

Middleware User Guide

TABLE OF CONTENTS

Table of Contents.....	1
List of Figures.....	2
1. Purpose.....	3
2. Scope	3
3. Definitions, Acronyms and Abbreviations.....	3
4. References	3
5. Overview.....	4
6. VMM.....	5
6.1. Setup.....	5
6.1.1. <i>Running the VMM as a background service for Windows</i>	5
6.1.2. <i>Running the VMM as a background service for Linux</i>	6
6.2. Operation.....	8
6.3. Command Structure	9
6.3.1. <i>Generic Commands</i>	9
6.3.2. <i>Module Commands</i>	9
6.3.3. <i>Usage Case</i>	10
6.3.4. <i>Error Code</i>	11
6.4. Code Snippets	12
6.4.1. <i>Python</i>	12
6.4.2. <i>C/C++</i>	12
7. Python Library	14
7.1. Setup.....	14
7.2. VBMB Return Struct.....	14
7.3. Operation.....	15
8. Revision History.....	16
9. Legal Notices	17

LIST OF FIGURES

<i>Figure 6.1: Check the service status in Task Manager.....</i>	<i>5</i>
<i>Figure 6.2: Change the access permission for the VBMB modules</i>	<i>6</i>
<i>Figure 6.3: Run the VMM as a background service</i>	<i>7</i>
<i>Figure 6.4: Check the processes in the System Monitor.....</i>	<i>7</i>
<i>Figure 6.5: VBMB Module Manager</i>	<i>8</i>
<i>Figure 6.6: Code snippet in Python</i>	<i>12</i>
<i>Figure 6.7: TestSocket.h.....</i>	<i>12</i>
<i>Figure 6.8: TestSocket.cpp</i>	<i>13</i>
<i>Figure 7.1: VBMBReturnStruct.....</i>	<i>14</i>
<i>Figure 7.2: Example code using VBMBPyLib</i>	<i>15</i>
<i>Figure 7.3: Example code output</i>	<i>15</i>

1. PURPOSE

The purpose of this document is to act as a user guide for development using VBMB's Middleware.

2. SCOPE

This document provides the details about the various components of VBMB's Middleware and how they can be used by end users for SW development.

3. DEFINITIONS, ACRONYMS AND ABBREVIATIONS

Word or Acronyms	Definition
FW	Firmware
SW	Software
API	Application Programming Interface
VSP	Smart Pump
VTC	TEC Controller
VPM	Precision Motion Controller
VFW	Motorized Filter Wheel
VMS	Motion Stage
VMM	VBMB Module Manager

4. REFERENCES

Document Description	Document No.
VTC Software API document	VTC-UM-0005
VSP Software API document	VSP-UM-0005
VFW Software API document	VFW-UM-0005

5. OVERVIEW

All VBMB modules can communicate via USB serial connection or RS-485 bus with the USB bridge module. A middleware API library is provided to assist end users in developing applications for VBMB modules. These APIs shorten development time by acting as an intermediary between the top-level user application and the low-level firmware running inside the module.

There are two methods of calling the APIs:

1. VMM Service

The VBMB Module Manager (VMM) hosts a localhost socket server on port 58288 and listens to incoming connections and messages from clients. A client can be any application developed by the end user to communicate with this server.

2. VBMB Python Library

The pip module which encapsulates the APIs for every VBMB module into a respective module class. Ideal for bring-up and testing of applications directly over the command line or in a Python environment.

The details of all APIs for each VBMB module are listed in the respective API document.

6. VMM

This section provides a detailed description of the VBMB Module Manager (VMM).

The VMM acts as an interface for implementing the API commands for VBMB modules in a non-blocking manner.

6.1. SETUP

The VMM is packaged as an executable file called **"VBMB_Module_Manager_x.y.z.exe"** for Windows and **"VBMB_Module_Manager_x.y.z"** for Linux where x.y.z are numbers indicating the release version of the file wherein:

- x is the major version
- y is the minor version
- z is the patch version

6.1.1. Running the VMM as a background service for Windows

The VMM can be run as a service that operates in the background by just double click the VMM executable file for the Windows system.

The run log can be find in the "C:\Users\UserName\VBMB\VMM\log.txt" folder for Windows.

The status of the service can be checked in the Task Manager.

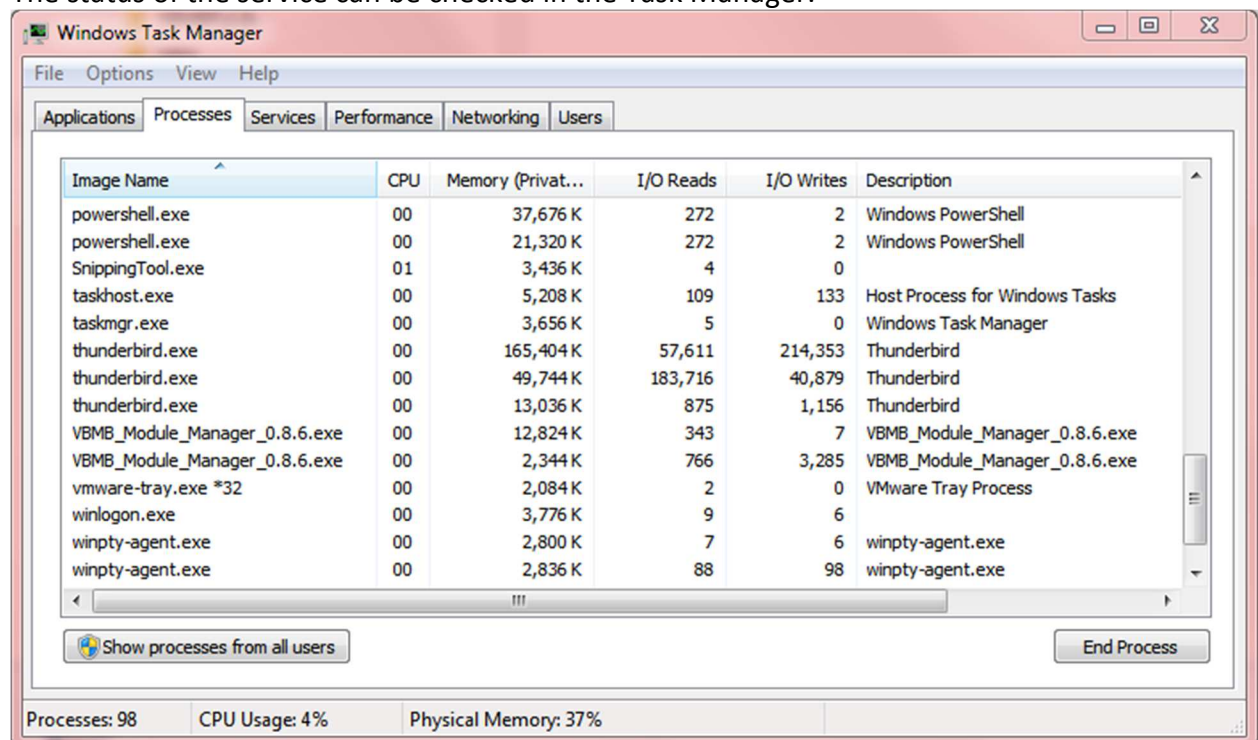


Figure 6.1: Check the service status in Task Manager

6.1.2. Running the VMM as a background service for Linux

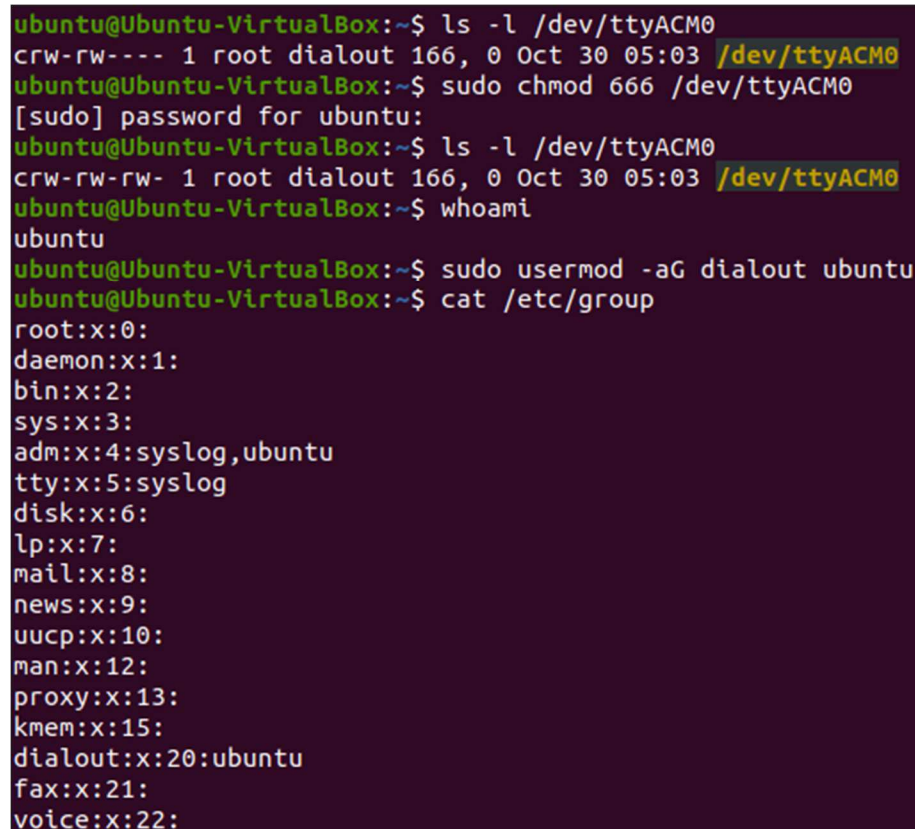
For the Linux system, user need to give the access permission for the VBMB modules.

Used the 'chmod' command to give the device access permission when device plugged in every time.

```
$ sudo chmod 666 /dev/ttyACM0
```

Used the 'usermod' command to give the device access permission that need reboot to make effect permanently.

```
$ sudo usermod -aG dialout <UserName>
```



```
ubuntu@Ubuntu-VirtualBox:~$ ls -l /dev/ttyACM0
crw-rw---- 1 root dialout 166, 0 Oct 30 05:03 /dev/ttyACM0
ubuntu@Ubuntu-VirtualBox:~$ sudo chmod 666 /dev/ttyACM0
[sudo] password for ubuntu:
ubuntu@Ubuntu-VirtualBox:~$ ls -l /dev/ttyACM0
crw-rw-rw- 1 root dialout 166, 0 Oct 30 05:03 /dev/ttyACM0
ubuntu@Ubuntu-VirtualBox:~$ whoami
ubuntu
ubuntu@Ubuntu-VirtualBox:~$ sudo usermod -aG dialout ubuntu
ubuntu@Ubuntu-VirtualBox:~$ cat /etc/group
root:x:0:
daemon:x:1:
bin:x:2:
sys:x:3:
adm:x:4:syslog,ubuntu
tty:x:5:syslog
disk:x:6:
lp:x:7:
mail:x:8:
news:x:9:
uucp:x:10:
man:x:12:
proxy:x:13:
kmem:x:15:
dialout:x:20:ubuntu
fax:x:21:
voice:x:22:
```

Figure 6.2: Change the access permission for the VBMB modules

The VMM can be run as a service that operates in the background.

Used the 'nohup' command to make the VMM immune to hangups which means it continues running in the background even if the terminal is closed.

```
$ nohup ./VBMB_Module_Manager_0.8.7 &
```

Used the 'disown' command to make the VMM independent of the shell.

```
$ disown %1
```

```

ubuntu@Ubuntu-VirtualBox:~$ cd Python/
ubuntu@Ubuntu-VirtualBox:~/Python$ ./VMBB_Module_Manager_0.8.7
2024-10-30 05:29:22,744 selector_events.py: Using selector: EpollSelector
2024-10-30 05:29:22,764 Module.py: /dev/ttyACM0: Serial port opened
2024-10-30 05:29:22,766 VPM.py: /dev/ttyACM0 S=> '/00#GetDeviceInfo()'
2024-10-30 05:29:22,766 VPM.py: /dev/ttyACM0 <=R '@F3#GetDeviceInfo(0x0000,VPM01
-A,00-2344-0012,0x2099326C5950,V01.99,USB)'
2024-10-30 05:29:23,768 vio_manager.py: VMM-F3 connected with USB.
2024-10-30 05:29:23,771 vio_manager.py: Serving on ('127.0.0.1', 58288)
^CVBMB Module Manager: User Shutdown
2024-10-30 05:29:39,276 vio_manager.py: Received stop signal. Cancelling tasks .
..
2024-10-30 05:29:39,338 vio_manager.py: Exiting now ...
ubuntu@Ubuntu-VirtualBox:~/Python$ nohup ./VMBB_Module_Manager_0.8.7 &
[1] 2847
ubuntu@Ubuntu-VirtualBox:~/Python$ nohup: ignoring input and appending output to
'nohup.out'
jobs
[1]+  Running                  nohup ./VMBB_Module_Manager_0.8.7 &
ubuntu@Ubuntu-VirtualBox:~/Python$ disown %1
ubuntu@Ubuntu-VirtualBox:~/Python$ jobs
ubuntu@Ubuntu-VirtualBox:~/Python$ ps
  PID TTY          TIME CMD
 2833 pts/0        00:00:00 bash
 2847 pts/0        00:00:00 VMBB_Module_Man
 2848 pts/0        00:00:00 VMBB_Module_Man
 2850 pts/0        00:00:00 ps
ubuntu@Ubuntu-VirtualBox:~/Python$

```

Figure 6.3: Run the VMM as a background service

The run log can be find in the "/Home/VBMB/VMM/log.txt" folder for Linux.

The status of the background service can be checked in the System Monitor.

Processes									
Process Name	User	% CPU	ID	Memory	Disk read tot	Disk write tot	Disk read	Disk write	Priority
systemd	ubuntu	0	1208	2.2 MiB	36.5 MiB	24.9 MiB	N/A	N/A	Normal
tracker-miner-fs	ubuntu	0	1217	7.8 MiB	4.6 MiB	N/A	N/A	N/A	Very Low
update-notifier	ubuntu	0	1832	6.0 MiB	65.3 MiB	2.5 MiB	N/A	N/A	Normal
VMBB_Module_Manager_0.8.7	ubuntu	0	1968	96.0 KiB	N/A	12.2 MiB	N/A	N/A	Normal
VMBB_Module_Manager_0.8.7	ubuntu	2	1969	9.7 MiB	N/A	8.0 KiB	N/A	N/A	Normal
VBoxClient	ubuntu	0	1399	324.0 KiB	N/A	N/A	N/A	N/A	Normal
VBoxClient	ubuntu	0	1400	3.2 MiB	280.0 KiB	4.0 KiB	N/A	N/A	Normal
VBoxClient	ubuntu	0	1409	320.0 KiB	N/A	N/A	N/A	N/A	Normal
VBoxClient	ubuntu	0	1410	396.0 KiB	64.0 KiB	4.0 KiB	N/A	N/A	Normal
VBoxClient	ubuntu	0	1416	324.0 KiB	N/A	N/A	N/A	N/A	Normal
VBoxClient	ubuntu	0	1417	376.0 KiB	N/A	4.0 KiB	N/A	N/A	Normal
VBoxClient	ubuntu	0	1422	328.0 KiB	N/A	N/A	N/A	N/A	Normal
VBoxClient	ubuntu	0	1423	380.0 KiB	64.0 KiB	4.0 KiB	N/A	N/A	Normal

Figure 6.4: Check the processes in the System Monitor

6.2. OPERATION

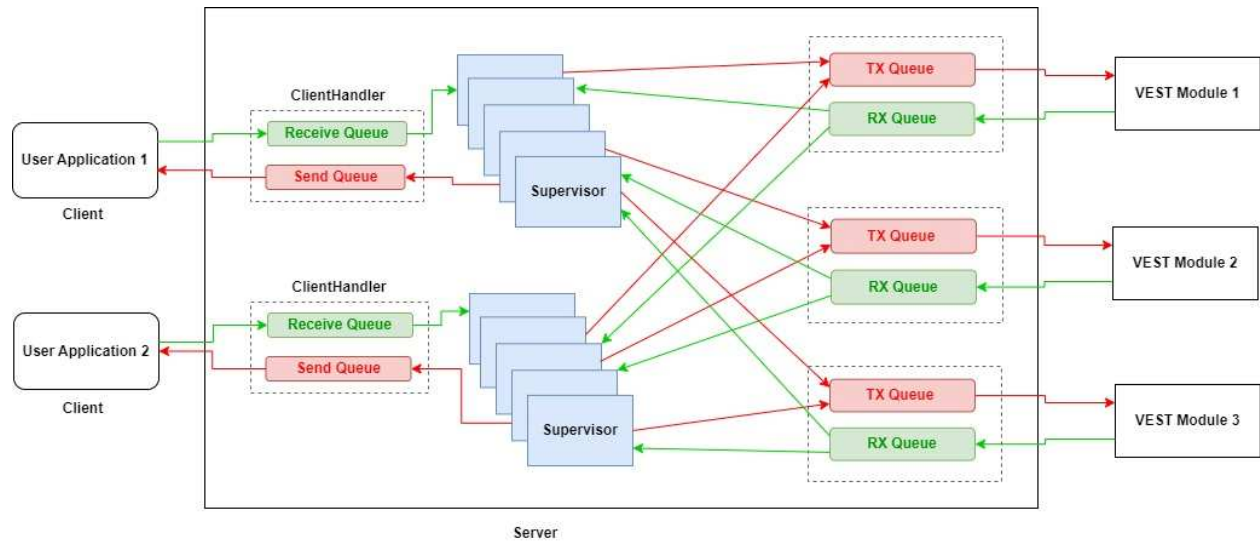


Figure 6.5: VBMB Module Manager

Figure 6.5 shows the structure of the VMM and a case when three different VBMB modules are connected to the users' computer. There are two user application's running on the computer and communicating with the VMM. The sequence of events from the VMM's perspective shall be as follows:

- The VMM hosts a local host server on port **58288** and instantiates a **ClientHandler** object for each client¹ connected to the VMM. A **ClientHandler** contains a pair of **Send and Receive queues**.
- Since all VBMB modules communicate via USB, the VMM scans all the COM ports on the user's computer, identifies the type of VBMB module connected and assigns a special **VMM ID** to each available module.
- VMM spawns a pair of **TX and RX queues** for each VBMB module. All commands pertaining to a specific VBMB module are transported through these queues.
- A **Supervisor** is instantiated for every message in the Receive Queue of each ClientHandler. The Supervisor parses the message and checks if the message was correctly transported to the respective VBMB module or if the task was aborted midway either by the user or the VMM.

VMM commands must contain the **VMM ID as a prefix** which allows the supervisor to parse the received command and send it to the correct VBMB module. The module's response is received and parsed by the supervisor and the relevant **VMM error code** is appended to the response before sending it out to the respective client via the respective ClientHandler.

¹ In this case, since two user applications are connected, two client handlers are present.

6.3. COMMAND STRUCTURE

All VMM commands are human readable ASCII commands terminated by a newline (0x0A) character. There are 2 types of commands available – Generic commands and Module commands

6.3.1. Generic Commands

A) ScanForDevices

Get the details of all VBMB modules connected to the VMM. **Recommended to send this command every time a module is connected/disconnected to know the VMM ID and status of all modules.**

Prototype: ScanForDevices()

Response:

Parameter	Description
response	Returns a dictionary in the following format <VMM ID> = { "device" : A string indicating the type of VBMB module - VSP, VTC etc. "port" : The COM port on which the module is present, "hw_version" : The hardware version of the module, "fw_version" : The firmware version of the module, "uuid" : The Universally Unique Identifier (UUID) of the module, "status": "connected"/"disconnected" depending on whether the module is connected/disconnected }

B) GetMWVersion

Get the VMM version.

Prototype: GetMWVersion()

Response: GetMWVersion(version)

Parameter	Description
version	Returns the VMM version

6.3.2. Module Commands

These are the API commands for each individual VBMB module when it is connected to the VMM. All module commands follow the same general format where the various parameters are separated by a colon (":") and terminated with a new line (0x0A) character.

Prototype: VMM_id:timeout:command

Parameter	Description	Value
VMM_id	The ID assigned to the VBMB module by the VMM.	Can be looked up by sending the ScanForDevices() command to the VMM
timeout	Time (in seconds) to wait for response before timing out	Any non-negative integer Default timeout value = 180 seconds
command	The API command to be sent to the VBMB module	Refer to the API document for the respective VBMB module for command format and parameters

Response: VMM_id:response

Parameter	Description
VMM_id	Same as above
response	The response from the VBMB module. Refer to the API document for the respective VBMB module for response format and error code details

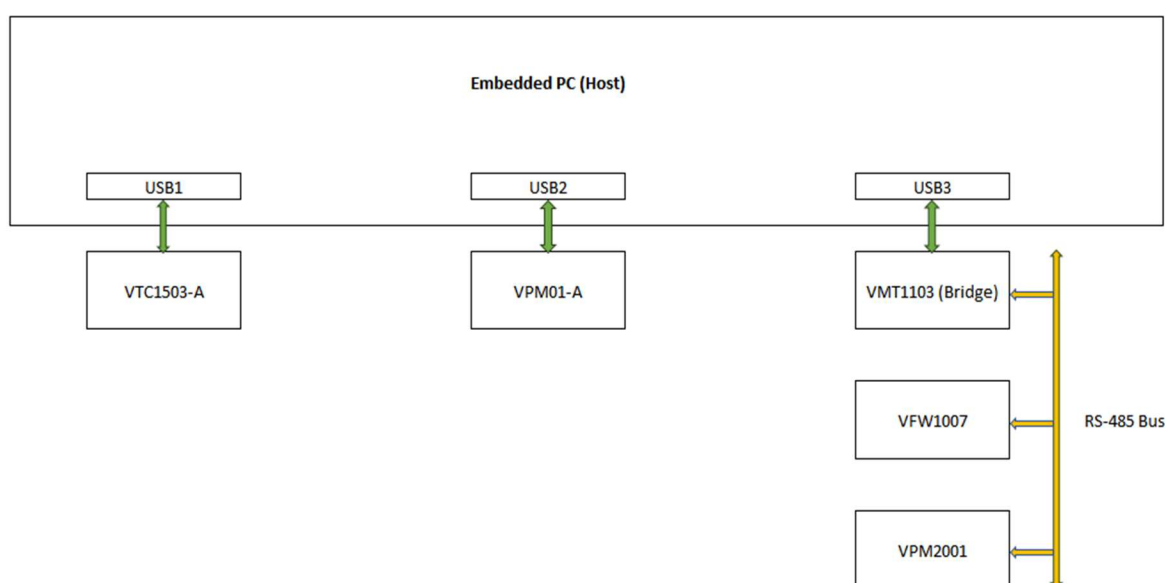
Example: Assuming user is trying to read the firmware version of a connected VTC to which the VMM has assigned an id VMM-75 and users wish to have a timeout of 1 minute.

Sent: VMM-0A:60:GetFWVersion()\n

Received: VMM-0A:GetFWVersion(E00,0408)\n

6.3.3. Usage Case

Example: Five VBMB modules connected to the system and the hardware architecture showed at the below figure.



```
Client ('127.0.0.1', 57170) ==> ScanForDevices()
Client ('127.0.0.1', 57170) <= {
  "VMM-0A": {"device": "VTC1503-A", "port": "COM30", "hw_version": "TC2327K153-RA0A00001",
    "fw_version": "0408", "uuid": "64206C325059", "status": "Connected+1"},
  "VMM-F3": {"device": "VPM01-A", "port": "COM43", "hw_version": "00-2344-0012",
    "fw_version": "V01.99", "uuid": "0x2099326C5950", "status": "Connected+1"},
  "VMM-2D": {"device": "VMT1103", "port": "COM49", "hw_version": "00-2425-0002",
    "fw_version": "V01.73", "uuid": "0x206732975950", "status": "Connected+1"},
  "VMM-1D": {"device": "VFW1007", "port": "COM49", "hw_version": "00-2425-0003",
    "fw_version": "VB1.16", "uuid": "0x206932965950", "status": "Connected+1"},
  "VMM-73": {"device": "VPM2001", "port": "COM49", "hw_version": "00-2422-0039",
    "fw_version": "V01.73", "uuid": "0x206832965950", "status": "Connected+1"}}
```

All of useful information can be captured by the response of ScanForDevices() command.

- #1. All devices distinguished by VMM ID and listed by the order of serial port.
- #2. "VMM-0A" is the VMM ID. 0x0A is the Bus ID reported by FW that generated by UUID 0x64206C325059. The Bus ID range from 0x01 to 0xFF. If the Bus ID conflicted in the system or want to fix function, the user can change it manually by the utility tools.
- #3. "VMM-0A"- "COM30", "VMM-F3"- "COM43" and "VMM-2D"- "COM49" used the different serial port that means the devices connected with the different USB cable.
- #4. "VMM-2D"- "COM49", "VMM-1D"- "COM49" and "VMM-73"- "COM49" used the same serial port that means "VMM-1D" and "VMM-73" devices connected with the RS-485 bus and "VMM-2D" is the USB bridge module.

6.3.4. Error Code

Error Code	Description
E00	Status OK.
E01	Internal error.
E02	Command cancelled by user.
E03	Command timeout.
E04	Invalid command.
E06	Invalid no. of parameters.
E07	Invalid parameters.
E08	Failed to execute.
E09	Interlock safeguard.
E10	Commands are disabled.

6.4. CODE SNIPPETS

This section contains a quick example on how to communicate with the localhost server. It is assumed that the VMM is already setup and running in the background.

6.4.1. Python

```
1 import socket
2
3 HOST = "127.0.0.1"
4 PORT = 58288
5
6 with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
7     s.connect((HOST, PORT))
8     s.sendall("ScanForDevices()\n".encode())
9     print(f"Received {s.recv(1024)}")
```

Figure 6.6: Code snippet in Python

6.4.2. C/C++

```
1 // TestSocket.h: Include file for standard system include files,
2 // or project specific include files.
3
4 #pragma once
5
6 #include <iostream>
7 #include <stdio.h>
8 #include <Winsock2.h>
9
10
11 #pragma comment(lib, "ws2_32.lib")
12
13 #define PORT 58288
14
15 // TODO: Reference additional headers your program requires here.
```

Figure 6.7: TestSocket.h

```
1 // TestSocket.cpp : Defines the entry point for the application.
2 //
3 #include "TestSocket.h"
4
5 int main(int argc, char *argv[])
6 {
7     WSADATA wsa;
8     int sock = 0, valread = 0, client;
9     struct sockaddr_in server;
10
11     char* msg = "ScanForDevices()\n";
12
13     printf("\nInitialising Winsock...");
14     if (WSAStartup(MAKEWORD(2, 2), &wsa) != 0)
15     {
16         printf("Failed. Error Code : %d", WSAGetLastError());
17         return 1;
18     }
19
20     char buf[1024] = { 0 };
21     if ((sock = socket(AF_INET, SOCK_STREAM, 0)) < 0)
22     {
23         printf("\n Socket creation error \n");
24         return -1;
25     }
26
27     server.sin_family = AF_INET;
28     server.sin_port = htons(PORT);
29     server.sin_addr.s_addr = inet_addr("127.0.0.1");
30
31     if ((client = connect(sock, (struct sockaddr*)&server, sizeof(server))) < 0)
32     {
33         printf("\nConnection Failed \n");
34         return -1;
35     }
36
37     send(sock, msg, strlen(msg), 0);
38     printf("\nMessage sent\n");
39     while(!valread)
40     {
41         valread = recv(sock, buf, 1024, 0);
42         printf("%s\n", buf);
43     }
44     // Closing the connected socket
45     closesocket(sock);
46     WSACleanup();
47
48     return 0;
49 }
```

Figure 6.8: TestSocket.cpp

7. PYTHON LIBRARY

The VBMB Python Library encapsulates APIs for each VBMB module into a python class and packages said classes into a pip package.

7.1. SETUP

The VBMB Python library is packaged as a wheel package titled ***“VBMBPyLib-x.y.z-py2.py3-none-any.whl”*** where x.y.z are numbers indicating the release version of the file wherein:

- x is the major version
- y is the minor version
- z is the patch version

Python 3.8.6 and above is the recommended python version to run the library.

The library can be installed using the following command.

```
pip install VBMBPyLib-x.y.z-py2.py3-none-any.whl
```

7.2. VBMB RETURN STRUCT

All functions in the Python library return a custom data type developed by VBMB called the VBMBReturnStruct.

```
class VBMBReturnStruct:
    def __init__(self):
        # If command is successful, status = True else status = False
        self.status = False

        # Stores return values if any. Used especially when sending commands
        # that are reading from device registers/variables
        self.retval = None

        # Device access ID replied by the device
        self.access_id = None

        # Any error code replied by the device
        self.fw_error = None

        # Stores the return type of the reply. Used mainly for VTC while converting hex strings to other data types
        self.rettype = None

        # Stores any prefix string which the VMM can use to validate the reply message
        self.prefix = ""

        # Stores the type of operation being performed by the command where 'read' or 'write'
        self.optype = "read"
```

Figure 7.1: VBMBReturnStruct

7.3. OPERATION

This section contains a quick example on how to use the Python library to communicate with a VBMB module.

It must be noted that:

- All commands issued via the Python library are **blocking** commands.
- The Python library can only communicate with one VBMB module at a time. If multiple modules are connected, the library shall connect to the first available module.
- The Python library can only support USB serial direct connection module.

```

1  # Use pip install VBMBPyLib.whl to install the VBMB Python Library
2  from VBMBPyLib import VPM
3
4  # It is assumed that the VPM is already connected and the serial port is not held by another application
5  # If multiple VPM are connected, first available VPM port will be opened
6  vpm = VPM.VPM()
7
8  # Uncomment commands as necessary. All commands are blocking.
9
10 # -----Printing VPM system info-----
11 print(vpm.GetFWVersion().retval)
12 print(vpm.Home(motor="MT0", speed="50.0", run_mode="NOW").retval)
13 # -----

```

Figure 7.2: Example code using VBMBPyLib

```

['0x0000', 'V01.73']
['MT0', '0x0081', 'DEVICE_BUSY']

```

Figure 7.3: Example code output

Figure 7.3 shows the output of the code snippet executed in Figure 7.2²

Users can refer to the individual API documents for a complete list of API commands and their syntax as supported by the VBMB Python Library. An example python file for each VBMB module shall also be provided along with the VBMB Middleware files.

² The values seen in Figure 7.3 are subject to change as new firmware is released.

8. REVISION HISTORY

Version	Date Released	Initials	Changes
0.0.1	17 May 2023	LJR	Initial Release
0.0.2	19 Feb 2024	KY	Changed based on the IOM v0.8.2
0.0.3	08 Jul 2024	KY	Changed based on the IOM v0.8.5
0.0.4	26 Jul 2024	KY	Changed based on the VMM v0.8.6
0.0.5	01 Nov 2024	KY	Changed based on the VMM v0.8.7
0.0.6	18 Dec 2024	KY	Add usage case for multi-modules

9. LEGAL NOTICES

The signed agreement between Purchaser and VBMB will govern the sale and purchase of Venture Biotech Modules Business Pte. Ltd.'s ("VBMB") products ("Products"). If no agreement has been concluded, VBMB's terms and conditions of supply will apply.

Testing and other quality control techniques are used to the extent that VBMB deems necessary to support its warranty.

Except where required by law, specific testing of all parameters of each Product is not necessarily performed.

Purchaser must provide adequate design and operating safeguards to minimize inherent or procedural and technical risks associated with Purchaser products and applications. Purchaser is solely responsible for its selection and use of VBMB Products. VBMB assumes no liability for applications assistance, Purchaser product design or any incompatibility of the Product with Purchaser product.

Products supplied by VBMB are not designed, intended, or authorized for use in life support, life sustaining, medical systems or devices, aircraft navigation, nuclear, or other applications, including, but not limited to, public transportation operating systems, in which the failure of such Products could reasonably be expected to result in personal injury, loss of life or severe property or environmental damage. Purchaser acknowledges that use of VBMB's Products in such product applications is understood to be fully at the risk of Purchaser and that Purchaser is responsible for verification and validation of the suitability of VBMB's Products in such applications. Purchaser agrees that VBMB is not and shall not be liable, in whole or in part, for any claim or damage arising from use in such applications. Purchaser agrees to indemnify, defend, and hold VBMB harmless from and against all claims, damages, losses, costs, expenses, and liabilities arising out of or in connection with any such use or application.

VBMB retains all rights to all proprietary intellectual property in the Products and associated manufacturing processes and has the right to file for and obtain intellectual property protection for same.