



Data Design Corporation
Gaithersburg, MD

TM1000

PERIPHERAL INSTRUMENT SYSTEM

TM1000-DK

TM1000 SYSTEM DEVELOPMENT KIT

June 2001

Data Design Corporation
7851-A Beechcraft Avenue
Gaithersburg, MD 20879
WWW.TM1000.COM
(301) 670-1157

CONTENTS

1.0 Introduction	3
1.1 What Is Included	3
1.2 The Development Environment	3
2.0 TM1000 Module Hardware	5
2.1 Mechanical	5
2.2 Electrical	6
2.2.1 Power Supply	7
2.2.2 Control Signals	8
2.2.3 Function And Data Signals	9
2.2.4 SCL and SDA	10
2.2.5 User Port	10
3.0 The TM1000 API	11
3.1 Opening A Path To A TM1000 Base Unit	12
3.2 Sharing A Base Unit	12
3.3 Singular Read And Write Functions	13
3.4 Block Read And Write Functions	13
3.5 Status Byte	14
3.6 User Port Operations	15
3.7 Hardware Identification	16
3.8 Module Calibration Data	17
4.0 TM1000 Programming Conventions	18
4.1 TM1000 System Manager	18
4.2 Application Shutdown Considerations	19
4.3 Development Considerations	19
5.0 Using The TMTest Application	20
5.1 Base Unit Status	21
5.2 Module Identification	21
5.3 Module I/O	22

1.0 Introduction

The Data Design TM1000 system is an instrument component which provides simple digital I/O connectivity to a host computer through the Universal Serial Bus (USB) and offers modular instrument expansion. The design is optimized for the rapid development of small custom instruments such as laboratory instrumentation, data acquisition, and automated test. Data Design makes available the hardware and software interface specifications in this development kit.

With the disappearance of traditional computer interfaces such as ISA and even the serial and parallel ports, there is a void to fill in the area of a simple desktop computer interfacing. With today's complex operating systems, programming any desktop interface can be a daunting task. Gone are the days of fiddling bits from a two line DOS program. The TM1000 is designed to bring back this nearly painless computer interfacing experience. The base unit interface is easier to use and more forgiving than even ISA ever was and the simple function call I/O can be handled from any programming environment, including popular graphical environment such as Visual Basic and LabVIEW. There is no need to know any of the technical details of the USB or Windows drivers. Just plug in the TM1000, and start thinking about your project.

1.1 What Is Included

The TM1000 System Development Kit includes the TM1000 base unit, power supply, USB cable, prototyping module, extender board, blank module lid, and a development kit CD. The intent of the development kit is to provide the tools and know how required to develop module hardware and the software to interface with it. Anyone who has ever done these things with other interfaces will find the TM1000 to be very easy.

1.2 The Development Environment

The TM1000 system is designed to operate on PC compatible computers running the Microsoft Windows 2000 and Windows 98 operating systems. The computer must have a USB port and appropriate Windows support must be installed. This will likely be the case for most PC's running these operating systems. Hardware capable of running these operating systems will be capable of running the TM1000 system software.

Complete the installation of the TM1000 system as described in the *Getting Started Manual* included with this kit. This process will install all the drivers and software infrastructure required for development.

The application programming interface (API) for the TM1000 system is implemented as a Windows DLL. Therefore, any development environment capable of calling functions in a Windows DLL can be used to access the TM1000 base unit and any installed module directly. If the target user group is intended to be users of TM1000 instruments, the software will have to follow certain conventions discussed later in this manual. However, if a captive audience is envisioned, the TM1000 can be addressed more loosely. Modern, powerful, compiled environments such as Visual Basic and Visual C are recommended for end user applications, though programming for application specific needs may make other environments desirable.

Copy the files from the \API directory on development kit CD to an appropriate place in the development environment. The API itself is identified as TMWIN32.DLL and should already be installed in the \%SYSTEM_ROOT%\SYSTEM32 directory. The TMWIN32.H is a header file which defines exported functions and thereby the API itself. This file can be used directly in Visual C applications. For compiled environments such as Visual C which require a link library, the TMWIN32.LIB file is also included.

Install the *TMTTest* program by running SETUP.EXE from the \TMTEST directory on the development kit CD. This program offers the ability to directly access the TM1000 module interface and certain common features of the modules as described later. This tool is invaluable in the development of module hardware and software checkout. Its features are described later in this manual. The *TMTTest* application is written in Visual Basic 6. The source code is included on the development kit CD in the \SOURCE\VB directory. This source code includes definitions for the API functions in VB syntax along with various other useful code for developers working in the VB environment.

That's all the installation that is required. The rest is up to the imagination. The rest of this manual covers the technical details of the hardware interface and the API.

2.0 TM1000 Module Hardware

The TM1000 base unit serves as a connection point between the host computer and the instrument module. There is no need to consider the interface medium. In fact, this is why the system is called a *Peripheral Instrument System*, since the interface could have as well been fire wire, SCSI, or anything else. The interface to the TM1000 hardware developer is a 96-pin DIN connector at the front of the base unit. The interface to the TM1000 software developer is the TM1000 API library. Everything in between is a black box.

The physical medium does affect data rate, however, among other details generally opaque to the developer. With the USB interface, the data rate approaches half a megabyte per second. Since the actual data rate will depend heavily on bus implementation and usage, the data rate can not be specified more accurately. The API and the underlying drivers use the USB bulk protocol which provides inherent protection against data error.

2.1 Mechanical

The TM1000 module must meet a strict mechanical specification to allow it to mate with the base unit. A module cover is available from Data Design, painted and ready for use. Unpainted covers and cover drawings can be obtained for quantity needs by contacting Data Design. A mechanical drawing of the module board can be found on the development kit CD in the \DRAWINGS directory.

2.2 Electrical

The 96-pin DIN connector used as the module interface is an industry standard connector traditionally divided into three rows of 32 pins labeled A, B, and C. Pin A1 is in the upper right corner of the connector as read from the front of the base unit. B1 and C1 are directly below A1. This follows standard layout for a connector of this type. However, the module connector has the benefit of being upside down backwards when installed. All specifications in this manual are for pins on the base unit. It is useful to design a connector pattern for module boards which is a mirror image of the layout on the base unit. Figure 2.1 below contains the pin definitions for the base unit connector. The rest of this section will describe the use and limits of each pin.

A1	GND	B1	*PD0	C1	GND
A2	+5V	B2	+5V	C2	+5V
A3	GND	B3	GND	C3	GND
A4	+24V	B4	+24V	C4	+24V
A5	GND	B5	GND	C5	HSC5
A6	-12V	B6	-12V	C6	-12V
A7	GND	B7	GND	C7	GND
A8	+12V	B8	+12V	C8	+12V
A9	GND	B9	GND	C9	GND
A10	Reserved	B10	Reserved	C10	Reserved
A11	FUNCT1	B11	FUNCT0	C11	*RESET
A12	FUNCT3	B12	FUNCT2	C12	*S1
A13	FUNCT5	B13	FUNCT4	C13	*S2
A14	FUNCT7	B14	FUNCT6	C14	*BUSY
A15	WDATA00	B15	RDATA00	C15	*LAM
A16	WDATA01	B16	RDATA01	C16	*X
A17	WDATA02	B17	RDATA02	C17	*Q
A18	WDATA03	B18	RDATA03	C18	CLK
A19	WDATA04	B19	RDATA04	C19	USR0
A20	WDATA05	B20	RDATA05	C20	USR1
A21	WDATA06	B21	RDATA06	C21	USR2
A22	WDATA07	B22	RDATA07	C22	Reserved
A23	WDATA08	B23	RDATA08	C23	Reserved
A24	WDATA09	B24	RDATA09	C24	Reserved
A25	WDATA10	B25	RDATA10	C25	Reserved
A26	WDATA11	B26	RDATA11	C26	Reserved
A27	WDATA12	B27	RDATA12	C27	Reserved
A28	WDATA13	B28	RDATA13	C28	Reserved
A29	WDATA14	B29	RDATA14	C29	GND
A30	WDATA15	B30	RDATA15	C30	GND
A31	SCL	B31	SDA	C31	GND
A32	GND	B32	*PD1	C32	GND

- Notes: 1) * = Active low signal
2) Reserved pins should be left unconnected

Figure 2.1 TM1000 Base Unit Connector Pinout

2.2.1 Power Supply

The TM1000 base unit is powered by an external +24V power supply. This power supply is similar to the familiar laptop computer power supply. However, this supply has a three wire connection including a separate earth ground return. This makes the TM1000 base unit earth grounded. More than one base unit can share a single power supply through the use of a power share cable.

The +24V power supply has a 2.5A rating. Each module can draw up to 2A from the +24V supply at the connector before protective circuitry in the base unit removes the supply from the module. It is up to the user in conjunction with information supplied by the module supplier to insure that each module has sufficient available capacity in a system using a shared power supply. If a module is to be designed which requires significant power, this issue should be addressed in the documentation.

The protective circuitry on the +24V supply at the module connector is also a hot swap controller. This allows modules with significant inrush current to be plugged into the base unit with the power on. The +5V logic supply is also hot swap protected at the module connector for the same reason and to prevent latchup of logic circuits being installed. However, the hot swap on the +5V is not automatic, but is controlled by the module. To turn on the +5V logic supply, the *HSC5* signal on pin C5 must be brought to +24V. This signal can be used to switch the +5V supply, but is typically just tied to +24V through a 1000 ohm resistor to activate +5V when the module is installed. The rise time of the +5V supply is appropriately controlled for hot swap purposes by circuitry on the base unit. Unlike the +24V circuit, the +5V hot swap circuitry does not provide any inherent short circuit protection. For this reason, a 2A resetable fuse exists on the +5V to the module connector. The maximum recommended module current draw on the +5V supply is 1.5A.

The +12V and -12V supplies are low current analog supplies and are not hot swap controlled. These supplies are *not* tightly regulated and have a rating of 0.5A each. They also have 2A resetable fuses in series with the module connector. Because these supplies typically do not drive logic or large inrush currents, they do not need to be turned off to insert a module with the power on. Moreover, the lack of an independent servo loop on these supplies will cause the voltage to droop if too much current is drawn thereby improving the hotswap performance.

Supply	Rating	Limiting	Hot Swap
+24V	2A	2A	Fully Automatic
+5V	1.5A	2A	Module Operated
+12V	0.5A	2A	Lack of servo feedback provides protection
-12V	0.5A	2A	Lack of servo feedback provides protection

Figure 2.2 Summary Of Module Power Supply Ratings

As described later, the logic levels provided by the base unit interface are 3.3V CMOS. These output levels are 5V TTL compatible and the inputs are 5V tolerant. However, it is common practice to provide a three terminal +3.3V linear regulator on the module from the +5V supply. This supply level is typically needed by today's high density logic designs and provides optimum interface performance.

2.2.2 Control Signals

The following is an operational summary of the module control and status signals. The output signals are 3.3V CMOS logic providing $>3.0V$ for logic 1 and $<0.4V$ for logic 0. Inputs share the same characteristics and are 5V tolerant. All inputs are pulled up to 3.3V through 4.7K ohms

**RESET Output*

The *RESET signal is low for at least 10 μS at some point shortly after power is first applied to the base unit. This signal is high at all other times and is not user controllable. It is an indication that the interface has been reset.

**S1 Output*

The *S1 signal is low for at least 500 ns when new data arrive at the WDATA lines. The data are guaranteed to be valid 200 ns before the rising edge and 200 ns after the rising edge. This signal is similar to the data write signal in microprocessor interface terms. However, module data lines are not three state. Beyond the rising edge, the WDATA lines will remain valid until another write is performed.

**S2 Output*

The *S2 signal is low for at least 500 ns while new data are being read from the RDATA lines. The module must make new data available to the interface at least 10 ns before the rising edge and must hold the data at least 2 ns after the rising edge. This signal is similar to the data read signal in microprocessor interface terms. However, module data lines are not three state. The module can drive the RDATA lines at all times, though it is not required.

**BUSY Input*

The *BUSY signal is a status input from the module. This status is returned to the host with a status read. The meaning is module design dependant. By convention this signal means that the module should not be disturbed.

**LAM Input*

The *LAM signal is a status input from the module. This status is returned to the host with a status read. The meaning is module design dependant. By convention this signal means that a special event has occurred on a module which requires attention. This convention makes it similar to an interrupt, though there is no special processing of this signal in the system. The pneumonic comes from *Look At Me*.

**X Input*

The *X signal is a status input from the module. This status is returned to the host with a status read. The meaning is module design dependant. This signal is only read on *S1 and *S2 rising edges. By convention this signal is activated when a function code relevant to the module is present on the FUNCT lines on occurrence of *S1 or *S2.

**Q Input*

The *Q signal is a status input from the module. This status is returned to the host with a status read. The meaning is module design dependant. This signal is only read on *S1 and *S2 rising edges. By convention this signal indicates a unique status point in the module as appropriate for a currently active function code. It is common to activate this signal on a read operation if more data exist to be read. Certain functions in the API can take advantage of this if the module is designed for it.

CLK

The CLK signal is a 24 MHz crystal controlled square wave clock. This clock signal is provided at 3.3V logic levels and has an independent driver. The signal is not guaranteed to be synchronous with any other signal on the module connector.

**PD0 And *PD1*

The *PD0 and *PD1 (presence detect) lines are used to detect the presence of a module. These signals should simply be grounded on the module.

2.2.3 Function And Data Signals

The following is an operational summary of the module function and data signals. The output signals are 3.3V CMOS logic providing >3.0V for logic 1 and <0.4V for logic 0. Inputs share the same characteristics and are 5V tolerant. These signals are positive true logic. The inputs are pulled up to 3.3V through 4.7K ohms.

FUNCT0 - FUNCT7

The FUNCT (function) signals are outputs to the module. The function byte is written every time a read or write cycle is performed. These signals are not three state and will remain valid between cycles. They will change state some time shortly before the next *S1 or *S2 cycle. The value written is specified by the software through the API upon initiating a read or write. The value can be taken as a command or address to the module to indicate which data should be read or written or what operation should be performed when the *S1 or *S2 arrives. In principle, these signals can also be used as general purpose discrete outputs.

WDATA00 – WDATA15

The WDATA (write data) signals are output signals to the module. The two byte word is written every time a write cycle is performed. These signals are not three state and will remain valid between write cycles. They will begin changing state sometime shortly before the next *S1 cycle and are guaranteed to be valid 200 ns before the rising edge of *S1 and at least 200 ns after the rising edge of *S1. The value is taken by the module as output data. In principle, these signals can also be used as general purpose discrete outputs.

RDATA00 – RDATA15

The RDATA (read data) signals are inputs from the module. They are pulled to +3.3V on the base unit through 4.7K ohms, so they can be left floating and will read as 1 unless driven to 0 by the module. The two byte word is read every time a read cycle is performed. These signals are not shared or three state and can be driven at any time by the module. The signals must be valid at least 10 ns before the rising edge of *S2 and at least 2 ns after the rising edge of *S2. The value is taken by the base unit and host software as input data. In principle, these signals can also be used as general purpose discrete inputs.

2.2.4 SCL and SDA

The SCL and SDA signals are connected to an I²C serial chain on the base unit. These signals are 3.3V logic and are pulled to +3.3V through 2.2K ohms as required by the I²C bus. These signals are not available for the free use of the module designer, but must be connected to a serial EEPROM of 128 or 256 bytes. The EEPROM must have device address pins and be wired as device one. Devices such as the Atmel AT24C01A and AT24C02 meet these requirements. Since the serial I/O is open drain, a 5V chip can be used, though a 3.3V chip is recommended if 3.3V is available on the module. A schematic diagram of the prototyping module is included on the development kit CD in the \DRAWINGS directory and includes the EEPROM device.

The EEPROM is accessed by the host software to automatically identify the module. The EEPROM can be programmed and read through the API and using the TMTTest software. The format of the data contained in the EEPROM is described later in this manual.

2.2.5 User Port

The user port includes three general purpose I/O pins located on the module connector at USR0, USR1, and USR2. These pins can be operated directly through the API. They can be configured individually as inputs or outputs. These signals are always pulled up to +3.3V on the base unit through 100K ohms. When configured as outputs, these signals will provide a totem poll drive at 3.3V logic levels. Upon power up, the USR lines are configured as inputs. Note that these signals may not be in the expected state when a module is swapped with the power on, so the module must be tolerant of any state on the USR lines upon insertion. Operation of the user port is discussed with the API later in this manual.

3.0 The TM1000 API

The TM1000 Application Programming Interface (API) is a multithreaded 32-bit Windows Dynamic Link Library (DLL). The functions in this library take care of the details of accessing the underlying hardware drivers and operating the TM1000 base unit. Windows DLL's can be called from most Windows software development environments, making choices about development environment for user level code virtually unlimited.

The DLL itself is located in the `\%SYSTEM_ROOT%\SYSTEM32` directory as the `TMWIN32.DLL` file. As a 32-bit Windows DLL, the API follows the C calling convention. The included C header file `TMWIN32.H` shall be used as the starting point for discussion. The available functions of the API are defined in the header file as follows.

```
DWORD TM_OpenDevice(BYTE DeviceId);
DWORD TM_CloseDevice(BYTE DeviceId);
DWORD TM_RequestOwnership(BYTE DeviceId);
DWORD TM_ReleaseOwnership(BYTE DeviceId);
DWORD TM_QueryOwnership(BYTE DeviceId);
DWORD TM_WriteWord(BYTE DeviceId, BYTE Function, DWORD *Data);
DWORD TM_WriteBlock(BYTE DeviceId, BYTE Function, BYTE Flags, DWORD DataLength, DWORD *Data);
DWORD TM_ReadWord(BYTE DeviceId, BYTE Function, DWORD *Data);
DWORD TM_ReadBlock(BYTE DeviceId, BYTE Function, BYTE Flags, DWORD DataLength, DWORD *Data);
DWORD TM_GetStatus(BYTE DeviceId, BYTE *Status);
DWORD TM_USRConfig(BYTE DeviceId, BYTE *ConfigValue);
DWORD TM_USRWrite(BYTE DeviceId, BYTE *DataValue);
DWORD TM_USRRead(BYTE DeviceId, BYTE *DataValue);
DWORD TM_GetBaseUnitRevisionLevel(BYTE DeviceId);
DWORD TM_Identify(BYTE DeviceId, char *Model, char *Vendor, char *Revision, char *Description);
DWORD TM_ProgramModuleIdentity(BYTE DeviceId, char *Model, char *Vendor, char *Revision, char *Description);
DWORD TM_ReadCalData(BYTE DeviceId, BYTE *CalData);
DWORD TM_ProgramCalData(BYTE DeviceId, BYTE *CalData);
```

Because the API is multithreaded, more than one application can use the API, a particular base unit, or a different base unit at the same time. An application simply needs to specify which base unit is to be used for a given operation using the *DeviceId* argument required with every function call. This convention requires that *DeviceId* information be determined and maintained by the user application, but allows the application to work with multiple base units.

The TM1000 system can work with up to 4 base units. Each of these units will have a *DeviceId* of 1 to 4. There is no physical way to tell which base unit has which *DeviceId*. While it is normally true that the first base unit connected will have a *DeviceId* of 1 and so on, the only reasonable way to proceed is to scan all *DeviceId*'s looking for base units and modules of interest. As described in section 4, using the TM1000 System Manager removes this task from the module user interface application design.

Most of the functions in the API return a success or failure result where a 1 indicates success and a 0 indicates failure. In the following discussion, this should be considered the case unless noted otherwise.

The following standard conventions are followed for arguments and return values. A `DWORD` is an unsigned 32-bit fixed point number. A `BYTE` is an unsigned 8-bit fixed point number. A `char` is a two's-complement signed 8-bit fixed point number used to represent ASCII characters, where `char*` is the convention used for a pointer to an array of characters forming a NUL terminated character string. The term "byte" can refer to a `BYTE` or a `char` value.

3.1 Opening A Path To A TM1000 Base Unit

Before a base unit can be used, a communications path must be established. This is accomplished by passing the *DeviceId* to the *TM_OpenDevice* function. The DLL will attempt to open a path to the base unit specified. The function will fail if no such base unit is connected. If the function succeeds, the DLL will maintain a handle to the base unit specified for use on future function calls.

When a user application has completed all calls to the API for a particular device, it should close the handle to the device by passing the *DeviceId* to the *TM_CloseDevice* function. If the specified *DeviceId* was already closed, no error is indicated as the intended state was reached.

Typically, *TM_OpenDevice* is called when the user application is launched and *TM_CloseDevice* is called when the user application is closed. These functions do not contain user level indications of errors, such as message boxes, in order to make it is possible to attempt to open a *DeviceId* for which there is no connected base unit while scanning the system.

3.2 Sharing A Base Unit

The TM1000 API does not inhibit more than one application from using a single base unit. There are reasons this might be necessary. For example, the TM1000 System Manager continuously monitors the health of each base unit even while a user application is working with an installed module. There may also be other module specific reasons why more than one application process may be involved in the operation of a module. The API does utilize standard multitasking techniques to prevent requests from one application from interfering with those from another application.

As a means of providing information about system wide usage of a particular base unit, an application can set a flag in a shared memory area within the DLL to indicate that it claims ownership of that unit. This is a polite sharing protocol. Obtaining this ownership does not prevent other applications from using the base unit, it only indicates that the base unit is in use in some way which it might be better not to disturb. It is, for example, more similar to a "do not disturb" sign than it is to a lock.

To see if another application has claimed ownership of a base unit, the *DeviceId* is passed to the *TM_QueryOwnership* function. A 1 is returned if an application has claimed ownership of the base unit with the given *DeviceId*. A 0 is returned if no application currently claims ownership of this base unit.

To request ownership of a base unit, the *DeviceId* is passed to the *TM_RequestOwnership* function. If another application has already claimed ownership of the requested *DeviceId*, the function will return 0. Otherwise a 1 will be returned indicating that the calling application has been granted ownership.

When an application no longer requires ownership of a base unit, it should release any ownership it has claimed by passing the *DeviceId* to the *TM_ReleaseOwnership* function. This function always succeeds. Again, this is a polite sharing protocol. Any application can release ownership of any *DeviceId*, but it is expected to do so only for a *DeviceId* for which it has previously been granted ownership.

User applications will typically attempt to obtain ownership when they are launched and will release any obtained ownership when they are terminated. If the application is standalone, it is possible to ignore ownership issues altogether. However, even in such cases it might be convenient to use ownership to identify if the user has already begun working with the module through another instance of the application. See section 4 for more information about ownership and how it is used in TM1000 system programming conventions.

3.3 Singular Read And Write Functions

The *TM_WriteWord* and *TM_ReadWord* functions act to perform a single write or read cycle on a module. Both functions will operate on the base unit with the specified *DeviceId*. Both functions will install the function code specified in the *Function* argument on FUNCT0-FUNCT7 lines at the module before the cycle starts.

In both functions the *Data* argument is a pointer to a 32-bit unsigned fixed point number. These functions will read or write the memory location pointed to by this argument. The *TM_WriteWord* function collects the argument and writes it to the WDATA0-WDATA15 lines causing the completion of an S1 cycle. The *TM_ReadWord* function reads the RDATA0-RDATA15 lines causing the completion of an S2 cycle and returns the value read in the argument. Only the lower 16 bits of the argument value are used. The *TM_ReadWord* function sets the upper 16 bits to zero.

3.4 Block Read And Write Functions

The *TM_WriteBlock* and *TM_ReadBlock* functions act to a perform block write or read on a module. Both functions will operate on the base unit with the specified *DeviceId*. Both functions will install the function code specified in the *Function* argument on FUNCT0-FUNCT7 lines at the module before the first cycle starts. This function code is written only once. A total of *DataLength* cycles are performed unless terminated early by a condition specified in the *Flags* argument.

In both functions the *Data* argument is a pointer to a 32-bit unsigned fixed point number. These functions will read or write the memory location pointed to by this argument on the first cycle. Each successive cycle will read or write following locations. The argument should point to an array or an allocated buffer of sufficient size to hold *DataLength* words. If there is not sufficient memory at the end of the *Data* pointer, these functions will fail catastrophically.

The *TM_WriteBlock* function collects data from successive locations of the argument array and writes them to the WDATA0-WDATA15 lines causing the completion of an S1 cycle for each word written. The *TM_ReadWord* function reads the RDATA0-RDATA15 lines causing the completion of an S2 cycle for each word read and returns each word read to successive locations in the argument array. Only the lower 16 bits of the values in the argument array are used. The *TM_ReadWord* function sets the upper 16 bits to zero.

The *Flags* argument is a bitwise value which can be used to further refine the block transfer process. Both functions require a *Flags* argument, though the current version of the API uses flags only in the *TM_ReadWord* function. The flag values are defined in the header file as *TM_FLAG_* constants. These values can be added or logically OR'ed together to use more than one of the refinements.

To perform no added data processing, use *TM_FLAG_NONE*. To set the upper 8 bits of the returned words to zero, use *TM_FLAG_MASK_MSB*. To set the lower 8 bits of the returned words to zero, use *TM_FLAG_MASK_LSB*.

The flag *TM_FLAG_Q_STOP* causes a block read to stop when the Q status line from the module goes false (logic high). This flag is useful for modules designed to set Q true (logic low) when there is data to be read and return Q to false when all data has been read. The caveat is that Q must be true before the first cycle is performed for any data to be transferred. Since Q changes state on an S1 or S2 cycle, the module and software must be designed to provide a Q status with this meaning during some previous cycle.

The *DataLength* specified for each block transfer must be less than or equal to the *TM_MAX_LENGTH* constant specified in the header file. The return value is the number of words actually read or written. This number will nominally be equal to *DataLength* if the operation was successful. It will be zero or short if the function fails. The exception is the *TM_ReadWord* function with the *TM_FLAG_Q_STOP* flag set. In this case, a short transfer may well be expected and the return value will indicate the number of words the module had available for transfer. If this number is equal to *DataLength*, the Q status did not go false before the last word was transferred.

3.5 Status Byte

The function *TM_GetStatus* will return a status byte from the base unit with the specified *DeviceId*. The status byte is returned by reference in the *Status* argument. That is, the *Status* argument is a pointer to an 8-bit unsigned fixed point number which shall be taken as a bitwise set of status flags. The number returned by value is a pass/fail indication. The status byte has the bit assignments shown in Figure 3.1 below. The flags are all positive true (1 equals true, set, or active) regardless of their hardware true state.

PD1	PD0	T	C	LAM	Q	X	BUSY
-----	-----	---	---	-----	---	---	------

Where 1 means:

PD1, PD0 = module presence detected, both should be 1 to accept

T = the space under the module is getting hot (> 40 °C)

C = excessive current is being drawn by this module

LAM = LAM signal active

Q = Q signal active on last S1 or S2

X = X signal active on last S1 or S2

BUSY = BUSY signal active

Figure 3.1 Status Byte Bit Assignments

3.6 User Port Operations

The user port is a set of three completely configurable discrete I/O points on the module connector at USR0, USR1, and USR2. These points are configured, read, and written from three 3-bit registers accessed with the API functions *TM_USRConfig*, *TM_USRWrite*, and *TM_USRRead*. The data arguments in these API functions are all pointers to an unsigned data byte to be read or written. The functions all act on the module at the specified *DeviceId*. The user port is shown schematically in Figure 3.2 below.

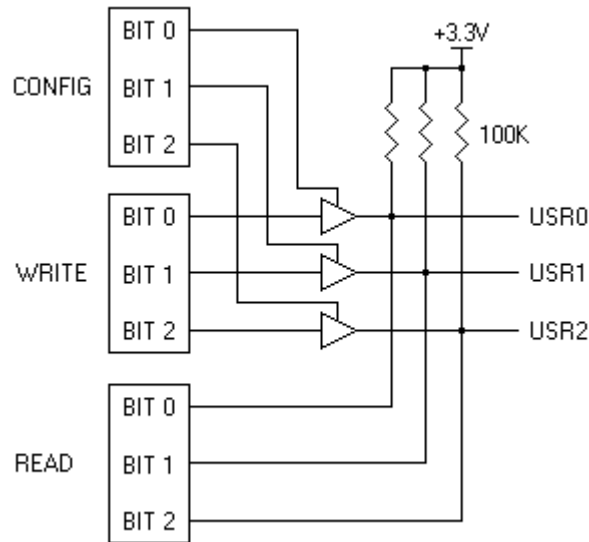


Figure 3.2 User Port Schematic Diagram

The *TM_USRConfig* function writes a value to the CONFIG register. This value enables a totem poll driver for any of the three USR lines as shown in the figure. The *TM_USRWrite* function writes to the WRITE register, setting the logic state of the USR lines which have been enabled by the CONFIG register. The *TM_USRRead* function reads the READ register, returning the current state of the USR lines.

If a USR line is configured to be driven by the base unit, it is considered to be an output and should not be driven by the module. If a USR line is not configured to be driven by the base unit, it is considered to be an input and can be driven by the module. The value read at the READ register will be the value on the USR lines, regardless of whether the base unit or the module is driving the lines. If nothing is driving a USR line, the line will read as 1.

Note that the USR lines can in principle be used as open drain outputs as well. This could allow a wire-OR connection with module signals. To do this, write zeros to the WRITE register bits and control the data outputs by writing inverted bits to the CONFIG register. A USR line will be low when the output is enabled and high through the passive pullup when the output is disabled.

3.7 Hardware Identification

There are provisions in the TM1000 design to allow the base unit and the installed module to be identified by a host application. This section discusses methods of accessing this information from the API.

The base unit is identified only by a revision level. This value can be retrieved by passing a *DeviceId* to the *TM_GetBaseUnitRevisionLevel* function. The revision level is returned by value. This is a simple integer revision level which will only change when there are changes to the base unit which would affect the API. At the release of this manual, there is only one version of the base unit. As such this function will always return a 1 to successfully indicate revision level 1 or a 0 to indicate a failure.

The module is identified with data held in a small serial EEPROM on each module. The contents of this EEPROM may be read with the *TM_Identify* function. The contents may be written with the *TM_ProgramModuleIdentity* function. These functions accept a *DeviceId* and a set of NUL terminated strings. These strings write or return the contents of the data structure held in the EEPROM. Using strings simplifies the call for some programming environments versus establishing a complex data structure as an argument. The down side is that sufficient memory must be allocated for each string individually. The data structure is shown in Figure 3.3 below. The pointers passed in the arguments to these functions must point to an array of at least the required number of bytes or the function call will fail catastrophically.

Field	Description	Maximum Characters	Bytes Required
Model	Module model number	8	9
Vendor	Vendor name	24	25
Revision	Revision level for module	5	6
Description	Descriptive text describing the module	50	51

Figure 3.3 Module Identification Data Structure

It is important to realize that in the use of these functions all required bytes will be programmed and all required bytes will be read. A NUL terminated string is defined to end at the first NUL character. The characters after the NUL could be any value and will be transferred though they should not be used by an application.

3.8 Module Calibration Data

In some module designs it is desirable to maintain a small amount of data permanently with the module hardware. With version 1.30 and greater of the API a provision is made to store a 128 byte field of undefined “calibration data” in the upper 128 bytes of a 256 byte EEPROM on the module. This is the same EEPROM for which the lower 128 bytes are used to store module identification information as described in 3.7 above. The meaning of the calibration data bytes is defined by the module application. They are generally used to hold calibration or extended configuration information.

The API functions *TM_ReadCalData* and *TM_ProgramCalData* allow the user application to read and write all 128 bytes to and from an array addressed by the **CalData* argument. This pointer must point to an array of at least the required number of bytes or the function call will fail catastrophically.

To use the calibration data area, the module must have a 256 byte EEPROM device such as the Atmel AT24C02. It is only necessary to install the larger EEPROM if the application calls for the use of these bytes. If a 128 byte device is used instead, these functions should not be used and simply won't work properly.

4.0 TM1000 Programming Conventions

While the TM1000 API does not in itself create any restrictions on programming style or techniques, there are some conventions which should be followed for satisfactory operation of the TM1000 system and some which must be followed for an application to work properly with the TM1000 System Manager. This section discusses these issues.

4.1 TM1000 System Manager

The TM1000 System Manager is the application which launches from the TM1000 shortcut icon and can be identified as TM1000.EXE in the TM1000 installation directory. The familiar window with four buttons opens in the upper right corner of the screen. This little program takes a lot of the work out of module application design if conventions are followed to work with it. Here is how it works.

The system manager is constantly scanning for the presence of base units and modules. If a new base unit and/or module is detected, only then does it scan for the module identity. It can detect any changes in installed modules and alert the user of base unit environmental conditions. By working with the system manager, this functionality does not need to be implemented in the module user interface software design.

For the system manager, the model number in the module identity data structure has substantial meaning. This is the string placed on the start button in the system manager window. It is also the base name for the executable used to operate the module. When the start button is clicked, the system manager will attempt to launch *<Model>.exe* from the TM1000 installation directory or the current system path. To work properly with the system manager, the module user interface executable must follow this naming convention.

The system manager will pass the *DeviceId* for the base unit in which the module is installed to the module user interface executable as a single digit text string command line argument when launching the executable. That is, the system manager will attempt to launch the command line *<Model>.exe <DeviceId>*. This eliminates any requirement for the module user interface to identify the module or to determine at which *DeviceId* it is located. If the *DeviceId* is not present, the application was probably started by someone not using the system manager. In this case a polite error message should appear and the application should terminate.

As the system manager polls the base units for status, it also polls the ownership status of each connected base unit. If an application has claimed ownership of a base unit, the button for that base unit is disabled. It is convention for the module user interface application to declare ownership of the base unit of the given *DeviceId* when the application is launched. This will cause the launch button to gray and prevent the user from launching the application again. When the application closes, it should release ownership of the base unit.

4.2 Application Shutdown Considerations

For the TM1000 system drivers to work properly, an application must close any open *DeviceId* when it shuts down. It should also not attempt to maintain a *DeviceId* open when the base unit is turned off or disconnected. If the application detects a failure in a poll for status, the base unit has probably been removed. This should cause the application to close the *DeviceId* and stop using the API. In the case of a module user interface application, it is recommended that the application proceed with normal shutdown procedures.

If the module application was launched by the system manager, the shutdown is taken care of automatically by the system manager. The system manager itself keeps running until closed by the user, but closes any *DeviceId* it may have opened which is no longer responsive and shuts down associated applications. If the application does not use the system manager, it should contain logic to perform the shutdown. If the *DeviceId* is kept open, then if the base unit is reconnected it will take on a different *DeviceId*. This can cause confusion at least and at worst can cause the driver to jam when it runs out of *DeviceId*'s to assign.

4.3 Development Considerations

While the debugging features of many development environments are a useful tool, it is recommended that they be avoided when the application is ready to call the API. This is because most development environments will hold onto a DLL when it is opened by running in the debugger. In this case, it is the development environment and not the application under development that opens the DLL. While in principle this should not be a problem unless you are planning on changing the DLL, the fact is it usually is a problem particularly when multiple threads are involved. The recommended practice is to get the application close to ready through dry run debugging and then add the API calls later. That way it will also not be necessary to have a TM1000 system always running on the development computer. When the application is ready to connect to the TM1000 system, it should be operated from an end product executable.

5.0 Using The TMTTest Application

The TMTTest application provides the developer with a means to perform direct low level operations on an installed base unit and module. This can be helpful while getting started with the TM1000 system and for use by the hardware developer testing aspects of a new module. The application can be installed by running SETUP.EXE from the \TMTTEST directory on the development kit CD.

The TMTTest application operates independently of the TM1000 System Manager and can run at the same time as any other TM1000 application. It does not declare ownership of any base unit *DeviceId*. When the TMTTest.exe executable is launched, the dialog window shown in Figure 5.1 will appear.

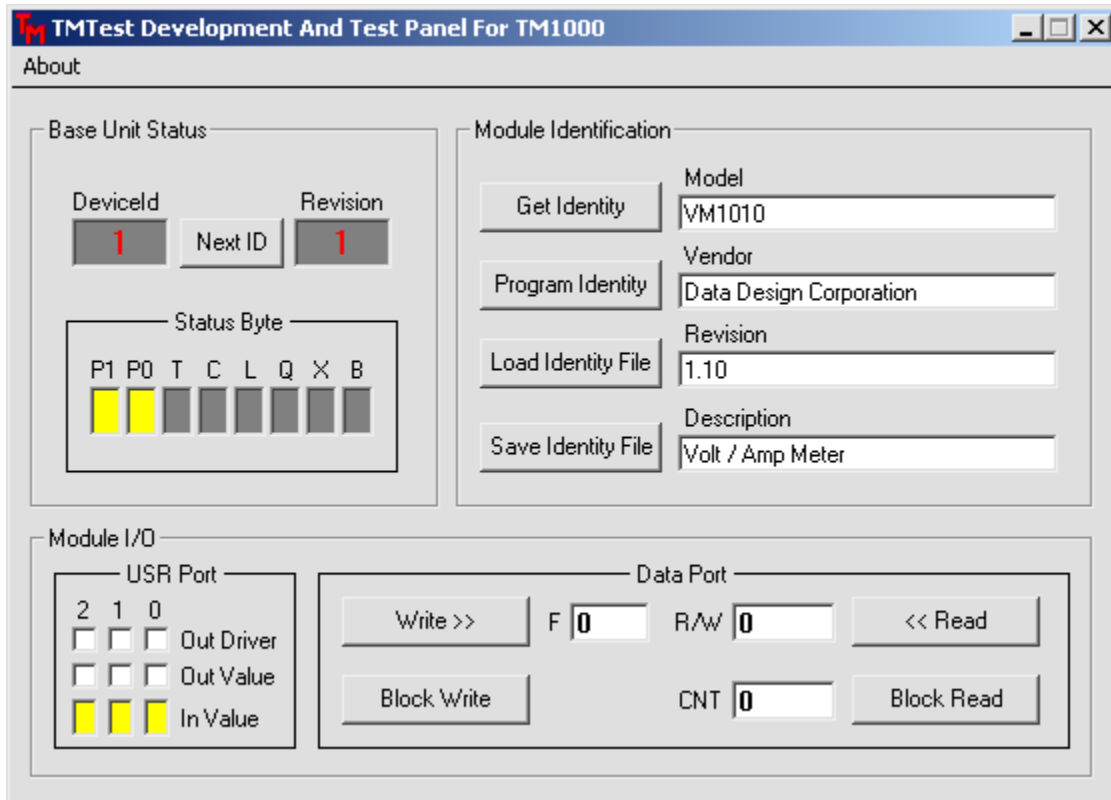


Figure 5.1 TMTTest Main Window

5.1 Base Unit Status

The controls in the *Base Unit Status* frame provide information about the connected base unit. All other controls in the TMTTest dialog will affect the base unit with the *DeviceId* shown. When the application is first started, it will search for the first available base unit. If no base units are available, the application will gracefully terminate.

If more than one base unit is connected to the host, additional base units can be addressed by clicking the *Next ID* button. This will begin the search again starting with the *DeviceId* one greater than that shown and returning from 4 to 1 as required. When any base unit is found, its revision level is read and displayed in the *Revision* box.

The *Status Byte* display is updated every half second with the status of the currently addressed base unit. The status bits are displayed as small lights which are yellow when a status bit is active and gray when it is not active. The status bits correspond directly to those presented in section 3.5 above.

5.2 Module Identification

The *Module Identification* frame provides a direct interface for working with the module identification EEPROM on the installed module. The strings correspond to those found in the module identification structure discussed in section 3.7 above. These controls provide an easy way for the developer to program the identification into the module EEPROM. If the system manager is used in the final software design, then between this application and the system manager it should never be necessary to write code to access the identification EEPROM.

Click the *Get Identity* button to read the current contents of the EEPROM on the installed module. This action will occur automatically when the application is started or a new base unit is addressed. If there is no module or the system is otherwise unable to read the identification information, "N/A" will be displayed in each field.

Type in a new identity for the installed module, taking into account the limits identified in section 3.7 above. These limits are enforced to some extent in the TMTTest application and are fully enforced by the API call. When the identity information is as desired, click the *Program Identity* button to install the new identity in the module. Obviously doing this has a dramatic effect on system operation, so it should be done carefully.

When a write operation is completed, the application reads the identity back and displays what it read. If the strings did not change, the write was successful. If the read back failed, "N/A" would appear. If the write failed, the text may be different. Most often differences would appear in text length or leading and following spaces. Leading and following spaces are always removed by the application before writing. The API does not make this adjustment.

The module identity can be saved in a text file for later retrieval by clicking the *Save Identity File* button. A standard file dialog will appear for completion of the process. The file is a normal text file which can be edited with a text editor. There is one string to each line in the order they appear in the dialog. The contents of the file can be recalled to the application with the *Load Identity File* button. Using an identity file can reduce the probability of mistakes if more than one module is to be programmed with the same information. This would be the case in a module production environment.

5.3 Module I/O

The controls in the *Module I/O* frame allow the user to directly operate the module interface. This is an invaluable tool for debug and test of module hardware and for tracking down subtle software bugs by comparison with manual operations.

The controls labeled *User Port* allow direct operation of the user port lines. The read register bits are displayed at *In Value*. These indicators are yellow when a bit is 1, gray when a bit is 0, and are refreshed every half second. The configuration register is controlled with the check boxes at *Out Driver*. A checked box will enable the corresponding output driver. The write register values themselves are controlled by the check boxes at *Out Value*. A checked box sets the corresponding output bit to 1.

The controls labeled *Data Port* allow direct operation of the main module interface bus. The function code to be written with any operation is entered at the box labeled *F* as a single byte (two character) hexadecimal value. Clicking the *Read* button at this point will perform an S2 cycle and present the data read from the RDATA lines in the *R/W* box as a two byte (four character) hexadecimal value. To write a word to the module, enter the value in the *R/W* box in hexadecimal and click the *Write* button. This will produce an S1 cycle with the given value on the WDATA lines.

The TMTTest application allows a block transfer to be performed for timing and demonstration purposes, but does not make an effort to look at individual data words in the array transferred. To perform a block transfer, enter the number of words to be transferred in the *CNT* box as a three character hexadecimal value less than or equal 800H. Click the *Write Block* button to perform the requested number of S1 cycles using the data in the *R/W* box through the block write API call. Click the *Read Block* button to perform the requested number of S2 cycles through the block read API call. The first word read will be displayed in the *R/W* box upon completion of the transfer.