DALLAS SEMICONDUCTOR

Application Note 155
1–Wire® Software Resource Guide

www.maxim–ic.com

INTRODUCTION

There are over 30 different 1–Wire devices including iButtons® that Dallas Semiconductor currently produces. Navigating the available API's, software examples, and other resources to communicate with this array of devices or finding the correct resource for a single device type can be a daunting task. This guide provides an overview of the available resources and a selection guide. All of the API's described in this document are free to use without restriction and in most cases include the complete source code.

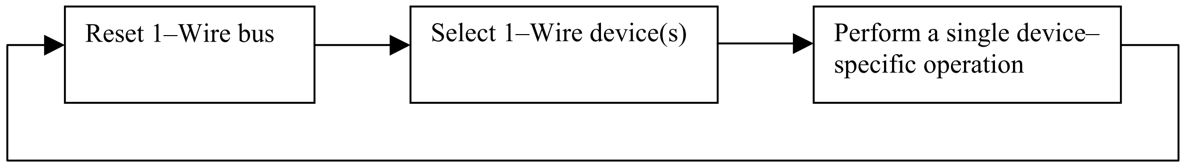
1–WIRE OVERVIEW

The Dallas Semiconductor 1–Wire bus is a simple signaling scheme that performs two–way communications between a single master and peripheral devices over a single connection. A powerful feature that all 1–Wire bus devices share is that each and every device, in a chip or an iButton, has a factory–lasered serial number that will never be repeated in any other device. That is to say, every device is unique. This allows any single device to be individually selected from among many that can be connected to the same bus wire. Because one, two, or even dozens of 1–Wire devices can share a single wire for communications, a binary searching algorithm is used to find each device in turn. Once each device serial number is known, any device can be uniquely selected for communication using that serial number to address it.

The first part of any communication involves the bus master issuing a "reset" which synchronizes the entire bus. A slave device is then selected for subsequent communications. This can be done by selecting all slaves, selecting a specific slave (using the serial number of the device), or by discovering the next slave on the bus using a binary search algorithm. These commands are referred to collectively as "network" or ROM (Read–Only–Memory) commands. Once a specific device has been selected, all other devices drop out and ignore subsequent communications until the next reset is issued.

Once a device is isolated for bus communication the master can issue device–specific commands to it, send data to it, or read data from it. Because each device type performs different functions and serves a different purpose, each has a unique protocol once it has been selected. Even though each device type may have different protocols and features, they all have the same selection process and follow the command flow as seen in Figure 1.

TYPICAL 1–WIRE COMMUNICATION FLOW Figure 1



An integral part of the unique serial number in each slave is an 8–bit family code. This code is specific to the device model. Because each device model performs different functions, this code is used to select the protocol that will be used to control or interrogate it. See Table 1 for a mapping of family codes to Dallas Semiconductor part numbers.

FAMILY CODE REFERENCE Table 1

Family Code	Part Number () iButton Package	Description (Memory size in bits unless specified)
01 (hex)	(DS1990A)*, DS2401	1–Wire net address (serial number) only
02	(DS1991), DS1425	MultiKey iButton, 1152–bit secure memory
04	(DS1994), DS2404	4K NVRAM memory and clock, timer, alarms
05	DS2405	Single addressable switch
06	(DS1993)	4K NVRAM memory
08	(DS1992)	1K NVRAM memory
09	(DS1982), DS2502	1K EPROM memory
0A	(DS1995)	16K NVRAM memory
0B	(DS1985), DS2505	16K EPROM memory
0C	(DS1996), (DS1996x2), (DS1996x4)	64K to 256K NVRAM memory
0F	(DS1986), DS2506	64K EPROM memory
10	(DS1920), DS1820, DS18S20	Temperature with alarm trips
12	DS2406, DS2407	1K EPROM memory, 2 channel addressable switch
14	(DS1971), DS2430A	256–bit EEPROM memory and 64–bit OTP register
18	(DS1963S)	4K NVRAM memory and SHA–1 engine
1A	(DS1963L)	4K NVRAM memory with write cycle counters
1D	DS2423	4K NVRAM memory with external counters
1F	DS2409	2 channel addressable coupler for sub–netting
20	DS2450	4 channel A/D
21	(DS1921), (DS1921H), (DS1921Z)	Thermochron temperature logger
22	DS1822	Econo–Temperature
23	(DS1973), DS2433	4K EEPROM memory
24	(DS1904), DS2415	Real–time–clock
26	DS2438	Temperature, A/D
27	DS2417	Real–time–clock with interrupt
28	DS18B20	Adjustable resolution temperature
2C	DS2890	Single channel digital potentiometer
30	DS2760	Temperature, current, A/D
33	(DS1961S), DS2432	1K EEPROM memory with SHA–1 engine
91	(DS1981)	512–bit EPROM memory (Uniqueware only)
96	(DS1955), (DS1957B)	Java™–powered Cryptographic iButton (64K–byte ROM, 6 to 134K–byte NVRAM)

API FUNDIMENTALS

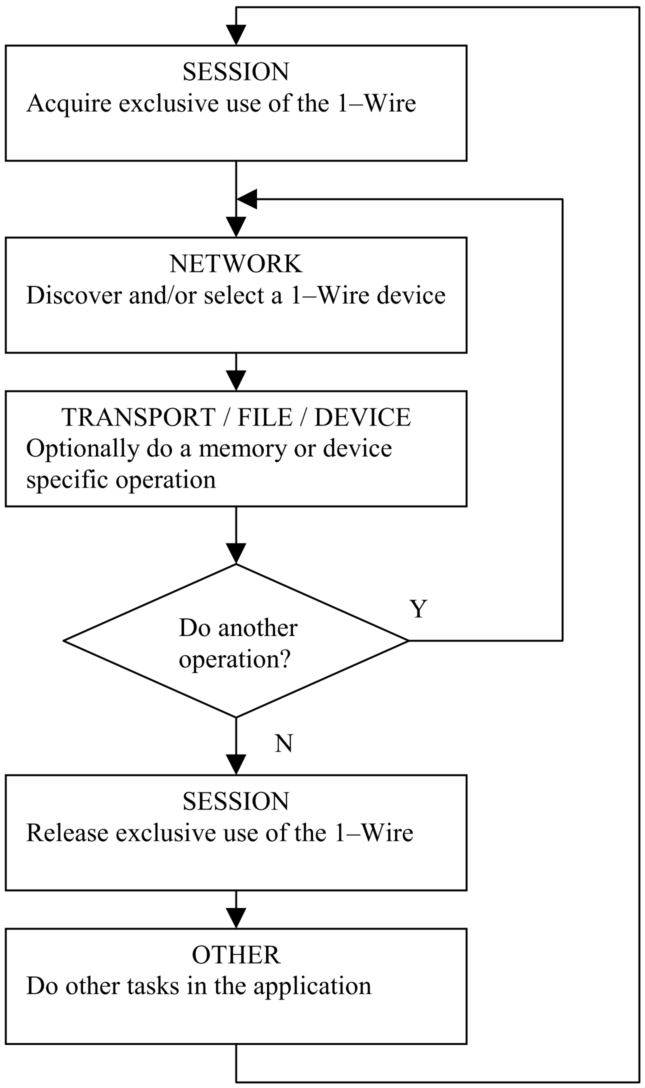
The different API's for communicating with 1–Wire devices have common features that reflect the fundamental communication issues arising from the protocol. Figure 2 outlines the common groupings of the functions for the different API's. Since most 1–Wire devices have memory, the memory IO functions are treated as a common API group even though the functions do not apply to all devices. All other non–memory specialty functions are lumped into the device–specific DEVICE grouping.

API FUNCTION GROUPINGS Figure 2

SESSION
Negotiate exclusive use of the 1–Wire bus. This is particularly important on operating systems or environments where several processes or threads could simultaneously want access to the same 1–Wire bus. Exclusive use of the network is required when doing multiple operations on a single device that must not be interrupted.
LINK
Primitive 1–Wire bus communication functions. All 1–Wire communication can be condensed down to the 'reset' that resets all devices and reading and writing bits. This group can also contain functions to set the electrical characteristics of the bus such as when providing special EPROM programming pulses or power delivery.
NETWORK
Network functions for device discovery and selection. The unique serial number lasered into each 1–Wire device is used as its network address. These functions can be constructed from the LINK level functions. Datasheets for 1–Wire devices refer to these as ROM commands since the serial number is **R**ead–**O**nly–**M**emory. Some 1–Wire masters contain built-in network functions that are more efficient then constructing them with the link functions.
TRANSPORT
Block communication and primitive read/write memory functions. It can also include packet read/write memory functions. These functions are constructed from the NETWORK and LINK group functions.
FILE
File memory level functions using the 1–Wire File Structure (see Application Note 114). These functions are constructed from the NETWORK and TRANSPORT level functions. Only useful for devices with more then one page of memory.
DEVICE
Device-specific 'high–level' functions. These functions are often constructed from the NETWORK, TRANSPORT, and LINK group functions and perform operations such as reading temperature values or setting a switch state.

The typical sequence to use these functions is outlined in Figure 3. The 'session' functions wrap around the communication calls to the device, which typically involve using a 'network' function followed by a memory or 'device' specific operation.

API USAGE FLOW Figure 3


SESSION
Acquire exclusive use of the 1–Wire
NETWORK
Discover and/or select a 1–Wire device
TRANSPORT / FILE / DEVICE
Optionally do a memory or device specific operation
Do another operation?
Y
N
SESSION
Release exclusive use of the 1–Wire
OTHER
Do other tasks in the application

The nature of iButton communication is inherently 'touch'. This means that contact with the device is not always reliable. The iButton might be inserted into the reader and bounce around during the read. Consequently a consistent methodology of error recovery must be rigorously followed. This usually entails doing retries when a spurious error is detected and utilizing CRC checks in data communication. The File IO functions in the API's use a standard file structure detailed in Application Note 114 '1–Wire File Structure'. This structure uses a CRC16 on every page of data to quickly verify the validity of the data being read. Most of the 1–Wire API functions have little or no automatic retries. The retries are under application control. See the Application Note 159 "Ultra Reliable 1–Wire Communication" for methodology of error recovery and risk assessment in doing 1–Wire communication.

API SELECTION

There are principally four different Application Program Interfaces (API) that are considered by this document. The API's operate on different platforms, use different languages, and have different capabilities. Table 2 displays the four API's with a brief description.

API DESCRIPTIONS Table 2

API	Abbreviation	Description
1–Wire Public Domain	PD	Complete open source C code that primarily supports the serial DS9097U adapter on many platforms.
1–Wire API for Java	OWAPI	Complete open source Java code that supports almost ALL 1–Wire devices. The only non–native 1–Wire master supported is the DS9097U serial adapter.
1–Wire COM	OWCOM	Windows Component Object Model (COM) wrapper for OWAPI. Accessible from standard languages and scripting languages like Java script and Visual Basic Script.
TMEX API	TMEX	Supports all 1–Wire master adapters on Windows 32–bit platforms. Provides link and file IO functions but no device functions. Drivers are closed source. This API called by other API's to get access to all of the 1–Wire adapter types.

Table 3 maps the operating system with the available API's divided by language. *Note that 'TINI' is an embedded platform with a Java-based OS made by Dallas Semiconductor http://www.ibutton.com/TINI.

API OPERATING SYSTEM AND LANGUAGE COVERAGE Table 3

Language / OS	Windows Language Independent	C	Java
Windows 2000	TMEX / OWCOM	PD	OWAPI
Windows ME	TMEX / OWCOM	PD	OWAPI
Windows 98	TMEX / OWCOM	PD	OWAPI
Windows 95	TMEX	PD	OWAPI
Windows XP	(TMEX) / (OWCOM)	(PD)	(OWAPI)
Win3.1		PD	
DOS		PD	
PALM		PD	
VISOR		PD	
PocketPC		PD	
Solaris		(PD)	OWAPI
Linux		PD	OWAPI
TINI*		(PD) – w/o TINI OS	OWAPI
( ) – support planned but not yet complete			

The support of the individual device families also varies from API to API. Table 4 lists all of the currently available 1–Wire devices with flags indicating the available support in each API. The flags 'key' is at the bottom of the table. Note that the device cells without shading are considered fully supported by the API. A light shaded cell indicates partial support and dark shaded cell indicates minimal support.

API SUPPORT BY DEVICE Table 4

Device	FC	Description	TMEX	PD	OWAPI	OWCOM
DS1425	02	MultiKey iButton, 1152–bit secure memory	AB	AB**I**	ABC	ABC
DS1427	04	4K NVRAM memory and clock, timer, alarms	ABDE	ABCDE **I**	ABCDE FGHI	ABCD**E** **FGHI**
DS1820	10	Temperature and alarm trips	AB	ABI	ABCI	ABCI
DS18B20	28	Adjustable resolution temperature	AB	AB	ABI	ABI
DS18S20	10	Temperature and alarm trips	AB	ABI	ABCI	ABCI
DS1981	91	512–bit EPROM memory (Uniqueware only)	ABCE	AB	AB	AB
DS1982	09	1K EPROM memory	ABCE	ABCDE	ABCDE	ABCD**E**
DS1985	0B	16K EPROM memory	ABCE	ABCDE	ABCDE	ABCD**E**
DS1986	0F	64K EPROM memory	ABCE	ABCDE	ABCDE	ABCD**E** **FGH**
DS1904	24	Real–time–clock	AB	AB	ABI	ABI
DS1920	10	Temperature and alarm trips	AB	ABI	ABCI	ABCI
DS1921 DS1921H DS1921Z	21	Thermochron temperature logger	ABDE	ABCDE I	ABCDE FGHI	ABCD**E** **FGHI**
DS1955 DS1957	96	Java Powered Cryptographic iButton (64K–bytes ROM, 6 to 134K–bytes NVRAM)	AB	ABI	ABI	ABI
DS1961S	33	1K EEPROM memory with SHA–1 engine	ABDE	ABCDE I	ABCDE FGHI	ABCD**E** **FGHI**
DS1963L	1A	4K NVRAM memory with write cycle counters	ABDE	ABCDE	ABCDE FGH	ABCD**E** **FGH**
DS1963S	18	4K NVRAM memory and SHA–1 engine	ABDE	ABCDE I	ABCDE FGHI	ABCD**E** **FGHI**
DS1971	14	256–bit EEPROM memory and 64–bit OTP register	ABD	ABCDI	ABCDI	ABCDI
DS1973	23	4K EEPROM memory	ABDE	ABCDE	ABCDE FGH	ABCDE **FGH**
DS1990A	01	1–Wire Address only	AB	AB	AB	AB
DS1991	02	MultiKey iButton, 1152–bit secure memory	AB	AB**C**	ABC	ABC
DS1992	08	1K NVRAM memory	ABDE	ABCDE	ABCDE FGH	**ABCD**E FGH
DS1993	06	4K NVRAM memory	ABDE	ABCDE	ABCDE FGH	ABCD**E** **FGH**
DS1994	04	4K NVRAM memory and clock, timer, alarms	ABDE	ABCDE **I**	ABCDE FGHI	ABCD**E** **FGHI**
DS1995	0A	16K NVRAM memory	ABDE	ABCDE	ABCDE FGH	ABCD**E** **FGH**

DS1996	0C	64K NVRAM memory	ABDE	ABCDE	ABCDE FGH	ABCD**E** **FGH**
DS1996x2 DS1996x4	0C	64K*2 (128K) NVRAM memory 64K*4 (256K) NVRAM memory	ABDE	ABCDE	ABCDE FGH	ABCD**E** **FGH**
DS2401	01	1–Wire Address only	AB	AB	AB	AB
DS2405	05	Single switch	AB	ABI	ABI	ABI
DS2404	04	4K NVRAM memory and clock, timer, alarms	ABDE	ABCDE **I**	ABCDE FGHI	ABCD**E** **FGH**I
DS2406 DS2407	12	1K EPROM memory, 2 channel addressable switch	ABCE	ABCDE I	ABCDE I	ABCD**E** I
DS2409	1F	dual switch, coupler	AB	ABI	ABI	ABI
DS2415	24	Real–time–clock	AB	AB	ABI	ABI
DS2417	27	Real–time–clock with interrupt	AB	AB	ABI	ABI
DS2423	1D	4K NVRAM memory with external counters	ABDE	ABCDE I	ABCDE FGHI	ABCD**E** **FGHI**
DS2430A	14	256–bit EEPROM memory and 64–bit OTP register	ABD	ABCDI	ABCDI	ABCDI
DS2432	33	1K EEPROM memory with SHA–1 engine	ABDE	ABCDE I	ABCDE FGHI	ABCD**E** **FGH**I
DS2450	20	quad A/D	AB	ABI	ABI	ABI
DS2438	26	Temperature, A/D	AB	ABI	ABCI	**ABCI**
DS2502	09	1K EPROM memory	ABCE	ABCDE	ABCDE	ABCD**E**
DS2505	0B	16K EPROM memory	ABCE	ABCDE	ABCDE	ABCD**E**
DS2506	0F	64K EPROM memory	ABCE	ABCDE	ABCDE	ABCD**E** **FGH**
DS2760	30	Temperature, current, A/D	AB	AB	ABI	ABI
DS2890	2C	single channel digital potentiometer	AB	AB	ABI	ABI

Support Shading Guide

☐ Full Support
░ Partial Support
▒ Minimal Support

Support Flags

A. 1–Wire link primitive support
B. 1–Wire network support
C. Transport memory byte read/write support
D. Transport memory packet read/write support
E. 1–Wire File Structure Type AA support (see Application Note 114 for File Structure types)
F. 1–Wire File Structure Type AB support
G. 1–Wire File Structure Type BA support
H. 1–Wire File Structure Type BB support
I. Other device–specific support

BOLD Flags

Indicates the current pre–release version of this API does not yet support this feature flag but it will be supported in the final release.

1–WIRE PUBLIC DOMAIN (PD) OVERVIEW

The functions provided in 1-Wire Public Domain API are completely written in 'C' and are intended to be used on platforms not supported by the TMEX API. The '1–Wire net' (or MicroLAN) is a one wire and ground network with one master and one or more slave devices. This API creates a 1–Wire master that can be used to identify and communicate with slave devices. It provides all of the 1–Wire, transport and file level services to communicate with all of the Dallas Semiconductor 1–Wire devices including iButtons.

The location of this API kit and sample platform builds is:
http://www.ibutton.com/software/1wire/wirekit.html

The 'C' source code to this API is designed to be portable. There are provided 'TODO' templates to be completed for a specific platform. Several platform example implementations have been provided including: Windows 32–bit, Linux, Visor, Palm, and PocketPC. There are also several example applications that use these platform implementations.

There are two sets of portable source files. The first set is general purpose and is intended for platforms that already have the primitive link 1–Wire communication functions (general). This is the lowest level that is hardware dependent. The other set of portable source files assumes that the user has a serial port (RS232) and wishes to utilize the 'Universal Serial 1–Wire Line Driver Master: DS2480B' (userial). This chip receives commands over the serial port, performs 1–Wire operations and then sends the results back to the serial port. The source code converts the intended 1–Wire operations into serial communications packets to the DS2480B. The only module that need be provided for a platform are the serial port read/write primitives. The DS2480B is the interface chip used in all of the DS9097U series serial adapters.

These two sets of portable source code files implement the same 1–Wire API functions and are interchangeable. Figure 3 shows the available API for the version 3.00 of the 1–Wire Public Domain code base. Note that the non–memory device specific functions are not listed in detail due to their large number. Figure 4 maps the source files that provide the functions and the required modules for new platforms.

In addition to the portable 'C' modules in the PD kit download, there are also a limited number of microprocessor assembly examples to do 1–Wire communication.

The file functions in this API implement the 1–Wire File Structure type 'AA' as defined by Application Note 114 which can be found here: http://pdfserv.maxim–ic.com/arpdf/AppNotes/app114.pdf

As the name of this API would imply, the source code provided has a license that is as close to 'Public Domain' as possible. Developers are free to use and integrate this code into their applications without restriction.

PD API FUNCTIONS Figure 3

SESSION

owAcquire – Acquires the 1–Wire net.
owRelease – Releases the previously acquired 1–Wire net.

LINK

owHasOverDrive – Indicates whether the adapter has overdrive capability.
owHasPowerDelivery – Indicates whether the adapter can deliver power.
owHasProgramPulse – Indicates whether or not EPROM programming voltage is available.
owLevel – Sets the 1–Wire net line level to Normal (5V weak pullup), Power Delivery (5V strong pullup), or Program Level (12V EPROM programming level).
owProgramPulse – Sends timed programming pulse for EPROM 1–Wire device writing.
owReadBitPower – Reads 1 bit and then optionally supplies power.
owReadByte – Receives 8 bits from the 1–Wire net by sending all 1's (0xFF).
owSpeed – Sets the speed of the 1–Wire net to Normal (16K bits) or Overdrive (142K bits).
owTouchBit – Sends and receives 1 bit from the 1–Wire net.
owTouchByte – Sends and receives 8 bits from the 1–Wire net.
owTouchReset – Resets all devices on the 1–Wire net and returns result.
owWriteByte – Sends 8 bits to the 1–Wire net and verifies the echo received matches.
owWriteBytePower – Sends 8 bits of communication to the 1–Wire net and then supplies power.

NETWORK

owAccess – Selects the current device and readies it for a device–specific command.
owFamilySearchSetup – Sets up the following search (owNext) to find a specific family type .
owFirst – Searches to find the 'first' 1–Wire device on the 1–Wire net.
owNext – Searches to find the 'next' 1–Wire device on the 1–Wire net.
owOverdriveAccess – Selects the current device and places it in Overdrive speed.
owSerialNum – Retrieves or sets the currently selected device serial number (ROM number).
owSkipFamily – Skips all of the 1-Wire devices with the family type that was found in the last search.
owVerify – Selects and verifies that the current device is present (alarming option).

TRANSPORT

owBlock – Sends and receives a block of data to the 1–Wire net with optional reset.
owCanLockPage – Checks to see if the given memory bank has pages that can be locked.
owCanLockRedirectPage – Checks to see if the given memory bank has pages that can be locked from being redirected.
owGetAlternateName – Gets the alternate part numbers or names.
owGetBankDescription – Gets a string description of the memory bank.
owGetDescription – Gets a short description of the 1–Wire device type.
owGetExtraInfoDesc – Gets a description of what is contained in the extra information.
owGetExtraInfoLength – Gets the length in bytes of extra information in this memory bank.
owGetMaxPacketDataLength – Gets maximum data length in bytes for a packet.
owGetName – Gets the part number of the 1–Wire device as a string.
owGetNumberBanks – Gets the number of memory banks for a certain 1–Wire family group.
owGetNumberPages – Gets the number of pages in a given memory bank.
owGetPageLength – Gets the raw page length in bytes for a given memory bank.
owGetSize – Gets the size of a given memory bank in bytes.

owGetStartingAddress – Gets the physical starting address of the given memory bank.

owHasExtraInfo – Checks to see if this memory bank's pages deliver extra information when read.

owHasPageAutoCRC – Checks to see if the memory bank has device generated CRC verification when reading a page.

owIsGeneralPurposeMemory – Checks to see if the memory bank is general purpose user memory.

owIsNonvolatile – Checks to see if current memory bank is non–volatile.

owIsReadOnly – Checks to see if the memory bank is read–only.

owIsReadWrite – Checks to see if the memory bank is read/write.

owIsWriteOnce – Checks to see if the memory bank is write once such as with EPROM.

owNeedsPowerDelivery – Checks to see if this memory bank requires 'PowerDelivery' to write.

owNeedsProgramPulse – Checks to see if this memory bank requires a 'ProgramPulse' to write.

owRead – Reads a portion of a memory bank in raw mode (no packets, CRC).

owReadPage – Reads an entire page of a memory bank in raw mode (no packets, CRC).

owReadPageCRC – Reads an entire page of a memory bank with device generated CRC verification.

owReadPageExtra – Reads an entire raw page of a memory bank including any ‘extra’ information (no packets, CRC).

owReadPageExtraCRC – Reads an entire raw page of a memory bank including any ‘extra’ information and with device generated CRC verification.

owReadPagePacket – Reads a Universal Data Packet from a page in a memory bank (See Application Note 114 for Universal Data Packet structure description).

owReadPagePacketExtra – Reads a Universal Data Packet from a page in a memory bank with ‘extra’ information.

owRedirectPage – Checks to see if the memory bank has pages that can be redirected.

owWrite – Writes a portion of a memory bank in raw mode.

owWritePagePacket – Writes a Universal Data Packet to a page in a memory bank.

FILE

owAttribute – Changes the attributes of a file.

owChangeDirectory – Changes the current directory.

owCloseFile – Closes a file.

owCreateDir – Creates a directory.

owCreateFile – Creates a file for writing.

owCreateProgramJob – Creates a write buffer for logging EPROM programming pending jobs.

owDeleteFile – Deletes a file.

owDoProgramJob – Write the pending EPROM programming jobs.

owFirstFile – Finds the first file in the current directory.

owFormat – Formats the 1–Wire File Structure file system.

owGetCurrentDir – Gets the current directory.

owNextFile – Finds the next file in the current directory.

owOpenFile – Opens a file for reading.

owReadFile – Reads an opened file.

owReadFile – Reads data from a file.

owRemoveDir – Removes a directory.

owReNameFile – Changes the name of a file.

owWriteFile – Writes to a file that has been created.

DEVICE
DoAtoDConversion – Does an A to D conversion on DS2450. *ReadSwitch12* – Reads the state of the DS2406 switch *GetFirmwareVersionString* – Gets the firmware version string for a Java–powered iButton … too numerous to list all device–specific functions

The Example 1 below shows a PD code fragment that follows the 'API Usage Flow' outlined in Figure 3. For simplicity each device on the 1-Wire network is discovered each pass through the work loop. A more sophisticated application could potentially find just one device type or perhaps select a device found in a previous search.

PD CODE EXAMPLE Example 1

```
int rslt, portnum=0, doing_work=1;
char portString[50]; // set to platform appropriate port string

// work loop
while (doing_work)
{
   // acquire the 1-Wire Net (SESSION)
   if (owAcquire(portnum, portString))
   {
      // find all devices (NETWORK)
      rslt = owFirst(portnum, TRUE, FALSE);
      while (rslt)
      {
         // do SOMETHING with device found (TRANSPORT/FILE/DEVICE)
         // . . .

         // find the next device (NETWORK)
         rslt = owNext(portnum, TRUE, FALSE);
      }

      // release the 1-Wire Net (SESSION)
      owRelease(portnum);
   }
   else
   {
      // Could not acquire 1-Wire network
      // . . .
   }

   // do other application work
   // . . .
}
```

Figure 4 ('a' and 'b') lists the C–language modules that make up each of the two sets of 1-Wire Public Domain libraries. Also displayed are the 'TODO' functions that must be provided to port the library to a new platform. Note several example platform link files that implement the 'TODO' functions are provided in the kit.

PD ‘USERIAL’ IMPLEMENTATION Figure 4a

SESSION					
owsesu.c					
LINK					
owllu.c	ds2480ut.c	ds2480.h			
NETWORK					
ownetu.c	crcutil.c				
TRANSPORT					
mbappreg.c	mbappreg.h	mbee.c	mbee.h	mbeprom.c	mbeprom.h
mbnv.c	mbnv.h	mbnvcrc.c	mbnvcrc.h	mbscr.c	mbscr.h
mbscrcrc.c	mbscrcrc.h	mbscree.c	mbscree.h	mbscrex.c	mbscrex.h
mbsha.c	mbsha.h	mbshaee.c	mbshaee.h	owtrnu.c	rawmem.c
rawmem.h					
FILE					
owcache.c	owfile.c	owfile.h	owpgrw.c	owprgm.c	
DEVICE					
ad26.c	ad26.h	atod20.c	cnt1d.c	jib96.c	jib96o.c
jib96.h	sha18.c	sha33.c	shadebit.c	shadbtvm.c	shaib.c
shaib.h	swt05.c	swt12.c	swt12.h	swt1f.c	temp10.c
thermo21.c	thermo21.h	weather.c	weather.h		
MISC UTILITY					
ioutil.c	owerr.c	findtype.c	ownet.h	screenio.c	sprintf.c
TODO					

Provided a SERIAL interface module that implements the following functions:
*BreakCOM** – Sends a 'BREAK' on the serial port for at least 2 milliseconds.
CloseCOM – Closes the previously opened serial port. (optional for some platforms)
*FlushCOM** – Allows any pending write operations to complete and clear input buffer.
*msDelay** – Delays at least the specified number of milliseconds.
msGettick – Returns an increment millisecond counter. (optional for some examples)
OpenCOM –Opens the specified serial port for communication. (optional for some platforms)
*ReadCOM** – Reads a specified number of bytes from the serial port.
SetCOMBaud – Changes the serial BAUD rate to the rate specified. (optional if need overdrive)
*WriteCOM** – Writes a specified number of bytes to the serial port.

* Minimum functions required for basic operation

PD ‘GENERAL’ IMPLEMENTATION Figure 4b

SESSION

(see TODO)

LINK

(see TODO)

NETWORK

ownet.c crcutil.c

TRANSPORT

Same as USERIAL implementation except ‘owtrnu.c’ is replaced by ‘owtran.c’.

FILE

Same as USERIAL implementation.

DEVICE

Same as USERIAL implementation.

MISC UTILITY

Same as USERIAL implementation.

TODO

Provided a LINK and SESSION interface module that implements the following functions:

owAcquire – Acquires the 1–Wire net.

owRelease – Releases the previously acquired 1–Wire net.

owHasOverDrive – Indicates whether the adapter has overdrive capability.

owHasPowerDelivery – Indicates whether the adapter can deliver power.

owHasProgramPulse – Indicates whether or not EPROM programming voltage is available.

owLevel – Sets the 1–Wire net line level to Normal (5V weak pullup), Power Delivery (5V strong pullup), or Program Level (12V EPROM programming level).

owProgramPulse – Sends timed programming pulse for EPROM 1–Wire device writing.

owReadBitPower – Reads 1 bit and then optionally supplies power.

owReadByte – Receives 8 bits from the 1–Wire net by sending all 1's (0xFF).

owSpeed – Sets the speed of the 1–Wire net to Normal (16K bits) or Overdrive (142K bits).

*owTouchBit** – Sends and receives 1 bit from the 1–Wire net.

owTouchByte – Sends and receives 8 bits from the 1–Wire net.

*owTouchReset** – Resets all devices on the 1–Wire net and return result.

owWriteByte – Sends 8 bits to the 1–Wire net and verifies the echo received matches.

owWriteBytePower – Sends 8 bits of communication to the 1–Wire net and then supplies power.

* Minimum functions required for basic operation

INSTALLATION

The 1-Wire Public Domain API is a set of C modules so there is no formal installation. As provided in the example ‘builds’, the required modules are compiled directly into the applications. This does not preclude developers from combining the modules into a loadable library such as a Windows DLL.

1–WIRE API FOR JAVA (OWAPI) OVERVIEW

The 1–Wire API for Java was designed from the ground up to be a very robust, highly object–oriented foundation for building 1–Wire applications in Java. It extends the ability of programmers to develop portable, cross–platform software and shortens the time to market for their 1–Wire integrated products.

The API consists of many Java classes and interfaces. One special group of Java classes in the 1–Wire API is the container (class OneWireContainer). Support for particular 1–Wire devices, including iButtons, is provided through containers. The API has over 30 different container types, representing most 1–Wire devices. Each container encapsulates and implements the functionality of an individual device.

A "container" interacts with a 1–Wire device through a 1–Wire adapter class which represents a physical 1–Wire adapter (class DSPortAdapter). The instance of the adapter is produced from the provider class (class OneWireAccessProvider). The actual implementations of the 1–Wire adapters vary from platform to platform but they all have the same interface. Some platforms use native drivers but most at least support the DS9097U-XXX serial adapters using the Java Communications API (or equivalent). This API is available from Sun's Web site: http://java.sun.com/products/javacomm.

The 1–Wire API for Java kit is located at: http://www.ibutton.com/software/1wire/1wire_api.html

Like the 1-Wire Public Domain kit, the complete Java source to OWAPI is provided under a 'Public Domain' style license.

Figure 5 shows the typical object creation sequence for this API. The 'provider' creates an instance (or enumeration) of 'adapter' which in turn can create instances of device 'container's. Communication to the device is then performed almost exclusively through the 'container'.

OWAPI OBJECT CREATION Figure 5

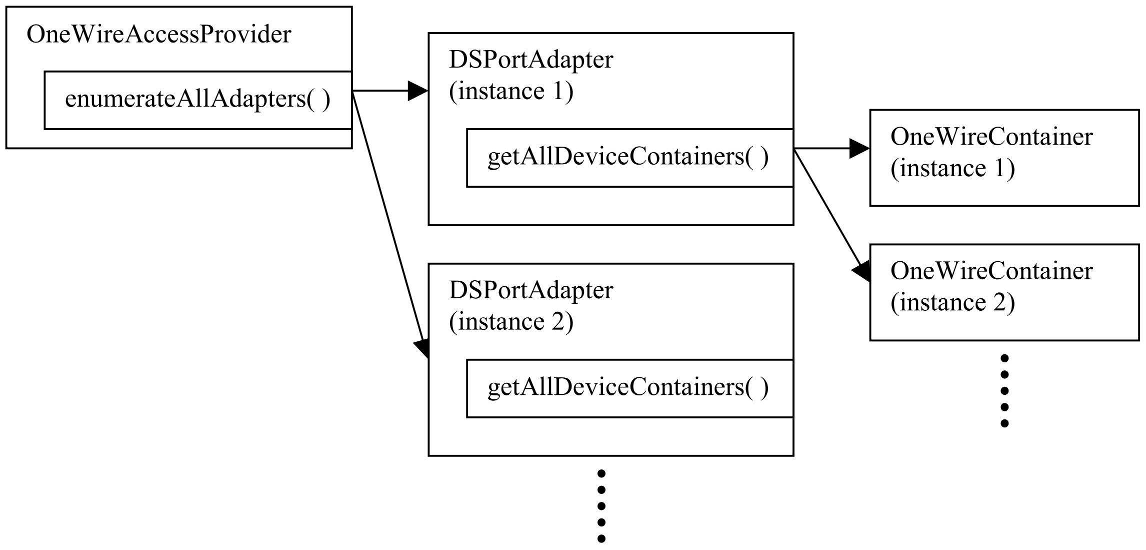


Figure 6 shows the common features of a 'container'. A device that contains memory will create a 'memory bank' instance for each memory bank. The memory is divided up into banks depending on the feature set of the bank. For example one bank could be volatile while another is non–volatile. Or a bank could be general purpose memory or it could be 'memory–mapped' to change the functionality of the device.

OWAPI ONEWIRECONTAINER FEATURES Figure 6

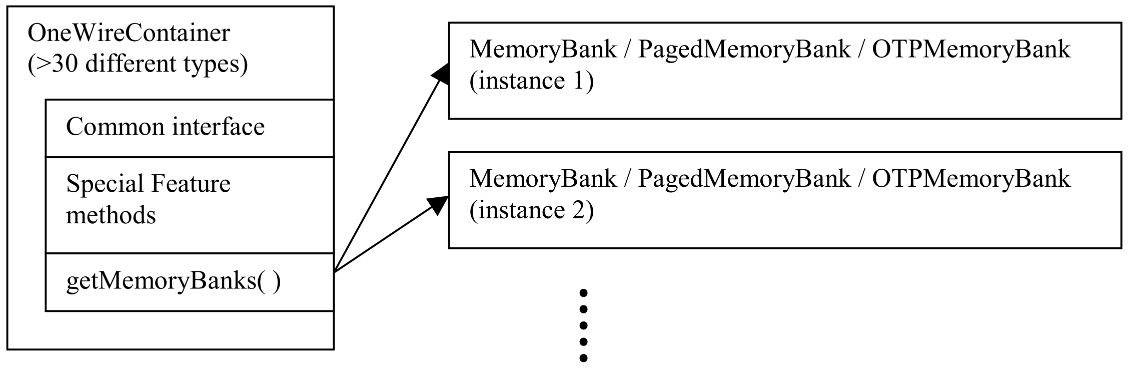

OneWireContainer
(>30 different types)
Common interface
Special Feature methods
getMemoryBanks( )
MemoryBank / PagedMemoryBank / OTPMemoryBank (instance 1)
MemoryBank / PagedMemoryBank / OTPMemoryBank (instance 2)

Figure 6 shows the methods provided by the base OWAPI classes. The class or package is displayed in **bold**. Note that since each container has high level methods to manipulate each device type, the LINK level methods in the adapter are not usually called directly.

OWAPI FUNCTIONS Figure 7

SESSION

com.dalsemi.onewire.adapter.DSPortAdapter

beginExclusive – Acquires exclusive use of the 1–Wire net.
endExclusive – Release the exclusive lock on the 1–Wire net.

LINK

com.dalsemi.onewire.adapter.DSPortAdapter

canBreak – Checks if a 1–Wire 'break' (long low) operation is supported by the adapter.
canDeliverPower – Checks if 'strong–pullup' power delivery is supported by the adapter.
canDeliverSmartPower – Checks if 'smart' power delivery is supported by the adapter. 'smart' power delivery is the ability to sense when power consumption has reduced and automatically stop the power delivery.
canFlex – Checks if flexible long–line communication timing is supported by the adapter.
canHyperdrive – Checks if hyperdrive communication speed is supported by the adapter.
canOverdrive – Checks if overdrive communication speed is supported by the adapter.
canProgram – Checks if 12Volt EPROM programming voltage is supported by the adapter.
dataBlock – Send and receive a block of data to the 1–Wire net.
getBit – Read a single bit from the 1–Wire net.
getBlock – Read a block from the 1–Wire net (by sending all (0xFF's)).
getByte – Read a byte from the 1–Wrire net by sending all 1's (0xFF).
getSpeed – Read the current 1–Wire communication speed.
putBit – Write a bit to the 1–Wire net.
putByte – Write a byte to the 1–Wire net and verify the echo is correct.
reset – Reset all of the 1–Wire net devices.
setPowerDuration – Set the power delivery duration.
setPowerNormal – Turn off the power delivery.
setProgramPulseDuration – Set the program pulse duration.
setSpeed – Set the 1–Wire communication speed.
startBreak – Start a break (low) on the 1–Wire net.
startPowerDelivery – Start the power delivery.
*startProgramPul*se – Start the program pulse.

NETWORK

com.dalsemi.onewire.adapter.DSPortAdapter

- *excludeFamily* – Exclude a family group from the search.
- *findFirstDevice* – Find the first device on the 1–Wire net without auto container creation.
- *findNextDevice* – Find the next device on the 1–Wire net without auto container creation.
- *getAllDeviceContainers* – Search and find all devices on the 1–Wire net with containers.
- *getDeviceContainer* – Get a device container for the 'current' device found.
- *getFirstDeviceContainer* – Find the first device and create a container for it.
- *getNextDeviceContainer* – Find the next device and create a container for it.
- *setNoResetSearch* – Set the 1–Wire net search to not issue a 1–Wire reset.
- *setSearchAllDevices* – Set the 1–Wire net search to include all devices (remove alarm–only).
- *setSearchOnlyAlarmingDevices* – Set the 1–Wire net search to only include alarming devices.
- *targetAllFamilies* – Set the 1–Wire net search to include all devices (remove exclusions).
- *targetFami*ly – Target a particular family group in the 1–Wire net search.

(also in **com.dalsemi.onewire.container.***)

- *isAlarming* – Check to see if the device is in an alarm state.
- *isPresent* – Check to see if the device is present on the 1–Wire net.
- *select* – Selects the 1–Wire net device to ready it for a device specific operation command.

TRANSPORT

com.dalsemi.onewire.container.MemoryBank

- *getBankDescription* – Return a text description of the memory bank.
- *getSize* – Get the size of the memory bank in bytes.
- *getStartPhysicalAddress* – Get the starting physical address of the memory bank.
- *isGeneralPurposeMemory* – Checks if the memory bank is general purpose (not memory mapped)
- *isNonVolatile* – Checks if the memory bank is not volatile.
- *isReadOnly* – Checks if the memory bank is read–only.
- *isReadWrite* – Checks if the memory bank is read and write capable.
- *isWriteOnce* – Checks if the memory bank is write–once such as EPROM.
- *needsPowerDelivery* – Checks if this memory bank requires power delivery to write.
- *needsProgramPulse* – Checks if this memory bank requires program pulse to write.
- *read* – Reads the memory bank without interpretation (no packet structure).
- *setWriteVerification* – Sets the API to do an extra verification after writes.
- *write* – Writes the memory bank raw (no packet structure).

com.dalsemi.onewire.container.PagedMemoryBank

- *getExtraInfoDescription* – Get a description of extra information associated with this bank.
- *getExtraInfoLength* – Get the length in bytes of the extra information in each page.
- *getMaxPacketDataLength* – Get the maximum length of data that can be contained in the 'packet' structure that will fit in each page of this memory bank.
- *getNumberPages* – Get the number of pages in this memory bank.
- *getPageLength* – Get the length in byte of the raw page in this memory bank.
- *hasExtraInfo* – Checks if this memory bank has extra information associated with each page.
- *hasPageAutoCRC* – Checks if the pages in this memory bank have CRC verification that is supplied by the device.
- *readPage* – Reads a page from the memory bank.
- *readPageCRC* – Read a page from the memory bank utilizing the device generated CRC.
- *readPagePacket* – Read a packet structure from a page in the memory bank.
- *writePagePacket* – Write a packet structure to a page in the memory bank.

com.dalsemi.onewire.container.OTPMemoryBank

canLockPage – Checks if the pages in the memory bank can be locked from further writes.
canLockRedirectPage – Checks to see if the redirection facilities in the memory bank can be locked to prevent further redirection.
canRedirectPage – Checks to see if the memory bank can have pages redirected as a way to update write–once pages.
getRedirectedPage – Get the page number that a page is redirected to.
isPageLocked – Checks if the page is locked from further writes.
isRedirectPageLocked – Checks if the page is locked from further redirection.
lockPage – Locks a page.
lockRedirectPage – Locks the page from being redirected.
redirectPage – Redirects a page to a new page. This is used to update a write–once device.

FILE

com.dalsemi.onewire.utils.OWFile

Same methods in java.io.File (for version 1.2 of the JDK) plus the following extra methods:
close – Close the file and releases any resources associated with it.
format – Format the 1–Wire File System associated with the device(s) provided to this OWFile.
getFD – Get a OWFileDescriptor for this file so the file can be 'synced' with the device.
getFreeMemory – Get the available free memory in the 1–Wire File System.
getLocalPage – Get a memory bank local page reference from the 1–Wire File System page.
getMemoryBankForPage – Get the memory bank instance that can be used to read/write the provided 1–Wire File System page.
getOneWireContainer – Get the container(s) that make up the file system.
getPageList – Get a list of 1–Wire File System pages that comprise the file.

com.dalsemi.onewire.utils.OWFileDescriptor

Same methods as in java.io.FileDescriptor (for version 1.2 of the JDK)

com.dalsemi.onewire.utils.OWFileOutputStream

Same methods as in java.io.FileOutputStream (for version 1.2 of the JDK)

com.dalsemi.onewire.utils.OWFileInputStream

Same methods as in java.io.FileInputStream (for version 1.2 of the JDK)

DEVICE

com.dalsemi.onewire.container.*

Over 30 different device specific container implementations including six different 'sensor' type interfaces:
ADContainer – Analog to Digital converter
ClockContainer – Clock
SwitchContainer – Switch
TemperatureContainer – Temperature sensor
PotentiometerContainer – Digital Potentiometer
HumidityContainer – Humidity sensor

com.dalsemi.onewire.application.*

SHA and 1–Wire Tagging utility classes.

com.dalsemi.onewire.jib.*

Provide OpenCard support for the Java Powered iButton (Opencard info at http:\\www.opencard.org)

The Example 2 below shows a OWAPI code fragment that follows the 'API Usage Flow' outlined in Figure 3. For simplicity each device on the 1-Wire network is discovered each pass through the work loop. A more sophisticated application could potentially find just one device type or perhaps select a device found in a previous search.

OWAPI CODE EXAMPLE Example 2

```
boolean doing_work=true;

// get the default adapter from the service provider
DSPortAdapter adapter = OneWireAccessProvider.getDefaultAdapter();

// work loop
while (doing_work)
{
   // get exclusive use of adapter (SESSION)
   adapter.beginExclusive(true);

   // clear any previous search restrictions (NETWORK)
   adapter.setSearchAllDevices();
   adapter.targetAllFamilies();
   adapter.setSpeed(adapter.SPEED_REGULAR);

   // enumerate through all the 1-Wire devices found (NETWORK)
   for (Enumeration owd_enum = adapter.getAllDeviceContainers();
        owd_enum.hasMoreElements(); )
   {
      // get a 'container' for each device
      OneWireContainer owd = ( OneWireContainer ) owd_enum.nextElement();

      // do SOMETHING with device found (TRANSPORT/FILE/DEVICE)
      // . . .
   }

   // end exclusive use of adapter (SESSION)
   adapter.endExclusive();

   // do other application work
   // . . .
}
```

1-WIRE TAGGING

As 1-Wire sensors get more numerous and diverse it becomes increasingly difficult to manage a 1-Wire network. For example, a sensor like an A/D could measure various different values so it becomes important to be able to 'tag' the sensor to describe it's function. A 1-Wire Tagging scheme using XML has been created and implemented in the 1-Wire API for Java. These tags allow an application to dynamically load and configure a sensor to give it a context. Please see Application Note 158 "1-Wire Tagging with XML" for details.

INSTALLATION

All of the required modules that make up the API calls described in Figure 7 are contained in a single jar file: OneWireAPI.jar. Placing this one module in the correct location or classpath provides the entire API. There are two noted exceptions to this: there could be native or communication API's required to be installed for a particular platform, or the platform could have size constraints so that it is undesirable to have the entire API available. These two exceptions are examined in detail in the OWAPI kit.

1–WIRE COM (OWCOM) OVERVIEW

The 1–Wire Windows COM (Component Object Model) framework is simply a wrapper for the 1–Wire API for Java (OWAPI). Almost any programming language that works on 32–bit Microsoft Windows operating systems can use COM objects. Since this utilizes OWAPI, our richest collection of support classes for Dallas Semiconductor's 1–Wire devices is available to a variety of programming environments including C++, VisualBasic, JavaScript, JScript, Perl, and VBScript.

Since Java and COM are not completely compatible, several workarounds had to be used to get the existing Java code-base to work in a COM interface. These workarounds are outlined in the next section.

The Software Developer Kit containing OWCOM (and TMEX) can be found on this site:
http://www.ibutton.com/software/tmex/index.html

The Java source code to create OWCOM is provided under a liberal 'Public Domain' style license. The only source not provided is for the TMEX API drivers that are called to support the different 1-Wire adapters (see TMEX API section). These drivers however can be redistributed without restriction.

OWCOM DIFFERENCES WITH OWAPI

Since the underlying structure of all the interfaces is actually implemented using the OWAPI, the documentation is the same. There are just a few rules of thumb for applying the Java documentation to a COM interface.

1. Java uses 64–bit longs and the interfaces used in COM don't support a number larger than 32–bits. For any function in Java that takes a long as a parameter or returns a long, a string is used. In loosely–typed programming languages (like VisualBasic or JavaScript), the conversion between long and string will be automatic and probably won't affect the code in any way. For all other languages, special care must be taken that parameters are converted to strings and return values are converted to 64–bit longs.
2. Several of the classes in OWAPI use an array of bytes for caching state information about an object. Unfortunately, languages that don't allow for passing values by reference (like JavaScript) actually cloned the data in the array for passing as a parameter. Since this breaks the functionality expected from passing array values as parameters, an object was created for containing arrays of bytes. The object, ByteArray, is the only object whose equivalent doesn't exist in the current Java OneWire API and, as such, the necessary documentation is produced below:

```
ByteArray
{
  // sets the item at the given index with the given value
  void setAt(nIndex,nValue);

  // returns the numerical value at the given index
  nValue getAt(nIndex);

  // grows or shrinks the size of the array, copies all data that fits
  void setSize(nSize);

  // returns the size of the array
  nSize getSize();
}
```

3. In Java, method overloading allows for methods to differ in types of their parameters as well as in the number of parameters. In COM, only the number of parameters can distinguish two methods with the same name. All the methods that would conflict with this requirement fit into the same pattern. They were either expecting a string, a long, or an array of bytes. The method that expects a string was left alone while the method expecting a long has a new name with "FromLong" on the end. The method expecting an array of bytes has "FromBytes" added on the end. For example:

Java		**COM**
isPresent(String)	—>	isPresent(String)
isPresent(long)	—>	isPresentFromLong(String) //By rule #1 and #3
isPresent(byte[])	—>	isPresentFromBytes(ByteArray) //By rule #2 and #3

4. Some containers in OWAPI depend on polymorphism for their functionality. Polymorphism is, essentially, the ability of a class to be treated as if it was actually an instance of a parent class. In COM, polymorphism doesn't manifest itself in quite the same way since COM doesn't support inheritance. Each COM component in our architecture represents an actual instance of an object from OWAPI, but usually a higher–level component (like MemoryBank) will actually contain an instance of a lower–level object (like PagedMemoryBank or OTPMemoryBank). In Java, simply cast the object to it's specific type. Unfortunately, it's not that easy in COM, so a helper method was added to three top–level containers: OneWireContainer, MemoryBank, and DSPortAdapter. The interface for the helper method is simply:

```
Component getMostSpecificComponent();
```

The return value depends on which class is being called. For example, an instance of OneWireContainer that is actually holding a Thermochron object (OneWireContainer21), calling getMostSpecificComponent will return an instance of the OneWireContainer21 COM interface.

5. Another feature that is lost with polymorphism is the mechanism by which it is discovered whether or not a particular 1–Wire device supports a particular interface. Polymorphism also allows a class to be treated as if it was actually an instance of an interface that the class implements. OneWireContainer21, for example, implements the interface TemperatureContainer. That is to say, OneWireContainer21 implements all methods that are common among TemperatureContainers (like "readTemperature"). In Java, it is common to compare an instance of a one wire object with a particular interface using the "instanceof" keyword. If the "myOneWireDevice instanceof TemperatureContainer" statement returned true, then the device implements all the methods necessary for reading temperatures. The solution in COM is to use these methods defined in all OneWireContainers:

```
  boolean isTemperatureContainer();
  boolean isADContainer();
  boolean isSwitchContainer();
  boolean isClockContainer();
  boolean isPotentiometerContainer();
  boolean isHumidityContainer(); [NOT YET IMPLEMENTED]
```

6. Functions in Java that used to return the java.util.Calendar object now return the current time in milliseconds since Jan 1, 1970 GMT. This is the unix standard for representing time.

7. The interfaces are not in Java style packages but now reside all at the same level. For example OneWireContainer21 no longer is in package com.dalsemi.onewire.container.OneWireContainer21 but is accessed from the COM package simply as OneWireContainer21.

The Example 3 below shows a OWCOM code fragment that follows the ‘API Usage Flow’ outlined in Figure 3. For simplicity each device on the 1-Wire network is discovered each pass through the work loop. A more sophisticated application could potentially only find one device type or perhaps select a device found in a previous search.

OWCOM ‘Java Script’ CODE EXAMPLE Example 3

```
boolean doing_work=true;

// get the 1-Wire access provider
var access = WScript.CreateObject("owapi.OneWireAccessProvider");

// get the default adapter
var adapter = access.getDefaultAdapter();

// work loop
while (doing_work)
{
   // get exclusive use of adapter (SESSION)
   adapter.beginExclusive(true);

   // clear any previous search restrictions (NETWORK)
   adapter.setSearchAllDevices();
   adapter.targetAllFamilies();
   adapter.setSpeed(adapter.SPEED_REGULAR);

   // enumerate through all the 1-Wire devices found (NETWORK)
   for (Enumeration owd_enum = adapter.getAllDeviceContainers();
        owd_enum.hasMoreElements(); )
   {
      // get a ‘container’ for each device
      owd = owd_enum.nextElement();

      // do SOMETHING with device found (TRANSPORT/FILE/DEVICE)
      // . . .
   }

   // end exclusive use of adapter (SESSION)
   adapter.endExclusive();

   // do other application work
   // . . .
}
```

INSTALLATION

There is a batch file provided that automatically installs the OWCOM driver. It registers the COM object with the regsvr32 Windows tool and then copies the OWAPI class files to a ‘trusted’ folder.

TMEX API (TMEX) OVERVIEW

The TMEX API is a set of language independent Windows 32–Bit DLLs that provides basic functionality to all 1–Wire devices including limited 1–Wire File Structure support to memory devices. The API is designed to work in multi–process multi–threaded applications all vying for the same or different 1–Wire ports. The API can support up to 16 different types of 1–Wire adapters each with 16 distinct ports. It supports all of the 1–Wire adapters created by Dallas Semiconductor. Consequently this API is used as the 'native' drivers for the 1–Wire API for Java on the Win32 platforms. Since the 1–Wire COM API is based on the 1–Wire API for Java it is also used there. Figure 8 shows graphically how the other API's take advantage of the Win32 native support that the TMEX API provides. Included in this figure are the actual driver filenames and how they are layered.

The Software Developer's Kit containing the TMEX API (and OWCOM) is located at:
http://www.ibutton.com/software/tmex/index.html

All of the examples provided in the TMEX SDK are provided with source code, however the source to the drivers is currently not provided. The drivers however can be redistributed without restriction.

Table 5 below lists the currently supported 1–Wire adapters along with the features of each adapter and the supported Windows platforms. * Note that the Windows XP support is still in development.

TMEX ADAPTERS SUPPORTED Table 5

Adapter	Port	Features	Win95	Win98	WinME	WinNT	Win2K	WinXP
DS1490F 2–in–1 Fob	USB	power delivery overdrive led indicator single iButton holder		X	X		X	X*
DS1490x (future product)	USB	power delivery overdrive RJ–11 or iButton holder Optional EPROM write		X	X		X	X
DS1410E	Parallel	power delivery overdrive dual iButton holder DS2401 ID	X	X	X	X	X	X
DS1410D	Parallel	dual iButton holder DS2401 ID	X	X	X	X	X	X
DS9097U–009	Serial	power delivery overdrive RJ–11 connector DS2502 ID	X	X	X	X	X	X
DS9097U–S09	Serial	power delivery overdrive RJ–11 connector	X	X	X	X	X	X
DS9097U–E25	Serial	power delivery overdrive RJ–11 connector EPROM write	X	X	X	X	X	X
DS1411	Serial	power delivery overdrive single iButton holder	X	X	X	X	X	X
DS9097E	Serial	RJ–11 connector EPROM write	X	X	X	X	X	X
DS9097	Serial	RJ–11 connector	X	X	X	X	X	X
DS1413	Serial	single iButton holder	X	X	X	X	X	X

TMEX API DRIVERS AND OTHER API CONNECTIVITY Figure 8

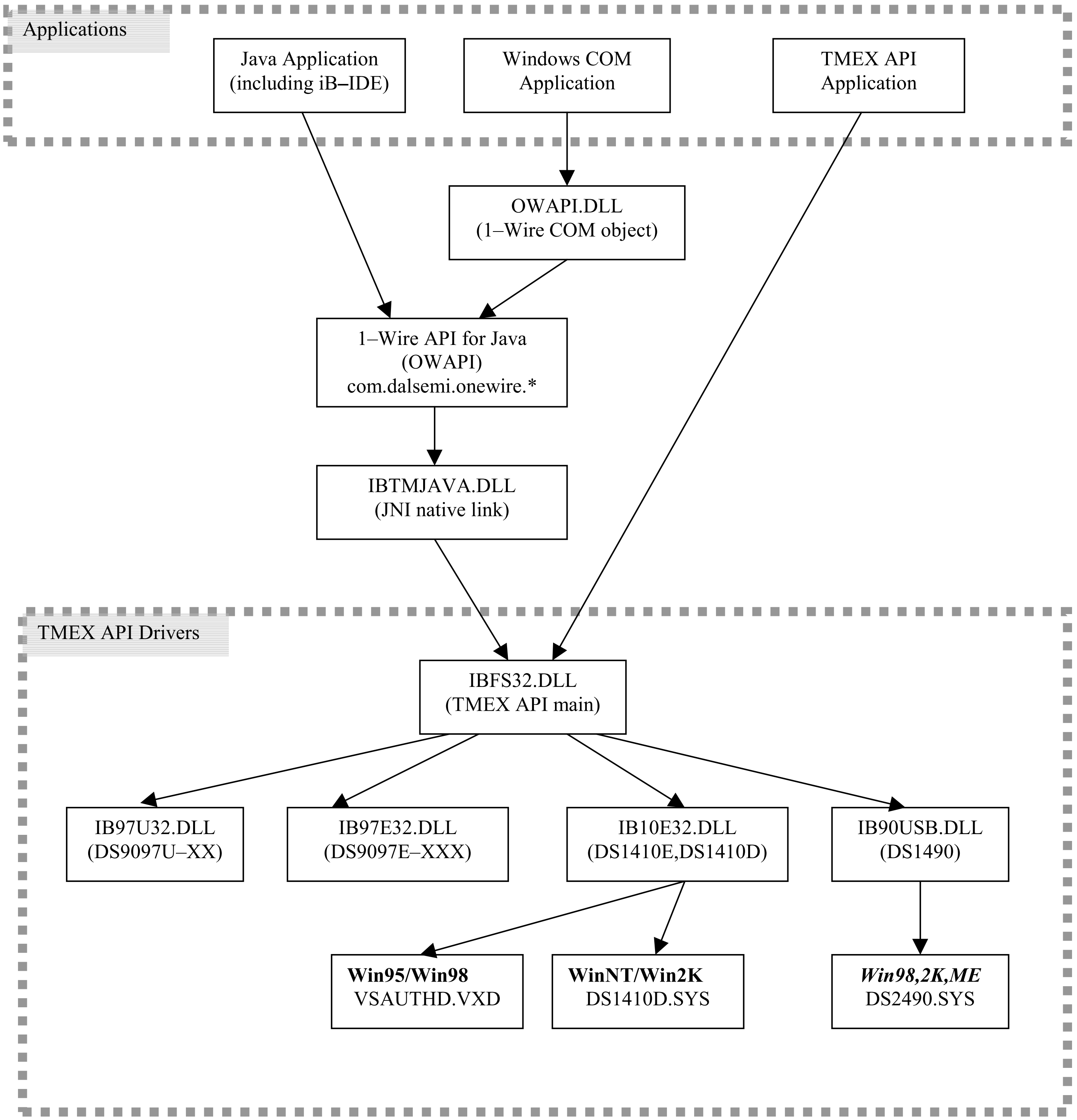

Applications
Java Application (including iB–IDE)
Windows COM Application
TMEX API Application
OWAPI.DLL (1–Wire COM object)
1–Wire API for Java (OWAPI) com.dalsemi.onewire.*
IBTMJAVA.DLL (JNI native link)
TMEX API Drivers
IBFS32.DLL (TMEX API main)
IB97U32.DLL (DS9097U–XX)
IB97E32.DLL (DS9097E–XXX)
IB10E32.DLL (DS1410E,DS1410D)
IB90USB.DLL (DS1490)
Win95/Win98 VSAUTHD.VXD
WinNT/Win2K DS1410D.SYS
Win98,2K,ME DS2490.SYS

Figure 9 lists functions provided by the TMEX API. Note that this API does not provide any non-memory device specific functions.

TMEX API FUNCTIONS Figure 9

SESSION

TMEndSession – relinquish the 1–Wire net.
TMExtendedStartSession – request exclusive use of the 1–Wire net.
TMValidSession – check to see if the current 1–Wire net session is valid

LINK

TMClose – Releases resources for the opened port (not always applicable)
TMOneWireCom – Sets the speed of the 1–Wire net to Normal (16K bits) or Overdrive (142K bits).
TMOneWireLevel – Sets the 1–Wire net line level to Normal (5V weak pullup), Power Delivery (5V strong pullup), or Program Level (12V EPROM programming level).
TMProgramPulse – Send a timed programming pulse to 1–Wire net for EPROM programming.
TMSetup – Checks and verifies the port and adapter is functioning.
TMTouchBit – Sends and receives 1 bit from the 1–Wire net.
TMTouchByte – Sends and receives 1 byte from the 1–Wire net
TMTouchReset – Resets all devices on the 1–Wire net and returns the result.

NETWORK

TMAccess – Selects the current device and readies it for a device–specific command.
TMAutoOverDrive – Sets the driver to automatically get the device in and out of overdrive speed
TMFamilySearchSetup – Sets the current search state to find a specified family type on the next search (TMNext or TMNextAlarm)
TMFirst – Searches to find the 'first' 1–Wire device on the 1–Wire net.
TMFirstAlarm – Searches to find the 'first' alarming 1–Wire device on the 1–Wire net.
TMNext – Searches to find the 'next' 1–Wire device on the 1–Wire net.
TMNextAlarm – Searches to find the 'next' alarming 1–Wire device on the 1–Wire net.
TMOverAccess – Select the current device and places it in Overdrive speed.
TMRom – Retrieves or sets the currently selected device serial number (ROM number).
TMSkipFamily – Skips all of the family type that was found in the last search.
TMStrongAccess – Selects and verifies that the current device is present.
TMStrongAlarmAccess – Selects and verifies that the current device is present AND alarming.

TRANSPORT

TMBlockIO – Sends and receives a block of data to the 1–Wire net preceded with a 1-Wire reset.
TMBlockStream – Sends and receives a block of data to the 1–Wire net with NO 1-Wire reset.
TMExtendedReadPage – Reads an entire page of a memory bank with device generated CRC verification (not applicable to all device types).
TMProgramByte – Programs a byte into an EPROM base 1–Wire device.
TMReadPacket – Reads a Universal Data Packet from a page. (See Application Note 114 for Universal Data Packet structure description).
TMWritePacket – Writes a Universal Data Packet to a page.

FILE

TMAttribute – Changes file or directory attributes.
TMChangeDirectory – Reads or changes the current working directory.
TMCloseFile – Closes an opened or created file to free up the file handle.
TMCreateFile – Creates a file for writing.
TMCreateProgramJob – Creates a write buffer for logging EPROM programming pending jobs.
TMDeleteFile – Deletes a file.
TMDirectoryMR – Makes or Removes a sub–directory.
TMDoProgramJob – Write the pending EPROM programming jobs.

TMFirstFile – Finds the first file in the current directory. *TMFormat* – Formats the 1–Wire File Structure file system. *TMNextFile* – Finds the next file in the current directory. *TMOpenFile* – Opens a file for reading. *TMReadFile* – Reads an opened file. *TMReNameFile* – Renames a file or directory. *TMTerminateAddFile* – Terminates an 'AddFile'. An ‘AddFile’ is a special file type on an EPROM device that can be appended to without rewriting. *TMWriteAddFile* – Appends or alters an 'AddFile' on an EPROM device. *TMWriteFile* – Writes to a file that has been created.
DEVICE
none
OTHER
TMGetTypeVersion – Gets the version information for the adapter driver. *Get_Version* – Gets the overall driver version.

The Example 4 below shows a TMEX code fragment that follows the ‘API Usage Flow’ outlined in Figure 3. For simplicity each device on the 1-Wire network is discovered each pass through the work loop. A more sophisticated application could potentially find just one device type or perhaps select a device found in a previous search.

TMEX 'C' CODE EXAMPLE Example 4

```
int PortNum, PortType; // port number and type set for adapter present
long session_handle; // session handle
unsigned char state_buffer[5120];
int doing_work=1, did_setup=0;
short rslt;

// work loop
while (doing_work)
{
   // aquire the 1-Wire Net  (SESSION)
   session_handle = TMExtendedStartSession(PortNum,PortType,NULL);
   if (session_handle > 0)
   {
      // check to see if TMSetup has been done once
      if (!did_setup)
      {
         if (TMSetup(session_handle) == 1)
            did_setup = 1;
         else
         {
            // error setting up port, adapter may not be present
            // . . .
         }
      }
      else
      {
         // find all devices (NETWORK)
         rslt = TMFirst(session_handle, state_buf);
         while (rslt > 0)
         {
            // do SOMETHING with device found (TRANSPORT/FILE/DEVICE)
            // . . .

            // find the next device (NETWORK)
            rslt = TMNext(session_handle, state_buf);
         }
      }

      // release the 1-Wire Net (SESSION)
      TMEndSession(session_handle);
   }
   else
   {
      // Could not acquire 1-Wire network
      // . . .
   }

   // do other application work
   // . . .
}
```

INSTALLATION

The TMEX API comes with an automated Install-Shield install program that loads all of the Windows drivers and registry keys for every supported 1-Wire adapter. The install is available with and without the iButton Viewer. The files are also provided for custom installations. The iButton Viewer is a demonstration program to exercise most of the 1-Wire devices and iButtons.

OTHER TOOLS

While the four API’s discussed in this document represent a large part of the available resources to communication with 1–Wire devices it is by no means the only resource. This section outlines some other available resources.

iB–IDE

The iB–IDE provides a complete programming environment designed to aid in rapid software development for the Java–powered iButton. A Java host application located on a personal computer or embedded system communicates with any applets installed on the iButton, and with the introduction of iB–IDE the design of both is easy. Some of iB–IDE's main features include:

- Fully–functioning Java–powered iButton simulator with Java source code level debugging
- Built–in text editor with Java keyword highlighting, macros, search and replace, and Java language–sensitive formatting
- Integrated Java–compile and run functionality, and the ability to communicate with and control iButtons connected to your system

Starting with version 2.0Beta, the iB–IDE uses the 1–Wire API for Java as its core to do the communication with the Java–powered iButton.

Information and download of the iB–IDE can be found here: http://www.ibutton.com/iB–IDE/

SOFTWARE AUTHORIZATION

Software Authorization is simply the locking of a software application to require the presence of a hardware token. The token in this case is an iButton connected to the user’s workstation. The application queries for the presence and validity of the iButton before running and potentially while the application executes. In practice any of the API’s outlined in this document can be used for this type of application, however there are specific concerns that must be considered. External drivers present a weakness to the security by allowing a user to replace the driver and thereby defeat the lock. The Software Authorization API has been developed specifically for this type of application and includes linkable modules instead of external drivers.

Information on the Software Authorization kit can be found here:
http://www.ibutton.com/software/softauth/index.html

1-WIRE SOFTWARE DEVELOPMENT INTEREST GROUP

It is recommended that 1-Wire developers join the ‘1-Wire Software Developer's Forum’. The purpose of this group is to explore, discuss, and answer questions about developing applications, tools, and uses for the 1-Wire family of products. This forum is monitored by Dallas Semiconductor Applications Engineers to answer questions posed by the group. Announcements about updates to the API’s and kit downloads are also distributed to this group. To subscribe go to http://lists.dalsemi.com/

THIRD PARTY

There is a large volume of third-party software available for 1-Wire devices. Some of these are applications produced by Dallas Semiconductor Authorized Solutions Developer (ASD) for purchase with their 'solution'. A complete list of ASD's and a solution locator is provided on this site: http://dbserv.maxim-ic.com/ibutton/solutions/search.cfm . There are also open source projects on public forums like 'Source Forge' http://sourceforge.net/

CONCLUSION

This document has provided an overview of the characteristics of any 1-Wire API. It has also detailed four different API's including the supported platforms and programming languages. A quick-reference table providing each API's coverage of each device type helped facilitate selection of which API to use. With the correct API in hand an application utilizing 1-Wire can readily be created.