. The application may choose to stop the playing and wait for additional input, skip back in the prompt and continue playing, or take some other action depending on the specific input received. If a Call progress event occurs and indicates that the user terminated the call, the application stops playing and resets itself to wait for the next call.

The Meridian ACCESS API library allows applications to associate an event handler procedure with each Meridian ACCESS event. Usually, the event handler procedure sets variables that it shares with the application, informing the application that the event has occurred.

Event Handler Installation

Event handler routines are installed by calling the appropriate event handler installation function. The installation functions require one parameter: a pointer to the handler routine to be installed. These functions return a pointer to the previously installed event handler, if any. Figure 3-2 illustrates the usage of an event handler installation function.

Figure 3-2
Sample event handler installation function

```
void DoTelephonyService ()
{
    /* Pointer to existing call progress handler */
    void (*OldCallProgHandler)(int, int);
    .
  .
    .
    /* Replace current call progress event handler with new one. */
    /* Call progress events will be handled by NewCallProgHandler */
    OldCallProgHandler = m_OnCallProgress(NewCallProgHandler);
    .
    .
    .
    /* Restore original call progress event handler */
    m_OnCallProgress(OldCallProgHandler);
}
/* Temporary call progress event handler */
static void NewCallProgHandler(StateChange, StateInfo)
int StateChange;        /* New state which has been entered     */
int StateInfo;      /* Additional information on state change     */
{
    ../* Handle the event */
}/* NewCallProgHandler */
```

Auto event notification

An application may receive events in using one of two methods: auto event notification or manual event notification. Applications may switch between modes using the m_AutoEventOn() and m_AutoEventOff() API library functions. Upon registration with the Link Handler, the default mode will be auto event notification.

An application using auto event notification will be notified of events as they occur; the application is interrupted asynchronously. In the figure above, the application will receive the event immediately. The reception happens through the use of UNIX signals, in particular SIGUSR2. Thus, when the Link Handler receives an event from Meridian Mail, SIGUSR2 will interrupt the appropriate application; the signal handler will execute and, in turn, call the application's event handler, if one is installed.

Note that any application with auto event notification turned on should be prepared to receive events immediately after registering with the Link Handler. It is recommended that event handler routines, for all events that are applicable to an application, be installed prior to registering with the Link Handler.

Manual event notification

Manual event notification allows an application to block receipt of Meridian Mail events; therefore, this notification requires that the application periodically check for events by using the m_EventCheck() function. To use manual event notification, you first must turn off auto event notification (since it is the default) using m_AutoEventOff(). Shutting off auto event notification guarantees that the application will not be interrupted asynchronously. You may prefer to turn off auto event notification for applications in which you can wait for events from the ACCESS link, since manual event notification, which has no signalling, may prove more efficient.

Applications using manual event notification may not receive events immediately. When the Link Handler receives an event (from Meridian Mail), and auto event notification is turned off, the event is queued behind any other events that may have previously occurred (while auto event notification was turned off).

The next time the application calls either m_EventCheck(), or m_AutoEventOn(), all queued events will be processed, and the application's event handlers will be called as required, if installed.

Note that the information returned in the events parameter of the m_EventCheck() function only provides boolean information. Each bit indicates that a particular event has either occurred at least once, or that it has not occurred. Although multiple occurrences of the same event are not reported, the appropriate event handler will be called once for each occurrence.

Applications that use the manual mode of event notification should check for events (call m_EventCheck()) regularly. The information conveyed by the occurrence of most events requires prompt action by the associated application.

The WHATKEY and WHATLINE sample programs discussed below illustrate the use of the some common events. A more complete example of event handling is provided in the LOTTO sample program.

Incoming call applications

A large percentage of voice processing applications involve the servicing of incoming calls. Using Meridian ACCESS, applications process incoming calls using the following sequence:

1 Register with the Meridian ACCESS system.

2 Acquire a voice session from Meridian Mail.

3 Wait for a call to arrive.

4 Answer the call.

5 Process the call.
This may involve playing voice prompts, and/or requesting and responding to caller input.

6 Disconnect the call (that is, hang up).

7 Prepare for another call.

Two basic algorithms are recommended when writing applications to process incoming calls: the first one acquires a voice session with Meridian Mail and retains it across calls on a dedicated channel, and the second algorithm establishes a new session for each call on a shared channel.

First Algorithm

The first algorithm is as follows:

1 Register with the Meridian ACCESS system.

2 Acquire a voice session from Meridian Mail.

3 Wait for an incoming call.

4 Answer the call.

5 Process the call.

6 Disconnect the call.

7 Wait for another call?

   a. If yes, return to to step 3

   b. If no, release the voice session and deregister from the Meridian ACCESS system.

Second Algorithm

The second algorithm is as follows:

1 Register with the Meridian ACCESS system.

2 Request a channel when a call arrives.

3 Wait for an incoming call.

4 Answer the call.

5 Process the call.

6 Disconnect the call.

7 Wait for another call?

   a. If yes, return to step 2

   b. If no, release and deregister from the Meridian ACCESS system.

Implementation

Notice that the only difference between these two algorithms lies in step 7, when the application is to wait for another incoming call. This requires a very minor difference in the actual code to implement each algorithm.

Omitting the actual decision in step 7 (and error handling!), and using Meridian ACCESS API functions, the first algorithm becomes similar to the lines of code as illustrated in Figure 3-3.

Figure 3-3
Differences between first and second algorithms

```
m_Register(&rc);
m_Acquire(class, &rc)
while (m_WaitingForCall(timetowait, FALSE, &rc)) {
 m_AnswerCall(&rc);
  code to process the call....

 m_DisconnectCall(&rc);
    }
m_Release(&rc);
m_Deregister(&rc);
```

The second algorithm becomes

```
m_Register(&rc);
m_AcquireOnIncomingCall(class, &rc);
while (m_WaitingForCall(timetowait, TRUE, &rc)) {
 m_AnswerCall(&rc);
   code to process the call....

 m_DisconnectCall(&rc);
    }
m_Release(&rc);
m_Deregister(&rc);
```

After selecting an algorithm, the developer must choose which API function, to acquire a voice session, is best suited to the application (step 2): m_AcquireOnIncomingCall() or m_Acquire.

Selection between these two functions should be based upon the following:

- Is a particular, physical Meridian Mail channel required?
  This will require some set up on Meridian Mail (see the Meridian ACCESS Configuration Guide for details).
- Is pre- first call processing on the Meridian Mail server required? (that is, log in to a particular account, open voice file, and so on)
- Are voice sessions shared with other applications (or other instances of the same application)?

If either of the first two features are required, then the application must use the m_Acquire() function in step 2. If the third feature is required, then the m_AcquireOnIncomingCall() function must be used.

For example, consider an application for which the first algorithm is to be used, but selection of a particular, physical Meridian Mail channel is not required, and the application does not have any pre- first call processing to perform on Meridian Mail. This application might use m_AcquireOnIncomingCall() to acquire a voice session with Meridian Mail.

Note that if the specifications for the application should change, and a requirement to select a particular physical Meridian Mail channel was introduced, for example, the only modification required to the actual code would be to replace m_AcquireOnIncomingCall() with m_Acquire().

Notes concerning either algorithm

In addition to various application specific design constraints, a number of criteria should be considered when designing an application to handle incoming calls, including the following:

- Availability of Meridian Mail voice sessions
- Expected frequency of incoming calls
- Amount of host computer processing required between calls
- Use of other Meridian Mail resources (that is, files)

The first algorithm, in which a voice session is acquired and held across calls, is best suited to applications that dedicate the same Meridian Mail resources for each call (that is, files). With this algorithm, little processing time is required between calls.

The second algorithm acquires and releases a voice session for each incoming call, and is best suited to applications that share different Meridian Mail resources for each call. With this algorithm, significant processing time is required between calls.

WhatKey sample program

WHATKEY is a simple example of a Meridian ACCESS application that services incoming calls and responds to telephone keypad input entered by the caller.

The WHATKEY application has the following two phases:

- ***Initialization*** In this phase, lobal variables are initialized, event handler routines are installed, and the user is prompted for account, password, and ACCESS Class. The application then registers with the Meridian ACCESS system.
- ***Main loop*** In this phase, WHATKEY repeatedly services incoming calls until a key is pressed on the terminal keyboard. WHATKEY tells the Meridian ACCESS system its associated class number and requests a voice session using m_AcquireOnIncomingCall().

  WHATKEY then uses m_WaitingForCall() to block while waiting for an incoming call. Notice that the AcqNewChannel parameter is set to TRUE. This means that WHATKEY releases its voice session at the end of each call, and acquires a new one for any subsequent calls. This is typical of most incoming call applications that share voice sessions. On SMDI systems, the channel automatically releases after the call disconnects. Note also that the MaxTime parameter is set to MIN_WAIT_TIME. This provides the WHATKEY with a way of breaking the blocked incoming call to either exit or continue looping.

When a call arrives, WHATKEY logs on to the specified Meridian Mail account, and opens the WHATKEY voice segment file (created by the PREPSAMPLE program). Note that the incoming call is answered after WHATKEY has logged on. Applications must answer incoming calls within 15 seconds of the IncomingCall event, so only minimal (if any) operations should be performed here.

After the call is answered (using m_AnswerCall()), WHATKEY plays a welcome prompt, and enters the active call subloop.

When the active call subloop terminates (caller hangs up), WHATKEY begins the main loop again. If WHATKEY does not receive a call, and a key on the terminal keyboard was pressed (either during the last call, or while waiting for a call) then the main loop terminates, WHATKEY ensures that the voice session is released, and deregisters from the Meridian ACCESS system.

A complete listing of the WHATKEY sample program is in Appendix A.

Whatline sample program

The WHATLINE sample is a simple example of an application that acquires a voice session, and holds it across several incoming calls (the first of the two algorithms described earlier). For each call, WHATLINE plays a series of voice segments corresponding to the local extension number dialed by the caller.

The WHATLINE sample has the following two phases:

- ***Initialization*** In addition to the initialization performed in the WHATKEY sample (event handler installation, prompting the user for account, password, and Meridian ACCESS class, and registering with the Meridian ACCESS system), the WHATLINE program also acquires a voice session (using m_Acquire()), logs on to the specified Meridian Mail account, and opens the WHATKEY voice segment file.
- ***Main loop*** Similar to WHATKEY, WHATLINE uses m_WaitingForCall() to block while waiting for an incoming call. However, in WHATLINE, the AcqNewChannel parameter is set to FALSE. This means that WHATLINE will not acquire a new voice session for each incoming call. It retains the one acquired during initialization for the duration of the program's execution. Again, notice that the MaxTime parameter is set to MIN_WAIT_TIME to break from the wait.

When a call arrives, WHATLINE simply answers the call and echoes a voice prompt followed by a series of voice segments that correspond to the extension dialed by the caller. It then waits for the caller to hang up.

When the call terminates (caller hangs up), WHATLINE begins the main loop again. If no calls arrive within MIN_WAIT_TIME, and a key was pressed on the terminal keyboard (either during the last call, or while waiting for a call), then the main loop terminates, and WHATLINE releases the voice session and deregisters from the Meridian ACCESS system.

Development environment

The Meridian ACCESS development environment is simply a relocatable API object library and a set of C-include files used to define Meridian ACCESS variables and constants. The application is developed using the C programming language coupled with the Meridian ACCESS API library functions. In addition, your application could be coupled with many other third party libraries such as a database package. The necessary API library objects are then linked into an executable program at compile time. The sections that follow describe the process of creating Meridian ACCESS applications.

The API Include Files

The API function declarations, data structures, variables, and constants needed to compile a Meridian ACCESS application are contained in various C-include or header files. There are several “include” files each of which provides a specific class of functionality as described below:

- ***m_acc.h*** General constants, structures, and return code declarations
- ***m_local.h*** Local Meridian ACCESS functions
- ***m_rm.h*** Meridian Mail resource management functions
- ***m_file.h*** Meridian Mail file access functions
- ***m_voice.h*** Meridian Mail voice operation functions
- ***m_msg.h*** Meridian Mail messaging functions
- ***m_seg.h*** Meridian Mail voice segment file functions
- ***m_admin.h*** Meridian Mail user administration functions
- ***m_event.h*** Event handling functions
- ***m_lh.h*** Link Handler functions
- ***m_hilev.h*** Meridian ACCESS high level API functions
- ***m_ens.h*** External Notification Service functions

Each file is declared in the application source using the C-include statement on an as required basis. The entire set of include files is contained in "include" directory which is created during the installation procedure. The absolute path of this directory is dependant on where the Meridian ACCESS package was installed on your UNIX workstation. Nonetheless, it is always located relative to $M1_ACCESSHOME (that is, $M1_ACCESSHOME/include).

The API Library

The API Library is a standard relocatable object library which provides most of the functionality within Meridian Mail voice services. The entire library is contained in the file m_acc.lib which is created during the installation procedure.

The absolute path of this directory is dependant on where the Meridian ACCESS package was installed on your UNIX workstation. It is always located under the "lib" directory relative to $M1_ACCESSHOME (that is, $M1_ACCESSHOME/lib/m_acc.lib). Thus, you could compile and link a Meridian ACCESS application with the following statement (assuming the source myprog.c declares the appropriate include files):

Figure 3-4
Example—Compile and link statement

```
cc -I $M1_ACCESSHOME/include -c myprog.c -o myprog myprog.o \
$M1_ACCESSHOME/lib/m_acc.lib
```

Sample Make File

An easy way to automate the creation of Meridian ACCESS application programs is through the UNIX “make” utility. The following sample portrays a typical make file that compiles and links a Meridian ACCESS application using a single source file.

Figure 3-5
Sample Make File

```
        Command file for Compilation and Linking        Program    : myprog

Define absolute path for ACCESS include and library directories

INC = $(M1_ACCESSHOME)/include
LIB = $(M1_ACCESSHOME)/lib

Define our local “C” compiler and option flags

CC = cc
CFLAGS = -O -I $(INC)

Define absolute path for ACCESS include and library files

HEADERFILES = $(INC)/m_acc.h $(INC)/m_rm.h $(INC)/m_seg.h \
        $(INC)/m_file.h $(INC)/m_voice.h $(INC)/m_msg.h \
        $(INC)/m_admin.h $(INC)/m_event.h
LIBS=  $(LIB)/m_acc.lib

Define our local object files

OBJECTS = myprog.o

Define the dependencies for local executable file “myprog”

myprog:   $(OBJECTS)
   $(CC) $(CFLAGS) -o myprog $(OBJECTS) $(LIBS)

Define the dependencies for local source file “myprog.c”

myprog.o : myprog.c $(HEADERFILES)
   $(CC) $(CFLAGS) -c myprog.c
```

Chapter 4: Development guidelines

Suggested user interface guidelines

While the requirements for user interfaces vary greatly depending on the application, there are a number of guidelines applicable to most voice-based applications. The following guidelines should be considered when designing an application.

Structure

- Menu structure should be simple, consistent, and effective.
- Determine whether “reverse menu traversal” is allowed, so that callers can get to previous menus. If so, use a consistent key for that purpose.
- Try to incorporate guidelines you already apply to screen-based applications. Consistency is particularly important, for it can save unnecessary prompts explaining inconsistencies.
- Verify important input. If a human operator would read back the input, have the application do the same. Verify telephone keypad input by echoing the spoken equivalent.

Assistance

- Ensure that users who don’t make the first response within a reasonable period are forwarded to an operator. This deals with rotary-dial callers and other callers who cannot or do not wish to interact with the system.
- Callers should be allowed to transfer to an agent at any point, usually by pressing “0”.
- Once a user has made at least one input to the system, a lack of further responses should generate prompts, reminding the user of the type and quantity of input now required.

Novice and experienced users

- When designing prompts, consider the user community. With users who will only use the system once or twice, prompts can be longer and more detailed than for those users who would use a system daily.
- Allow prompts to be interrupted and prematurely terminated by user input whenever possible. This allows experienced users to skip repetitive prompts.
- If a system will be used by novices and experts, consider providing an interface which will permit novices to use the system without slowing down or irritating experts. There are some strategies available, including the provision of different interfaces, perhaps determined by a user's identification number.

Prompt recording

- Keep the number of different voices to a minimum (unless additional information can be conveyed)
- Select a voice that will not sound too unusual to users. Select a speaker whose voice would be considered pleasant. The speaker's accent, style, volume, and pitch all merit careful attention.
- Record in a quiet, echo-free environment. Do not underestimate the noise caused by office air circulation systems. An office may sound very loud when heard by someone calling from a quiet residence.
- Avoid prompts which scold users, in words or tone. Regardless of how many mistakes users make, they won't appreciate rude directions.

Errors and Feedback

- Callers should always be informed of anything wrong. If possible, inform them about the nature of the problem, how it affects them, what their options are, and when the problem will be corrected.
- Callers should be allowed to provide feedback to the system administrator (perhaps by leaving a message in a Meridian Mail mailbox).

The above are only recommendations. There are circumstances in which some of these guidelines may not be suitable, and each application and user group should be considered carefully.

Suggested programming model

Any telephone based voice application process developed using Meridian 1 ACCESS library must be able to handle many types of asynchronous events. In fact, most voice based applications are typically driven by these asynchronous events. Events may stem from an internal source such as a UNIX signal. They may also stem from an external source such as a Meridian Mail error, or a person at the end of a phone line disconnecting.

An ideal programming model for implementing an event driven process is the “STATE MACHINE”. A state process will define all the various states (or situations) within an application and the events that drive the process into a particular state. In the next section we will attempt to demonstrate how a typical telephone based voice application would be implemented using the the state machine model.

LOTTO sample application

The LOTTO sample program is a typical Interactive Voice Response (IVR) application implemented using the function calls included in the standard ACCESS API Library. An IVR system must protect itself both from the caller and from all components that it interfaces with. The IVR system must also anticipate any possible scenario where the caller is left “hanging”, or where the caller hangs the system. With this in mind, LOTTO will exhibit the correct form of error detection and recovery for the common error scenarios. It would be impractical to demonstrate all possible error scenarios in the short span of this manual.

In addition, LOTTO will also demonstrate the following activities which are typical in an IVR systems:

- Collecting single digits for voice menus
- Collecting a series of digits terminated by a designated key
- Playing individual voice segment files
- Playing a list of concatenated voice segment files

The ‘lotto’ is a sample program implemented using the ACQUIRE model, and does not release its voice channel until it is terminated.

Description

The LOTTO program is a simple IVR application that accepts incoming calls, generates a series of random numbers in a designated range, and plays these numbers back to the caller. It begins by waiting for the next call to arrive and “interrupt”. Once it receives a call, it plays a “welcome” prompt and asks the caller to enter a “seed” (a number between 0 and 99,999) using the key pad of a touch-tone phone. The “seed” will be used in generating the sequence of random numbers.

The caller is then prompted to make a selection from the following menu:

- Press “1” to generate 6 random numbers between 1 and 49.
- Press “2” to generate 6 numbers between 1 and 80.
- Press “3” to repeat the last numbers generated.
- Press “4” to quit.

The caller remains in the menu until “quit” is selected or until a call disconnect is detected (that is, the caller hung up). When the call is eventually disconnected the entire process is repeated.

Application States

There are seven different states within the LOTTO application. Although these states are specific to this application the general concept can be applied to most IVR systems. These generic states are described below in their chronological order as they would occur under normal conditions:

INIT

This is the initialization state, and should only be executed once during the life span of the application. In this state, the application registers with Meridian ACCESS, acquires a voice channel, logs on to the appropriate Meridian Mail account, and sets up all of the Meridian Mail event handlers. If it is interrupted by a termination signal, the application enters the EXIT state. Otherwise, it enters the WAIT state.

WAIT

This state simply waits for the next incoming call to arrive. It remains blocked, waiting for a call until it has been interrupted by an incoming call or a termination signal. If it is interrupted by a call or termination signal the application enters the ANSWER or EXIT state, respectively. Otherwise, it remains in this state.

ANSWER

After receiving notification of an incoming call this state will answer the call and greet the user by playing a “welcome” prompt. The caller can interrupt the playing of this prompt by simply hitting any key on his/her touch-tone phone. At any point, the caller can “hang up”, which puts the application back in the WAIT state.

ENTER

The caller is now prompted to enter a number followed by “#” key. This is analogous to a customer entering an account number in a more realistic application. This state checks for the time-out condition during entry of the number. A time-out condition causes a more detailed “help” prompt to be played to the caller. If a retry limit is reached the call is disconnected and the application is put back in the WAIT state.

MENU

The caller is now presented with a “voice menu” and is prompted to enter an appropriate selection. One of the possible selections is to “quit”, which disconnects the call and puts the application back in the WAIT state. The caller could also select one of the play options (putting the application in the PLAY state). The MENU state checks for a time-out condition and invalid options during entry of a menu selection. A time-out condition or invalid selection causes a more detailed “help” prompt to be played to the caller. If a retry limit is reached the call is disconnected and the application is put back in the WAIT state.

PLAY

The caller has selected one of the play options. At this point, the application plays the voice segment files corresponding to the information the caller requested. After receiving a “play stop” or a DTMF (touch-tone) interrupt, the application is placed in the MENU state again. This would allow the caller to repeat the process once again or to quit.

EXIT

The EXIT state is only executed once during the life span of the application. In this state, the application performs an orderly shutdown just before terminating itself. It releases the previously acquired voice channel, logs out of Meridian Mail, and de-registers from Meridian ACCESS before finally terminating. The EXIT state can never be executed with an active call. Therefore, it can only be reached from the INIT or WAIT state.

Pseudo-code

The following pseudo-code represents the general program flow for LOTTO as it would be implemented using the state machine model. There are a number of global variables that are maintained throughout the process.

They are:
STATE – the current state of the process
CALL – state of current call (which could be connected/disconnected)
TERMINATE – TRUE indicates we should exit the process gracefully
Main
– Set up UNIX interrupt handlers signals SIGHUP, SIGINT , SIGQUIT, and SIGTERM;
– Set STATE = INIT;
– Set CALL = disconnected;
– LOOP until its time to TERMINATE
  Switch on the current value of STATE:
  case INIT:
    – Set up ACCESS interrupt handlers for OnCallProgress, OnIncomingCall;
    – Register with ACCESS;
    – Acquire a voice channel;
    – If initialization is OK set STATE = WAIT otherwise STATE = INIT;
  case WAIT:
    – Wait for the next call to arrive;
    – If we have a call set STATE = ANSWER;
    – If it is time to terminate set STATE = EXIT otherwise STATE = WAIT;
  case ANSWER:
    – Log into Meridian Mail;
    – Set up Play so that it is interruptible;
    – Play the “WELCOME” Prompt;
    – If call is disconnected set STATE = WAIT; otherwise STATE = ENTER;
  case ENTER:
    – Play the “ENTER SEED” Prompt;
    – Collect digits for seed;
    – If seed is OK set STATE = MENU;

- If time out occurred play “HELP” prompt and collect digits again;
- If exceed retries disconnect call and set STATE = WAIT;
- If call is disconnected set STATE = WAIT;

case MENU:

- Play the “MENU OPTIONS” Prompt;
- Collect digit for menu selection;
- If time out occurred play “HELP” prompt and collect digit again;
- If exceed retries disconnect call and set STATE = WAIT;
- If call is disconnected set STATE = WAIT;
- Switch on the menu selection:

case QUIT:

- Play “GOODBYE” prompt;
- Disconnect call;
- Set STATE = WAIT;

case REPEAT;

- If we have prev. generated nums Play “REPEATING LAST NUMS” prompt otherwise Play “ERROR” prompt and set STATE = PLAY;

case 6/49:

- Generate 6 random numbers between 1 and 49;
- Set STATE = PLAY;

case 6/80:

- Generate 6 random numbers between 1 and 80;
- Set STATE = PLAY;

default:

- Play “ERROR” prompt;
- Set STATE = WAIT;

End MENU Switch

case PLAY:

- Play the “YOUR NUMS ARE” Prompt concatenated with the random numbers generated;
- If Play is OK set STATE = MENU;
- If call is disconnected set STATE = WAIT;

case EXIT:

- Release the Meridian Mail Session;
- Set STATE = TERMINATED;

End STATE Switch

- End of LOOP

End Main

Parent process

Different methods exist for starting up multiple voice–based application processes (the options are the same as for starting up the link handler process). The method of starting multiple application processes under program control using UNIX system calls provides control when the application processes must be killed or restarted. This startup method is usually performed by a process that's main purpose is to perform the startup function; this process is usually referred to as the "parent" process.

System startup

The main purpose of the parent process is to start the required number of application processes. This is typically done using the UNIX "execl" and "fork" system calls (one execl and fork sequence per application process). The number of voice channels that the system is to control is usually kept in a UNIX file or in a database that the parent process examines.

Monitoring and maintenance

Most target systems have the parent process monitor the status of the processes that it is responsible for starting. Typically, these are the link handler, the application processes, and any other processes required on the system (system administrator console interface, or host session managers, for example).

Monitoring can be performed using several techniques, one of which is to check a portion of shared memory for an up-to-date timestamp from each monitored process. The shared memory is usually created by the parent process prior to starting any of its child processes. The child processes should then regularly get the system time and update their timestamp in shared memory. This technique is not used for checking the status of the link handler process; rather, it is recommended that the parent register itself as the link handler monitor, or that the parent call specific link handler Meridian ACCESS API functions that return the status of the link handler on a regular basis.

If a process does not appear to be sane (not updating its timestamp, for example), the parent would typically try a graceful shutdown of the process by sending a UNIX signal and then restarting that process.

Some provision for ensuring that one or more processes are not consuming too much of the application processor's computing power may also be desirable. Consuming too much CPU time may also be an indication of a sick process or of a process whose environment is causing it to require a disproportionate amount of time. The UNIX "ps" and "grep" commands may be used to capture information about the amount of real time that processes are using.

Some indication of the process restart should be made (either to the target system's system administrator via a screen interface, or to a logfile as described in the following section).

Remote maintenance and diagnostics

In order to facilitate remote support by the development organization, a logfile should be maintained on the system's hard disk. This can always be examined remotely by dialing into the application processor's modem if a problem requires some diagnosis. It may be beneficial to have a single process (typically the parent) log information to the log file so that this information is easy to read and not interspersed from several processes trying to write at the same time.

System and call statistics may also assist with any problem diagnosis. This information is usually kept in a different file than the log file and is logged by a process assigned this function.

Having separate processes log information to files involves inter-process communication which is typically handled by UNIX messages being sent from the various processes to the logging process's message queue.

The application should have a console to alert the system administrator of any unusual occurrences on the system. The screen interface should not be cryptic and should provide information to direct the administrator to take the appropriate action (take note of the unusual happenings, correct the problem in a particular way, or contact the developer's support number with information specific to the problem).

The application's console should be available remotely over a modem so that the developer can get additional information and possibly assume full control of the application from the remote console.

The developer may want to design the target system's user interface to appear similar to the Meridian Mail administrator user interface to reduce any confusion by the system administrator.

Other processes it controls

The parent process may also maintain processes other than the link handler and the application processes. Host communication, console interface, and statistics are examples of activities that may be performed by other processes on the target system. It is recommended that the parent monitor these processes regularly.

For processes that handle host communication, it has been found that these are most efficiently treated as a server to be used by the application processes since a typical IVR script call flow time is usually much longer than the host's response and transmittal time. Some engineering formulas are included in this document to help determine the number of host sessions (processes) for a given application. Message queues are typically used to implement the server philosophy.

Some other niceties for a host communication process are: an indication of host response and transmittal times to be kept in a file on the target system, an indication of the number and time duration of callers whose host requests are waiting in the message queue to the host communication processes (so that more sessions could be configured required), and an indication that the host application is not reacting as it should.

Developing flexible applications

Voice processing applications, like most applications, evolve over time and are thus subject to change. You can anticipate some of these changes to avoid the rigorous task constant modification and re-compilation. Some of the items most likely to change are listed below:

- ***Meridian Mail account/password*** The account and password a particular application logs into on your Meridian Mail machine.
- ***Class*** The Class number a particular application will use to acquire a Meridian Mail voice channel.
- ***Prompt files*** Names of the Meridian Mail voice files your application uses to play its voice prompts.
- ***Log file*** Name of the UNIX log file your application uses to record error data and operational measurements.
- ***Operational Measurements*** Whether or not your application collects operational measurements during a particular session.
- ***Link Selection*** The link that your application will use.

This list is dependant on the requirements of a given application. However, your application should be flexible enough to accommodate such changes. One method of achieving this is to place this dynamic information in a configuration file. The application would read this file once during its initialization phase. Another common method is to make dynamic information parameters of your application. However, this is only practical if the list is short.

Programming hints

System response time

The IVR system should provide adequate performance when handling the system's intended peak caller volume. Callers usually want a system to respond to digit entry in less than 0.5 seconds. System response times in the order of seconds are very annoying to many callers. System response time is a function of load on one or more of the three major components in an IVR system: the application processor, the Meridian ACCESS link, and the Meridian Mail platform. If the load on any of these is high, overall IVR system response time will suffer.

Application processor tips

- If all of the processes do not fit into the available RAM, page swapping will take place by the O/S. This takes time and affects system response. If it is thought that there is enough RAM but page swapping still occurs, memory assignment (or other memory affecting parameters such as buffers) may not have been properly set in the UNIX kernel.
- Use a smart multiport serial I/O card for the Link Handler. This will help reduce the amount of CPU used by the application processor.
- Try to avoid writing to the hard disk during a call but write at the end of a call. It may be better to pass the information to be written to a single task that will write to disk and free up the application processes to handle the next caller.
- Use *lint* to ensure that each process is written efficiently and is not wasteful.
- Be aware that all pending events will interrupt the application process. Because of this, event handlers should only set global data structures which are checked by the main routine. More complex event handlers are possible, but must be designed carefully. The general rule of thumb is to keep an event handler as small as possible.

- Regularly check global data structures (including flags) which event handlers modify, to ensure that the application is aware of the various events that may have occurred.
- If you are in a critical area in an application and do not want the process to be interrupted by an event, use the AutoEventOff mode of operation. You can later return to AutoEventOn mode of operation.
- Never call ACCESS functions within event handler routines. Calls to these functions can be interrupted by other pending events. Stack overflow is possible if many events are pending.
- Avoid creating Meridian Mail files with duplicate names, unless your program is prepared to handle them properly, or performs filing functions by file number.
- Your program should exit gracefully by calling m_Release () and m_Deregister (). This prevents the ACCESS channel and virtual connection to the ACCESS system from being blocked, and will also close any open voice files. If you do not exit gracefully, the resources will take a few minutes to free themselves, so you will not be able to reuse them quickly.
- A separate file system can be used to run the IVR system if it utilizes any disk I/O.
- Reduce any unused processes (for example, gettys on consoles that probably won't be used). This reduces the number of entries in the UNIX process ID table and should reduce overhead for O/S time slicing.

Link issues

- Experimenting with the link handler process' priority from the parent/shell may improve system response.
- Making sure that the UNIX kernel parameters are set as recommended (or if peripheral drivers or UNIX resources are used by the IVR application, then modified accordingly) should improve performance.
- When an application process is through with its ACCESS session, it should release to free up MM resources. De-registering also reduces the number of processes that the link handler has to support.

- Using multiple links involves using the default link (link six), or configuring a specific link. Altogether, there are eight links. You configure the link that you want to use by modifying its Link Handler configuration (lh.config) file or by issuing the m_SetEnv API function. In the Link Handler's ‘‘lh.config” file, you set two parameters (the base key and the environment variable ‘‘ACCESS”) to equal the link number that you want to use. If these two parameters do not equal, the Link Handler will not even find the default link. To configure a link with the m_SetEnv function, you issue it and, then, specify the ‘LogicalLink’ value (default one) to equal the link number that you want to use. Please note that changing the ‘‘lh.config” file's environment variable ‘‘ACCESS” overrides the ‘LogicalLink’.

Timers and alarms

Meridian ACCESS uses the UNIX alarm mechanism for timeouts on message queue (used to receive Meridian Mail responses from the Link Handler) block reads. The timeout can be set using the m_SetTimeout function.

Whenever an API function is called, Meridian 1 ACCESS will reset any previous alarms system calls within the developer's process. The developer's process may use its own alarm, however they should not be embedded in code where API function calls are made.

Naming conventions

Do not use “m_” or “mu_” prefixes since they are used for Meridian ACCESS API procedure names.

Return codes from Meridian ACCESS always have the prefixes “MME” “MMS”. Meridian ACCESS uses other internal constants that don't have a naming convention. These may be discovered during compilation.

Password expiry

The m_Logon function will return information about the mailbox that was logged into. This information will contain the fact that the mailbox may already be logged into or that the password for that mailbox has expired. The Meridian Mail system will allow the m_Logon function to login anyway but the application should be aware of the information and take appropriate action (for example, printing a message when the password has expired so an administrator can change the password to a new one).

Limits on number of channels

The variables that will affect the number of voice channels that can adequately be controlled include

- number of calls and length of each call (as gauged by CCS – Connect Centi Seconds, or “hundreds of call seconds”)
- length of voice prompts
- interfacing to an outside host
- interfacing to a local database
- performing a lot of call processing (limitations may be found on the switch if it is already heavily loaded with thousands of ACD queue positions for example)
- performing plenty of voice processing
- number of links
- link speed

A number of components may restrict the number of channels useable for an application. These include

- the application processor itself (memory, speed, processing power)
- number of links
- link speed
- Meridian Mail (sharing resources with other services, amount of voice processing)
- the amount of call processing taking place on the AML/CSL link

Multiple key press effect

It is very important to be able to service DTMF collection efficiently to accommodate experienced callers if the application allows type-ahead.

Use digit event handling functions that incorporate circular queues to store the digits. Do not process them until all the digits expected have been received or when the collection time has timed out at a particular menu node.

Caching of voice segment files on Meridian Mail

Segment files use MM disk caching mechanism for playback. In addition, all voice segments are indexed and can usually be located on the first disk read. The caching mechanism used even allows a file located on node 2, to have its own private cache on other nodes in the system.

UNIX signals reserved by Meridian ACCESS

Meridian ACCESS uses SIGUSR2 signal when AutoEvent checking is on. The developer should avoid using the SIGUSR2 signal for handling any other sort of interrupt–driven processing. The link handler process will send a SIGUSR2 signal to a process that has just received an asynchronous event from Meridian Mail and has AutoEvent checking on.

Meridian Mail resource utilization

Voice segments should be concatenated before being sent to Meridian Mail to be played instead of requesting Meridian Mail to play them one by one. It is impossible to do that if the application's voice segments are scattered between voice files, so voice files should be planned to contain segments that will be concatenated together out of the same voice file.

Call control

It should be stressed that when the Meridian Mail platform is under Meridian ACCESS control for one or more voice channels it is up to the Meridian ACCESS application to control the use of Meridian Mail platform resources. The application should guard against abusive callers, since even a few callers pressing digits very rapidly can deteriorate system performance to others on the system.

Each call should be timed so no one caller can hold up the line forever. Since alarms are not a good way to verify the length of the call (see alarms and timers section), assign the time to a variable when the call is connected and compare this variable to the current system time throughout the IVR script.

Use timeouts while collecting digits so if the caller doesn't enter digits after a period of time (variable depending upon the particular menu node), the application requests again for digits to be entered. If the caller does not obey then the call should be transferred to an agent or the call disconnected. The number of errors should be kept track of during the call session. If these exceed an application threshold, then the call should be transferred to an agent or the call disconnected.

A method of pacing the caller's digit entry should be devised since even following the two suggestions above could not stop a caller with a valid menu traversal pattern of digits from repeatedly entering this sequence at an incredible rate (possibly using an autodialer) during the controlled length of the call. An IVR system should be very robust.

Post call-processing processes

If information about a call session must be kept, it should be written to hard disk at the end of a call to not slow down the caller's session. Disk Access can affect the session's performance.

Meridian ACCESS timeouts

The application should protect itself from Meridian ACCESS failing to report a problem about the Meridian ACCESS session. m_TimeOutContinue can be used to deal with impending Meridian ACCESS session timeouts due to inactivity, but there is a possibility that the Meridian ACCESS session can die without the application process receiving an event informing it of this. The application process would then never know that the session is down and would assume that callers are not calling in. This would result in an inactive voice channel and given enough time, possibly all of the voice channels would go inactive.

The application must protect itself from this by polling the Meridian Mail platform to verify the operation of the Meridian ACCESS session. This can be performed in the application's state machine's IDLE state (when it is not currently handling a call). An API such as m_GetSysVersion can be used as the poll API. If a problem is found then the application should perform its recovery for that state.

As an alternative to polling, you can use the m_OnTimeout function. This event is sent if no non-local Meridian ACCESS command has been issued for one minute. The m_OnTimeout function resets a ‘‘watchdog” timer, which runs down as a result of inactivity from the application.

Similarly, there exists a window of time just prior to doing a blocked read on a message queue where events can go unnoticed by an application process. The application should break out of the message queue periodically to check if any events have been received.

Meridian Mail Stale File SEER

The Meridian Mail SEER 11-50: Stale File Version was implemented to keep track of old files and is a warning that a file has been kept open which is not the newest version. This applies to all system files as well as application voice prompt files. This SEER will occur when a file is kept open, and a change has been made to that file. The Meridian Mail operating system cannot close the old file since it is being kept open. This SEER may occur when using the Acquire model, since system files, user profiles and prompt files can be kept open for extended periods.

An example of how this SEER would be encountered would be

- an application has a voice file open for playback
- that voice file is updated by another application in Update mode

To stop this warning SEER, the applications should be "cycled" to close any open files and log off, and then log on and re-open the files. This has to be done so that the new recorded voice files can be used. If voice files are being changed while applications are "live", a mechanism should be built in to signal all applications to close files and then log off so that the newer recordings may be used. This does not have to be done immediately–it can be done between calls.

If the AcquireOnIncomingCall model is used and separate logons and opening of voice files are done for each call, this SEER will not be encountered. The logging off and re-opening of voice files will ensure that old files are closed and the newest files are always opened.

Developer's checklist

The developer should train the customer's IVR system administrator. Training should include use of the IVR application, what it requires to maintain the application, how to recognize when something is wrong, and what to do if something is wrong. This training should also include formal documentation that the administrator can occasionally review or use as a reference.

The following information should be passed on to the system administrator (for CSE and Meridian Mail configuration purposes):

- Disk space and channel requirements
- Application class number

- Channel allocation method (does the application require shared or dedicated channels?)
- Meridian Mail account (mailbox) requirements
- Link requirements

Chapter 5: System configuration

Link Handler

The ACCESS Link Handler requires the following three pieces of information for it to function properly:

- the link number being controlled
- the device name which connects your workstation to the port on Meridian Mail
- the unique base key that will be used to identify the UNIX message queues between the Link Handler and its client applications

The above information is configurable through the Link Handler's configuration (lh.config) file. This text file is located within the $M1_ACCESSHOME/bin directory, and is exclusively used by the Link Handler. Parameters are identified through keywords within the file as described in Table 5-1.

Links

Applications select a link by setting an environment variable or by issuing an API command.

Table 5-1
lh.config file information

Keyword	Parameter	Defaults
Device1=	<device_name_of_port>	Not an optional entry. There is no default.
Key=	<unique_key_identifier>	Optional entry. Default base key is 106 (or 0x6a) which will create the two UNIX message queues with the queue keys 0x6a000000 and 0x6a000001 for the first link, and 0x6a000002 and 0x6a000003 for the second link and so on.
CarrierDetect=	<TRUE>	Optional entry. Enables carrier detection if it is available.

When you install Meridian ACCESS, a default lh.config file is created with a single dummy entry in it:

Device1=/dev/tty

You must modify this entry to reflect the real physical port device connected to the port on Meridian Mail. The Link Handler will attempt to open this device and configure its RS-232 characteristics so that it can talk to Meridian Mail. Make sure there is no "getty" process running against this port. If a getty process is running, remove the appropriate entry from the /etc/inittab file.

Device1=/dev/tty15 (for example, uses /dev/tty15 as port)

For systems configured with multiple ACCESS links, more than one device line should appear. To add a second Link Handler specify another line as

Device2=/dev/tty16 (for example, uses /dev/tty16 as port)

The "Logical Link" value is taken from the last character of the device string. For example, to configure logical link "4," you would add a line "Device4=." The default is logical link 1. For example, "Device=/dev/tty15" configures logical link 1.

Provided that the base key is not overridden, you can issue the API command, m_SetEnv, to select from multiple links by changing the "logical link" value to the link number you want to use.

One link handler is started for each link defined in lh.config. The Link Handler will attempt to create the two message queues used to communicate to its clients. The two queues for link number 1 are identified by the queue keys 0x6a000000 and 0x6a000001; the two queues for link number 2 are identified by the queue keys 0x6a000002 and 0x6a000003; and so on. The queues are derived from the default base key (106 or 0x6a). If the two queue keys conflict with existing applications, you must make an entry in lh.config with an explicit key. The entry applies only to the link specified. Accordingly, the following example shows an entry with an explicit key:

Key=8 (for example, create message queues 0x08000000 and 0x08000001)

If you override the default key with the key parameter, you must also set the application environment variable ACCESS to the new key. The m_SetEnv API can no longer be used to select a link by specifying a logical link number. See the “Setting up a user’s environment” section later in this chapter for more details.

If carrier detection is available on the serial port, the carrier detection feature can be enabled with the line

CarrierDetect=TRUE

This feature will generate a UNIX signal if the serial port connection is broken. This line should not be entered if carrier detection is not available.

Link Handler startup methods

After you have properly modified the lh.config file for your environment, you can start the Meridian ACCESS Link Handler process. The Link Handler executable file, lh, and the lh.config file must be in the same working directory. By default, the installation process places both files in the directory $M1_ACCESSHOME/bin.

The Link Handler process should be invoked as a background task that services all your Meridian ACCESS application processes for a link. Because the lh process is a server task, it must remain running in the background at all times. All standard output messages from this task should be routed to the console in order to capture operator attention if any malfunction occurs.

For systems configured with multiple ACCESS links, specify the link number to the lh program when it is started. For example, to start a second Link Handler enter the following command:

```
./lh 2 &
```

The lh started with parameter 2 here will look for Device2 in the lh. config file.

As a Daemon Process within /etc/rc2.d directory

This is the preferred method of invoking the lh process since the process is automatically started as a Daemon when the system is initially booted. There is no need to manually invoke the process every time a Meridian ACCESS application is started. The process should remain operational for the lifetime of the system unless it has been killed.

To run the lh as a Daemon process, you must make the appropriate file entry in the /etc/rc2.d directory. The file should contain the command statement which appears below in bold text. However, you must slightly modify the statement depending on the link that you want to use, such as link three (lh 3) or link five (lh 5). The following example of the command statement applies to the first link, as indicated by the lh 1:

```
$M1_ACCESSHOME/bin/lh 1 > /dev/console &
```

To run the lh as a Daemon process, see your UNIX Administrator's Reference Manual for details.

From a Meridian ACCESS Monitor Process

If your voice processing application demands very tight control, it is best to execute your entire application, including the lh, from within a Parent/Monitor process. This will enable you to closely monitor the application as well as the Link Handler. It may even respawn a process that is deemed to be nonfunctional. This method allows you to automate any WatchDog-type activity.

To run the lh from within your Parent/Monitor process, you must call the Link Handler API function m_StartLink. The C code within your Parent/Monitor source file should look like the following:

```
sprintf(lh_path,"%s%s%d",M1_ACCESSHOME,"/bin/lh",linknum);
if (!m_StartLink(lh_path,rc)) {
    YourErrorRtn("Could not spawn lh process for link %d:
RetCode=%s\n",linknum,rc);
    exit(–1);
}
```

Consult the *Meridian ACCESS API Reference Manual* (NTP 555-7001-317) for more detail on m_StartLink.

From a UNIX Shell

This is a simple manual method of invoking the Link Handler. However, when your system is shutdown you will manually have to invoke the Link Handler once again.

To run the lh on the first link from within your UNIX shell, type the command statement which appears below in bold text. However, you must slightly modify the statement depending on the link that you want to use, such as link two (lh 2) or link four (lh 4). The following example of the command statement applies to the first link, as indicated by the lh 1:

```
$M1_ACCESSHOME/bin/lh 1 > /dev/console &
```

You must make sure you detach the process form your shell using "&"; otherwise, you will tie up your terminal until the lh is terminated.

Once the lh process has been invoked it will attach itself to the device port indicated in the lh.config file. It will also spawn a child process, lhrx, whose sole task is to constantly monitor the device port for messages from Meridian Mail. The lh process creates a log file known as lh.log every time it is invoked. If one already exists it will rename it to lh_old.log. These files will contains a problem log of lh activity and a summary of communication statistics during a particular session. The log files should be checked if you suspect you are having problems communicating with Meridian Mail.

If you pass the link number to the lh when it is started, the log file names will be numbered (that is, lh1.log and lh1_old.log).

***Note*:** If the link number is not specified at startup it will default to link number one and use the Device1 entry of the lh.config file.

Link Handler events

Events inform the Link Handler monitor process (typically the parent) of asynchronous information about the state of the link/Link Handler session.

For the Link Handler process, the event information available deals with the link synchronization status, any problems that the Link Handler has, and any Meridian Mail platform problems.

For the link monitor process to receive events, it must have an event handler installed. It does this using m_OnLhEvent to pass the developer-written event handler's name to the Meridian ACCESS internals. Whenever an event message is received by the Meridian ACCESS internals of the monitor process, the installed event handler procedure (whose name was passed using the m_OnLhEvent event handler installation function) is executed. Execution is then returned to the Meridian ACCESS internals and eventually to the monitor process.

Multiple ACCESS Links

Your MM system can have various options such as a single ACCESS link or multiple ACCESS links. You can support a maximum number of 48 active calls on a single link at 9.6 Kbps. For each link configured, a separate Link Handler process must be configured and started.

Each link is completely independent of the other except for sharing the same configuration file. There is no load sharing or redundancy between the ACCESS links. However, there are no restrictions from having load sharing and redundancy at an ACCESS application level. Sessions are maintained for their duration on the link where they originated.

Applications specify which link they want to use, before they issue the m_Register command, by calling the m_SetEnv command and specifying the LogicalLink parameter to equal the link number to be used (provided the default base key mechanism is not overridden). Unless specified, the LogicalLink default value of one is used. For more information about the m_SetEnv command, see the *Meridian ACCESS Application Programming Interface (API) Reference Manual* (NTP 555-7001-317).

> ***Note:*** You also can select from multiple links by specifying an explicit key in lh.config and setting the application's environment variable "ACCESS" to this value.

Applications may be registered only with one link at any point in time. You may switch links by resetting the LogicalLink value; however, you can switch links only when not registered.

There are several ways in which you can use the multiple ACCESS link capability. The method you choose will depend on your application requirements.

- a single Meridian Mail connected to multiple UNIX workstations

  This can be useful if a single workstation does not have the capacity to accommodate the load generated by a given number of channels. It can also be utilized to provide a level of redundancy in case of workstation failure or shutdown.

- a single UNIX workstation connected to several MM systems

  A single workstation can be configured to handle up to eight ACCESS links at the same time. This is useful for running administrative- or monitoring-type applications or controlling a few IVR channels at several remote locations.

- a single MM with multiple ACCESS links connected to a single UNIX workstation

  In this configuration, the system is best configured so that the application load is split between the links connected.

Development environment

Tuning your UNIX kernel for Meridian ACCESS

Performance tuning is an activity that may need your attention when you first set up your voice processing application. The default kernel resource configuration installed on your Motorola Delta V/68 workstation may be satisfactory for your application. However, this configuration may not be suitable for voice processing applications that service many Meridian Mail voice channels. These types of applications typically exhaust the message queue and semaphore resources of your system and, hence, require tuning.

You can use the following formulas as a guideline for tuning UNIX kernel message and semaphore parameters:

MSGMAP = UNIX Default + 2 * (# of Meridian Access Applications)
MSGMNB = UNIX Default + 600 * (# of Meridian Access Applications)
MSGSEG = UNIX Default + 64 * (# of Meridian Access Applications)
MSGTQL = UNIX Default + 10 * (# of Meridian Access Applications)
SEMMNU = UNIX Default + (# of Meridian Access Applications)
MSGSSZ = 16

In addition, you may need to increase the MAXUP and NPROC parameters if you anticipate that the number of processes that will be executed on the system will exceed the UNIX default values. You can use the following formulas as a guideline for tuning UNIX kernel process parameters:

MAXUP = min (180, UNIX Default + # of Meridian Access Applications)

NPROC = min (200, UNIX Default + # of Meridian Access Applications)

These new settings can be made with the sysgen utility. Consult your UNIX System Administrator's Guide for further details.

Setting up a user's environment

You may need to modify the User's .profile to gain entry into the Meridian ACCESS system. The following list details the changes required for you to easily utilize Meridian ACCESS tools and libraries:

Procedure 5-1
Setting up a user's environment

1 If you plan to develop Meridian ACCESS applications, define the environment variable $M1_ACCESSHOME.

M1_ACCESSHOME=/usr/m1access

export M1_ACCESSHOME

Defining the environment variable easily allows you to define and include the library directories in a make file. As well, you can use the make files for the Meridian ACCESS sample programs without modifying them.

2 If you plan to use any Meridian ACCESS tools (for example, the Voice Prompt Editor or ACCDIAG), add the Meridian ACCESS bin directory to their path.

PATH=$PATH:$M1_ACCESSHOME/bin:

Adding the bin directory to the path allows you to invoke these tools from any directory.

3 If you want to use multiple Meridian ACCESS links and you have overridden the default base key in the lh.config file to equal a specific number, you also must change the environment variable named "ACCESS."

ACCESS=key
export ACCESS

Defining the environment variable and the key determines which link your application will use.

ACCESS tools

The Meridian ACCESS system comes with various tools used to aid you in either creating and maintaining voice files or diagnosing problems. The tools reside on both the UNIX workstation and the Meridian Mail platform.

ACCDIAG

ACCDIAG resides within the $M1_ACCESSHOME/bin directory on your UNIX workstation. It analyzes the status of the critical components of a Meridian ACCESS system. If it deems a component nonfunctional, it will recommend corrective action.

The executable file must be within your search path. You do not have to be a superuser. The TERM environment variable must be set correctly (to a known terminal type that supports cursor addressing) before running the tool.

Procedure 5-2
Using the ACCDIAG tool

1 Type the following command followed by <Return>:

accdiag [ConfigFilePath] [–/ LogFileName] [–n Link Number] [?]

where:

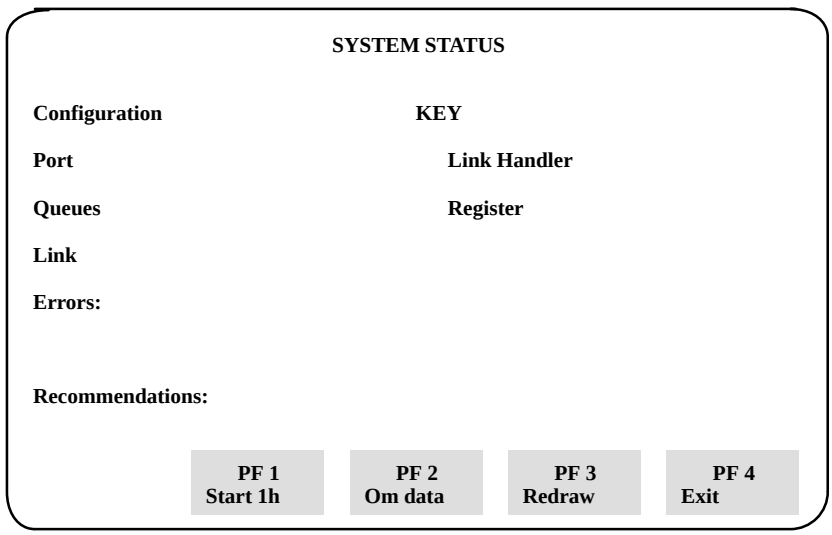
ConfigFilePath optional) provides the pathname of the directory containing both the Link Handler executable file and the configuration file. If unspecified, the Link Handler executable file and the configuration file must be found in the same directory as accdiag.

LogFileName is the name of the file to which status information is written. If not specified, the information will be written to the file accdiag.log in the current directory. Any previous log file of the specified name will be renamed with the suffix .old.

Link Number (optional) sets the context for the ACCESS link to be diagnosed. This number is the same device number found in the lh.config file. The default is 1 if not specified.

The status screen will appear (see Figure 5-1), and will be updated every 10 seconds.

Figure 5-1
ACCDIAG status screen



2 Press one of the following softkeys, as necessary:

- *Start LH* (F1)—Try to start the Link Handler if it is down.
- *OM Data* (F2)—Display the OM data screen.
- *Redraw* (F3)—Redraw the screen.
- *Exit* (F4)—Exit from the tool.

If the appropriate softkeys (F1 to F4) are not available on your terminal, press the first letter of the command on the softkey (for example, **S** *for Start LH or* **O** *for OM Data).*

Voice Prompt Transfer Tool

The Voice Prompt Transfer Tool resides on the Meridian Mail system. It facilitates the transfer of voice segment files between Meridian Mail systems (and different nodes on a single Meridian Mail system). See *Meridian Mail System Administration Tools* (NTP 555-7001-305) for more information on this tool.

Meridian ACCESS Diagnostics Tool

The Diagnostics Tool resides on the Meridian Mail system. You can use the tool to diagnose and/or monitor system activity related to Meridian ACCESS running on a UNIX processor. See *Meridian Mail System Administration Tools* (NTP 555-7001-305) for more information on this tool.

Voice Prompt Editor

The Voice Prompt Editor (VPE) is a tool that resides on the UNIX workstation within $M1_ACCESSHOME/bin directory. It is used for the creation and maintenance of voice prompts which are used by Meridian ACCESS applications. See the *Meridian ACCESS Voice Prompt Editor User's Guide* (NTP 555-7001-318) for more information on this tool.

Chapter 6: Engineering guidelines

This chapter discusses engineering considerations for those developing ACCESS applications. For each of the primary system components involved in the installation of the Interactive Voice Response (IVR) system the major service affecting parameters are noted.

System components

The following components must be taken into account for the IVR application:

- PBX
- Meridian Mail
- UNIX workstation
- Mainframe

PBX

There are three ways to set up the PBX to service callers which may terminate on an IVR service. The IVR application may be designed to work in a stand alone environment where it is the first and final destination of a call. It can be setup to screen calls (that is, preprocess them) coming into a call center before they are handled by a live agent. Finally, they may be setup to handle queue overflow conditions. The needs of the application and the customer determine which type of call routing and queuing setup you have in place.

Important factors to be considered when determining the demands your IVR application makes on the switch should include the number of calls per hour you are expecting to handle and the average length for the call.

If your IVR application is off loading existing call traffic from live agents the required number of incoming trunks are likely already in place. If this is a brand new service being provided, new incoming trunks etc. will have to be installed and connected on the PBX. ACCESS applications that do not utilize telephony do not use any PBX resources.

When setting up the Meridian Mail ACD queues on the switch, using features such as timed overflow to other queues or limiting queue size are approaches which can be used to limit or increase the number of calls in the system at any one time.

Meridian Mail

Whether your Meridian Mail system is being used exclusively for IVR purposes or IVR is just one of several features being used on the system, you must calculate the number of channels that need to be allocated for your IVR service. Allocating eight channels to IVR for example will allow eight callers to access the system at one time.

Channel requirements are determined using standard traffic engineering principles that take into account busy hour traffic and desired grade of service. The busy hour traffic is the highest traffic hour for a customer. Traffic capacity is stated in CCS (hundred of call connect seconds per hour).

To determine the number of channels needed for your application, use the following steps:

1 Estimate the average length of a call.

The range should normally be somewhere between 15 to 300 seconds and is determined by the type of application in use. For example information type applications generally have a long call length, while an automated attendant type application will have a very short call length. Also it may be important to include any application post call processing time in this figure as well. ACCESS applications have the ability to block the next incoming call in order to complete database updates which may be required. If your application is spending time doing this or any other call cleanup activity on a regular basis include this in your average length of call. This time must be included even though there is no active call, because the channel remains allocated during this time.

2 Estimate the number of calls for the busiest hour.

3 Multiply the figure from step 1 and 2 together. The total activity for each application is the average length of a call multiplied by the number of calls per hour.

4 Divide the total call seconds figure from step 3 and divide by 100 to determine the number of CCS.

5 Determine the number of voice ports by referring to Chapter 1, ‘‘Determining System Size”, in the *Meridian Mail Site and Installation Planning Guide* (555-70x1-200) for your platform. Include the CCS figure obtained in step 4 of Procedure 1-5.

Meridian ACCESS Link

The Meridian ACCESS Link is an asynchronous serial connection which runs at 4 800 kbyte/s through to and including 19 200 kbyte/s, with 9 600 kbyte/s as the default. The link connects the Meridian Mail system to the UNIX workstation. A proprietary full duplex protocol is used to pass messages between Meridian Mail and the UNIX workstation. Every ACCESS application establishes a virtual circuit on the link when an ‘acquire’ command is issued. The link is shared on a first in first out (fifo) basis and no priority is given to any particular application or message type. Messages may queue on either side while waiting to be delivered.

Your system may be configured with more than one ACCESS link. The maximum number of sessions for a link is 48. If you are using multiple ACCESS links it is usually a good idea to divide the load evenly between them when possible. For example, on a 64–channel system, configure the first 32 applications to use link 1 and the next 32 to use link 2.

As a general rule when an application calls an ACCESS API function, a message is sent across the link to Meridian Mail. The message is processed and a message containing the result of the command request is returned to the API. The application is blocked for the time in which it takes to complete the command.

For the most part, messages which get passed in either direction are very small (that is, less than 16 bytes) and are primarily used for signaling such things as ‘answer’ or ‘disconnect’. API functions only send as many bytes as they need. This reduces the time it takes to send a command which in turn makes it possible to send the next message sooner. Unsolicited event messages, such as the digit event, that occur during a call, do not require that the application respond. This means there is no message returned by the application in these cases.

The more complex API functions (which are used for listing files, messaging, or voice prompt maintenance) transfer larger messages as the amount of data that is sent or returned can be up to several hundred bytes. One can easily identify these commands by the amount of data which is either passed into or returned from a given API function.

UNIX workstation

Some of the more important things to remember when writing IVR applications for your workstation include the following points:

- All software which may be used during a call should remain memory resident at all times and must never be swapped out to disk. This includes not only ACCESS applications but any database or host communications software as well.
- All software written should be event driven, recognizing that they should not waste CPU time in busy loops that don't do any real work.
- Please read the section on 'Tuning your UNIX kernel' in this manual for new settings that should be used. If your application is a heavy user of system resources noted in this section, the numbers presented may need to be increased further.

The larger the IVR application is, in terms of the number of channels, the more attention you must pay to resources on all of the system components. In particular some of the fundamental resources on the UNIX workstation are things such as RAM (random access memory) and clock speed of the processor. It also may include the software performance of things such as database throughput in terms of transactions per second. The ACCESS link is also one such resource that should be considered when building your application.

It is possible for an application to monopolize the link to the extent where it degrades not only its own response time but also the response time of the other ACCESS applications. However, giving proper consideration to the usage of API functions will ensure this does not happen.

On an active system, the time taken for APIs to execute will increase as system processes compete for cpu time and as messages get temporarily queued at either end of the ACCESS link. The worst case is where all 48 channels for one link are being used for IVR and are active at once. Under these conditions, the response time for an individual API can be upward of one to two seconds in some cases.

m_Acquire versus m_AcquireOnIncoming Call Channel Allocation

One important point to keep in mind when using the m_AcquireOnIncomingCall (AOIC) method of accepting incoming call traffic is the number of API function requests required to play the first prompt when a call arrives.

For example, when using the m_Acquire method to answer an incoming call after the incoming call event arrives, the application must issue:

1 m_AnswerCall

2 m_PlaySegs

at a minimum before the caller will hear voice at the end of the line.

If the application was using the AOIC method, the application must issue at a minimum:

1 m_AnswerCall

2 m_Logon

3 m_OpenFile

4 m_PlaySegs

With AOIC, the application must issue at least twice as many commands to send the first prompt out, causing the amount of message traffic on the link to increase and the amount of time to almost double before the caller will hear voice after the call has been answered. Whether or not these effects will impact your particular application depends on the number of channels that you attempt to run at one time. Due to the extra number of API requests required to play the first prompt to the caller, you should not consider running more than 24 IVR channels on one ACCESS link using the AOIC scheme without thoroughly testing and verifying the particular application that you want to use. As well, you need to completely test and verify other applications before putting them into online situations with real calls.

Factors affecting link traffic

The following factors can have a significant affect on link traffic:

Length of call

An application average call length of less than 15 seconds will put an above normal amount of traffic on the ACCESS link. There are few IVR applications where calls on average are this short. This is because call setup and disconnection usually are the most API intensive parts of the calls.

Number of DTMF digits being collected

Each DTMF digit generates a message which invokes your application digit event handler. If your application uses 12 digit account numbers and/or passwords, is used by experienced callers only, and command type ahead is supported, average call length will be short. A voice-menu like application usually has relatively little DTMF input and a lot of voice playback which results in little link traffic and a long average call time.

Number of voice segments

When playing a series of prompts together such as an account balance, the application can minimize the number of times the function m_PlaySegs needs to be called. Take for example an application wishing to play the following account balance prompt: “Your savings account balance is currently $1488.45”. A well written application could play this entire prompt to the caller using one m_PlaySegs request. Another application could issue up to 10 or more individual m_PlaySegs requests and accomplish the same thing.

Length of prompts being played

Once prompt playback for an application has been initiated, no application link traffic is normally generated for this call until the prompt completes its playback. Long prompts lead to a longer average call length and a decrease in link traffic.

Optimum usage of API functions

Never write code which loops continually trying to perform an API without a delay between each call. For example, in your application recovery logic, if an m_Acquire fails with an MME_NO_TASK error, a small delay (that is, two or more seconds) should be introduced so the application attempting to recover a channel does not monopolize the link.

Mainframe

The following applies if your IVR application includes some form of mainframe or host access:

System Capacity

If your application is accessing a mainframe or other host for database information you should calculate the anticipated load that the IVR application can place on the system.

Database throughput

Callers will only wait so long for information. If the host system cannot respond fast enough with information to the caller they may hang up or not use the service in the future.

Rigorous testing is always important to ensure an application will be a reliable product. For example if an IVR application is supposed to support 32 host connections and 32 voice ports simultaneously, then it should be tested at this peak load for several hours to ensure that the UNIX workstation, mainframe, PBX, and so on, in the configuration have sufficient resources such as cpu and communications bandwidth available.

Chapter 7: Telephony

Meridian Mail can be connected to a variety of different switches including but not limited to the Meridian 1, DMS-100, or SL-100. In all of these configurations, there is both a voice connection and a data connection between Meridian Mail and the switch. The voice connection provides the speech path and is used to transfer voice data during such things as record and playback of voice. The type of data connection, either Simplified Message Desk Interface (SMDI) or Application Module Link/Computer Switch Link (AML/CLS) determine how calls are set up and controlled.

Your application design will be affected by the type of switch you are connected to, because all switches do not necessarily provide the same capability to Meridian Mail. Application developers writing an application intended to be run on more than one type of switch should read this chapter, and note the differences between an SMDI and AML/CSL environment.

AML/CSL configurations

The AML/CSL is an intelligent link provided between the Meridian 1 and Meridian Mail. It is a proprietary synchronous protocol. When present, it is used to pass all telephony related commands and status messages including message waiting indication (MWI).

In the event that the AML/CSL link goes down, existing calls in progress remain unaffected. All telephony-related commands and status messages including incoming calls announcement will cease to function.

Using the m_AcquireOnIncoming Call (AOIC) function with AML/CSL configuration causes the Toolkit not to terminate once a call has hung up. As a result, the Toolkit makes the voice channel unavailable to calls.

For the Toolkit to receive any more calls, you must issue the m_AcquireOnIncomingCall (AOIC) command. If using the m_Acquire function with AML/CSL configuration, you must issue the m_AcceptCall command to receive any more calls.

SMDI Configurations

SMDI, on the other hand, is an industry standard asynchronous protocol primarily used for voice messaging support. It is quite limited in functionality and provides only incoming call announcement and MWI control. Telephony commands (such as transfer and so forth) are handled by Meridian Mail through switch hook flashes and are considerably slower than in an AML/CSL environment. Note that DNIS and CLID are not available.

In the event that the SMDI link goes down, existing calls remain intact. All telephony commands will still work, and incoming calls will be presented; however, the incoming call information will not be available.

Using AOIC with SMDI configuration causes the Toolkit to terminate after you hang up a call. As a result, the voice channel can accept calls.

Voice Service DN Table

When a call reaches Meridian Mail, the Voice Service DN (VSDN) table is used to control the type of service which will be started. The VSDN table contains a list of recognizable DN's for which known voice services exist.

If you log on to the Meridian Mail administration terminal, you can view the VSDN table which displays a field labelled the Access DN. The switch typically passes an Access DN (the originally dialed number) to Meridian Mail via the data link, as the call arrives. If a call terminates on an ACCESS voice service, the dialed number, which is referred to as the ToDN in the ACCESS programming interface, is passed to the application's incoming call event handler.

There are several factors which may influence the actual ToDN value assigned to a call. These are whether the call originates internally on the switch or externally on a network, and the type of trunk on which the external calls arrive.

When a call is forwarded to Meridian Mail by the switch, there are a number of associated identifiers which Meridian Mail is aware of, one of which gets picked and assigned as the ToDN. It is important to understand what these identifiers are as they affect which voice service can be started, and the information passed to the ACCESS applications when the call arrives.

In general, the number which a caller dials is used by the system to determine what the caller wants to do. This permits a single system to provide a variety of voice services without having to query the caller directly for this.

When initiating a voice service, Meridian Mail may have one or more of the following identifiers available. They are listed here in order of precedence:

Network supplied-dialed number

- ***Dialed Number Identification Service*** Calls which arrive on a Meridian 1 with Direct Inward Dial (DID) trunks configured for DNIS and are forwarded to Meridian Mail, will have the DNIS number passed to Meridian Mail. For example, if the caller dialed 1-800-533-2222, the value 2222 would be passed to Meridian Mail as the Dialed Number Identification Service (DNIS).

- ***PRA Trunks*** Calls which arrive on a Meridian 1 with PRA trunks configured and are forwarded to Meridian Mail may have the dialed number assigned as the Called DN. For example, if a caller at 833-7470 dialled 533-2222, the phone number 533-2222 would be passed to Meridian Mail as the Called DN value.

  In this configuration, the FromDN 833-7470, more popularly known as the Calling Line Identification (CLID) will be passed to Meridian Mail and to the ACCESS application's incoming call event handler. ACCESS applications can use this number to determine who is actually dialing the voice service.

ACD Queue DN (ACD DN)

- ***External Switch Calls*** If the type of incoming trunk provides no network information to the Meridian 1, Meridian Mail assumes the ACD DN of the queue onto which the trunk is routed to is the ToDN.

- ***Internal Switch Calls*** For calls that originate internally on the Meridian 1 PBX and are forwarded to Meridian Mail, both the FromDN and ToDn are always available. For example, if extension 7470 dials the voice service ACD queue 3650, the ToDN is the ACD DN 3650, and the FromDN is the extension of the caller which in this case is 7470.

  In the case of a call being forwarded from one ACD queue to another ACD queue, only the originally dialed ACD queue DN is available as the FromDN.

- ***Channel DN*** Even if the SMDI link goes out of service, the Channel DN is always available from the hardware database.

- ***ToDN*** If the SMDI link goes out of service, in a SMDI configuration, the ACD DN assigned in the Channel Allocation Table will be substituted as the ToDN.

Handling Large VSDN Tables

For environments where there are a very large number of VSDN entries required, a shortcut method is available which you may be able to use that will simplify administration of the VSDN table.

When Meridian Mail receives the dialed number from the public network and there is no matching entry in the VSDN table, the primary ACD queue is used to determine the type of voice service. If the ACD DN of the primary ACD queue is present in the VSDN table, this voice service will be started; however, the ToDN will remain that which is passed in by the network.

For example, if you have 500 DNIS numbers being handled by your system, all of which terminate on the primary ACD queue DN, the only entry needed in the VSDN table is the primary ACD queue DN. All calls arriving will have the DNIS number assigned and passed as the ToDN to the ACCESS applications even though they are not in the VSDN table.

Call Model

For applications which need to monitor the call state at all times, it is useful to understand the Call Model. The following diagrams show the possible state transitions for different call scenarios.

Figure 7-1
Basic Outbound Call Model

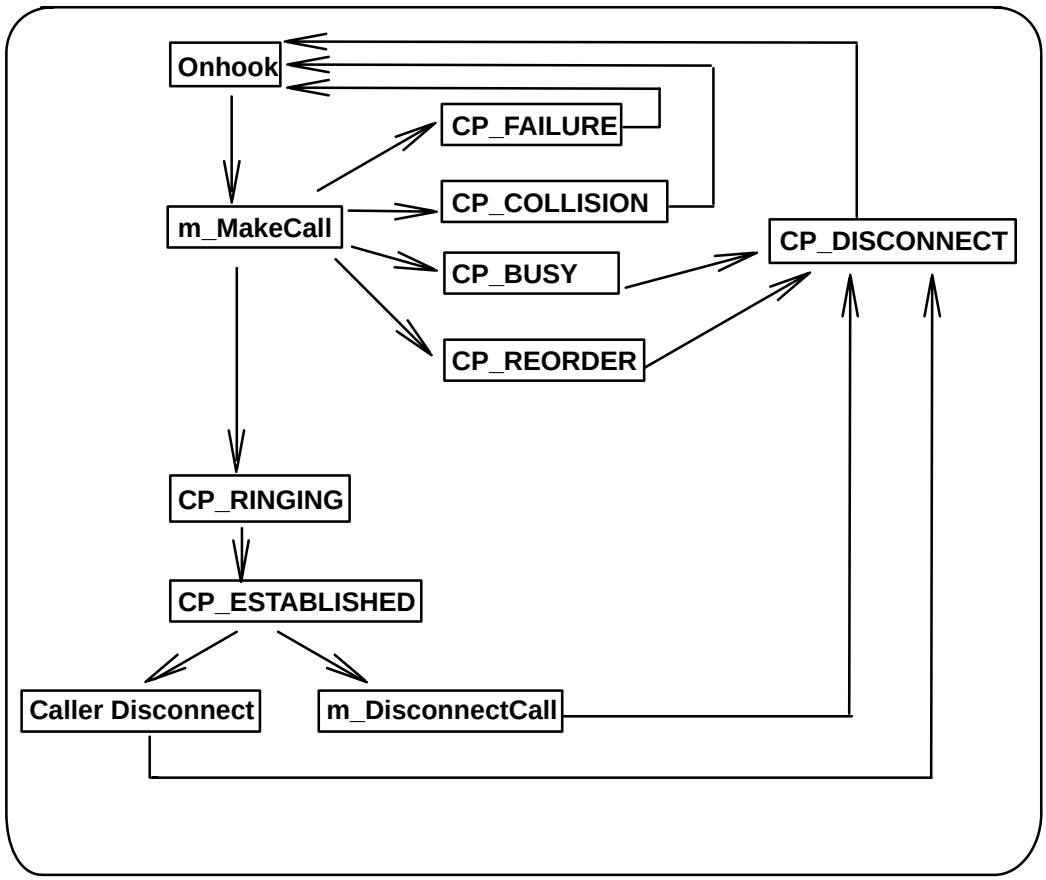
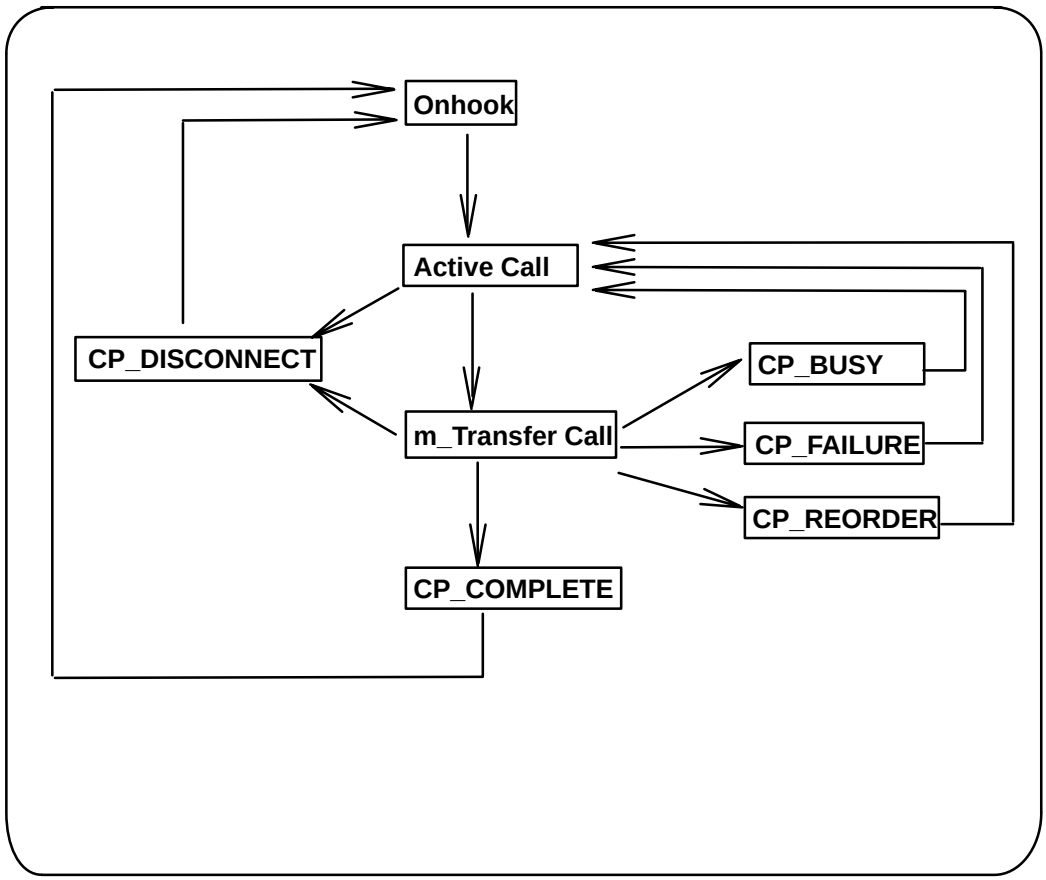
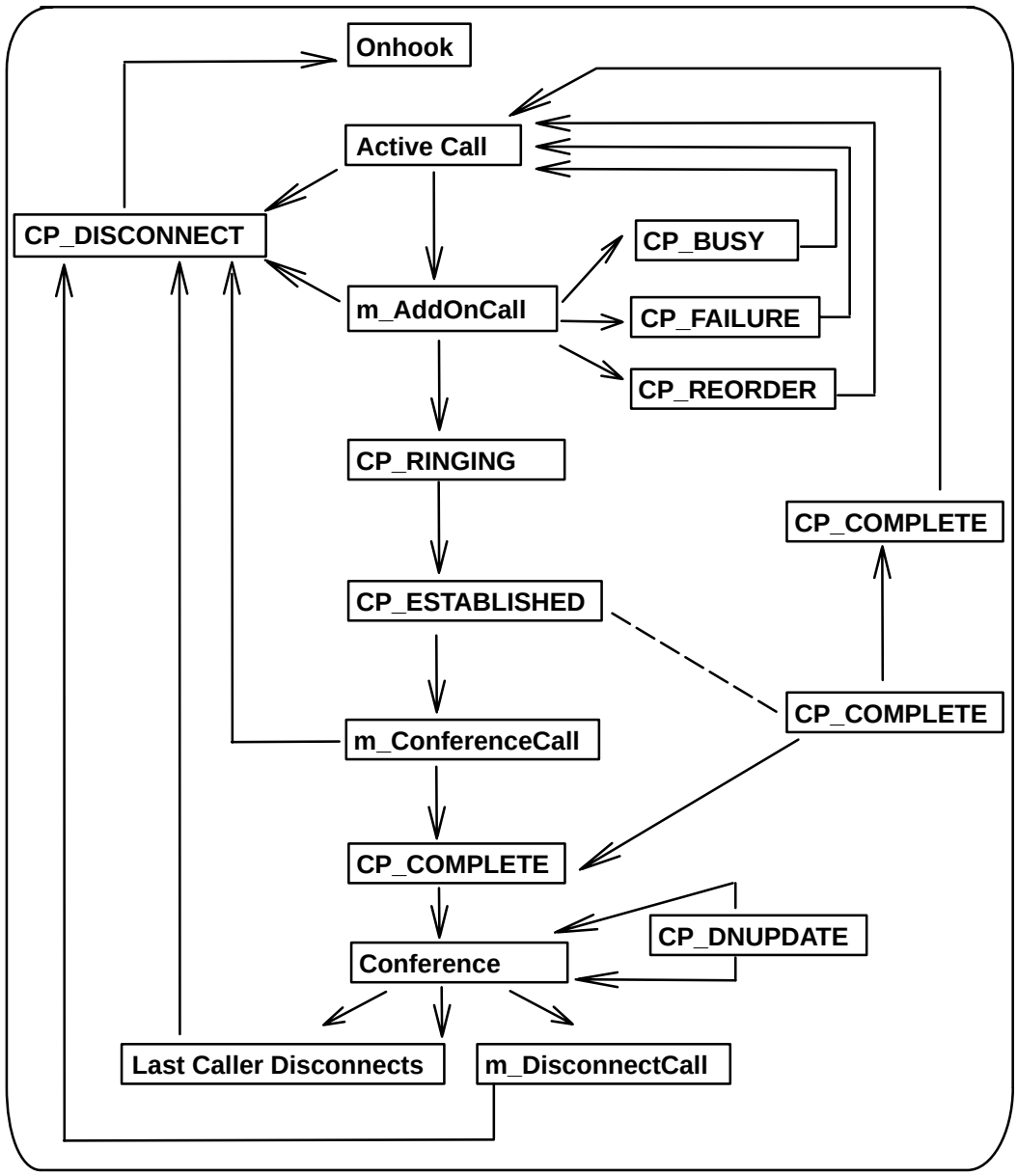


Figure 7-2
Transferring a Call



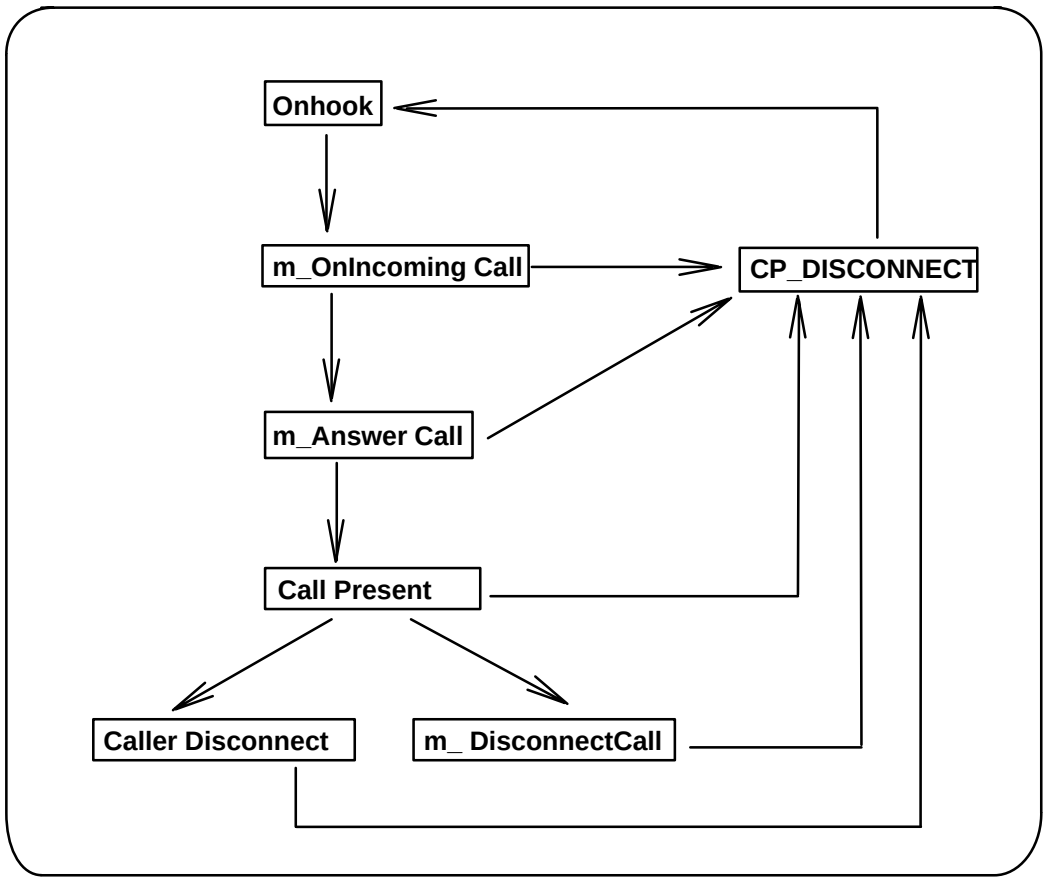
Note: The active call may have originated from either inside or outside of the switch.

Figure 7-3
Conferencing a Call



Note: The active call may have originated from either inside or outside of the switch.

Figure 7-4
Incoming Call Presentation



Switch Dependencies

Given that not all switches provide the same capability to Meridian Mail, there are some minor differences when operating an application in conjunction with them. The basic call model shown in Figures 7–1 through 7–4 remains intact and an application written for one environment should work in the other with the exception of the items discussed below.

Inbound Call Handling

The Accept Call Function

When using the m_Acquire function in an AML/CSL environment, you can block incoming calls so applications can perform post call processing. You do not have to worry about being ready to handle the next incoming call because calls are not presented to an application unless it asks for one. After the post call processing work is complete, the application can ask for a call by issuing the m_AcceptCall function. The next call waiting (or if none, the next arriving) in queue will be presented to the application.

In an SMDI environment, calls must be honored when they arrive and are presented to the voice port. Once a channel has been acquired, a call may be presented at any time to the ACCESS application. Similarly, once an application has received the call disconnect event, a call waiting in queue will be presented immediately. The m_AcceptCall function has no effect in this environment and is supported only to maintain compatibility of ACCESS applications between switch types.

Outbound Call Handling

Call Progress Events

When calls originate (that is, m_MakeCall) from applications, the calls return one or more events to their call progress event handlers indicating the status of the call. As the same call model is used for all switches, the same call states will exist. Depending on the type of switch being used, the current state of the call may be returned from the switch itself or by Meridian Mail voice port DSP.

For calls which originate and terminate entirely within the Meridian 1 itself, quick and reliable call state information is always available. For all calls which go outside of the Meridian 1, Meridian Mail itself determines the call state including answer supervision.

Answer supervision messages which may arrive from the public network are not passed onto ACCESS applications because Meridian Mail can usually detect an answer faster than the network can supply this indication to the switch.

Calls which originate on switches other than the Meridian 1 will always have Meridian Mail determine the call state such as ringing, busy, reorder, and so on.

CP_DNUPDATE Call Progress Event

This event is available only in AML/CSL environments. It occurs when someone joins a simple call and thus makes it a conference call, when someone leaves a conference call and thus reduces it to a simple call, and when an attendant (for example. secretary or service) transfers an external caller and accesses a person's Meridian Mail or when ACCESS transfers a call.

CP_RINGING Call Progress Event

This event will always be received in an AML/CSL environment for internal switch calls. For external switch calls in AML/CSL environment, and for all calls in SMDI environments a full ring cycle must occur before CP_RINGING event will be returned. Therefore, it is possible that a call may be answered and a CP_RINGING event will never be presented to the application.

CP_ESTABLISHED Call Progress Event

This event indicates that the call has been answered.

CP_BUSY Call Progress Event

This event indicates that the called party is busy.

CP_REORDER Call Progress Event

This event indicates that the call has been rejected.

CP_FAILURE Call Progress Event

This event indicates that the call connection attempt has failed.

CP_COMPLETED

This event indicates that the call transfer, conference, or reconnect was successful.

CP_DISCONNECT

This event indicates that the set has gone on-hook.

CP_COLLISION Call Progress Event

All telephone calls, either incoming or outgoing, pass through the phone switch. When using the same voice channel for both inbound and outbound calls, one must realize that there could be contention between a call arriving and being assigned to a voice port, and the voice port attempting to initiate a call. It is simpler if you avoid this condition. However, if it is necessary, the following should be read.

When using a voice channel for handling both inbound and outbound calls, it is possible to create a race condition where an inbound call arrives just as an application attempts to place an outbound call. When this happens, the application must honor the inbound call and try the outbound call at a later time. This prevents the incoming call from being lost (that is, the caller hears ringing followed by a disconnect click, and no voice service is provided). This is very undesirable and should always be avoided.

Meridian Mail/ACCESS enforces the honoring of an inbound call when possible. For example, in an AML/CSL configuration, the application can prevent a collision situation from arising by not issuing an m_AcceptCall. If an m_AcceptCall is issued and the application uses m_MakeCall, it is possible for a call collision to happen. When this occurs, the CP_COLLISION event is returned. After an application receives a CP_COLLISION event, it will immediately receive incoming call notification. It must then answer the incoming call and attempt the outgoing call at a later time. If the application fails to answer the call, it will be sent a session disconnect event.

For Meridian Mail GP systems with SMDI, the collision condition cannot be handled in a graceful manner. The incoming call will be lost, and the outgoing call will fail. It is not recommended that applications attempt to use the same voice port for both inbound and outbound applications. In this configuration, separate queues should be configured on the switch for inbound call traffic. This dedicates outbound channels and avoids call collisions from happening.

For MSM systems, when a collision occurs, it will be reported to the application which must then answer the incoming call. Although more work for the application developer, it does allow inbound and outbound calls to be handled in a graceful manner .

m_AcquireOnIncomingCall

In SMDI environments where Meridian Mail is not able to provide call blocking, calls must be honored as they arrive. When using the m_AcquireOnIncomingCall method to process calls, the application will receive a Session Disconnect event at the end of each call as opposed to a Call Disconnect event. This ensures that the next incoming call will always be honored by other applications which have outstanding AOIC request waiting to be serviced.

m_GetCallInfo

This API function returns more detailed information about a call in progress. Not all fields are valid all the time. This function is usually used in conjunction with outbound applications or those that transfer inbound calls which have arrived. The CallInfo structure returned by this API contains switch-specific information as indicated below:

CallState	- Always valid
CallInfo	- Always valid
CallingDN	- Always valid if available
CallingDNType	- AML/CSL configurations only
CalledDN	- Always valid if available
CalledDNType	- AML/CSL configurations only
CalledTN	- AML/CSL (outbound calls to internal DNs)
CallType	- AML/CSL configurations only
DeviceType	- AML/CSL configurations only
OtherDN	- AML/CSL configurations with DNIS configured
CallId	- AML/CSL configurations with CCR option

Appendix A: Sample program listings

This appendix contains listings for the sample programs supplied with Meridian ACCESS, and the header file corresponding to the voice segment file used in the sample programs. Each listing describes the function of the program, the configuration required to run the program, and the running of the program.

Following is a summary of the sample programs:

KEYSEGS
This file is called by many of the sample programs and contains indices for voice segments in the voice segment file.

PHONEME
This program calls a telephone number and plays a prompt, demonstrating the rudimentary ACCESS requirements for outbound calling.

MAILME
This program calls a telephone number and prompts the user for a message. The message is recorded and sent by the Meridian Mail voice messaging system to a recipient. This program demonstrates integration with Meridian Mail and the recording of a user message.

WHATKEY
This program answers incoming calls to one Meridian Mail channel, and then speaks the name of keys pressed on the calling telephone. This demonstrates the event handling required for answering inbound calls and detecting keypresses.

WHATLINE
This application answers calls made to various telephone numbers connected to the same Meridian Mail channel and then announces which number was called. This demonstrates the ability of a Meridian ACCESS application to perform different functions depending on the called number.

LOTTO

This application accepts incoming calls, generates a series of random numbers in a designated range, and plays these numbers back to the caller. For more information on this application, see “Chapter 3: Development guidelines”. The lotto.h (include file) and lotto.c (code) are both listed here.

KEYSEGS

```
/*
 * KEYSEGS.H — defines the indices into the WhatKey voice segment file
 * for the various prompts in that file
 *
 * Copyright (C) Northern Telecom Limited, 1991 – 1993
 *
 */
                    */
#define ONE         1     /* “one” */
#define TWO         2     /* “two” */
#define THREE             3     /* “three” */
#define FOUR        4     /* “four” */
#define FIVE        5     /* “five” */
#define SIX         6     /* “six” */
#define SEVEN       7     /* “seven */
#define EIGHT       8     /* “eight” */
#define NINE        9     /* “nine” */
#define ZERO        10    /* “zero” */
#define ASTERISK    11    /* “asterisk” */
#define OCTOTHORPE  12    /* “octothorpe” */
#define PRESSKEY          13    /* “Press any key, it will be echoed” */
#define NUMDIALED   14    /* “The number dialed is” */
#define HANGUP      15    /* “Please hang up the phone now.” */
```

PHONEME

This program prompts the user at the terminal for a Meridian Mail account number, password, and telephone number. It then calls the telephone, plays a voice file stored in that account and waits for keyboard input which indicates the termination of the program. A free ACCESS channel is required to run PHONEME.

```
/*********************************************************************/
/*                                                                   */
/* PHONEME                                                           */
/*                                                                   */
/* This program prompts the user for a Meridian Mail account         */
/* number, password, and telephone number. It then calls the         */
/* telephone, plays a voice file stored in that account and waits    */
/* for keyboard input which indicates the termination of the         */
/* program. A free ACCESS channel is required to run PHONEME.        */
/*                                                                   */
/* Synopsis: phoneme                                                 */
/*                                                                   */
/* Copyright (C) Northern Telecom Limited, 1990 – 1993               */
/*                                                                   */
/*********************************************************************/

static char * version = "@(#)phoneme.c       1.3 4/3/91   (NT)";

#include <machine.h>     /* Machine dependent macros                 */
#include <stdio.h>       /* UNIX file for standard I/O routines      */
#include <termio.h>      /* UNIX file for term I/O control           */
#include <ctype.h>       /* Char class. & conversion routines        */
#include <m_acc.h>       /* Meridian Mail ACCESS general functions*/
#include <m_rm.h>        /* Meridian Mail ACCESS resource mgmt       */
#include <m_local.h>     /* Meridian Mail ACCESS local access        */
#include <m_file.h>      /* Meridian Mail ACCESS file access         */
#include <m_voice.h>     /* Meridian Mail ACCESS voice operation     */
/* Valid program status levels, used by "ExitProgram" routine */
#define NOTREGISTERED 0  /* Have not registered with the ACCESS      */
#define REGISTERED    1  /* Have registered with the MM ACCESS       */
#define ACQUIRED      2  /* Have acquired a MM ACCESS session        */
#define MAX_TIME      20 /* Max time for connect to complete (sec)*/
/* Prototypes */
                /* Main Program                                      */
void main PROTO((int,char **));
                /* Routine to gracefully exit from program*/
static void ExitProgram PROTO((short));
                /* Get a string from terminal keyboard.  */
void getNumericStr PROTO((char *,short,short));
                /* set terminal for punctual input.         */
void setPunctual PROTO((void));
```

```
                    /* save original terminal settings        */
void saveTerm PROTO((void));
                    /* restore terminal to original setting   */
void restoreTerm PROTO((void));
/* Global Variables */
                    /* Information struct for terminal control*/
static      struct   termio tbufsave;
/*            */
/* Main program */
/*            */
void main (argc,argv)
int   argc;
char **argv;
{
  short  LogonInfo;        /* Info parameter from an ACCESS logon    */
  short  RetCode;    /* Return code from an ACCESS function    */
  short  FileHandle;   /* File handle for opened voice file      */
  char PhoneNum[DN_SIZE];      /* Tele num. of user's local phone*/
  char AccountNum[USERID_SIZE];/* MM account to get voice file   */
  char Password[PSWD_SIZE];    /* Password for account           */
  /* Retrieve the program argument (message queue key). */
  /* If not there exit with usage message.              */
  if (argc > 1) {
    printf("usage: phoneme \n\n");
    exit(0);
  }
  /* Save the orignal terminal mode                     */
  saveTerm();
  /* Set the host terminal characteristics for punctual */
  /* input and no echo (i.e. input characters are not   */
  /* assembled into lines before they are read, but     */
  /* made available as they are typed).                 */
  setPunctual();
  /* Get account number, password, and telephone number */
  /* from the user. Do not echo password. */
  printf("\nEnter your Meridian Mail account number: ");
  getNumericStr(AccountNum,USERID_SIZE,TRUE);
  printf("\nPassword: ");
  getNumericStr(Password,PSWD_SIZE,FALSE);
  printf("\nEnter the telephone number to be called: ");
  getNumericStr(PhoneNum,DN_SIZE,TRUE);
  /* Register with Meridian Mail ACCESS.                */
  /* Establish local resources for an application to perform */
  /* communications with ACCESS link handler            */
  if (!m_Register(&RetCode)) {
    printf("\nUnable to register with MM ACCESS. Error=%d",
          RetCode);
    ExitProgram(NOTREGISTERED);
  }
```

```
/* Acquire a MM ACCESS session. Use non–dedicated voice channel */
/* (AC_SHARED) and do not allow incoming calls to that channel  */
if (!m_Acquire(AC_SHARED,&RetCode)) {
  printf("\nUnable to acquire an ACCESS session. Error=%d",
        RetCode);
  ExitProgram(REGISTERED);
}
/* Logon to the account containing the PHONEME voice file */
if (!m_Logon(AccountNum,Password,&LogonInfo,&RetCode)) {
  printf("\nUnable to logon to account. Error=%d",RetCode);
  ExitProgram(ACQUIRED);
}
/* Open the PHONEME voice file for reading only. */
if (!m_OpenFile("PhoneMe","r",&FileHandle,&RetCode)) {
  printf("\nUnable to open PhoneMe voice file. Error=%d",RetCode);
  ExitProgram(ACQUIRED);
}
printf("\nPlease pick up the phone when it starts to ring...");
/* Initiate a voice connection from the acquired channel to */
/* a telephone and wait (TR_ON_COMPLETE) up to MAX_TIME for */
/* the connection to be established                         */
if (!m_MakeCall(PhoneNum,TR_ON_COMPLETE,MAX_TIME,&RetCode)) {
  printf("\nUnable to connect to telephone %s. Error=%d",
        PhoneNum,RetCode);
  ExitProgram(ACQUIRED);
}
/* Play the PHONEME voice file from the beginning (TRUE) */
if (!m_PlayVoice(FileHandle,TRUE,&RetCode)) {
  printf("\nUnable to play PHONEME voice file. Error=%d",
        RetCode);
  ExitProgram(ACQUIRED);
}
/* Wait for a keypress on keyboard before exiting the program */
printf("\n\nPress any key to exit PHONEME");
printf(" once the voice prompt has been played:");
getchar();
/* Release ACCESS session (logs off & closes files) */
if (!m_Release(&RetCode)) {
  printf("\nUnable to release ACCESS Session. Error=%d",
        RetCode);
  ExitProgram(REGISTERED);
}
/* Deregister from Meridian Mail ACCESS */
if (!m_Deregister(&RetCode)) {
  printf("\nUnable to deregister from MM ACCESS. Error=%d"
        ,RetCode);
  ExitProgram(NOTREGISTERED);
}
```

```
  /* Restore the host terminal characteristics to */
  /* their original settings and exit            */
  restoreTerm();
  exit(0);
} /* End of main */
/*                                    */
/* This routine will gracefully exit the program */
/*                                    */
static void ExitProgram (StatusLevel)
short StatusLevel; /* Current program status level */
{
  short ErrCode;  /* Error code returned by MM ACCESS function  */
  /* Exit the program gracefully – no need to check error codes */
  /* since are only interested in terminating the program.     */
  switch (StatusLevel) {
    case ACQUIRED     : /* Have acquired a MM ACCESS session   */
                m_Release(&ErrCode);
    case REGISTERED   : /* Have registered with the MM ACCESS  */
                m_Deregister(&ErrCode);
    case NOTREGISTERED: /* No MM ACCESS connections up yet     */
                restoreTerm();
                exit(0);
    } /* switch */
} /* End of ExitProgram */

/*                                          */
/* This Routine will retrieve a string of numeric characters */
/* (character by character up to ”size” chars) and store them*/
/* in memory locations beginning at ”startLocn”. Echoing is  */
/* performed if requested.                  */
/*                                          */
void getNumericStr (strLocn,size,echo)
char *strLocn;    /* Mem location of destination string.    */
short  size;      /* Maximum number of char to be collected. */
short  echo;      /* Boolean – true if char are to be echoed.*/
{
 short stringEmpty = TRUE;
 char *charPtr   = strLocn;
                   /* Loop until entry is terminated */
                   /* by newline ’\n’ and string is  */
                   /* not empty                      */
 while (stringEmpty) {
   while ((*charPtr = getchar()) != ’\n’) {
     if (echo)
       putchar(*charPtr);
     if (!isdigit(*charPtr)) {  /* Char received not a numeric. */
       printf(“\nNumeric characters only.”);
       printf(“\n\nPlease reenter the number: ”);
       stringEmpty = TRUE; /* Restart the collection. Reset to */
```

```
        charPtr = strLocn;  /* the 1st location of destn string */
        break;
      }
      else {              /* Got a valid numeric char */
        stringEmpty = FALSE;
        charPtr++;        /* Dest addr of next char.  */
      }
      if ((charPtr – strLocn) == size – 1)  /* Leave a byte for */
        break;                        /* end of str char. */
    } /* end while *charPtr = getchar */
  } /* end while stringEmpty */
  *charPtr = '\0';            /* Set end of string char. */
} /* end getNumericStr */
/*                                                  */
/* This routine will set the host terminal characteristics */
/* for punctual input with no echo. Input characters are   */
/* not assembled into lines before they are read, but      */
/* made available as they are typed.                       */
/*                                                  */
void setPunctual  ()
{
  struct termio tbuf;             /* terminal information buffer */
                        /* get the original terminal setting */
  tbuf = tbufsave;
                        /* set host terminal for punctual */
                        /* input, no echo, min 1 char buff*/
  tbuf.c_lflag &= (ICANON | ECHO);
  tbuf.c_cc[VMIN]  = 1;
  tbuf.c_cc[VTIME] = 0;
  if (ioctl(0,TCSETAF,&tbuf) == –1)
    printf("ioctl SET failed!\n");
} /* end setPunctual */
/*                                                  */
/* This routine will save the host terminal characteristics */
/* original settings in a global buffer                    */
/*                                                  */
void saveTerm ()
{
                        /* get host terminal characteristics */
  if (ioctl(0,TCGETA,&tbufsave) == –1)
    printf("ioctl GET (original) failed!\n");
} /* end saveTerm */
/*                                                  */
/* This routine will restore the host terminal characteristics */
/* to their original settings.                             */
/*                                                  */
void restoreTerm ()
{
                        /* set host terminal characteristics */
```

```
                        /* to the original settings.       */
  if (ioctl(0,TCSETAF,&tbufsave) == –1)
     printf("ioctl SET (original) failed!\n");
} /* end restoreTerm */
```

MAILME

This program prompts the user at the terminal for a Meridian Mail account number, password, telephone number and a recipient's account number. It then calls the telephone, plays a message asking the user to record a message. After the message is recorded, it is forwarded to the specified recipient. To run MAILME a free Meridian ACCESS channel is required.

```
/*********************************************************************/
/* MAILME                                                            */
/*                                                                   */
/* This program prompts the user for a Meridian Mail account         */
/* number, password, telephone number and a recipient's account      */
/* number. It then calls the telephone, plays a message asking       */
/* the user to record a message. After the message is recorded,      */
/* it is forwarded to the specified recipient. To run MAILME a       */
/* free ACCESS channel is required.                                  */
/*                                                                   */
/* Synopsis: mailme                                                  */
/*                                                                   */
/* Copyright (C) Northern Telecom Limited, 1990 – 1993               */
/*                                                                   */
/*********************************************************************/

static char * version = "@(#)mailme.c  1.2   4/3/91  (NT)";

#include <machine.h> /* machine dependent definitions               */
#include <stdio.h>   /* UNIX file for standard I/O routines         */
#include <termio.h>  /* UNIX file for term I/O control routines     */
#include <ctype.h>   /* Character class. & conversion routines      */
#include <m_acc.h>   /* Meridian Mail ACCESS general functions      */
#include <m_rm.h>    /* Meridian Mail ACCESS resource mgmt          */
#include <m_local.h> /* Meridian Mail ACCESS local access           */
#include <m_file.h>  /* Meridian Mail ACCESS file access            */
#include <m_msg.h>   /* Meridian Mail ACCESS messaging              */
#include <m_seg.h>   /* Meridian Mail ACCESS voice segment          */
#include <m_voice.h> /* Meridian Mail ACCESS voice operations       */
#include <m_event.h> /* Meridian Mail ACCESS event handler          */
#define MAX_TIME 20  /* Max time for connect to complete (sec)      */
#define LEAVEMSG  1  /* Segment offset in the "MAILME" seg file     */
#define LIST_END  0  /* Marks the end of list of voice seg indices */
/* Valid program status levels, used by "ExitProgram" routine       */
#define NOTREGISTERED 0   /* Have not registered with MM ACCESS     */
#define REGISTERED    1   /* Have registered with the MM ACCESS     */
#define ACQUIRED      2   /* Have acquired a MM ACCESS session      */
/* Prototypes */
                /* Main program                                    */
void main PROTO((int,char **));
                /* Routine to gracefully exit from program*/
```

```
static void ExitProgram PROTO((short));
                /* Get a string from terminal keyboard.  */
void getNumericStr PROTO((char *,short,short));
                /* set terminal for punctual input.      */
void setPunctual PROTO((void));
                /* save original terminal settings       */
void saveTerm PROTO((void));
                /* restore terminal to original setting  */
void restoreTerm PROTO((void));
                 /* Call Progress event handler          */
static void CallProgHandler PROTO((short,short));
/* Global Variables */
char FullName[FULLNAME_SIZE]; /* Full name of user addressed      */
char FullBox[BOX_SIZE];       /* Box number of user addressed     */
char CallActive;              /* BOOLEAN – TRUE if call is active */
                        /* Info struct for term control     */
static       struct   termio tbufsave;
/*            */
/* Main program */
/*            */
void main (argc,argv)
int   argc;
char **argv;
{
  short  LogonInfo;         /* Info parameter from ACCESS logon  */
  short  RetCode;           /* Return code from ACCESS functions */
  short  FileHandle;        /* File handle for opened voice file */
  short  SegList[SEGLIST_SIZE]; /* List of voice segments to play */
  char PhoneNum[DN_SIZE];       /* Tele num of user's local phone */
  char AccountNum[USERID_SIZE]; /* Acc containing voice seg file  */
  char Password[PSWD_SIZE];     /* Password of account            */
  char Recipient[USERID_SIZE];  /* Acc number of the recipient    */
  /* Retrieve the program argument (message queue key). */
  /* If not there exit with usage message.              */
  if (argc > 1) {
    printf("usage: mailme \n\n");
    exit(0);
  }
  /* Save the origanl host terminal settings            */
  saveTerm();
  /* Set the host terminal characteristics for punctual */
  /* input and no echo (i.e. input characters are not   */
  /* assembled into lines before they are read, but     */
  /* made available as they are typed).                 */
  setPunctual();
  /* Install the Call Progress" event handler routine   */
  m_OnCallProgress(CallProgHandler);
  /* No Call is not active at this point                */
  CallActive = FALSE;
```

```
/* Get account# & password, telephone#, and          */
/* recipients account# from the user                 */
printf("\nEnter your Meridian Mail account number: ");
getNumericStr(AccountNum,USERID_SIZE,TRUE);
printf("\nPassword: ");
getNumericStr(Password,PSWD_SIZE,FALSE);
printf("\nEnter the telephone number to be called: ");
getNumericStr(PhoneNum,DN_SIZE,TRUE);
printf("\nEnter Recipient's Meridian Mail account number: ");
getNumericStr(Recipient,USERID_SIZE,TRUE);
/* Register with Meridian Mail ACCESS.               */
/* Establish local resources for an application to perform */
/* communications with ACCESS link handler           */
if (!m_Register(&RetCode)) {
  printf("\nUnable to register with MM ACCESS. Error=%d",
         RetCode);
  ExitProgram(NOTREGISTERED);
}
/* Acquire a MM ACCESS session. Use non–dedicated voice channel */
/* (AC_SHARED) and do not allow incoming calls to that channel  */
if (!m_Acquire(AC_SHARED,&RetCode)) {
  printf("\nUnable to acquire an ACCESS session. Error=%d",
         RetCode);
  ExitProgram(REGISTERED);
}
/* Log in to the account containing the MAILME voice segments. */
if (!m_Logon(AccountNum,Password,&LogonInfo,&RetCode)) {
  printf("\nUnable to logon to account. Error=%d",RetCode);
  ExitProgram(ACQUIRED);
}
/* Open the MAILME file for reading only. */
if (!m_OpenFile("MailMe","r",&FileHandle,&RetCode)) {
  printf("\nUnable to open MAILME voice segment file. Error=%d",
         RetCode);
  ExitProgram(ACQUIRED);
}
printf("\nPlease pick up your phone when it starts to ring...");
/* Initiate a voice connection from the MM ACCESS session to */
/* a telephone(PhoneNum) and wait (TR_ON_COMPLETE) up to     */
/* MAX_TIME for the connection to be established.            */
if (!m_MakeCall(PhoneNum,TR_ON_COMPLETE,MAX_TIME,&RetCode)) {
  printf("\nUnable to connect to specified number. Error=%d",
         RetCode);
  ExitProgram(ACQUIRED);
}
CallActive = TRUE;          /* Call is active at this point */
printf("\n\nWhen message has finished playing,");
printf(" press any key to begin recording...");
/* Play the MAILME voice segment. */
```

```
 SegList[0] = LEAVEMSG;        /* Set up a segment list to played */
 SegList[1] = LIST_END;        /* Only 1 segment to be played     */
 if (!m_PlaySegs(FileHandle,SegList,FALSE,&RetCode)) {
   printf("\nUnable to play MAILME voice segment. Error=%d",
          RetCode);
   ExitProgram(ACQUIRED);
 }
 /* Wait until a key is hit on the keyboard */
 getchar();
 /* Close the MAILME file */
 if (!m_CloseFile(FileHandle,FALSE,&RetCode)) {
   printf("\nUnable to close MAILME file. Error=%d",RetCode);
   ExitProgram(ACQUIRED);
 }
/* Create a new voice msg file "CustomerMsg" */
/* to record customer's message             */
if (!m_CreateFile("CustomerMsg",CL_VOICE_MESSAGE,
   &FileHandle,&RetCode)) {
   printf("\nUnable to create new voice message file. Error=%d",
          RetCode);
   ExitProgram(ACQUIRED);
}
 /* Address the message to the specified account number  */
 /* (Recipient) Full username & box# returned (not used) */
 if (!m_AddBoxToAddr(FileHandle,Recipient,FullName,FullBox,
     &RetCode)) {
   printf("\nUnable to address voice message. Error=%d",
          RetCode);
   ExitProgram(ACQUIRED);
 }
 /* Give the message a subject: sender's user ID. */
 if (!m_SetFileSubject(FileHandle,AccountNum,&RetCode)) {
   printf("\nUnable to give subject to voice message. Error=%d",
          RetCode);
   ExitProgram(ACQUIRED);
 }
 if (!CallActive)    /* Check if Call is still active */
   ExitProgram(ACQUIRED);
 /* Start recording of the message from the beginning */
 /* (TRUE) of the voice file                          */
 if (!m_RecordVoice(FileHandle,TRUE,&RetCode)) {
   printf("\nUnable to start recording of voice message. Error=%d",
          RetCode);
   ExitProgram(ACQUIRED);
 }
 printf("\n\nPlease leave a message...");
 printf(
  "\n...recording will stop when you press any key on keyboard.");
```

```
  /* Keep recording the voice message until a key is */
  /* hit on the keyboard or the phone goes on–hook   */
  getchar();
  /* Send the voice message (file is closed automatically) */
  if (!m_SendMsg(FileHandle,&RetCode)) {
    printf("\nUnable to send voice message. Error=%d",RetCode);
    ExitProgram(ACQUIRED);
  }
  printf(
   "\n\nYour message has been sent to Meridian Mail user %s.",
    Recipient);
  printf("\nSubject of the message is: %s", AccountNum);

  if (CallActive)
    printf("\n\nPlease hang up your phone now...");
    printf("Thank you for using Meridian Mail.");
  /* Release ACCESS session (logs off & closes files) */
  if (!m_Release(&RetCode)) {
    printf("\nUnable to release ACCESS Session. Error=%d",RetCode);
    ExitProgram(REGISTERED);
  }
  /* Deregister from the Meridian Mail ACCESS Gateway */
  if (!m_Deregister(&RetCode)) {
    printf("\nUnable to deregister from MM. Error=%d",
           RetCode);
    ExitProgram(NOTREGISTERED);
  }
  /* Restore Host terminal characteristics to */
  /* there original settings                  */
  restoreTerm();
  exit(0);
}  /* End of main */
/*                                          */
/* This routine will gracefully exit the program */
/*                                          */
static void ExitProgram (StatusLevel)
short StatusLevel; /* Current program status level */
{
  short ErrCode;  /* Error code returned by MM ACCESS function  */
  /* Exit the program gracefully – no need to check error codes */
  /* since are only interested in terminating the program.     */
  switch (StatusLevel) {
    case ACQUIRED     : /* Have acquired a MM ACCESS session   */
                m_Release(&ErrCode);
    case REGISTERED   : /* Have registered with the MM ACCESS  */
                m_Deregister(&ErrCode);
    case NOTREGISTERED: /* No MM ACCESS connection yet         */
                restoreTerm();
                exit(0);
```

```
    } /* switch */
} /* End of ExitProgram */
/*                                                    */
/* "Call Progress" Event Handler Routine;             */
/* Automatically called when current call has changed states.   */
/*                                                    */
static void CallProgHandler (StateChange,StateInfo)
short StateChange;     /* New State of call      */
short StateInfo;       /* Info about StateChange */
{
  /* Only expected state is CP_DISCONNECT.   */
  /* Set CallActive to let mainline know      */
  /* that call has disconnected.              */
  if (StateChange == CP_DISCONNECT)
       CallActive = FALSE;
} /* End of CallProgressHandler */
/*                                                    */
/* This Routine will retrieve a string of numeric characters */
/* (character by character up to "size" chars) and store them*/
/* in memory locations beginning at "startLocn". Echoing is  */
/* performed if requested.                            */
/*                                                    */
void getNumericStr (strLocn,size,echo)
char *strLocn;    /* Mem location of destination string      */
short  size;      /* Maximum number of char to be collected. */
short  echo;      /* Boolean – true if char are to be echoed.*/
{
 short stringEmpty = TRUE;
 char *charPtr   = strLocn;
                  /* Loop until entry is terminated */
                  /* by newline '\n' and string is  */
                  /* not empty                      */
 while (stringEmpty) {
   while ((*charPtr = getchar()) != '\n') {
     if (echo)
       putchar(*charPtr);
     if (!isdigit(*charPtr)) {  /* Char received not numeric */
       printf("\nNumeric characters only.");
       printf("\n\nPlease reenter the number: ");
       stringEmpty = TRUE;     /* Restart collection. Reset */
       charPtr = strLocn;      /* to 1st location of destn  */
       break;
     }
     else {                    /* Got a valid numeric char  */
       stringEmpty = FALSE;
       charPtr++;              /* Dest addr of next char.  */
     }
     if ((charPtr – strLocn) == size – 1) /* Leave a byte    */
       break;                          /* for end of str  */
```

```
    } /* end while *charPtr = getchar */
  } /* end while stringEmpty */
  *charPtr = '\0';            /* Set end of string char. */
} /* end getNumericStr */
/*                                          */
/* This routine will set the host terminal characteristics */
/* for punctual input with no echo. Input characters are   */
/* not assembled into lines before they are read, but      */
/* made available as they are typed.                       */
/*                                          */
void setPunctual  ()
{
   struct termio tbuf;       /* terminal information buffer */
                     /* get the original terminal setting */
   tbuf = tbufsave;
                     /* set host terminal for punctual */
                     /* input, no echo, min 1 char buff*/
   tbuf.c_lflag &= ~(ICANON | ECHO);
   tbuf.c_cc[VMIN]  = 1;
   tbuf.c_cc[VTIME] = 0;
   if (ioctl(0,TCSETAF,&tbuf) == –1)
      printf("ioctl SET failed!\n");
} /* end setPunctual */
/*                                          */
/* This routine will save the host terminal characteristics */
/* original settings in a global buffer                     */
/*                                          */
void saveTerm ()
{
                     /* get host terminal characteristics */
   if (ioctl(0,TCGETA,&tbufsave) == –1)
      printf("ioctl GET (original) failed!\n");
} /* end saveTerm */
/*                                            */
/* This routine will restore the host terminal characteristics */
/* to their original settings.                              */
/*                                            */
void restoreTerm ()
{
                     /* set host terminal characteristics */
                     /* to the original settings.        */
   if (ioctl(0,TCSETAF,&tbufsave) == –1)
      printf("ioctl SET (original) failed!\n");
} /* end restoreTerm */
```

WHATKEY

This program services incoming calls by verbally prompting the caller to press a telephone keypad digit, and playing voice segments corresponding to the pressed digit. The program prompts for an account number and password of a Meridian Mail account containing the voice segment file required to service a call. It also prompts for the class of this application. The program will continue to answer calls until a key is pressed at the terminal keyboard. The program makes use of a buffer queue to save outstanding telephone key presses which are later echoed.

```
/*****************************************************************/
/* WHATKEY                                                       */
/*                                                               */
/* This program services incoming calls by verbally prompting    */
/* the caller to press a telephone keypad digit, and playing     */
/* voice segments corresponding to the pressed digit. The        */
/* program prompts for an account# and password of a Meridian    */
/* Mail account containing the voice segment file required to    */
/* service a call. It also prompts for the class of this         */
/* application. The program will continue to answer calls        */
/* until a key at the terminal keyboard is pressed. The program  */
/* makes use of a buffer queue to save outstanding telephone     */
/* key presses which are later echoed.                           */
/*                                                               */
/* Synopsis: whatkey                                             */
/*                                                               */
/* Copyright (C) Northern Telecom Limited, 1990 – 1993           */
/*                                                               */
/*****************************************************************/

static char * version = "@(#)whatkey.c 1.7 4/3/91   (NT)";

#include <machine.h>   /* Machine dependent macros               */
#include <stdio.h>     /* UNIX file for standard I/O routines    */
#include <termio.h>    /* UNIX file for term I/O routines        */
#include <ctype.h>     /* Char class. & conversion routines      */
#include <m_acc.h>     /* Meridian Mail ACCESS general functions*/
#include <m_rm.h>      /* Meridian Mail ACCESS resource mgmt     */
#include <m_local.h>   /* Meridian Mail ACCESS local access      */
#include <m_file.h>    /* Meridian Mail ACCESS file access       */
#include <m_seg.h>     /* Meridian Mail ACCESS voice segment     */
#include <m_voice.h>   /* Meridian Mail ACCESS voice operations */
#include <m_event.h>   /* Meridian Mail ACCESS event handler     */
#include "keysegs.h"   /* Indices for segments in voice seg file*/
#define QSIZE     20   /* Max # of outstanding digits to played */
#define LIST_END   0   /* Marks the end of a list of voice seg  */
                       /* indicies.                             */
```

```
#define BUFF_SIZE  6   /* # of chars required to house an int:  */
                       /* 5 digits plus an end of string char. */
/* Valid program status levels, used by "ExitProgram" routine   */
#define NOTREGISTERED 0  /* Not yet registered with MM ACCESS   */
#define REGISTERED    1  /* Have registered with MM ACCESS      */
#define ACQUIRED      2  /* Have acquired a MM ACCESS session   */
/* Prototypes */
                       /* Main program                  */
void main PROTO((int, char ** ));
                       /* Get numstring from terminal keybd*/
static void getNumericStr PROTO((char *,short, short));
                       /* Get an integer from PC keybd     */
static void CallProgressHandler PROTO((short,short));
                       /* Digit received event handler     */
static void DigitHandler PROTO((short));
                       /* Session Disconnect event handler */
static void SessionEndHandler PROTO((short));
                       /* Error notification handler       */
static void ErrorHandler PROTO((short));
                       /* Graceful exit from program       */
static void ExitProgram PROTO((short));
                       /* set terminal for punctual input  */
void setPunctual PROTO((void));
                       /* set terminal for Raw input       */
void setRaw PROTO((void));
                       /* save terminal's original setting */
void saveTerm PROTO((void));
                       /* restore term to original setting */
void restoreTerm PROTO((void));
                       /* Checks for keystroke on terminal */
static short kbhit PROTO((void));
/* Global Variables */
static short CallActive; /*Boolean for state of current call*/
static short KeyHit[QSIZE]; /* Queue for tele keys pressed  */
static short Headptr;     /* Pointer to head of KeyHit queue*/
static short Tailptr;     /* Pointer to tail of KeyHit queue*/
                        /* Info struct for term control   */
static       struct   termio tbufsave;
/*            */
/* Main program */
/*            */
void main (argc,argv)
int argc;
char *argv[];
{
 short RetCode=0; /* Return code from an ACCESS function      */
 short FileHandle; /* File handle for opened voice seg file */
 char AccountNum[USERID_SIZE]; /* User acc to get voice file*/
 char Password[PSWD_SIZE];     /* Password for account      */
```

```
char ClassNo[BUFF_SIZE];  /* Class NO of WHATKEY appl (str)*/
short  Class;             /* Class NO of WHATKEY appl (int)*/
short  SegList[SEGLIST_SIZE]; /* List of voice seg to play */
short  LogonInfo;             /* info parameter for logon */
short  MaxTime;           /* time to block waiting for a call */
short  AcqNewChannel;     /* Boolean – used by WaitingForCall */
 /* Retrieve the program argument (message queue key). */
 /* If not there exit with usage message.              */
 if (argc > 1) {
    printf("usage: whatkey \n\n");
    exit(0);
 }
 /* Save the original host terminal characteristics    */
 saveTerm();
 /* Set the host terminal characteristics for punctual */
 /* input and no echo (i.e. input characters are not   */
 /* assembled into lines before they are read, but     */
 /* made available as they are typed).                 */
 setPunctual();

 CallActive = FALSE;     /* No call is active.         */

 Headptr = Tailptr = 0; /* tel–key digit queue is empty*/
 /* Install the event handlers */
 m_OnCallProgress(CallProgressHandler);
 m_OnDigit(DigitHandler);
 m_OnSessionEnd(SessionEndHandler);
 m_OnError(ErrorHandler);
 m_TimeoutOff();           /* Disable timeout event    */
 /* Get acc number, password, and class from the user  */

 printf("\nEnter your Meridian Mail account number: ");
 getNumericStr(AccountNum,USERID_SIZE,TRUE);
 printf("\nPassword:");
 getNumericStr(Password,PSWD_SIZE,FALSE);
 printf("\nEnter ACCESS class of application (0–8999): ");
 getNumericStr(ClassNo,BUFF_SIZE,TRUE);
 sscanf(ClassNo,"%hd",&Class);

 /* Set the host terminal characteristics for Raw input*/
 setRaw();
 /* Register with the ACCESS Link Handler process */
 if (!m_Register(&RetCode)) {
    printf("\nUnable to register with the LH. Error=%d\n", RetCode);
    ExitProgram(NOTREGISTERED);
 }

 printf("\nNote! Press any key to exit the program at any time.\n");
```

```
/* Acquire a MM ACCESS session when an incoming call */
/* arrives for the WHATKEY application.              */
if (!m_AcquireOnIncomingCall(Class,&RetCode)) {
  printf("\nUnable to acquire an ACCESS session. Error=%d\n", RetCode);
  ExitProgram(REGISTERED);
}
printf("\n\nPlease call the telephone number of the");
printf(" WHATKEY application. (Class = %d) ...",Class);
fflush (stdout);
/*                                */
/* Main loop – one pass per incoming call */
/*                                */
MaxTime = MIN_WAIT_TIME;  /* Time to block for an incoming call */
AcqNewChannel = TRUE; /* Get a new channel for each incoming call */

while (TRUE) {

  /* Wait until an incoming call arrives, the user   */
  /* wishes to exit, OR an error occurs              */
  /* Note: WaitingForCall will acquire a new channel */
  /* for each incoming call – AcqNewChannel parameter*/
  /* is set to TRUE.                 */
  if (m_WaitingForCall(MaxTime, AcqNewChannel, &RetCode) != TRUE) {
    if (kbhit())
      break;
    /* Check if timeout occurred, or some other error. */
    if (RetCode == MME_OPER_TIMEOUT)
      continue;
    /* if WaitingForCall didn't time out, print error */
    if (RetCode != MMS_OKAY)
      printf("\nProblem waiting for a call.  Error=%d", RetCode);

    /* exit loop.*/
    break;
  }
  /* Log in to the desired Meridian Mail account */
  if (!m_Logon(AccountNum,Password,&LogonInfo,&RetCode)) {
    printf("\nUnable to logon to account. Error=%d\n", RetCode);
    ExitProgram(ACQUIRED);
  }
  /* Open the WHATKEY voice file for reading only */
  if (!m_OpenFile("WhatKey","r",&FileHandle,&RetCode)) {
    printf("\nUnable to open WHATKEY voice file. Error=%d\n", RetCode);
    ExitProgram(ACQUIRED);
  }

  /* Call is ringing, answer the call. */
  if (!m_AnswerCall(&RetCode)) {
    printf("Unable to answer an incoming call. Error=%d\n", RetCode);
```

```
  ExitProgram(ACQUIRED);
}
CallActive = TRUE;
printf("\nCall received.");

SegList[0] = PRESSKEY; /* Set up voice seg list for play. */
SegList[1] = LIST_END; /* Only 1 segment to be played.    */
/* Play a voice segment asking caller to press a key. */
/* Do not send an end of playback event (FALSE).      */
if (!m_PlaySegs(FileHandle,SegList,FALSE,&RetCode)) {
  printf("\nUnable to play voice segment. Error=%d\n", RetCode);
  ExitProgram(ACQUIRED);
}
/*                                */
/* ACTIVE CALL SUB–LOOP               */
/* Keep checking events, play keys hit, */
/* until caller hangs up              */
/*                                */
while (CallActive) {
  sleep(2); /* want to be woken when an event arrives */
        /* so take a litle nap                  */
        /* did we recieve a digit ?             */
  if (Headptr != Tailptr) {
    /* Yes. Assign a voice segment to play  */
    /* for the digit key hit              */
    SegList[1] = LIST_END; /* Pre–set list for 1 seg play. */
    switch (KeyHit[Headptr]) {
      case '1' : SegList[0] = ONE;  /* ONE, TWO, etc. are */
                 break;          /* constants from the */
      case '2' : SegList[0] = TWO;  /* KEYSEGS.H hdr file */
                 break;          /* which act as index */
      case '3' : SegList[0] = THREE;/* into the WHATKEY   */
                 break;          /* voice segment file */
      case '4' : SegList[0] = FOUR;
                 break;
      case '5' : SegList[0] = FIVE;
                 break;
      case '6' : SegList[0] = SIX;
                 break;
      case '7' : SegList[0] = SEVEN;
                 break;
      case '8' : SegList[0] = EIGHT;
                 break;
      case '9' : SegList[0] = NINE;
                 break;
      case '0' : SegList[0] = ZERO;
                 break;
      case '*' : SegList[0] = ASTERISK;
                 break;
```

```
            case '#' : SegList[0] = ONE;  /* Octothorpe causes   */
                       SegList[1] = TWO;  /* ONE thru ZERO to be */
                       SegList[2] = THREE;/* played              */
                       SegList[3] = FOUR;
                       SegList[4] = FIVE;
                       SegList[5] = SIX;
                       SegList[6] = SEVEN;
                       SegList[7] = EIGHT;
                       SegList[8] = NINE;
                       SegList[9] = ZERO;
                       SegList[10]= LIST_END ; break;
          } /* switch */
                            /* Flush digit from Queue. */
          Headptr = (Headptr + 1) % QSIZE;
          /* Play voice segment announcing digit key pressed */
          if (!m_PlaySegs(FileHandle,SegList,FALSE,&RetCode)) {
          /* Filter the case of the user hanging up */
          /*  during segment playing               */
            if (RetCode == MME_NO_ACTV_CHNL) {
              printf("\nUser has hung up");
              printf(" before all the digits were echoed.");
            }
            else {
              printf(
               "\nUnable to play voice segment list. Error=%d\n", RetCode);
              ExitProgram(ACQUIRED);
            } /* end if(RetCode == ) */
          }  /* if !m_PlaySegs */
        } /* if (Headptr != Tailptr) */
      } /* while (CallActive) */
      /* allow time for call to terminate */
      while (CallActive)
        sleep(1);
      printf("\n\nPlease call the telephone number of the");
      printf(" WHATKEY application,\n or hit any key to exit.");
    } /* while TRUE */
    /* Release ACCESS session (logs off & closes files) */
    if (!m_Release(&RetCode) && RetCode != MME_NOT_ACQUIRED) {
      printf("\nUnable to release ACCESS Session. Error=%d\n", RetCode);
    } /* !m_Release */
    /* Deregister from the Meridian Mail ACCESS and exit  */
    ExitProgram(REGISTERED);
} /* End of main */
/*                                  */
/* "Call Progress" Event Handler:         */
/* Automatically called when the Current   */
/* call has changed states.                */
static void CallProgressHandler (StateChange,StateInfo)
short StateChange;     /* New State of call      */
```

```
short StateInfo;       /* Info about StateChange */
{
  /* Only expected state is CP_DISCONNECT.    */
  /* Set CallActive to let mainline know      */
  /* that call has disconnected.              */
  if (StateChange == CP_DISCONNECT) {
        CallActive = FALSE;
        Headptr = Tailptr; /* reset digit queue  */
  }                        /* to empty          */
} /* End of CallProgressHandler */
/*                                */
/* "Digit Received" Event Handler:     */
/* Automatically called when a key is  */
/* hit on the telephone keypad.        */
/*                                */
static void DigitHandler (RecDigit)
short RecDigit; /* Digit key hit by user */
{
  short queuePos;  /* Temp variable indicating next queue pos */
  /* Store the received digit in the circular queue */
  /* Calculate next tail pointer value              */
  queuePos = (Tailptr+1) % QSIZE;
  if (queuePos==Headptr)
    /* Queue is full, so print message on screen */
    printf("\nDigit queue already full. One digit lost.");
  else {
    /* There's room in queue, so put digit */
    /* in queue & update tail pointer      */
    KeyHit[Tailptr] = RecDigit;
    Tailptr = queuePos;
  }
} /* End of DigitHandler */
/*                                          */
/* "Error notification" event handler                  */
/* Automatically called when Meridian Mail Error occurs */
/*                                          */
static void ErrorHandler (Type)
short Type;       /* type of error */
{
  printf("\n\nMeridian Mail error. Type= %d\n",Type);
} /* end of ErrorHandler */
/*                                */
/* "Session Disconnect" event handler:     */
/* Automatically called when Meridian Mail */
/* is releasing the session                */
/*                                */
static void SessionEndHandler (Reason)
short Reason;     /* Reason for disconnect. */
{
```

```
 printf(
  "\n\nMeridian Mail has disconnected the session. Reason=%d",
      Reason);
 ExitProgram(NOTREGISTERED);
} /* End of SessionEndHandler */

/*                                          */
/* This function will check the terminal for any keystroke */
/* It will return: TRUE value if there was a keystroke and */
/*               FALSE is there was no keystroke activity*/
/*                                          */
static short kbhit ()
{
                  /* check for keyboard input */
 if (getchar() == EOF)
    return(FALSE);/* if End–Of–File there was no input */
 else {
    return(TRUE);  /* We have keyboard input */
 }
}
/*                                      */
/* This routine will gracefully exit the program */
/*                                      */
static void ExitProgram (StatusLevel)
short StatusLevel; /* Current program status level */
{
  short ErrCode;  /* Error code returned by MM ACCESS function */
  /* Exit the program gracefully – no need to check error codes */
  /* since are only interested in terminating the program.      */
  switch (StatusLevel) {
    case ACQUIRED     : /* Have acquired a MM ACCESS session */
                m_Release(&ErrCode);
    case REGISTERED   : /* Have registered with MM ACCESS */
                m_Deregister(&ErrCode);
    case NOTREGISTERED: /* No MM ACCESS connections yet */
                restoreTerm();
                exit(0);
  } /* switch */
} /* End of ExitProgram */

/*                                          */
/* This Routine will retrieve a string of numeric characters */
/* (character by character up to "size" chars) and store them*/
/* in memory locations beginning at "startLocn". Echoing is  */
/* performed if requested.                  */
/*                                          */
static void getNumericStr (strLocn,size,echo)
char *strLocn;    /* Mem location of destination string */
short  size;      /* Maxi num of chars to be collected  */
```

```
short  echo;      /* Boolean – true if chars are echoed */
{
  short stringEmpty = TRUE;
  char *charPtr   = strLocn;
                  /* Loop until entry is terminated */
                  /* by newline '\n' and string is  */
                  /* not empty                      */
  while (stringEmpty) {
    while ((*charPtr = getchar()) != '\n') {
      if (echo)
        putchar(*charPtr);
      if (!isdigit(*charPtr)) {  /* Char received not a numeric. */
        printf("\nNumeric characters only.");
        printf("\n\nPlease reenter the number: ");
        stringEmpty = TRUE; /* Restart the collection. Reset to */
        charPtr = strLocn;  /* the 1st location of destn string */
        break;
      }
      else {              /* Got a valid numeric char */
        stringEmpty = FALSE;
        charPtr++;        /* Dest addr of next char.  */
      }
      if ((charPtr – strLocn) == size – 1) /* Leave a byte for */
        break;                      /* end of string    */
    } /* end while *charPtr = getchar */
  } /* end while stringEmpty */
  *charPtr = '\0';            /* Set end of string char. */
} /* end getNumericStr */
/*                                          */
/* This routine will set the host terminal characteristics */
/* for punctual input with no echo. Input characters are   */
/* not assembled into lines before they are read, but      */
/* made available as they are typed.                       */
/*                                          */
void setPunctual  ()
{
  struct termio tbuf;       /* terminal information buffer       */
                  /* get the original terminal setting */
  tbuf = tbufsave;
                  /* set host terminal for punctual */
                  /* input, no echo, min 1 char buff*/
  tbuf.c_lflag &= ~(ICANON | ECHO);
  tbuf.c_cc[VMIN]  = 1;
  tbuf.c_cc[VTIME] = 0;
  if (ioctl(0,TCSETAF,&tbuf) == –1)
    printf("ioctl SET failed!\n");
} /* end setPunctual */
/*                                          */
/* This routine will set the host terminal characteristics */
```

```
/* for raw. Input characters are not echoed, and are       */
/* not assembled into lines before they are read, but      */
/* made available as they are typed. If there is nothing   */
/* to read the function reurns false.                      */
/*                                                         */
void setRaw  ()
{
  struct termio tbuf;       /* terminal information buffer       */
                /* get the original terminal setting */
  tbuf = tbufsave;
                /* set host terminal for punctual    */
                /* input, no echo, NON–BLOCKING      */
  tbuf.c_lflag &= ~(ICANON | ECHO);
  tbuf.c_cc[VMIN]  = 0;
  tbuf.c_cc[VTIME] = 0;
  if (ioctl(0,TCSETAF,&tbuf) == –1)
    printf("ioctl SET failed!\n");
} /* end setRaw */
/*                                                         */
/* This routine will save the host terminal characteristics */
/* original settings in a global buffer                    */
/*                                                         */
void saveTerm ()
{
                /* get host terminal characteristics */
  if (ioctl(0,TCGETA,&tbufsave) == –1)
    printf("ioctl GET (original) failed!\n");
} /* end saveTerm */
/*                                                         */
/* This routine will restore the host terminal characteristics */
/* to their original settings.                             */
/*                                                         */
void restoreTerm ()
{
                /* set host terminal characteristics */
                /* to the original settings.         */
  if (ioctl(0,TCSETAF,&tbufsave) == –1)
    printf("ioctl SET (original) failed!\n");
} /* end restoreTerm */
```

WHATLINE

This program services incoming calls by verbally echoing the telephone number the caller dialed to reach the WHATLINE application. The program prompts for an account number and password of a Meridian Mail account containing the voice segment file required to service a call. It also prompts for the class of this application. The program will continue to answer calls until a key is pressed at the terminal keyboard.

```
/*****************************************************************/
/* WHATLINE                                                      */
/*                                                               */
/* This program services incoming calls by verbally echoing      */
/* the telephone number the caller dialed to reach the           */
/* WHATLINE application. The program prompts for an account#     */
/* and password of a Meridian Mail account containing the voice */
/* segment file required to service a call. It also prompts      */
/* for the class of this application. The program will continue */
/* to answer calls until a key on the terminal keyboard is       */
/* pressed.                                                      */
/*                                                               */
/* Synopsis: whatline                                            */
/*                                                               */
/* Copyright (C) Northern Telecom Limited, 1990 – 1993           */
/*                                                               */
/*****************************************************************/

static char *version = ”@(#)whatline.c 1.5    4/3/91 (NT)”;

#include <stdio.h>       /* UNIX file for standard I/O routines */
#include <termio.h>      /* UNIX file for term I/O control      */
#include <ctype.h>       /* Char class & conversion routines    */
#include <machine.h>     /* header files for conditional compile*/
#include <m_acc.h>       /* Meridian Mail ACCESS general functs */
#include <m_rm.h>        /* Meridian Mail ACCESS resource mgmt  */
#include <m_local.h>     /* Meridian Mail ACCESS local access   */
#include <m_file.h>      /* Meridian Mail ACCESS file access    */
#include <m_seg.h>       /* Meridian Mail ACCESS voice segment  */
#include <m_voice.h>     /* Meridian Mail ACCESS voice operation*/
#include <m_event.h>     /* Meridian Mail ACCESS event handler  */
#include ”keysegs.h”     /* Indices for segs in voice seg file  */
#define LIST_END      0  /* Marks the end of a list of voice seg*/
                         /* indicies.                           */
#define BUFF_SIZE     6  /* # of chars required to house an int:*/
                         /* 5 digits plus an end of string char.*/
/* Valid program status levels, used by ”ExitProgram” routine   */
```

```
#define NOTREGISTERED 0  /* Not yet registered with MM ACCESS   */
#define REGISTERED    1  /* Have registered with MM ACCESS      */
#define ACQUIRED      2  /* Have acquired a MM ACCESS session   */
/* Prototypes */
                /* Main program                        */
void main PROTO((int, char ** ));
                /* Get numeric string from keyboard        */
static void getNumericStr PROTO((char *,short, short));
                /* Incoming call event handler          */
static void IncomingCallHandler PROTO((char *,char *));
                /* Call Progress event handler          */
static void CallProgressHandler PROTO((short,short));
                /* Session Disconnect event handler     */
static void SessionEndHandler PROTO((short));
                /* Graceful exit from program           */
static void ExitProgram PROTO((short));
                /* set terminal for punctual input.     */
void setPunctual PROTO((void));
                /* set terminal for Raw input.          */
void setRaw PROTO((void));
                /* save terminal's original settings    */
void saveTerm PROTO((void));
                /* restore terminal to original setting */
void restoreTerm PROTO((void));
                /* Checks for keystroke on terminal     */
static short kbhit PROTO((void));
/* Global Variables */
static short CallActive;  /* Boolean for state of current call  */
static char PhoneNum[DN_SIZE]; /* Num dialled for WHATLINE appl.*/
                /* Info struct for terminal control     */
static      struct   termio tbufsave;
/* Main program */
void main(argc,argv)
int argc;
char *argv[];
{
  short RetCode;    /* Return code from an ACCESS function     */
  short FileHandle; /* File handle for opened voice seg file   */
  char AccountNum[USERID_SIZE]; /* User acct to get voice file */
  char Password[PSWD_SIZE];     /* Password for account        */
  char ClassNo[BUFF_SIZE];      /* Class of WHATLINE appl (str)*/
  short Class;                  /* Class of WHATLINE appl (int)*/
  short SegList[SEGLIST_SIZE];  /* List of voice segs to play  */
  short index;                  /* Voice segment list index    */
  short LogonInfo;              /* Logon info parameter        */
  /* Retrieve the program argument (message queue key).        */
  /* If not there exit with usage message.                     */
  if (argc > 1) {
    printf("usage: whatline\n\n");
```

```
   exit(0);
 }
 /* Save the original host terminal characteristics    */
 saveTerm();
 /* Set the host terminal characteristics for punctual */
 /* input and no echo (i.e. input characters are not   */
 /* assembled into lines before they are read, but     */
 /* made available as they are typed).                 */
 setPunctual();

 /* Install the event handlers */
 m_OnIncomingCall(IncomingCallHandler);
 m_OnCallProgress(CallProgressHandler);
 m_OnSessionEnd(SessionEndHandler);
 m_TimeoutOff();        /* Disable timeout event       */
 CallActive = FALSE;    /* No call is active           */
/* Get account num, password, and class from the user  */

printf("\nEnter your Meridian Mail account number: ");
getNumericStr(AccountNum,USERID_SIZE,TRUE);
printf("\nPassword:");
getNumericStr(Password,PSWD_SIZE,FALSE);
printf("\nEnter ACCESS class of application (0–8999): ");
getNumericStr(ClassNo,BUFF_SIZE,TRUE);
sscanf(ClassNo,"%hd",&Class);

 /* Set the host terminal characteristics for Raw input*/
 setRaw();
/* Register with the ACCESS Link Handler process */
if (!m_Register(&RetCode)) {
   printf("\nUnable to register with the LH. Error=%d\n", RetCode);
   ExitProgram(NOTREGISTERED);
}

printf("\nNote! Press any key to exit the program at any time.\n");
 /* Acquire a MM ACCESS session when an incoming call */
 /* arrives for the WHATLINE application.             */
 if (!m_Acquire(Class,&RetCode)) {
   printf("\nUnable to acquire an ACCESS session. Error=%d\n", RetCode);
   ExitProgram(REGISTERED);
 }
 /* Log in to the desired Meridian Mail account */
 if (!m_Logon(AccountNum,Password,&LogonInfo,&RetCode)) {
    printf("\nUnable to logon to account. Error=%d\n",RetCode);
    ExitProgram(ACQUIRED);
 }

 /* Open the WHATKEY voice file for reading only */
```

```
if (!m_OpenFile("WhatKey","r",&FileHandle,&RetCode)) {
  printf("\nUnable to open WHATKEY voice file. Error=%d\n", RetCode);
  ExitProgram(ACQUIRED);
}

printf("\n\nPlease call the telephone number of the");
printf(" WHATLINE application. (Class = %d) ...",Class);
fflush (stdout);

/*                                */
/* Main loop – one pass per incoming call */
/*                                */
while (TRUE) {
  /* Wait until an incoming call arrives, the user */
  /* wishes to exit, OR an error occurs          */
  /* Note: WaitingForCall will NOT release the    */
  /* channel between calls – the AcqNewChannel    */
  /* parameter is FALSE.                          */
  if (m_WaitingForCall(MIN_WAIT_TIME,FALSE,&RetCode)!=TRUE) {
    if (kbhit())
      break;
    /* Check if WaitingForCall timed out */
    if (RetCode == MME_OPER_TIMEOUT)
      continue;
    /* Check if error occurred */
    if (RetCode != MMS_OKAY)
      printf("\nProblem waiting for a call. Error=%d", RetCode);

    /* exit loop. */
    break;
  }

  /* Call is ringing, answer the call. */
  if (!m_AnswerCall(&RetCode)) {
    printf("Unable to answer an incoming call. Errof=%d\n", RetCode);
    ExitProgram(ACQUIRED);
  }

  CallActive = TRUE;
  printf("\nCall received.");

  /* PhoneNum is set in the IncomingCall handler.  */
  printf("\n\nThe number dialed is: %s.",PhoneNum);
  /* Set up a voice segment list to be played to caller. */
  /* List includes segments corresponding to the tel num */
  /*  dialled by user                          */
        /* First segment = "The number dialled is .."*/
  SegList[0] = NUMDIALED;
  index = 1;
```

```
    while (PhoneNum[index–1] != '\0') {
      switch (PhoneNum[index–1]) {
        case '1': SegList[index] = ONE;  /* ONE, TWO, etc. are */
                  break;                 /* constants from the */
        case '2': SegList[index] = TWO;  /* KEYSEGS.H hdr file */
                  break;                 /* which are indices  */
        case '3': SegList[index] = THREE;/* into the WHATKEY   */
                  break;                 /* voice segment file */
        case '4': SegList[index] = FOUR;
                  break;
        case '5': SegList[index] = FIVE;
                  break;
        case '6': SegList[index] = SIX;
                  break;
        case '7': SegList[index] = SEVEN;
                  break;
        case '8': SegList[index] = EIGHT;
                  break;
        case '9': SegList[index] = NINE;
                  break;
        case '0': SegList[index] = ZERO;
                  break;
      } /* end switch */
      index++;
    } /* end while */
    SegList[index] = HANGUP;    /* Last seg="Please hang–up phone"*/
    SegList[index+1] = LIST_END;/* Mark end of voice segment list */
    /* Play list of voice segments announcing dialled number. */
    /* Do not return an end of playback event (FALSE).        */
    if (!m_PlaySegs(FileHandle,SegList,FALSE,&RetCode)) {
      printf("\nUnable to play voice segment list. Error=%d\n", RetCode);
      ExitProgram(ACQUIRED);
    }
    while (CallActive)    /* Wait for caller to hang–up */
      sleep(2);
    printf("\n\nPlease call the telephone number of the");
    printf(" WHATLINE application,\n or hit any key to exit.");
  } /* while (TRUE) */
  /* Release ACCESS session (logs off & closes files) */
  if (!m_Release(&RetCode)) {
    printf("\nUnable to release ACCESS Session. Error=%d", RetCode);
  } /* !m_Release */
  /* Deregister from the Meridian Mail ACCESS  and exit */
  printf ("\n");
  ExitProgram(REGISTERED);
} /* End of main */
/*                          */
/* "Incoming Call" Event Handler  */
```

```
/*  A new call has just arrived.  */
/*                              */
static void IncomingCallHandler(FromDN,ToDN)
char *FromDN; /* Calling telephone number */
char *ToDN;   /* Called telephone number  */
{
  /* Save the number dialled for mainline to echo */
  strcpy(PhoneNum,ToDN);

  /* The application will be notified  that this     */
  /* event has occurred by using the API command     */
  /* ”m_WaitingForCall” in the mainline. Therefore   */
  /* do not need to set a flag here indicating that  */
  /* this event has occurred                         */
} /* End of IncomingCallHandler */
/*                                    */
/* ”Call Progress” Event Handler:             */
/* Automatically called when the Current call */
/* has changed states.                        */
/*                                    */
static void CallProgressHandler(StateChange,StateInfo)
short StateChange;     /* New State of call      */
short StateInfo;       /* Info about StateChange */
{
  /* Only expected state is CP_DISCONNECT.    */
  /* Set CallActive to let mainline know      */
  /* that call has disconnected.              */
  if (StateChange == CP_DISCONNECT) {
        CallActive = FALSE;
  }
} /* End of CallProgressHandler */

/*                                    */
/* ”Session Disconnect” event handler:     */
/* Automatically called when Meridian Mail */
/* is releasing the session                */
/*                                    */
static void SessionEndHandler(Reason)
short Reason;      /* Reason for disconnect. */
{
 printf(”\n\nMM has disconnected the session. Reason=%d\n”, Reason);
 ExitProgram(NOTREGISTERED);
} /* End of SessionEndHandler */

/*                                          */
/* This function will check the terminal for any keystroke */
/* It will return: TRUE value if there was a keystroke and */
/*                 FALSE is there was no keystroke activity*/
/*                                          */
```

```
static short kbhit()
{
                    /* check for keyboard input */
  if (getchar() == EOF)
      return(FALSE);/* if End–Of–File there was no input */
  else {
      return(TRUE);  /* We have keyboard input */
  }
}
/*                                        */
/* This routine will gracefully exit the program */
/*                                        */
static void ExitProgram(StatusLevel)
short StatusLevel; /* Current program status level */
{
   short ErrCode;  /* Error code returned by MM ACCESS */
   /* Exit the program gracefully – no need to check error codes */
   /* since are only interested in terminating the program.      */
   switch (StatusLevel) {
      case ACQUIRED     : /* Have acquired a MM ACCESS session */
                    m_Release(&ErrCode);
      case REGISTERED   : /* Have registered with MM ACCESS    */
                    m_Deregister(&ErrCode);
      case NOTREGISTERED: /* No MMl ACCESS connections up yet  */
                    restoreTerm();
                    exit(0);
   } /* switch */
} /* End of ExitProgram */

/*                                              */
/* This Routine will retrieve a string of numeric characters */
/* (character by character up to ”size” chars) and store them*/
/* in memory locations beginning at ”startLocn”. Echoing is  */
/* performed if requested.                      */
/*                                              */
static void getNumericStr(strLocn,size,echo)
char *strLocn;    /* Mem location of destination string      */
short  size;      /* Max num of chars to be collected.       */
short  echo;      /* Boolean – true if chars are echoed      */
{
  short stringEmpty = TRUE;
  char *charPtr   = strLocn;
                    /* Loop until entry is terminated */
                    /* by newline ’\n’ and string is  */
                    /* not empty                      */
  while (stringEmpty) {
     while ((*charPtr = getchar()) != ’\n’) {
        if (echo)
           putchar(*charPtr);
```

```
      if (!isdigit(*charPtr)) { /* Char received not a num. */
        printf("\nNumeric characters only.");
        printf("\n\nPlease reenter the number: ");
        stringEmpty = TRUE; /* Restart the collection. Reset to */
        charPtr = strLocn;  /* the 1st location of destn string */
        break;
      }
      else {                  /* Got a valid numeric char */
        stringEmpty = FALSE;
        charPtr++;            /* Dest addr of next char.  */
      }
      if ((charPtr – strLocn) == size – 1) /* Leave a byte for */
        break;                        /* end of string    */
    } /* end while *charPtr = getchar */
  } /* end while stringEmpty */
  *charPtr = '\0';            /* Set end of string char. */
} /* end getNumericStr */
/*                                          */
/* This routine will set the host terminal characteristics */
/* for punctual input with no echo. Input characters are   */
/* not assembled into lines before they are read, but      */
/* made available as they are typed.                       */
/*                                          */
void setPunctual ()
{
  struct termio tbuf;       /* terminal information buffer       */
                  /* get the original terminal setting */
  tbuf = tbufsave;
                  /* set host terminal for punctual    */
                  /* input, no echo, min 1 char buff   */
  tbuf.c_lflag &= ~(ICANON | ECHO);
  tbuf.c_cc[VMIN]  = 1;
  tbuf.c_cc[VTIME] = 0;
  if (ioctl(0,TCSETAF,&tbuf) == –1)
    printf("ioctl SET failed!\n");
} /* end setPunctual */
/*                                          */
/* This routine will set the host terminal characteristics */
/* for raw. Input characters are not echoed, and are       */
/* not assembled into lines before they are read, but      */
/* made available as they are typed. If there is nothing   */
/* to read the function reurns false.                      */
/*                                          */
void setRaw ()
{
  struct termio tbuf;       /* terminal information buffer       */
                  /* get the original terminal setting */
  tbuf = tbufsave;
                  /* set host terminal for punctual    */
```

```
                    /* input, no echo, NON–BLOCKING      */
  tbuf.c_lflag &= ~(ICANON | ECHO);
  tbuf.c_cc[VMIN]  = 0;
  tbuf.c_cc[VTIME] = 0;
  if (ioctl(0,TCSETAF,&tbuf) == –1)
    printf("ioctl SET failed!\n");
} /* end setRaw */
/*                                          */
/* This routine will save the host terminal characteristics */
/* original settings in a global buffer                     */
/*                                          */
void saveTerm()
{
                    /* get host terminal characteristics */
  if (ioctl(0,TCGETA,&tbufsave) == –1)
    printf("ioctl GET (original) failed!\n");
} /* end saveTerm */
/*                                           */
/* This routine will restore the host terminal characteristics */
/* to their original settings.                              */
/*                                          */
void restoreTerm()
{
            /* set host terminal characteristics */
            /* to the original settings.        */
  if (ioctl(0,TCSETAF,&tbufsave) == –1)
    printf("ioctl SET (original) failed!\n");
} /* end restoreTerm */
```

LOTTO

lotto.h

```
/*************************************************************************/
/* MM ACCESS VPE, Ver. SP5.08                                            */
/*                                                                       */
/* Copyright (C) Northern Telecom Ltd, 1989 – 1993                       */
/*                                                                       */ /* Voice Segment ID values for Voice Segment File:
     */ /*                    NAME : lotto                    */ /*                    SUBJECT : lotto – voice
segments      */ /*              CREATION DATE : 91/02/11 13:59:37          */ /*
MODIFICATION DATE : 91/02/12 15:45:51          */
/*************************************************************************/

#define lotto_VERSION 910212154551

#define one            1   /* The number ”1” */
#define two            2   /* The number ”2” */
#define three          3   /* The number ”3” */
#define four           4   /* The number ”4” */
#define five           5   /* The number ”5” */
#define six            6   /* The number ”6” */
#define seven          7   /* The number ”7” */
#define eight          8   /* The number ”8” */
#define nine           9   /* The number ”9” */
#define ten           10   /* The number ”10” */
#define eleven        11   /* The number ”11” */
#define twelve        12   /* The number ”12” */
#define thirteen      13   /* The number ”13” */
#define fourteen      14   /* The Number ”14” */
#define fifteen       15   /* The number ”15” */
#define sixteen       16   /* The number ”16” */
#define seventeen     17   /* The number ”17” */
#define eighteen      18   /* The number ”18” */
#define nineteen      19   /* The number ”19” */
#define twenty        20   /* The number ”20” */
#define twenty_one    21   /* The number ”21” */
#define twenty_two    22   /* The number ”22” */
#define twenty_three  23   /* The number ”23” */
#define twenty_four   24   /* The number ”24” */
#define twenty_five   25   /* The number ”25” */
#define twenty_six    26   /* The number ”26” */
#define twenty_seven  27   /* The number ”27” */
#define twenty_eight  28   /* The number ”28” */
#define twenty_nine   29   /* The number ”29” */
#define thirty        30   /* The number ”30” */
#define thirty_one    31   /* The number ”31” */
#define thirty_two    32   /* The number ”32” */
#define thirty_three  33   /* The number ”33” */
#define thirty_four   34   /* The number ”34” */
```

```
#define thirty_five      35  /* The number ”35” */
#define thirty_six       36  /* The number ”36” */
#define thirty_seven     37  /* The number ”37” */
#define thirty_eight     38  /* The number ”38” */
#define thirty_nine      39  /* The number ”39” */
#define forty            40  /* The number ”40” */
#define forty_one        41  /* The number ”41” */
#define forty_two        42  /* The number ”42” */
#define forty_three      43  /* The number ”43” */
#define forty_four       44  /* The number ”44” */
#define forty_five       45  /* The number ”45” */
#define forty_six        46  /* The number ”46” */
#define forty_seven      47  /* The number ”47” */
#define forty_eight      48  /* The number ”48” */
#define forty_nine       49  /* The number ”49” */
#define fifty            50  /* The number ”50” */
#define fifty_one        51  /* The number ”51” */
#define fifty_two        52  /* The number ”52” */
#define fifty_three      53  /* The number ”53” */
#define fifty_four       54  /* The number ”54” */
#define fifty_five       55  /* The number ”55” */
#define fifty_six        56  /* The number ”56” */
#define fifty_seven      57  /* The number ”57” */
#define fifty_eight      58  /* The number ”58” */
#define fifty_nine       59  /* The number ”59” */
#define sixty            60  /* The number ”60” */
#define sixty_one        61  /* The number ”61” */
#define sixty_two        62  /* The number ”62” */
#define sixty_three      63  /* The number ”63” */
#define sixty_four       64  /* The number ”64” */
#define sixty_five       65  /* The number ”65” */
#define sixty_six        66  /* The number ”66” */
#define sixty_seven      67  /* The number ”67” */
#define sixty_eight      68  /* The number ”68” */
#define sixty_nine       69  /* The number ”69” */
#define seventy          70  /* The number ”70” */
#define seventy_one      71  /* The number ”71” */
#define seventy_two      72  /* The number ”72” */
#define seventy_three    73  /* The number ”73” */
#define seventy_four     74  /* The number ”74” */
#define seventy_five     75  /* The number ”75” */
#define seventy_six      76  /* The number ”76” */
#define seventy_seven    77  /* The number ”77” */
#define seventy_eight    78  /* The number ”78” */
#define seventy_nine     79  /* The number ”79” */
#define eighty           80  /* The number ”80” */
#define WELCOME          81  /* The ”welcome” prompt for lotto. */
#define ENTER_SEED       82  /* Enter seed prompt for lotto. */
#define HELP_SEED        83  /* Help prompt for enter seed in Lotto. */
```

```
#define ERROR_ATT       84  /* Error prompt for disconnecting lotto */
#define GOOD_BYE        85  /* Good–Bye prompt for lotto. */
#define THANK_YOU       86  /* Thank–you prompt for lotto. */
#define ENTER_MENU      87  /* Menu selection prompt for lotto. */
#define HELP_MENU       88  /* Help for enter menu prompt in lotto. */
#define HELP_MENU2      89  /* Second help for enter menu prompt in
                           lotto. */
#define YOURNUMBERS     90  /* Play numbers prompt for lotto. */
#define LOTTO_VOICE_FILE ”lotto”  /* Lotto voice file name */
```

lotto.c

```
/*********************************************************************/
/*                                                                   */
/*              tt      M e r i d i a n  1  A C C E S S    */
/*             ttt                                                   */
/*  nnn  nnn    ttttttt          Northern Telecom                    */
/*  nnn nnnnn   tttttttt       Customer Applications Center         */
/*  nnn   nnn     ttt              Toronto                           */
/*  nnn   nnn     ttt                                                */
/*  nnn   nnnnn   ttttt                                             */
/*  nnn    nnn    ttt                                               */
/*                                                                   */
/*  Copyright (C) Northern Telecom Limited, 1991 – 1993              */
/*___________________________________________________________________*/
/*                                                                   */
/*                                                                   */
/* 1.0 Description:                                                  */
/*                                                                   */
/*     ”lotto” is a sample IVR application for Meridian 1 ACCESS.    */
/*                                                                   */
/* 2.0 Purpose:                                                      */
/*                                                                   */
/*     The intent of ”lotto” is to demonstrate how a typical         */
/* Interactive Voice Response (IVR) application would be             */
/* implemented using the commands included in the standard           */
/* ACCESS API Library. In view of the fact that a typical IVR        */
/* application is a process which must handle many types of          */
/* asynchronous events we have adopted the ”State Machine” model     */
/* in implementing ”lotto”. The ”State Machine” is the ideal         */
/* model in implementing an event driven process. The typical        */
/* IVR system must  protect itself both from the caller and from     */
/* all components that it interfaces with. The IVR system must       */
/* anticipate any possible scenario where the caller is left         */
/* ”hanging”, or the caller hangs the system. Thus, we exhibit       */
/* the correct form of error detection and recovery .                */
/*                                                                   */
/* In addition, ”lotto” will also demonstrate the following          */
/* activities which we would consider as being typical in an         */
/* IVR systems:                                                      */
/*                                                                   */
/*     – Collecting single digits for voice menus                    */
/*     – Collecting a series of digits terminated                    */
/*       by a designated key                                         */
/*     – Playing individual voice segment files                      */
/*     – Playing a list of concatenated  voice segment files         */
/*                                                                   */
/* 3.0 Application Description:                                      */
/*                                                                   */
/* In a nutshell, the ”lotto” program is a simple IVR application    */
```

```
/* that accepts incoming calls, generates a series of random       */
/* numbers in a designated range, and plays these numbers back     */
/* to the caller. In examining the process flow of the             */
/* application. We begin by  waiting for the next call to arrive   */
/* and interrupt us. Once we receive a call we play a ”welcome”    */
/* prompt and ask the caller  to enter a ”seed” (a number between  */
/* 0 and 99,999) using the key pad of a touch tone phone. The      */
/* ”seed”  will be used in  generating the sequence of random      */
/* numbers. The caller is then prompted to make a selection from   */
/* the following menu:                                             */
/*                                                                 */
/*   – Press ”1” to generate 6 random numbers between 1 and 49     */
/*   – Press ”2” to generate 6 numbers between 1 and 80            */
/*   – Press ”3” to repeat the last numbers generated              */
/*   – Press ”4” to quit                                           */
/*                                                                 */
/* The caller remains in the menu until ”quit” is selected or      */
/* until a call disconnect is detected (i.e. caller hung up).      */
/* When the call is eventually disconnected the entire process     */
/* is repeated.                                                    */
/*                                                                 */
/*                                                                 */
/*_________________________________________________________________*/
/*                                                                 */
/* Release:                                                        */
/*      1.0 – Initial release                  JAN/91    */
/*                                                                 */
/*******************************************************************/
#include <stdio.h>            /* UNIX standard I/O routines   */
#include <ctype.h>            /* Char class & conv routines   */
#include <signal.h>           /* UNIX signal handling routines*/
#include <machine.h>          /* MM1 ACCESS target machine    */
#include <m_acc.h>            /* MM1 ACCESS general functions */
#include <m_rm.h>             /* MM1 ACCESS resource mgmt     */
#include <m_local.h>          /* MM1 ACCESS local access      */
#include <m_file.h>           /* MM1 ACCESS file access       */
#include <m_msg.h>            /* MM1 ACCESS messaging         */
#include <m_seg.h>            /* MM1 ACCESS voice segment     */
#include <m_voice.h>          /* MM1 ACCESS voice operations  */
#include <m_event.h>          /* MM1 ACCESS event handler     */
#include <m_hilev.h>          /* MM1 ACCESS High level API’s  */
#include ”lotto.h”            /* local definitions for ”lotto”*/
/*
 *
 *
 * T Y P E   D E F S  &  M A C R O S
 *
 */
```

```
#define DEBUG          FALSE    /* Current state of debug mode  */
#define MAXWAIT        120      /* Maximum wait time for calls  */
typedef enum {                  /* Enumerate all possible Comm. */
                        /* status with LH and MM        */
            NOTREGISTERED,
            REGISTERED,
            ACQUIRED
          } CommStatus;
typedef enum {                  /* Enumerate all possible states*/
            INIT,
            WAIT,
            ANSWER,
            ENTSEED,
            ENTMENU,
            PLAY,
            EXIT
          } ProcStates;
/*
 * L O C A L  F U N C T I O N S
 *
 */
ProcStates DoInit ()    ;       /* State func to handle INIT    */
ProcStates DoWait ()    ;       /* State func to Handle WAIT    */
ProcStates DoAnswer ()  ;       /* State func to Handle ANSWER  */
ProcStates DoEnter ()   ;       /* State func to Handle ENTSEED */
ProcStates DoMenu ()    ;       /* State func to Handle ENTMENU */
ProcStates DoPlay ()    ;       /* State func to Handle PLAY    */
ProcStates DoExit ()    ;       /* State func to Handle EXIT    */
void IncomingCallHandler ();    /* Incoming Call event Handler  */
void CallProgHandler ();        /* Call Progress event Handler  */
void SessionEndHandler ();      /* Session End event Handler    */
void MMErrorHandler ()  ;       /* MM Error event Handler       */
void TermSignalHandler ()  ;    /* MM Error event Handler       */
void Error ()           ;       /* Generic Error handler        */
/*
 * G L O B A L  V A R I A B L E S
 *
 */
static CommStatus Status;       /* Curr Comm status of process   */
static ProcStates State;        /* Current state of the process  */
static ProcStates NextState;    /* Next state of the process     */
static short FileHandle;        /* Open voice file handle no.    */
static short RetCode;           /* Return code from ACCESS funct */
static short CallisActive;      /* Boolean: state of curr call   */
static short Terminated;        /* Boolean: process termination  */
static short ShutDown  ;        /* Boolean: graceful shutdown    */
static char  Seed[10];          /* Seed used to gen random nums  */
static char  MenuSel[10];       /* Menu selection                */
static char  LastKey;           /* Last key depressed on keypad  */
```

```
static  char User[USERID_SIZE]; /* Application user account     */
static  char Passwd[PSWD_SIZE]; /* Password of user account     */
static  short Class;            /* Application Class Number     */
static  short RandomNums[8] = { /* Array of random numbers gen'd*/
              YOURNUMBERS,
              0,0,0,0,0,0,0
            };
/*
 *
 *
 * M A I N   P R O G R A M
 *
 */
void main (argc,argv)
int    argc;                /* lotto program arg counter    */
char *argv[];               /* lotto program args pointers  */
{
   /*
    * Check lotto usage. if incorrect
    * exit with proper usage message.
    */
   if (argc != 4) {
      printf("usage: lotto (MM account) (passwd) (class) \n\n");
      exit(0);
   }
   /*
    * Retrieve our MM acount, Password,
    * and Applicatione Class No.
    *
    */
   strncpy(User,argv[1],USERID_SIZE);
   strncpy(Passwd,argv[2],PSWD_SIZE);
   Class = atoi(argv[3]);
   /*
    * Install UNIX Signal Handlers that
    * this application will trap on and
    * terminate gracefully.
    */
   signal (SIGINT,TermSignalHandler);
   signal (SIGQUIT,TermSignalHandler);
   signal (SIGTERM,TermSignalHandler);
   signal (SIGHUP,TermSignalHandler);
   /*
    * The following loop is the state table switcher
    * for this process. It will loop until it is
    * terminated by an external signal or it encounters
    * an unrecoverable error.
    */
   State = INIT;               /* We begin in the Init State   */
```

```
  Terminated = FALSE;          /* Process not terminated yet   */
  ShutDown   = FALSE;
  while (!Terminated) {
    if (DEBUG) {               /* if we are in DEBUG mode      */
                               /* Trace the progress of the    */
                               /* state machine.               */
      Error(">> Current State of lotto :%d – %s",State,
           CallisActive ? "We have a Call" : "No Call");
    }
    switch(State) {
    case INIT:                 /* Intialization State          */
      NextState = DoInit();
      break;
    case WAIT:                 /* Waiting for the next call    */
      NextState = DoWait();
      break;
    case ANSWER:               /* Answer the call              */
      NextState = DoAnswer();
      break;
    case ENTSEED:              /* Ask caller to Enter "Seed"   */
      NextState = DoEnter();
      break;
    case ENTMENU:              /* Ask caller to make Selection */
      NextState = DoMenu();
      break;
    case PLAY:                 /* Play Selection to caller     */
      NextState = DoPlay();
      break;
    case EXIT:                 /* Graceful shutdown of process */
      NextState = DoExit();
      break;
    } /* switch(State) */
    State = NextState;         /* Change state of the process  */
  } /* !Terminated */
} /* end of main */
/*
 *
 *
 * D o I n i t
 *
 * This function will register with the ACCESS Link Handler,
 * acquire a dedicated voice channel, and set up our
 * Meridian Mail event handlers. This function will only
 * be executed once during the the life span of the process.
 *
 */
ProcStates DoInit ()
{
```

```
struct  EnvInfo Env;        /* Env struct for hi lev api's  */
short   info;               /* Logon information            */
/*
 * Install the Meridian Mail Event Handlers
 * that we expect to occur in this application.
 * OnDigit and PlayEnd are automatically handled
 * by enabling the Hi Level APIs. All other events
 * can be ignored by this application (their occurrence
 * would not effect the state of this application) .
 * These events would include:
 *          m_OnNewMessage
 *          m_OnRecordEnd
 *          m_OnTimeout
 *          m_OnBRWarn
 */
m_OnIncomingCall(IncomingCallHandler);
m_OnCallProgress(CallProgHandler);
m_OnSessionEnd(SessionEndHandler);
m_OnError(MMErrorHandler);
/*
 * Set Timeout OFF so we do not
 * have to trap on TimeOut events from
 * MM (i.e. automatically handled by API
 * layer).
 *
 m_TimeoutOff();

/*
 * Register with MM Link Handler.
 * If we have problems stay in this state and
 * retry until we are asked to shutdown.
 *
 */
Status = NOTREGISTERED;
while (!m_Register(&RetCode) ) {
   Error("Can not Register with MM Link Handler: %d.",RetCode);
   if (ShutDown) {          /* shutdown if we've been asked */
      return(EXIT);
   }
   sleep(3);                /* wait a bit and try again     */
} /* while !m_Register */
Status = REGISTERED;        /* we are now registered        */
/*
 * Acquire a dedicated voice channel from MM.
 * If we have problems stay in this state and
 * retry until we are asked to shutdown.
 *
 */
while (!m_Acquire(Class,&RetCode) ) {
```

```
    Error("Can not Acquire a Dedicated MM Channel: %d.",RetCode);
    if (ShutDown) {          /* shutdown if we've been asked */
      return(EXIT);
    }
    sleep(3);               /* wait a bit and try again     */
  } /* while !m_Acquire */
  Status = ACQUIRED;          /* we've  acquired a MM toolkit */
  /*
   * Logon to the designated MM acount.
   * If we have problems shutdown the process
   * (i.e. go to 'EXIT' state ).
   *
   */
  if (!m_Logon(User,Passwd,&info,&RetCode) ) {
    Error("Can not Logon into designated MM Account: %d.",RetCode);
    return(EXIT);
  } /* if !m_Logon */
  /*
   * Initialize the High Level API Commands
   * If we have problems shutdown the process
   * (i.e. go to 'EXIT' state ).
   *
   */
  m_GetEnv(&Env,&RetCode);
  Env.HiLevel = TRUE;
  if (!m_SetEnv(&Env,&RetCode) ) {
    Error("Can not set environment: %d.",RetCode);
    return(EXIT);
  } /* if !m_SetEnv */

  /*
   * Open the "lotto" voice files on MM.
   * If we have problems shutdown the process
   * (i.e. go to 'EXIT' state ).
   *
   */
  if (!m_OpenFile(LOTTO_VOICE_FILE ,"r",&FileHandle,&RetCode) ) {
    Error("Can not open LOTTO voice file on MM %s: %d.",
          LOTTO_VOICE_FILE, RetCode);
    return(EXIT);
  } /* if !m_OpenFile */
  CallisActive = FALSE;       /* No active call at this point */
  if (ShutDown)
    return(EXIT);           /* We've been asked to shutdown */
  else
    return(WAIT);           /* Everthing OK return nxt state*/
} /* end DoInit */
/*
 *
```

```
 *
 * D o W a i t
 *
 * This function simply waits for the next incoming call to
 * arrive. It will pause indefinitely until it is interrupted
 * be an incoming call or a termination signal. At this point
 * the process enters the ANSWER or EXIT states respectively.
 *
 */
ProcStates DoWait ()
{
  /*
   * Before we accept and wait for
   * for the next call lets make
   * sure the previous call has
   * disconnected first (i.e. wait for
   * Call Progress disconnect event) .
   *
   */
  if (!CallisActive) {
    /*
     * We are ready to accept Calls.
     * Wait for the next incoming Call or
     * we are interrupted by a MM event.
     * Do not acquire a new channel.
     *
     */
    if ( (!m_WaitingForCall(MAXWAIT,FALSE,&RetCode)) &&
         (RetCode != MME_OPER_TIMEOUT) ){
      Error("Can not Wait for any Incoming Calls: %d.", RetCode);
    } /* if !WaitingForCall */
  } else {
                        /* Disconnect the current Call  */
     if (!m_DisconnectCall(&RetCode) ) {
       Error("Could not Disconnect Call: %d. ", RetCode);
       CallisActive = FALSE; /* we can't possible have a Call*/
     }
     sleep(5);              /* wait for the disconnect event*/
  } /* !CallisActive */
  /*
   * If we reach this point we have either have
   * a call or have been interrrupted by a MM event.
   * Determine the source of the interruption and
   * process it accordingly.
   *
   */
  if (ShutDown)
    return(EXIT);           /* We've been asked to shutdown */
```

```
  if (CallisActive)
    return(ANSWER);          /* Received Call return nxt stat*/
  return(WAIT);              /* Otherwise, stay in this state*/
} /* end DoWait */
/*
 *
 *
 * D o A n s w e r
 *
 * This function will answer the incoming call and greet the
 * caller with a welcome prompt.
 *
 */
ProcStates DoAnswer ()
{
                           /* the ”Welcome” Prompt        */
static short welcome_prompt[] = { WELCOME, 0 } ;
  /*
   * Call is ringing, try and answer it.
   *
   */
  if (!m_AnswerCall(&RetCode)) {
    Error(”Could not Answer Incoming Call: %d.”, RetCode);
    CallisActive = FALSE;     /* we can’t possible have a Call*/
    return(WAIT);             /* Can’t answer, WAIT next call */
  } /* m_AnswerCall */
  if (CallisActive) {
    /*
     * Call is Active , Greet
     * the Caller with Welcome
     * Prompt.
     *
     */
                           /* play–30 sec TO, interruptable*/
                           /* do clear the key buffer      */
    if (!m_PlayPrompt(FileHandle, welcome_prompt, 30,
                TRUE, TRUE, &RetCode)) {
      Error(”Could not Play Welcome: %d. ”, RetCode);
      return(WAIT);          /* Error Playing, WAIT nxt call */
    }
    RandomNums[1] = 0;        /* init the random no. array    */
    return(ENTSEED);          /* Call active, return nxt state*/
  } else {
    return(WAIT);             /* Caller hung up, WAIT nxt call*/
  } /* CallisActive */
} /* end DoAnswer */
/*
 *
 *
```

```
 * D o E n t e r
 *
 * This function will ask the Caller to enter a ”seed”
 * which will be used to generated the 6 random numbers.
 *
 */
ProcStates DoEnter  ()
{
  short done;                /* Loop completion flag        */
  short interrupt;           /* Boolean: interruptable play  */
  short patience;            /* our patience level          */
                             /* the ”Enter” Prompt          */
  static short enter_prompt[] = { ENTER_SEED, 0 };
                             /* the ”HELP ” Prompt for enter */
  static short help_prompt[] =  { HELP_SEED, 0 };
                             /* the ”ERROR” Prompt for enter */
  static short error_prompt[] = { ERROR_ATT, GOOD_BYE, 0 };
  short *prompt;             /* the Current Prompt          */
  /*
   * Prompt the Caller to enter a SEED
   * using key pad of a touch tone phone.
   * Check for time outs and play ”HELP” prompts.
   * Try up to 3 times before exiting.
   *
   */
  prompt = enter_prompt;       /* set current prompt to ”Enter”*/
  patience = 1;                /* set our patience level      */
  interrupt = TRUE;            /* set interruptable prompts    */
  done = FALSE;
  while(!done) {
                             /* play–30 sec TO, interruptable*/
                             /* do not clear the key buffer  */
    if (!m_PlayPrompt(FileHandle, prompt, 30, interrupt, FALSE, &RetCode)) {
      Error(”Could not Play Enter Seed: %d. ”, RetCode);
      return(WAIT);          /* Error Playing, WAIT nxt call */
    }

    if (patience <= 3) {      /* Have we exhausted patience ? */
                             /* get upto 5 digits Term by *  */
                             /* or #, Do not clear buffer     */
      if ( (!m_CollectDigits(5,”*#”,Seed,&LastKey, FALSE, &RetCode)) &&
          (RetCode != MME_INTER_KEY_TO ) ){
        Error(”Could not Collect Seed: %d. ”, RetCode);
        return(WAIT);        /* Error Collect, WAIT nxt call */
      }

      if (RetCode != MMS_OKAY){ /* Check for Digit Time Out  */
        if (++patience > 3) /* We’ve exhausted patience ?   */
          prompt=error_prompt;/* set curr prompt to Error  */
```

```
          else
            prompt=help_prompt; /* set curr prompt to Help   */
        } else {
          srand(atoi(Seed));  /* Convert the seed to integer  */
                        /* and Plant the Seed           */
          return(ENTMENU);    /* Got Seed, return nxt state   */
        } /* RetCode == MMS_OKAY  */
      } else {
                        /* Disconnect the current Call  */
        return(WAIT);         /* by going to WAIT state       */
      } /* patience */
      interrupt = FALSE;        /* set non–interruptable prompts*/
    } /* while !done */
} /* end DoEnter */
/*
 *
 *
 * D o M e n u
 *
 * This function will ask the Caller to make one of
 * the following Menu  selections:
 *   1 – generate 6 random numbers between 1 and 49
 *   2 – generate 6 random numbers between 1 and 80
 *   3 – to repeat the last numbers generated
 *   4 – quit
 *
 */
ProcStates DoMenu  ()
{
  short i,j;                /* local indices                */
  short done;               /* Loop completion flag         */
  short foundnum;               /* Boolean: found double numbers*/
  short interrupt;              /* Boolean: interruptable play  */
  short patience = 1;           /* set our patience level       */
                        /* the ”MENU ” Prompt           */
  static short menu_prompt[] =  { ENTER_MENU, 0 };
                        /* the ”HELP ” Prompt for menu  */
  static short help_prompt[] =  { HELP_MENU, 0 };
                        /* another ”HELP ” Prompt       */
  static short help2_prompt[] = { HELP_MENU2, 0 };
                        /* the ”ERROR” Prompt for menu  */
  static short error_prompt[] = { ERROR_ATT, GOOD_BYE, 0 };
                        /* the ”BYE” Prompt for menu    */
  static short bye_prompt[] =   { THANK_YOU, GOOD_BYE, 0 };
  short *prompt;                /* the Current Prompt           */
  /*
   * Prompt the Caller to make a Menu selection
   * using key pad of a touch tone phone.
   * Check for time outs and play ”HELP” prompts.
```

```
 * Try up to 3 times before exiting.
 *
 */
prompt = menu_prompt;        /* set current prompt to ”Menu” */
patience = 1;                /* set our patience level       */
interrupt = TRUE;            /* set interruptable prompts    */
done = FALSE;
while(!done) {
                        /* play–30 sec TO, interruptable*/
                        /* do clear the key buffer      */
  if (!m_PlayPrompt(FileHandle, prompt, 30, interrupt, TRUE, &RetCode)) {
    Error(”Could not Play Menu Selection: %d. ”, RetCode);
    return(WAIT);          /* Error Playing, WAIT nxt call */
  }

  if (patience <= 3) {      /* Have we exhausted patience ? */
                        /* get 1 digit only, No Term.   */
                        /* Do not clear key buffer      */
    if ( (!m_CollectDigits(1,””,MenuSel,&LastKey, FALSE, &RetCode)) &&
        (RetCode != MME_INTER_KEY_TO ) ){
      Error(”Could not get Menu Selection: %d. ”, RetCode);
      return(WAIT);       /* Error Collect, WAIT nxt call */
    }

    if (RetCode != MMS_OKAY){ /* Check for Digit Time Out  */
      if (++patience > 3) /* We’ve exhausted patience ?   */
        prompt=error_prompt;/* set curr prompt to Error  */
      else
        prompt=help_prompt; /* set curr prompt to Help   */
    } else {
      switch(LastKey) {   /* Got menu selection, check it */
      case ’1’:           /* Selected item 1: 6/49        */
                        /* Generate 6 numbers (1..49)   */
        i = 1;
        while ( i < 7) {
          RandomNums[i] = ( (rand())%49 ) + 1;
          foundnum = FALSE;
                        /* has number been prev. gen. ? */
          for (j=1; j<i; j++)
            if (RandomNums[i] == RandomNums[j] )
              foundnum = TRUE;
                    /* generate next num if not found*/
          if (!foundnum)
            i++;
        }
        return(PLAY);    /* PLAY the numbers to caller   */
        break;
      case ’2’:           /* Selected item 2: 6/80        */
                        /* Generate 6 numbers (1..80)   */
```

```
            i = 1;
            while ( i < 7) {
              RandomNums[i] = ( (rand())%80 ) + 1;
              foundnum = FALSE;
                        /* has number been prev. gen. ? */
              for (j=1; j<i; j++)
                 if (RandomNums[i] == RandomNums[j] )
                   foundnum = TRUE;
                       /* generate next num if not found*/
              if (!foundnum)
                i++;
            }
            return(PLAY);    /* PLAY the numbers to caller   */
            break;
          case '3':          /* Selected item 3: Repeat Last */
                             /* if there is something to rep */
            if (RandomNums[1] == 0 ) {
              if (++patience > 3) /* We've exhaust patiencE?*/
                prompt=error_prompt; /* set prompt to Err  */
              else
                prompt=help2_prompt; /* set prompt to Help2 */
            } else {
              return(PLAY);  /* PLAY the numbers to caller   */
            }
            break;
          case '4':          /* Selected item 4: Quit        */
            prompt=bye_prompt;  /* set curr prompt to Bye &  */
                             /* Disconnect the current Call  */
                             /* by faking patience           */
            patience = 4;
            break;
          default:
            if (++patience > 3) /* We've exhausted patience ?*/
              prompt=error_prompt;/* set curr prompt to Err */
            else
              prompt=help_prompt; /* set curr prompt to Help */
            break;
          } /* switch LastKey */
        } /* RetCode == MMS_OKAY  */
      } else {
                             /* Disconnect the current Call  */
        return(WAIT);        /* by going to WAIT state       */
      } /* patience */
      interrupt = FALSE;     /* set non–interruptable prompts*/
    } /* while !done */
  } /* end DoMenu */
  /*
   *
   *
```

```
 * D o P l a y
 *
 * This function will play the numbers
 * generated randomly from the Seed.
 * The Caller can always interrupt the
 * playing by hitting any key on his touch
 * tone phone.
 *
 */
ProcStates DoPlay  ()
{
  if (CallisActive) {         /* Check if the call is Active  */
                        /* play the caller's lucky nums */
                        /* wait 30 sec for playend, Do  */
                        /* not allow inter. and clearbuf*/
    if (!m_PlayPrompt(FileHandle, RandomNums, 30, FALSE, TRUE, &RetCode)) {
      Error("Could not Play Caller's Numbers: %d. ", RetCode);
      return(WAIT);         /* Error Playing, WAIT nxt call */
    } else
      return(ENTMENU);      /* Everthing OK, goto MENU state*/
  } else {
    return(WAIT);           /* No Call, goto WAIT state     */
  } /* CallisActive */
} /* end DoPlay */
/*
 *
 *
 * D o E x i t
 *
 * This routine will gracefully exit the program.
 *
 */
ProcStates DoExit ()
{
  /*
   * Exit the program gracefully.
   * No need to check error codes
   * since are only interested in
   * terminating the program.
   *
   */
  switch (Status) {

  case ACQUIRED    :         /* Have acquired a MM session,  */
    m_Release(&RetCode);     /* release will clean up for us */
                        /* then fall through to Dereg.  */

  case REGISTERED  :         /* Have registered with LH.     */
    m_Deregister(&RetCode);  /* Free up our comm. with the LH*/
```

```
  case NOTREGISTERED:          /* No connections yet, there is */
                               /* nothing to do but terminate. */
    break;
  } /* switch */
  Terminated = TRUE;           /* We are now terminated        */
  return(EXIT);
} /* DoExit */
/*
 *
 *
 * I n c o m i n g C a l l H a n d l e r
 *
 * "On Incoming Call" Event Handler Routine.
 * Automatically called when a call is placed
 * to the voice channel associated with the
 * active ACCESS session.
 *
 */
void IncomingCallHandler (FromDN,ToDN)
char  *FromDN;               /* DN of caller (if Known)          */
char  *ToDN;                 /* DN of local set                  */
{
  CallisActive = TRUE;
} /* IncomingCallHandler */
/*
 * C a l l P r o g H a n d l e r
 *
 * "Call Progress" Event Handler Routine.
 * Automatically called when the current call
 * has changed states.
 *
 */
void CallProgHandler (StateChange,StateInfo)
short StateChange;           /* New State of call            */
short StateInfo;             /* Info about StateChange       */
{
  /*
   * Only expected state is CP_DISCONNECT.
   * Set CallisActive to let mainline know
   * that call has disconnected.
   */
  if (StateChange == CP_DISCONNECT)
       CallisActive = FALSE;
} /* End of CallProgHandler */
/*
 * S e s s i o n E n d H a n d l e r
 *
 * "On Session End " Event Handler Routine.
```

```
 * Automatically called when MM releases the
 * the current session and voice channel. Note
 * this is always initiated by MM and not the
 * application itself.
 *
 */
void SessionEndHandler(Reason)
short  Reason;            /* Reason for the session end       */
{
   CallisActive = FALSE;   /* We definitely do not have a Caller*/
   ShutDown = TRUE;        /* Initiate graceful shutdown Process*/
   Error("We have received Session End from MM: %d.",Reason);
} /* SessionEndHandler*/
/*
 * M M E r r o r H a n d l e r
 *
 * "On Error Notification" Event Handler Routine.
 * Automatically called when MM notifies the
 * the current session some type of asynchronous
 * error has occured.
 *
 */
void MMErrorHandler(Type)
short  Type;              /* The type of async error on MM     */
{
   Error("We have received Async Error from MM: %d.",Type);
} /* MMErrorHandler */
/*
 * S i g n a l H a n d l e r
 *
 * This is our common UNIX signal handler
 * for SIGHUP,SIGINT,SIGQUIT, and SIGTERM.
 * Opon receipt of any of the above signals
 * the process will print a message and
 * commence a graceful shutdown.
 *
 */
void TermSignalHandler()
{
   /*
    * We got a UNIX signal, ignore any
    * other signal coming in.
    *
    */
   signal(SIGHUP,SIG_IGN);
   signal(SIGINT,SIG_IGN);
   signal(SIGQUIT,SIG_IGN);
   signal(SIGTERM,SIG_IGN);
```

```
  /*
   * Check to see if
   * Shutdown already in
   * Process
   */
  if (ShutDown) {
    Error("Shutdown already in progress.");
  } else {
    Error("We have recieved a local Termination Signal.");
    if (CallisActive)
      Error("Commencing Shutdown after processing current call.");
  }
  ShutDown = TRUE;        /* Initiate graceful shutdown Process*/
  /*
   * Restore all the UNIX signals
   * once again.
   *
   */
  signal(SIGINT,TermSignalHandler);
  signal(SIGQUIT,TermSignalHandler);
  signal(SIGTERM,TermSignalHandler);
  signal(SIGHUP,TermSignalHandler);
} /* TermSignalHandler */
/*
 * E r r o r
 *
 * This is our common error handler.
 * It will simply print the given error
 * message on the user's stdin screen.
 * In a real application one will probably
 * want to log it to an error file and/or
 * inform a parent process that an error
 * has occured.
 *
 */
void Error (fmt, a1, a2, a3, a4, a5)
char  *fmt;
{
  printf(fmt,a1,a2,a3,a4,a5);  /* print message buffer        */
  printf("\n");
}
```

Index

A

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V

W

NORTEL
NORTHERN TELECOM

Reader's Response Form for

Meridian ACCESS
***Developer's Guide*, (NTP 555-7001-316)**
January 1998

Tell us about yourself:

Name: _____ **Date:** _____

Company: _____

Address: _____

_____

Occupation: _____ **Phone:** _____

1. What is your level of experience with this product?

   ❑ New user ❑ Intermediate ❑ Experienced ❑ Programmer

2. How do you use this book?

   ❑ Learning ❑ Procedural ❑ Problem solving ❑ Reference

3. Did this book meet all of your needs?

   ❑ Yes ❑ No

   If you answered **No** to this question, please answer the following questions.

4. What chapters, sections, or procedures did you find hard to understand?

   _____

   _____

   _____

5. What information (if any) was missing from this book?

   _____

   _____

   _____

6. How could we improve this book? (For example, books can also be evaluated in many other ways, including: ease of information retrieval, presentation, and use of reading aids, such as diagrams.)

   _____

   _____

Please return your comments by fax to (416) 597-7104, or mail your comments to: Nortel Product Training and Documentation, Toronto Lab, 522 University Ave., Toronto, Ontario, Canada. M5G 1W7.

Reader's Response Form

Meridian ACCESS

Developer's Guide

Toronto Information Products
Nortel
522 University Avenue, 12th Floor
Toronto, Ontario, Canada
M5G 1W7

Publication number: 555-7001-316
Product release: 12
Document release: Standard 1.0
Date: January 1998

Printed in the United States of America

NORTEL
NORTHERN TELECOM