# Programming the Twinklers

Version 1.1
July 2010

This application note describes the serial protocol used by the "Twinkler" hardware, and provides programming examples for a few languages and platforms.

## Contents

Introduction . . . 1
Protocol overview . . . 1
Support code . . . 1
Using the serial port API . . . 2
Using C or $C^{++}$, for Microsoft Windows . . . 2
Using VB.NET . . . 3
Using C# . . . 4
In Linux, using C/$C^{++}$ . . . 5
Using the native API . . . 7
High-level interface . . . 9
For $C^{++}$ . . . 9
For C# . . . 11
The protocol: commands and parameters . . . 13
Setting colours . . . 13
Fading and blinking/twinkling . . . 14
Applying a range . . . 14
Clock synchronization . . . 15
Resetting the Twinklers . . . 16
Changing the Baud rate . . . 16
Storing scenes . . . 16
Appendices . . . 17
A: Internals . . . 18
Index . . . 19

*CompuPhase*

Typeset with TEX in the “Computer Modern” and “Pandora” typefaces at a base size of 11 points.

# Introduction

The Twinkler hardware is controlled —for example through a personal computer, via a serial protocol. To connect a Twinkler, or a chain of Twinklers, to a PC, you need an interface, of which several kinds exist. At the logical (i.e. software) level, the protocol is RS232[1], with the following communication settings:
◇ Baud rate: 9600 bps (may be switched to 57600 bps)
◇ 8 data bits, 1 stop bit
◇ No parity, no flow control

Although the RS232 protocol was designed as a point-to-point protocol, with only two machines communicating with each other, the Twinkler hardware allows to chain an arbitrary number of Twinklers together, on a kind of serial bus. A single PC controls all Twinklers on that serial bus.

There are several ways to control a chain of Twinklers, depending on what kind of interface you have. For the RS232 interface (product H0611), your program sends its data over a COM port. The USB interface (product H0610) simulates a virtual COM port, so a program that connects to a COM port works on either interface. The USB interface has a second, native, API as well. The native API supports the four auxiliary inputs, for external switches or relays (for example).

## Protocol overview

The communication is set up with one "talker" and potentially many "listeners" on a serial bus. The talker is the personal computer; it sends out commands and data. The Twinklers accept and process incoming data, but do not reply to it. However, by looping the communication link of the last Twinkler in a chain back to the interface —thereby creating a ring, the PC can receive a response.

Commands and parameters are sent as bytes, where two bytes sometimes form a pair. A Twinkler board distinguishes commands from data by their value. All commands have values in the range 240. . . 255 (inclusive), all data bytes have values in the range 0. . . 239. Details of the command set is in chapter "The protocol: commands and parameters" on page 13.

The next chapter gives programming examples for accessing the serial port on several platforms and languages. Following that is a chapter on the native API, which is relevant for the USB interface (product H0610) only. The last chapter builds on these two chapters by describing the commands and data that you can send to the Twinklers through either of these interfaces (RS232 or native).

## Support code

The majority of this document provides the specification of the basic protocol and the methods to access the Twinkler hardware. For several programming languages and platforms, we also provide support code with a higher level interface, that you may be able to use directly. See the chapter "high-level interface" (page 9) for details.

[1] At the physical level, the protocol differs from RS232 because it uses TTL logic signal levels (5V). For an application program, the signal levels are not relevant, though.

# Using the serial port API

The serial port API is widely supported on programming languages and operating systems. It is also portable across the interfaces for the Twinkler: both the RS232 interface and the USB interface can be used with a program that is using the serial port API.

Note that in addition to the code snippets in this application note, there are solutions in the form of communication libraries. One example of such libraries is the COMM-DRV product from WCSC (`http://www.wcscnet.com`) that is available for several languages.

## Using C or C$^{++}$, for Microsoft Windows

Microsoft Windows views a serial port like a character device, and operations to it are similar as operations on a file. An RS232 port is opened with the `CreateFile` function, and you send data to it by calling `WriteFile`. There are a few special functions too, to set the Baud rate and other transfer settings.

```
HANDLE Com_OpenPort(LPCTSTR PortName, unsigned BaudRate)
{
    HANDLE hCom;
    DCB dcb;

    /* set up the connection */
    hCom = CreateFile(PortName, GENERIC_READ | GENERIC_WRITE, 0, NULL, OPEN_EXISTING,
                      FILE_ATTRIBUTE_NORMAL, NULL);
    if (hCom == INVALID_HANDLE_VALUE)
        return INVALID_HANDLE_VALUE;

    /* set the configuration, disable flow control */
    GetCommState(hCom, &dcb);
    dcb.BaudRate    = BaudRate;
    dcb.ByteSize    = 8;
    dcb.StopBits    = ONESTOPBIT;
    dcb.Parity      = NOPARITY;
    dcb.fDtrControl = DTR_CONTROL_DISABLE;
    dcb.fRtsControl = RTS_CONTROL_DISABLE;
    dcb.fOutX       = FALSE;
    dcb.fInX        = FALSE;
    SetCommState(hCom, &dcb);

    return hCom;
}
```

In this function, the parameter `PortName` must be the name of the COM port, for example "`COM1`" for the first RS232 port of a PC. COM ports above COM9 require a prefix, to open COM10, the filename is "`\\.\COM10`". Note that in a C/C$^{++}$ string, all backslashes must be doubled, so you would write "`\\\\.\\COM10`".

To close the RS232 port, use the standard Win32 function `CloseHandle`. Sending a buffer is straightforward as well, using `WriteFile`. Wrapper functions are below:

```
void Com_ClosePort(HANDLE hCom)
{
    CloseHandle(hCom);
}

size_t Com_SendData(HANDLE hCom, const unsigned char *Data, size_t DataSize)
{
    DWORD written;
    if (!WriteFile(hCom, Data, DataSize, &written, NULL))
        written = 0;
    return written;
}
```

To reset the Twinklers in a chain, the program must force a BREAK condition on the transmit line. In Microsoft Windows, the BREAK can be of arbitrary length, but 0.2 seconds is a common duration.

```
void Com_SendBreak(HANDLE hCom)
{
    SetCommBreak(hCom);
    Sleep(200);
    ClearCommBreak(hCom);
}
```

## Using VB.NET

The .Net 2.0 platform provides the SerialPort class for setting up and using a serial port. In the earlier .Net versions, programmers used the MSComm control (`MSCOMM32.OCX`), well known to Visual Basic programmers.

An application that uses the `SerialPort` class first adds the namespace to the "Imports" section near the top of the module:

```
Imports System.IO.Ports
```

Opening a COM port requires that you declare an instance of the class, initialize its fields and call the `Open` method.

```
Dim ComPort As New SerialPort

Function Com_OpenPort(ByVal PortName As String, BaudRate As Integer) As Boolean
    Try
        ComPort.PortName = PortName
        ComPort.BaudRate = BaudRate
        ComPort.Parity = Parity.None
        ComPort.DataBits = 8
        ComPort.StopBits = StopBits.One
        ComPort.Handshake = Handshake.None
        ComPort.Open()
        Com_OpenPort = ComPort.IsOpen
    Catch ex As Exception
        Com_OpenPort = False
    End Try
End Sub
```

As in C/C$^{++}$, you have to use the syntax "`\\.\COM`*xx*" for COM10 and higher.

To close a COM port, you use the `Close` method of the `SerialPort` class. To write an array of bytes to it, use the method `Write`. Wrappers are below:

```
Sub Com_ClosePort()
    ComPort.Close
End Sub

Sub Com_SendData(ByVal Data As Byte(), DataSize As Integer)
    ComPort.Write(Data, 0, DataSize)
End Sub
```

To reset the Twinklers in a chain, the program must force a BREAK condition on the transmit line. In Microsoft Windows .Net, the BREAK can be of arbitrary length, but 0.2 seconds is a common duration. The code snippet below assumes that the namespace "`System.Threading`" is in the "Imports" section.

```
Sub Com_SendBreak()
    ComPort.BreakState = True
    Thread.Sleep(200)
    ComPort.BreakState = False
End Sub
```

## Using C#

The .Net 2.0 platform provides the SerialPort class for setting up and using a serial port. An application that uses the `SerialPort` class first includes the namespace:

```
using System.IO.Ports;
```

Opening a COM port requires that you declare an instance of the class, initialize its fields and call the `Open` method.

```
static SerialPort ComPort;

private static bool Com_OpenPort(string PortName, int BaudRate)
{
    ComPort = new SerialPort(PortName, BaudRate, Parity.None, 8, StopBits.One);
    ComPort.Handshake = Handshake.None;
    try {
        ComPort.Open();
        return ComPort.IsOpen;
    } catch {
        return false;
    }
}
```

As in C/C$^{++}$, you have to use the syntax "`\\.\COM`*xx*" for COM10 and higher.

To close a COM port, you use the `Close` method of the `SerialPort` class. To write an array of bytes to it, use the method `Write`. Wrappers are below:

```
private static void Com_ClosePort()
{
    ComPort.Close();
}

private static void Com_SendData(byte[] Data, int DataSize)
{
    ComPort.Write(Data, 0, DataSize);
}
```

To reset the Twinklers in a chain, the program must force a BREAK condition on the transmit line. In Microsoft Windows .Net, the BREAK can be of arbitrary length, but 0.2 seconds is a common duration. The code snippet below assumes that the namespace "`System.Threading`" is included.

```
private static void Com_SendBreak()
{
    ComPort.BreakState = true;
    Thread.Sleep(200);
    ComPort.BreakState = false;
}
```

One of the new methods in the `SerialPort` class is the ability to list all COM ports installed on the computer (e.g. `COM1`, `COM2`, `COM4`). This is helpful if the user must pick the COM port to use.

```
    foreach (string s in SerialPort.GetPortNames())
        System.Diagnostics.Trace.WriteLine(s);
```

## In Linux, using C/C++

In Linux, as in Unix and similar operating systems, the serial port is a character device. The port is configured with functions for terminal control (specifically, `tcsetattr`).

```
HANDLE Com_OpenPort(LPCTSTR PortName, unsigned BaudRate)
{
    int fdCom;
    struct termios newtio;

    /* open the serial port device file
     * O_NDELAY   - tells port to operate and ignore the DCD line
     * O_NONBLOCK - same as O_NDELAY under Linux
     * O_NOCTTY   - this process is not to become the controlling
     *              process for the port. The driver will not send
     *              this process signals due to keyboard aborts, etc.
     */
    fdCom = open(PortName, O_RDWR | O_NOCTTY | O_NONBLOCK | O_NDELAY);
    if (fdCom < 0)
        return -1;

    memset(&newtio, 0, sizeof(newtio));

    /* B9600  - 9600 bps (may also be B57600)
     * CS8    - 8 data bits, 1 stop bit, no parity
     * CREAD  - receiver enabled
     * CLOCAL - ignore modem control lines
     */
    newtio.c_cflag = CS8 | CLOCAL | CREAD;
    switch (BaudRate) {
        case 1152000: newtio.c_cflag |= B1152000; break;
        case  576000: newtio.c_cflag |= B576000;  break;
        case  230400: newtio.c_cflag |= B230400;  break;
        case  115200: newtio.c_cflag |= B115200;  break;
        case   57600: newtio.c_cflag |= B57600;   break;
        case   38400: newtio.c_cflag |= B38400;   break;
        case   19200: newtio.c_cflag |= B19200;   break;
        case    9600: newtio.c_cflag |= B9600;    break;
        default:      return -1;
    } /* switch */
    newtio.c_iflag = IGNPAR | IGNBRK; /* ignore parity and BREAK conditions */

    if (tcsetattr(fdCom, TCSANOW, &newtio))
        return -1;

    return fdCom;
}
```

Parameter `PortName` is the full path name to the COM port. For Linux, this is “`/dev/ttyS0`” for the first physical COM port. USB virtual COM ports typically have the name “`/dev/ttyUSB0`” for the first port and “`/dev/ttyUSB1`” for the second, and so forth.

Once opened, the COM port is closed with the standard function `close` and data is sent to it with the standard function `write`. Wrappers for these functions are below:

```
void Com_ClosePort(int fdCom)
{
    close(fdCom);
}

size_t Com_SendData(int fdCom, const unsigned char *Data, size_t DataSize)
{
    return write(fdCom, Data, DataSize);
}
```

To send a BREAK condition —as required to reset a chain of Twinklers, you use, again, a terminal control function. The duration of the BREAK is implementation-dependent, but it is at least 0.2 seconds (and not more than 0.5 seconds).

```
void Com_SendBreak(int fdCom)
{
    tcsendbreak(fdCom, 0);
}
```

## Using the native API

The USB interface is based on the USB-to-serial ICs produced by FTDI. These ICs feature a native API in addition to a virtual RS232 port —provided that the drivers are correctly installed. Using the native API has the advantage that four additional pins can be read or be set. These pins are available on the USB interface on a pin header, and they are typically used for external switches or relays.

This application note does not cover the full native API. FTDI provides a document with this information on its web site (`http://www.ftdichip.com`). The document the look for is the "D2XX Programmer's Guide".

FTDI provides libraries/drivers for Microsoft Windows, Mac OS X and Linux, using a common API. This means that applications using the native API are portable (as far as the communication interface to the Twinklers is concerned). For the .Net platform, FTDI provides a "managed .Net wrapper class" for its native API, complete with Intellisense documentation. Examples for this wrapper class are in C#. Additionally, there exist third-party ports of the native API to Java, Delphi, PERL, Python and other languages. See the FTDI web site for links and up-to-date information.

This section only covers the "official" API, in C/C$^{++}$—but see page 11 for an implementation for C#. Below are code snippets that open a communication channel, close it and send data or BREAK conditions over it.

When using the native API, you must make sure that the D2XX drivers are installed. For Microsoft Windows, there is a combined installation of the VCP (virtual COM port) and D2XX drivers, for Linux, the D2XX drivers must be installed separately. See the FTDI web site for downloads and installation instructions.

```
FT_HANDLE D2XX_OpenPort(int Index, unsigned BaudRate)
{
    FT_HANDLE hFT;
    if (FT_Open(Index, &hFT) != FT_OK)
        return INVALID_HANDLE_VALUE;
    FT_ResetDevice(hFT);
    FT_SetBaudRate(hFT, BaudRate);
    FT_SetDataCharacteristics(hFT, FT_BITS_8, FT_STOP_BITS_1, FT_PARITY_NONE);
    FT_SetFlowControl(hFT, FT_FLOW_NONE, 0, 0);
    return hFT;
}

void D2XX_ClosePort(FT_HANDLE hFT)
{
    FT_Close(hFT);
}

int D2XX_SendData(FT_HANDLE hFT, const unsigned char *Data, size_t DataSize)
{
    DWORD written;
    return (FT_Write(hFT, (void*)Data, DataSize, &written) == FT_OK) ? written : 0;
}

void D2XX_SendBreak(FT_HANDLE hFT)
{
    FT_SetBreakOn(hFT);
    #if defined _WIN32
        Sleep(200);
    #else
        /* for Linux */
        usleep(200*1000);
    #endif
    FT_SetBreakOff(hFT);
}
```

An extra feature provided by the USB interface is that it has four pins to which external switches or external logic can be connected. An application can set or read these pins, using, as FTDI calls it, "CBUS Bit Bang" mode. The programming manual provided by FTDI gives the details on this mode. Below is a snippet that reads or writes the values of the four pins.

```
unsigned char D2XX_cbus(FT_HANDLE hFT, unsigned mask, unsigned bits)
{
    unsigned char result;
    unsigned char msk = (unsigned char)((mask << 4) | bits);
    FT_SetBitMode(hFT, msk, 0x20);
    FT_GetBitMode(hFT, &result);
    return (unsigned char)(result & 0x0f);
}
```

The function expects two parameters (in addition to the device handle), both of which are bit masks. The parameter `mask` determines which of the pins are "input" and which are "output". When one of the lower four bits in `mask` is set (meaning: 1), the respective pin is an output and its new value is the related bit in parameter `bits`. When a bit in parameter `mask` is clear (meaning: 0), the respective pin is an input and the related bit in parameter `bits` is ignored. The function returns the value of the input pins.

# High-level interface

This application note is distributed in combination with support code for the serial port API and the native API, plus auxiliary routines to make the handling of the protocol (whose details are in a subsequent chapter) more convenient. If you installed the Twinkler configuration & control utility, these files are in the "`support`" subdirectory of where you installed the Twinkler software. Currently, the support code covers $C^{++}$ under Microsoft Windows and Linux, and C#.

## For $C^{++}$

To use the $C^{++}$ interface, include the file `twinkler.h` in your source code. This header file defines classes for the serial interface and the native interface, both based on the same abstract base class. You will typically work with another class, simply called `Twinkler`, that envelopes the low-level classes. A skeleton program is below.

Listing: **High-level interface for C++**

```
#include "twinkler.h"

int main(void)
{
    TwinklerCom *port = new TwinklerCom;
    Twinkler twinkler(port);
    twinkler.OpenPort("com1", 9600);
    twinkler.SendBreak();
    twinkler.SetColour(0, 3, 2);
    return 0;
}
```

The `twinkler.h` file defines the classes `TwinklerCom` for serial-port IO and `TwinklerNative` for the native interface. Before opening a port, you have to decide which of the two interfaces to use, and create an instance of the corresponding object. This skeleton program will use the serial port and therefore creates a `TwinklerCom` instance. It passes this instance to the enveloping class `Twinkler`.

The skeleton program opens a port, sends a BREAK to reset all Twinklers in the chain and sets them all to the same colour. The methods defined in the `Twinkler` class are:

`bool OpenPort(const char *PortName, unsigned BaudRate)`
: Opens the port and sets the requested Baud rate. For purposes of driving the Twinklers, the Baud rate should be 9600 or 57600.

`void ClosePort()`
: Closes the port (if it was opened). The destructor of the class also closes the port. Manually closing the port is necessary when switching Baud rate.

`bool SendData(const unsigned char *Data, size_t DataSize)`
: This is a low level method to send a buffer to the Twinkler chain. For the contents of the buffer, see page 13.

`bool Reset()`
: Sets a temporary BREAK condition on the interface, causing all Twinklers in the chain to be reset.

`bool ColourArray(const unsigned char *Data, size_t DataSize)`
: Sets the Twinklers to a colour. Each Twinkler gets a colour from the consecutive element in the array `Data`, so the first Twinkler gets the colour in `Data[0]`, the second from `Data[1]` and so forth. The colours are encoded as byte values, according to the equation at page 13. Parameter `DataSize` holds the number of elements in the array.

`bool SetColour(unsigned char Colour)`
: Sets all Twinklers to the same colour. The colour is encoded as a byte value, according to the equation at page 13.

`bool SetColour(unsigned char Red, unsigned char Green, unsigned char Blue)`
: Sets all Twinklers to the same colour, now specified in red, green and blue components. The range of the parameters `Red`, `Green` and `Blue` is between 0 and 5.

`bool FadePeriod(unsigned char Period)`
: Sets the duration of the fade. See page 14 for details.

`bool BlinkRate(unsigned char Rate)`
: Sets the "quickness" of blinking. See page 14 for details.

`bool SetBlinkColour(unsigned char Colour)`
: Sets the alternate colour when blinking —blinking switches between the current colour of the Twinkler and this alternate colour. The colour is encoded as a byte value, according to the equation at page 13. The default is "black".

`bool SetState(unsigned char State)`
: Switches all Twinklers to a programmed scene (this scene must have been stored in the Twinklers earlier). See page 16 for details.

`bool SetRange(int First, int Last)`
: After applying a range, the commands listed above only affect the Twinklers in that range. The parameters `First` and `Last` are zero-based; the Twinkler indicated in `Last` is *included* in the range. To remove a range, call `SetRange` without any parameters.

`bool SetBaud(unsigned char Baud)`
: This command makes the Twinklers switch from a low Baud rate to a high Baud rate (or vice versa). The parameter must be 1 for the low Baud rate (9600 bps) and 2 for the high Baud rate (57600 bps). After sending the command, you should typically close and re-open the port. See page 16 for details.

`bool StoreState(unsigned char Hold, unsigned char State)`
: Stores a scene in the Twinklers. See page 16 for details.

`bool TickMark()`
: A tick mark synchronizes the blinking of all Twinklers. If you need synchronized blinking, your application should call this method at a 1-second interval.

`unsigned char ReadInputs()`
: Reads the auxiliary inputs on the USB interface (CBUS, see page 8). The USB interface has four additional pins and the status of these pins is returned as a bit mask. This method does internal debouncing, reading the inputs multiple times if a change is detected. When using the `TwinklerCom` interface (for a plain serial interface), this method always returns `0xff`.

## For C#

After including the `Twinkler.cs` file in you project, you get a class called `Twinkler` that is based on the interface `TwinklerIO`. Two such interfaces are available: `TwinklerCom` for the serial communications API and `TwinklerNative` for the native API.

The `TwinklerNative` class contains its own, minimal, declarations for the "unmanaged" native functions; it does not depend on the ".Net wrapper DLL" that is distributed by FTDI. The motivation for this is that otherwise the high-level interface would always require a reference to the FTDI wrapper DLL, even if only the plain serial interface is used.

A minimal example that initializes a Twinkler chain (with a full reset) and then sets all Twinklers in the chain to "turquoise" is:

```
class Program
{
    static void Main(string[] args)
    {
        TwinklerCom port = new TwinklerCom();
        Twinkler twinkler = new Twinkler(port);
        twinkler.OpenPort("com1", 9600);
        twinkler.Reset();
        twinkler.SetColour(0, 3, 2);
    }
}
```

This example program assumes that the Twinkler chain is connected to `COM1`. First, the program chooses the interface to use: `TwinklerCom` or `TwinklerNative`. This interface class is then passed-in to create an instance of the `Twinkler` class. Once the instances are created, you can use the methods of the `Twinkler` class to open the port, reset all Twinklers and set them to a colour. Note that the channel values for red, green and blue channels in method `SetColour` must be in the range 0 to 5.

The methods defined in the `Twinkler` class are:

`bool OpenPort(string PortName, uint BaudRate)`
: Opens the port and sets the requested Baud rate.

`void ClosePort()`
: Closes the port (if it was opened). The destructor of the class also closes the port. Manually closing the port is necessary when switching Baud rate.

`bool SendData(byte[] Data, int DataSize)`
: This is a low level method to send a buffer to the Twinkler chain. For the contents of the buffer, see page 13.

`bool Reset()`
: Sets a temporary BREAK condition on the interface, causing all Twinklers in the chain to be reset.

`bool ColourArray(byte[] Data, int DataSize)`
: Sets the Twinklers to a colour. Each Twinkler gets a colour from the consecutive element in the array `Data`, so the first Twinkler gets the colour in `Data[0]`, the second from `Data[1]` and so forth. The colours are encoded as byte values, according to the equation at page 13. Parameter `DataSize` holds the number of elements in the array.

`bool SetColour(byte Colour)`
: Sets all Twinklers to the same colour. The colour is encoded as a byte value, according to the equation at page 13.

`bool SetColour(byte Red, byte Green, byte Blue)`
: Sets all Twinklers to the same colour, now specified in red, green and blue components. The range of the parameters `Red`, `Green` and `Blue` is between 0 and 5.

`bool FadePeriod(byte Period)`
: Sets the duration of the fade. See page 14 for details.

`bool BlinkRate(byte Rate)`
: Sets the "quickness" of blinking. See page 14 for details.

`bool SetBlinkColour(unsigned char Colour)`
: Sets the alternate colour when blinking —blinking switches between the current colour of the Twinkler and this alternate colour. The colour is encoded as a byte value, according to the equation at page 13. The default is "black".

`bool SetState(byte State)`
: Switches all Twinklers to a programmed scene (this scene must have been stored in the Twinklers earlier). See page 16 for details.

`bool SetRange(int First, int Last)`
: After applying a range, the commands listed above only affect the Twinklers in that range. The parameters `First` and `Last` are zero-based; the Twinkler indicated in `Last` is *included* in the range. To remove a range, call `SetRange` without any parameters.

`bool SetBaud(byte Baud)`
: This command makes the Twinklers switch from a low Baud rate to a high Baud rate (or vice versa). The parameter must be 1 for the low Baud rate (9600 bps) and 2 for the high Baud rate (57600 bps). After sending the command, you should typically close and re-open the port. See page 16 for details.

`bool StoreState(byte Hold, byte State)`
: Stores a scene in the Twinklers. See page 16 for details.

`bool TickMark()`
: A tick mark synchronizes the blinking of all Twinklers. If you need synchronized blinking, your application should call this method at a 1-second interval.

`byte ReadInputs()`
: Reads the auxiliary inputs on the USB interface (CBUS, see page 8). The USB interface has four additional pins and the status of these pins is returned as a bit mask. This method does internal debouncing, reading the inputs multiple times if a change is detected. When using the `TwinklerCom` interface (for a plain serial interface), this method always returns `0xff`.

# The protocol: commands and parameters

The data that you send to the Twinkler chain consists of commands and parameters. Commands are always forwarded, so that all Twinkler boards always see the same commands. All commands and all parameters are single-byte values. The commands set is:

| **Command** | **Parameters** | **Summary** |
|---|---|---|
| `0xF0` | colour,... | New colour values for all Twinklers follows. The number of data bytes must be equal to the number of LEDs to set. |
| `0xF1` | colour | Set all Twinklers to the same colour. Only a single parameter byte follows this command. |
| `0xF2` | period | Set fade speed; the step duration for fading follows in the next byte, in increments of (approximately) 25 ms. |
| `0xF3` | cycles | Blink speed as a value between 1 (slow) and 10 (fast), or or “twinkling” speed between 11 (slow) and 14 (fast), or 0 for no blinking/twinkling. |
| `0xF4` | state | Go to stored state. The state number is in the parameter byte, starting at 1. |
| `0xF5` | range (4 bytes) | Set range. This command is followed by two bytes (carrying 7-bits each) for the start number and two bytes with the count. The next commands only apply to all Twinklers in the range. To cancel a range, send 0xF5 0x00 0x00 0x7F 0x7E (select all, for a maximum of 16,255 Twinklers). |
| `0xF6` | speed | Switch Baud rate; the next byte is 1 for 9600 bps, and 2 for 57600 bps. |
| `0xF7` | hold, state | Store current state under the scene number given in the second parameter byte (1-based). The hold time is in the first parameter byte, in increments of 0.5 s. |
| `0xF8` | | Tick mark, for clock synchronization for the Twinklers in a chain. |
| `0xF9` | colour | Alternate colour for blinking. When blinking is active, the Twinkler alternates between its current foreground colour and the colour set with this command. |

The values in the above table (and elsewhere in this chapter) are in hexadecimal, using the C/C$^{++}$ syntax. For example, `0xF0` is the decimal value 240.

## Setting colours

The commands `0xF0` and `0xF1` carry colour data. The colour values for the LEDs are the values for the red, green and blue channels combined in a single byte. The values for the channels are in the range 0 to 5, where 0 is off, and 5 is full brightness. The channels are combined in a value according to the equation:

$$c = 36 \times r + 6 \times g + b$$

The value of c is between 0 and 215 (inclusive).

To set an individual colour for all Twinklers in a chain, send the command byte `0xF0` followed by the colour bytes of all Twinklers. The first Twinkler will get the colour of the byte that you send first, the next Twinkler gets second byte, and so on.

Setting all Twinklers to the same colour is simpler: send the command `0xF1` followed by a single byte with the colour.

To change the colours for only a subset of the Twinklers in a chain, you need to select a range (command `0xF5`), followed by either command `0xF0` or `0xF1`. See the section on the range command for an example, on page 14.

## Fading and blinking/twinkling

When setting new colours for the Twinklers, each Twinkler can optionally fade its current colour to the new colour. The delay between fading steps is in increments of approximately 25ms. The number of steps in the fade depends on the old and new colours —more specifically: the number of steps is the difference in the red, green and blue channels of the old and new colours. For example, if fade step duration is set to 8 (command `0xF2`, parameter `0x08`), each fade step will take roughly 0.2 second ($8 \times 25$ms). A fade from off to full on (five steps) then completes in 1 second.

The blink speed is a value between 0 and 10, with 0 for no blinking/twinkling, 1 for slow blinking and 10 for quick blinking. At the blink setting of 1, the full blink cycle takes three seconds; at the blink setting of 10, there are six blink cycles in a second. At the blink setting 3, a blink cycle takes exactly one second.

When the value for the blink speed is between 11 and 14, the effect changes to “twinkling”. The twinkle effect is like blinking with pseudo-random intervals. Each Twinkler acquires a seed for the pseudo-random number generator from a non-linear source. The Twinklers in a chain will therefore blink out of sync. The pseudo-random blinking is also asymmetric: the LED’s average “on” time is longer that its average “off” time.

By default, Twinklers blink by alternatingly turning on (to their current colour) and off. Instead of turning off, you can also set a “blink colour” with command `0xF9`. If set, the Twinkler will alternate between its current (foreground) colour and the blink colour.

The blink/twinkle routine takes fading into account. When blinking and fading are both active, the blink cycle duration should be at least five times bigger than the fade duration, to ensure that the fade actually reaches its destination colour. This is less relevant to the twinkle effect, as reversing the fade destination midway makes the twinkle effect more realistic.

When changing a blink parameter, the Twinkler first finishes its current blink cycle before changing the blink rate. When changing a fade parameter and a fade is currently in progress, that fade is cancelled, and the new parameter will be used on the next fade.

To synchronize blinking of all Twinklers in a chain, see the section “Clock synchronization” further on in this chapter. Clock synchronization is redundant for the twinkle effect —the Twinklers are, by intent, asynchronous while twinkling.

## Applying a range

By default, every command runs over a complete chain of Twinklers. When turning blinking on, for example, the all Twinklers in a chain will blink. To apply blinking (or fading, or a colour change) to only a subset of the Twinklers in a chain, you must first send a “range” command

(`0xF5`) and then the commands that apply only to that range. Command `0xF5` must be followed by four parameter bytes. The first two bytes are for the sequential number of the first Twinkler in the range, counting from zero. The first byte holds the low 7 bits (bits 0..6) of this number and the second byte the high bits (bits 7..13). The last two bytes hold the count of Twinklers in the range, again split in two bytes of seven bits.

To select a sequence of ten Twinklers starting at the fifth Twinkler in a chain, the full command sequence would be “`0xF5 0x05 0x00 0x0A 0x00`”.

A range is active until it is overruled by a new range. To cancel a range completely, select the full range, starting from zero. The command to select all Twinklers in a chain is: “`0xF5 0x00 0x00 0x7F 0x7E`”. Note that the final value must be `0x7E` (not `0x7F`). This command selects up to 16,255 Twinklers starting from the first. The implication is that a chain of Twinklers should not be longer than 16,255 units.

A function that sets a range of Twinklers in a chain to one specific colour is below (this example is in C/C$^{++}$, and it assumes the native API). The most complex part of the function is the decomposing of the `first` and `last` parameters into byte pairs holding 7-bits in each byte. The function does not clear the range, if you wish to clear the range after sending the colour, you need to issue another `0xF5` command.

```
void SetRangeColour(FT_HANDLE hFT, int first, int last, unsigned char colour)
{
    unsigned char buffer[10];

    /* set the range */
    buffer[0] = 0xf5; /* "set range" command */
    buffer[1] = first & 0x7f;
    buffer[2] = (first >> 7) & 0x7e;
    buffer[3] = (last - first) & 0x7f;
    buffer[4] = ((last - first) >> 7) & 0x7e;
    D2XX_SendData(hFT, buffer, 5);

    /* set the colour */
    buffer[0] = 0xf1; /* "set single colour" command */
    buffer[1] = colour;
    D2XX_SendData(hFT, buffer, 2);
}
```

## Clock synchronization

Each Twinkler has its own internal clock to handle fading and blinking. With multiple Twinklers in a chain, minor variations in the oscillator frequency (which drive the clock) cause the clocks to drift. The blinking effects of the Twinklers might then go out of sync.

To offset clock drift, you can insert “Tick mark” commands on the serial bus (command `0xF8`). These tick marks should be sent out every second. All Twinklers in a chain receive the tick marks and synchronize their clock on them.

In absence of an application driving the Twinklers, the first Twinkler in a chain takes on the responsibility of generating the tick mark commands. However, the design of the Twinkler serial bus is such that there may be only one sender (and multiple receivers). So if an application drives the Twinklers, and synchronized blinking is desired, the application should generate the tick mark commands.

For an even blink rate, the tick marks should be spaced apart 1 second, as precisely as possible. When an application stops generating tick marks, blinking or fading will still continue to operate normally, but if the application stops generating ticks marks for a fairly long period (more than 60 seconds), clock drift may become noticible.

Clock synchronization is also important for serial communication, especially when the Twinklers are installed at a location where the temperature may vary and when using high baud rates. The Twinklers use the regular interval of the tick marks to calibrate their oscillators to, so that all oscillators are synchronized to that of the device that transmits the tick marks.

## Resetting the Twinklers

A BREAK condition on the serial bus resets the Twinklers. Upon detection of a BREAK condition, each Twinkler resets and forces a BREAK condition on its output port, in order to cascade the BREAK over the serial bus. Each Twinkler board stays in "reset" (and keeps a BREAK on its output port asserted) as long as the BREAK condition exists at its input port.

The BREAK condition on the output port is at least 100ms. A BREAK condition on the input port should be 10ms or longer. Modern operating systems have system calls to set a BREAK condition; a typical duration is 0.2 second (200ms). See also the code snippets in the chapters "Using the serial port API" and "Using the native API" for setting a BREAK condition.

A BREAK condition will be detected by the Twinkler boards regardless of the Baud rate that is active at the time. After reset, the Baud rate of a Twinkler has switched back to its default: 9600bps.

Depending on the configuration of the Twinklers, there may be a visible start-up sequence after the resume from a BREAK condition, see the Twinkler "User Guide" and appendix A for details. This start-up sequence takes approximately 200ms. It is therefore advised to wait at least 250ms before sending a first command, after reset.

## Changing the Baud rate

To change the Baud rate from the default 9600bps to 57600bps, send command byte `0xF4` followed by parameter byte `0x02`. Wait at least 25ms before sending any data to the Twinklers after changing the Baud rate. When the LEDs are on, the change in Baud rate may cause a very brief flicker of the LED. It is (therefore) advised to change the Baud rate after reset of the Twinklers.

Note that after having switched the Twinklers to another Baud rate, you must close the communication channel and re-open it with the new Baud rate. Also note that after a reset, the Twinklers drop back to the low Baud rate of 9600 bps. There are therefore two ways to change the Baud rate from 57600 bps back to 9600 bps: either issue command `0xF4` with parameter 1, or send a BREAK condition to reset all Twinklers.

## Storing scenes

Scenes (colour states plus fading and blinking/twinkling parameters) can be stored on the Twinklers themselves. Instead of sending colour information for all Twinklers, a controller then only has to send a single command/parameter pair (two bytes), to switch all Twinklers in a chain to a scene.

To store a colour on the Twinklers, first set all Twinklers to the colour that needs to be saved. Optionally, you can also set the blink interval (or twinkle effect) and the fade duration for the Twinklers, or a selected subset of the Twinklers. Then issue command `0xF7` followed by the hold time and the index number for the state. The index number must be between 1 and 64 (inclusive). This command makes the Twinkler store the current values for the colour, the blink interval and the fade duration under the given index number. The hold time is in increments of 0.5 second, so a value of 10 means to hold the scene for 5 seconds. When the hold time is zero, the scene will not time-out.

The same command is also used to erase the table, using a special (otherwise invalid) parameter. When sending the command `0xF7` followed by `0x00` and `0xDF` (hold time 0, index `0xDF`), the table for all states is erased.

After sending the "index" byte for the command `0xF7`, a delay of at least 12ms must be taken into account before sending a next command. When erasing the entire table, (command `0xF7`, hold time 0, index byte `0xDF`), a delay of at least 1.5 seconds is needed.

After power-on and after a reset, the Twinkler boards automatically switch to state index 1, if it is defined. If state 1 is undefined, the Twinkler waits for a serial command.

When the hold times are valid (meaning non-zero), a daisy-chain of Twinklers can run a scenario without requiring a controller. The chain will restart the show after every reset. As explained in the section "Resetting the Twinklers", pulling the serial input line low for approximately 0.1 second forces a reset (use the serial input line of the first Twinkler of a chain). The power repeater boards have a jumper block for this purpose.

APPENDIX A

# Internals

After reset, if a Twinkler sees a valid scene in the first slot, it branches to that first state. Each Twinkler will also forward the state that it selects to the following Twinkler in the chain. The Twinklers in a chain will therefore assume the colour (plus the fading and blinking effects) of that first scene.

The Twinklers will also transmit the "tick mark" command (`0xF8`) in the case that a valid scene table exists. The tick mark command is repeated at an interval of 1 second. After start-up or reset, this command is generated by the timer of each Twinkler. However, when a Twinkler receives a "tick mark" command, it will stop generating this command itself and instead forward the command that it receives. Consequently, within a few clock ticks, only one Twinkler in a chain remains generating tick marks: all Twinklers will have received the tick mark from their predecessor in the chain, *except the first.* This first Twinkler becomes the "master" Twinkler of the chain: not only does it generate the tick marks on which all Twinklers synchronize, it also sends out the commands to switch to a new scene (after the "hold time" for the active scene expires).

# Index

◇ Names of persons or companies (not products) are in *italics*.
◇ Function names, constants and compiler reserved words are in `typewriter font`.

!

.Net, 3, 4, 7, 11

A

API, *see* Native API or Serial port API or High-level interface

B

Baud rate, 16
  synchronization, 16
Bit bang, 8
Blink
  ~ colour, 14
  ~ speed, 14
  ~ synchronization, 14, 15
`BlinkRate`, 10, 12
Break condition, 3–5, 9, 11, 16

C

C++, 2, 5, 9
C#, 4, 7, 11
CBUS, 8, 10, 12
Chain of Twinklers, 1
Clock drift, 15
Clock synchronization, 15, *see also* Tick mark
`CloseHandle`, 2
`ClosePort`, 9, 11
Colour
  blink ~, 14
  setting ~, 13
`ColourArray`, 10, 11
COMM-DRV, 2
`CreateFile`, 2

D

D2XX, *see* Native API
D2XX drivers, 7
Debounce (switches), 10, 12
Delphi, 7
Diagnostics, *see* Start-up diagnostics

F

`FadePeriod`, 10, 12
*FTDI*, 7, 8, 11

H

Hexadecimal, 13
High-level interface, 9

J

Java, 7

L

Linux, 5, 7
List COM ports, 4

M

Macintosh, 7
Managed class, 7
Maximum chain length, 15
MSComm Control, 3

N

Namespace, 3
Native API, 7, 9

O

`OpenPort`, 9, 11
OS X (Mac), 7

P

Python, 7

R

Range, 14
  clear ~, 15
`ReadInputs`, 10, 12
Reset, 3–5, 16
`Reset`, 9, 11
RS232 API, *see* Serial port API
RS232 interface, 1

S

Scene table, 16
  erase ~, 17
`SendData`, 9, 11
Serial bus, 1, *see also* Twinkler chain
Serial port API, 2
`SerialPort`, 3, 4
`SetBaud`, 10, 12
`SetBlinkColour`, 10, 12
`SetColour`, 10, 11
`SetCommBreak`, 3
`SetRange`, 10, 12
`StoreState`, 10, 12
Support code, 9
Synchronized blinking, 15

T

`tcsendbreak`, 5
`tcsetattr`, 5
Temperature compensation, 16
Tick mark, 15, 18
`TickMark`, 10, 12
TTL logic levels, 1
Twinkler
~ chain, 1
`Twinkler class`, 9, 11
Twinkling, 14

U

USB interface, 1

V

Visual Basic, 3

W

*WCSC*, 2
Wrapper DLL, 11
`WriteFile`, 2