

Polychrome 3000 Software Development Kit



Agilent Technologies
Lochamer Schlag 19
82166 Gräfelfing
Germany

The Polychrome 3000 Software Development Kit, including this documentation, is produced by Bruyton Corporation, and is copyright 2007-2008 by Bruyton Corporation, Seattle, Washington, USA. The Polychrome 3000 Software Development kit is licensed to Agilent Technologies, Gräfelfing, Germany.

Table of Contents

Introduction	4	TILLOligochrome_GetFilterDescription TILLOligochrome_GetStatus	
Polychrome 3000	4	Actions	17
Release	4	TILLOligochrome_DigitalInputRead TILLOligochrome_DigitalOutputWrite TILLOligochrome_NoOperation TILLOligochrome_SetFilterPosition	
Function Naming	4	Variables	19
Overview	5	TILLOligochrome_VariableDeclareInteger TILLOligochrome_VariableDeclareString TILLOligochrome_VariableFree TILLOligochrome_VariableReadInteger TILLOligochrome_VariableReadString TILLOligochrome_VariableWriteInteger TILLOligochrome_VariableWriteString	
Getting Started	5	Arithmetic	20
Open and Close Identification Error Reporting		TILLOligochrome_VariableAppendString TILLOligochrome_Variable... Arithmetic TILLOligochrome_VariableIncrement TILLOligochrome_VariableDecrement TILLOligochrome_Variable... Comparisons	
Operations	6	Events	22
Filter Position Digital Input and Output		TILLOligochrome_EventDefineDigitalInput TILLOligochrome_EventDefineJumper TILLOligochrome_EventDefineSoftware TILLOligochrome_EventDefineWait TILLOligochrome_EventDefine... Comparisons TILLOligochrome_EventFree TILLOligochrome_EventSoftware	
Protocols	6	Protocol Control	24
Simple Protocol Loops Responses		TILLOligochrome_ProtocolBegin TILLOligochrome_ProtocolEnd TILLOligochrome_ProtocolCancel TILLOligochrome_ProtocolDelayEvent TILLOligochrome_ProtocolDelayTime TILLOligochrome_ProtocolDelayTrigger TILLOligochrome_ProtocolLoopBegin TILLOligochrome_ProtocolLoopEnd TILLOligochrome_ProtocolLoopDisable TILLOligochrome_ProtocolLoopEnd TILLOligochrome_ProtocolLoopEnable TILLOligochrome_ProtocolLoopGetIndex TILLOligochrome_ProtocolIf TILLOligochrome_ProtocolElse TILLOligochrome_ProtocolEndIf	
Timing	9		
Duration Timer			
Control	10		
Variables Arithmetic Events Conditionals			
C/C++ API	13		
Management	13		
TILLOligochrome_Create TILLOligochrome_OpenPort TILLOligochrome_Close TILLOligochrome_GetLastError			
Information	14		
TILLOligochrome_GetConfiguration TILLOligochrome_GetFeatures TILLOligochrome_GetIntervalResolution			

Protocol Actions 27

TILLOligochrome_ProtocolDigitalInputRead
 TILLOligochrome_ProtocolDigitalOutputWrite
 TILLOligochrome_ProtocolExperimentTimerRead
 TILLOligochrome_ProtocolExperimentTimerReset
 TILLOligochrome_ProtocolMark
 TILLOligochrome_ProtocolNoOperation
 TILLOligochrome_ProtocolSetFilterPosition

Protocol Variable 29

TILLOligochrome_ProtocolVariableReadInteger
 TILLOligochrome_ProtocolVariableReadString
 TILLOligochrome_ProtocolVariableWriteInteger
 TILLOligochrome_ProtocolVariableWriteString

Protocol Arithmetic 29

TILLOligochrome_ProtocolVariableAppendString
 TILLOligochrome_ProtocolVariable... Arithmetic
 TILLOligochrome_ProtocolVariableIncrement
 TILLOligochrome_ProtocolVariableDecrement
 TILLOligochrome_Protocol... Comparisons

Protocol Response 31

TILLOligochrome_GetResponse
 TILLOligochrome_ResponseDigitalInputRead
 TILLOligochrome_ResponseExperimentTimerRead
 TILLOligochrome_ResponseExperimentTimerReset
 TILLOligochrome_ResponseGetLoopIndex
 TILLOligochrome_ResponseMark
 TILLOligochrome_ResponseVariableReadInteger
 TILLOligochrome_ResponseVariableReadString

Introduction

Polychrome 3000

The Polychrome 3000 is a compact light source for microscope illumination based on rapid switching between up to 5 filters. The Polychrome 3000 offers timing controlled by a host computer or by digital trigger signals.

This document describes the Polychrome 3000 Software Development Kit (SDK). The SDK allows applications running on a host computer to control one or more Polychrome 3000 devices. The controlling application can instruct the Polychrome 3000 to select a specific filter, or to select a filter when triggered by external digital signals. This document describes the software functions used by a computer application to control the Polychrome 3000.

Release

This document describes the Polychrome 3000 SDK Release 1.

Function Naming

In this document, the application interface to the Polychrome 3000 software development kit has the prefix 'TILLOligochrome_' on each function call, as in 'TILLOligochrome_OpenPort'. This is for historical reasons. In the future, the SDK will be available with a more appropriate name. The existing application interface calls will continue to be supported in the future.



Overview

Getting Started

Open and Close

Use TILLOligochrome_Create to create an Polychrome 3000 handle, and TILLOligochrome_Close to discard the handle. Use TILLOligochrome_OpenPort to open an Polychrome 3000:

```
void* handle;
int status;
status = TILLOligochrome_Create(&handle);
if (status != 0)
    ... // error
status = TILLOligochrome_OpenPort(handle, "COM4");
if (status != 0)
    ... // error
...
TILLOligochrome_Close(handle);
```

Notice that all operations use a '*handle*'. The handle is created by TILLOligochrome_Create, and released by TILLOligochrome_Close.

Identification

Use TILLOligochrome_GetConfiguration to obtain identification information about the Polychrome 3000:

```
char model[51];
char identification[13];

TILLOligochrome_GetConfiguration(handle, model, sizeof(model), identification, sizeof(identification));
```

This information may be helpful to a user, or useful when reporting a problem.

Use TILLOligochrome_GetFeatures to determine the Polychrome 3000 features, including the number of filter positions.

Use TILLOligochrome_GetIntervalResolution to determine the timing interval resolution.

Error Reporting

Use TILLOligochrome_GetLastError to obtain a text message corresponding to the most recent error:

```
status = TILLOligochrome_...(handle, ...);
if (status != 0)
{
    char message[512]
    void TILLOligochrome_GetLastError(handle, message, sizeof(message));
    ...
}
```

Notice that the return message may be very long if it includes a long file path. However, most messages are short.

Operations

Filter Position

The fundamental Polychrome 3000 operation is the ability to set a filter position:

```
int TILLOligochrome_SetFilterPosition(void* handle, int filter_position, double intensity);
```

The filter positions are numbered 1 to 5, and intensity is expressed as a fraction of full intensity for the filter. The intensity value is in the range 0 to 1.0.

Digital Input and Output

The Polychrome 3000 can read digital inputs, and write digital outputs. These operations are typically used to synchronize with or to control external electronic devices:

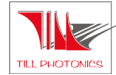
```
int TILLOligochrome_DigitalInputRead(void* handle, int bank, int mask, int* value);
```

```
int TILLOligochrome_DigitalOutputWrite(void* handle, int bank, int mask, int value);
```

The back panel of the Polychrome 3000 has a connector for trigger input and another connector for trigger output. The trigger input is one bit of the digital inputs, the trigger output is one bit of the digital outputs.

Protocols

An application records a protocol in the Polychrome 3000. The Polychrome 3000 then executes the protocol without intervention from the application.



Simple Protocol

A protocol is started using `TILLOligochrome_ProtocolBegin`, and ends with `TILLOligochrome_ProtocolEnd`:

```
TILLOligochrome_ProtocolBegin(handle);
TILLOligochrome_ProtocolSetFilterPosition(handle, 1, 1.0);
TILLOligochrome_ProtocolDigitalOutputWrite(handle, 1, 0x01, 0x01);
TILLOligochrome_ProtocolEnd(handle);
```

The protocol begins executing immediately after the application sends `TILLOligochrome_ProtocolEnd`.

Notice that the protocol operations all have the prefix 'TILLOligochrome_Protocol', to show that the operations will execute as part of a protocol. Many of the protocol operations have different parameters than the equivalent immediate operations.

Loops

A loop is bracketed by `TILLOligochrome_ProtocolLoopBegin` and `TILLOligochrome_ProtocolLoopEnd`:

```
TILLOligochrome_ProtocolLoopBegin(handle, count);
...
TILLOligochrome_ProtocolLoopEnd(handle);
```

The `TILLOligochrome_ProtocolLoopBegin` *count* parameter is the number of times the loop will be executed.

Loops can be nested:

```
TILLOligochrome_ProtocolLoopBegin(handle, count_1);
...
    TILLOligochrome_ProtocolLoopBegin(handle, count_2);
    ...
    TILLOligochrome_ProtocolLoopEnd(handle);
...
TILLOligochrome_ProtocolLoopEnd(handle);
```

The outer loop, at loop nesting level 1, is executed *count_1* times. The inner loop, at loop nesting level 2, is executed *count_2* times.

A loop can be terminated using `TILLOligochrome_ProtocolLoopDisable`:

```
TILLOligochrome_ProtocolLoopBegin(handle, count);
...
    TILLOligochrome_ProtocolLoopDisable(handle);
    ...
TILLOligochrome_ProtocolLoopEnd(handle);
```

After `TILLOligochrome_ProtocolLoopDisable` executes, each `TILLOligochrome_ProtocolLoopEnd` operation



executed terminates the associated loop, regardless of any remaining loop repetition count. Usually, TILLOligochrome_ProtocolLoopDisable is executed within a conditional block.

TILLOligochrome_ProtocolLoopDisable sets a mode that causes all executed TILLOligochrome_ProtocolLoopEnd operations to terminate the associated loop, until TILLProtocolLoopEnable is executed:

```
TILLOligochrome_ProtocolLoopBegin(handle, count_1);
...
TILLOligochrome_ProtocolLoopBegin(handle, count_2);
...
TILLOligochrome_ProtocolLoopDisable(handle);
...
TILLOligochrome_ProtocolLoopEnd(handle);
...
TILLOligochrome_ProtocolLoopEnable(handle);
...
TILLOligochrome_ProtocolLoopEnd(handle);
```

Given the placement of TILLOligochrome_ProtocolLoopEnable in the example above, the TILLOligochrome_ProtocolLoopDisable will terminate the inner loop at nesting level 2, but not the outer loop at nesting level 1.

Responses

A protocol operation cannot read information directly from the Polychrome 3000. Consider reading the digital inputs using an immediate operation:

```
int TILLOligochrome_DigitalInputRead(void* handle, int bank, int mask, int* value);
```

The TILLOligochrome_DigitalInputRead operation directly returns the *value* read from the specified Polychrome 3000 digital input *bank*. The corresponding protocol operation cannot return a value, because the value is not available until the protocol executes:

```
int TILLOligochrome_ProtocolDigitalInputRead(void* handle, int bank, int mask);
```

The TILLOligochrome_ProtocolDigitalInputRead operation writes the resulting digital input value to a queue. The operation TILLOligochrome_ResponseDigitalInputRead reads the value:

```
int TILLOligochrome_ResponseDigitalInputRead(void* handle, int* bank, int* value);
```

Each protocol operation that can return a value has a corresponding TILLOligochrome_Response operation to read the value.

When a protocol executes, the application uses TILLOligochrome_GetResponse to determine when a response is available to be read from the Polychrome 3000. TILLOligochrome_GetResponse indicates the type of response to read. For example:


```
// Create protocol
TILLOligochrome_ProtocolBegin(handle);
TILLOligochrome_ProtocolDigitalInputRead(handle, 1, 0x01);
TILLOligochrome_ProtocolEnd(handle);

// Monitor protocol execution
bool executing = true;
while (executing)
{
    TILLOligochrome_GetResponse(handle, &response);
    switch (response)
    {
        case TILL_OLIGOCHROME_RESPONSE_PROTOCOL_END:
            executing = false;
            break;
        case TILL_OLIGOCHROME_RESPONSE_DIGITAL_INPUT_READ:
            TILLOligochrome_ResponseDigitalInputRead(handle, &bank, &value);
            break;
        default:
            break;
    }
    ...
}
```

Timing

The Polychrome 3000 maintains a high-resolution hardware timer, with a resolution of 10µs. The resolution can be read using the Polychrome 3000 SDK function `TILLOligochrome_GetIntervalResolution`. An application can use this timer to precisely control the duration of Polychrome 3000 operations.

Duration

If a sequence of operations does not specify timing, the Polychrome 3000 performs operations successively with no time delay.

For example, consider the sequence:

1. Assert digital output bank 1, bit 0, the Polychrome 3000 back panel trigger output signal
2. Set filter position 2, full intensity (1.0)
3. Deassert digital output bank 1, bit 0

This is implemented using the commands:



```
TILLOligochrome_ProtocolDigitalOutputWrite(handle, 1, 0x01, 0x01);
TILLOligochrome_ProtocolSetFilterPosition(handle, 2, 1.0);
TILLOligochrome_ProtocolDigitalOutputWrite(handle, 1, 0x01, 0x00);
```

Suppose that the trigger output must be asserted 0.5ms before changing the filter position, and must remain asserted for 2ms following the change in filter position. In this case, delay operations can be added to cause the operations to be performed in sequence:

```
TILLOligochrome_ProtocolDigitalOutputWrite(handle, 1, 0x01, 0x01);
TILLOligochrome_ProtocolDelayTime(handle, 0.0005);
TILLOligochrome_ProtocolSetFilterPosition(handle, 2, 1.0);
TILLOligochrome_ProtocolDelayTime(handle, 0.002);
TILLOligochrome_ProtocolDigitalOutputWrite(handle, 1, 0x01, 0x00);
```

Timing can be based either on a timer, or on events.

Timer

The operation TILLOligochrome_ProtocolDelayTime sets the delay from the start of the preceding operation to the start of the succeeding operation:

```
TILLOligochrome_ProtocolA
TILLOligochrome_ProtocolDelayTime(handle, 1.0);
TILLOligochrome_ProtocolB
```

In this case, the operation TILLOligochrome_ProtocolB begins 1.0 seconds after the start of TILLOligochrome_ProtocolA.

The timer operates to a resolution of 10µs, as specified by TILLOligochrome_GetIntervalResolution. The maximum delay is approximately 21,000 seconds, or 5.9 hours.

Control

Variables

The Polychrome 3000 supports integer and string variables. To create a variable, use TILLOligochrome_VariableDeclareInteger or TILLOligochrome_VariableDeclareString. The operation creates the variable, sets the initial value of the variable, and returns with a handle that can be used to refer to the variable.

The Polychrome 3000 can store only a limited number of integer and string variables at one time, so an application should release variables when they are no longer needed. To release a variable, use TILLOligochrome_VariableFree.



```
int variable_a;
TILLOligochrome_VariableDeclareInteger(handle, &variable_a, 100);
...
TILLOligochrome_VariableFree(handle, variable_a);
```

Arithmetic

The Polychrome 3000 can perform arithmetic on integer variables. The simplest arithmetic operation is an increment or decrement:

```
TILLOligochrome_ProtocolLoopBegin(handle, 100);
...
TILLOligochrome_ProtocolDecrement(handle, variable_a);
TILLOligochrome_ProtocolLoopEnd(handle);
```

Notice that a variable may be used in a protocol, but the variable must be declared before the protocol is built.

The Polychrome 3000 can also perform arithmetic and conditional operations using integer variables. The operation is performed between two variables, with results written into a third:

```
int variable_a;
int variable_b;
int variable_c;
int variable_d;
TILLOligochrome_VariableDeclareInteger(handle, &variable_a, 1000);
TILLOligochrome_VariableDeclareInteger(handle, &variable_b, 3000);
TILLOligochrome_VariableDeclareInteger(handle, &variable_c, 0);
TILLOligochrome_VariableDeclareInteger(handle, &variable_d, 0);
...
TILLOligochrome_ProtocolBegin(handle);
TILLOligochrome_ProtocolAdd(handle, variable_a, variable_b, variable_c);
TILLOligochrome_ProtocolGreaterGreater(handle, variable_a, variable_c, variable_d);
```

After this code is executed, the value of variable_c will be 4000 (= 1000+3000), and variable_d will contain 0 (false), because the value of variable_a is not greater than the value of variable_c.

Events

An Polychrome 3000 protocol can use events to control timing. To use an event, an application must first define the event. For example, an event might be the back panel input trigger line asserted (high):

```
int event_trigger;
TILLOligochrome_EventDefineDigitalInput(handle, &event_trigger, 1, 0x01, 0x01);
```

The TILLOligochrome_EventDefine... operations create events, and return a handle to the event. The Poly-

chrome 3000 can support only a limited number of events defined at the same time, so when an event is no longer needed it should be released using `TILLOligochrome_EventFree`:

```
TILLOligochrome_EventFree(handle, event_trigger);
```

A protocol can wait for an event to occur:

```
TILLOligochrome_ProtocolDelayEvent(handle, event_trigger);  
TILLOligochrome_ProtocolSetFilterPosition(handle, filter_position, 1.0);
```

Conditionals

A protocol can perform conditional operations using `TILLOligochrome_ProtocolIf`. A conditional is based on an event:

```
TILLOligochrome_ProtocolIf(handle, event_trigger);  
...  
TILLOligochrome_ProtocolEndIf(handle);
```

A conditional can even include an else block:

```
TILLOligochrome_ProtocolIf(handle, event_trigger);  
...  
TILLOligochrome_ProtocolElse(handle);  
...  
TILLOligochrome_ProtocolEndIf(handle);
```

C/C++ API

This section describes the function calls provided by the C/C++ API for the Polychrome 3000.

Management

The operations listed in this section are used to manage the connection between the application and the Polychrome 3000.

TILLOligochrome_Create

```
int TILLOligochrome_Create(void** handle);
```

This operation creates a Polychrome 3000 object for use with other functions, and write a 'void*' *handle* to the object. The object is discarded when TILLOligochrome_Close is called.

TILLOligochrome_OpenPort

```
int TILLOligochrome_OpenPort(void* handle, const char* port);
```

This operation opens a Polychrome 3000 on the specified *port*, returning a *handle* that can be used to access the Polychrome 3000 for further operations. The application program should discard the *handle* when done by calling TILLOligochrome_Close.

The *handle* is a pointer to a 'void*' value to receive the handle of the open Polychrome 3000. The *port* is a text string containing the name of the communications port to use. For example, under Microsoft Windows, a typical value might be 'COM5;'.

An application should not use TILLOligochrome_OpenPort to open a Polychrome 3000 that is already open on the same port.

TILLOligochrome_Close

```
void TILLOligochrome_Close(void* handle);
```

This operation closes the *handle* to the Polychrome 3000. The operation always succeeds. After this operation is called, the *handle* is no longer valid.



TILLOligochrome_GetLastError

```
void TILLOligochrome_GetLastError(void* handle, char* text, int text_size);
```

This operation returns a text error message corresponding to the most recent error. The most recent error is the most recent TILLOligochrome_... call that returned a non-zero status value.

The operation writes the text string into *text*. The operation will not write a string of more than *size* characters. If *size* is zero, the operation does nothing. The operation terminates the *text* string with a null. The *size* value should be at least 80 characters to accommodate status text strings.

The *handle* is the handle of the open Polychrome 3000.

Information

This operations in this section are used to obtain information about the Polychrome 3000.

TILLOligochrome_GetConfiguration

```
void TILLOligochrome_GetConfiguration(  
    void* handle,  
    char* model,  
    int model_size,  
    char* identification,  
    int identification_size);
```

This operation provides configuration information about an open Polychrome 3000 unit. For information about the command set and features, see the operation TILLOligochrome_GetFeatures.

The *handle* is the handle of an open Polychrome 3000. The return value is the operation status, zero if the operation succeeded.

The operation writes a text string identifying the device model into the character array at *model*. The model string is typically of the form 'Agilent Technologies Polychrome 3000 ...'. The string represents the specific model, including the construction level of the hardware and the firmware version. The text string is intended for display to a user. An application program may find it convenient to report the model string for support purposes. An application program should not attempt to decode the model string. The model string is never longer than 50 characters, 51 characters including the terminating null.

The operation never writes more than *model_size* characters into the *model* string. If *model_size* is smaller than the number of characters required to hold the entire model description, the operation fails. The operation never returns a truncated model string.

The operation writes the identification text string for the Polychrome 3000 into the character array at *identification*. The identification text string is typically the serial number of the Polychrome 3000. The identi-



ification string uniquely identifies the Polychrome 3000, and is never longer than 12 characters, that is, 13 characters including the terminating null.

The operation never writes more than *identification_size* characters into the *identification* buffer. If *identification_size* is smaller than the number of characters required to hold the unique identification string, the operation fails. The operation never returns a truncated identification string.

TILLOligochrome_GetFeatures

```
void TILLOligochrome_GetFeatures(
    void* handle,
    int* filter_position_count,
    int* UART_port_count,
    int* focus_clamp);
```

This operation provides feature information about an open Polychrome 3000 unit.

The *handle* is the handle of an open Polychrome 3000. The return value is the operation status, zero if the operation succeeded.

The operation writes an integer number of filter positions into the *filter_position_count*. For the Polychrome 3000, this value is always 5.

The operation writes the number of available UART ports into the *UART_port_count*. For an Polychrome 3000, this value is always 3 or 4, the specific value depending on the model. The UART ports are numbered 1 to *UART_port_count*. The Polychrome 3000 always returns zero for this value.

The operation writes the value TRUE into *focus_clamp* if the Polychrome 3000 supports the focus clamp mode, and the value FALSE if the Polychrome 3000 does not.

TILLOligochrome_GetIntervalResolution

```
void TILLOligochrome_GetIntervalResolution(void* handle, double* resolution, double* maximum)
```

This operation returns the time interval resolution supported by the Polychrome 3000 when specifying timing.

The *handle* is the handle of an open Polychrome 3000. The return value is the status code, zero if the operation succeeded. The operation will always succeed, if the *handle* is valid.

The operation writes the timing units into *resolution*. The timing units are expressed as a number of microseconds. For the Polychrome 3000, this value is always 10, that is, timing is specified in with a resolution of 10 μ s. For example, if an application specifies a timing interval as 28 μ s, the actual interval used by an Polychrome 3000 will be the closest available value given the resolution of 10 μ s, that is, 30 μ s.

The operation writes the maximum timing interval into *maximum*. For a 32-bit program, this value will be more than 20,000 seconds, that is, more than 5 hours. For a 64-bit program, this value may be more than

42,000 seconds, that is, almost 12 hours.

TILLOligochrome_GetFilterDescription

```
int TILLOligochrome_GetFilterDescription(
    void* handle,
    int filter_position,
    char* filter_name,
    int filter_name_size,
    int* filter_status,
    int* filter_type,
    double* filter_wavelength,
    double* filter_bandwidth);
```

This operation provides information about a specific filter position.

The information returned by this operation is written into the Polychrome 3000 by other software. The Polychrome 3000 hardware cannot detect the filter installed in the specified filter position. The filter description information is usually not present, and if present, may be inaccurate.

The *handle* is the handle of an open Polychrome 3000. The return value is the operation status, zero if the operation succeeded.

The *filter_position* is the integer index of the filter position, in the range 1 to 5.

The operation writes a text string identifying the filter into the character array at *filter_name*. If the filter description is not valid, the *filter_name* is an empty string (a null). The *filter_name* string is never longer than 20 characters, 21 characters including the terminating null.

The operation never writes more than *filter_name_size* characters into the *filter_name* string. If *filter_name_size* is smaller than the number of characters required to hold the entire filter name, the operation fails. The operation never returns a truncated string.

The *filter_status* is an integer value to receive the status of the filter position. The defined status values are:

<i>filter_status</i>	Status
0	no filter status
1	filter installed, see <i>filter_type</i>
2	position open (white light)
3	position closed

The *filter_type* is an integer value to receive the type of filter installed. The value will always be zero, unless the *filter_status* is 1. The defined filter type values are:

<i>filter_type</i>	Description
0	no filter description
1	shortpass filter
2	longpass filter
3	bandpass filter
4	multiband filter

The *filter_wavelength* is the nominal wavelength value for the filter, expressed in nanometers (nm). The *filter_wavelength* is zero, unless the filter has a defined wavelength (shortpass filter, longpass filter, bandpass filter, or multiband filter).

The *filter_bandwidth* is the nominal bandwidth value for the filter, expressed in nanometers (nm). The *filter_bandwidth* value is zero, unless the filter has a defined bandwidth (bandpass filter).

TILLOligochrome_GetStatus

```
int TILLOligochrome_GetStatus(
    void* handle,
    double* lamp_time);
```

This operation provides information about the current device status.

The *handle* is the handle of an open Polychrome 3000. The return value is the operation status, zero if the operation succeeded.

The *lamp_time* is the total time the lamp has been on, measured in seconds. A useful way to present this value to a user is in units of hours, that is, units of (*lamp_time* / 3600).

Actions

The operations in this section are used to control the Polychrome 3000.

TILLOligochrome_DigitalInputRead

```
int TILLOligochrome_DigitalInputRead(void* handle, int bank, int mask, int* value);
```

This operation reads the value of digital input *bank* into *value*. The digital inputs are:

<i>bank</i>	Bit	Value	Description
1	0	0x01	Camera trigger input (panel)
	1	0x02	always 0

<i>bank</i>	Bit	Value	Description
	2	0x04	Connector Din 6
	3	0x08	Connector Din 5
	4	0x10	Connector Din 4
	5	0x20	Connector Din 3
	6	0x40	Connector Din 2
	7	0x80	Connector Din 1
2	0...7		
3	0...7		

The returned value is:

value = *input* & *mask*

That is, a bit is set in *value* if the bit is set in the digital input, and the bit is also set in *mask*. To read the entire digital input bank, use the *mask* value -1.

TILLOligochrome_DigitalOutputWrite

`int TILLOligochrome_DigitalOutputWrite(void* handle, int bank, int mask, int value);`

This operation writes the *value* to digital output *bank* subject to *mask*. This operation alters the value of the digital output as follows:

$\text{output} = (\text{value} \& \text{mask}) | (\text{output} \& \sim \text{mask})$

That is:

1. If a bit is set (1) in *mask*, the new output value of the bit is taken from *value*.
2. If a bit is clear (0) in *mask*, the output value of the bit does not change.

The digital outputs are:

<i>bank</i>	Bit	Value	Description
1	0	0x01	Camera trigger output (panel)
	1	0x02	Camera trigger output 2
	2...7	0xfc	not available (ignored)
2	0...7	0xff	
3	0...7	0xff	

TILLOligochrome_NoOperation

```
int TILLOligochrome_NoOperation(void* handle);
```

This operation sends a test message to the Polychrome 3000 and verifies the response. The operation verifies that the Polychrome 3000 is communicating with the application.

TILLOligochrome_SetFilterPosition

```
int TILLOligochrome_SetFilterPosition(void* handle, int filter_position, double intensity);
```

This operation sets the Polychrome 3000 to the specified *filter_position* and *intensity* value. The *filter_position* is in the range 1 to the value reported by TILLOligochrome_GetFeatures. The *intensity* value is in the range 0 to 1.

Note that an *intensity* value of 0 (zero) corresponds to no light output, this is equivalent to closing a shutter.

Note that the *intensity* value is relative to the full light output of the Polychrome 3000 for the specified *filter_position*. It is not a calibrated value: an *intensity* value of 1 (one) is full light output.

The operation returns immediately, regardless of the time needed to set the new filter position.

Variables

TILLOligochrome_VariableDeclareInteger

```
int TILLOligochrome_VariableDeclareInteger(void* handle, int* variable, int value);
```

This operation creates an integer *variable* with the initial contents *value*.

The operation returns the handle in *variable*. Only 32 integer variables can exist at the same time. Use TILLOligochrome_VariableFree to release a variable.

The *value* must be in the range -2^{23} to $2^{23}-1$, that is, the *value* is represented as a signed 24-bit integer.

TILLOligochrome_VariableDeclareString

```
int TILLOligochrome_VariableDeclareString(void* handle, int* variable, const char* value);
```

This operation creates an integer *variable* with the initial contents *value*.

The operation returns the handle in *variable*. Only 16 string variables can exist at the same time. Use TILLOligochrome_VariableFree to release a variable.

The *value* must be no longer than 128 characters, that is, 129 characters including the trailing null.

TILLOligochrome_VariableFree

```
int TILLOligochrome_VariableFreeInteger(void* handle, int variable);
```

This operation releases an integer *variable*. The *variable* cannot be used after this operation.

TILLOligochrome_VariableReadInteger

```
int TILLOligochrome_VariableReadInteger(void* handle, int variable, int* value);
```

This operation writes the integer value of *variable* into *value*. The variable must be an integer variable.

TILLOligochrome_VariableReadString

```
int TILLOligochrome_VariableReadString(void* handle, int variable, int value_size, char* value);
```

This operation writes the value of character string *variable* into *value*. The operation will not write more than *value_size* characters, including the trailing null. If the variable value is longer than *value_size*-1 characters, the operation returns an error.

TILLOligochrome_VariableWriteInteger

```
int TILLOligochrome_VariableWriteInteger(void* handle, int variable, int value);
```

This operation writes the integer *value* into *variable*, replacing the previous value of *variable*. The *variable* must be an integer variable.

TILLOligochrome_VariableWriteString

```
int TILLOligochrome_VariableWriteString(void* handle, int label, const char* value);
```

This operation writes the character string *value* into the variable *label*. The variable *label* must be a string variable. The character string *value* must not exceed 128 characters.

Arithmetic

TILLOligochrome_VariableAppendString

```
int TILLOligochrome_VariableAppendString(void* handle, int variable_source, int variable_integer, int result);
```



This operation converts the value of the *variable_integer* to a text string, and appends this text string to the value of the string variable *variable_source*. The resulting string is written to the string variable *result*.

TILLOligochrome_Variable... Arithmetic

```
int TILLOligochrome_VariableAdd(void* handle, int variable_1, int variable_2, int result);
```

```
int TILLOligochrome_VariableSubtract(void* handle, int variable_1, int variable_2, int result);
```

```
int TILLOligochrome_VariableMultiply(void* handle, int variable_1, int variable_2, int result);
```

```
int TILLOligochrome_VariableDivide(void* handle, int variable_1, int variable_2, int result);
```

```
int TILLOligochrome_VariableDivideScaled(void* handle, int variable_1, int variable_2, int result);
```

These operations perform the specified integer arithmetic operation on *variable_1* and *variable_2*, and write the result into *result*. All variables must be integer variables. The operations performed are:

Operation	Action
TILLOligochrome_VariableAdd	$result = variable_1 + variable_2$
TILLOligochrome_VariableSubtract	$result = variable_1 - variable_2$
TILLOligochrome_VariableMultiply	$result = variable_1 * variable_2$
TILLOligochrome_VariableDivide	$result = variable_1 / variable_2$
TILLOligochrome_VariableDivideScaled	$result = (2^{23} * variable_1) / variable_2$

TILLOligochrome_VariableDivideScaled is useful if $abs(variable_1)$ is less than $abs(variable_2)$.

TILLOligochrome_VariableIncrement

```
int TILLOligochrome_VariableIncrement(void* handle, int variable);
```

This operation increments the value of integer *variable*. That is:

```
variable = variable + 1
```

TILLOligochrome_VariableDecrement

```
int TILLOligochrome_VariableDecrement(void* handle, int variable);
```

This operation decrements the value of integer *variable*. That is:

```
variable = variable - 1
```

TILLOligochrome_Variable... Comparisons

```
int TILLOligochrome_VariableEqual(void* handle, int variable_1, int variable_2, int result);

int TILLOligochrome_VariableNotEqual(void* handle, int variable_1, int variable_2, int result);

int TILLOligochrome_VariableGreater(void* handle, int variable_1, int variable_2, int result);

int TILLOligochrome_VariableGreaterOrEqual(void* handle, int variable_1, int variable_2, int result);

int TILLOligochrome_VariableLess(void* handle, int variable_1, int variable_2, int result);

int TILLOligochrome_VariableLessOrEqual(void* handle, int variable_1, int variable_2, int result);
```

These operations compare the value of integer *variable_1* to the value of integer *variable_2*, and write the result to integer variable *result*. That is:

```
if variable_1 comparison variable_2
    result = 1
else
    result = 0
```

The operations are:

Operation	Comparison
TILLOligochrome_VariableEqual	<i>variable_1</i> == <i>variable_2</i>
TILLOligochrome_VariableNotEqual	<i>variable_1</i> != <i>variable_2</i>
TILLOligochrome_VariableGreater	<i>variable_1</i> > <i>variable_2</i>
TILLOligochrome_VariableGreaterOrEqual	<i>variable_1</i> >= <i>variable_2</i>
TILLOligochrome_VariableLess	<i>variable_1</i> < <i>variable_2</i>
TILLOligochrome_VariableLessOrEqual	<i>variable_1</i> <= <i>variable_2</i>

Events

Events are used to control timing within a protocol.

TILLOligochrome_EventDefineDigitalInput

```
int TILLOligochrome_EventDefineDigitalInput(void* handle, int* event, int bank, int mask, int value);
```

This operation defines a digital input event, and writes the event value into *event*. This event value can be used for operations that require an event, until TILLOligochrome_EventFree releases the event value.

The event is defined as true when the value of the specified digital input *bank* has the specified *value*, under the specified *mask*. That is, the event is true when:

```
(digital_input[bank] & mask) == value
```

TILLOligochrome_EventDefineJumper

```
int TILLOligochrome_EventDefineJumper(void* handle, int* event, int jumper);
```

This operation defines a jumper event, and writes the event value into *event*. This event value can be used for operations that require an event, until TILLOligochrome_EventFree releases the event value.

The event is true if the specified hardware *jumper* is present in the Polychrome 3000. The *jumper* is in the range 1...5.

TILLOligochrome_EventDefineSoftware

```
int TILLOligochrome_EventDefineSoftware(void* handle, int* event);
```

This operation defines a software event, and writes the event value into *event*. This event value can be used for operations that require an event, until TILLOligochrome_EventFree releases the event value.

The event is true after the application issues a TILLOligochrome_EventSoftware operation. The event remains true until reset by use of an event defined by a TILLOligochrome_EventDefineWait.

TILLOligochrome_EventDefineWait

```
int TILLOligochrome_EventDefineWait(void* handle, int* event);
```

This operation defines a wait event, and writes the event value into *event*. This event value can be used for operations that require an event, until TILLOligochrome_EventFree releases the event value.

When a wait event is used in a protocol, it resets the event used in the previous TILLOligochrome_ProtocolDelayEvent operation. The operation resets only TILLOligochrome_EventDefineSoftware events.

TILLOligochrome_EventDefine... Comparisons

```
int TILLOligochrome_EventDefineEqual(void* handle, int* event, int variable_1, int variable_2, int result);
```

```
int TILLOligochrome_EventDefineNotEqual(void* handle, int* event, int variable_1, int variable_2, int result);
```

```
int TILLOligochrome_EventDefineGreater(void* handle, int* event, int variable_1, int variable_2, int result);
```

```
int TILLOligochrome_EventDefineGreaterOrEqual(void* handle, int* event, int variable_1, int variable_2, int result);
```

```
int TILLOligochrome_EventDefineLess(void* handle, int* event, int variable_1, int variable_2, int result);
```

```
int TILLOligochrome_EventDefineLessOrEqual(void* handle, int* event, int variable_1, int variable_2, int result);
```

These operations define a comparison event. Comparison events are evaluated when referenced, not when defined. A comparison event compare the value of integer *variable_1* to the value of integer *variable_2*, and writes the result to integer variable *result*. That is:

```
if variable_1 comparison variable_2
    result = 1
else
    result = 0
```

The operations are:

Operation	Comparison
TILLOligochrome_EventDefineEqual	<i>variable_1</i> == <i>variable_2</i>
TILLOligochrome_EventDefineNotEqual	<i>variable_1</i> != <i>variable_2</i>
TILLOligochrome_EventDefineGreater	<i>variable_1</i> > <i>variable_2</i>
TILLOligochrome_EventDefineGreaterOrEqual	<i>variable_1</i> >= <i>variable_2</i>
TILLOligochrome_EventDefineLess	<i>variable_1</i> < <i>variable_2</i>
TILLOligochrome_EventDefineLessOrEqual	<i>variable_1</i> <= <i>variable_2</i>

TILLOligochrome_EventFree

```
int TILLOligochrome_EventFree(void* handle, int event);
```

This operation frees the specified *event* value so it can be redefined.

TILLOligochrome_EventSoftware

```
int TILLOligochrome_EventSoftware(void* handle, int event);
```

This operation generates a software *event*. The *event* must have been previously defined by a TILLOligochrome_EventDefineSoftware operation.

Protocol Control

A protocol is a sequence of operations that can be executed repeatedly. The operations in this section control the execution of a protocol.

An application starts recording a protocol in the Polychrome 3000 using TILLOligochrome_ProtocolBegin.



An application ends recording a protocol in the Polychrome 3000 using `TILLOligochrome_ProtocolEnd`. The protocol begins execution. While recording a protocol, do not issue any operations to the Polychrome 3000 other than `TILLOligochrome_Protocol...` operations.

TILLOligochrome_ProtocolBegin

```
int TILLOligochrome_ProtocolBegin(void* handle);
```

This operation begins recording a protocol in the Polychrome 3000.

TILLOligochrome_ProtocolEnd

```
int TILLOligochrome_ProtocolEnd(void* handle);
```

This operation ends recording of a protocol in the Polychrome 3000, and begins execution of the protocol.

TILLOligochrome_ProtocolCancel

```
int TILLOligochrome_ProtocolCancel(void* handle);
```

This operation cancels recording of a protocol in the Polychrome 3000. The protocol is discarded.

TILLOligochrome_ProtocolDelayEvent

```
int TILLOligochrome_ProtocolDelayEvent(void* handle, int event);
```

This operation delays the beginning of the next `TILLOligochrome_Protocol...` operation until the specified *event* occurs.

TILLOligochrome_ProtocolDelayTime

```
int TILLOligochrome_ProtocolDelayTime(void* handle, double interval);
```

This operation delays the beginning of the next `TILLOligochrome_Protocol...` operation until *interval* seconds following the beginning of the previous command. The *interval* must be positive. The *interval* is performed in the protocol with a resolution specified by `TILLOligochrome_GetIntervalResolution`.

TILLOligochrome_ProtocolDelayTrigger

```
int TILLOligochrome_ProtocolDelayTrigger(void* handle, int trigger_as_gate);
```

This operation delays the beginning of the next `TILLOligochrome_Protocol...` operation until a trigger is recognized. If *trigger_as_gate* is 0 (zero), the operation waits for a trigger transition. If *trigger_as_gate* is TRUE,

the operation waits for a gate.

TILLOligochrome_ProtocolLoopBegin

```
int TILLOligochrome_ProtocolLoopBegin(void* handle, int count);
```

This operation can only be issued during recording of a protocol in a Polychrome 3000. All operations added to the protocol following TILLOligochrome_ProtocolLoopBegin and before the corresponding TILLOligochrome_ProtocolLoopEnd will be repeated *count* times. The repetition count must be at least one. Protocol loops can be nested.

TILLOligochrome_ProtocolLoopEnd

```
int TILLOligochrome_ProtocolLoopEnd(void* handle);
```

This operation ends the loop started by the most recent TILLOligochrome_ProtocolLoopBegin.

TILLOligochrome_ProtocolLoopDisable

```
int TILLOligochrome_ProtocolLoopDisable(void* handle);
```

This operation does not immediately exit the current loop. Instead, the operation causes TILLOligochrome_ProtocolLoopEnd operations to avoid repeating the loop. Execute a TILLOligochrome_ProtocolLoopEnable to enable loops.

TILLOligochrome_ProtocolLoopEnd

```
int TILLOligochrome_ProtocolLoopEnd(void* handle);
```

This operation can only be issued during recording of a protocol in a Polychrome 3000. The operation ends a loop in a protocol in the Polychrome 3000. Every TILLOligochrome_ProtocolLoopBegin must have a corresponding TILLOligochrome_ProtocolLoopEnd operation.

TILLOligochrome_ProtocolLoopEnable

```
int TILLOligochrome_ProtocolLoopEnable(void* handle);
```

This operation enables TILLOligochrome_ProtocolLoopEnd operations after a TILLOligochrome_ProtocolLoopDisable.

TILLOligochrome_ProtocolLoopGetIndex

```
int TILLOligochrome_ProtocolLoopGetIndex(void* handle, int level);
```



This operation issues the current loop index of the loop at the specified *level*. The loop index can be detected by TILLOligochrome_GetStatus.

TILLOligochrome_ProtocolIf

```
int TILLOligochrome_ProtocolIf(void* handle, int event);
```

This operation creates an 'if' branch in a protocol, based on the *event*. All protocol operations between the TILLOligochrome_ProtocolIf and the following TILLOligochrome_ProtocolElse or TILLOligochrome_ProtocolEndIf are performed only if the *event* is true.

TILLOligochrome_ProtocolElse

```
int TILLOligochrome_ProtocolElse(void* handle);
```

This operation creates an 'else' branch in a protocol, based on a preceding TILLOligochrome_ProtocolIf. The 'else' branch executes only if the *event* specified in the preceding TILLOligochrome_ProtocolIf is false.

TILLOligochrome_ProtocolEndIf

```
int TILLOligochrome_ProtocolElse(void* handle);
```

This operation ends an 'if' or 'else' branch in a protocol. Execution of the protocol resumes following this operation, regardless of the *event* specified in the preceding TILLOligochrome_ProtocolIf.

Protocol Actions

TILLOligochrome_ProtocolDigitalInputRead

```
int TILLOligochrome_ProtocolDigitalInputRead(void* handle, int bank, int mask);
```

This operation reads the value of digital input *bank* subject to the *mask* value. The resulting digital input value can be read by TILLOligochrome_ResponseDigitalInputRead.

The digital inputs are described in TILLOligochrome_DigitalInputRead.

TILLOligochrome_ProtocolDigitalOutputWrite

```
int TILLOligochrome_ProtocolDigitalOutputWrite(void* handle, int bank, int mask, int value);
```

This operation writes the *value* to digital output *bank* subject to *mask*.

The digital outputs are described in `TILLOligochrome_DigitalOutputWrite`.

TILLOligochrome_ProtocolExperimentTimerRead

```
int TILLOligochrome_ProtocolExperimentTimerRead(void* handle);
```

This operation writes the current experiment timer value. The value can be read using `TILLOligochrome_ResponseExperimentTimerRead`.

TILLOligochrome_ProtocolExperimentTimerReset

```
int TILLOligochrome_ProtocolExperimentTimerReset(void* handle);
```

This operation resets the experiment timer. The value prior to the reset can be read using `TILLOligochrome_ResponseExperimentTimerReset`.

TILLOligochrome_ProtocolMark

```
int TILLOligochrome_ProtocolMark(void* handle, int mark);
```

This operation issues the specified *mark* value. The *mark* value can be detected by `TILLOligochrome_GetStatus`.

The *mark* value must be a positive integer value in the range 0 (zero) to $2^{31}-1$, that is, a positive 32-bit value.

TILLOligochrome_ProtocolNoOperation

```
int TILLOligochrome_ProtocolNoOperation(void* handle);
```

This operation performs no action. Use this operation in conjunction with the delay operations to control timing:

```
TILLOligochrome_ProtocolDelayTime(handle, 0.100);  
TILLOligochrome_ProtocolNoOperation(handle);
```

TILLOligochrome_ProtocolSetFilterPosition

```
int TILLOligochrome_ProtocolSetFilterPosition(void* handle, int filter_position, double intensity);
```

This operation sets the Polychrome 3000 to the specified *filter_position* and *intensity* value. The *filter_position* is in the range 1 to the value reported by `TILLOligochrome_GetFeatures`. The *intensity* value is in the range 0 to 1.

Protocol Variable

TILLOligochrome_ProtocolVariableReadInteger

```
int TILLOligochrome_ProtocolVariableReadInteger(void* handle, int variable);
```

This operation reads the value of integer *variable*. The value can be read using TILLOligochrome_ResponseVariableReadInteger.

TILLOligochrome_ProtocolVariableReadString

```
int TILLOligochrome_ProtocolVariableReadString(void* handle, int variable);
```

This operation reads the value of character string *variable*. The value can be read using TILLOligochrome_ResponseVariableReadString.

TILLOligochrome_ProtocolVariableWriteInteger

```
int TILLOligochrome_ProtocolVariableWriteInteger(void* handle, int variable, int value);
```

This operation writes the integer *value* into *variable*, replacing the previous value of the *variable*.

TILLOligochrome_ProtocolVariableWriteString

```
int TILLOligochrome_ProtocolVariableWriteString(void* handle, int variable, const char* value);
```

This operation writes the character string *value* into *variable*.

Protocol Arithmetic

TILLOligochrome_ProtocolVariableAppendString

```
int TILLOligochrome_ProtocolVariableAppendString(void* handle, int label_source, int label_integer, int label_result);
```

This operation converts the value of the integer variable *label_integer* to a text string, and appends this text string to the value of the string variable *label_source*. The resulting string is written to the string variable *label_result*.

TILLOligochrome_ProtocolVariable... Arithmetic

```
int TILLOligochrome_ProtocolVariableAdd(void* handle, int variable_1, int variable_2, int result);
```

```
int TILLOligochrome_ProtocolVariableSubtract(void* handle, int variable_1, int variable_2, int result);
```

```
int TILLOligochrome_ProtocolVariableMultiply(void* handle, int variable_1, int variable_2, int result);
```

```
int TILLOligochrome_ProtocolVariableDivide(void* handle, int variable_1, int variable_2, int result);
```

```
int TILLOligochrome_ProtocolVariableDivideScaled(void* handle, int variable_1, int variable_2, int result);
```

These operations perform the specified arithmetic operation on *variable_1* and *variable_2*, and write the result into *result*. All variables must be integer variables. The operations performed are:

Operation	Action
TILLOligochrome_ProtocolVariableAdd	$result = variable_1 + variable_2$
TILLOligochrome_ProtocolVariableSubtract	$result = variable_1 - variable_2$
TILLOligochrome_ProtocolVariableMultiply	$result = variable_1 * variable_2$
TILLOligochrome_ProtocolVariableDivide	$result = variable_1 / variable_2$
TILLOligochrome_ProtocolVariableDivideScaled	$result = (2^{23} * variable_1) / variable_2$

TILLOligochrome_ProtocolVariableIncrement

```
int TILLOligochrome_ProtocolVariableIncrement(void* handle, int variable);
```

This operation increments the value of integer variable *variable*.

TILLOligochrome_ProtocolVariableDecrement

```
int TILLOligochrome_ProtocolVariableDecrement(void* handle, int variable);
```

This operation decrements the value of integer variable *variable*.

TILLOligochrome_Protocol... Comparisons

```
int TILLOligochrome_ProtocolEqual(void* handle, int variable_1, int variable_2, int result);
```

```
int TILLOligochrome_ProtocolNotEqual(void* handle, int variable_1, int variable_2, int result);
```

```
int TILLOligochrome_ProtocolGreater(void* handle, int variable_1, int variable_2, int result);
```

```
int TILLOligochrome_ProtocolGreaterOrEqual(void* handle, int variable_1, int variable_2, int result);
```

```
int TILLOligochrome_ProtocolLess(void* handle, int variable_1, int variable_2, int result);
```

```
int TILLOligochrome_ProtocolLessOrEqual(void* handle, int variable_1, int variable_2, int result);
```

These operations compare the value of integer *variable_1* to the value of integer *variable_2*, and write the result to integer variable *result*. That is:

```
if variable_1 comparison variable_2
    result = 1
else
    result = 0
```

The operations are:

Operation	Comparison
TILLOligochrome_ProtocolEqual	<i>variable_1</i> == <i>variable_2</i>
TILLOligochrome_ProtocolNotEqual	<i>variable_1</i> != <i>variable_2</i>
TILLOligochrome_ProtocolGreater	<i>variable_1</i> > <i>variable_2</i>
TILLOligochrome_ProtocolGreaterOrEqual	<i>variable_1</i> >= <i>variable_2</i>
TILLOligochrome_ProtocolLess	<i>variable_1</i> < <i>variable_2</i>
TILLOligochrome_ProtocolLessOrEqual	<i>variable_1</i> <= <i>variable_2</i>

Protocol Response

TILLOligochrome_GetResponse

```
int TILLOligochrome_GetResponse(void* handle, int* response);
```

This operation returns an Polychrome 3000 response. A protocol can send responses to the host computer. This operation reads those responses, in order. The final response of a protocol is the response for TILLOligochrome_ProtocolEnd, ending the protocol.

For each response, the application should call the appropriate TILLOligochrome_Reponse... operation.

The *handle* is the handle of an open Polychrome 3000.

The operation writes a value into *response*. If the Polychrome 3000 has issued a response as a result of a protocol command, the value of *response* is non-zero. The possible values of *response* are:

Name	Reader
..._RESPONSE_DIGITAL_INPUT_READ	..._ResponseDigitalInputRead
..._RESPONSE_EXPERIMENT_TIMER_READ	..._ResponseExperimentTimerRead

Name	Reader
..._RESPONSE_EXPERIMENT_TIMER_RESET	..._ResponseExperimentTimerReset
..._RESPONSE_GET_LOOP_INDEX	..._ResponseGetLoopIndex
..._RESPONSE_MARK	..._ResponseMark
..._RESPONSE_NONE	
..._RESPONSE_PROTOCOL_END	
..._RESPONSE_VARIABLE_READ_INTEGER	..._ResponseVariableReadInteger
..._RESPONSE_VARIABLE_READ_STRING	..._ResponseVariableReadString

TILL_OLIGOCHROME_RESPONSE_NONE and TILL_OLIGOCHROME_RESPONSE_PROTOCOL_END do not have associated TILLOligochrome_Response... functions, because they have no parameters.

For example, typical C/C++ code might be:

```
int response;
int status = TILLOligochrome_GetResponse(handle, &response);
int mark;

switch (response)
{
case TILL_OLIGOCHROME_RESPONSE_NONE:
    break;
case TILL_OLIGOCHROME_RESPONSE_MARK:
    TILLOligochrome_ResponseMark(handle, &mark);
    ...
    break;
case TILL_OLIGOCHROME_RESPONSE_PROTOCOL_END:
    ...
    break;
default:
    break;
}
```

TILLOligochrome_ResponseDigitalInputRead

```
int TILLOligochrome_ResponseDigitalInputRead(void* handle, int* bank, int* value);
```

This operation writes the digital input value recorded by TILLOligochrome_ProtocolDigitalInputRead into *value*. The digital input bank used is written into *bank*.

An application should call TILLOligochrome_ResponseDigitalInputRead after TILLOligochrome_GetResponse returns TILL_OLIGOCHROME_RESPONSE_DIGITAL_INPUT_READ.



TILLOligochrome_ResponseExperimentTimerRead

```
int TILLOligochrome_ResponseExperimentTimerRead(void* handle, double *timer);
```

This operation writes the experimental timer value recorded by TILLOligochrome_ProtocolExperimentTimerRead into *timer*. The value is expressed in seconds, with a resolution as specified by TILLOligochrome_GetIntervalResolution.

TILLOligochrome_ResponseExperimentTimerReset

```
int TILLOligochrome_ResponseExperimentTimerReset(void* handle, double *reset_timer);
```

This operation writes the experimental timer value recorded by TILLOligochrome_ProtocolExperimentTimerReset into *reset_timer*, the value of the experimental timer just prior to reset. The value is expressed in seconds, with a resolution as specified by TILLOligochrome_GetIntervalResolution.

TILLOligochrome_ResponseGetLoopIndex

```
int TILLOligochrome_ResponseGetLoopIndex(void* handle, int* level, int* index);
```

This operation writes the loop index written by TILLOligochrome_ProtocolGetLoopIndex into *index*. The operation writes the loop level into *level*.

TILLOligochrome_ResponseMark

```
int TILLOligochrome_ResponseMark(void* handle, int* mark);
```

This operation writes the mark value written by TILLOligochrome_ProtocolMark into *mark*.

TILLOligochrome_ResponseVariableReadInteger

```
int TILLOligochrome_ResponseVariableReadInteger(void* handle, int* variable, int* value);
```

This operation writes the integer variable value written by TILLOligochrome_ProtocolVariableRead into *value*. The variable into *variable*.

TILLOligochrome_ResponseVariableReadString

```
int TILLOligochrome_ResponseVariableReadString(void* handle, int* variable, int value_size, char* value);
```

This operation writes the character string variable value written by TILLOligochrome_ProtocolVariableRead into *value*. The variable is written into *variable*.

The operation will not write more than *value_size* characters, including the trailing null. If the variable



value is longer than *value_size*-1 characters, the operation returns an error.