DS8R PROGRAMMING REFERENCE

Digitimer Limited
www.digitimer.com

Contents

Introduction

The DS8R PC Software provides a simple Graphical User Interface (GUI) with which the user can view and set parameters on their DS8R. Additionally, a Dynamic Link Library (DLL) is provided, exposing a simple Application Programming Interface (API), which the user may employ in bespoke applications (clients). This allows users to retrieve and set parameters on the DS8R programmatically. It is with this same API that the DS8R GUI relies on in order to communicate with attached devices.

The API allows control of all the parameters accessible on the front panel, as well as the ability to control some features only available programmatically.

Multiple Clients

Multiple applications can simultaneously connect to the DS8R(s) through the API. The underlying Device Management Service will serialise all requests by the client applications. The API is completely passive requiring regular polling by the client application in order to ensure consistency between the API maintained state and the actual state of the DS8R. As multiple clients are all able to make changes to the devices as well as changes made by the front panel, the programmer must determine the frequency at which their application will poll for changes.

Where multiple clients are connected and polling quickly, the API will restrict the number of times it physically communicates with each device to once every 100ms (10Hz). Any polls made by client APIs during this time will simply return the status from the last time the device was queried.

Non-Block & Blocking API Calls

All of the physical communications with the connected devices are performed by the underlying Device Management Service (DGHost.exe). This runs as a separate process performing the serialization and caching of the device statuses. Because of this, the calling of an API function and the physical execution of the command on the DS8R is somewhat asynchronous. For this reason all the API functions are implemented as either blocking or non-blocking calls. To implement non-blocking calls, each API can be supplied a pointer to a callback completion routine. When supplied, the API function will return immediately, however, the function can only be guaranteed to have completed once the completion routine has been called.

Important. The completion callback routines are called from a separate API thread. It is the responsibility of the Programmer to ensure variables and resources accessed are thread safe.

API Latency

Although it is possible to programmatically trigger a DS8R, it should not be relied upon for accurate pulse interval times. The latency between making the API call and the device actually performing the requested function will not be consistent. The rear panel Trigger Input should be used when precise stimulus timings are required.

64bit/32bit API

When the DS8R Software is installed on a MS Windows 64bit OS, it will also install the 32bit API. This allows 32bit compiled applications running on a 64bit OS to access the API. It is also possible for mixed 32 and 64 bit clients to simultaneously access the API and control/poll the devices.

DGD128_Initialise function

Initializes and connects to the DS8R Device Management Services. Returns a reference to a handle with which subsequent calls to **DGD128_Update** and **DGD128_Close** must be passed. The function can be called as either blocking or non-blocking.

By passing a pointer to a call-back routine, it is possible for the user's application to continue processing whilst the Device Management Service is processing the request as a separate process. Once the initialization process is complete, the call-back routine will be called.

Syntax

```
INT __stdcall DGD128_Initialise(
      INT * lpReference,
      INT * lpInitResult,
      DGClientInialiseProc * lpCallbackproc,
      void CallbackParam
      );
```

Parameters

lpReference

A pointer to a variable to receive a reference for the current session. If a call-back completion routine has been provided, the returned value of this will be unknown.

If a call-back completion routine has been provided, the value returned by **DGD128_Initialise** will be invalid. Do not attempt to use this value to make calls to **DGD128_Update** or **DGD128_Close**.

Multiple calls to **DGD128_Initialise** can be made and each call will return a separate session reference. It is important to ensure a **DGD128_Close** for each opened session is made in order to correctly close and release the resources allocated by the Device Management Service.

lpInitResult

A pointer to a variable to receive the error code returned by the Device Management Service.

lpInitResult, defines the error as returned by the remote Device Management Service. This is distinctly different from the function return result, which defines the nature of the error raised by the API.

If a call-back completion routine has been provided, the value returned by **DGD128_Initialise** will be invalid. A valid value will be returned with the call to the completion routine supplied.

For information on the possible return codes, see **DGD128 API Error Codes**.

lpCallbackproc

A pointer to a function of type DGClientInitialiseProc. This parameter can be NULL.

By supplying a NULL for this parameter, no completion routine will be called. Instead, the call to DGD128_Initilise will block until the function is complete.

Note When either the completion routine or the function is called with no completion routine, it is not possible to determine how long the underlying service will take to process the command and call the completion routine or return.

Note The call to the completion routine is made from a separate thread within the callers own process. It is important to ensure that proper consideration is given to access of variables and resources within a call to the completion routine and that these are thread safe.

CallbackParam

Application defined callback data.

DGD128_Update function

The DGD128_Update permits writing or receiving device parameters for either a single device or all connected DS8R's.

This function can be called as either blocking or non-blocking. If a call-back completion routine is supplied, the routine will return immediately. Once the background Device Management Service has processed the update, the supplied completion routine will be called.

> **Note** The calling of this function and the subsequent change to the device or devices is asynchronous. Do not rely on the API when performing time sensitive changes.
>
> If it is necessary to ensure that the device(s) have been changed, always use the API calls as blocking.

Syntax

```
INT __stdcall DG128_Update(
      int Reference,
      int * UpdateResult,
      PD128 * NewState,
      int cbNewState,
      PD128 * CurrentState,
      int cbCurrentState,
      DGClientUpdateProc * Callbackproc,
      void * CallbackParam
      );
```

Parameters

Reference

Session reference returned by **DGD128_Initialise**.

UpdateResult

Receives an error code returned by the remote Device Management Service. For information refer to **DGD128 API Error Codes**.

This result is invalid if a CallbackProc has been supplied.

NewState

Pointer to a **D128** structure. The size of the structure is dependent on the number of devices being described.

If supplied, this structure can be used to update a single device or multiple devices, if connected.

If this parameter is NULL, no change to the device(s) will be made.

cbNewState

Must be a valid size of the structure pointed to by **NewState**. If **NewState** is NULL this this value must be 0 (zero).

CurrentState

Pointer to a **D128** structure to receive the current state of ALL the connected DS8R's.

Regardless of the number of devices described by **NewState**, this structure will always return the states for ALL of the connected devices.

It is the callers responsibility to ensure that sufficient memory has been reserved to receive the full structure. If the amount of memory is too small, the function will return **ERROR_BAD_ARGUMENTS**.

cbCurrentState

Is used to pass the size, in BYTES, of the memory referenced by CurrentState. On return, this will be modified with the actual size of the structure returned.

To determine the size of the memory to reserve, call DGD128_Update with

```
DGD128_Update ( Ref, *UpdateError, NULL, NULL, NULL, *SizeOfD128, NULL, NULL);
```

This allows the caller to then allocate sufficient memory and perform the call again, this time passing a reference to the reserved memory.

Note The Device Management Service caches the device states. This makes repeated calls to DGD128_Update very efficient.

CallbackProc

A pointer to a function of type DGClientUpdateProc. This parameter can be NULL.

By supplying a NULL for this parameter, no completion routine will be called. Instead, the call to DGD128_Initilise will block until the function is complete.

Note When either the completion routine or the function is called with no completion routine, it is not possible to determine how long the underlying service will take to process the command and call the completion routine or return.

Note The call to the completion routine is made from a separate thread within the callers own process. It is important to ensure the proper consideration is given to access of variables and resources within a call to the completion routine and these are thread safe.

CallbackParam

Application defined callback data.

Remarks

The client application must be aware that the number of devices can change as they are physically switched on or off.

The order in which the devices appear in the **D128** structure is in ascending serial number order.

DGD128_Close function

Closes and disconnects from the underlying device management services. Frees resources and invalidates the reference. Continuing to use the reference will result in returned errors.

Syntax

```
INT __stdcall DG128_Close(
      int * Reference,
      int * CloseResult,
      DGClientCloseProc * Callbackproc,
      void * CallbackParam
      );
```

Remarks

Once called, if there are no remaining clients connected to the underlying device services, they will automatically close, freeing all communications resources used.

D128 structure

Syntax

Used to describe the state of 0 (zero) or more DS8R devices.

```
typedef struct _D128 {
      DEVHDR Header;
      D128STATE State[];
      } D128, *PD128;
```

Members

Header

For information see **DEVHDR.**

State

Array of **D128DEVICESTATE** structures. The number of elements depends on the number of DS8R devices connected.

Remarks

Only pointers to this structure are exchanged between the client application and the API. The actual size of memory required is dependent on the number of devices defined in State. It is important to ensure that when supplying the size of the structure, the following equation is used :

Size = Number of Devices * sizeof(**D128DEVICESTATE**) + sizeof(**DEVHDR**)

To determine the actual number of bytes to reserve for the structure, make a call to **DGD128_Update** with the following parameters.

DGD128_Update (Ref, &UpdateError, NULL, NULL, NULL, &SizeOfD128, NULL, NULL);

This will return in **SizeOfD128** the number of bytes to allocate for receiving the State for all devices.

DEVHDR structure

Used to maintain the number of DS8R devices described in the **D128** structure.

Syntax

```
typedef struct _DEVHDR {
      INT DeviceCount;
      } DEVHDR;
```

Members

DeviceCount

An integer which maintains the number of devices described by the containing structure.

D128DEVICESTATE structure

Describes the identified, error, firmware and state of a specific connected device.

Syntax

```
typedev _D128STATE {
      INT D128_DeviceID;
      INT D128_VersionID;
      INT D128_Error;
      D128STATE D128_State;
      };
```

Members

D128_DeviceID

Serial number for the device.

D128_VersionID

Currently installed firmware version.

D128_Error

Error code returned by the device. For information on the possible error codes see **D128 API Error Codes**.

D128_State

Structure of type **D128STATE.**

D128STATE structure

Contains information on the current state of the device. By modifying the members calling **DGD128_Update** it is possible to update the settings of a specific device.

All parameters are validated by the device before being applied.

To obtain the current value on the device, set the parameters to -1 (minus one). **DGD128_Update** will return the current value in the CurrentState structure.

Syntax

```
typedef _D128STATE {
      CONTROLFLAGS Control;
      int Demand;
      int Width;
      int Recovery;
      int Dwell;
      unsigned int CPULSE;
      unsigned int COOC;
      STATEFLAGS SFlags;
      } D128STATE;
```

Members

Control

See **CONTROLFLAGS.**

Demand

Controls the demand of the stimulus when internal selected.

The value is (mA * 10) i.e. for a demand of 500.0mA set this member to 5000.

Width

Controls the stimulus pulse duration in µs (micro-seconds)

Recovery

Controls the recovery pulse duration when the BI-PHASIC mode is selected. The value represents the percentage Amplitude the recovery pulse will have. The recovery pulse duration is automatically adjusted to ensure the pulse energy is the same as the stimulus pulse.

Valid values

10 – 100

Dwell

Controls the period between the end of the stimulus pulse and the start of the recovery pulse when BI-PHASIC mode is enabled. The value is in µs.

Valid values

1 – 990

CPULSE

Number of stimulus pulses delivered since the device was ENABLED. This value will only be reset when the output is DISABLED and the ENABLED.

COOC

Number of out of compliance events since the device was ENABLED. This value will only be reset when the output is DISABLED and the ENABLED.

CTOOFAST

Number of trigger too fast events since the device was ENABLED. This value will only be reset when the output is DISABLED and the ENABLED.

SFlags

See **STATEFLAGS.**

CONTROLFLAGS structure

The control flags contain status as well as initiate commands. Some of the members will always be returned as zero, however, if set they will perform a specific action.

Syntax

```
typedef struct _CONTROLFLAGS {
      union {
        struct {
          int Enable : 2;
          int Mode : 3;
          int Polarity : 3;
          int Source : 3;
          int Zero : 2;
          int Trigger : 2;
          int NoBuzzer : 2;
          int Reserved : 15;
        }
        int VALUE;
        }
      };
```

Members

Enable

Control of output enable state.

Valid values

0x01 DISABLED
0x02 ENABLED
0x03 No change

Note It is not possible to clear the Over Energy error using the API. This must be cleared by the user operating the front panel control.

Mode

Controls which pulse mode is implemented.

Valid values

0x01 MONO-PHASIC
0x02 BI-PHASIC
0x07 No change

Polarity

Controls the pulse polarity.

Valid values

0x01	Positive
0x02	Negative
0x03	Alternating
0x07	No change

Source

Controls the source for the stimulus demand.

Valid values

0x01	Internal (front panel)
0x02	External (rear panel BNC analog input)
0x07	No change

Zero

Causes the device auto zero function to be performed on the rear panel BNC input.

Valid values

0x01	Start auto zero
0x03	No change

Trigger

Will initiate a trigger as long as the output is enabled.

Valid values

0x01	Initiate trigger
0x03	No change

NoBuzzer

Controls the audible indication on the device for OOC events.

Valid values

0x00	Buzzer enabled.
0x01	Buzzer DISABLED

DGD128 API Error Codes

The following lists all the DS8R specific error codes that can be returned either by an API function or as the returned error.

ERROR_NOT_INITIALISED

100002 (0x186A2)

Attempt to call either **DGD128_Update** or **DGD128_Close** without a prior call to **DGD128_Initialise**

ERROR_PROCESS_TERMINATED

100003 (0x186A3)

Attempted to communicate with DGHost.exe but the process had terminated.

ERROR_UNEXPECTED_TERMINATION

100004 (0x186A4)

Either D128API or DGHost attempted to communicate with a thread that had terminated unexpectedly.

ERROR_INITIALISE_TIMEOUT

100006 (0x186A6)

Timed out wait for call to **DGD128_Initialise** to complete.

ERROR_INVALID_CMD_PACKET

100008 (0x186A8)

Returned by the Digitimer Device, the command packet received was invalid.

ERROR_INVALID_REPLY_PACKET

100009 (0x186A8)

The returned packet, in response to a command, from the device is invalid.

ERROR_PACKET_RECEIVE_TIMEOUT

100011 (0x186AB)

Timed out waiting for packet to be received from device.

ERROR_INVALID_REFERENCE

100012 (0x186AC)

Returned packet reference invalid.

ERROR_INVALID_PACKET

100013 (0x186AD)

The packet return by the device was invalid (too small or too large).

ERROR_TERMINATE_TIMEOUT

100014 (0x186AE)

When attempting to terminate a thread, timed out waiting for acknowledge.

ERROR_RECEIVE_QUEUE_EMPTY

100015 (0x186AF)

Expected receive packet, but receive queue was empty.

ERROR_INITIALISED

100016 (0x186B0)

DGHost already initialised.

ERROR_INITIALISE_FAILED

100017 (0x186B1)

DGHost initialise attempt failed.

ERROR_DEVICE_NOT_FOUND

100018 (0x186B2)

Device no longer connected.

ERROR_INVALID_PARAMETER

100019 (0x186B3)

Device returned invalid parameter in command block.

ERROR_INVALID_STRUCTURE

100020 (0x186B4)

Update structure size does not meet the expected size.

ERROR_INVALID_POINTER

100021 (0x186B5)

Invalid pointer reference supplied.

ERROR_IN_PROGRESS

100022 (0x186B6)

ERROR_DGHOST_NOT_REGISTERED

100023 (0x186B7)

The DGHost application has not been registered.

ERROR_DGHOST_NOT_FOUND

100024 (0x186B8)

The DGHost.exe application file cannot be found.

ERROR_CLIENT_DLL_NOT_FOUND

100025 (0x186B9)

The DGHost.exe application cannot find the device API DLL.

ERROR_DGHOST_STARTUP_TIMEOUT

100026 (0x186BA)

Timed out waiting for the DGHost start-up semaphore.

ERROR_CLIENT_RESOURCE_TIMEOUT

100027 (0x186BB)

API Error

ERROR_CLIENT_THREAD_ABORT

100028 (0x186BC)

API Error

ERROR_PIPE_WRITE_TIMEOUT

100029 (0x186BD)

Timed out waiting to write packet to pipe stream.

ERROR_PIPE_READ_TIMEOUT

100030 (0x186BE)

Timed out waiting for read from pipe read stream.

ERROR_PIPE_READ_NULL

100031 (0x186BF)

NULL data read from pipe stream.

ERROR_INTERNAL_ERROR

100032 (0x186C0)

Internal error.

ERROR_INVALID_HOST_PACKET

100033 (0x186C1)

Invalid pip data packet received.

ERROR_INTERNAL_EXCEPTION

100034 (0x186C2)

Internal exception

ERROR_DEVICE_CMD_ERROR

100035 (0x186C3)

Device command error.

Example Application

The following is a simple C program to demonstrate the implementation of the D128API. A compressed folder containing a Microsoft Visual Studio 2015 project is installed on your computer when the DS8R Software is installed.

```
#include "stdafx.h"
#include "stdio.h"
#include "windows.h"
#include "winbase.h"
#include “D128API.h”

HMODULE libD128API;

DGD128_Initialise procInitialise;
DGD128_Update procUpdate;
DGD128_Close procClose;

int loadD128APILibrary() {

   libD128API = LoadLibraryA("D128API.DLL");
   if (libD128API) {
      if (!(procInitialise = (DGD128_Initialise) GetProcAddress(libD128API, "DGD128_Initialise"))) {
         printf("Failed! GetProcAddress(""DGD128_Initialise"")");
         return FALSE;
      };

      if (!(procUpdate = (DGD128_Update) GetProcAddress(libD128API, "DGD128_Update"))) {
         printf("Failed! GetProcAddress(""DGD128_Update"")");
         return FALSE;
      };

      if (!(procClose = (DGD128_Close) GetProcAddress(libD128API, "DGD128_Close"))) {
         printf("Failed! GetProcAddress(""DGD128_Close"")");
         return FALSE;
      };

      return TRUE;

   } else {
      printf("Failed to load D128API.DLL.");
      return FALSE;
   }
}

void printDeviceState(D128DEVICESTATE * State) {

   printf("Serial Number = %d\n\r", State->D128_DeviceID);
   printf("Firmware Version = %0.2x.%0.2x.%0.2x.%0.2x\n\r",
      (State->D128_VersionID>>24) & 0xFF,
      (State->D128_VersionID>>16) & 0xFF,
      (State->D128_VersionID>>8) & 0xFF,
      (State->D128_VersionID) & 0xFF
      );

   printf("State\n\r");
   printf("    Demand = %0.1f\n\r",(float)State->D128_State.DEMAND/10.0);
   printf("     Width = %d\n\r", State->D128_State.WIDTH);
   printf("  Recovery = %d\n\r", State->D128_State.RECOVERY);
   printf("     Dwell = %d\n\r", State->D128_State.DWELL);
   printf("\n\r");
}

int apiRef;
int retAPIError;
int retError;
int cbState;
int devIdx;
PD128 CurrentState;
D128DEVICESTATE devState;
```

```
int main()
{
   if (loadD128APILibrary()) {

      apiRef = 0;
      retError = procInitialise(&apiRef, &retAPIError, NULL, NULL);

      if ((retError == ERROR_SUCCESS) && (retAPIError == ERROR_SUCCESS)) {

         retError = procUpdate(apiRef, &retAPIError, NULL, 0, NULL, &cbState, NULL, NULL);

         if ((retError == ERROR_SUCCESS) && (retAPIError == ERROR_SUCCESS)) {

            CurrentState = (PD128) malloc(cbState);
            retError = procUpdate(apiRef, &retAPIError, NULL, 0, CurrentState, &cbState, NULL, NULL);

            if (CurrentState->Header.DeviceCount) {

               for (devIdx = 0; devIdx < CurrentState->Header.DeviceCount; devIdx++) {
                  printDeviceState(&CurrentState->State[devIdx]);

               }

            }
         } else {

            //Handle Error

         }
      } else {

         //Handle error

      }
   }

   printf("Press any key.\n\r");
   fgetc(stdin);

   if (apiRef) {
      retError = procClose(&apiRef, &retAPIError, NULL, NULL);
   }

   return 0;
}
```