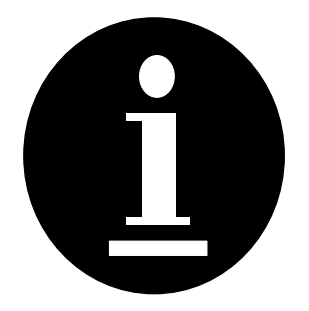
iButton-TMEX

PROFESSIONAL SOFTWARE DEVELOPER'S KIT

Version 3.10



Dallas Semiconductor

DS0621-SDK

rev April 1,1998

Copyright (c) 1992-1998 Dallas Semiconductor

Contents

Introduction

The Dallas Semiconductor DS0621-SDK iButton-TMEX Professional Software Developer's Kit, Version 3.10, is a tool for professional programmers to facilitate the development of programs utilizing the TMEX Application Program Interface (API) calls. TMEX is the set of drivers, utilities and other interface modules required to interface with the Dallas Semiconductor iButton products utilizing the 1-Wire protocol. iButtons are high-capacity, general-purpose electronic data carriers, each with a unique registration serial number. Communication to iButtons is done over 1 data line and ground using the '1-Wire protocol'. This wire in conjunction with the protocol is called the 'MicroLan'. The DS0621-SDK iButton-TMEX Professional Software Developer's Kit, henceforth referred to as SDK, contains software drivers, utilities, example source code and a design guide. The iButton-TMEX version 3.10 software drivers provided in DS0621-SUL version 3.10 support DOS and Microsoft Windows 3.1/95/NT environments.

This kit was designed to work in conjunction with iButton-TMEX (DS0621-SUL), or just 'iButton-TMEX'. IButton-TMEX will install the drivers for each platform and make the computer ready for applications created with this developer's kit. IButton-TMEX comes on three disks with three different setup programs. There is a disk for DOS, Windows 3.1 and Windows 95/NT.

The TMEX version 3.10 drivers have been successfully tested with:

- Microsoft DOS Versions 5.0 and 6.2
- Microsoft Windows 3.1 and 3.11
- Windows 95 (DOS, 16, and 32 bit drivers) Version 4.0
- Windows NT (32 bit drivers only) Version 3.51

This Software Developer's Kit provides the following components:

- A general design guide for writing iButton applications using the TMEX Application Program Interface (API) calls.

- TMEX drivers (TSRs and DLLs) for PCs and other similar operating environments, including several different hand-held computers. The supported hardware interfaces for these drivers are the PC COM port using the DS9097E PC Serial Port Adapter or the PC parallel port using the DS1410E PC Parallel Port Adapter. The modularity of these drivers supports expansion into other hardware media by writing custom low level hardware interfaces. Also included is a description of all the application program interface (API) calls provided in TMEX version 3.10 and how to make these calls from DOS and Microsoft Windows. Platforms other than DOS and Microsoft Windows for TMEX are under development.

- Description of the iButton-TMEX utility programs provided in the SDK. There are simple utility/example programs for execution under DOS which perform standard file operations on TMEX data files, similar to the familiar file management utilities

(FORMAT, DIR, COPY, TYPE, etc.) available in DOS. A description of each of these utilities can be found in the section '3.1. TMEX DOS Utility Programs'. These utilities return meaningful error codes to DOS upon completion. A program could spawn these utility programs and use the error codes to interpret the results of the operation.

- Example programs have been written in several different languages in the DOS and Microsoft Windows environment to illustrate the usage and calling methodology required for each language in calling TMEX. The source for all of the example programs is in sub-directories under \SOURCE. The languages used to implement the examples are, 'Borland Pascal', 'Borland C', 'Microsoft Visual Basic', 'Quick Basic', 'Borland Delphi', and 'Microsoft Visual C'.

- A description of the entire Extended File Structure for iButton data with special notes indicating the level of implementation in TMEX 3.10. This will allow users to develop iButton applications which will be compatible with the current and future editions of TMEX.

- A list and description of TMEX 3.10 DOS and Microsoft Windows software drivers.

- A description of the Universal Data Packet (UDP). The UDP is the structure used in storing data on iButtons using the Extended File Structure.

The SDK files are organized into three main sub-directories BIN, DRIVERS, and SOURCE. BIN contains all the executables of the DOS and Microsoft Windows utilities and examples. DRIVERS contains the DOS TSR and Microsoft Windows DLL TMEX drivers. SOURCE contains the source code for the TMEX example programs.

The programming code examples in this documentation will be in 'C'. This is not the only programming language that can call on TMEX. TMEX was designed to be language independent.

The documentation and the accompanying drivers, utility programs, and programming examples should serve as a comprehensive foundation on which to develop highly effective iButton application programs.

Chapter 1 TMEX APPLICATION DESIGN GUIDE

Most communication with iButtons is accomplished through some sort of temporary contact. Since this contact is temporary the underlying design philosophy of any iButton application is 'try and try again'. It must be assumed at any time the user of the application will break contact temporarily or permanently with the iButton that the application was communicating with. With this in mind, TMEX provides a wide variety of application programming interface (API) calls to verify which iButton is being communicated with and the results of that communication.

Each iButton has a guaranteed unique serial number (ROM). TMEX provides API calls that read these CRC verified serial numbers. Since each serial number is unique, multiple iButtons can be communicated with on the same communication channel. This channel is referred to as a 1-Wire since communication is done over 1 wire and ground. Each data-oriented API call to an iButton is preceded by the unique serial number thus preventing any ambiguity.

To facilitate data integrity in the iButton, all data is stored in a structure that contains a length, data, and an inverted CRC-16 of that length and data. This structure is referred to as the Universal Data Packet (UDP). See APPENDIX B for details on UDP.

Since the Microsoft Windows environment is multi-tasking, TMEX provides the ability for several applications to communicate on the same MicroLan. This is accomplished through the use of semaphore functions that provide a 'session_handle' that gives exclusive use of that MicroLan. The time that this 'session_handle' is valid is referred to as a Session. To be 'friendly' to other applications the duration of a Session should be as small as possible. Each application running in Microsoft Windows may have a different state so each call to the TMEX API must provide a buffer for state variables.

All of the TMEX API calls are designed such that if their operation is interrupted no data corruption occurs. An interruption would most likely occur due to intermittent contact. To complete an interrupted function, simply call it again. If the device does not return to the MicroLan then that call was not done and the iButton has not been changed.

There is one exception to the rule that API calls can be interrupted, programming EPROMs. Any EPROM programming is destructive, meaning it uses up data space. To get around this TMEX has two special API calls, TMCreateProgramJob and TMDoProgramJob. TMCreateProgramJob sets up a buffer that keeps track of any writes to the EPROM without actually doing the write. TMDoProgramJob then proceeds to transfer the buffered changes to the EPROM device. TMDoProgramJob MUST complete successfully for the data integrity of the EPROM device to remain. TMDoProgramJob is set up such that if it returns an ERROR it can be called again to complete the operation. Programming EPROMs requires special hardware. For the PC COM port this hardware is a DS9097E PC Serial Port Adapter with an optional but

recommended external power supply.

Here is a possible iButton Session. Note that all TMEX API calls are prefixed with 'TM'.

TMExtendedStartSession - get permission to communicate on the MicroLan. (not applicable in DOS)

TMSetup - called only once at the beginning of the application to verify that the MicroLan exists.

TMFirst,TMNext,TMStrongAccess,TMRom... - get the unique registration number of the iButton to communicate with.

TMCreateProgramJob - called only if the iButton is an EPROM device AND a write operation is going to be attempted.

TMFirstFile,TMChangeDirectory,TMDeleteFile,TMFormat... - do a file operation.

TMDoProgramJob - called only if TMCreateProgramJob was called. Keep calling until it returns success. If the iButton loses contact, have the application force the user to make contact again.

TMClose - called only once at the end of the application to power down the MicroLan. (not always applicable, but always callable)

TMEndSession - relinquish use of the MicroLan. (not applicable in DOS)

Chapter 2 TMEX API

The TMEX API is divided into five layers.

- Session
- File_Operations
- Transport
- Network
- Hardware_Specific

The Hardware_Specific layer contains the only API calls that are hardware specific. If a new hardware interface was needed only these API calls and the Session Layer to arbitrate there use would have to be re-written. The Session layer allows MicroLan arbitration in a multi-tasking environment. This layer is not used in DOS since DOS is normally not multi-tasking. The Network layer refers to API functions that read and verify the unique serial registration numbers of the iButtons. The Transport layer has data functions such as reading and writing Universal Data Packets. The File_Operations layer contains API calls to read and write files and do directory operations. Here is a partial list of the TMEX 3.10 drivers and layers implemented in each driver.

```
               -File_Operations
               |
               |       -Transport/Network
               |       |
               |       |       -Hardware_Specific
               |       |       |
               |       |       |       -Session
               |       |       |       |      (note)
-------------------------------------------------------------------------
DOS - Terminate and start resident (TSRs)
-----------------------------------------------
IBFS97E.EXE    ======================>       (DS9097E, COM)
IBFS97U.EXE    ======================>       (DS9097U, COM)
IBFS10E.EXE    ======================>       (DS1410E, LPT)
IBFS.EXE       ==============>               (no hardware specific)

-------------------------------------------------------------------------
WINDOWS - Dynamic link libraries (DLL's)
-----------------------------------------------
IBFS.DLL       ==============>               (no hardware specific) (WIN31)
IB97E.DLL                     =============>(DS9097E, COM) (WIN31)
IB97U.DLL                     =============>(DS9097U, COM) (WIN31)
IB10E.DLL                     =============>(DS1410E, LPT) (WIN31)
IBFS32.DLL     ==============>               (no hardware specific) (WIN95NT)
IB97E32.DLL                   =============>(DS9097E, COM) (WIN95NT)
IB97U32.DLL                   =============>(DS9097U, COM)(WIN95NT)
IB10E32.DLL                   =============>(DS1410E, LPT) (WIN95NT)
IBFSCE.DLL     ==============>               (no hardware specific) (WINCE)
IB97ECE.DLL                   =============>(DS9097E, COM) (WINCE)
IB97UCE.DLL                   =============>(DS9097U, COM)(WINCE)
```

The Capability of the adapters

	Port Type	Overdrive	Strong Pullup	ProgramPulse
Parallel Port DS1410D	2			
DS1410E	2	X	X	
Serial Port DS9097 DS1413	1			
DS9097E	1			X
DS9097U	5	X	X	X
DS9097U-9 DS1411	5	X	X	

The DS1410D and DS1410E are Parallel Port Adapters that work on the LPT port (Type 2) and do not support programming the EPROM-based iButton products. The DS1410E Adapter supports Overdrive which is a faster communication rate that some iButton products support. This adapter also supports Strong Pull-up which provides power to special purpose iButtons such as the temperature iButton..

The DS9097Xs, DS1413, and DS1411 are Serial Port Adapters. The DS9097, DS1413 and DS9097E Adapters work on the COM port (Type 1) and do not support Overdrive or Strong Pull-up. The DS9097U, DS9097U-9 and DS1411 adapters work on the COM port (Type 5) and support Overdrive and Strong Pull-up. The DS9097E, DS9097U, and DS1411 adapters support programming voltage. They are capable of producing a 480 microsecond 12 volt pulse on the MicroLan to program iButton EPROM devices such as the DS1982. Caution must be maintained when using programming voltage, however, since any non-EPROM iButtons on the MicroLan during a program pulse may be damaged.

The basic design of TMEX is patterned after the International Standards Organization (ISO) reference model of Open System Interconnection (OSI), which specifies a layered protocol having up to seven layers, denoted as Physical, Link, Network, Transport, Session, Presentation, and Application. According to this model, the electrical and timing requirements of iButton and the characteristics of the MicroLan comprise the Physical layer. The software API calls TMTouchReset, TMTouchByte, and TMTouchBit correspond in this model with the Link layer, which also includes hardware initialization and fault detection facilities. The Link layer is referred to as the Hardware_Specific layer above. The multi-drop functions TMFirst, TMNext, TMAccess, etc., that support selection of individual network nodes correspond to the Network layer. The software that transfers NV RAM data to and from individual network

nodes corresponds to the Transport layer. In multi-user or multi-tasking environments such as Microsoft Windows, a Session layer effectively manages sharing of the MicroLan among multiple instances (simultaneous invocations) of iButton application programs. The Presentation layer provides a file structure that allows iButton data to be organized into independent files and randomly accessed (as with a diskette). The Application layer represents the final application program designed by the software developer.

2.1 DOS TMEX Considerations

The drivers for DOS are in the form of TSRs. These TSRs must be installed before an iButton application is run. The iButton-TMEX installation program will automatically copy the drivers into a sub-directory and modify the AUTOEXEC.BAT to install the selected TMEX driver. It is recommended that the end user install iButton-TMEX and then install the target application. The default port is also set in the installation in the form of an environment variable “OneWirePort=X” where X is the default port number. It is recommended that while running a TMEX compatible application, the TMEX default port be used as the the application's default port. It is further recommended that when the application's port is changed, the TMEX default port be unaffected.

Calling TMEX under DOS consists of making interrupt calls to the interrupts on which TMEX installs itself. To use all of the TMEX API in DOS there must be at least two free interrupts in the range 60 Hex to 66 Hex. One interrupt is used by the File_Operations layer and contains an ID string that begins with 'TMEX'. The ID string is described below. The other interrupt is used for the Network_Transport layer and Hardware_Specific layer and have an ID string that begins with 'DOW'. A DOS application program would first have to find the appropriate interrupts to call. Each TMEX interrupt has a specific structure at the interrupt call that can be recognized by an application. The interrupts are all software interrupts in the range of 60 Hex to 66 Hex. The TMEX interrupt contains a vector to an interrupt service routine having the following structure:

- Field #1
  This field contains a five byte far jump to the main entry point of the service routine (field #4). The contents of the field are:

  i. EA Hex (The one byte far jump instruction.)
  ii. Two byte offset address of field #4.
  iii. Two byte segment address of field #4.

- Field #2
  This field consists of one byte specifying the length of the ID string which follows in field #3.

- Field #3
  This field contains the body of the ID string. At the end of the ID string is a NULL (0) character not included in the length. A detailed explanation of the TMEX 3.10 DOS ID string can be found in APPENDIX A.

- Field #4
  Actual interrupt service executable code. (The placement of the code at this offset from the base of the structure is optional, since the far jump of field #1

allows placement of this field number anywhere within the range of the far jump.)

Here is the same structure expressed as a 'C' structure:

```
Struct
{
    const char d = 0xEA;    /* far jump instruction */
    void far *start;        /* pointer to start of ISR */
    unsigned char len;      /* length of ID string */
    char IDString[];        /* ID String */
} ISR_Header;
```

A technique to find the TMEX interrupts would be to start at interrupt 60 Hex and see if it had an interrupt vector. If it did then check to make sure that the first instruction is a far jump. If it is then read the sixth byte as the length of the ID string followed by the ID string. If the ID string was not correct then search the next interrupt up to 66 Hex. The File_Operations interrupt will have an ID string that begins with 'TMEX' and the Network_Transport / Hardware_Specific interrupt has an ID string that begins with 'DOW'. Following these designators are hex values that represent the number of valid API calls supported and following that is the version number and date. The function 'FindInts' in the library TMEXLIB.C provided in the \SOURCE\UTILS directory does this search and returns the valid interrupt numbers. TMEXLIB.C also contains calls to all of the TMEX API calls. The interface to these calls is similar but not identical to the Microsoft Windows TMEX API. The main difference is where the TMEX DLL has a 'session_handle' and 'state_buffer' TMEXLIB.C just has a 'port' argument.

A detailed description of all of the Version 3.10 TMEX drivers and their version strings is provided in APPENDIX A.

2.2 Microsoft Windows TMEX Considerations

The TMEX DLL drivers must be in a directory that can be found by the Microsoft Windows application. The iButton-TMEX installation program will automatically copy the drivers into the Windows\System sub-directory. It is recommended that the end user install iButton-TMEX and then install the target application. The DLL drivers are designed such that there is a main driver and then a series of Hardware_Specific sub-drivers. For example in the Windows 3.1 environment, the main driver is IBFS.DLL. For Windows 95/NT the main driver is IBFS32.DLL. When requesting a MicroLan port, the port number and type are specified in the function TMExtendedStartSession. The main driver then calls one of the Hardware_Specific sub-drivers. Here is a list of the default port type drivers.

Windows 3.1

Type #	Port	Driver Name	Description
0	*na*	IBTSR.DLL	calls on DOS TSR Hardware_Specific
1	COM	IB97E.DLL	uses the DS9097E adapter on COM ports

2	LPT	IB10E.DLL	uses the DS1410E adapter on LPT ports
5	COM	IB97U.DLL	uses the DS9097U adapter on COM ports

Windows 95

Type #	Port	Driver Name	Description
1	COM	IB97E32.DLL	uses the DS9097E adapter on COM ports
2	LPT	IB10E32.DLL	uses the DS1410E adapter on LPT ports
5	COM	IB97U32.DLL	uses the DS9097U adapter on COM ports

Windows NT

Type #	Port	Driver Name	Description
1	COM	IB97E32.DLL	uses the DS9097E adapter on COM ports
2	LPT	IB10E32.DLL	uses the DS1410E adapter on LPT ports
5	COM	IB97U32.DLL	uses the DS9097U adapter on COM ports

Windows CE

Type #	Port	Driver Name	Description
1	COM	IB97ECE.DLL	uses the DS9097E adapter on COM ports
5	COM	IB97UCE.DLL	uses the DS9097U adapter on COM ports

These defaults can be changed or new drivers can be registered by means of an INI file in Windows 3.1 and the Registry in Windows 95 and NT. These default settings are set when iButton-TMEX is installed with the installation programs.

In Windows 3.1 the iButton-TMEX settings are kept in a initialization file called iButton.INI with the following section.

```
[iButton-TMEX General Settings]
MainDriver=IBFS.DLL
TYPE0=IBTSR.DLL
TYPE1=IB97E.DLL
TYPE2=IB10E.DLL
TYPE5=IB97U.DLL
DefaultPortNum=1
DefaultPortType=1
```

An additional TMEX Hardware_Specific driver can be added by including another type, for example: TYPE3=MYDRIVER.DLL This can be done with any text editor.

The Window 95/NT iButton-TMEX installation will save the default types to the Registry. The registry key that the values are saved under are:

\HKEY_LOCAL_MACHINE\Software\Dallas Semiconductor\iButton-TMEX\3.10

The values are all string values
MainDriver “IBFS32.DLL”
TYPE1 “IB97E32.DLL”
TYPE2 “IB10E32.DLL” (Win95,WINNT)
TYPE5 “IB97U32.DLL”
DefaultPortNum “1”
DefaultPortType “1”

During iButton-TMEX installation the default port number and type are also set. It is recommended that when starting your application for the first time, these defaults should be used. Should the user then change the port, this information should remain with the setting information with the specific application.

An additional TMEX Hardware_Specific driver can be added by adding another string value to the key, for example: TYPE3 MYDRIVER.DLL This can be done with the registry editor that comes with Windows 95 or NT.

The TMEX DLLs have an extra API call 'Get_Version' to get the version number of the main TMEX DLL. There is also an API call to get the version of each of the registered Hardware_Specific types called TMGetTypeVersion. It is recommended that these API calls be used and their information provided somewhere in the application, perhaps in an 'about' box.

Since Microsoft Windows is multi-tasking, each application must provide a state buffer to the TMEX driver in each API call. To have the state preserved from session to session make this buffer global and do not alter its contents. The size of this buffer must be at least 5K (5120) bytes if no EPROM write operation is needed in the application or 15K (15360) bytes if EPROM write operations will be done. The size of this state buffer may increase with later versions of TMEX.

2.3 TMEX API Quick Reference

This is a brief description of all of the TMEX 3.10 API calls divided by layer.

Session

TMExtendedStartSession - request a MicroLan port
TMEndSession - relinquish a MicroLan port
TMValidSession - check to see if the MicroLan port session is still valid

File_Operations

TMFirstFile - get first file name in the current directory
TMNextFile - get next file name in the current directory
TMOpenFile - open a file for reading
TMCreateFile - create a file for writing
TMCloseFile - close an opened/created file
TMReadFile - read an opened file
TMWriteFile - write a created file
TMDeleteFile - delete a file
TMFormat - format iButton for file structure
TMAttribute - change file/directory attributes
TMReNameFile - rename a file
TMChangeDirectory - read or change the current directory
TMDirectoryMR - Make or Remove a sub-directory
TMCreateProgramJob - ready the write buffer for EPROM programming
TMDoProgramJob - do the EPROM programming
TMWriteAddFile - append or alter an ‘AddFile’ on an EPROM iButton
TMTerminateAddFile - terminate an ‘AddFile’ with Universal Data Packets
Get_Version - read the version ID for the main driver

Transport

TMReadPacket - read a CRC16 validated packet
TMWritePacket - write a CRC16 validated packet
TMBlockIO - transfer a block to the MicroLan
TMExtendedReadPage - read a device-CRC validated page (regular or status)
TMProgramByte - program a byte in an EPROM iButton

Network

TMFirst - find the first device on a MicroLan
TMNext - find the next device on a MicroLan
TMAccess - select the current device
TMStrongAccess - verify that the current device is still there and select
TMStrongAlarmAccess - same as TMStrongAccess except it must be alarming
TMOverAccess - select the current device and switch it into overdrive speed
TMRom - read the current device ROM or set the ROM for the next select
TMFirstAlarm - find the first alarming device on a MicroLan
TMNextAlarm - find the next alarming device on a MicroLan
TMFamilySearchSetup - set the search to find a family type on the next search
TMSkipFamily - skip the current device family type on the next search
TMAutoOverDrive - automatically get in and out of overdrive speeds

Hardware_Specific

TMSetup - verify the port exists
TMTouchByte - one byte communication to MicroLan
TMTouchReset - reset iButtons on MicroLan
TMTouchBit - one bit communication to MicroLan
TMProgramPulse - send a programming pulse to MicroLan
TMOneWireLevel - set the 1-Wire communication level
TMOneWireCom - set the 1-Wire communication speed
TMClose - power down a port (not always applicable)
TMGetTypeVersion - read the version ID for the hardware driver
TMBlockStream – transfer a stream to the MicroLan

2.4 TMEX API Calls

All of the following function prototypes and source code examples in this documentation are written in ‘C’. TMEX by designed however is ‘Language independent’. For language specific function prototypes, refer to the source code examples included with this kit. The following types are used in the function prototypes and code examples:

short - 16 bit signed integer
long - 32 bit signed integer
char - 8 bit signed number
unsigned char - 8 bit unsigned number
far pascal - far function using pascal calling convention
void far * - generic far pointer
short far * - far pointer to a 16 bit signed integer
(other structure are defined within the specific API descriptions)

Session

The Session layer API calls control access to the MicroLans. TMExtendedStartSession returns a non-zero 'session_handle' if the specified MicroLan is available. This 'session_handle' is then used as an argument to ALL of the other TMEX API calls. Without a valid 'session_handle' from TMExtendedStartSession, no other API call will work. These API calls return the following Session error codes for the Microsoft Windows.

-200 => INVALID_SESSION - session not valid (not applicable in DOS TMEX)
-201 => HS_NOT_FOUND - Hardware_Specific driver not found and is required

TMExtendedStartSession

This API call takes as an argument a port number “PortNum” and a port type “PortType” of the MicroLan to be used. It returns the session handle number if the session has been established, and 0 to indicate that the MicroLan is busy with another session in progress. A session handle of less than 0 indicates that there is no driver for the indicated type number. See the section “2.2 Microsoft Windows TMEX Considerations” for details on what drivers are associated with which type numbers. It is recommended that the API call TMGetTypeVersion be used to verify the port type. A session handle should be used and then as soon as possible ended with TMEndSession. The session handle is good indefinitely if no other application attempts to start a new session on that MicroLan port. The session handle can become invalid under the following circumstances:

- the session handle has not been used for at least 1 second
  And
  a request for the same MicroLan has been made

- the handle has been used in the last second
  And
  the handle is over 35 seconds old
  And
  a request for the same MicroLan has been made

This API function also contains the additional argument of a far pointer. This is a reserved value and should be set to NULL.

Microsoft Windows TMEX DLL function prototype:

long far pascal TMExtendedStartSession(short PortNum, short PortType, void far *Reserved);

This function returns:
>0 => session on port given session_handle returned
=0 => port not available, it is being used by another application
..<0 => TMEX Session Error Return Codes

DOS TMEX TSR calling procedure:
not applicable

Code Example:

```
long session_handle;
short PortNum = 1, PortType = 1;

/* attempt to get a session on Port */
session_handle = TMExtendedStartSession(PortNum,PortType,NULL);

if (session_handle > 0)
{
    /* call TMEX API function with session_handle */
    ...
}
else if (session_handle == 0)
{
    /* port not available */
    ...
}
else if (session_handle == -1)
{
    /* failure, indicated type does not have a driver */
    ...
}

/* relinquish port */
TMEndSession(session_handle);
```

TMStartSession

This function is provided for compatibility with TMEX version 2.XX. TMEX 3.10 applications should use the TMExtendedStartSession function. This API call takes as an argument a port number "PortNumType" of the MicroLan to be used. It returns the session handle number if the session has been established, and 0 to indicate that the MicroLan is busy with another session in progress. A session handle should be used and then as soon as possible ended with TMEndSession. The session handle is good indefinitely if no other application attempts to start a new session on that MicroLan. The session handle can become invalid under the following circumstances:

- the session handle has not been used for at least 1 second
  And
  a request for the same MicroLan has been made

- the handle has been used in the last second
  And
  the handle is over 35 seconds old
  And
  a request for the same MicroLan has been made

Microsoft Windows TMEX DLL function prototype:

long far pascal TMStartSession(short PortNumType);

This function returns:
>0 => session on port given session_handle returned
=0 => port not available
<0 => TMEX Session Error Return Codes

DOS TMEX TSR calling procedure:
not applicable

Code Example:

```
long session_handle;
short port = 1;

/* attempt to get a session on Port */
session_handle = TMStartSession(port);

if (session_handle > 0)
{
    /* call TMEX API function with session_handle */
    ...
}
else
{
    /* fail condition, port not available */
    ...
}

/* relinquish port */
TMEndSession(session_handle);
```

Special Note:
The TMStartSession API call has a special function in the TMEX 3.10 drivers. The main driver IBFS.DLL has two standard and one flexible hardware specific interface type. The two standard hardware types are PC COM port with the DS9097E and the PC parallel port with the DS1410E. The flexible hardware type of IBFS.DLL is that it can communicate with a Hardware_Specific TSR that was installed before Microsoft Windows was started. In this way by creating one small TSR there is automatically a DOS and Microsoft Windows driver available for future and existing applications. The 32 version of the main driver IBFS32.DLL does not have the ‘TSR’ type driver. To select the default hardware types, OR (|) the 'port' number provided in the call to TMStartSession with 0x0200 for parallel port, 0x0100 for COM port and 0x0000 for the flexible hardware type. For example to open a session with COM port 2 call IBFS.DLL with:

```
port = 2;

session_handle = TMStartSession( 0x0100 | port );
...
```

TMValidSession

This API call determines whether a session is still valid with the specified session handle 'session_handle'. It returns a 1 if the session is still valid, and 0 otherwise. The conditions where a session could become invalid are described in TMExtendedStartSession.

Microsoft Windows TMEX DLL function prototype:

short far pascal TMValidSession(long session_handle);

This function returns:
0 => session_handle is no longer valid
1 => session_handle is still valid for port
<0 => TMEX Session Error Return Codes

DOS TMEX TSR calling procedure:

not applicable

Code Example:

```
/* session_handle from previous call to TMExtendedStartSession */
...

if (TMValidSession(session_handle) == 1)
{
    /* session is still valid */
    ...
}
else
{
    /* the session is no longer valid */
    ...
}
```

TMEndSession

This API call ends the session on the MicroLan that was designated by the session handle 'session_handle'. It returns a 1 if the session specified by 'session_handle' was successfully closed, and 0 if there was no valid session established with the specified session handle.

Microsoft Windows TMEX DLL function prototype:
short far pascal TMEndSession(long session_handle);

This function returns:
0 => session_handle already invalid
1 => session ended, session_handle no longer valid
<0 => TMEX Session Error Return Codes

DOS TMEX TSR calling procedure:
not applicable

Code Example:

```
/* session_handle from previous call to TMExtendedStartSession */
...

if (TMEndSession(session_handle) == 0)
{
    /* the session_handle was not valid to end */
    ...
}
else
{
    /* session successfully ended */
    ...
}
```

File Operations

The File_Operations layer API calls do file and directory operations. These API calls return the following FILE_OPERATIONS error codes for the Microsoft Windows and (DOS) TMEXLIB.C TMEX calls. The DOS TSR calls return the absolute value (non-negative) of these return codes with the carry set:

-1 => NO_DEVICE - no device found on MicroLan
-2 => WRONG_TYPE - wrong type of device for specified operation
-3 => FILE_READ_ERR - file/directory read error
-4 => BUFFER_TOO_SMALL - buffer is smaller than the file to read
-5 => HANDLE_NOT_AVAIL - all file handles are used
-6 => FILE_NOT_FOUND - file specified is not in the current directory
-7 => REPEAT_FILE - file already exists with name provided
-8 => HANDLE_NOT_USED - given file handle is not assigned a file
-9 => FILE_WRITE_ONLY - trying to read a write file handle

-10 => OUT_OF_SPACE - not enough room on device to write
-11 => FILE_WRITE_ERR - write error, part may have expired
-12 => FILE_READ_ONLY - trying to write a read file handle
-13 => FUNC_NOT_SUP - function not supported
-14 => BAD_FILENAME - illegal file/directory name
-15 => CANT_DEL_READ_ONLY - trying to delete read only file
-16 => HANDLE_NOT_EXIST - file handle does not exist
-17 => ONE_WIRE_PORT_ERROR - error in the MicroLan, MicroLan may not be setup using TMSetup
-18 => INVALID_DIRECTORY - directory given is invalid
-19 => DIRECTORY_NOT_EMPTY - directory must be empty to delete it
-20 => UNABLE_TO_CREATE_DIR - maximum depth of 10 sub-directories is reached
-21 => NO_PROGRAM_JOB - there is no program job started
-22 => PROGRAM_WRITE_PROTECT - device is written in a way that can not be changed
-23 => NON_PROGRAM_PARTS - there are non-EPROM parts on the MicroLan during an attempt to program
-24 => ADDFILE_TERMINATED - the AddFile is terminated and cannot be appended to
-200=> INVALID_SESSION - session not valid (not applicable in DOS TMEX)
-201=> HS_NOT_FOUND - Hardware_Specific TMEX driver not found and is required.

Note that the ROM pattern for the desired iButton must already be in the internal eight-byte buffer before any of the File_Operations API functions can be called. This can be accomplished by directly writing to the internal buffer by using the TMRom API call or by using one of the Network functions TMFirst, TMNext, TMFirstAlarm or TMNextAlarm. This constraint enables all of these API calls to be multi-drop compatible. 'multi-drop compatible' refers to successful operation even though there are more than one device on a single MicroLan.

TMFirstFile

This API call finds the first file in the current device and directory on the MicroLan specified by 'session_handle' and copies the information about the file into the buffer 'fentry'. The information is in the form of a structure. The organization of the structure is as follows:

```
typedef struct {
  unsigned char name[4];      /* four-character file name */
  unsigned char extension;    /* extension number, range 0 - 99, 127 */
  unsigned char startpage;    /* page number where file starts */
  unsigned char numpages;     /* number of pages occupied by file */
  unsigned char attrib;           /* file/directory attribute */
  unsigned char bitmap[32];   /* current bitmap of the device */
} FileEntry;
```

This is a packed structure on 1 byte boundaries. The ‘name’ is a four characters left justified with spaces. The ‘extension’ is the file extension in the range of 0-99 for normal files, 100 for an add file and 101 for a Monetary file. If the extension is 127 the entry is a directory entry. The ‘startpage’ indicates the page number that the file starts on. The number of pages is the estimated number of pages that the file takes up. ‘attrib’ indicates the attributes that the file or directory entry has. See ‘TMAttribute’ for an explanation of the supported attributes. The ‘bitmap’ indicates the used pages in the device. A ‘1’ bit indicates a used page and a ‘0’ indicates an unused page. The ‘bitmap’ has the least significant byte first with the first page being in the least significant bit of the first byte. The ‘bitmap’ of the device is read and placed in the FileEntry structure even if there are no files in the directory (return 0).

Microsoft Windows TMEX DLL function prototype:

short far pascal TMFirstFile(long session_handle, void far *state_buffer, FileEntry far *fentry);

The function returns:
>0 => first file entry is in buffer 'fentry'
0 => current directory has no files but bitmap returned in fentry
<0 => a FILE_OPERATIONS error has occurred

DOS TMEX TSR calling procedure:

Prior to function call:
- Upper nibble of AH is MicroLan number.
- Lower nibble of AH is 01 hex.

Call a File_Operations layer TMEX interrupt (60-66 hex) with 'TMEX' at the beginning of the ID string. See APPENDIX A for details.

Upon return from function:
- Carry is set if there is any error in execution.
- AL is the FILE_OPERATIONS error code if the carry is set.
- ES:BX is a far pointer (Segment:Offset) to the FileEntry of the first file. The FileEntry is a structure described above.

The TMEXLIB.C function prototype is:

short far pascal TMFirstFile(short port, FileEntry far *fentry);

The function returns the same as the TMEX DLL call.

Code Example for TMEX Microsoft Windows call:

```
long session_handle;
unsigned char state_buffer[15360];
FileEntry fentry;
char buf[100];
short result;
```

```
/* session_handle set from a call to TMExtendedStartSession */
...

/* get the unique registration number of the device to communicate with
using TMFirst,TMNext,TMRom... */
...

result = TMFirstFile(session_handle,state_buffer,&fentry);

if (result < 0)
{
    /* FILE_OPERATIONS error */
    ...
}
else
{
    /* success, first file is in fentry */
    sprintf(buf,"First file is %.4s.%03d",
                        fentry.name,fentry.extension);
    ...
}

/* close the session with a call to TMEndSession */
...
```

TMNextFile

This API call finds the next file in the current device on the MicroLan specified by 'session_handle' and copies information about the file into the buffer 'fentry'. The information is in the form of a structure. The organization of the structure is described in 'TMFirstFile'.

Microsoft Windows TMEX DLL function prototype:

short far pascal TMNextFile(long session_handle, void far *state_buffer,
FileEntry far *fentry);

This function returns:
1 => if the next file was found, data is in buffer 'fentry'
0 => if there are no more files in the directory
<0 => a FILE_OPERATIONS error has occurred

DOS TMEX TSR calling procedure:

Prior to function call:
- Upper nibble of AH is MicroLan number.
- Lower nibble of AH is 02 hex.

Call a File_Operations layer TMEX interrupt (60-66 hex) with 'TMEX' at the beginning of the ID string. See APPENDIX A for details.

Upon return from function:
- Carry is set if there is any error in execution.
- AL is 01 hex if a next file was found or 00 hex if one was not. AL is a FILE_OPERATIONS error code if the carry is set.
- ES:BX is a far pointer (Segment:Offset) to the FileEntry structure of the file found. See the FileEntry structure in TMFirstFile.

The TMEXLIB.C function prototype is:

short far pascal TMNextFile(short port, FileEntry far *fentry);

The function returns the same as the TMEX DLL call.

Code Example for TMEX Microsoft Windows call:

```
long session_handle;
unsigned char state_buffer[15360];
FileEntry fentry;
char buf[100];
short result;

/* session_handle set from a call to TMExtendedStartSession */
...

/* get the unique registration number of the device to communicate with
using TMFirst,TMNext,TMRom... */
...

/* loop to get all of the files in the current directory */
/* get the first file */
result = TMFirstFile(session_handle,state_buffer,&fentry)

/* loop while files are being found */
while(result > 0)
{
   /* get the file information */
   sprintf(buf,"File is %.4s.%03d",
                        fentry.name,fentry.extension);
   ...

   /* get the next file */
   result = TMNextFile(session_handle,state_buffer,&fentry);
}

/* close the session with a call to TMEndSession */
...
```

TMOpenFile

This API call opens a file for a future call to one of several file API functions. The ‘file_handle’ returned from this function can be used with the TMReadFile and TMWriteAddFile API calls. The filename to open is specified in 'fentry' which is in the structure FileEntry as described in TMFirstFile. Only the 'name' and 'extension' portions of FileEntry need to be set before calling TMOpenFile. The file must be in the current directory on the current device on the MicroLan specified by the session handle 'session_handle'.

Microsoft Windows TMEX DLL function prototype:

short far pascal TMOpenFile(long session_handle, void far *state_buffer, FileEntry far *fentry);

This function returns:
>= 0 => file found, and this is the file handle
< 0 => a FILE_OPERATIONS error has occurred

DOS TMEX TSR calling procedure:

Prior to function call:
- Upper nibble of AH is MicroLan number.
- Lower nibble of AH is 03 hex.
- ES:BX is a far pointer (Segment:Offset) to the name of the file to be opened. See the FileEntry structure in TMFirstFile.

Call a File_Operations layer TMEX interrupt (60-66 hex) with 'TMEX' at the beginning of the ID string. See APPENDIX A for details.

Upon return from function:
- Carry is set if there is any error.
- AL is the file handle number or a FILE_OPERATIONS error code if the carry is set.

The TMEXLIB.C function prototype is:

short far pascal TMOpenFile(short port, FileEntry far *fentry);

The function returns the same as the TMEX DLL call.

Code Example for TMEX Microsoft Windows call:

```
long session_handle;
unsigned char state_buffer[15360];
FileEntry fentry;
short file_handle;
short result;
```

```
/* session handle set from a call to TMExtendedStartSession */
...

/* get the unique registration number of the device to communicate with
using TMFirst,TMNext,TMRom... */
...

/* get the first file in the current directory */
result = TMFirstFile(session_handle,state_buffer,&fentry);

/* make sure a file was found that was not a directory reference */
if ((result > 0) && (fentry.extension != 0x7F))
{
   /* try and open the file that was just found */
   file_handle = TMOpenFile(session_handle,state_buffer,&fentry);
   if (file_handle >= 0)
   {
       /* file is opened, ready to read using file_handle */
       ...
   }
   else
   {
       /* error getting file handle */
       ...
   }
}
else
{
    /* error getting first file or file entry is a directory */
    ...
}

OR

fentry.name[0] = 'D';
fentry.name[1] = 'E';
fentry.name[2] = 'M';
fentry.name[3] = 'O';
fentry.extension = 0;

/* try to open the file DEMO.000 */
file_handle = TMOpenFile(session_handle,state_buffer,&fentry);

if (file_handle >= 0)
{
   /* file is opened, ready to read */
   ...
}
else
{
   /* error opening file */
```

```
}

/* close the opened file with a call to TMCloseFile */
...

/* close the session with a call to TMEndSession */
...
```

TMCreateFile

This API function creates a new file name in the current directory on the current device on the MicroLan specified with the session 'session_handle'. The new file name is provided in the FileEntry structure 'fentry'. Only the 'name' and 'extension' portions of FileEntry need to be set before calling TMCreateFile. Repeat file names are not permitted. This API returns a file handle >= 0. The API call also returns the estimated maximum number of bytes available for this new file 'maxwrite'. The file name must be left justified padded with spaces. The valid file name characters are (ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789!#$%&'-@^_`{}~). The valid value of the extension is 0-99 decimal for normal files, 100 for an ‘AddFile’ and 101 for a ‘Monetary File’. The ‘AddFile’ is described in the API function description of TMWriteAddFile. The ‘Monetary File’ is a file that is only created on a DS1962, DS1963, DS2422 and DS2423. The data page of this file can only be located on a page that has a corresponding counter. If a page with a counter is not available on the device then the ‘Monetary File’ will not be created. In case of the ‘Monetary File’, some of its other values are set upon returning from this function. In particular, the page number where the file is copied to is set in the ‘start page’ value of the FileEntry structure. Also, the Counter and Tamper bytes are written into the first eight bytes of the ‘bitmap’ section of the FileEntry structure. Note that the Counter value reported is the current Counter value. If the counter is a page write cycle counter as in the DS1963, then 1 must be added to the counter value to reflect the upcoming copy.

Microsoft Windows TMEX DLL function prototype:

short far pascal TMCreateFile(long session_handle, void far *state_buffer,
short far *maxwrite, FileEntry far *fentry);

This function returns:
>= 0 => file created, and this is the file handle
< 0 => a FILE_OPERATIONS error has occurred

DOS TMEX TSR calling procedure:

Prior to function call:
- Upper nibble of AH is MicroLan number.
- Lower nibble of AH is 04 hex.
- ES:BX is a far pointer (Segment:Offset) to the name of the file to be created. See the FileEntry structure in TMFirstFile.

Call a File_Operations layer TMEX interrupt (60-66 hex) with 'TMEX' at the beginning of the ID string. See APPENDIX A for details.

Upon return from function:
- Carry is set if there is any error in execution.
- AL is the file handle number or a FILE_OPERATIONS error code if the carry is set.
- CX is the estimated maximum number of bytes available for a file.

The TMEXLIB.C function prototype is:
short far pascal TMCreateFile(short port, short far *maxwrite, FileEntry far *fentry);

The function returns the same as the TMEX DLL call.

Code Example for TMEX Microsoft Windows call:

```
long session_handle;
unsigned char state_buffer[15360];
short file_handle, maxwrite;
FileEntry fentry;

/* session_handle set from a call to TMExtendedStartSession */
...

/* get the unique registration number of the device to communicate with
using TMFirst,TMNext,TMRom... */
...

fentry.name[0] = 'D';
fentry.name[1] = 'E';
fentry.name[2] = 'M';
fentry.name[3] = 'O';
fentry.extension = 63;

/* try to create the file DEMO.063 */
file_handle = TMCreateFile(session_handle,state_buffer,&maxwrite,
                &fentry);

if (file_handle >= 0)
{
   /* file is created, ready to write using file_handle */
   ...
}
else
{
   /* error creating file */
   ...
}
```

```
/* close the opened file with a call to TMCloseFile */
...

/* close the session with a call to TMEndSession */
...
```

TMCloseFile

This API call closes the file specified by the file handle 'file_handle'. All files that have been opened for reading with TMOpenFile or created for writing with TMCreateFile must be closed after use.

Microsoft Windows TMEX DLL function prototype:

short far pascal TMCloseFile(long session_handle, void far *state_buffer, short file_handle);

The function returns:
1 => file closed
< 0 => a FILE_OPERATIONS error has occurred

DOS TMEX TSR calling procedure:

Prior to function call:
- Upper nibble of AH is MicroLan number.
- Lower nibble of AH is 05 hex.
- AL (lower nibble) is the file handle to be closed.

Call a File_Operations layer TMEX interrupt (60-66 hex) with 'TMEX' at the beginning of the ID string. See APPENDIX A for details.

Upon return from function:
- Carry is set if there is any error in execution.
- AL is a FILE_OPERATIONS error code if the carry is set.

The TMEXLIB.C function prototype is:

short far pascal TMCloseFile(short port, short file_handle);

The function returns the same as the TMEX DLL call.

Code Example for TMEX Microsoft Windows call:

```
long session_handle;
unsigned char state_buffer[15360];
short file_handle,result;
```

```
/* session handle set from a call to TMExtendedStartSession */
...

/* get the unique registration number of the device to communicate with
using TMFirst,TMNext,TMRom... */
...

fentry.name[0] = 'D';
fentry.name[1] = 'E';
fentry.name[2] = 'M';
fentry.name[3] = 'O';
fentry.extension = 0;

/* try to open the file DEMO.000 */
file_handle = TMOpenFile(session_handle,state_buffer,&fentry);

if (file_handle >= 0)
{
   /* file is opened, ready to read */
   ...

   /* done with file so close it */
   result = TMCloseFile(session_handle,state_buffer,file_handle);

   if (result < 0)
   {
      /* error closing file */
      ...
   }
}
else
{
   /* error opening file */
   ...
}

/* close the session with a call to TMEndSession */
...
```

TMReadFile

This API call reads from the file specified by the file handle number 'file_handle' and copies it into the buffer 'read_buffer', not to exceed 'max_read' bytes. 'file_handle' must be set as a result of a call to TMOpenFile. If the file to read is an ‘add’ file with extension 100 decimal then the file length will be in page increments. For example if the last data byte is in the 30th byte of the file, the length will go to the end of the page which would give a length of (2*28) or 56 bytes. If the file to read is a ‘Monetary File ‘ with extension 101, the length of the buffer is at least 36 bytes long (28 bytes for the ‘Monetary File’ + 8bytes for the Counter and Tamper bytes to append).

Microsoft Windows TMEX DLL function prototype:

short far pascal TMReadFile(long session_handle, void far *state_buffer, short file_handle, unsigned char far *read_buffer, short max_read);

The function returns:
>= 0 => file read, and this is the number of bytes
< 0 => a FILE_OPERATIONS error has occurred

DOS TMEX TSR calling procedure:

Prior to function call:
- Upper nibble of AH is MicroLan number.
- Lower nibble of AH is 06 hex.
- AL (lower nibble) is the file handle to be read.
- CX is the maximum number of bytes to read.
- ES:BX is a far pointer (Segment:Offset) to an empty buffer for the data to be read.

Call a File_Operations layer TMEX interrupt (60-66 hex) with 'TMEX' at the beginning of the ID string. See APPENDIX A for details.

Upon return from function:
- Carry is set if there is any error in execution.
- AL is a FILE_OPERATIONS error code if the carry is set.
- CX is the number of bytes read.

The TMEXLIB.C function prototype is:

short far pascal TMReadFile(short port, short file_handle, unsigned char far *read_buffer, short max_read);

The function returns the same as the TMEX DLL call.

Code Example for TMEX Microsoft Windows call:

```
long session_handle;
unsigned char state_buffer[15360];
unsigned char read_buffer[500];
short len, file_handle;

/* session_handle set from a call to TMExtendedStartSession */
...

/* get the unique registration number of the device to communicate with
using TMFirst,TMNext,TMRom... */
...
```

```
/* file handle set from call to TMOpenFile */
...

len = TMReadFile(session_handle, state_buffer, file_handle, read_buffer,
500);

if (len >= 0)
{
   /* file read with length 'len' */
   ...
}
else
{
   /* error reading the file */
   ...
}

/* close the opened file with a call to TMCloseFile */
...

/* close the session with a call to TMEndSession */
...
```

TMWriteFile

This API call writes the file specified by the file handle 'file_handle' with the data from the buffer 'write_buffer', containing 'num_write' bytes. 'file_handle' must be set as a result of a call to TMCreateFile. This API call works only with regular files with extensions 0-99 decimal and the Monetary File with extension 101. For ‘add’ files with extension 100 decimal, use the API call TMWriteAddFile. In case of the ‘Monetary File’, multiple retries will NOT be attempted. If the file was written many times, then the counter would not end up where it was expected. If this function fails then the file must be closed and then recreated with TMCreateFile. If the Counter has changed then the file needs to be re-encrypted.

Microsoft Windows TMEX DLL function prototype:

short far pascal TMWriteFile(long session_handle, void far *state_buffer, short file_handle, unsigned char far *write_buffer, short num_write);

The function returns:
>= 0 => file written, and this is the number of bytes
< 0 => a FILE_OPERATIONS error has occurred

DOS TMEX TSR calling procedure:

Prior to function call:
- Upper nibble of AH is MicroLan number.
- Lower nibble of AH is 07 hex.

- AL (lower nibble) is the file handle to write.
- CX is the number of bytes to write.
- ES:BX is a far pointer (Segment:Offset) to the buffer containing the data to be written.

Call a File_Operations layer TMEX interrupt (60-66 hex) with 'TMEX' at the beginning of the ID string. See APPENDIX A for details.

Upon return from function:
- Carry is set if there is any error in execution.
- AL is a FILE_OPERATIONS error code if the carry is set.
- CX is the number of bytes written.

The TMEXLIB.C function prototype is:

short far pascal TMWriteFile(short port, short file_handle, unsigned char far *write_buffer, short numread);

The function returns the same as the TMEX DLL call.

Code Example for TMEX Microsoft Windows call:

```
long session_handle;
unsigned char state_buffer[15360];
unsigned char write_buffer[500];
short len,file_handle;

/* session_handle set from a call to TMExtendedStartSession */
...

/* get the unique registration number of the device to communicate with
using TMFirst,TMNext,TMRom... */
...

/* file_handle set from call to TMCreateFile */
...

/* create the message to write to 'file_handle' */
sprintf(write_buffer,"This is a test");

/* write the file */
len = TMWriteFile(session_handle, state_buffer, file_handle,
write_buffer, strlen(write_buffer));

if (len > 0)
{
   /* file was written */
   ...
}
else
```

```
r
   /* error writing file */
   ...
}

/* close the opened file with a call to TMCloseFile */
...

/* close the session with a call to TMEndSession */
...
```

TMDeleteFile

This API call deletes the file specified in the FileEntry structure 'fentry'. Only the 'name' and 'extension' portions of 'fentry' need be set before calling TMDeleteFile. The file must be in the current directory on the current device on the MicroLan specified by the session handle 'session_handle'.

Microsoft Windows TMEX DLL function prototype:

short far pascal TMDeleteFile(long session_handle, void far *state_buffer, FileEntry far *fentry);

The function returns:
1 => file successfully deleted
< 0 => a FILE_OPERATIONS error has occurred

DOS TMEX TSR calling procedure:

Prior to function call:
- Upper nibble of AH is MicroLan number.
- Lower nibble of AH is 08 hex.
- ES:BX is a far pointer (Segment:Offset) to the name of the file to be Deleted. See the FileEntry structure in TMFirstFile.

Call a File_Operations layer TMEX interrupt (60-66 hex) with 'TMEX' at the beginning of the ID string. See APPENDIX A for details.

Upon return from function:
- Carry is set if there is any error in execution.
- AL is the FILE_OPERATIONS error code if the carry is set.

The TMEXLIB.C function prototype is:

short far pascal TMDeleteFile(short port, FileEntry far *fentry);

The function returns the same as the TMEX DLL call.

Code Example for TMEX Microsoft Windows call:

```
long session_handle;
unsigned char state_buffer[15360];
FileEntry fentry;
short result;

/* session_handle set from a call to TMExtendedStartSession */
...

/* get the unique registration number of the device to communicate with
using TMFirst,TMNext,TMRom... */
...

fentry.name[0] = 'D';
fentry.name[1] = 'E';
fentry.name[2] = 'M';
fentry.name[3] = 'O';
fentry.extension = 0;

/* delete the file DEMO.000 */
result = TMDeleteFile(session_handle, state_buffer, &fentry);

if (result >= 0)
{
   /* file is deleted */
   ...
}
else
{
   /* error deleting file */
   ...
}

/* close the session with a call to TMEndSession */
...
```

TMFormat

This API call writes an empty directory into the current device on the MicroLan specified by the session handle 'session_handle'.

Microsoft Windows TMEX DLL function prototype:

short far pascal TMFormat(long session_handle, void far *state_buffer);

The function returns:
1 => iButton successfully formatted
<0 => a FILE_OPERATIONS error has occurred

DOS TMEX TSR calling procedure:

Prior to function call:
- Upper nibble of AH is MicroLan number.
- Lower nibble of AH is 09 hex.

Call a File_Operations layer TMEX interrupt (60-66 hex) with 'TMEX' at the beginning of the ID string. See APPENDIX A for details.

Upon return from function:
- Carry is set if there is any error in execution.
- AL is the FILE_OPERATIONS error code if the carry is set.

The TMEXLIB.C function prototype is:

short far pascal TMFormat(short port);

The function returns the same as the TMEX DLL call.

Code Example for TMEX Microsoft Windows call:

```
long session_handle;
unsigned char state_buffer[15360];
FileEntry fentry;
short result;

/* session_handle set from a call to TMExtendedStartSession */
...

/* get the unique registration number of the device to communicate with
using TMFirst,TMNext,TMRom... */
...

/* format the device found */
result = TMFormat(session_handle, state_buffer);

if (result >= 0)
{
   /* device formatted */
   ...
}
else
{
   /* error formatting the device */
   ...
}

/* close the session with a call to TMEndSession */
...
```

TMAttribute

This API call changes the attributes of the file or directory 'fentry' to the attributes 'attribute'. Only the 'name' and 'extension' portions of 'fentry' need be set before calling TMAttribute. See API TMFirstFile for a description of the ‘FileEntry’ type. 'attribute' contains bits that are set to apply an attribute and cleared to remove an attribute. 'bit 0' represents the least significant bit in 'attribute'. Here are the attributes for TMEX 3.10:

bit 0 : set to make file read-only, for files only
bit 1 : set to make directory hidden, for directories only
bit 2-7 : reserved for future attributes

Microsoft Windows TMEX DLL function prototype:

short far pascal TMAttribute(long session_handle, void far *state_buffer, short attribute, FileEntry far *fentry);

The function returns:
1 => file attribute changed to 'attribute'
< 0 => a FILE_OPERATIONS error has occurred

DOS TMEX TSR calling procedure:

Prior to function call:
- Upper nibble of AH is MicroLan number.
- Lower nibble of AH is 0A hex.
- AL is the file attribute. 01 hex for write protect file, 02 hex for hidden directory and 00 hex for remove attribute.
- ES:BX is a far pointer (Segment:Offset) to the name of the file. See the FileEntry structure above.

Call a File_Operations layer TMEX interrupt (60-66 hex) with 'TMEX' at the beginning of the ID string. See APPENDIX A for details.

Upon return from function:
- Carry is set if there is any error in execution.
- AL is the error code if the carry is set.

The TMEXLIB.C function prototype is:

short far pascal TMAttribute(short port, short attribute, FileEntry far *fentry);

The function returns the same as the TMEX DLL call.

Code Example for TMEX Microsoft Windows call:

```
long session_handle;
unsigned char state_buffer[15360];
FileEntry fentry;
```

```
short result;

/* session_handle set from a call to TMExtendedStartSession */
...

/* get the unique registration number of the device to communicate with
using TMFirst,TMNext,TMRom... */
...

fentry.name[0] = 'D';
fentry.name[1] = 'E';
fentry.name[2] = 'M';
fentry.name[3] = 'O';
fentry.extension = 0;

/* make file DEMO.000 read-only */
result = TMAttribute(session_handle, state_buffer, 0x01, &fentry);

if (result < 0)
{
   /* error setting file attribute */
   ...
}

fentry.name[0] = 'S';
fentry.name[1] = 'U';
fentry.name[2] = 'B';
fentry.name[3] = ' ';
fentry.extension = 0x7F;

/* make sub-directory SUB hidden */
result = TMAttribute(session_handle, state_buffer, 0x02, &fentry);

if (result < 0)
{
   /* error setting attribute on sub-directory */
   ...
}

/* close the session with a call to TMEndSession */
...
```

TMReNameFile

This API call changes the name of a previously opened file specified by the 'file_handle'. The new name is given in the FileEntry structure 'fentry'. Only the 'name' and 'extension' portions of 'fentry' need be set before calling TMReNameFile. 'file_handle' must be set as a result of a call to TMOpenFile. Note that sub-directory names can not be changed to a file name and vise versa.

Microsoft Windows TMEX DLL function prototype:

short far pascal TMReNameFile(long session_handle, void far *state_buffer, short file_handle, FileEntry far *fentry);

The function returns:
1 => file name changed to new name
< 0 => a FILE_OPERATIONS error has occurred

DOS TMEX TSR calling procedure:

Prior to function call:
- Upper nibble of AH is MicroLan number.
- Lower nibble of AH is 0B hex.
- AL (lower nibble) is the handle of the file to be re-named.
- ES:BX is a far pointer (Segment:Offset) to the new name of the file. See the FileEntry structure in TMFirstFile.

Call a File_Operations layer TMEX interrupt (60-66 hex) with 'TMEX' at the beginning of the ID string. See APPENDIX A for details.

Upon return from function:
- Carry is set if there is any error in execution.
- AL is a FILE_OPERATIONS error code if the carry is set.

The TMEXLIB.C function prototype is:

short far pascal TMReNameFile(short port, short file_handle, FileEntry far *fentry);

The function returns the same as the TMEX DLL call.

Code Example for TMEX Microsoft Windows call:

```
long session_handle;
unsigned char state_buffer[15360];
FileEntry fentry;
short file_handle,result;

/* session_handle set from a call to TMExtendedStartSession */
...

/* get the unique registration number of the device to communicate with
using TMFirst,TMNext,TMRom... */
...

fentry.name[0] = 'D';
fentry.name[1] = 'E';
fentry.name[2] = 'M';
```

```
fentry.name[3] = 'O';
fentry.extension = 0;

/* rename DEMO.000 to BACK.014 */
file_handle = TMOpenFile(session_handle, state_buffer, &fentry);

if (file_handle >= 0)
{
   /* file opened correctly */

   fentry.name[0] = 'B';
   fentry.name[1] = 'A';
   fentry.name[2] = 'C';
   fentry.name[3] = 'K';
   fentry.extension = 14;

   result = TMReNameFile(session_handle, state_buffer,
                         file_handle, &fentry);

   if (result == 1)
   {
      /* file name changed */
      ...
   }
   else
   {
      /* error renaming the file */
      ...
   }
}
else
{
   /* error opening the file */
   ...
}

/* close the opened file with a call to TMCloseFile */
...

/* close the session with a call to TMEndSession */
...
```

TMChangeDirectory

This API call sets or reads the current working directory. There is an 'operation' flag that indicates if the call is to set or read the current directory. Here are the values of 'operation':

0 : current directory is set

1 : current directory is read

The value in 'cd_buf' is a DirectoryPath structure that either contains the directory to be set or the current directory read depending on the 'operation' flag. 'cd_buf' must be the following DirectoryPath structure.

```
typedef struct
{
   unsigned char NumEntries;       /* number of entries in structure */
   char Ref;                       /* directory reference character */
   char Entries[10][4];            /* entry items */
} DirectoryPath;
```

This is a packed structure on 1 byte boundaries. The DirectoryPath structure is modeled after the way in which working directory paths are written and read using the 'CD' DOS command. For example when typing 'CD' in a sub-directory in DOS you may get this:

CD<ENTER>
C:\WORK\C\NEW

The string that was given by 'CD' has a depth of 3 sub-directories referenced from the root. If this information was to be returned in the DirectoryPath structure then:

NumEntries = 3 (there are 3 entries, one for each sub-directory)
Ref = \ (reference character is from the root \ )
Entries[0] = "WORK" (first sub-directory from root)
Entries[1] = "C " (second sub-directory from root)
Entries[2] = "NEW " (third sub-directory from root)

This analogy continues to work for setting the working directory with the DOS 'CD' command. For example:

CD \MAIL\WK3\TUES\MORN <ENTER>

This command would set the current directory to '\MAIL\WK3\TUES\MORN' as referenced from the root. To do this with the DirectoryPath structure, first set the 'Operation' flag to 0 to set the current directory and fill out the value as:

NumEntries = 4
Ref = \
Entries[0] = "MAIL"
Entries[1] = "WK3 "
Entries[2] = "TUES"
Entries[3] = "MORN"

If the current directory is '\MAIL\WK3\TUES\MORN' and the desired directory is '\MAIL\WK3\WED' there is a short cut using the 'CD':

CD .\..\WED <ENTER>

This syntax of this command says that from the current directory '.' go to the previous directory '..' and then go to the sub-directory 'WED'. This short cut can also be done with the DirectoryPath structure:

NumEntries = 2 (there are 2 entries)
Ref = . (reference character is from the current directory .)
Entries[0] = ".. " (special case entry that goes to previous directory)
Entries[1] = "WED " (sub-directory entry from current)

Note that the 'Entries' are left justified and padded with spaces to 4 characters. The sub-directory depth is limited to 10 deep. If the 'NumEntries' is set to or is read to be 0 then the current directory is at the ROOT. Here are the allowable values for all of the components of the DirectoryPath structure:

NumEntries: 0-10

Ref: \ (reference from ROOT directory (set or read))
. (reference from current directory (set only))

Entries: (left justified padded with spaces with 4 characters)

(special case Entries)
". " (reference entry to the current directory)
".. " (back reference entry to the previous directory)

(valid characters)
ABCDEFGHIJKLMNOPQRSTUVWXYZ
0123456789
!#$%&'-@^_`{}~

Microsoft Windows TMEX DLL function prototype:

short far pascal TMChangeDirectory(long session_handle, void far *state_buffer, short operation, DirectoryPath far *cd_buf);

The function returns:
1 => Operation is successful
<0 => a FILE_OPERATIONS error has occurred

DOS TMEX TSR calling procedure:

Prior to function call:
- Upper nibble of AH is the MicroLan number.
- Lower nibble of AH is the function code 0C hex.
- Lower nibble of AL is 1 for read current directory or 0 for set current directory.
- ES:BX is a far pointer (Segment:Offset) to the DirectoryPath structure.

Call a File_Operations layer TMEX interrupt (60-66 hex) with 'TMEX' at the beginning of the ID string. See APPENDIX A for details.

Upon Return from function:
- Carry is set if there is any error in execution.
- AL is the FILE_OPERATIONS error code if carry is set.
- ES:BX is a far pointer (Segment:Offset) to the DirectoryPath structure now filled with the current directory.
- The function was successful if the carry was not set.

The TMEXLIB.C function prototype is:
short far pascal TMChangeDirectory(short port, short operation, DirectoryPath far *cd_buf);

The function returns the same as the TMEX DLL call.

Code Example for TMEX Microsoft Windows call:

```
long session_handle;
unsigned char state_buffer[15360];
short result;
DirectoryPath cd_buf;

/* session_handle set from a call to TMExtendedStartSession */
...

/* get the unique registration number of the device to communicate with
using TMFirst,TMNext,TMRom... */
...

/* set the current directory to \WORK\TMP */
cd_buf.NumEntries = 2;
cd_buf.Ref = '\\';
cd_buf.Entries[0][0] = 'W';
cd_buf.Entries[0][1] = 'O';
cd_buf.Entries[0][2] = 'R';
cd_buf.Entries[0][3] = 'K';
cd_buf.Entries[1][0] = 'T';
cd_buf.Entries[1][1] = 'M';
cd_buf.Entries[1][2] = 'P';
cd_buf.Entries[1][3] = ' ';

result = TMChangeDirectory(session_handle, status_buffer, 0, &cd_buf);

if (result == 1)
{
   /* current directory set */
   ...
}
else
```

```
r
   /* error setting current directory to \WORK\TMP */
   ...
}

/* read the current directory */
result = TMChangeDirectory(session_handle, status_buffer, 1, &cd_buf);

if (result == 1)
{
   /* current directory read, should be \WORK\TMP */
   ...
}
else
{
   /* error reading current directory */
   ...
}

/* close the session with a call to TMEndSession */
...
```

TMDirectoryMR

This API call Makes or Removes a sub-directory from the current directory. There is an 'operation' flag that indicates if the call is to make or remove a sub-directory. Here are the values of 'operation':

0 : make sub-directory specified by FileEntry 'dir_buf'
1 : remove sub-directory specified by FileEntry 'dir_buf'

Only the 'name' field of the FileEntry structure need be filled in before calling TMDirectoryMR. Note that a sub-directory is created empty and only an empty sub-directory can be removed.

Microsoft Windows TMEX DLL function prototype:

short far pascal TMDirectoryMR(long session_handle, void far *state_buffer, short operation, FileEntry far *fentry);

The function returns:
1 => Operation is successful
<0 => a FILE_OPERATIONS error has occurred

DOS TMEX TSR calling procedure:

Prior to function call:
- Upper nibble of AH is the MicroLan number.
- Lower nibble of AH is the function code 0D hex.

- Lower nibble of AL is 0 for make directory or 1 for remove directory.
- ES:BX is a far pointer (Segment:Offset) to the FileEntry structure of the sub-directory to make or remove.

Call a File_Operations layer TMEX interrupt (60-66 hex) with 'TMEX' at the beginning of the ID string. See APPENDIX A for details.

Upon Return from function:
- Carry is set if there is any error in execution.
- AL is the FILE_OPERATIONS error code if carry is set.
- The function was successful if the carry was not set.

The TMEXLIB.C function prototype is:

short far pascal TMDirectoryMR(short port, short operation, FileEntry far *dir_buf);

The function returns the same as the TMEX DLL call.

Code Example for TMEX Microsoft Windows call:

```
long session_handle;
unsigned char state_buffer[15360];
FileEntry dir_buf;
short result;

/* session_handle set from a call to TMExtendedStartSession */
...

/* get the unique registration number of the device to communicate with
using TMFirst,TMNext,TMRom... */
...

dir_buf.name[0] = 'T';
dir_buf.name[1] = 'E';
dir_buf.name[2] = 'M';
dir_buf.name[3] = 'P';

/* make a sub-directory called TEMP */
result = TMDirectoryMR(session_handle, state_buffer, 0, &dir_buf);

if (result == 1)
{
   /* sub-directory TEMP made */
   ...
}
else
{
   /* error making sub-directory */
   ...
```

```
`

/* remove the sub-directory just made */
result = TMDirectoryMR(session_handle, state_buffer, 1, &dir_buf);

if (result == 1)
{
   /* sub-directory TEMP removed */
   ...
}
else
{
   /* error removing sub-directory */
   ...
}

/* close the session with a call to TMEndSession */
...
```

TMCreateProgramJob

This API call starts a program job with the current EPROM device. This function must be called immediately before any TMEX function that writes to an EPROM device. The writes in the TMEX function are not actually done, they are just logged in the program job data space. After the TMEX function has been completed successfully, then another specialty function TMDoProgramJob must be called to carry out the EPROM programming. EPROMs are treated differently by TMEX because any write to an EPROM is destructive, meaning it uses up device memory. In NV-RAM devices, if an operation is unsuccessful then the function could then be just called again with no ill effects. If EPROMs were regarded in this way then some or all of the memory space could be lost. The second specialty API call, TMDoProgramJob, is called again and again by the application program until all of the jobs are completed. This puts the onus on the application programmer to make sure that the EPROM device does not go away in mid write and waste memory space. For instance if after doing a TMCreateProgramJob and TMWriteFile successfully, the TMDoProgramJob comes back saying that the device has left the MicroLan then the application should put up some message saying that the device needs to come back and call TMDoProgramJob again. The data space needed to write EPROMs is located in an internal buffer in the DOS TSR and at the end of the status buffer provided in the Windows functions. There can be only one program job at a time. If after TMCreateProgramJob is called, the internal ROM buffer is changed then the program job is invalidated. See the 'TMEX APPLICATION DESIGN GUIDE' section for more discussion of TMCreateProgramJob.

Microsoft Windows TMEX DLL function prototype:

short far pascal TMCreateProgramJob(long session_handle, void far *state_buffer);

The function returns:
1 => Program Job is ready

<0 => a FILE_OPERATIONS error has occurred

DOS TMEX TSR calling procedure:

Prior to function call:
- Upper nibble of AH is the MicroLan number.
- Lower nibble of AH is the main function code 0E hex.
- AL is the sub function code 01 hex to indicate the TMCreateProgramJob function.

Call a File_Operations layer TMEX interrupt (60-66 hex) with 'TMEX' at the beginning of the ID string. See APPENDIX A for details.

Upon Return from function:
- Carry is set if there is any error in execution.
- AL is the error code if carry is set.
- The function was successful if the carry was not set.

The TMEXLIB.C function prototype is:

short far pascal TMCreateProgramJob(short port);

The function returns the same as the TMEX DLL call.

Code Example for TMEX Microsoft Windows call:

```
long session_handle;
unsigned char state_buffer[15360];
short result;

/* session_handle set from a call to TMExtendedStartSession */
...

/* get the unique registration number of the device to communicate with
using TMFirst,TMNext,TMRom... */
...

/* create a program job because the device is an EPROM
   and a write operation is desired */
result = TMCreateProgramJob(session_handle, state_buffer);

if (result == 1)
{
   /* TMProgramJob is create, now ready for a write operation
      and then calls to TMDoProgramJob */
   ...
}
else
{
   /* error creating TMProgramJob */
```

```
}

/* close the session with a call to TMEndSession */
...
```

TMDoProgramJob

This API call writes all of the data that has been written to the EPROM device since the TMCreateProgramJob function was called. The write jobs are done one at a time until complete. This function marks off the jobs it has done so that if writing can not continue because the device has lost contact it can begin where it left off when it is called again. If this function fails and is not called again and allowed to complete, then the target device may be left in an unknown state. EPROMs are treated differently by TMEX because any write to an EPROM is destructive, meaning it uses up device memory. In NV-RAM devices, if an operation is unsuccessful then the function could then be just called again with no ill effects. If EPROMs were regarded in this way then some or all of the memory space could be lost. The TMDoProgramJob function is called again and again by the application program until all of the jobs are completed. This puts the onus on the application programmer to make sure that the EPROM device does not go away in mid write and waste memory space and leave the device in an unknown state. If the ROM is changed at any time before TMDoProgramJob is finished then the program job becomes invalid.

Microsoft Windows TMEX DLL function prototype:

short far pascal TMDoProgramJob(long session_handle, void far *state_buffer);

The function returns:
1 => Program Job completed
<0 => a FILE_OPERATIONS error has occurred, correct and call again

DOS TMEX TSR calling procedure:

Prior to function call:
- Upper nibble of AH is the MicroLan number.
- Lower nibble of AH is the main function code 0E hex.
- AL is the sub function code 02 hex to indicate the TMDoProgramJob function.

Call a File_Operations layer TMEX interrupt (60-66 hex) with 'TMEX' at the beginning of the ID string. See APPENDIX A for details.

Upon Return from function:
- Carry is set if there is any error in execution.
- AL is the error code if carry is set.
- The function was successful if the carry was not set.

The TMEXLIB.C function prototype is:

short far pascal TMDoProgramJob(short port);

The function returns the same as the TMEX DLL call.

Code Example for TMEX Microsoft Windows call:

```
long session_handle;
unsigned char state_buffer[15360];
short result;

/* session_handle set from a call to TMExtendedStartSession */
...

/* get the unique registration number of the device to communicate with
using TMFirst,TMNext,TMRom... */
...

/* TMCreateProgram job called and a successful write operation has been
done */
...

do
{
   result = TMDoProgramJob(session_handle, state_buffer);

   if (result < 0)
   {
      /* error doing Program Job, prompt user to correct */
      ...
   }
}
while (result != 1);

/* close the session with a call to TMEndSession */
...
```

TMWriteAddFile

This API adds to an opened 'add' file specified by the file handle 'file_handle'. The 'file_handle' can be acquired with TMCreateFile for a new file or TMOpenFile for an existing 'add' file. An 'add' file is a file with extension 100 (decimal) that resides on an EPROM iButton device. The 'add' file is special because the contents of the file can be added to without deleting and then rewriting it. The TMWriteAddFile API call has two modes of operation. It can append data to the end of the 'add' file or it can add data at an offset from the beginning of the file. The 'operation' flag is:

1 : Append to the file starting after the last programmed byte (not FF Hex) in the last page of the file.

0 : Write the data starting at the specified 'offset'. If there is data already there then the result will be to program any zeroes in the data. For instance:

```
00110101 current byte in device
01010111 data byte to program
---------
00010101 result
```

Note that with this feature, individual bits can be programmed.

If the operation is 1 'append' then the 'offset' entry is not used. The data is provided in the 'write_buffer' and the length of the data to write is 'num_write' bytes.

Microsoft Windows TMEX DLL function prototype:

```
short far pascal TMWriteAddFile(long session_handle, void far *state_buffer,
    short file_handle, short operation, short offset, unsigned char far
    *write_buffer, short num_write);
```

The function returns:
>= 0 => file written, and this is the number of bytes
< 0 => a FILE_OPERATIONS error has occurred

DOS TMEX TSR calling procedure:

Prior to function call:
- Upper nibble of AH is the MicroLan number.
- Lower nibble of AH is the main function code 0E hex.
- AL is the sub function code 03 hex to indicate the TMWriteAddFile function.
- ES:BX is a far pointer (Segment:Offset) to a buffer to hold that data to write
- The 14 least significant bits of CX is the number of bytes to write. The 2 most significant bits is the file handle number.
- The 14 least significant bits of DX is the offset to write data at (only if not appending). Bit 14 is a flag to indicate if the operation is to append(1) or writ at an offset(0). Bit 15 is not used and should be set to zero.

Call a File_Operations layer TMEX interrupt (60-66 hex) with 'TMEX' at the beginning of the ID string. See APPENDIX A for details.

Upon Return from function:
- Carry is set if there is any error in execution.
- AL is the error code if carry is set.
- The function was successful if the carry was not set.
- CX is the number of bytes written.

The TMEXLIB.C function prototype is:

```
short far pascal TMWriteAddFile(short port, short file_handle, short operation,
    short offset, unsigned char far *write_buffer, short num_write);
```

The function returns the same as the TMEX DLL call.

Code Example for TMEX Microsoft Windows call:

```
long session_handle;
unsigned char state_buffer[15360];
unsigned char write_buffer[500];
short len, file_handle;

/* session_handle set from a call to TMExtendedStartSession */
...

/* get the unique registration number of the device to communicate with
using TMFirst,TMNext,TMRom... */
...

/* file_handle set from call to TMOpenFile */
...

/* create the message to append to the 'add' file */
sprintf(write_buffer,"This is a test");

/* append to the file */
len = TMWriteAddFile(session_handle, state_buffer, file_handle, 1, 0,
write_buffer, strlen(write_buffer));

if (len > 0)
{
   /* file was written */
   ...
}
else
{
   /* error writing file */
   ...
}

OR

/* clear bit 0 of the 10th byte in the 'add' file specified by
'file_handle */

write_buffer[0] = 0xFE;

/* clear the bit */
len = TMWriteAddFile(session_handle, state_buffer, file_handle, 0, 10,
write_buffer, 1);

if (len > 0)
{
   /* bit was cleared */
```

```
}
else
{
   /* error writing to the 'add' file */
   ...
}

/* close the opened file with a call to TMCloseFile */
...

/* close the session with a call to TMEndSession */
...
```

TMTerminateAddFile

This API call writes the length and CRC bytes into an 'add' type file to terminate it. Doing this prevents further change to the file and it makes the file compatible with the other extended file structure files. The filename to Terminate is specified in 'fentry' which is in the structure FileEntry as described in TMFirstFile. Only the 'name' and 'extension' portions of FileEntry need to be set before calling TMTerminateAddFile. The file must be in the current directory on the current device on the MicroLan specified by the session handle 'session_handle'.

Microsoft Windows TMEX DLL function prototype:

short far pascal TMTerminateAddFile(long session_handle, void far *state_buffer, FileEntry far *fentry);

The function returns:
1 => 'add' file terminated successfully
<0 => a FILE_OPERATIONS error has occurred

DOS TMEX TSR calling procedure:

Prior to function call:
- Upper nibble of AH is the MicroLan number.
- Lower nibble of AH is the main function code 0E hex.
- AL is the sub function code 04 hex to indicate the TMTerminateAddFile function.
- ES:BX is a far pointer (Segment:Offset) to the name of the file to terminate. See the FileEntry structure in TMFirstFile.

Call a File_Operations layer TMEX interrupt (60-66 hex) with 'TMEX' at the beginning of the ID string. See APPENDIX A for details.

Upon Return from function:
- Carry is set if there is any error in execution.
- AL is the error code if carry is set.

- The function was successful if the carry was not set.

The TMEXLIB.C function prototype is:

short far pascal TMTerminateAddFile(short port, FileEntry far *fentry);

The function returns the same as the TMEX DLL call.

Code Example for TMEX Microsoft Windows call:

```
long session_handle;
unsigned char state_buffer[15360];
FileEntry fentry;
short result;

/* session_handle set from a call to TMExtendedStartSession */
...

/* get the unique registration number of the device to communicate with
using TMFirst,TMNext,TMRom... */
...

/* call TMCreateProgramJob to set up EPROM writing */
...

fentry.name[0] = 'D';
fentry.name[1] = 'E';
fentry.name[2] = 'M';
fentry.name[3] = 'O';
fentry.extension = 100;

/* try to Terminate the 'add' file DEMO.100 */
result = TMTerminateAddFile(session_handle,state_buffer,&fentry);

if (result >= 0)
{
   /* 'add' file is terminated */
   ...
}
else
{
   /* error terminating 'add' file */
   ...
}

/* Call TMDoProgramJob until EPROM programming is complete */
...

/* close the session with a call to TMEndSession */
...
```

Get_Version

This API call gets the main version ID string of the TMEX driver. Make sure that the buffer 'ID_buf' is at least 80 bytes to hold the ID string.

Microsoft Windows TMEX DLL function prototype:

short far pascal Get_Version(char far *ID_buf);

The function returns:
1 => ID string is in ID_buf.
0 => error could not get the ID

DOS TMEX TSR calling procedure:

See the function 'GetVectID' in the 'C' library 'TMEXLIB.C' in the source examples.

Code Example for TMEX Microsoft Windows call:

```
char ID_buf[100];

/* get the ID string of the TMEX DLL being used */
Get_Version(ID_buf);
```

TRANSPORT

The Transport API calls described in this section are provided to enable block level data transfer to iButtons. This level includes TMReadPacket and TMWritePacket that read and write CRC-16 verified blocks of data. When doing normal file oriented communication with iButtons there is no need to call any of the Transport API functions. The TRANSPORT error codes for this section are for the Microsoft Windows and TMEXLIB.C TMEX calls. The DOS TSR calls return the absolute value (non-negative) of these return codes with the carry set:

-1 => PORT_NOT_INITIALIZED - MicroLan not initialized with TMSetup
-2 => PORT_NOT_EXIST - specified MicroLan nonexistent
-3 => NO_SUCH_FUNCTION - function not supported
-4 => ERROR_READ_WRITE - error reading or writing a packet
-5 => BUFFER_TOO_SMALL - packet larger than provided buffer
-6 => DEVICE_TOO_SMALL - not enough room for packet on device
-7 => NO_DEVICE - device not found
-8 => BLOCK_TOO_BIG - block transfer too long
-9 => WRONG_TYPE - wrong type of device for this function
-10 => PAGE_REDIRECTED - the page being read is redirected

-11 => PROGRAM_NOT_POSSIBLE - the device is written in a way that can not be changed
-200=> INVALID_SESSION - session not valid (not applicable in DOS TMEX)
-201=> HS_NOT_FOUND - Hardware_Specific driver not found and is require.

TMReadPacket

This API call reads a Universal Data Packet (UDP) starting on 'start_page' in an iButton. The data is placed into 'read_buffer' up to a maximum of 'max_read' bytes. Returns a read count length greater then or equal to 0 for success or one of the TRANSPORT error values for a failure. See APPENDIX B for a detailed description of the Universal Data Packet.

Microsoft Windows TMEX DLL function prototype:

short far pascal TMReadPacket(long session_handle, void far *state_buffer, short start_page, unsigned char far *read_buffer, short max_read);

The function returns:
>=0 => length of valid data read
<0 => a TRANSPORT error has occurred

DOS TMEX TSR calling procedure:

Prior to function call:
- Upper nibble of AH is MicroLan number.
- Lower nibble of AH is 0A hex.
- AL is the page number that the packet begins on, 0 - 255.
- CX is the maximum number of bytes that can be read into the buffer.
- ES:BX is a far pointer (Segment:Offset) to an empty buffer for the data to be read.

Call a Transport layer TMEX interrupt (60-66 hex) with 'DOW' at the beginning of the ID string. See APPENDIX A for details.

Upon return from function:
- Carry is set if there is an error in execution.
- If carry is set then AL contains the error code.
- CX is the number of bytes read into the buffer.
- There may be diagnostic results in DX. (see text at the beginning of the NETWORK section)

The TMEXLIB.C function prototype is:

short far pascal TMReadPacket(short port, short start_page, unsigned char *read_buffer, short max_read);

The function returns the same as the TMEX DLL call.

Code Example for TMEX Microsoft Windows call:

```
long session_handle;
unsigned char state_buffer[15360], read_buffer[29];
short len;

/* session_handle set from a call to TMExtendedStartSession */
...

/* the internal 8-byte ROM buffer contains a valid ROM that is going to
be selected to do a device specific operation */
...

/* attempt to read a Universal Data Packet on page 1 */
len = TMReadPacket(session_handle, state_buffer, 1, read_buffer, 29);

if (len >= 0)
{
   /* success, a Universal Data Packet was read and
      its length is len */
   ...
}
else
{
   /* TRANSPORT error reading the DDS */
   ...
}

/* close the session with a call to TMEndSession */
...
```

TMWritePacket

This API call writes a Universal Data Packet (UDP) starting on 'start_page' in an iButton. The 'num_write' bytes of data in 'write_buffer' is written. Returns a byte length greater than or equal to 0 for success or one of the TRANSPORT error values for a failure. TMWritePacket works only with the nvram iButtons such as DS1993 See APPENDIX B for a detailed description of the Universal Data Packet.

Microsoft Windows TMEX DLL function prototype:

short far pascal TMWritePacket(long session_handle, void far *state_buffer, short start_page, unsigned char far *write_buffer, short num_write);

The function returns:
>=0 => length of data written
<0 => a TRANSPORT error has occurred

DOS TMEX TSR calling procedure:

Prior to function call:
- Upper nibble of AH is MicroLan number.
- Lower nibble of AH is 0B hex.
- AL is the page number that the packet begins on, 0 to 255.
- CX is the number of bytes to write.
- ES:BX is a far pointer (Segment:Offset) to the buffer of data to write.

Call a Transport layer TMEX interrupt (60-66 hex) with 'DOW' at the beginning of the ID string. See APPENDIX A for details.

Upon return from function:
- Carry is set if there is an error in execution.
- If carry is set then AL contains the TRANSPORT error code.
- CX is the number of bytes written.
- There may be diagnostic results in DX. (see text at the beginning of the NETWORK section)

The TMEXLIB.C function prototype is:

short far pascal TMWritePacket(short port, short start_page, unsigned char *write_buffer, short num_write);

The function returns the same as the TMEX DLL call.

Code Example for TMEX Microsoft Windows call:

```
long session_handle;
unsigned char state_buffer[15360], write_buffer[100];
short len;

/* session_handle set from a call to TMExtendedStartSession */
...

/* the internal 8-byte ROM buffer contains a valid ROM that is going to
be selected to do a device specific operation */
...

/* attempt to write a Universal Data Packet starting at page 0 */
sprintf(write_buffer,"HELLO TOUCH MEMORY");
len = TMWritePacket(session_handle, state_buffer, 0, write_buffer,
strlen(write_buffer));

if (len >= 0)
{
   /* success, a Universal Data Packet was written and
      its length is len */
   ...
}
```

```
else
{
   /* TRANSPORT error writing the DDS */
   ...
}

/* close the session with a call to TMEndSession */
...
```

TMBlockIO

This API call is a general purpose block transfer function. A TMTouchReset is done followed by TMTouchBytes of all of the 'num_tran' bytes in the 'tran_buffer' data buffer. The values returned from the TMTouchBytes are placed back into the 'tran_buffer' data buffer. This call returns a byte length greater than or equal to 0 for success or one of the TRANSPORT error values described for a failure. The maximum number of 'num_tran' is 1023 bytes.

Microsoft Windows TMEX DLL function prototype:

short far pascal TMBlockIO(long session_handle, unsigned char far *tran_buffer, short num_tran);

The function returns:
>=0 => length of data sent and received from MicroLan
<0 => a TRANSPORT error has occurred

DOS TMEX TSR calling procedure:

Prior to function call:
- Upper nibble of AH is MicroLan number.
- Lower nibble of AH is 0C hex.
- CX is the number of bytes to transfer (1023 max).
- ES:BX is a far pointer (Segment:Offset) to the data buffer to read and write.

Call a Transport layer TMEX interrupt (60-66 hex) with 'DOW' at the beginning of the ID string. See APPENDIX A for details.

Upon return from function:
- Carry is set if there is an error in execution.
- If carry is set then AL contains the TRANSPORT error code.
- CX is the number of bytes transferred. (0 if no device found)
- There may be diagnostic results in DX. (see text at the beginning of the NETWORK section)

The TMEXLIB.C function prototype is:

short far pascal TMBlockIO(short port, unsigned char *tran_buffer, short num_tran);

The function returns the same as the TMEX DLL call.

Code Example for TMEX Microsoft Windows call:

```
long session_handle;
unsigned char state_buffer[15360], tran_buffer[100];
short result;

/* session_handle set from a call to TMExtendedStartSession */
...

/* construct a buffer to read the scratchpad of a DS199X */
tran_buf[0] = 0xCC;   /* skip ROM */
tran_buf[1] = 0xAA;   /* read scratchpad */
for (i = 0; i < 35; i++)
   tran_buf[2+i] = 0xFF;  /* area to read address, data */

result = TMBlockIO(session_handle, tran_buffer, 37);

if (result == 37)
{
   /* the contents of the scratchpad are in the buffer
      in location tran_buf[2] to tran_buf[36] */
   ...
}
else
   /* TRANSPORT error */

/* close the session with a call to TMEndSession */
...
```

TMExtendedReadPage

This API call reads a self generated CRC8 or CRC16 verified status or data page from an EPROM iButton. For the current devices the page length of a status page is always 8 bytes and the length of a data page is always 32 bytes for the current devices. Some iButtons have a self generated CRC so that just a single bit can be changed in a page and still be read with CRC verification. This function reads the specified status or data page and puts it in the supplied buffer without the CRC bytes. When reading a data page from a DS1985, D1986, DS1982 or DS2407 the 33rd byte of the provided buffer will be a redirection byte , a zero there indicates that the page is not redirected Note that the ROM pattern for the desired iButton must already be in the internal eight-byte buffer before this function is called. This can be accomplished by directly writing to the internal buffer using the TMRom API call or by use of the network functions TMFirst, TMNext, TMFirstAlarm or TMNextAlarm. This constraint enables this function to be

multi-drop compatible. The page type 'page_type' is as follows:

0 : regular memory page
1 : status memory page

The current devices that have an extended read memory command are DS1982, DS1985, DS1986, and DS2407.

Microsoft Windows TMEX DLL function prototype:

short far pascal TMExtendedReadPage(long session_handle, void far *state_buffer, short page, unsigned char far *read_buffer, short page_type);

The function returns:
>=0 => length of data read
<0 => a TRANSPORT error has occurred

DOS TMEX TSR calling procedure:

Prior to function call:
- Upper nibble of AH is the MicroLan number.
- Lower nibble of AH is the main function code 0D hex.
- AL is the sub function code 03 hex to indicate the TMExtendedReadPage function.
- CL is the page number to read.
- The LSBit of CH indicates if the page is to be read from status memory (LSBit = 1) or from data memory (LSBit = 0). The 7 most significant bits of CH are 0.
- ES:BX is a far pointer (Segment:Offset) to a buffer to hold the data read of length of at least 8 bytes for a status page read and 33 bytes for a data page read for current devices.

Call a Transport layer TMEX interrupt (60-66 hex) with 'DOW' at the beginning of the ID string. See APPENDIX A for details.

Upon Return from function:
- Carry is set if there is any error in execution.
- AL is the TRANSPORT error code if carry is set.
- The function was successful if the carry was not set.
- The data is in the buffer provided by ES:BX upon success. If the page to read was a data page then the 33rd byte is the redirection byte.

The TMEXLIB.C function prototype is:
short far pascal TMExtendedReadPage(short port, short page, unsigned char far *read_buffer, short page_type);

The function returns the same as the TMEX DLL call.

Code Example for TMEX Microsoft Windows call:

```
long session_handle;
unsigned char state_buffer[15360], read_buffer[100];
short len;

/* session_handle set from a call to TMExtendedStartSession */
...

/* the internal 8-byte ROM buffer contains a valid ROM that is going to
be selected to do a device specific operation */
...

/* attempt to read page 0 of the status memory */
len = TMExtendedReadPage(session_handle, state_buffer, 0, read_buffer,
1);

if (len >= 0)
{
   /* success, status page read and its length is len */
   ...
}
else
{
   /* TRANSPORT error reading extended page (did this device have
      extended read?) */
   ...
}

/* close the session with a call to TMEndSession */
...
```

TMProgramByte

This API call writes a byte to an EPROM device DS1982, DS1985, DS1986 or DS2407. The byte to write 'write_byte', the location (address 'address' and status/regular space designator 'page_type') and the number of bits 'bits' to program at a time are specified to the function. If the number of bits to program at a time is not successful then the function will automatically fall back to a lower value. To not waste time on the next call to TMProgramByte the number of bits per pass is returned. If the function is forced to fall back to a lower value then use this value on subsequent calls to the function. Note that the ROM pattern for the desired iButton must already be in the internal eight-byte buffer before this function is called. This can be accomplished by direct writing to the internal buffer using the API call TMRom or by use a network API call TMFirst, TMNext, TMFirstAlarm or TMNextAlarm. This constraint enables this function to be multi-drop compatible with other EPROM devices on the MicroLan. It is the responsibility of the calling program to make sure that there are no non-EPROM devices on the MicroLan at the time of programming. A non-EPROM device can be damaged from the programming pulse. The page type 'page_type' is as follows:

0 : regular memory page

1 : status memory page

The valid values for the number of bits to program at each pass 'bits' is 2,4 or 8. The 'zeros' flag indicates:

0 : 'write_byte' must be written exactly as indicated
1 : only the 0 bits must be programmed correctly (used in individual bit programming application)

Microsoft Windows TMEX DLL function prototype:

short far pascal TMProgramByte(long session_handle, void far *state_buffer, short write_byte, short address, short page_type, short far *bits, short zeros);

The function returns:
1 => byte written successfully
<0 => a TRANSPORT error has occurred

DOS TMEX TSR calling procedure:

Prior to function call:
- Upper nibble of AH is the MicroLan number.
- Lower nibble of AH is the main function code 0D hex.
- AL is the sub function code 04 hex to indicate the TMProgramByte function.
- CX is the address of the byte to write.
- DL is the byte to write.
- DH has three different parts. Bit 0 (the LSBit) and Bit 1 indicate the number of bits to program at a time. In Bit 1, Bit 0 fashion:

00 - 2 bits per program pass
01 - 4 bits per program pass
10 - 8 bits per program pass

Bit 2 is a flag to indicate if the address provided is in the regular data space (0) or is in the status data space (1) of the EPROM. Bit 3 is a flag to indicate if the byte must be exactly the same as indicated (0) or if only the 0 bits must be programmed correctly (1). The remaining bits of DH, Bit 4 through Bit 7 are 0.

Call a Transport layer TMEX interrupt (60-66 hex) with 'DOW' at the beginning of the ID string. See APPENDIX A for details.

Upon Return from function:
- Carry is set if there is any error in execution.
- AL is the TRANSPORT error code if carry is set.
- The function was successful if the carry was not set.
- Bit 0 and Bit 1 of DH are the bits per pass that were successful if the carry was not set.

The TMEXLIB.C function prototype is:

short far pascal TMProgramByte(short port, short write_byte, short address, short page_type, short far *bits, short zeros);

The function returns the same as the TMEX DLL call.

Code Example for TMEX Microsoft Windows call:

```
long session_handle;
unsigned char state_buffer[15360], read_buffer[100];
short result, address, bits, write_byte;

/* session_handle set from a call to TMExtendedStartSession */
...

/* the internal 8-byte ROM buffer contains a valid ROM that is going to
be selected to do a device specific operation */
...

/* clear bit 4 of the 6th byte of page 3 in regular data space in EPROM
*/
address = 3 * 32 + 6 - 1;   /* calculate the address */
bits = 8;                   /* set bits to program on each pass */
write_byte = 0xEF;          /* write byte with bit 4 cleared */

result = TMProgramByte(session_handle, state_buffer, write_byte,
address, 0, &bits, 1);

if (result == 1)
{
   /* success, bit cleared */
   ...
}
else
{
   /* TRANSPORT error clearing bit */
   ...
}

/* close the session with a call to TMEndSession */
...
```

NETWORK

The Network API calls described in this section are provided to support the multi-drop capability of the 1-Wire protocol. They allow the calling program to locate and individually communicate with one of many iButton devices connected in parallel on a MicroLan. The NETWORK error codes for this section are for the Microsoft Windows and TMEXLIB.C TMEX calls. The DOS TSR calls return the absolute value (non-negative) of these return codes with the carry set:

-1 => PORT_NOT_INITIALIZED - Specified MicroLan has not been initialized with a call to the TMSetup function.
-2 => PORT_NOT_EXIST - Specified MicroLan nonexistent.
-3 => NO_SUCH_FUNCTION - Function not supported.
-200=> INVALID_SESSION - session not valid (not applicable in DOS TMEX)
-201=> HS_NOT_FOUND - Hardware_Specific driver not found and is required.

Any (or all) of the NETWORK, TRANSPORT and HARDWARE_SPECIFIC layer TMEX DOS API calls can perform optional diagnostics on the MicroLan. Upon return from the interrupt service, register DH can be examined to determine which tests (if any) were made. If DH contains a non-zero value then DL will contain test results. A value of zero in DH indicates that no tests were performed. If these optional diagnostic tests are implemented then the format of the results is as follows:

Register DH bits indicate which test(s) were performed. A bit is set for each test performed.

Bit 0: Test for short to ground (1-Wire stuck low).
Bit 1: Test for short to Vcc (1-Wire stuck high).
Bit 2: Test for Alarm Interrupt.
Bits 3-7: Reserved for future expansion.

Register DL bits contain the results of tests indicated by DH. A bit is set for each tested condition that was detected.

Bit 0: Fail ground short test (1-Wire is shorted).
Bit 1: Fail Vcc short test (1-Wire is shorted).
Bit 2: Alarm Interrupt condition was detected.
Bits 3-7: Reserved for future expansion.

A special case exists where the 1-Wire interface hardware is not able to explicitly detect the polarity of a short (all that is known is that the interface is somehow non-compliant). In this case the function performing the test will set both bits 0 and 1 of register DH to indicate that a test for shorts was performed and if an error is detected both bits 0 and 1 of register DL will be set. The nature of the test should then be clear to the calling routine since normally only one of those bits in DL would be set after any given call to the function.

Please note that the dynamic nature of iButton contact with the MicroLan will occasionally cause some of the above conditions to be detected. The conditions actually occur for brief periods in normal operation. Only a persistent condition should be interpreted as indicating a hardware fault.

TMFirst

This API call finds the first multi-drop device on the MicroLan specified with the ‘session_handle’. A ROM search algorithm is used to find the first unique serial registration number (ROM) data pattern on the MicroLan. The ROM data pattern that

was found is stored in an internal eight byte buffer. The internal eight byte buffer can be read using the TMRom API call.

Microsoft Windows TMEX DLL function prototype:

short far pascal TMFirst(long session_handle, void far *state_buffer);

The function returns:
0 => device not found
1 => first device on the MicroLan found
<0 => NETWORK error

DOS TMEX TSR calling procedure:

Prior to function call:
- Upper nibble of AH is MicroLan number.
- Lower nibble of AH is 04 hex.

Call a Network layer TMEX interrupt (60-66 hex) with 'DOW' at the beginning of the ID string. See APPENDIX A for details.

Upon return from function:
- Carry is set if there is an error in execution.
- If carry is set then AL contains the NETWORK error code. If carry is not set then AL contains a 00 hex if a successful search was not completed and 01 if a valid ROM was read and placed in the internal eight byte buffer.
- There may be diagnostic results in DX. (see text at the beginning of this section)

The TMEXLIB.C function prototype is:

short far pascal TMFirst(short port);

The function returns the same as the TMEX DLL call.

Code Example for TMEX Microsoft Windows call:

```
long session_handle;
unsigned char state_buffer[15360];
short result;

/* session_handle set from a call to TMExtendedStartSession */
...

/* attempt to find the first device on MicroLan */
result = TMFirst(session_handle, state_buffer);

if (result == 1)
{
   /* device ROM number found, call TMRom to get number */
```

```
}
else
{
   /* no device on MicroLan at this time or error */
   ...
}

/* close the session with a call to TMEndSession */
...
```

TMNext

This API call finds the next multi-drop device on the MicroLan specified with the session_handle. A ROM search algorithm is used to find the next unique registration number (ROM) data pattern on the MicroLan. The ROM data pattern that was found is stored in an internal eight byte buffer. The internal eight byte buffer can be read using the TMRom API call. After the last device on the MicroLan is found the next call to TMNext will return a 0. When TMNext returns a 0, the search algorithm will be reset and the next call to TMNext will be equivalent to a call to TMFirst.

Microsoft Windows TMEX DLL function prototype:

short far pascal TMNext(long session_handle, void far *state_buffer);

The function returns:
0 => device not found
1 => next device on the MicroLan found
<0 => NETWORK error

DOS TMEX TSR calling procedure:

Prior to function call:
- Upper nibble of AH is MicroLan number.
- Lower nibble of AH is 05 hex.

Call a Network layer TMEX interrupt (60-66 hex) with 'DOW' at the beginning of the ID string. See APPENDIX A for details.

Upon return from function:
- Carry is set if there is an error in execution.
- If carry is set then AL contains the NETWORK error code. If carry is not set then AL contains a 00 hex if a successful search was not completed and 01 if a valid ROM was read and placed in the internal eight byte buffer.
- There may be diagnostic results in DX. (see text at the beginning of this section)

The TMEXLIB.C function prototype is:

short far pascal TMNext(short port);

The function returns the same as the TMEX DLL call.

Code Example for TMEX Microsoft Windows call:

```
long session_handle;
unsigned char state_buffer[15360];
short result, cnt = 0;

/* session_handle set from a call to TMExtendedStartSession */
...

/* attempt to find the first device on MicroLan */
result = TMFirst(session_handle, state_buffer);

/* loop to count all of the devices on the MicroLan */
while (result > 0)
{
   cnt++;

   /* find next device */
   result = TMNext(session_handle, state_buffer);
}

/* close the session with a call to TMEndSession */
...
```

TMAccess

This API call transmits a TMTouchReset signal and returns 0 if no Presence signal is detected. If a Presence signal is detected, it accesses the device whose ROM code is in the internal eight byte buffer. The access readies the iButton to accept memory function commands such as read scratchpad. Note that a successful return of 1 from TMAccess does not guarantee that the device whose serial ROM number is in the internal eight byte buffer is actually on the MicroLan. A successful return only guarantees that some iButton device is on the MicroLan and if the desired iButton is present then it is selected and ready for a device specific command.

Microsoft Windows TMEX DLL function prototype:

short far pascal TMAccess(long session_handle, void far *state_buffer);

The function returns:
0 => no presence on MicroLan
1 => presence on MicroLan and ROM selected
<0 => NETWORK error

DOS TMEX TSR calling procedure:

Prior to function call:
- Upper nibble of AH is MicroLan number.
- Lower nibble of AH is 06 hex.
- AL contains 00 hex

Call a Network layer TMEX interrupt (60-66 hex) with 'DOW' at the beginning of the ID string. See APPENDIX A for details.

Upon return from function:
- Carry is set if there is an error in execution.
- If carry is set then AL contains the NETWORK error code. If carry is not set then AL contains a 00 hex if no Presence signal was detected and 01 if the access was successful.
- There may be diagnostic results in DX. (see text at the beginning of this section)

The TMEXLIB.C function prototype is:

short far pascal TMAccess(short port);

The function returns the same as the TMEX DLL call.

Code Example for TMEX Microsoft Windows call:

```
long session_handle;
unsigned char state_buffer[15360];
short result;

/* session_handle set from a call to TMExtendedStartSession */
...

/* the internal 8-byte ROM buffer contains a valid ROM that is going to
be selected to do a device specific operation */
...

/* access the current device */
result = TMAccess(session_handle, state_buffer);

if (result == 1)
{
   /* device accessed, ready for memory operation */
   ...
}
else
{
   /* no device on MicroLan or an error */
   ...
}
```

```
/* close the session with a call to TMEndSession */
...
```

TMStrongAccess

This API call verifies that the device in the ROM buffer is on the MicroLan and selects it for a device specific command. This is the same as TMAccess except instead of using a match ROM command the search ROM command is used. This only disadvantage to this API call is that it takes three times as long to run as TMAccess.

Microsoft Windows TMEX DLL function prototype:

short far pascal TMStrongAccess(long session_handle, void far *state_buffer);

The function returns:
0 => device in ROM buffer not on MicroLan
1 => device in ROM buffer on MicroLan and it is selected
<0 => NETWORK error

DOS TMEX TSR calling procedure:

Prior to function call:
- Upper nibble of AH is MicroLan number.
- Lower nibble of AH is 06 hex.
- AL contains F0 hex for a TMStrongAccess

Call a Network layer TMEX interrupt (60-66 hex) with 'DOW' at the beginning of the ID string. See APPENDIX A for details.

Upon return from function:
- Carry is set if there is an error in execution.
- If carry is set then AL contains the NETWORK error code. If carry is not set then AL contains a 00 hex if device not found and 01 if it was.
- There may be diagnostic results in DX. (see text at the beginning of this section)

The TMEXLIB.C function prototype is:

short far pascal TMStrongAccess(short port);

The function returns the same as the TMEX DLL call.

Code Example for TMEX Microsoft Windows call:

```
long session_handle;
unsigned char state_buffer[15360];
short result;
```

```
/* session_handle set from a call to TMExtendedStartSession */
...

/* the internal 8-byte ROM buffer contains a valid ROM that is going to
be selected to do a device specific operation */
...

/* verify and access the current device */
result = TMStrongAccess(session_handle, state_buffer);

if (result == 1)
{
   /* device on MicroLan and accessed, ready for
      memory operation */
   ...
}
else
{
   /* device not on MicroLan or an error */
   ...
}

/* close the session with a call to TMEndSession */
...
```

TMStrongAlarmAccess

This API call starts a new communication session with a particular device on the MicroLan. This is the same as TMStrongAccess except this function requires that the iButton must have an alarm interrupt condition to be accessed. Returns a value of 1 if the selected part is on the MicroLan and alarming, and 0 otherwise. The DS1994, DS1920, and DS2407 potentially have an alarm condition.

Microsoft Windows TMEX DLL function prototype:

short far pascal TMStrongAlarmAccess(long session_handle, void far *state_buffer);

The function returns:
0 => device in ROM buffer not on MicroLan or not alarming
1 => device in ROM buffer is on MicroLan and alarming and it is selected
<0 => NETWORK error

DOS TMEX TSR calling procedure:

Prior to function call:
- Upper nibble of AH is MicroLan number.

- Lower nibble of AH is 06 hex.
- AL contains 00 hex for the default access, F0 hex for a TMStrongAccess and EC hex for a TMStrongAlarmAccess.

Call a Network layer TMEX interrupt (60-66 hex) with 'DOW' at the beginning of the ID string. See APPENDIX A for details.

Upon return from function:
- Carry is set if there is an error in execution.
- If carry is set then AL contains the NETWORK error code. If carry is not set then AL contains a 01 if alarming device found and 00 if it was not.
- There may be diagnostic results in DX. (see text at the beginning of this section)

The TMEXLIB.C function prototype is:

short far pascal TMStrongAlarmAccess(short port);

The function returns the same as the TMEX DLL call.

Code Example for TMEX Microsoft Windows call:

```
long session_handle;
unsigned char state_buffer[15360];
short result;

/* session_handle set from a call to TMExtendedStartSession */
...

/* the internal 8-byte ROM buffer contains a valid ROM that is going to
be selected to do a device specific operation */
...

/* verify the device is alarming and access it */
result = TMStrongAlarmAccess(session_handle, state_buffer);

if (result == 1)
{
   /* device on MicroLan and alarming and accessed, ready for
      memory operation */
   ...
}
else
{
   /* device not on MicroLan or not alarming or error */
   ...
}

/* close the session with a call to TMEndSession */
...
```

TMOverAccess

This API call transmits a TMTouchReset signal and returns 0 if no Presence signal is detected. If a Presence signal is detected, it sends an Overdrive match to the device whose ROM code is in the internal eight byte buffer. The Overdrive match readies the iButton to accept memory function commands such as read scratchpad and also sets the device in Overdrive speed. A successful return only guarantees that some iButton device is on the MicroLan and if the desired iButton is present then it is selected, in Overdrive communication mode, and ready for a device specific command.

Microsoft Windows TMEX DLL function prototype:

short far pascal TMOverAccess(long session_handle, void far *state_buffer);

The function returns:
0 => no presence on MicroLan
1 => presence on MicroLan, in Overdrive mode, and ROM selected
<0 => NETWORK error

DOS TMEX TSR calling procedure:

Prior to function call:
- Upper nibble of AH is MicroLan number.
- Lower nibble of AH is 06 hex.
- AL contains 69 hex

Call a Network layer TMEX interrupt (60-66 hex) with 'DOW' at the beginning of the ID string. See APPENDIX A for details.

Upon return from function:
- Carry is set if there is an error in execution.
- If carry is set then AL contains the NETWORK error code. If carry is not set then AL contains a 00 hex if no Presence signal was detected and 01 if the access was successful.
- There may be diagnostic results in DX. (see text at the beginning of this section)

The TMEXLIB.C function prototype is:

short far pascal TMOverAccess(short port);

The function returns the same as the TMEX DLL call.

Code Example for TMEX Microsoft Windows call:

```
long session_handle;
unsigned char state_buffer[15360];
short result;
```

```
/* session_handle set from a call to TMExtendedStartSession */
...

/* the internal 8-byte ROM buffer contains a valid ROM that is going to
be selected to do a device specific operation */
...

/* access the current device */
result = TMOverAccess(session_handle, state_buffer);

if (result == 1)
{
   /* device accessed, in Overdrive and ready for memory operation */
   ...
}
else
{
   /* no device on MicroLan or an error */
   ...
}

/* close the session with a call to TMEndSession */
...
```

TMRom

This API call transfers a ROM data pattern between the internal eight-byte buffer maintained by the driver and an array 'ROM' of eight integers declared in the application program. The direction of data transfer is specified by the value of the first integer in the array. If the first integer is zero, then the eight bytes from the internal buffer are transferred into the eight integer variables of the integer array. If the first integer is non-zero, then the least significant bytes of the eight integers are transferred into the internal eight-byte buffer. This function allows an application program to obtain the ROM data of a device that has been found with the TMFirst or TMNext functions. It also allows the application program to specify the ROM data of a specific device to be addressed with a multi-drop API call such as TMFirstFile. Note that only the least significant byte of each short integer in the ‘ROM’ array is used.

Microsoft Windows TMEX DLL function prototype:

short far pascal TMRom(long session_handle, void far *state_buffer, short far * ROM);

The function returns:
1 => ROM read or set
<0 => NETWORK error

DOS TMEX TSR calling procedure:
The DOS implementation of the TMRom function differs from the Microsoft Windows implementation in that the DOS TMEX API call only provides the DOS application with a pointer to the 8-byte internal ROM buffer. Using this pointer the ROM buffer can be read and set directly.

Prior to function call:
- Upper nibble of AH is MicroLan number.
- Lower nibble of AH is 07 hex.

Call a Network layer TMEX interrupt (60-66 hex) with 'DOW' at the beginning of the ID string. See APPENDIX A for details.

Upon return from function:
- Carry is set if there is an error in execution.
- If carry is set then AL contains the NETWORK error code.
- ES:BX is a far pointer (Segment:Offset) to the eight byte internal ROM buffer.
- There may be diagnostic results in DX. (see text at the beginning of this section)

The TMEXLIB.C function prototype is:

short far pascal TMRom(short port, short far *ROM);

The function returns the same as the TMEX DLL call.

Code Example for TMEX Microsoft Windows call:

```
long session_handle;
unsigned char state_buffer[15360];
short result;
short ROM[8];

/* session_handle set from a call to TMExtendedStartSession */
...

/* call TMFirst to find the first device on the MicroLan */
result = TMFirst(session_handle, state_buffer);

if (result == 1)
{
   /* device found, now read its unique ROM number */
   ROM[0] = 0;  /* zero the first integer to indicate a read */
   result = TMRom(session_handle, state_buffer, ROM);

   if (result == 1)
   {
      /* ROM buffer now has unique ROM in it */
      /* ROM[0] has the family code and ROM[7] has the 8 bit CRC */
      ...
```

```
   }
   else
     /* session not valid */
}
else
{
   /* device not found on MicroLan */
   ...
}

/* now set the internal ROM buffer to a previously read value */
ROM[0] = 0x0C;
ROM[1] = 0xE2;
ROM[2] = 0x01;
ROM[3] = 0x00;
ROM[4] = 0x00;
ROM[5] = 0x00;
ROM[6] = 0x00;
ROM[7] = 0x8F;

result = TMRom(session_handle, state_buffer, ROM);

if (result == 1)
{
    /* ROM set, now can do mult-drop function such as TMFirstFile */
    ...
}
else
   /* session not valid */

/* close the session with a call to TMEndSession */
...
```

TMFirstAlarm

This API call operates exactly like the function TMFirst described above, except that the search is limited to those parts with an active alarm interrupt pending. This allows a program to limit the scope of the search algorithm to only those parts that may require attention because an alarm condition has been set. The DS1994, DS1920, and DS2407 potentially have an alarm condition. The unique registration number ROM found is stored in the internal 8-byte buffer that can be read and set with the API call TMRom.

Microsoft Windows TMEX DLL function prototype:

short far pascal TMFirstAlarm(long session_handle, void far *state_buffer);

The function returns:
0 => device not found

1 => first alarming device on the MicroLan found
<0 => NETWORK error

DOS TMEX TSR calling procedure:

Prior to function call:
- Upper nibble of AH is MicroLan number.
- Lower nibble of AH is 08 hex.

Call a Network layer TMEX interrupt (60-66 hex) with 'DOW' at the beginning of the ID string. See APPENDIX A for details.

Upon return from function:
- Carry is set if there is an error in execution.
- If carry is set then AL contains the NETWORK error code. If carry is not set then AL contains a 00 hex if no alarming part is on the MicroLan and 01 if a valid ROM was read and placed in the internal eight byte buffer.
- There may be diagnostic results in DX. (see text at the beginning of this section)

The TMEXLIB.C function prototype is:

short far pascal TMFirstAlarm(short port);

The function returns the same as the TMEX DLL call.

Code Example for TMEX Microsoft Windows call:

```
long session_handle;
unsigned char state_buffer[15360];
short result;

/* session_handle set from a call to TMExtendedStartSession */
...

/* attempt to find the first alarming device on MicroLan */
result = TMFirstAlarm(session_handle, state_buffer);

if (result == 1)
{
   /* alarming device ROM number found, call TMRom to get number */
   ...
}
else
{
   /* no alarming device on MicroLan at this time or error */
   ...
}

/* close the session with a call to TMEndSession */
```

```
...
```

TMNextAlarm

This API call operates exactly like the function TMNext described above, except that the search is limited to those parts with an active alarm interrupt pending. The DS1994, DS1920, and DS2407 potentially have an alarm condition. The unique registration number ROM found is stored in the internal 8-byte buffer that can be read and set with the API call TMRom.

Microsoft Windows TMEX DLL function prototype:

short far pascal TMNextAlarm(long session_handle, void far *state_buffer);

The function returns:
0 => next alarming device not found
1 => next alarming device on the MicroLan found
<0 => NETWORK error

DOS TMEX TSR calling procedure:

Prior to function call:
- Upper nibble of AH is MicroLan number.
- Lower nibble of AH is 09 hex.

Call a Network layer TMEX interrupt (60-66 hex) with 'DOW' at the beginning of the ID string. See APPENDIX A for details.

Upon return from function:
- Carry is set if there is an error in execution.
- If carry is set then AL contains the NETWORK error code. If carry is not set then AL contains a 00 hex if no more alarming parts are on the MicroLan and 01 if a valid ROM was read and placed in the internal eight byte buffer.
- There may be diagnostic results in DX. (see text at the beginning of this section)

The TMEXLIB.C function prototype is:

short far pascal TMNextAlarm(short port);

The function returns the same as the TMEX DLL call.

Code Example for TMEX Microsoft Windows call:

```
long session_handle;
unsigned char state_buffer[15360];
short result, cnt = 0;
```

```
/* session handle set from a call to TMExtendedStartSession */
...

/* attempt to find the first alarming device on MicroLan */
result = TMFirstAlarm(session_handle, state_buffer);

/* loop to count all of the alarming devices on the MicroLan */
while (result == 1)
{
   cnt++;

   /* find next alarming device */
   result = TMNextAlarm(session_handle, state_buffer);
}

/* close the session with a call to TMEndSession */
...
```

TMFamilySearchSetup

This API call sets the ROM buffer and search status to attempt to find a certain family type. This does not effect the MicroLan in any way, it just sets the search algorithm up to find a certain family type the next time a search function is called (i.e. TMNext or TMNextAlarm). This function will be successful if the port requested to do this on is valid. This function sets up the search to find a certain type on the next call only if there is one available. If there is not one available, TMNext or TMNextAlarm may still find a device of a different type. The normal sequence to search for a particular family type is as follows:

1) Call TMFamilySearchSetup with the desired family type
2) Call TMNext or TMNextAlarm
3) if step 2 is successful then check the family code using TMRom to see if the correct type was found.
4) if step 2 is not successful or the incorrect family code was found in step 3 then no devices of that type were on the MicroLan at the time of the search.

Microsoft Windows TMEX DLL function prototype:

short far pascal TMFamilySearchSetup(long session_handle, void far *state_buffer, short family_type);

The function returns:
1 => search setup to find family_type
0 => invalid family type
<0 => NETWORK error

DOS TMEX TSR calling procedure:

Prior to function call:

- Upper nibble of AH is the MicroLan number.
- Lower nibble of AH is the main function code 0D hex.
- AL is the sub function code 01 hex to indicate the TMFamilySearchSetup function.
- CL is the desired family type.
- CH is 0.

Call a Network layer TMEX interrupt (60-66 hex) with 'DOW' at the beginning of the ID string. See APPENDIX A for details.

Upon Return from function:
- Carry is set if there is any error in execution.
- AL is the NETWORK error code if carry is set.
- The function was successful if the carry was not set.

The TMEXLIB.C function prototype is:

short far pascal TMFamilySearchSetup(short port, short family_type);

The function returns the same as the TMEX DLL call.

Code Example for TMEX Microsoft Windows call:

```
long session_handle;
unsigned char state_buffer[15360];
short result, ROM[8];

/* session_handle set from a call to TMExtendedStartSession */
...

/* setup the search to find family type 0x0C on the next search */
result = TMFamilySearchSetup(session_handle, state_buffer, 0x0C);

/* attempt to find the first 0x0C family device on MicroLan */
result = TMNext(session_handle, state_buffer);

if (result == 1)
{
   /* device ROM number found, call TMRom to get ROM */
   ROM[0] = 0; /* reset to read */
   result = TMRom(session_handle, state_buffer, ROM);
   if ((result == 1) && (ROM[0] == 0x0C))
   {
      /* success, a device with 0x0C family code found */
      ...
   }
   else
      /* device type not on MicroLan at time of search */
}
else
```

```
r
   /* no device on MicroLan at this time or error */
   ...
}

/* close the session with a call to TMEndSession */
...
```

TMSkipFamily

This API call sets the current search state to skip the current family code. This function will only work if there is a search in progress. The normal sequence to skip a particular family type is as follows:

1) Call TMFirst or TMFirstAlarm the first pass or TMNext or TMNextAlarm in subsequent passes.
2) if step 1 is successful then check the family type using TMRom to see if the family code needs to be skipped.
3) call TMSkipFamily if the family type needs to be skipped
4) if step 1 is not successful then no more devices are on the MicroLan.
5) go back to step1 if not done with search

Microsoft Windows TMEX DLL function prototype:

short far pascal TMSkipFamily(long session_handle, void far *state_buffer);

The function returns:
1 => search setup to skip current family type
<0 => NETWORK error

DOS TMEX TSR calling procedure:

Prior to function call:
- Upper nibble of AH is the MicroLan number.
- Lower nibble of AH is the main function code 0D hex.
- AL is the sub function code 02 hex to indicate the TMSkipFamily function.
- CX is 0000 hex.

Call a Network layer TMEX interrupt (60-66 hex) with 'DOW' at the beginning of the ID string. See APPENDIX A for details.

Upon Return from function:
- Carry is set if there is any error in execution.
- AL is the NETWORK error code if carry is set.
- The function was successful if the carry was not set.

The TMEXLIB.C function prototype is:

short far pascal TMSkipFamily(short port);

The function returns the same as the TMEX DLL call.

Code Example for TMEX Microsoft Windows call:

```
long session_handle;
unsigned char state_buffer[15360];
short result, cnt = 0, ROM[8];

/* session_handle set from a call to TMExtendedStartSession */
...

/* attempt to find the first device on MicroLan */
result = TMFirst(session_handle, state_buffer);

/* loop to count all of the devices that don't have the family code 0x0C
on the MicroLan */
while (result == 1)
{
   /* read the ROM to see if need to skip */
   ROM[0] = 0; /* reset to read */
   result = TMRom(session_handle, state_buffer, ROM);
   if (result != 1)
      /* error, session not valid */

   if (ROM[0] == 0x0C)
   {
      /* skip this ROM */
      result = TMSkipFamily(session_handle, state_buffer);
      if (result != 1)
         /* error session not valid */
   }
   else
      cnt++;  /* count this device because not 0x0C */

   /* find next device */
   result = TMNext(session_handle, state_buffer);
}

/* close the session with a call to TMEndSession */
...
```

TMAutoOverDrive

This API call sets TMEX to automatically switch to Overdrive communication speed if the iButton, MicroLan and driver support it. This is done by doing all serial ROM searches at normal 1-Wire speeds and switching to Overdrive speeds when the device is accessed with TMAccess. Automatic Overdrive is selected if the ‘Mode’ is 1 and not

selected if ‘Mode’ is 0. Under the following conditions the TMEX driver will switch to Overdrive:

1. TMAutoOverDrive has be called with Mode = 1
2. The serial ROM number family type supports Overdrive
3. TMAccess is called.
4. Successful communication to the part is achieved at Overdrive speeds

The following conditions will switch the TMEX driver out of Overdrive if communication is currently in Overdrive due to TMAutoOverDrive:

1. Any search NETWORK API call such as TMFirst or TMNext. A change to the serial ROM number using TMRom.
2. A failure of TMAccess at Overdrive speeds
3. End to the Session on that MicroLan.

Microsoft Windows TMEX DLL function prototype:

short far pascal TMAutoOverDrive(long session_handle, void far *state_buffer, short Mode);

The function returns:
1 => Auto-overdrive mode set
<0 => NETWORK error

DOS TMEX TSR calling procedure:

Prior to function call:
- Upper nibble of AH is the MicroLan number.
- Lower nibble of AH is the main function code 0D hex.
- AL is the sub function code 05 hex to indicate the TMAutoOverDrive function.
- CL bit 0 indicates the mode, bit 2 through 7 are 0
- CH is 00 hex.

Call a Network layer TMEX interrupt (60-66 hex) with 'DOW' at the beginning of the ID string. See APPENDIX A for details.

Upon Return from function:
- Carry is set if there is any error in execution.
- AL is the NETWORK error code if carry is set.
- The function was successful if the carry was not set.

The TMEXLIB.C function prototype is:

short far pascal TMAutoOverDrive(short port);

The function returns the same as the TMEX DLL call.

Code Example for TMEX Microsoft Windows call:

```
long session_handle;
unsigned char state_buffer[15360];
short result, cnt = 0, ROM[8];

/* session_handle set from a call to TMExtendedStartSession */
...

/* Set into Auto Overdrive */
result = TMAutoOverDrive(session_handle, state_buffer, 1);

/* check the result */
if (result == 1)
{
   /* The driver will go in and out of Overdrive automatically */
   ...
}
else
   /* Error, invalid mode */
   ...

/* close the session with a call to TMEndSession */
...
```

Hardware Specific

The Hardware_Specific API layer functions described in this section are provided to enable the lowest level of communication to iButtons. These API calls are the only ones that would need to be changed for a new hardware interface so that all of TMEX could function. With the exception of TMSetup and TMClose, these API calls should not need to be called when using normal file-oriented TMEX operations. The more complex levels of TMEX call this layer for all operations. The HARDWARE_SPECIFIC error codes for this section are for the Microsoft Windows and TMEXLIB.C TMEX calls. The DOS TSR calls return the absolute value (non-negative) of these return codes with the carry set:

```
 -1  => PORT_NOT_INITIALIZED - Specified MicroLan has not been
        initialized with a call to the TMSetup function.
 -2  => PORT_NOT_EXIST - Specified MicroLan nonexistent.
 -3  => NO_SUCH_FUNCTION - Function not supported.
-12  => BCOM_FAILURE  Failure to communicate with hardware adapter
-13  => BCOM_EVENT  An unsolicited event occurred on the 1-Wire.
-200 => INVALID_SESSION - session not valid (not applicable in DOS TMEX)
-201 => HS_NOT_FOUND - Hardware_Specific driver not found and is
        required.
```

TMSetup

This API function must be called before any other TMEX functions except the Session

layer of API calls. No other TMEX API calls will operate correctly until the MicroLan has been called with this function. It is intended to be called once at the beginning of an application to initialize the MicroLan and verify the physical integrity of the MicroLan. Note that execution of this function will reset all the parts on the specified MicroLan.

Microsoft Windows TMEX DLL function prototype:

short far pascal TMSetup(long session_handle);

The function returns:
0 => Setup failed.
1 => Setup OK.
2 => Setup OK but MicroLan shorted.
3 => MicroLan does not exist.
4 => TMSetup not supported
-200 => Invalid Session

DOS TMEX TSR calling procedure:

Prior to function call:
- Upper nibble of AH is MicroLan number.
- Lower nibble of AH is 00 hex.

Call a Hardware_Specific layer TMEX interrupt (60-66 hex) with 'DOW' at the beginning of the ID string. See APPENDIX A for details.

Upon return from function:
- Carry is set if there is an error in execution.
- If carry is set then AL contains the HARDWARE_SPECIFIC error code. If carry is not set then AL has 00 hex if the MicroLan is shorted and 01 otherwise.
- There may be diagnostic results in DX. (see text at the beginning of the NETWORK section)

The TMEXLIB.C function prototype is:

short far pascal TMSetup(short port);

The function returns the same as the TMEX DLL call.

Code Example for TMEX Microsoft Windows call:

```
long session_handle;
short result;

/* session_handle set from a call to TMExtendedStartSession */
...

/* try to verify MicroLan setup, note MicroLan is known to
   TMEX through the session_handle that was given by
```

```
   TMExtendedStartSession */
result = TMSetup(session_handle);

if (result == 1)
{
   /* MicroLan is valid and setup */
   ...
}
else if (result == 2)
{
   /* MicroLan exists but is currently shorted. This could indicate a
      problem such as a missing DS9097 adapter or it could be only
      a transient problem */
   ...
}
else
{
   /* error in setting up MicroLan */
   ...
}

/* close the session with a call to TMEndSession */
...
```

TMTouchReset

This API call performs the Reset function on the MicroLan, resetting all of the devices on the MicroLan. The function returns the result of the operation.

Microsoft Windows TMEX DLL function prototype:

short far pascal TMTouchReset(long session_handle);

The function returns:
0 => No presence pulse detected.
1 => Non-alarming presence pulse detected.
2 => Alarming presence pulse detected. (not available on all platforms)
3 => MicroLan is shorted.
4 => TMSetup has not been run on MicroLan.
5 => TMTouchReset not supported
-200 => Invalid Session

DOS TMEX TSR calling procedure:

Prior to function call:
- Upper nibble of AH is MicroLan number.
- Lower nibble of AH is 01 hex.

Call a Hardware_Specific layer TMEX interrupt (60-66 hex) with 'DOW' at the beginning of the ID string. See APPENDIX A for details.

Upon return from function:
- Carry is set if there is an error in execution.
- If carry is set then AL contains the HARDWARE_SPECIFIC error code. If carry is not set then AL has 01 hex if a Presence indicator was detected and 00 hex otherwise.
- There may be diagnostic results in DX. (see text at the beginning of the NETWORK section)

The TMEXLIB.C function prototype is:

short far pascal TMTouchReset(short port);

The function returns the same as the TMEX DLL call.

Code Example for TMEX Microsoft Windows call:

```
long session_handle;
short result;
char result_str[80];

/* session_handle set from a call to TMExtendedStartSession */
...

/* reset the MicroLan and provide a string to represent the result */
result = TMTouchReset(session_handle);

switch(result)
{
   case 0: sprintf(result_str,"No device on MicroLan"); break;
   case 1: sprintf(result_str,"Device on MicroLan"); break;
   case 2: sprintf(result_str,"Alarming device on MicroLan"); break;
   case 3: sprintf(result_str,"MicroLan shorted"); break;
   case 4: sprintf(result_str,"MicroLan not setup"); break;
   case 5: sprintf(result_str,"TMTouchReset not supported"); break;
   default: sprintf(result_str,"unknown result"); break;
};

/* close the session with a call to TMEndSession */
...
```

TMTouchByte

This API call transmits the least significant byte of the variable 'outbyte' on the 1-wire bus and concurrently receives a byte value from the 1-wire bus. The received byte is the return value of the function. For a write operation, provide the byte to send in 'outbyte'. For a read operation, where the iButton is sending a byte, provide 0xFF in the 'outbyte'.

Microsoft Windows TMEX DLL function prototype:

short far pascal TMTouchByte(long session_handle, short outbyte);

The function returns:
>=0 => byte returned from MicroLan while sending byte
255 => 1-wire bus is shorted.
<0 => HARDWARE_SPECIFIC error

DOS TMEX TSR calling procedure:

Prior to function call:
- Upper nibble of AH is MicroLan number.
- Lower nibble of AH is 02 hex.
- AL is the byte to transmit

Call a Hardware_Specific layer TMEX interrupt (60-66 hex) with 'DOW' at the beginning of the ID string. See APPENDIX A for details.

Upon return from function:
- Carry is set if there is an error in execution.
- If carry is set then AL contains the HARDWARE_SPECIFIC error code. If carry is not set then AL has the returned byte of MicroLan communication.
- There may be diagnostic results in DX. (see text at the beginning of the NETWORK section)

The TMEXLIB.C function prototype is:

short far pascal TMTouchByte(short port, short outbyte);

The function returns the same as the TMEX DLL call.

Code Example for TMEX Microsoft Windows call:

```
long session_handle;
short result, i;
unsigned char ROM[8];

/* session_handle set from a call to TMExtendedStartSession */
...

/* read the ROM number of a single iButton on the MicroLan */
result = TMTouchReset(session_handle);
if (result == 1 || result == 2)
{
   /* device reset now, send read ROM command */
   TMTouchByte(session_handle, 0x33);

   /* now loop to receive all 8 bytes of the ROM */
```

```
   for (i = 0; i < 8; i++)
      ROM[i] = (unsigned char)TMTouchByte(session_handle, 0xFF);
}
else
{
   /* device not on MicroLan to read ROM */
   ...
}

/* close the session with a call to TMEndSession */
...
```

TMTouchBit

This API call transmits the least significant bit of the variable 'outbit' on the 1-wire bus and concurrently receives a bit from the 1-wire bus. The received bit is the return value of the function. For a write operation, provide the bit to send in 'outbit'. For a read operation, where the iButton is sending a bit, provide 0x01 in the 'outbit'. This API call is mostly used for ROM search operations.

Microsoft Windows TMEX DLL function prototype:

short far pascal TMTouchBit(long session_handle, short outbit);

The function returns:
0,1 => bit returned from MicroLan while sending bit
0x01 => 1-wire bus is shorted.
<0 => HARDWARE_SPECIFIC error

DOS TMEX TSR calling procedure:

Prior to function call:
- Upper nibble of AH is MicroLan number.
- Lower nibble of AH is 03 hex.
- LSB of AL has the bit to transmit.

Call a Hardware_Specific layer TMEX interrupt (60-66 hex) with 'DOW' at the beginning of the ID string. See APPENDIX A for details.

Upon return from function:
- Carry is set if there is an error in execution.
- If carry is set then AL contains the HARDWARE_SPECIFIC error code. If carry is not set then the LSB of AL contains the bit received from the MicroLan.
- There may be diagnostic results in DX. (see text at the beginning of the NETWORK section)

The TMEXLIB.C function prototype is:

short far pascal TMTouchBit(short port, short outbit);

The function returns the same as the TMEX DLL call.

Code Example for TMEX Microsoft Windows call:

```
long session_handle;
short result;

/* session_handle set from a call to TMExtendedStartSession */
...

/* send a 1 bit to the MicroLan */
result = TMTouchBit(session_handle, 0x01);

if (result >= 0)
{
   /* bit returned from MicroLan is LSB if result */
   ...
}
else
   /* session handle is not valid */

/* close the session with a call to TMEndSession */
...
```

TMProgramPulse

This API call requires a hardware adapter that supports programming voltage such as the DS9097E COM serial port brick adapter. It should be used only if there is an iButton EPROM device on the MicroLan. This function in conjunction with suitable hardware puts out a 480 microsecond 12 volt pulse on the MicroLan to program EPROM bits in an iButton EPROM device such as the DS1982. If an NV-RAM or memory-less iButton is on the MicroLan, damage may occur from the 12 volt pulse.

Microsoft Windows TMEX DLL function prototype:

short far pascal TMProgramPulse(long session_handle);

This function returns:
1 => 480us 12 volt pulse sent to MicroLan.
<0 => HARDWARE_SPECIFIC error

DOS TMEX TSR calling procedure:

Prior to function call:
- Upper nibble of AH is the MicroLan number.
- Lower nibble of AH is the main function code 0E hex.

- AL is the sub function code 01 hex to indicate the TMProgramPulse function.

Call a Hardware_Specific layer TMEX interrupt (60-66 hex) with 'DOW' at the beginning of the ID string. See APPENDIX A for details.

Upon Return from function:
- Carry is set if there is any error in execution.
- AL is the HARDWARE_SPECIFIC error code if carry is set.
- The function was successful if the carry was not set.

The TMEXLIB.C function prototype is:

short far pascal TMProgramPulse(short port);

The function returns the same as the TMEX DLL call.

Code Example for TMEX Microsoft Windows call:

```
long session_handle;
short result;

/* session_handle set from a call to TMExtendedStartSession */
...

/* set up an EPROM for programming and get to the point of sending
   programming pulse */
result = TMProgramPulse(session_handle);

if (result == 1)
{
   /* pulse sent to MicroLan */
   ...
}
else
{
   /* error sending pulse */
   ...
}

/* close the session with a call to TMEndSession */
...
```

TMClose

This API call closes a particular MicroLan. After this function is called on a MicroLan, the only way to use it again is to call TMSetup. A common use of this API call is in hardware situations where a port needs to be powered down when not in use. Note that in the Windows situation where multiple programs may be using a MicroLan,

closing a port may be disastrous so this function is callable but it does not have an effect in Windows.

Microsoft Windows TMEX DLL function prototype:

short far pascal TMClose(long session_handle);

The function returns:
1 => MicroLan is closed
<0 => HARDWARE_SPECIFIC error

DOS TMEX TSR calling procedure:

Prior to function call:
- Upper nibble of AH is MicroLan number.
- Lower nibble of AH is 0F hex.

Call a Hardware_Specific layer TMEX interrupt (60-66 hex) with 'DOW' at the beginning of the ID string. See APPENDIX A for details.

Upon return from function:
- Carry is set if there is an error in execution.
- If carry is set then AL contains the error code.

The TMEXLIB.C function prototype is:

short far pascal TMClose(short port);

The function returns the same as the TMEX DLL call.

Code Example for TMEX Microsoft Windows call:

```
long session_handle;
short result;

/* session_handle set from a call to TMExtendedStartSession */
...

/* close the MicroLan */
result = TMClose(session_handle);

if (result == 1)
{
   /* MicroLan closed */
   ...
}
else
   /* session handle is not valid */

/* close the session with a call to TMEndSession */
```

```
...
```

TMOneWireCom

This function reads (‘Operation’ = 1) or sets (‘Operation’ = 0) the 1-Wire communication state. The following are the values allowed for ‘TimeMode’:

00 hex - Normal timing (15-60 microsecond time slots)
01 hex - Overdrive timing
02 hex - Relaxed timing for long lines

Note that not all of the above modes will be available for all hardware platforms.

Microsoft Windows TMEX DLL function prototype:

short far pascal TMOneWireCom(long session_handle, short Operation, short TimeMode);

The function returns:
>=0 => TimeMode that was set or read
<0 => HARDWARE_SPECIFIC error

DOS TMEX TSR calling procedure:

Prior to function call:
- Upper nibble of AH is the 1-Wire port number.
- Lower nibble of AH is the main function code 0E hex.
- AL is the sub function code 03 hex to indicate the TMOneWireCom function.
- CH - Bit 0 is 1 if reading 1-Wire communication state and 0 if setting 1-Wire state. Bit 1 through Bit 7 are 0.
- CL - 1-Wire communication state if setting

Call a Hardware_Specific layer TMEX interrupt with 'DOW' at the beginning of the ID string.

Upon Return from function:
- Carry is set if there is any error in execution.
- AL is the HARDWARE_SPECIFIC error code if carry is set.
- The function was successful if the carry was not set.
- CL - 1-Wire communication state if reading

The TMEXLIB.C function prototype is:

short far pascal TMOneWireCom(short port, short Operation, short TimeMode);

The function returns the same as the TMEX DLL call.

Code Example for TMEX Microsoft Windows call:

```
long session_handle;
short result;

/* session_handle set from a call to TMExtendedStartSession */
...

/* read the OneWireCom state */
result = TMOneWireCom(session_handle, 1, 0);

switch (result)
{
   case 0x00: printf("Normal communication speed\n"); break;
   case 0x01: printf("Overdrive communication speed\n"); break;
   case 0x02: printf("Relaxed communication speed\n"); break;
   default: printf("Error %d\n"); break;
};

/* close the session with a call to TMEndSession */
...
```

TMOneWireLevel

This function reads ('Operation' = 1) or sets ('Operation' = 0) the 1-Wire communication level. The following are the allowed values to 'LevelMode':

00 hex - Normal high impedance (1500 ohm) normal voltage (3-5 volt)
01 hex - Strong pull-up (low impedance normal voltage state capable of sourcing 50 ma at 3 volts)
02 hex - Break (low impedance zero voltage)
03 hex - Program voltage (12 volt at 10 ma suitable for EPROM programming)

Primed(1) indicates the voltage remains unchanged now but gets changed after the next TMTouchBit. Primed(2) is the same but after the next TMTouchByte. Primed(0) means that the level change is immediate. Primed and value are not used when mode is reading(1).

Microsoft Windows TMEX DLL function prototype:

short far pascal TMOneWireLevel(long session_handle, short Operation, short LevelMode, short primed);

The function returns:
>=0 => LevelMode that was set, primed or read
<0 => HARDWARE_SPECIFIC error

DOS TMEX TSR calling procedure:

Prior to function call:
- Upper nibble of AH is the 1-Wire port number.

- Lower nibble of AH is the main function code 0E hex.
- AL is the sub function code 02 hex to indicate the TMOneWireLevel function.
- CH - Bit 0 is 1 if reading 1-Wire level and 0 if setting 1-Wire level. Bit 1 and Bit 2 indicate the 'primed' state 0,1, or 2 as discussed above. Bit 3 through Bit 7 are 0.
- CL - 1-Wire state if setting

Call a Hardware_Specific layer TMEX interrupt with 'DOW' at the beginning of the ID string.

Upon Return from function:
- Carry is set if there is any error in execution.
- AL is the HARDWARE_SPECIFIC error code if carry is set.
- The function was successful if the carry was not set.
- CL - 1-Wire level

The TMEXLIB.C function prototype is:

short far pascal TMOneWireLevel(short port, short Operation, short LevelMode, short primed);

The function returns the same as the TMEX DLL call.

Code Example for TMEX Microsoft Windows call:

```
long session_handle;
short result;

/* session_handle set from a call to TMExtendedStartSession */
...

/* read the OneWireLevel state */
result = TMOneWireLevel(session_handle, 1, 0, 0);

switch (result)
{
   case 0x00: printf("Normal communication speed\n"); break;
   case 0x01: printf("Overdrive communication speed\n"); break;
   case 0x02: printf("Relaxed communication speed\n"); break;
   default: printf("Error %d\n"); break;
};

OR

/* prime a change to OneWireLevel state after next TMTouchByte */
result = TMOneWireLevel(session_handle, 0, 1, 2);

if (result >= 0)
{
   /* start a temperature conversion on a DS1920 and do a strong pull-up
```

```
   right after the TMTouchByte is done */
   result = TMTouchByte(session_handle,0x44);
   ...
}
else
   /* error invalid LevelMode */
   ...

/* close the session with a call to TMEndSession */
...
```

TMGetTypeVersion

This API call reads the version ID string for the specified HARDWARE_SPECIFIC TMEX driver. The provided buffer 'ID_buf' must be at least 80 bytes to hold the ID string. The valid hardware specific type numbers ‘HSType’ are in the range 0 to 15. See the section ‘2.2 Microsoft Windows TMEX Considerations’ for a description of the type numbers and their default drivers.

Microsoft Windows TMEX DLL function prototype:

short far pascal TMGetTypeVersion(short HSType, char far *ID_buf);

The function returns:
1 => ID string is in ID_buf.
0 => error could not get the ID
<0 => HARDWARE_SPECIFIC error

DOS TMEX TSR calling procedure:

See the function 'GetVectID' in the 'C' library 'TMEXLIB.C' in the source examples.

Code Example for TMEX Microsoft Windows call:

```
char ID_buf[100];

/* get the ID string of the type 1 TMEX Hardware Specific driver */
TMGetTypeVersion(1,ID_buf);
```

TMBlockStream

This API call is a general purpose block transfer function This function simply does a stream of TMTouchByte (s) of all of the 'num_tran' bytes in the 'tran_buffer' data buffer. The values returned from the TMTouchByte(s) are placed back into the 'tran_buffer' data buffer. This call returns a byte length greater than or equal to 0 for success or one of the HARDWARE SPECIFIC error values described for a failure. This API call is similar to TMBlockIO but without the TMTouchReset at the beginning of

communication. The maximum number of 'num_tran' is 1023 bytes.

Microsoft Windows TMEX DLL function prototype:

short far pascal TMBlockStream(long session_handle, unsigned char far *tran_buffer, short num_tran);

The function returns:
>=0 => length of data sent and received from MicroLan
<0 => a TRANSPORT error has occurred

DOS TMEX TSR calling procedure:

Prior to function call:
- Upper nibble of AH is MicroLan number.
- Lower nibble of AH is the main function code OE hex.
- AL is the sub function code 04 hex to indicate the TMBlockStream function.
- CX is the number of bytes to transfer (1023 max)
- ES: BX is a far pointer (Segment: Offset) to the buffer of data to read and write.

Call a Network layer TMEX interrupt (60-66 hex) with 'DOW' at the beginning of the ID string. See APPENDIX A for details.

Upon return from function:
- Carry is set if there is an error in execution.
- If carry is set then AL contains the TMEX Hardware Specific Error Return Code.
- CX is the number of bytes transferred.

The TMEXLIB.C function prototype is:

short far pascal TMBlockStream(short port, unsigned char *tran_buffer, short num_tran);

The function returns the same as the TMEX DLL call.

Code Example for TMEX Microsoft Windows call:

```
long session_handle;
unsigned char state_buffer[15360], tran_buffer[100];
short result, flag;

/* session_handle set from a call to TMExtendedStartSession */
...

/* construct a buffer to read the scratchpad of a DS199X */
tran_buf[0] = 0xCC;   /* skip ROM */
tran_buf[1] = 0xAA;   /* read scratchpad */
for (i = 0; i < 35; i++)
```

```
   tran_buf[3+i] = 0xFF;  /* area to read address, data */

flag = TMTouchReset(session_handle);
if (flag ==1) or (flag == 2)
{
  result = TMBlockStream(session_handle, tran_buffer, 37);

  if (result == 37)
  {
     /* the contents of the scratchpad are in the buffer
      in location tran_buf[2] to tran_buf[36] */
   ...
  }
  else
     /* TRANSPORT error */
};
/* close the session with a call to TMEndSession */
```

Chapter 3 TMEX DOS and Windows Utility Programs

This section describes sample programs which have been provided to demonstrate the use of iButton-TMEX. The sample programs for the MS DOS environment are designed to perform familiar utility operations on iButton data files, similar to those provided by standard DOS commands. These programs are written in C. The C program examples call on a 'C' library (TMEXLIB.C) of routines that are very similar in calling convention to the DLL routines described above. The library source is provided. See section 'V. TMEX Source Examples'. The utility program for the Microsoft Windows environment is a typical 'multi-drop' application that provides file management.

3.1 TMEX DOS Utility Programs

The following utility programs provide file and directory functions similar to DOS operating system utilities. These utility programs require TMEX Version 3.10 for DOS to be installed prior to their execution. See section APPENDIX A for details on installing TMEX TSRs. Each of these utilities provides syntax instruction when run with '?' as a command line argument. The default MicroLan number for all of the utilities is 1. If the MicroLan is not 1 then the number must be provided on the command line or by two other methods. The first method is to set an environment variable. For example if the MicroLan number was 2 then add the following line to the 'AUTOEXEC.BAT': SET OneWirePort=2

The second method only works for the current directory the utility is executed from. Create a file called 'DEFAULT.PRT' and place the port number in the beginning of the file. For example: 2

The results of all of the utilities can be redirected into a file such as:

```
TDIR 2 >RESULTS.TXT
```

Optional command-line arguments are denoted with <>.

The format of a 'filename' argument is 'NAME.XXX' where 'NAME' is a 1 to 4 character name. A valid character is (ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789!#$%&'-@^_`{}~). 'XXX' is a decimal number extension in the range 0-99 inclusive. If a 'XXX' extension is not provided then 0 is assumed.

These utilities were made to be write-EPROM capable with the proper hardware (DS9097E) with the exception of the utility TOPT. Since each of the EPROM write operations are disjoint they are not very efficient. For example to put two files on an EPROM the utilities TFORMAT, TCOPY, and TCOPY could be called successfully. It is more efficient however to write the directory and the two files all at once. To do this

a custom utility would be needed. The source code to most of these utilities is provided to expedite such a utility.

All of the utilities provide meaningful DOS return codes enabling these utilities to be spawned by other programs to do iButton file operations. The following is a list of the return codes returned by the utilities:

1 => NO_DEVICE - no device found on MicroLan
2 => WRONG_TYPE - wrong type of device for specified operation
3 => FILE_READ_ERR - file/directory read error
4 => BUFFER_TOO_SMALL - buffer is smaller than the file to read
5 => HANDLE_NOT_AVAIL - all file handles are used
6 => FILE_NOT_FOUND - file specified is not in the current directory
7 => REPEAT_FILE - file already exists with name provided
8 => HANDLE_NOT_USED - given file handle is not assigned a file
9 => FILE_WRITE_ONLY - trying to read a write file handle
10 => OUT_OF_SPACE - not enough room on device to write
11 => FILE_WRITE_ERR - write error, part may have expired
12 => FILE_READ_ONLY - trying to write a read file handle
13 => FUNC_NOT_SUP - function not supported
14 => BAD_FILENAME - illegal file/directory name
15 => CANT_DEL_READ_ONLY - trying to delete read only file
16 => HANDLE_NOT_EXIST - file handle does not exist
17 => ONE_WIRE_PORT_ERROR - error in the MicroLan, MicroLan may not be setup using TMSetup
18 => INVALID_DIRECTORY - directory given is invalid
19 => DIRECTORY_NOT_EMPTY - directory must be empty to delete it
20 => UNABLE_TO_CREATE_DIR - maximum depth of 10 sub-directories is reached
21 => NO_PROGRAM_JOB - there is no program job started
22 => PROGRAM_WRITE_PROTECT - device is written in a way that can not be changed
23 => NON_PROGRAM_PARTS - there are non-EPROM parts on the MicroLan during an attempt to program
24 => ADDFILE_TERMINATED - the AddFile is terminated and cannot be appended to
50 => BAD_ARGUMENT - TMEX utility has been provided with a bad command-line argument
51 => NO_DRIVERS - TMEX utility could not find the correct TMEX TSR drivers
52 => KEY_ABORT - TMEX utility has been aborted with a key press
53 => OUT_MEMORY - TMEX utility could not run due to insufficient memory
0 => NORMAL_EXIT - TMEX utility has ended normally, no errors

TAPPEND

The TMEX utility 'TAPPEND' appends the provided DOS file to an ‘AddFile’ on an EPROM iButton. If the ‘AddFile’ does not exist it is created. If the iButton is not formatted then it is first formatted and then the ‘AddFile’ is created.

usage: TAPPEND DOS_source Touch_Memory_destination <OneWirePort>

TATTRIB

The TMEX utility 'TATTRIB' is similar to the DOS command 'ATTRIB' in that it changes the attribute of a file or sub-directory in the current directory on an iButton device. For TMEX Version 3.10, the attributes are read-only for files (R) and hidden for sub-directories (H).

usage: TATTRIB (+R | -R | +H | -H) (file/directory)name <OneWirePort>

TBIT

The TMEX utility 'TBIT' does bit manipulation on an ‘AddFile’ on an EPROM iButton. If the ‘AddFile’ does not exist it is created. The parameter ‘Bit_Number’ indicates the bit number that is to be read or cleared. The optional parameter ‘CLEAR’ will clear the indicated ‘Bit_Number’. The bits are numbered such that the most significant bit of the first byte is bit 0 and the most significant bit of the second byte is bit 8 etc.

usage: TBIT Touch_Memory_Addfile Bit_Number <CLEAR> <OneWirePort>

TCD

The TMEX utility 'TCD' is similar to the DOS command 'CD' or 'CHDIR' in that it reads or sets the current directory on an iButton. This utility specifies the MicroLan differently than the other utilities. It can be incorporated into a path. For example to change the iButton’s current directory on MicroLan 3: TCD 3:\TEMP\SRCE. The path follows standard DOS path convention. Running 'TCD' with no argument prints the current directory on the default MicroLan.

usage: TCD <<OneWirePort:>path>

TCOPY

The TMEX utility 'TCOPY' is similar to the DOS command 'COPY' in that it can copy a DOS file to an iButton device's current directory. It can not copy a file from an iButton to a DOS file however. To copy a file from an iButton, use 'TTYPE' and redirection.

usage: TCOPY DOS_source Touch_Memory_destination <OneWirePort>

TDEL

The TMEX utility 'TDEL' is similar to the DOS command 'DEL' in that it deletes a file from the current directory on the current iButton device. It can only delete a single file at a time. If all files must be deleted, use 'TFORMAT'.

usage: TDEL filename <OneWirePort>

TDIR

The TMEX utility 'TDIR' is similar to the DOS command 'DIR' in that it prints a list of the files on a device from the current directory. It also provides the length, starting location, and attributes of each file and sub-directory.

usage: TDIR <OneWirePort>

TFORMAT

The TMEX utility 'TFORMAT' is similar to the DOS command 'FORMAT' in that it re-formats an iButton device and readies it for files. Unlike DOS however, this is a quick operation. 'TFORMAT' will delete all files on an iButton device regardless of the attributes.

usage: TFORMAT <OneWirePort>

TMD

The TMEX utility 'TMD' is similar to the DOS command 'MD' or 'MKDIR' in that it makes a new sub-directory in the current directory. A valid new directory name has the format 'NAME' which is a 1 to 4 character string with the same character set as a file name. The directory name must be unique for the current directory.

usage: TMD dir_name <OneWirePort>

TPEEK

The TMEX utility 'TPEEK' searches the interrupts 60 Hex through 66 Hex and reports any TMEX TSR drivers loaded.

usage: TPEEK

TRD

The TMEX utility 'TRD' is similar to the DOS command 'RD' or 'RMDIR' in that it removes a sub-directory from the current directory. It can only remove an empty directory.

usage: TRD dir_name <OneWirePort>

TREN

The TMEX utility 'TREN' is similar to the DOS command 'REN' in that it renames a file. It can only rename a single file at a time.

usage: TREN oldfilename newfilename <OneWirePort>

TTYPE

The TMEX utility 'TTYPE' is similar to the DOS command 'TYPE' in that it prints an iButton file in the current directory to standard output. See 'TCD' for setting the current directory.

usage: TTYPE filename <OneWirePort>

TCHK

The TMEX utility 'TCHK' checks the extended file structure of an iButton device for errors or fragmentation. Each file is checked and its status given. If any error or fragmentation is found the TMEX utility 'TOPT' will be recommended.

usage: TCHK <OneWirePort>

TMEMCOPY

The TMEX utility 'TMEMCOPY' copies the extended file structure tree from one iButton to another. This is equivalent to running 'DISKCOPY' in DOS except the source and destination iButton can be different types as long as the destination device is large enough to contain all of the files/directories of the source.

usage: TMEMCOPY <OneWirePort>

TOPT

The TMEX utility 'TOPT' checks the extended file structure as in 'TCHK' and then corrects any problems found. This utility only works for NV-RAM type iButton.

usage: TOPT <OneWirePort>

TTREE

The TMEX utility 'TTREE' is similar to the DOS command 'TREE' in that it provides a tree list of all files and sub-directories.

usage: TTREE <OneWirePort>

TTEMP

The TMEX utility ‘TTEMP’ reads the temperature from DS1920/DS1820 iButton

Usage: TTEMP <Loop> <OneWirePort>

TTIME

The TMEX utility ‘TTREE’ is similar to the DOS command ‘TIME’ in that it reads the current time from a DS1994

Usage: TTIME <OneWirePort> <dXX/XX/XX> <tXX:XX:XX.XX>

TVIEW

The TMEX utility 'TVIEW' displays a list of the files, directories and device information on an iButton. Using the arrow keys, a file can be selected and its contents displayed. Selecting a sub-directory will result in the current directory changing to that sub-directory and the new file list being displayed. This version of TVIEW has a limit of only displaying 13 files/directories at a time. An optional parameter 'bw' will put 'TVIEW' into a black and white mode for mono monitors.

usage: TVIEW <OneWirePort> <bw>

Chapter 4 TMEX Source Code Examples

Source code is provided in the SDK for a variety of programs that call on TMEX 3.10 drivers in both DOS and Microsoft Windows environments. Examining these examples can provide valuable insight into the nuances of iButton application programming.

4.1 DOS Utility Programs

The C source code for most of the TMEX DOS utilities described Chapter 3 is included in the SDK. The TMEX API calls are made through a C library called TMEXLIB.C. This library has a function to find the TMEX interrupts and to make a call to each of the TMEX API functions. These utilities can be compiled with either Borland C or Microsoft C compilers.

4.2 RAY

The example program 'RAY' is provided with source code for several different programming languages in both DOS and Microsoft Windows. 'RAY' allows the inspection and editing of file data stored in a network of iButtons.

The graphical user interface (GUI) versions of 'RAY' are graphically oriented. The main 'RAY' window provides a list of the registration numbers on the MicroLan. A particular iButton can be opened for inspection of its directory by double-clicking on the registration number or by clicking once on the registration number and then clicking the Select button. A sub-directory can be opened in the same manner. A file in the file list window can then be selected for editing in the same way. The GUI version of ‘RAY’ for Microsoft Windows has source code for Delphi 1.0, Delphi 2.0, Microsoft Visual Basic 4.00 (RAYVB) (16 bit version, and 32 bit version) and Borland C++ Builder Version 1.0 (32 bit)(RAYCBLD). The Microsoft Visual C version of ‘RAY’ for Windows 3.1 and 95/NT were written as console applications with a text interface. See a description of the text oriented ‘RAY’ below.

The DOS versions of 'RAY' are text oriented. Each window described in the GUI versions of 'RAY' have a corresponding text menu. The registration number menu is displayed when the 'L' key is pressed. Selecting an iButton is done by pressing the number '0' to '9'. A menu list of the files on that iButton is then displayed. Selecting a sub-directory from this list will change the current directory and the new list is displayed. A file is selected for viewing and replacement the same way. The DOS version of 'RAY' has source code for Borland Pascal 7.0 (RAYDOSP), Borland C 4.0 (RAYDOSC), and Microsoft Visual Basic for DOS 1.0 (RAYDOSVB).

Additional source code examples included in this distribution as belows:

- Switch : The example program 'SWITCH' displays the information on DS2407. It

is provided with source code for several different programming languages in Microsoft Windows : Delphi 1.0 (SWDL), Delphi 2.0(SWDL32), Microsoft Visual C++ 4.00 (SWVC32), Microsoft Visual C++ 1.51 (SWVC) and , Microsoft Visual Basic 5.00 (SWVB32).

- Temp : The example program 'TEMP' displays the information on DS1920. It is provided with source code for several different programming languages in Microsoft Windows : Delphi 1.0 (TEMPDL), Delphi 2.0(TEMPDL32), Microsoft Visual C++ 4.00 (TEMPVC32), Microsoft Visual C++ 1.51 TEMPVC) and , Microsoft Visual Basic 5.00 (TEMPVB32).
- Time : The example program 'TIME' displays the information on DS1994. It is provided with source code for several different programming languages in Microsoft Windows : Delphi 1.0 (TIMEDL), Delphi 2.0(TIMEDL32), Microsoft Visual C++ 4.00 (TIMEVC32), Microsoft Visual C++ 1.51 (TIMEVC) and , Microsoft Visual Basic 5.00 (TIMEVB32).
- Demo1971: The program lists all DS1971 listed on the default MicroLan port as defined when installing TMEX and then allows the user to read and write to and from both the 256-bit EEPROM and the 64-bit one time programmable register of a selected device.
- OLE : The example program uses Object Linking and Embedding(OLE) to create an OLE control. OLETEMP creates an iBTemp control that displays the temperature of iButton (DS1920 or DS1980) found in the MicroLan port and OLELIST creates a RomList control that displays a list of iButtons in the MicroLan port. Both examples are written in Microsoft Visual C++ 4.00
- DDE : The example program uses Dynamic Data Exchange. The DDESERVER program will read the serial number out of an iButton and display the number and a time stamp in a text box. The DDECLIENT program can receive this info from the server and display it in a text box as well. Both programs are written in Delphi 2.0.
- Access: The example program is written in Visual Basic for Applications . (ACCESS95 with the development environment for Access 95 and ACCESS97 with the development environment for Access 97). It contains the database.The program will read an iButton's serial number, produce a timestamp to go with it, and stuff the results in a table.
- Window CE : The example program is written in Microsoft Visual C++ 5.0. It is run on the Hitachi SH-3 and MIPs Windows CE 1.0 palmtops. READROM will display the serial ROM number of iButton in the middle of the Window. READDIR will display the serial ROM number and the directory contents of the iButton in the middle of the Window .
- Microprocessor : the example program is written in different Assembly languages (Motorola 68HC11, 68HC11and 808x Assembly) and run on the Microprocessors (Hitachi 6301, 6303 and Motorola 68HC11,Microprocessor 8051, Microprocessor 808x).

Chapter 5 Specification of the Extended File Structure

The Extended File Structure is the foundation of the Presentation Layer of iButton-TMEX. It provides a directory structure for iButton data, allowing named files to be randomly accessed as they are on a diskette. The "Book of DS19xx iButton Standards" provides a basic specification of the Extended File Structure. This document extends the basic definition to include bitmap files for large capacity iButtons, multiple sub-directories, extended file attributes, passwords, date-stamps, owner identification, and other useful constructs.

The definitions and rules of the Extended File Structure are sufficient to store multiple files in nested directories using device capacities up to 64K bytes. These devices may be organized as 4...256 pages of 32...256 bytes. The rules given in this document are scaleable to all combinations of memory organization within this range. (Note that in all discussions of iButton data structure, the first available page of memory is called page 0.)

IButton-TMEX supplied with this kit implements a subset of the following Extended File Structure specification. Features denoted with a (*) have not yet been implemented in TMEX version 3.10.

5.1 Data Organization

The data organization of the Extended File Structure is very similar to that of a floppy disk. A sector of a floppy roughly corresponds to a page of an iButton. The directory tells which files are stored, where the data is located in the device, and how many pages it occupies. In this way information can be randomly accessed for quick response.

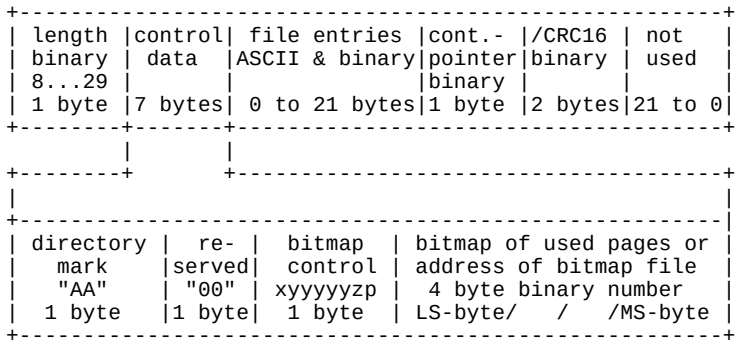
The organization of data within a page of a file or directory is shown below. (The numbers shown assume a 32 byte page length, but the same structure applies for devices with page lengths up to 256 bytes.)

length	data	cont.-	/CRC16	not
binary	ASCII or binary	pointer	binary	used
1...29		binary		
1 byte	0 to 28 bytes	1 byte	2 bytes	28 to 0

Each page of a file or directory begins with a length byte, contains a continuation pointer, and ends with an inverted CRC16 check. The continuation pointer is the page address where the file or directory is continued. A continuation pointer value of 0 marks the last page. The length byte indicates how many valid bytes a page contains,

not counting the length byte itself or the CRC. The CRC calculation, however, also includes the length byte. The CRC accumulator is initialized by setting it equal to the iButton page number. Every byte of a page is transmitted to or from an iButton least significant bit first. The length byte is the first to be transmitted. Of the two CRC bytes, the least significant will be sent first.

Each touch memory must be formatted before it can be used with the extended file structure. During the process of formatting, the root directory file is created. The root directory always begins in the first page of the iButton (page 0). The organization of data within the first page of the root directory is shown below:



Instead of data, the directory contains management information and file entries. The control field of seven bytes has the same length as a file entry. The bitmap supports TMEX in allocation of memory space for writing files. In the bitmap, used pages are marked with a 1, empty pages with a 0. The least significant bit corresponds to page 0. This local bitmap is only used for non-EPROM devices with less then 32 pages of data. All other devices have remote bitmap files. In large NV-RAM iButton the bitmap file page number refers to normal data space. In EPROM devices the page number refers to status memory. For EPROM devices the unprogrammed state is 1 and the programmed state is 0. Due to this constraint the bits in the EPROM bitmaps are inverted. An empty page has a 1 and an occupied page has a 0.

The most significant bit, "x", of the bitmap control byte specifies whether the bitmap is stored immediately in the first directory packet or in a separate file. If this bit is a 1, the 4 byte bitmap immediately follows the bitmap control byte. If this bit is a 0, the two bytes following the bitmap control byte are zero. The "y" bits in the bitmap control byte specify directory attributes. These bits are from most significant to least significant: (*) These attributes are not yet implemented in TMEX 3.10.

read-only : The directory that contains this attribute and all files and sub-directories can be read but not deleted or modified. If it is set on the root level then the entire device is read-only.
archive : The directory that contains this attribute has had some or all of its files modified since the last backup.
system : The directory that contains this attribute is designated to have system files.
encrypt : The directory that contains this attribute has all of its files encrypted.

The "z" bit in the bitmap control byte is not used. The least significant bit "p" is an in-progress bit. This bit can be set when doing non-interruptible operations like file

optimization. Note that TMEX does not set this attribute because none of its operations are non-interruptible. The bit is cleared after the operations are complete.

The next two bytes contain the starting page address and the number of pages required by the bitmap file. Note that the bitmap or the nameless bitmap file is created during the process of formatting. Continuation pages of the directory don't need a control field. This space is available to store another file entry.

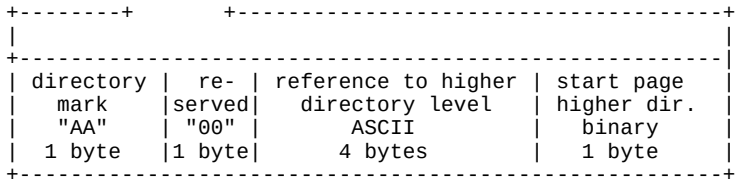
File entries consist of the 4-byte file name, one-byte file extension, the starting page address where the file begins, and the number of pages the file occupies. This structure is shown below:

file name ASCII, blank filled, left justified 4 bytes	extension binary aeeeeeee 1 byte	start page binary 1 byte	# pages binary 1 byte

File names must consist of ASCII characters only, as with DOS. However, if the first byte of a file entry has a value greater than 127, then it represents the first byte of an extended directory entry. The extended directory entry is a 7-byte or multiple 7-byte data packet that applies to the next directory entry. Each 7-byte data packet must have a first byte greater than 127. The additional bytes in an extended directory entry may be used to specify additional file attributes, passwords, file ownership, date/time stamps, and other special-purpose information regarding the file. (*) The extended directory entry is not supported in TMEX 3.10.

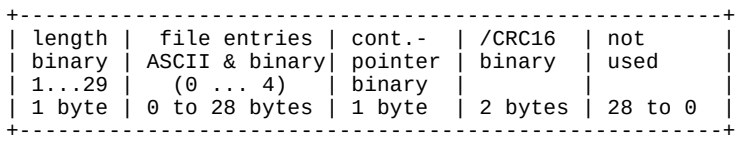
The 7 least significant "e" bits of the extension represent the extension number of the file entry. Extensions 0-99 decimal are reserved for normal file entries, 100 for an AddFile that resides on an EPROM iButton device and 101 for a Monetary File that is only created on a DS1962, DS1963, DS2422 and DS2423. The data page of the Monetary File can only be located on a page that has a corresponding counter. If a page with a counter is not available on the device then the Monetary File will not be created. Extension 127 decimal designates a sub-directory. Extensions 102-126 are reserved for future specialty file types. The most significant bit of the file extension "a" is an attribute flag for the entry. If it is set and the entry is a normal file with extensions 0-99 then the file is read-only. If it is set and the entry is a sub-directory with extension 127 then the sub-directory is hidden. File entries denoting sub-directories contain the real "start page" of the sub-directory file. The "# pages" specifier for a sub-directory entry does not contain a valid number. It always contains 0 and need not be updated when the length of a sub-directory file changes. A sub-directory file contains a back reference to the next higher directory file. If there is no higher sub-directory file the entry will be "ROOT". The start page of ROOT is always 0. The first packet of a sub-directory file has the following structure:

length binary 8...29 1 byte	control data 7 bytes	file entries ASCII & binary 0 to 21 bytes	cont.-pointer binary 1 byte	/CRC16 binary 2 bytes	not used 21 to 0



directory mark	re-served	reference to higher directory level	start page higher dir.
"AA"	"00"	ASCII	binary
1 byte	1 byte	4 bytes	1 byte

Again, the numbers shown above presume a 32-byte page length, but the basic structure applies for greater page lengths. A continuation packet of a root directory or a sub-directory file follows the definition of a data packet. It has the following structure:



length binary	file entries ASCII & binary	cont.-pointer binary	/CRC16	not used
1...29	(0 ... 4)			
1 byte	0 to 28 bytes	1 byte	2 bytes	28 to 0

5.2 Features

The iButton Extended File Structure is carefully designed to provide high speed and the best performance in a Touch environment. Every memory page can be read, CRC-checked or written without the need to access other pages. If a file is modified, only the affected pages need to be rewritten. This provides a significant speed advantage. Pages of a file need not be contiguous. Files can be extended by redefining continuation pointers. Files can be grouped into nested sub-directories. Attributes defined for a directory apply for all files within it. The iButton file structure also accommodates future types of iButtons with up to 256 pages with a maximum of 256 bytes per page.

5.3 Properties

The notes below summarize many of the significant properties of the Extended File Structure as specified above. These notes may be helpful in understanding the implications of the Extended File Structure with regard to data storage and management.

- The page length is derived from the family code. iButton devices currently available have a 32 byte page length, but the definition of the Extended File Structure also supports greater page lengths.

- Each page of data in the Extended File Structure is a Universal Data Packet (UDP), starting at a page boundary. The Universal Data Packet of the Extended File Structure is somewhat limited compared with the general Universal Data Packet. This Universal Data Packet may be shorter but never longer than one page.

- Each Universal Data Packet starts with a length byte, contains a continuation pointer and ends with a CRC16 check. The pointer indicates the number of the page where a file is continued. A continuation pointer "0"

marks the last packet of a file.

- The length byte indicates the number of bytes between length byte and CRC, not counting the length byte and the CRC. The CRC calculation includes all of the preceding data, including the length byte. The CRC accumulator is initialized by setting it equal to the iButton page number.

- Each Universal Data Packet may be read and CRC-checked independently, without reference to data from other packets.

- All types of files consist of one or several Universal Data Packets. The Universal Data Packets contain either application data (data packet) or directory information (directory packet).

- Every byte of a data packet is transmitted least significant bit first. The length byte is the first to be transmitted.

- The maximum number of application data bytes within a data packet is (page length - 4).

- There is exactly one main directory file in a touch memory. This file is called the root directory and always starts at page 0.

- The maximum number of entries in the first packet of a (sub)directory file is (page length-11)/7. The maximum number of entries in continuation packets is (page length-4)/7.

- The bitmap serves TMEX as a fast lookup table to find free pages. The data of the bitmap are treated as a 4 byte binary number. The least significant bit corresponds to page 0. A used page is marked with a "1", an empty page with a "0". The least significant byte of the bitmap is stored at the lower address and is transmitted first. The bitmap file must be used with iButtons of capacity greater than 32 pages. It may be used with smaller devices depending on formatting.

- The only reserved extensions are 102 - 126. 0 - 99 are file extensions ,100 for an AddFile, 101 for a Monetary File and 127 is for sub-directories.
- It is allowed to have data packets with no data. A packet with only a length, pointer and CRC16 bytes is considered empty. These empty data packets act as pointers to the next data packet of the same file. It is allowed to use empty or partly filled (sub)directory continuation packets. Partly filled (sub)directory continuation packets contain data left justified, i. e. the unused space must follow the CRC.

Appendix A TMEX 3.10 drivers

This section describes each of the TMEX 3.10 drivers and their ID strings. See section

'II. TMEX APPLICATION DESIGN GUIDE' for an overview of the TMEX API layers and section 'III. TMEX API' for the specific API function calls.

A.1 DOS TMEX DRIVERS

The TMEX drivers for DOS are all in the form of terminate and stay resident (TSR) programs that provide interrupt service routines (ISR's) to call all of the TMEX 3.10 API functions.

IBFS97E.EXE

This TSR provides all of the TMEX layers for DOS using the PC COM port with a DS9097(E) type adapter. The Session layer is not provided in DOS since DOS is not normally multi-tasking. IBFS97E.EXE contains the functions to write EPROM iButton. IBFS97E.EXE automatically scans all of the interrupts 60 hex to 66 hex to find two that are empty. The default configuration is to place the 'DOW' interrupt containing the Hardware_Specific, Network, and Transport layers on the first available interrupt and to place the 'TMEX' interrupt containing the File_Operations on the next available interrupt. The interrupt numbers can also be specified on the command line. For example to put the 'TMEX' interrupt on 65 hex and the 'DOW' interrupt on 66 hex use the following command line parameters:

```
IBFS97E 65 66 <enter>
```

If there are not at least two free interrupts between 60 hex and 66 hex then IBFS97E.EXE will not be installed. A 'free' interrupt is determined if the vector is 0000:0000 or if the vector points to an IRET (return from interrupt). The ID string

```
DOWF0305 V3.10      4/1/98
```

is for the 'DOW' interrupt. Note that the 'F' following 'DOW' indicates the highest valid function number in hex. The '03' following the 'F' indicates the highest numbered valid sub-function for the Hardware_Specific layer. The sub-functions for the Hardware_Specific layer are set by using the function number 'E' in the lower nibble of AH and the sub-function number in AL. The '05' following the '03' indicates the highest numbered valid sub-function for the combined Network and Transport layers. The sub-functions for the combined Network and Transport layers are set by using the function number 'D' in the lower nibble of AH and the sub-function number in AL. The ID string

```
TMEXF04 V3.10      4/1/98
```

is for the 'TMEX' interrupt. Note that the 'F' following the 'TMEX' indicates the highest valid function number in hex. The '048' following the 'F' indicates the highest numbered valid sub-function for the File_Operations layer. The sub-functions for the File_Operations layer are set by using the function number 'E' in the lower nibble of AH and the sub-function number in AL. See section 'III TMEX API' for specifics on calling the API functions.

IBFS97U.EXE*

This TSR is functionally equivalent to IBFS97E.EXE except the Hardware_Specific layer communicates with the PC COM port using the DS9097(U) type adapter. The ID string

DOWF0305 V3.10 4/1/98 DS9097U

is for the 'DOW' interrupt. The ID string

TMEXF04 V3.10 4/1/98

is for the 'TMEX' interrupt.

IBFS10E.EXE

This TSR is functionally equivalent to IBFS97E.EXE except the Hardware_Specific layer communicates with the PC parallel LPT port using the DS1410E adapter. The ID string

DOWF0305 V3.10 4/1/98 for LPT

is for the 'DOW' interrupt. The ID string

TMEXF04 V3.10 4/1/98

is for the 'TMEX' interrupt.

IBFS.EXE

This TSR provides the TMEX File_Operations, Transport and Network layers. IBFS.EXE requires the Hardware_Specific layer to be already installed. IBFS.EXE needs an interrupt 60 hex to 66 hex. The default interrupt to install on is the first available. If a specific interrupt is desired it can be specified on the command line. To install on 64 hex then use:

IBFS 64 <enter>

If a specific Hardware_Specific TSR is desired it can be specified as a second command line argument. To install on 64 hex and use the TSR loaded on 61 hex then use:

IBFS 64 61 <enter>

The ID string is:

TMEXF04 V3.10 4/1/98

Note that the 'F' following the 'TMEX' indicates the highest valid function number in hex. The '04' following the 'F' indicates the highest number valid sub-function for the File_Operations layer. The sub-functions for the File_Operations layer is set by using the function number 'E' in lower nibble of AH and the sub-function number in AL. See section 'III TMEX API' for specifics on calling the API functions.

***** DS99097U DOS environment variables.

The following environment variables can be set before the DOS TMEX driver IBFS97U.EXE is loaded. These variables limit the operation of the IBFS97U.EXE driver to conform to machine limitations. For example to limit the driver to 19200 baud you could put the following statement in the Autoexec.bat before the loading of the IBFS97U.EXE driver:

SET IBMAXBAUD=19200

---

name: IBFIFO
parameters: ON, OFF
description: enables or disables the fifo in the com port.

name: IBMAXBAUD
parameters: 9600, 19200, 57600, 115200
description: sets the maximum baud rate that the driver will attempt

name: IBMAXSEND
parameters: 4, 8, 16, 32, 64
description: sets the maximum number of characters that are set at any one time

name: IBMINSEND
parameters: 2, 4, 8
description: sets the suggested minimum number of characters to send

A.2 Microsoft Windows TMEX Drivers

The TMEX drivers for Microsoft Windows are all in the form of dynamic link libraries (DLL). A TMEX DLL must be in the same directory as the application program or in the \WINDOWS\SYSTEM directory. Installing the TMEX Single User License will automatically copy the TMEX drivers into the WINDOWS\SYSTEM directory.

The TMEX 3.10 Windows drivers have a main driver and sub hardware specific sub-drivers. The version string of the main driver can be read using the ‘Get_Version’ API function. The format for the main identification (ID) string is "xx_DLLNAME_Vz.zz_month/day/year (FULLNAME.DLL)" where:

_ - represent spaces

xx hardware type code 00 to 99 that the DLL uses.
- 00 general DLL requiring a hardware specific type interrupt
- 01 COM port DLL
- 02 LPT port DLL
- 03 contain type 00,01, and 02
- FF indicates a version 3.10 main driver that has a selectable and configurable hardware drivers.

DLLNAME - the name of the DLL that ranges in length from 1 to 8 characters.

Vz.zz - version number. (i.e. V1.00) It could include other specifier such as 'Alpha' or ''.

month/day/year - date DLL released

(FULLNAME.DLL) - is the full name of the DLL driver.

The version string of the hardware specific sub-drivers are read using the new API call 'TMGetTypeVersion'. The format for the hardware specific ID string is "tttt_HARDWARENAME_Vz.zz_month/day/year (FULLNAME.DLL)" where:

tttt - is the one to four letter short version of the hardware type. This short name can be used by applications as an indicator of the port type such as 'COM' for the DS9097E COM port driver.

HARDWARENAME - is the name of the hardware such as the DS9097E or DS1410E 1-wire adapters.

The rest of the fields in the ID string represent the same as in the main ID string.

Windows 3.1

IBFS.DLL

This is the main TMEX DLL driver for Windows 3.1. It provides all of the file and optionally transport and network function layers. The ID string is:

FF IBFS V3.10 4/1/98 (IBFS.DLL)

It in turn calls one of the hardware specific DLL drivers provided in the SDK or from a third party.

- IB97E.DLL is the TMEX DLL hardware specific driver for the DS9097E COM port adapter for Windows 3.1. The ID string is:

COM DS9097E V3.10 4/1/98 (IB97E.DLL)

- IB97U.DLL is the TMEX DLL hardware specific driver for the DS9097U COM port adapter for Windows 3.1. The ID string is:

COM DS9097U V3.10 4/1/98 (IB97U.DLL)

- IB10E.DLL is the TMEX DLL hardware specific driver for the DS1410E LPT port adapter for Windows 3.1. This driver is called by IBFS.DLL. The ID string is:

LPT DS1410E V3.10 4/1/98 (IB10E.DLL)

- IBTSR.DLL is the TMEX DLL hardware specific driver to call a TMEX

compatible DOS driver outside of Windows 3.1. The ID string is:

TSR V3.10 4/1/98 (IBTSR.DLL)

Windows 95/NT

IBFS32.DLL

This is the main TMEX DLL driver for Windows 95/NT. It provides all of the file and optionally transport and network function layers. The ID string is:

FF IBFS32 V3.10 4/1/98 (IBFS32.DLL)

It in turn calls one of the hardware specific DLL drivers provided in the SDK or from a third party.

- IB97E32.DLL is the TMEX DLL hardware specific driver for the DS9097E COM port adapter for Windows 95/NT. The ID string is:

COM DS9097E V3.10 4/1/98 (IB97E32.DLL)

- IB97U32.DLL is the TMEX DLL hardware specific driver for the DS9097U COM port adapter for Windows 95/NT. The ID string is:

COM DS9097U V3.10 4/1/98 (IB97U32.DLL)

- IB10E32.DLL is the TMEX DLL hardware specific driver for the DS1410E LPT port adapter for Windows 95/NT. The ID string is:

LPT DS1410E V3.10 4/1/98 (IB10E32.DLL)

Windows CE

IBFSCE.DLL

This is the main TMEX DLL driver for Windows CE. It provides all of the file and optionally transport and network function layers. The ID string is:

FF IBFSCE V3.10 4/1/98 (IBFSCE.DLL)

It in turn calls one of the hardware specific DLL drivers provided in the SDK or from a third party.

- IB97ECE.DLL is the TMEX DLL hardware specific driver for the DS9097E COM port adapter for Windows CE. The ID string is:

COM DS9097E V3.10 4/1/98(IB97ECE.DLL)

- IB97UCE.DLL is the TMEX DLL hardware specific driver for the DS9097U COM port adapter for Windows CE. The ID string is:

COM DS9097U V3.10 4/1/98 (IB97UCE.DLL)

Appendix B Universal Data Packet (UDP)

The Universal Data Packet (UDP) is a structure to store data on iButton. It contains one to two length bytes, data and two inverted CRC16 bytes. The structure is:

length	data	CRC16
		binary
0...508	0...508	LO/HI
1-2 bytes	bytes	2 bytes

The UDPs always start on page boundaries but can end anywhere. The length is the number of data bytes not including the length byte(s) and the CRC16 bytes. There is one length byte if the number of data bytes is less then 255. If there are 255 or more data bytes then the first length byte is 255 and the next length byte is 0 to 253. The first and second length bytes added together provide the number of data bytes. The CRC16 is first initialized to the starting page number. This provides a check to verify the page that was intended is being read. The CRC16 is then calculated over the length and data bytes. The CRC16 is then inverted and stored low byte first followed by the high byte. A detailed description of the CRC16 can be found in APPENDIX A of the Book of DS19XX iButton Standards.

The Extended File Structure implemented in TMEX uses a subset of the Universal Data Packet. It limits the length so that the whole structure will fit in one page. For the current devices with 32 bytes per page the length is limited to 29 data bytes. The last data byte is used as a continuation pointer leaving 28 true data bytes.

The Default Data Structure (DDS) is an old standard that specifies only one Universal Data Packet starting at page zero. This standard does not provide a directory or file type operations but it has speed advantages for simple applications.

- iButton, Touch Memory, MicroLan, TMEX, and MicroLan are trademarks of Dallas Semiconductor Corporation.

- MICROSOFT is a registered trademark and WINDOWS is a trademark of Microsoft Corporation.