CPS model BP2595 bipolar power supply API user guide

Introduction

This document describes the application program interface (API) for the BP2595 bipolar power supply. The API allows application software to monitor and control up to eight model BP2595 power supplies over USB. The API is distributed as part of a free software development kit (SDK).

Windows SDK

The Windows SDK includes the following files:

- `BP2595.dll` – API executable. This must reside in the same directory as your application program (or in the DLL search path). This is a 32-bit DLL, which means it can be used on both 32- and 64-bit machines, but only with 32-bit applications.
- `BP2595.lib` – Linker library for `BP2595.dll`. This is used when building your application, to link the application to the DLL.
- `apiBP2595.h` – C/C++ header. This must be included in all C/C++ source files that call API functions.
- `bp2595.vb` – VB.NET module. This must be included in all VB.NET projects that call API functions.
- Source code, project files, and pre-built executables of C and VB.NET applications that show how to use the API.

Overview

Terminology

The following terms are used throughout this document:

- A *device* is a BP2595 unit.
- A *meter* is circuitry within a device that measures voltage, current or temperature.
- A *supply* is circuitry within a device that generates the HV output.
- A *setpoint* is the programmed output voltage of a supply. Note that this is the desired output level, which may not match the associated meter due to output loading or other reasons.
- *HVEN* is the programmed output enable of a supply. High voltage may be generated when HVEN is enabled, and will not be generated when HVEN is disabled.
- A *client* is a computer which connects to a device via USB.
- A *console* is the operator controls on a device front panel.
- The *mode* determines the entity that controls the supply: console (local mode) vs. client (remote mode). The mode is selected via a console switch.

Data types

These fundamental data types are used in API function arguments:

- u8 – unsigned 8-bit integer
- u16 – unsigned 16-bit integer
- s32, int – signed 32-bit integer

Error codes

Most API functions return an error code. `BP2595_ERR_OK` (zero) is returned if no errors are detected; all other codes indicate the function failed. See the header file (`apiBP2595.h` or bp`2595.vb`) for a list of error codes.

Device address

Most API functions communicate with a particular device and thus include a `devaddr` argument to designate the device. The device address is an integer value between 0 and 7, corresponding to the device's USB address switch position. When calling an API function, the `devaddr` value must match the target device's switch setting. By default, devices are factory-programmed to address 0.

Admin functions

BP2595_OpenApi

```
int BP2595_OpenApi(int *devflags);
```

Arguments

`devflags` – buffer to receive bit flags indicating detected devices.

Description

This function opens the API and detects all devices. Each bit in `devflags` is associated with a particular device address. Bit 0 is associated with address 0, bit 1 with address 1, etc. A logic '1' bit indicates a device is detected at the associated address; '0' indicates no device is detected. The application program must call this once before calling any other API functions. No other API calls are permitted before calling this function.

Thread safety

This function is not thread safe.

Example

```
int addr, mask, devflags;
int errcode = BP2595_OpenApi(&devflags);
if (errcode == BP2595_ERR_OK) {
  if (devflags == 0)
    printf("no devices detected\n");
  else {
    printf("detected the following devices:\n");
    for (addr = 0, mask = 1; addr < 8; addr++, mask <<= 1) {
      if (devflags & mask)
        printf("address %d\n", addr);
    }
  }
}
else
  printf("ERROR! Problem opening API, error code = %d.\n", errcode);
```

BP2595_CloseApi

```
int BP2595_CloseApi(void);
```

Description

This function closes all open devices and frees all resources used by the API. It must be the called once before the application program closes.

No API calls are permitted after calling this function (except `BP2595_OpenApi`, to re-open the API after closing it). Consequently, multi-threaded applications must explicitly close all devices (via `BP2595_CloseDevice`) and wait for all device-dependent threads to terminate before calling this function.

Thread safety

This function is not thread safe.

Example

```
int errcode = BP2595_CloseApi();   // Close the API.
if (errcode != BP2595_ERR_OK)
  printf("ERROR! Problem closing API, error code = %d.\n", errcode);
```

BP2595_OpenDevice

```
int BP2595_OpenDevice(int devaddr, BP2595_STATUS *status);
```

Arguments

`devaddr` – device address in the range 0 to 7.
`status` – buffer to receive device status.

Description

This function opens a device and returns its current status information in `status`. The application program must call this function once for each device to enable communication with the device.

This function does not affect the device's control settings (setpoint, HVEN). The current control settings can be obtained from the returned `status`.

Thread safety

Two or more threads can safely open mutually-exclusive devices at the same time (i.e., it is unsafe for two threads to simultaneously open the same device).

Example

```
BP2595_STATUS status;
int errcode = BP2595_OpenDevice(0, &status);  // Open device at address 0.
if (errcode != BP2595_ERR_OK)
  printf("ERROR! Problem opening system, error code = %d.\n", errcode);
```

BP2595_CloseDevice

```
int BP2595_CloseDevice(int devaddr);
```

Arguments

`devaddr` – device address in the range 0 to 7.

Description

This function closes a device. If an application thread is waiting in a blocking API function (`BP2595_GetStatusUpdate` or `BP2595_GetMeterUpdate`) then the blocking function will unblock and return an error code indicating the device was closed. The application must call this function when it has finished communicating with a device.

This function does not affect the device's control settings (setpoint, HVEN). Consequently, if the setpoint or HVEN are required to be set to particular values (e.g., 0 kV, HV disabled) when the device is closed, the application must program those values before calling this function.

Thread safety

Two or more threads can safely close mutually-exclusive devices at the same time (i.e., it is unsafe for two threads to simultaneously close the same device).

Example

```
BP2595_CloseDevice(0);  // Close device at address 0.
```

Utility functions

BP2595_ErrString

```
const char * BP2595_ErrString(int errcode);
```

Arguments

`errcode` – error code returned by an API function.

Description

This function accepts an error code and returns a pointer to a text string that describes the error.

Thread safety

This function is thread safe.

Example

```
int errcode = BP2595_OpenDevice(0, &status);   // Try to open device at address 0.
if (errcode != BP2595_ERR_OK)                  // If failed to open device
  printf("BP2595_OpenDevice error: %s\n", BP2595_ErrString(errcode)); // display error message.
```

BP2595_GetApiVersion

```
int BP2595_GetApiVersion(BP2595_VERSION *ver);
```

Arguments

`ver` – buffer to receive API version info.

Description

This function returns the API version number.

Thread safety

This function is thread safe.

Example: Display API version

```
BP2595_VERSION ver;
int errcode = BP2595_GetApiVersion(&ver);
if (errcode != BP2595_ERR_OK)
  printf("ERROR: %s\n", BP2595_ErrString(errcode));
else
  printf("api version = %d.%d.%d\n", ver.Major, ver.Minor, ver.Build);
```

BP2595_GetDeviceVersions

```
int BP2595_GetDeviceVersions(int devaddr, BP2595_VERSION_INFO *info);
```

Arguments

`devaddr` – device address in the range 0 to 7.
`info` – receives device version info.

Description

This function returns the firmware and hardware versions.

Thread safety

This function is thread safe.

Example: Display device versions

```
BP2595_VERSION_INFO info;
int errcode = BP2595_GetDeviceVersions(0, &info);
if (errcode != BP2595_ERR_OK)
  printf("ERROR: %s\n", BP2595_ErrString(errcode));
else {
  printf("firmware version %d.%d.%d\n", info.DevFw.Major, info.DevFw.Minor, info.DevFw.Build);
  printf("hardware rev %c\n", info.PwbRev);
}
```

Status functions

Device status

Device status is conveyed in a `BP2595_STATUS` structure. The status information includes setpoint, HVEN, mode, and various flags that indicate faults and logged events.

The client initially receives the current device status when it calls `BP2595_OpenDevice`. Once the device is open, it may call `BP2595_GetStatusUpdate` to obtain device status and to be notified when a status change occurs.

Status updates

The device automatically transmits a status update message to the client upon any of these events:

- Setpoint value change.
- HVEN change.
- Mode change.
- Fault condition change.

The API automatically receives these update messages and keeps track of the current device status, and optionally will notify the application when device status changes. This allows an application program to be structured for event-driven or polled operation:

- In event-driven applications, a client thread can block (in `BP2595_GetStatusUpdate`) while waiting for a status change and be automatically notified (unblocked) when a status change occurs.
- In polling applications, a client application can read the current status at any time without blocking.

BP2595_GetStatusUpdate

```
int BP2595_GetStatusUpdate(int devaddr, BP2595_STATUS *status, int tmax);
```

Arguments

`devaddr` – device address in the range 0 to 7.
`status` – buffer to receive status information.
`tmax` – maximum time (in milliseconds) to wait for a status report.

Description

This function receives notification of a device status change and returns the current device status. It blocks the calling thread for up to `tmax` milliseconds while waiting to receive a status update. The received status information is returned in user-allocated `status` buffer. The content of `status` will be changed only if the function executes normally (without error).

The returned `status` information includes `Events` flags that indicate which event(s) occurred. Event flags "accumulate" until this function is called to read them. This ensures that events will not be lost if a new status update arrives before the client reads the previous update.

The device maintains an internal log of fault events which act as permissives for various hardware functions. The returned `status` information includes `Log` flags that indicate the logged events. Logged events must be cleared by calling `BP2595_ClearEventLog`. This mechanism ensures that the client must acknowledge fault conditions (and presumably take corrective action) before resuming normal hardware operation.

Blocking behavior is controlled by `tmax`. In all cases, the function will return immediately if a status update has already been received. `tmax` may be set to any of these values:

- Zero. The function will execute normally if an update was received, otherwise it will return `BP2595_ERR_TIMEOUT`.
- `BP2595_INFINITE_WAIT`. The function will block until an update is received and then execute normally. If another thread closes the device while the function is blocking, the function will fail immediately and return `BP2595_ERR_DEVICE_CLOSED`.
- A positive, non-zero time specified in milliseconds (e.g. 1500 = 1.5 seconds). The function will block until an update is received or `tmax` milliseconds have elapsed. It will execute normally if an update is received, otherwise it will fail and return `BP2595_ERR_TIMEOUT`. If another thread closes the device while the function is blocking, the function will fail immediately and return `BP2595_ERR_DEVICE_CLOSED`.

Thread safety

This function is thread-safe, however multiple threads are not allowed to wait for status updates from the same device at the same time. This function will fail and return `BP2595_ERR_RESOURCE_BUSY` if it is currently being executed by another client thread for the same device.

Example: Event-driven operation

```
void StatusMonitorThread(int *devaddr)
{
  BP2595_STATUS status;
  while (BP2595_GetStatusUpdate(devaddr, &status, BP2595_INFINITE_WAIT) == BP2595_ERR_OK)
    DisplayStatusUpdate(&status);
}
```

Example: Polled operation

```
void PollStatus(int *devaddr)
{
  BP2595_STATUS status;
  int errcode = BP2595_GetStatusUpdate(devaddr, &status, 0); // tmax=0 to check for update without blocking
  switch (errcode) {
    case BP2595_ERR_OK:       DisplayStatusUpdate(&status); break;
    case BP2595_ERR_TIMEOUT:  printf("status unchanged\n"); break;
    default:                  printf("ERROR: %s\n", BP2595_ErrString(errcode));
  }
}
```

Example: Report status changes

```
void DisplayStatusUpdate(BP2595_STATUS *status)
{
  int e = status->StatusSystem.Events;
  int s = status->StatusSystem.Live;
  printf("events:\n");
//IF (THIS VALUE CHANGED) REPORT ASSOCIATED STATUS CHANGE ---------------------
  if (e & NOTIFY_SETPT  ) printf("setpoint = %d V\n",  status->SetpointV);
  if (e & NOTIFY_HVEN   ) printf("hven     = %s\n", (s & STATE_HVEN) ? "on" : "off");
  if (e & NOTIFY_REMOTE ) printf("mode     = %s\n", (s & STATE_REMOTE) ? "remote" : "local");
}
```

BP2595_ClearEventLog

```
int BP2595_ClearEventLog(int devaddr, u8 flags);
```

Arguments

`devaddr` – device address in the range 0 to 7.
`flags` – event flags to clear.

Description

This function clears a device's internal event log. See `BP2595_GetStatusUpdate` for an explanation of logged faults.

Thread safety

This function is thread safe.

Example: Clear logged faults

```
u8 flags = EVLOG_UVLO | EVLOG_OVERTEMP | EVLOG_COM_TOUT;
if (BP2595_ClearEventLog(0, flags) != BP2595_ERR_OK)
  printf("ERROR: %s\n", BP2595_ErrString(errcode));
```

Communication watchdog

The device implements a communication watchdog timer which monitors the elapsed time between commands. The watchdog is disabled by default upon device power-up or reset.

When enabled, the watchdog will timeout if it fails to receive a command within the maximum allowed time. When this happens, the device will:

- Set the `STATE_COM_TOUT` status flag to indicate communication timeout.
- Set the `STICKY_COM_TOUT` event log flag.
- Transmit a status update to indicate the status change.
- Disable the HV supply (clear HVEN). The HV supply will remain disabled until the `STICKY_COM_TOUT` event log flag is cleared. The client can clear this flag by calling `BP2595_ClearEventLog`.

The client must execute commands frequently enough to avoid a watchdog timeout. If there is no need to execute a command other than to avoid watchdog timeout, it is recommended to call `BP2595_KeepAlive`.

BP2595_SetComWatchdog

```
int BP2595_SetComWatchdog(int devaddr, u16 interval);
```

Arguments

`devaddr` – device address in the range 0 to 7.

`interval` – maximum allowed time in milliseconds between commands; set to 0 to disable.

Description

This function configures a device's communication watchdog timer. The `interval` argument specifies the maximum time allowed between consecutive client commands. Set `interval` to 0 (default upon power-up) to disable the watchdog.

Thread safety

This function is thread safe.

Example

```
int errcode = BP2595_SetComWatchdog(0, 250);  // allow 0.25 seconds max between commands
if (errcode != BP2595_ERR_OK)
  printf("ERROR: %s\n", BP2595_ErrString(errcode));
```

BP2595_KeepAlive

```
int BP2595_KeepAlive(int devaddr);
```

Arguments

`devaddr` – device address in the range 0 to 7.

Description

This function sends a "no operation" command to a device. When the device's communication watchdog is enabled, this function can be used to prevent watchdog timeouts if there are no other necessary communications.

Thread safety

This function is thread safe.

Example

```
int errcode = BP2595_KeepAlive(0);  // "ping" device at address 0 to prevent watchdog timeout
if (errcode != BP2595_ERR_OK)
  printf("ERROR: %s\n", BP2595_ErrString(errcode));
```

Meters

Meter updates

The client acquires meter data via automatically transmitted "meter update" messages. A device can be configured to transmit meter data ("meter updates") to the client on a periodic schedule. The API buffers the most recently received meter data and, if desired, will notify the client when new meter data arrives. This mechanism allows any strategy of single- or multi-threaded meter monitoring:

- In event-driven applications, the client can wait for a meter update and be automatically notified when a meter update arrives.
- In polled applications, the client can check for meter updates at any time and, if an update is available, receive the new meter data.

BP2595_SetMeterInterval

```
int BP2595_SetMeterInterval(int devaddr, u16 interval);
```

Arguments

`devaddr` – device address in the range 0 to 7.
`interval` – update interval in milliseconds.

Description

This function enables or disables meter updates and controls the update rate. Upon device power up or reset, automatic updates are disabled by default. The `interval` argument specifies the time between updates in milliseconds. Set `interval` to a non-zero value to program the update rate and enable automatic updates (note: if updates are already enabled, this will change the update rate "on-the-fly"). Set `interval` to zero to disable automatic updates.

Example

```
if (BP2595_SetMeterInterval(0, 200) != BP2595_ERR_OK)  // send report every 200 milliseconds
  printf("ERROR: %s\n", BP2595_ErrString(errcode));
```

BP2595_GetMeterInterval

```
int BP2595_GetMeterInterval(int devaddr, u16 *interval);
```

Arguments

`devaddr` – device address in the range 0 to 7.
`interval` – buffer to receive meter update interval in milliseconds.

Description

This function returns the programmed meter update interval.

Thread safety

This function is thread safe.

Example

```
u16 interval;
if (BP2595_GetMeterInterval(0, &interval) == BP2595_ERR_OK)
  printf("meter update interval = %d ms\n", interval);
else
  printf("ERROR: %s\n", BP2595_ErrString(errcode));
```

BP2595_GetMeterUpdate

```
 int BP2595_GetMeterUpdate(int devaddr, BP2595_METERS *meters, int tmax);
```

Arguments

`devaddr` – device address in the range 0 to 7.
`meters` – buffer to receive meter data.
`tmax` – maximum time to wait in milliseconds.

Description

This function waits for a meter update and returns the meter data. It will block the calling thread for up to tmax milliseconds while waiting to receive a meter update. The received meter data is returned in meters. The contents of meters will change only if the function executes normally (without error).

The returned `meters` information is the most recently received meter data. The previous meter data will be lost if a new meter update arrives before the client reads the previous update.

Blocking behavior is controlled by `tmax`. In all cases, the function will return immediately if a meter update has already been received. `tmax` may be set to any of these values:

- Zero. The function will execute normally if an update was received, otherwise it will return `BP2595_ERR_TIMEOUT`.
- `BP2595_INFINITE_WAIT`. The function will block until an update is received and then execute normally. If another thread closes the device while the function is blocking, the function will fail immediately and return `BP2595_ERR_DEVICE_CLOSED`.
- A positive, non-zero time specified in milliseconds (e.g. 1500 = 1.5 seconds). The function will block until an update is received or `tmax` milliseconds have elapsed. It will execute normally if an update is received, otherwise it will fail and return `BP2595_ERR_TIMEOUT`. If another thread closes the device while the function is blocking, the function will fail immediately and return `BP2595_ERR_DEVICE_CLOSED`.

Thread safety

This function is thread-safe, however multiple threads are not allowed to wait for meter updates at the same time. This function will fail and return BP2595_ERR_RESOURCE_BUSY if it is currently being executed by another client thread for the same device.

Example: Event-driven operation

```
void MeterMonitorThread(int *devaddr)
{
  BP2595_METER meters;
  while (BP2595_GetMeterUpdate(0, &meters, BP2595_INFINITE_WAIT) == BP2595_ERR_OK)
    printf("meters: %d V, %d uA \n",  meters->outV, meters->outUA);
}
```

Example: Polled operation

```
void PollMeters(int *devaddr)
{
  BP2595_METER meters;
  int errcode = BP2595_GetMeterUpdate(devaddr, &meters, 0); // tmax=0 to check for update w/ noblocking
  if (errcode == BP2595_ERR_OK)
    printf("meters: %d V, %d uA \n",  meters->outV, meters->outUA);
  else if (errcode == BP2595_ERR_TIMEOUT)
    printf("meters unchanged\n");
  else
    printf("ERROR: %s\n", BP2595_ErrString(errcode));
}
```

HV control

In remote mode, the client controls the device's setpoint and HVEN by calling `BP2595_ProgramSetpoint` and `BP2595_ProgramHvEnable`. When the device is operating in local mode, the client is not permitted to change the setpoint or HVEN and any attempt to do so will have no effect on device control settings.

Bumpless setpoint transfer

The API implements bumpless setpoint transfer, meaning that the device's setpoint will not change when the device is switched between local and remote modes.

Bumpy/bumpless HV enable transfer

HVEN transfers are bumpless when the device is switched from local to remote mode. However, when the device is switched from remote to local mode, HVEN may suddenly change as a result of the mode change:

HVEN prior to mode change	Console HV Enable switch	Transfer type	Effect on HVEN
Enabled	Off	Bumpy	HVEN becomes disabled upon switching to local mode.
	On	Bumpless	No change: HVEN remains enabled.
Disabled	Off	Bumpless	No change: HVEN remains disabled.
	On	Bumpy	HVEN becomes enabled upon switching to local mode. **Note: This may cause high voltage to appear on the HV output.**

BP2595_ProgramSetpoint

```
int BP2595_ProgramSetpoint(int devaddr, s32 volts);
```

Arguments

`devaddr` – device address in the range 0 to 7.
`volts` – desired output voltage in Volts.

Description

When the device is operating in remote mode, this function will program the output voltage setpoint. In local mode, this function has no effect on the device and will return `BP2595_ERR_MODE_LOCKOUT`.

Upon successful execution of this command, the device will transmit a status update to indicate the setpoint has changed. The output voltage will change to match the setpoint if all of the following conditions are met:

- HV is enabled.
- The HV load is not drawing excessive current.
- The device has no faults (undervoltage lockout, over-temperature, communication watchdog timeout).

Thread safety

This function is thread safe.

Example

```
if (BP2595_ProgramSetpoint(0, -30000) != BP2595_ERR_OK)  // Change HV output to -30.0 kV
  printf("ERROR: %s\n", BP2595_ErrString(errcode));
```

BP2595_ProgramHvEnable

```
int BP2595_ProgramHvEnable(int devaddr, int enable);
```

Arguments

`devaddr` – device address in the range 0 to 7.
`enable` – 1 = set HVEN; 0 = clear HVEN.

Description

When the device is operating in remote mode, this function will enable or disable the HV supply by setting or clearing HVEN. In local mode, this function has no effect on the device and will return `BP2595_ERR_MODE_LOCKOUT`.

Upon successful execution of this command, the device will transmit a status update to indicate HVEN has changed.

Thread safety

This function is thread safe.

Example

```
if (BP2595_ProgramHvEnable(0, 1) != BP2595_ERR_OK)  // Enable HV output
  printf("ERROR: %s\n", BP2595_ErrString(errcode));
```