VESA®

Display Data Channel Command Interface (DDC/CI) Standard

Video Electronics Standards Association

920 Hillview Court, Suite 140
Milpitas, CA 95035

Phone: (408) 957-9270
Fax: (408) 957-9277

DISPLAY DATA CHANNEL COMMAND INTERFACE (DDC/CI) STANDARD (formerly DDC2Bi)

Version 1
Adoption Date : August 14, 1998

Purpose

Define I^2C based protocols with various levels of complexity which operate over the DDC channel for the purpose of controlling the monitor and optional annex devices.

Summary

In response to the Plug and Play needs by end-users, VESA has defined the DDC standard, made of different levels of communication. DDC2Bi, DDC2B+ and DDC2AB levels offer bi-directional communication between the computer graphic host and the display device. This standard describes and compares each display control interface.

Notes

All of these serial communications are independent of display technology (CRT, LCD, PDP), and are compatible with different video interfaces (VGA, P&D, EVC, FPDI)

[This page left intentionally blank]

PREFACE

Scope

This revision of the DDC/CI Standard is intended to extend the current DDC standard and provide flexibility and expandability to DDC, MCCS and P&D standards.

Intellectual Property

Trademarks

Patents

VESA proposals and standards are adopted by the Video Electronics Standards Association without regard to whether their adoption may involve any patents on articles, materials, or processes. Such adoption does not assume any liability to any patent owner, nor does it assume any obligation whatever to parties adopting the proposals or standards documents.

Support for this standard

Clarifications and application notes to support this standard may be written. To obtain the latest standard and any support documentation, contact VESA.

If you have a product which incorporates DDC/CI, you should ask the company that manufactured your product for assistance. If you are a manufacturer, VESA can assist you with any clarification you may require. All comments or reported errors should be submitted in writing to VESA using one of the following methods.

- Fax 408-957-9277, *direct this note to Technical Support at VESA*
- e-mail support@vesa.org (with subject: [monitor-DDC/CI])
- mail to Technical Support
  VESA - Video Electronics Standards Association
  920 Hillview Court, Suite 140
  Milpitas, CA 95035

Acknowledgments

This document would not have been possible without the efforts of the VESA Monitor Committee. In particular, the following individuals and their companies contributed significant time and knowledge.

Name	Company	Contribution
D. Balthaser	STB Systems	Feedback
K. Bennett	SciTech	SSC Committee, VBE SCI spec
T. Block	Number 9	Graphics Card Host (H/W)
R. Cyr	Number 9	Graphics Card Host (S/W)
J. Frederick	Compaq	PC Home Theater and CEMA
O. Garreau	Siemens Semiconductor	Feedback
F. Grilli	SGS-Thomson	Feedback
W. Johnson	Diamond Multimedia	Graphics Card Host (S/W)
D. Loucks	EloTouch	Touch Screen Device
M. Marentic	Hitachi	Application to FPD
S. Marsanne	SGS-Thomson	System concept, PC S/W Tools
B. Milford	STB Systems	Feedback
Y. Narui	Sony Corp.	Feedback
A. Pakkala	Planar	Feedback (FPD)
M. Phillips	Panasonic	Display Device
A. Morrish	National Semiconductor	Feedback (GTF)
T. Sasaki	Panasonic	Display Device
C. Scott	Microsoft	Feedback (O/S)
M. Shiota	Panasonic	Display Device
A-L. Sixou	SGS-Thomson	Implementation
H. Tanizoe	Mitsubishi	Feedback
O. Tomita	Toshiba	Feedback (FPD)
D. Virag	Thomson Multimedia	Feedback, CEMA
R. J. Visser	Philips	MCCS workgroup chair

Terms and Abbreviations

Term or Abbreviation	Description
ABIG	Access Bus Industry Group
CC	[TV] Close Caption
CEMA	Consumer Electronics Manufacturer Association (www.cemacity.org)
CPU	Central Processor Unit (computer)
CRT	Cathode Ray Tube (display type)
DCI	[VESA] Display Control Interface (serial communication)
DDC	[VESA] Display Data Channel (serial communication)
DDC2B	Simplest of the DDC2B modes defined in VESA DDC standard
DDC2Bi	This current proposal to upgrade the functionality of existing DDC2B standard
DDC2B+	Adds bi-directional communication to DDC2B
DDC2AB	An Access Bus Mode defined in DDC Standard
DLL	Dynamic Linked Library (Windows S/W programming)
DPMS	[VESA] Display Power Management Signaling standard
EDID	[VESA] Extended Display Identification Data
EDS	[TV] Extended Data System
EEPROM	Electrically Erasable Programmable Read Only Memory (memory type)
FPD	Flat Panel Display (display type)
F/W	Firmware (the whole MCU program embedded in an application)
GTF	[VESA] General Timing Formula
H/W	Hardware
I^2C	Trademark of Philips, Inter Integrated Circuit Bus
IEC	International Electrotechnical Commission
ISO	International Organization for Standardization
LCD	Liquid Crystal Display
MCI	Monitor Command Interface (Workgroup)
MCCS	Monitor Control Command Set
MCU	Micro Controller Unit (Embedded in application)
MT	I^2C Bus Master Transmitter Communication Mode
MR	I^2C Bus Master Receiver Communication Mode
OSD	On Screen Display
P&D	[VESA] Video Plug and Display Standard
RAM	Random Access Memory
ROM	Read Only Memory
SR	I^2C Slave Receiver Communication Mode
ST	I^2C Slave Transmitter Communication Mode
SSC	[VESA] Software Standard Committee
S/W	Software
USB	Universal Serial Bus (serial communication)
VBE-SCI	Video Bios Extension-Serial Communication Interface
VCP	Virtual Control Panel (Access bus)
VDIF	[VESA] Video Display Identification Format
VESA	Video Electronics Standards Association

I^2C Bus notation: (case sensitive)

Code	Description	Comment
S	Start bit	Generated by the master to start communication (Bus becomes BUSY)
XX	Data byte, hexadecimal	Made of 8 data bits, may be sent or received by the master
a	Acknowledge bit	This bit is generated in the opposite way than the data bits
n	Non acknowledge bit	Signals the end of the data transfer, a stop bit should follow to free the bus
P	Stop bit	Signals the end of the communication, the bus becomes free.

Table of Contents

1. OVERVIEW

1.1 Summary

All the Display Data Channel Levels described in this document allow the display to interact with its graphic host. They all have provisions to support existing and future EDID, DDC, VBE-SCI & MCCS standards.

1.2 Background

DDC2B capable graphic hosts have limited and monodirectionnal dialog capabilities with the display device. DDC2Bi, 2B+ and 2AB offer similar display control interfaces based on I^2C bus:

- DDC2Bi is a S/W upgrade of DDC2B graphic hosts using I^2C single master communication.
- DDC2B+ is an upgrade of DDC2B graphic hosts using I^2C master/slave communication.
- DDC2AB is based on Access Bus standard and is an I^2C multi-master communication.

1.3 Standard Objectives

DDC2Bi was developed by VESA to meet, exceed and / or complement certain criteria. These criteria are set forth as Standard Objectives as follows :

- Provide display controls using the DDC2B H/W standard, making DDC2Bi Displays compatible with existing and pervasive DDC2B compliant graphic hosts.
- Support Microsoft® Plug and Play definition.
- Compatible with existing DDC levels.
- Ensure scaleable, low cost, fast market acceptance and implementation.
- Provide information in a compact and scaleable format to allow the graphic sub-system to be configured based on the capabilities of the attached display.
- Provide for communication between the graphic host and other display dependent devices.
- Provide for integration of display dependent devices in the display device.
- Scaleable to Flat Panel Display Interfaces.

1.4 References Documents

Note : Versions identified here are current, but users of this standard are advised to ensure they have the latest versions of referenced standards and documents.

- VESA, Display Data Channel Standard, V 3.6p, September 97
- VESA, Monitor Control Command Set, V 1.0p, September 97
- Access Bus Specification V 3.0, Sept 95
- VESA, Extended Display Identification Data, V 2.1, September 97
- VESA, Plug & Display Standard, V 1.0, June 97
- VESA, Video BIOS Extensions For Display Data Channel - VBE/SCI - Standard.
- VESA, Video Image Area Definition Standard, Revision 1.0, August 12th 1993
- Microsoft / Intel Plug and Play ISA Specification, Version 1.0, May 28th 1993.
- Microsoft / Intel Plug and Play Errata and Clarification Document, 12/10/93.

This table summarizes the DDC level upgrade requirements for both the Graphic Host and the Display Device.

From	To	Graphic Host H/W upgrade	Graphic Host S/W upgrade	Display Monitor H/W upgrade	Display Monitor S/W upgrade
DDC1	DDC2B	I^2C Bus Single Master (2 I/Os)	BIOS	I^2C Slave address A0/A1 support	DDC2B driver
DDC2B	DDC2Bi	No upgrade	DDC.DLL driver	6E/6F Slave address support	DDC2Bi driver (Simplified Access Bus)
DDC2B	DDC2B+	50/51 Slave address support	Access Bus Host driver (single device)	6E/6F Slave address support	Access Bus driver
DDC2B	DDC2AB	50/51 Slave address support	Access Bus Host driver (full spec)	6E/6F Slave address support	Access Bus driver

2. DDC2Bi System Architecture

2.1 DDC2Bi Introduction

This protocol relies on the DDC2B H/W definition and the Access Bus messages protocol.
The graphic host behaves as an I^{2}C Single Master Host.
The display device behaves as an I^{2}C Slave Device.
The DDC2Bi is a modification of the Access Bus Multi-Master protocol to fit Single Master communication.

2.2 DDC2Bi Display Device

The DDC2Bi display is considered a Fixed Address Access Bus Display Device (0x6E/6F), and uses only I^{2}C Slave Mode to communicate with the graphic host.
Like DDC2B+, only one display device per video channel is supported.

2.3 DDC2Bi Graphic Host

The DDC2Bi graphic host is considered as an I^{2}C single master capable device.
The “virtual” I^{2}C slave address of the host is 0x50/51.

2.4 Display Dependent Devices

A display dependent device is geographically located around the display and follows the same DDC2Bi data protocol as the display device. Pointer, Calibration and Audio devices are example of display dependent devices.

These devices can be classified in two groups:
- External to the display device, can be attached or detached from the DDC/I^{2}C bus (“add-on” device).
- Internal to the display device (integrated, “all-in-one” device).

2.5 Fixed I^{2}C Slave Address Devices

This category of device groups all the existing stand-alone and “brainless” I^{2}C slave devices, such as memories, TV tuners, audio processors, etc. These ICs can coexist and be connected to the DDC/I^{2}C bus. However, it is strongly recommended to limit their number, and locate them in the COMPUTER. These devices are not expected to support hot-plugging nor to follow the DDC2Bi data protocol, and as such, must be considered a custom device.
No 10-bit I^{2}C bus slave devices can be present on the DDC/I^{2}C Bus.

3. DDC2Bi H/W Implementation

3.1 Display Device

The H/W requirements are similar to DDC2B capable display.

3.2 Graphic Host

H/W requirements are similar to DDC2B capable host.

3.3 Display Dependent devices

Display dependent devices are classified in two types:

3.3.1 External Display Dependent Devices:

These devices are connected to the DDC/I^{2}C Bus: As such, to avoid conflict with the Display Slave address, a fixed I^{2}C address is defined for the device. The address range is 0xF0..FF: up to 8 additional external display dependent devices can be added on the DDC bus. As such, the 10 bit I^{2}C addressing mode is NOT SUPPORTED.

Examples of External Display Dependent Devices (connected on the DDC/I^{2}C Bus:)

I^{2}C Slave Address	Display Dependent Device Type	Example
0xF0/F1	Pointer	Touch Screen, Light pen or Remote Control Track Ball
0xF2/F3	Audio Device	Speaker/Microphone
0xF4/F5	Serial Communication	Home Network IF (power line modem)
0xF6/F7	Calibration Device	Luminance Probe or Colorimeter
0xF8/F9	Input Device	IR Keyboard and Remote control pad (shared IR channel)
0xFA/FB	Reserved	Reserved for future use
0xFC/FD	Reserved	Reserved for future use
0xFE/FF	Reserved	Reserved for future use

3.3.2 Internal Display Dependent Devices:

The external device may be integrated inside the display device structure, and as such becomes part of the display H/W platform, not directly connected to the DDC/I^{2}C bus.
The device dependent function is accessed thought the display device. (See the chapter 5 for more details.)

Technical clarification: Internal and external display dependent devices can coexist without conflict.

3.4 Fixed address I^{2}C devices

These devices can be connected to the same I^{2}C/DDC bus, and must have a 7 bit I^{2}C slave address.
The I^{2}C address is defined and registered by Philips or ABIG.

Example of fixed I^{2}C address devices:

I^{2}C Slave Address	I^{2}C Device
0x12/13	Smart Battery Charger
0x14/15	Smart Battery Selector
0x16/17	Smart Battery

I^{2}C Slave Address	I^{2}C Device
0x80/81	Audio Processor
0x40/41	PAL/NTSC Encoder
0xA0/A1	DDC2B Monitor (memory)

4. DDC2Bi S/W Implementation

A DDC2Bi system follows the Access Bus 3.0 specification with the following modifications:

4.1 Graphic Host to Display Device Messages

In order to tell the display that the received message is of DDC2Bi type, the Source Address Byte bit 0 is set to 1.

Example: The graphic host wants to enable the Display Application Messages
The graphic host sends an "Enable Application Report" message:

Access Bus: Host to Display: MT to SR

Dest	Source	Length	Data/Cmd	Data	Checksum
6E	50	82	F5	01	48

```
I2C Sequence: S-6Ea-50a-82a-F5a-01a-48a-P
```

DDC2Bi: Host to Display: MT to SR

Dest	Source	Length	Data/Cmd	Data	Checksum
6E	51	82	F5	01	49

```
I2C Sequence: S-6Ea-51a-82a-F5a-01a-49a-P
```

Technical Clarification: The Access Bus spec does not use Odd Source I^{2}C addresses, thus allowing both types of communication to coexist (See spec point 2.1.4: "I^{2}C Addressing", page 2-3).

Note: Based on the DDC spec V3.2p, spec point 7.4 "Additional DDC Protocols", it is recommended for the host to start reading the EDID using the DDC2B method before using the DDC2Bi protocol to interact with the display.

4.2 Display Device to Graphic Host Messages

When the graphic host expects an answer from the display, the host READS the answer message at the display device Slave Address 0x6F. Note that the checksum is still computed by using the 0x50 virtual host address.

Example: The graphic host wants to get the Display Self-Test Report:
The graphic host sends an "Application Test" Message:

Access Bus: Host to Display, MT to SR

Dest	Source	Length	Data/Cmd	Checksum
6E	50	81	B1	0E

```
I2C Sequence: S-6Ea-50a-81a-B1a-0Ea-P
```

DDC2Bi: Host to Display, MT to SR

Dest	Source	Length	Data/Cmd	Checksum
6E	51	81	B1	0F

```
I2C Sequence: S-6Ea-51a-81a-B1a-B1a-0Fa-P
```

The "Application Test Reply" Message is read by the Host:

Access Bus: Display to Host, MT to SR

Dest	Source	Length	Data/Cmd	Data/Status	Checksum
50	6E	82	A1	00	1D

```
I2C Sequence: S-50a-6Ea-82a-A1a-00a-1Da-P
```

DDC2Bi: Display to Host, Slave Transmit to Master Receive

Dest	Source	Length	Data/Cmd	Data/Status	Checksum
6F	6E	82	A1	00	1D

```
I2C Sequence: S-6Fa-6Ea-82a-A1a-00a-1Dn-P
```

In this example, the display returns its current status [00], indicating no problem.

4.3 *Definition and use of the "Null Message"*

A NULL message can be defined as an Access Bus message without any data bytes, i.e., the "length byte" is 0x80:

DDC2Bi: Display to Host, ST to MR

Dest	Source	Length	Checksum
6F	6E	80	BE

`I`^{2}`C Sequence: S-6Fa-6Ea-80a-BEn-P`

The NULL message is used in the following cases:
- To detect that the display is DDC2Bi capable (by reading it at 0x6F I^{2}C slave address)
- To tell the host that the display does not have any answer to give to the host (not ready or not expected)
- The "Enable Application Report" has not been sent before using Application Messages

4.4 *Communication between the Host and its Devices*

Several basic rules apply to the DDC2Bi host and its devices in order to have good communication performances.

4.4.1 Communication Error Recovery

With DDC2Bi, there is no concept of retrials when a communication fails (bus error, bad checksum):
It is the responsibility of the host to re-send its message and try to get an answer from the display again.
If the communication fails, the host MUST wait 40msec and then retry at least once to communicate with the device.

4.4.2 Message Buffer Size Requirements

The host must be able to send AND receive messages of any size. The absolute maximum is 127+4=131 bytes.

It is recommended to have independent transmit and receive buffers in order to simplify the implementation of error recovery and retrial mechanism in case of failed communication.

Note: For simpler implementation, it is possible for the host to systematically try to read an answer after any sent messages to the display.

Obviously, a device must properly send and receive all its supported messages. This determines the maximum internal data communication buffer size for proper display operation.

The device must acknowledge all received data bytes from the host, even if the message is larger than the maximum size supported by the device.

If the host attempts to read more data bytes than specified by the "length byte", extra bytes of any dummy value will be read, in order to avoid a "hang" situation. However, the Host is responsible to read the right number of bytes.

Technical clarification: With DDC2Bi, it is possible to share the same memory in the device for both the receive and transmit buffers, due to the smart host communication error recovery mechanism.

Note: Since DDC2Bi does not use the Access Bus "Assign Address" message, the buffer size may be much smaller.

4.5 *I^{2}C Bus timings*

The host must implement I^{2}C bus error recovery systems (see appendix for examples).
The host must abort and perform an error recovery if the SCL line is stretched low by other devices for more than 2 msec, as specified in the Access Bus Spec 3.0 (for master systems).
Since devices are slave devices and not driving the SCL line, they do NOT need to implement a 2msec SCL Low Watchdog system, but must make sure that in a worst-case timing situation, the device does not stretch the clock low over 2 msec (i.e. MCU maximum interrupt latency). The SCL clock stretching duration should be kept as short as possible.

Technical Clarification: When the host sends a message to the display, the host must wait at least 40 msec before trying to read an answer from the display in order to avoid I^2C bus bandwidth overhead (40msec stands for the Access Bus Response Timing specification). Since the display commands are initiated by the end-user, the 40msec response latency (equivalent of several video frames or one keyboard scan debouncing period) is not critical.
(See Access Bus Spec Point 2.1.8.2 Response Timing, Page 2-9.) If other I^2C devices are on the bus, the time-interleaved message method is easier to implement on the host side.

4.6 Access Bus Messages Support

Some simplifications can be done based on DDC2Bi functionality and are described hereafter.

4.6.1 System Messages

The following Access Bus System messages are NOT required:

System Message Name	CMD
RESET	F0
ATTENTION	E0
ASSIGN ADDRESS	F2
PRESENCE CHECK	F7

All OPTIONAL Access Bus System messages are NOT required by DDC2Bi.

Note: Application Messages must be enabled by the “Enable Application Report” message.

4.6.2 Power Management

If the power management can be controlled by DDC2Bi, it must be handled by using the MCCS code, not by using the Access Bus Power Management system.
If the MCCS solution must coexist with DPMS, some guidelines are shown in Appendix of this document. However, if the host supports both MCCS and DPMS, both methods must be used by the OS to notify the display of any change in the requested power management level.

4.6.3 ID String

The ID String must be unique per device, and Vendor/Model Names must be consistent with EDID information.

4.6.4 Capability String

This table describes each capability string field and their support by DDC2Bi.
(For more details, See Access Bus Spec 3.0, 2.1.6 Capabilities Information, Page 2-5.)

Field	Comment
prot()	Mandatory
type()	Mandatory
model()	Mandatory
pwr()	Not used by DDC2Bi. If the MCCS power management is supported, the VCP code MUST be present in the vcp() field
edid()	Mandatory. Can be of any version/revision because its size is scaleable using “bin xxx()” keyword. The Capability String EDID can be different from DDC2B EDID, but MUST share the SAME common information (except for Version & Revision Fields)
vdif()	Optional use in DDC2Bi. Its data must be consistent with EDID() information
cmds()	Mandatory.
vcp()	Mandatory. If MCCS power management is supported, the corresponding VCP code must be put in.
vcpname()	Used to define specific additional VCP code not referenced in MCCS standard. All codes not described in the MCCS specification should be described using this field. (See notes)
custom fields	Additional vendor specific fields can be added in the capability string. The keyword must be compatible with the Access Bus Capability String Syntax The keyword shall be approved by VESA in order to avoid conflict with other vendors.

Important: Generic PC host S/W must discard any unsupported capability fields by default.

Note: vcpname() is a special field to define some control codes that are not fully defined in the Access Bus or MCCS specification. Some non-MCCS codes described here are DDC2Bi specific.
DDC2Bi devices must use the vcpname() field to define user accessible vendor specific controls. Such specific controls must be notified and approved by VESA in order to avoid code conflict between different vendors. This is for the benefit of the end-user in addition to the cohesion of the system and its interoperability.

Non-MCCS control codes (but defined in Access Bus specification) and support:

VCP Code	Function	Standard
0x14	Select Color Preset	All temperatures must be put using vcpname(). The order to enumerate them must be the same as in the EDID. See appendix.
0x62	Audio Speaker Volume	Still acceptible and valid to this method; however, it is recommended to use the “Audio” internal display dependent device.
0x64	Audio Microphone Volume	same remark as above
0x66	On Screen Display Enable/Disable	Used to enable/disable the OSD display function
0x68	OSD Language Select	vcpname() should enumerate the list of the possible language selections, using the three-letter country-name codes defined by the ISO/IEC specification 3166

4.6.5 Vendor Specific Messages

Also, as defined in the Access Bus specification (2.1.10.1 Command coding, Page 2-13):
- “Data Stream messages” are vendor specific (“Protocol Flag” bit 7 of “Length byte” cleared)
- All “control/status messages” (except 0xC0-C8 vendor specific) are reserved for future versions of DDC2Bi

Technical Recommendation: If the vendor uses a different serial communication channel (i.e. RS232) to set-up the display during production, it is suggested to implement an I^{2}C alternate method (using for example 0xC0/C1 commands), where the factory messages are encapsulated.
Both communication channels can co-exist at the same time for backward compatibility.

Example: RS232 is used in factory, and the adjustment machine wants to get the current Contrast value:

“GetContrast Message”, host to display (assume a 4 byte message)

XX	XX	XX	XX

Then, the display returns the contrast value to the PC: (3 byte message)

YY	YY	YY

These messages can be encapsulated in DDC2Bi as follows:

Factory message from Host to Display (code C0)

Dest	Source	Length	Data/Cmd	Data	Data	Data	Data	Checksum
6E	51	85	C0	XX	XX	XX	XX	CHK

```
I2C Sequence: S-6Ea-51a-85a-C0a-XXa-XXa-XXa-XXa-CHKa-P
```

Note: CHK is the Access Bus checksum

Factory message read by Host from Display: (code C1)

Dest	Source	Length	Data/Cmd	Data	Data	Data	Checksum
6F	51	84	C1	YY	YY	YY	CHK’

```
I2C Sequence: S-6Fa-51a-84a-C1a-YYa-YYa-CHK’n-P
```

The advantage of this solution is that it offers standard factory communication using existing DDC2B H/W (which is supported by most existing and future systems) without the need for specific after-service communication boxes.

Furthermore, it may eliminate the removal of the monitor's plastic cover, and the I^2C Bus 100kbps generally offers higher communication speed than most current factory serial communication.

Technical detail: Access Bus Vendor Command codes (0xC0..C8) must NOT be declared in the cmd() field of the Capability String.

It is recommended to add a "Write EDID" vendor command message in the display since this method is more secure than using the "VSYNC signal write protect" method.

4.6.6 Application Specific Messages

The display does not have to initiate any message by itself:
- "Timing Report" message is sent only after the host sends "Get Timing Report" message.
- "Key Report" message is sent only after the host sends "Get Key Report" message.

MCCS Cross Reference Table

MCCS Host Message	Access Bus Message(s)	Comment
Get_Display_Status	Get Timing Report, Timing Report	The MCCS Message format (6 data bytes) is recommended; however, old Access Bus Timing Report message (5 bytes) is valid implementation
Get_Supported Controls	Capability String	
Get_Max	Get_VCP, VCP_Reply	
Get_Current	Get_VCP, VCP_Reply	
Set_Current	Set_VCP	
Get_Possible	Capability (vcpname)	Abbreviations for capability string to be defined

Note: It is useful to implement both the Keyboard and OSD Enable/Disable commands so that the host can automatically disable them when the control application is active.

4.6.7 Hot Plugging mechanism

DDC2Bi supports hot plugging, provided the display can detect a disconnection of the video cable. When the display detects an "unplug" event, it resets its DDC2Bi function and disables the Application Message Reports.

The host should regularly poll the device ID String to check for device presence (i.e. every 6 sec).

If the DDC2Bi slave address is not acknowledged after "trial and error recovery" attempts, the host must consider that the DDC2Bi function is no longer available (detached).

If the DDC2Bi slave address is acknowledged but a NULL message is returned in place of a application reply message, the host must consider that a new DDC2Bi device has been attached.

Anytime a DDC2Bi device is attached, the host should get the both the ID String and Capability String, then enable the Application Message Report.

Note: It is possible for the host to reduce I^2C bus traffic by saving (i.e. Registry) a string table containing capability strings, indexed by the ID String (unique for each display). As such, when a registered display is hot plugged, the host reads the capability string from the registry without having to use the I^2C Bus.

5. DDC2Bi Support of Display Dependent Devices

The DDC2Bi communication allows optional addition of display dependent devices.
As an example, a touch screen device is used in this chapter.

There are 2 different ways to implement the touch screen:
- As a slave device using the DDC2Bi protocol, but located at a different I^2C slave address (e.g. 0xF0/F1)
- As integrated/embedded in the display device H/W architecture (unique I^2C address for both: 0x6E/6F)

Note: In a multiple display configuration with touch screen devices, it is necessary to know which touch-screen matches which display. This is possible with DDC2Bi because there is a unique DDC bus per video channel.

5.1 External Display Dependent Device

In this example, the implementation is simple: a fixed I^2C Slave address is defined for the touch screen device (0xF0/F1). The application specific touch screen commands are defined in the Access Bus Locator Device Protocol spec.

5.1.1 Message sent to the External Device

The “source byte” (0x51) is replaced by the odd device address (i.e. 0xF1)

5.1.2 Message replied from the External Device

The “source byte” (0x6E) is logically replaced by the even device address (i.e. 0xF0)

5.2 Internal Display Dependent Device

When the touch screen is integrated in the display device, there is only one I^2C Slave address (0x6E/6F) shared by both the display and the touch screen. In this configuration, the message discrimination/routing is done as follows:

5.2.1 Message sent to the Internal Device

The “source byte” (0x51) is replaced by the external device odd address (i.e. 0xF1).

5.2.2 Message replied from the Internal Device

The “source byte” (0x6E) address byte of the Access Bus message is 0xF0.
Except for the destination I^2C address, the communication of a display dependent device is the SAME for both internal and external devices.

5.3 Detection of Display Dependent Device

The device detection is done by attempting to access the external I^2C address first (acknowledge).
Then the host must detect the presence of an internal device by sending an “Identification Request” to the internal device and check if the “Identification Reply” is successful. If not, a NULL message will be returned, meaning no internal devices are present.

5.4 Example of Internal and External Device Communication

We will consider both situations where the touch screen is external or internal:

Example: The graphic host wants to get the Touch Screen Self-Test Report:
The graphic host sends an "Application Test" Message:

External: Host to Touch Screen

Dest	Source	Length	Data/Cmd	Checksum
F0	F1	81	B1	31

`I²C Sequence: S-F0a-F1a-81a-B1a-31a-P`

Internal: Host to Touch Screen

Dest	Source	Length	Data/Cmd	Checksum
6E	F1	81	B1	AF

`I²C Sequence: S-6Ea-F1a-81a-B1a-AFa-P`

The "Application Test Reply" Message is read by the Host:

External: Touch Screen to Host

Dest	Source	Length	Data/Cmd	Data/Status	Checksum
F1	F0	82	A1	00	83

`I²C Sequence: S-F1a-F0a-82a-A1a-00a-83n-P`

Internal: Touch Screen to Host

Dest	Source	Length	Data/Cmd	Data/Status	Checksum
6F	F0	82	A1	00	83

`I²C Sequence: S-6Fa-F0a-82a-A1a-00a-83n-P`

In this example, the touch screen returns its current status [00], indicating no problem.

5.5 Dependencies between the Display and Integrated devices

Some commands sent to the display, such as those concering power management, may affect integrated devices.

Technical Clarification: When the host communicates with an internal device, the host MUST read any expected answer from the internal device before attempting to communicate with the display. This allows for cost optimized implementations, where both the display device and the internal device are sharing the same communication buffer. (RAM optimization on Display MCU)

6. DDC2Bi System Architecture

This chapter provides some examples of DDC2Bi system implementations.

6.1 Multiple Video Channel Support and Implementation

A unique DDC I^2C bus for each video channel MUST be implemented.

6.2 Television/Home Theater Support and Specific Commands

Specific dedicated functions exist in the MCCS specification.

6.3 Video Switch Boxes

Video switch boxes allow a single display to be attached to more than one computer system. System designers should assume that the video channel is linked with the DDC bus, meaning that switching the video channel from one host to another will also apply to the DDC channel.

6.4 Multiple Video Output Expander Boxes

Expander boxes are used to generate multiple video outputs from a single source. They may:
- Send the same video to multiple displays (school or conference rooms). This is a video “duplicator” system..
- Split the video to make a 3x3 video wall (using 9 displays). This is a video “splitter” system..

The expander box should primarily behave as a single display for the benefit of the host application S/W.

In both examples, it is good design practice to put a DDC2Bi “hub” that will have one DDC channel going to the PC and a separate independent I^2C bus (I^2C master) going to each attached display. This hub could build the correct EDID and properly translate and dispatch the DDC2Bi display control commands to each attached display device.

If the expander box supports both “duplicator” and “splitter” modes, the modes can be controlled by the host using a custom VCP code (using vcpname keyword in the capability string).

For instance, in splitter mode, DDC2Bi can be used to address each display individually by considering them as “internal only” display dependant devices. In the example of the 3x3 wall display, this could be done by using free addresses (“source byte”) 00/01, 02/03, 04/05 for the top row, 06/07, 08/09, 0A/0B for the middle, and 0C/0D, 0E/0F, 10/11 for the bottom.

For the duplicator mode, the previous enumeration could correspond to the video output channel number of the expander box.

Note: Since the expander box is relaying messages to its attached displays, the response time must be increased from 40msec to 120msec.

6.5 Video Projection Displays

Video projectors are display devices and can support DDC2Bi.
If a secondary monitor display can be connected to the video projector, a number of implementations are possible and can vary in complexity. One solution would be to consider the video projector as a multiple video expander box.

7. DDC2Bi Compliance

Compliance with the VESA DDC2Bi Standard requires that the requirements of all sections are met.

7.1 Existing Display Designs

Existing display designs (any DDC layer) will not conflict with a DDC2Bi capable host.

7.2 New Display Designs

DDC2Bi does not interfere with any other DDC communication layer.
DDC2B+ or 2AB displays can be easily modified to support DDC2Bi, but new designs need only support the DDC1/2B/2Bi layers for cost reduction/simplification.

However, it is recommended to put a 12 to 15 kOhm pull-up resistor on both SDA and SCL signal lines in order to achieve optimum communication speed in a noisy environment, or when a VGA expander cable is used.

Technical clarification: Placing a 12k Ohm resistor in parallel sharpens the rising edges at the display side without significantly affecting the equivalent resistor on the bus or the maximum sinking current, thus allowing for optimum communication.

7.3 Existing Graphic Host Systems

Existing host systems (any DDC layers) are not in conflict with DDC2Bi capable displays.

7.4 New Graphic Host Systems

New host designs shall not use less than 2.2k Ohm (5%) pull-up resistor on both SDA and SCL lines.

8. DDC2Bi Flat Panel Displays

Flat panel displays can implement DDC2Bi regardless of their configuration and system architecture, since the DDC2Bi I^{2}C slave address is the same and unique per video channel: It is independent of the cable or physical connector interface.

8.1 EDID Support

The EDID data (regardless of its version, revision and size) may be extracted from the "capability string".

8.2 Specific Control Support

DDC2Bi supports flat panel specific controls such as backlight level, video inversion, power management, luminance adjustments, power controls (currently as vendor specific VCP code), etc.

8.3 Power Management Support

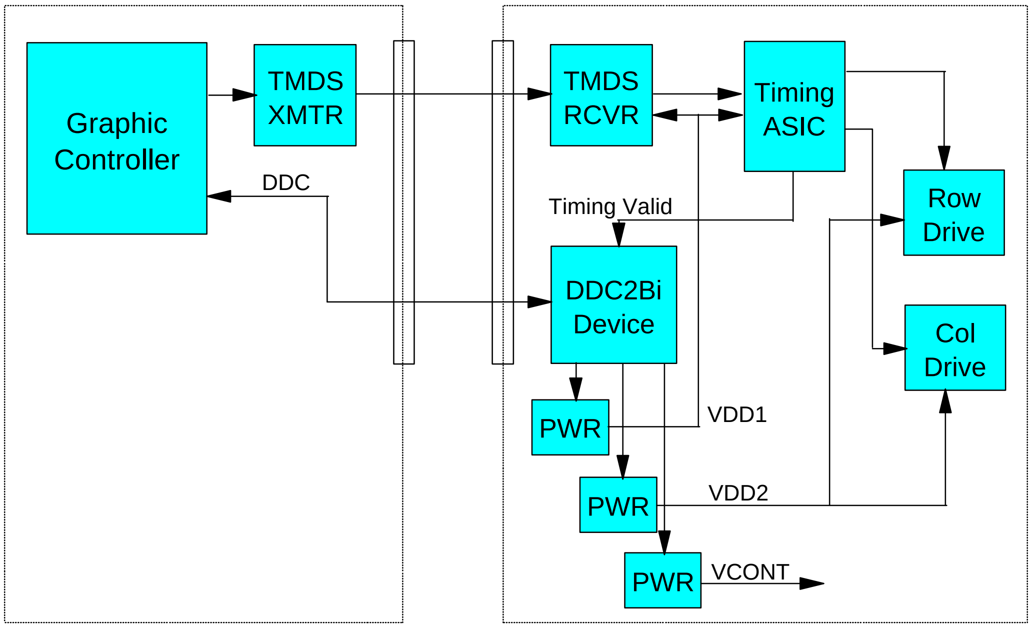
Some flat panel displays are critically sensitive to hot plugging and power cycling sequences. Using DDC2Bi, the power management control is handled by the MCCS dedicated display control code. The power sequencing and timings become transparent to the computer/graphic system, since they are completely handled by the display. Also, since DDC2Bi requires an intelligent device on the flat panel board, some S/W protections (for safety) can be implemented.

Flat Panel Specific Code:

VCP Code	Description	Comments
0xF6	Back light Power Level	00 means that the back light is deactivated

8.4 FPDI-2 and P&D Interfaces

Using DDC2Bi, the P&D interface can be built from FPDI-2 Main 20-Pin interface without requiring the additional 8-pin connector, thus giving more flexibility and easier implementation.



Backward compatibility is technically possible. This solution makes the flat panel display an abstract device, and can be considered as a normal display. No technology specific signal is used between the graphic host and the flat panel display. Display dependent devices (i.e. touch screen) and backlight controls can be added to the DDC2Bi interface. In the example of an external monitor connected to a laptop, the same EDID and I^{2}C address can be used for both the monitor and the laptop display, since each one has its own specific DDC/I^{2}C bus.

DDC2B+ Display Control Interface Level

9. DDC2B+ System Architecture

9.1 DDC2B+ Introduction

This protocol relies fully on the Access Bus Specification, but with support for only one Access Bus Device (i.e. display device) on the bus (master slave communication). This can be performed by either a simplified H/W implementation or an intensive S/W polling solution on the graphic controller host.

9.2 DDC2B+ Display Device

The requirements for the DDC2B+ display are the same as DDC2AB.

9.3 DDC2B+ Graphic Host

The DDC2B+ graphic host is considered as an I^{2}C master-slave capable device following the Access Bus spec.
The I^{2}C slave address of the host is 0x50/51.
The host must only support I^{2}C master/slave comunication which means that only one Access Bus device (i.e. the display device) is supported.

9.4 Fixed I^{2}C Slave Address Devices

All I^{2}C Fixed I^{2}C Slave Address devices are defined and reserved by Access Bus Industry Group (ABIG).

10. DDC2B+ H/W Implementation

10.1 Display Device

The H/W requirements are similar to DDC2AB capable displays.

10.2 Graphic Host

The requirements are similar to a DDC2AB capable host.

11. DDC2Bi S/W Implementation

A DDC2B+ system must comply with the Access Bus 3.0 specification.

12. DDC2B+ Compliance

Compliance with the VESA DDC2B+ Standard requires that the requirements of all sections are met.

DDC2AB Display Control Interface Level

13. DDC2AB System Architecture

13.1 DDC2AB Introduction

This protocol relies fully on the Access Bus specification. This allows for the support of multiple Access Bus devices such as keyboards, pointer devices, etc.

13.2 DDC2AB Display Device

The requirements are defined in the Access Bus specification 3.0.

13.3 DDC2AB Graphic Host

The DDC2AB graphic host is considered as an I^2C multi-master/slave capable device following the Access Bus spec.
The I^2C slave address of the host is 0x50/51.

13.4 Fixed I^2C Slave Address Devices

All I^2C Fixed I^2C Slave Address Devices are defined and reserved by ABIG.

14. DDC2AB H/W Implementation

14.1 Display Device

The H/W requirements are similar to DDC2AB capable displays.

14.2 Graphic Host

The requirements are described in the Access Bus 3.0 specification.

15. DDC2AB S/W Implementation

A DDC2AB system must comply with the Access Bus 3.0 specification.

16. DDC2AB Compliance

Compliance with the VESA DDC2AB Standard requires Access Bus compliance.

17. APPENDIX A - DDC2Bi Development Support Tools

Note: Appendixes are NOT part of the standard.

This appendix describes the available tools for developing, testing, and debugging DDC2Bi systems.

17.1 Display Devices S/W Implementation

The implementation of DDC2Bi on a DDC2B+ capable display is very simple, as the exchanged messages between the graphic host and the display are very similar. Typical memory increase in the display DDC MCU is approximately 50 ROM bytes, and 1 RAM bit (detection flag of the 0x51 “source byte”). Also, if the Display supports only DDC2Bi by simplifying the DDC2B+ MCU F/W code, the required memory will be significantly less.

Flowcharts, C source code and application notes are available from the VESA ftp site, as well as some Windows tools (below) for testing and debugging the DDC2Bi display function.

DCP.EXE:	Display Control Panel is a DDC2Bi debugging tool
PND.EXE:	Plug and Display is an end-user tool to demonstrate the DDC2Bi function

17.2 Graphic Host S/W Implementation

DDC2Bi operation requires the graphic host to be DDC2B capable.

The current S/W implementation on existing computers is using a DLL that gives access to the I^2C bus I/O port lines of a graphic controller, using 4 simple functions: GetSDA(), SetSDA(), GetSCL(), SetSCL().
(The final DLL specification is under development and linked with the VESA SSC committee. In future host designs, the DLL will use some low level BIOS function as described in the VBE-SCI specification.)
DDC2Bi capable monitors, sample Windows S/W and source code are all available for demonstration/verification purposes.

17.3 Existing DDC2Bi Graphic Systems

The following list is for guidance/information only. ALL DDC2B COMPLIANT GRAPHICS CARDS ARE DDC2BI CAPABLE.
The following graphics cards are compatible with the P&D and DCP Windows Demonstration Tools

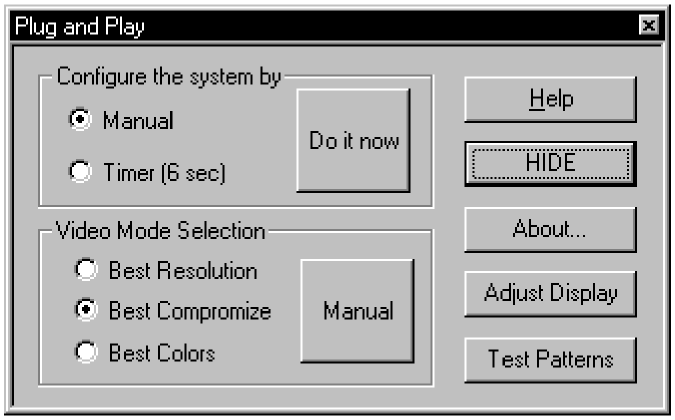
DDC2Bi Hosts	Status	Contact
Number 9 Reality 332, PCI	Available	Computer shops, VESA Rep.
“All Path” S3D Virge, PCI	Available	Computer shops
Toshiba Brezza 133 (S3D Virge)	Available	Shops (T-Zone, Comp USA)
Diamond Viper 330 (nVidia Riva 128)	Available	Shops (Fry’s Electronics), VESA Rep.
Most “S3D Virge On Board” Graphics Cards	Available	Computer shops

17.4 Existing DDC2Bi Displays

This list is for guidance/information only.
DDC2Bi displays tested and compatible with P&D and DCP Windows Demonstration Tools

DDC2Bi Displays	Status	Contact
Panasonic 17” (HV9)	Available	VESA Representative

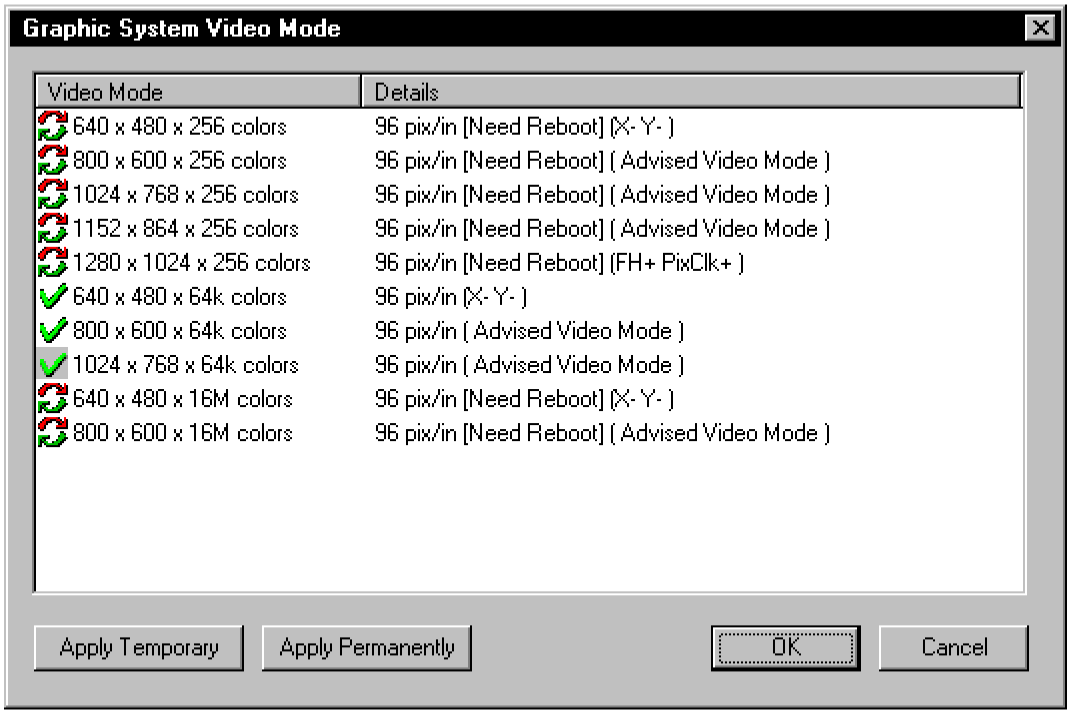
This is an overview of the Plug and Display (P&D) user interface (available at VESA FTP server). This graphic interface is for illustration and example purposes only, and is not part of the standard.



Configure the system by
[✓ Manual] Timer (6 sec)

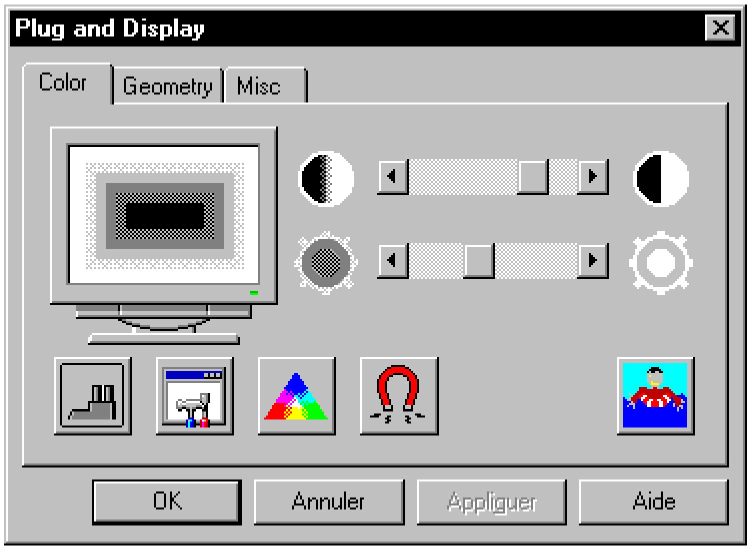
Video Mode Selection
Best Resolution [✓ Best Compromize] Best Colors

P&D Tool main user interface

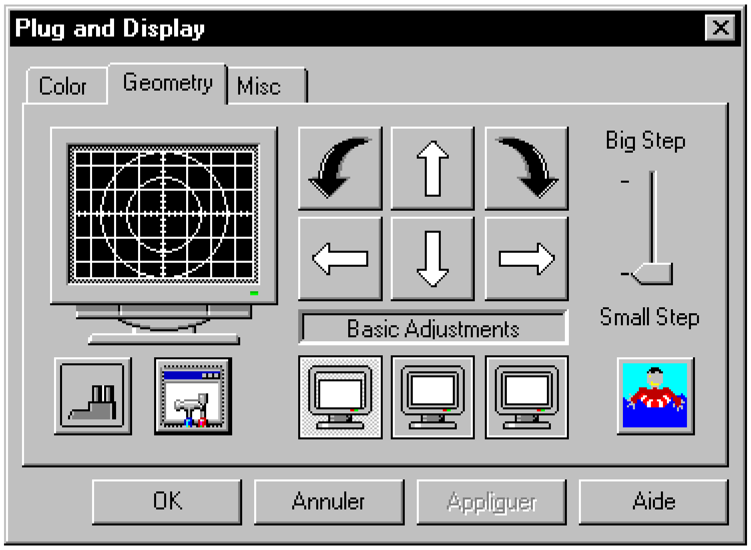


Video Mode	Details
🔄 640 x 480 x 256 colors	96 pix/in [Need Reboot] (X- Y-)
🔄 800 x 600 x 256 colors	96 pix/in [Need Reboot] (Advised Video Mode)
🔄 1024 x 768 x 256 colors	96 pix/in [Need Reboot] (Advised Video Mode)
🔄 1152 x 864 x 256 colors	96 pix/in [Need Reboot] (Advised Video Mode)
🔄 1280 x 1024 x 256 colors	96 pix/in [Need Reboot] (FH+ PixClk+)
✓ 640 x 480 x 64k colors	96 pix/in (X- Y-)
✓ 800 x 600 x 64k colors	96 pix/in (Advised Video Mode)
✓ 1024 x 768 x 64k colors	96 pix/in (Advised Video Mode)
🔄 640 x 480 x 16M colors	96 pix/in [Need Reboot] (X- Y-)
🔄 800 x 600 x 16M colors	96 pix/in [Need Reboot] (Advised Video Mode)

P&D Tool, demonstration of the Automatic Video Mode Resolution Algorithm, using EDID data



P&D Tool, color adjustment main control panel



P&D Tool, geometry adjustment control panel

This application was used for the DDC2Bi proof of concept on real systems. The low level interface source code is enclosed in this document.

18. APPENDIX B - Color Adjustments

Note: Appendixes are NOT part of the standard.

There is currently an on-going discussion regarding the implementation of color matching functionality on the computer. However, the performance of such a system depends on its flexibility. Below is an example using MCCS codes and EDID as a possible solution.

Upon completion of this study, this appendix may be merged into a future DDC-CI revision.

EDID: The EDID has a provision to describe 3 White Points (W0xy, W1xy, W2xy)

Assume: W0[0.3, 0.4]
W1[0.5, 0.6]
W2[0.7, 0.8]

MCCS: The Graphic host has a provision to control the Display Device color adjustments using the MCCS codes.
As an example, we will consider the R, G, B Gain controls.
Using GetVCP() and SetVCP(), we can get the RGB gains for each white point.

Wx (R gain, G gain, B Gain)
W0 (10, 20, 30)
W1 (40, 50, 60)
W2 (70, 80, 90)

DDC2Bi: Using the Capability String “vcpname(xx(9300,6500,5000))”, we can get the corresponding 3 White Point Temperatures.

Based on this data, the graphic host can deduce the following:

W0[9300] = (0.3, 0.4) = (10, 20, 30)
W1[6500] = (0.5, 0.6) = (40, 50, 60)
W2[5000] = (0.7, 0.8) = (70, 80, 90)

We can use polynomial interpolation using the temperature as reference.

(The P&D.exe Windows 95 utility implements such interpolation and can be used for investigation.)

Advantages:
- The end-user can directly select the white temperature
- The corresponding gains are automatically adjusted
- The corresponding (x,y) coordinates are known (interpolated)
- The user white temperatures can be saved in the Registry (user preference).
- Using the other EDID RGB chromacity coordinates, it is possible to interpolate any point on the color space.

Web Pointers to learn more about colors:

http://www.inforamp.net/~poynton/Poynton-colour.html
http://cctpwww.cityu.edu.hk/public/graphics/g3_display.htm http://www.yarc.com/colortut.htm

19. APPENDIX C - New Commands and VCP support

Note: Appendixes are NOT part of the standard.

This section gives suggestions of additional commands that MAY be part of the specification in future.

TV specific commands that may be supported in future:
TV tuner control
TV close caption and EDS data sent to the graphic host
TV remote control feedback to the PC when connected

GTF alternate timing values return and selection (For optimum use of the display capability)

Video timing notification from host to display device (and configuration procedure using serial communication)

Gamma values adjustments (theory)

20. APPENDIX D - DMPS and MCCS Power Management Handling

Note: Appendixes are NOT part of the standard.

While DPMS and MCCS can coexist on the display side, conflict can result if it is not handled properly by the host. For best results, if the host supports both power management systems, then both should be used together, as follows:

In a display supporting both DPMS and MCCS, the **suggested** rules are:

A. The GetVCP(power management) always returns the current physical monitor power state.
B. By default, the DPMS solution is used.
C. The MCCS solution is used over DPMS once the host has sent "Enable Application Report" message,
AND has set the power management level using the SetVCP() command message.
D. When the video cable is DISCONNECTED (e.g. sensing VGA Pin 9), DPMS is used.
If the "Disable Application Report" message is received by the display, DMPS is used.

This Appendix may be part of the specification in future version once the concept is fully validated.

The implementation in the Graphic Host driver is possible (ACPI-On Now Power Management) using:

```
DWORD __cdecl GetMonitorPowerStateCaps(DEVNODE devnode)

          Parameters:
                    None.

          Returns:
                    Bitmask of:
                    CM_POWERSTATE_D0
                    CM_POWERSTATE_D1
                    CM_POWERSTATE_D2
                    CM_POWERSTATE_D3
```

For more information, see:

http://www.microsoft.com/hwdev/pcfuture/ondisp.htm

Note that DDC2Bi can control the backlight using the DDC channel, as required for LCD screens.
(See "OnNow" spec: http://www.microsoft.com/hwdev/ONNOW.HTM#pmSPECS)

When the display uses the DPMS mode by default, using "GetVCP(0xD6)" will then report the Display Current power management level, regardless of the power management request's source. As such, if the monitor enters into a safety mode (X-Ray protect, for example), Power Off mode will automatically follow. If the host then sends GetVCP(0xD6), the display will naturally respond 00.
Zero power and host wake-up by a device: If the host is in stand-by mode, a device could wake-up the host (interrupt) by sending the Start/Stop bit sequence on the I^2C bus (currently under investigation)

21. APPENDIX E - Answers To Commonly Asked Questions

Note: Appendixes are NOT part of the standard.
Dxx, Gxx, Sxx = Display Device, Graphic Host, System Related Questions

Ref. #	Question	Answer
D01	Current displays cannot return from DDC2AB+ back to DDC2B mode. What about DDC2Bi?	DDC2Bi is identical to DDC2B+ in this regard. However, if the display is capable of decoding both A0/A1 and 6E/6F addresses, such a problem does not exist for DDC2Bi and DDC2AB+.
S01	Does DDC2Bi support hot plugging?	Yes, by regularly checking the device ID or I^{2}C Address acknowledge. This is the classic method.
D02	Does DDC2Bi requires additional slave address decoding in the Display H/W?	No, the 6E/6F address is the same as DDC2B+, except that the 6F address had been previously undefined.
G01	What are the main advantages of DDC2Bi compared to DDC2B+ ?	DDC2B+ requires an intensive CPU S/W polling method to emulate host slave addressing (with interrupts disabled) and can disturb other computer functions. DDC2Bi uses the I^{2}C single master function (like DDC2B) and does not require additional CPU bandwidth. Furthermore, no interrupts are suspended during the communication.
S02	What is the minimum I^{2}C bus speed?	From the Access Bus Spec 3.0, chapter 2.1.8 Timing Rules, Paragraph 2.1.8.2 “Bus Timing”: No device can stretch the clock low more than 2 msec. However, except for the host, all devices shall have an I^{2}C byte H/W interface and minimize any clock stretching delays. NOTE: The “2msec time-out” S/W implementation in a device is NOT required in DDC2Bi mode (no multi-master mode)
D03	Why not use the A0/A1 I^{2}C address?	Because it would be in conflict with Extended EDID definition in the VESA DDC Standard. Other AX addresses cannot be used because of current P&D and FPDI standards. Also, some other possible devices (e.g. 24LC21) may be connected on the same bus and decode all AX I^{2}C addresses.
S05	What is the advantage of DDC2Bi compare to DDC2B?	Color Matching and Abstract Implementation is possible, regardless of the video channel implementation and display type. Possibility to perform specific color adjustments and save the display settings host hard disk drive for each user profile. (User registry in the case of Win95.)
G03	Is DDC2Bi host S/W graphic card dependant?	For existing graphics cards in the market, a DLL should be created and used. The DLL is graphic controller chipset dependent, and its identification can be done though PCI ID process. In the future, VBE-SCI will provide a unique interface using the BIOS and will be independent of the chipset.
S06	Does the DDC2Bi host S/W depend on the monitor used? Does the monitor vendor need to supply the DLL to the OS?	No, the host S/W is generic, and will work with any display that complies to the DDC2Bi specification. The DLL may be supplied with the control panel software, i.e. with the graphic board driver software.
S07	What would one need to demonstrate the operation of DCP.exe and PND.exe?	See Appendix A of this specification. The described graphics cards (Number 9, Diamond, All path) are tested and working as expected with DDC2Bi displays.
D04	Can we ship monitors with DDC2Bi function enabled while the spec is still at proposal stage?	The new DDC spec (in voting process) allows other custom serial bus protocols to dialog between the host and the display. DDC2Bi will be in TOTAL HARMONY with the DDC specification. Displays could have DDC2Bi disabled during production, then re-enabled with an I^{2}C custom command on the end-user’s PC when the proposal becomes a standard.

Ref. #	Question	Answer
D06	Can a DDC2AB monitor able to support DDC2Bi environment?	Yes, and this does not require a long development time.
D07	Does modifying the monitor's DDC2AB software to DDC2Bi reduce the ROM size in the MCU?	If DDC2Bi is supported INSTEAD of DDC2B+/AB, then yes, it will cost less ROM and RAM in the monitor MCU. RAM savings come from the communication buffers, since the "Assign address" message is no longer necessary. Also, sharing the Receive/Transmit communication buffers can now be done, which can save additional memory.. ROM savings come from the capability string (no pwr() field), and some Access Bus system messages are no longer required, such as Reset/Attention/PresenceCheck, etc.
S08	Where can I get the Access Bus spec?	The VESA Office has the Access Bus specification in the Word format. Contact the VESA office at (408) 435.0333 for more detail.
S09	Is there any conflict with USB?	Not really, and in fact, in the monitor achitecture, DDC2Bi makes things simpler for USB: If the display design has to optionnally support USB HUB + function, then the USB HUB can translate USB monitor commands into DDC2Bi commands and interface with the main monitor MCU using the same DDC wires.
G05	How can I test my graphics card DDC2Bi drivers?	One solution is to contact DDC2Bi Display manufacturers, or test with the DDC2Bi monitors that are at the VESA Meeting Room.
G06	Why is DDC2Bi likely to have faster market acceptance than Access Bus?	There is one other subtle but significant difference between Access bus and DDC2Bi. The way Access Bus implements the I^2C interface may require a license/royalty payment to the I^2C/Access Bus patents holder. DDC2Bi does not - it is implemented in the S/W only, with the CPU toggling the bits on an I/O port. Since Access bus is multi-master, the graphics controller would have to implement the I^2C-like interface in hardware, resulting in a potentially large licensing and royalty fee to the patent holder. The license/royalty issue was probably enough to prevent many companies from implementing the Access bus.
S10	Who should develop the drivers and application for Control interface on the OS?	It would seem that the graphics board vendors would be the logical choice for this activity. This control panel could support and subsequently display vendor specific control windows to the end-user, but only if monitor vendor/product ID is detected. For example, Windows 95 supports such a feature in their current Control Panel system.
G04	How does one get started?	Use the existing demo tools and the existing source codes: They will greatly accelerate the S/W development.

22. APPENDIX F - I^2C Bus Implementation on Graphic Host

Note: All Appendixes are not part of the standard.

This source code is extracted from SGS-THOMSON demonstration tools (P&D and DCP). They can be reused for quick implementation of I^2C bus on graphic host systems. This code is for guidance only, and is not part of the standard. Code written in Microsoft Visual C++ 4.0 and tested on various Monitors. This code may evolve in the future, and it is recommended to get the latest S/W from the VESA FTP server.

This code generates a global object of type CI2C and is named I2C.
This code statically link with a DDC.DLL which purpose is very similar to VBE-SCI specification.

```
//////////////////////////////////////////////////////////////////////////////
// i2c.h : header file
extern "C" {
extern void enableSerialPort(void);     // to select the external I2C bus
extern void disableSerialPort(void);    // to restore the internal I2C bus
extern BOOL IsHWSupported(void);        // to detect if the DLL works with the H/W
};

// Some I2C error code
#define SCL_TIMEOUT 1
#define BAD_NACK    2
#define   NOT_IDLE  4
#define SLVADR_NACK 8

// ARLO may also occur when we want to send a data bit 1 and we see a 0 on the bus
#define ARLOSS                16

// Access Bus layer error only
#define BAD_CHECKSUM          32
#define BAD_MESSAGE 64

//////////////////////////////////////////////////////////////////////////////
// CI2C view
class CI2C : public CObject
{
public:
BOOL      Configure(BYTE LptNb);        // Select printer port (no more used)
CI2C() {  DLL_Loaded = TRUE;  HWAccess = IsHWSupported();   if(!HWAccess) return;
          enableSerialPort();Configure(1);SetSpeed(10);};
~CI2C() { UnloadDLL(); if(!HWAccess) return;      disableSerialPort();};
// DLL Implementation to detect the PCI ID numbers
WORD      WGetVendorID();
WORD      WGetDeviceID();
BOOL      HWAccess;
// ACCESS BUS LAYER MANAGEMENT COMMUNICATION
BYTE      AbTransfer( BYTE SlvAdr, BYTE *ptr, BYTE& L, BYTE HostAdr = 0x50);    // Data tranfer
// HIGH LEVEL I2C BUS MANAGEMENT COMMUNICATION
BYTE      Transfer( BYTE SlvAdr, WORD& Length, BYTE* ptr, BOOL Stop = TRUE, BOOL AbMsg = FALSE, BOOL Forced = FALSE);   // Data
tranfer
BYTE      SendBlock( WORD Length, BYTE* ptr );
BYTE      ReadBlock( WORD Length, BYTE* ptr, BOOL Ack = FALSE);
// BYTE LEVEL I2C BUS MANAGEMENT COMMUNICATION
BYTE      ReadByte( BYTE& byte, BOOL Ack = TRUE, BOOL Forced = FALSE );         // Send one data byte
BYTE      SendByte( BYTE byte, BOOL Forced = FALSE );       // Read one data byte, clean or not
// BIT LEVEL I2C BUS MANAGEMENT COMMUNICATION
BYTE      ReadBit(BYTE& Bit); // Read one bit on I2C
BYTE      SendStart( BOOL Forced = FALSE );                 // Generate a Start bit,
BYTE      SendStop( BOOL Forced = FALSE );                  // Generate a Stop bit,
BYTE      SendBit( BYTE bit, BOOL Forced = FALSE );         // Send one data bit,
void      ErrorRecovery(BYTE Status);                       // When an I2C error occur,
void      NineStop();                                       // Generate 9 stop bits on the bus
// I/O H/W INTERFACE LEVEL I2C BUS MANAGEMENT COMMUNICATION
BYTE      WGetSDA();
BYTE      WGetSCL();
void      WSetSDA(BYTE Level = 1);
void      WSetSCL(BYTE Level = 1);
// DLL related functions
BOOL      DLL_Loaded;
BOOL      LoadDLL();
BOOL      UnloadDLL();
HINSTANCE _hDdcLibrary;
// TIME LEVEL I2C BUS MANAGEMENT COMMUNICATION
inline BOOL         WaitSCLHigh( INT Delay = 2 ); // Timeout delay in milisecond (getcurrenttime)
BOOL      SetSpeed( WORD speed );                 // I2C Speed Calibration for I2C: 100 kHz (CPU independent)
WORD      GetSpeed();                             // Get current I2C speed
inline void         WaitHalfClock();              // Wait half clock period for sampling data properly
inline void         Wait();                       // Wait routine, in miliseconds...
// For speed calibration: Identical functions but no real I2C communication
BYTE      VoidReadByte( BYTE& byte, BOOL Ack = TRUE, BOOL Forced = FALSE );     // Send one data byte
BYTE      VoidSendByte( BYTE byte, BOOL Forced = FALSE );   // Read one data byte, clean or not
BYTE      VoidReadBit(BYTE& Bit);       // Read one bit on I2C
BYTE      VoidSendBit( BYTE bit, BOOL Forced = FALSE ); // Send one data bit, clean or not
// Internal variables
WORD      LoopCounter;        // The adjusted loop counter for speed adjustment... speed calibration
BYTE      buf;                // The byte buffer for serialization
};
```

```
////////////////////////////////////////////////////////////////////////////
// I2C.cpp : implementation file
#include "stdafx.h"
#include "i2c.h"
#include "conio.h"

#ifdef _DEBUG
#undef THIS_FILE
static char BASED_CODE THIS_FILE[] = __FILE__;
#endif

// MARKER // statically linked DLL functions. (only here)
extern "C" {
extern BYTE GetSCL(void);
extern BYTE GetSDA(void);
extern void SetSCL(BYTE Level);
extern void SetSDA(BYTE Level);
extern WORD GetVendorID(void);
extern WORD GetDeviceID(void);
};


////////////////////////////
// ACCESS BUS/DDC2Bi LAYER
BYTE      CI2C::AbTransfer( BYTE DestAdr, BYTE *ptr, BYTE& L, BYTE HostAdr )
// If Apli = TRUE, bit7 of length = 1
{         // Tranfer to the internal buffer
          WORD Lgth;
          BYTE Chksum;
          BYTE AbMsg[256];
          BYTE Status;

          if((DestAdr & 0x01)==0) // Send data in access bus format
          {
                    AbMsg[0] = HostAdr;                               // Sent from host
                    AbMsg[1] = L;                           // Length including bit 7
                    Chksum = DestAdr ^ HostAdr ^ L;         // Temporary

                    Lgth = L & 0x7F;
                    for( WORD i = 0; i < Lgth; i++){ AbMsg[i+2] = ptr[i]; Chksum ^= ptr[i]; };

                    AbMsg[Lgth+2] = Chksum; Lgth += 3;
                    return Transfer( DestAdr, Lgth, AbMsg );
          };

          ////////////////////////////////////
          // Receive mode (proprietary mode)
          L = 0;
          Status = 0;
          ErrorRecovery(0);   // Force Idle Mode on the bus: SDA and SCL must be high

          // Normally here, it should be okay...
          Status |= SendStart();                  // Generate a Start Bit
          if(Status!=0) {ErrorRecovery(0); return Status;};

          Status |= SendByte(DestAdr);  // Send the destination slave address
          if(Status!=0)
          {
                    ErrorRecovery(0);
                    if(Status & BAD_NACK) Status |= SLVADR_NACK;
                    return Status;      // No acknowledge received
          };

          // Here, we have to check if the acknowledge is successful...
          // Receive mode
          BYTE byte;

          Status |= ReadByte( byte );
          ptr[0] = byte;
          if(Status!=0) {ErrorRecovery(0); return Status;};

          Status |= ReadByte( byte );
          ptr[1] = byte;
          if(Status!=0) {ErrorRecovery(0); return Status;};

          Lgth = (byte & 0x7F)+1; // Add checksum to read

          Status |= ReadBlock( Lgth, ptr + 2 );   // fill all extra byte from offset 2
          Status |= SendStop();         // Send Stop for the send of the communication

          // We have to compute the checksum...
          Chksum = 0x50; for(int i=0; i<(Lgth+2); i++ ) Chksum ^= ptr[i];

          if(Chksum!=0) Status |= BAD_CHECKSUM;   // Bad Checksum

          // We check that the header is 6E hex
          if(ptr[0] != 0x6E) Status |= BAD_MESSAGE;

          if(Status==0) L = ptr[1];               // Hazard for 8x length value...
          if(Status==BAD_CHECKSUM) L = ptr[1];
          Lgth = L & 0x7F;
          // Here, Data must start at ptr[0] instead of ptr[2]... (addendum)
          for(i = 0; i<Lgth; i++ ) ptr[i] = ptr[i+2];

          return Status;
};

/////////////////////////////////////////////////////////////////////////////
// I2C Plateform
BOOL      CI2C::Configure(BYTE LptNb)   // Select printer port
{         // obsolete, used when LPT port was used to make I2C bus.
          LoopCounter = 10;   // Slow speed...

          WSetSDA(1);
          WaitHalfClock();
          WSetSCL(1);         // to configure the data as output... for good SCL behaviour?

          ErrorRecovery(0);   // Make sure that SDA and SCL are set...
          return TRUE;
};

//////////////////////////////////////
// DLL related functions
BOOL      CI2C::LoadDLL()
{         DLL_Loaded = TRUE;  // in the future will be used for DLL dynamic linking
          return DLL_Loaded;
};

BOOL      CI2C::UnloadDLL() { return TRUE; }

//////////////////////////////////////////////
// HIGH LEVEL I2C BUS MANAGEMENT COMMUNICATION
BYTE      CI2C::Transfer( BYTE SlvAdr, WORD& Length, BYTE* ptr, BOOL Stop, BOOL AbMsg, BOOL Forced) // Data tranfer
{         BYTE Status;
          ErrorRecovery(0);

          // Normally here, it should be okay...
          Status = SendStart();                   // Generate a Start Bit
          if(Status!=0) {ErrorRecovery(0); return Status;};

          Status |= SendByte(SlvAdr);   // Send the destination slave address
          if(Status!=0)
          {
                    ErrorRecovery(0);
                    if(Status & BAD_NACK) Status |= SLVADR_NACK;
                    return Status;      // No acknowledge received
          };

          // Here, we have to check if the acknowledge is successful...
          if((SlvAdr & 0x01)!=0)
          {         // Receive mode
                    Status |= ReadBlock( Length, ptr );
                    if(Status!=0) {ErrorRecovery(0); return Status;};
          }
          else
          {         // Transmit mode
                    Status |= SendBlock( Length, ptr );
                    if(Status!=0) {ErrorRecovery(0); return Status;};
          };

          if(Stop) Status |= SendStop();          // No restart expected
          return Status;
};

BYTE      CI2C::SendBlock( WORD Length, BYTE* ptr )
{         BYTE Status = 0;
          for(WORD i = 0; i < Length; i++ )
          {         Status |= SendByte(ptr[i]);
                    if(Status!=0) return Status;
          };
          return 0;
};

BYTE      CI2C::ReadBlock( WORD Length, BYTE* ptr, BOOL Ack )         // Ack is no more used! full stop!
{         BYTE Status = 0;
          for(WORD i = 0; i<Length; i++)
          {         Status |= ReadByte(ptr[i], (i!=(Length-1)) );
                    if(Status!=0) return Status;
          };
          return 0;
};

//////////////////////////////////////////////
// BYTE MANIPULATION FUNCTIONS
BYTE      CI2C::ReadByte( BYTE& byte, BOOL Ack, BOOL Forced )         // Send one data byte, clean or not
{         BYTE      b0,b1,b2,b3,b4,b5,b6,b7;
          BYTE Status;
          Status = ReadBit(b7);         Status |= ReadBit(b6);        Status |= ReadBit(b5);        Status |= ReadBit(b4);
          Status |= ReadBit(b3);        Status |= ReadBit(b2);        Status |= ReadBit(b1);        Status |= ReadBit(b0);
          Status |= SendBit(!Ack);

          byte = b7*128 + b6*64 + b5*32 + b4*16 + b3*8 + b2*4 + b1*2 + b0; // I'm a bit lazy...
          return Status;
};

BYTE      CI2C::SendByte( BYTE byte, BOOL Forced )          // Read one data byte, clean or not
{         BYTE ack = 0;
```

```
          BYTE Status;
          Status = SendBit(byte&128);   Status |= SendBit(byte&64 );  Status |= SendBit(byte&32 );  Status |= SendBit(byte&16);
          Status |= SendBit(byte&8  );  Status |= SendBit(byte&4  );  Status |= SendBit(byte&2  );  Status |= SendBit(byte&1 );
          Status |= ReadBit(ack);
          if(ack==1) Status |= BAD_NACK;
          return Status;
};

/////////////////////////
// I2C BIT MANIPULATION
BYTE      CI2C::SendStart( BOOL Forced )          // Generate a Start bit, clean or rude ?
{         //Assert SDA==1
          WSetSDA(1);
          Wait();

          if(WGetSDA()==0) return NOT_IDLE;       // should not occur, have to generate 9 stop bit?
          Wait();
          if(!WaitSCLHigh()) return SCL_TIMEOUT;
          Wait();   // Delay between stop and start? TBD...

          // The real start sequence
          WSetSDA(0);
          Wait();                                 // seems better for 4 us delay due to calibration
          WaitHalfClock();    // SDA FallingEdge
          WSetSCL(0);
          Wait();   // SCL low half period
          return 0;
};

BYTE      CI2C::SendBit( BYTE bit, BOOL Forced )  // Send one data bit, clean or not
{         // Assert(SCL==0)
          WSetSCL(0);                                                   // SCL forced low optionnal
          Wait();                                                       // Delay optionnal
          WSetSDA(bit);                                                 // update SDA
          Wait();                                                       // wait half period

          if(!WaitSCLHigh()) return SCL_TIMEOUT;  // Here is the delay for clock stretching
          WaitHalfClock();                                  // Wait SCL
          // if(GetSDA()!=bit) return ARLOSS;
          WSetSCL(0);
          Wait();
          return 0;
};

BYTE      CI2C::ReadBit(BYTE& Bit)      // Read one bit on I2C
{         WSetSCL(0);                             // SCL forced low
          Wait();             // Delay
          WSetSDA(1);                             // Raise SDA, for the other guy to handle it low or not
          Wait();
          WaitSCLHigh();                          // Here is the delay for clock stretching ***
          WaitHalfClock();              // Wait SCL
          Bit = WGetSDA();              // Sampling SDA
          WSetSCL(0);                             // Add-on
          Wait();
          return 0;
};

BYTE      CI2C::SendStop( BOOL Forced ) // Generate a Stop bit, clean or not..
{         WSetSCL(0);
          Wait();
          WSetSDA(0);
          Wait();
          if(!WaitSCLHigh()) return SCL_TIMEOUT;
          WaitHalfClock();
          WSetSDA(1);
          WaitHalfClock();
          return 0;
};

void      CI2C::ErrorRecovery(BYTE Status)                  // When an I2C error occur,
{         // What we want is to recover any stuck situation...
          for( int i=0; i<9; i++ )                          // Maximum 9 stop bits...
          {         if((WGetSDA()==1)&(WGetSCL()==1)) break;
                    SendStop();
          };
}

void      CI2C::NineStop()
{         if(!HWAccess) return;
          enableSerialPort();
          for( BYTE i=0; i<9; i++ ) SendStop();
};

////////////////////////////////////
// H/W DEPENDENT LOW LAYER ROUTINES
BYTE      CI2C::WGetSDA()     // Working
{         if(!HWAccess) return 1;
          if( GetSDA()==0 ) return 0; else return 1;
};



BYTE      CI2C::WGetSCL()     // Working
{         if(!HWAccess) return 1;
          if( GetSCL()==0) return 0; else return 1;
};
```

```
void      CI2C::WSetSDA(BYTE Level)     // Working
{         if(!HWAccess) return;
          if(Level==0) SetSDA(0); else SetSDA(1);
};

void      CI2C::WSetSCL(BYTE Level)     // Working
{         if(!HWAccess) return;
          if(Level==0) SetSCL(0); else SetSCL(1);
};

/////////////////////////////////////////////
// TIME DEPENDENT LOW LEVEL ROUTINES
BOOL      CI2C::WaitSCLHigh( INT Delay )          // Timeout delay for polling in msec
{         // the timeout is 2 msec at 100 kHz; WaitHalfClock is 5 us
          // 2000 / 5 us = 400 times wait halfclock... Get Current Time doesn't work
          WSetSCL(1);         // Want to release SCL anyway

          for( WORD i=0; i<500; i++ )   // timeout of 400 half bit =
          {         if(WGetSCL()==1) break;
                    WaitHalfClock();
          };
          return (WGetSCL()==1);
};

//////////////////////
// Speed calibration
void      CI2C::WaitHalfClock()         // Wait half clock period for sampling data properly
{         Wait(); Wait(); };

void      CI2C::Wait()        // Wait routine, in miliseconds...
{         BYTE j;   for(BYTE i= 0;i<LoopCounter;i++, j++); }

WORD      CI2C::GetSpeed()    { return LoopCounter;};       // Get current I2C speed

BOOL      CI2C::SetSpeed( WORD speed ) // Calibration for I2C speed at nominal 100 kHz
{         WORD      NbByte;
          DWORD     Time0, Time;
          BYTE      byte;
          NineStop(); // Error Recovery

          // 981 bytes around 1000 should be 100 msec.
          for(BYTE LoopCounter = 1; LoopCounter<255; LoopCounter++)
          {         Time0 = GetCurrentTime();
                    for(NbByte=0;NbByte<100;NbByte++)       // Send 100 bytes
                    {         VoidSendByte( 0xFF );                   // Envoyer FF et NACK
                              VoidReadByte( byte, FALSE );
                    };

                    // Here 100x9x10 us should have elapsed = 18 msec limit
                    Time = GetCurrentTime() - Time0; if(Time>=18) break;        // We have reached the speed limit!
          };
          LoopCounter++;                // Final evaluation
          ErrorRecovery(0);
          return TRUE;
}

/////////
// For I2C bus speed calibration
BYTE      CI2C::VoidReadByte( BYTE& byte, BOOL Ack, BOOL Forced )     // Send one data byte
{         BYTE      b0,b1,b2,b3,b4,b5,b6,b7;
          BYTE Status = 0;
          Status |= VoidReadBit(b7);    Status |= VoidReadBit(b6);    Status |= VoidReadBit(b5);    Status |= VoidReadBit(b4);
          Status |= VoidReadBit(b3);    Status |= VoidReadBit(b2);    Status |= VoidReadBit(b1);    Status |= VoidReadBit(b0);
          Status |= VoidSendBit(!Ack);

          byte = b7*128 + b6*64 + b5*32 + b4*16 + b3*8 + b2*4 + b1*2 + b0;
          return Status;
};

BYTE      CI2C::VoidSendByte( BYTE byte, BOOL Forced )      // Read one data byte
{         BYTE Status = 0;
          BYTE ack = 0;
          Status |= VoidSendBit(byte&128);        Status |= VoidSendBit(byte&64 );        Status |= VoidSendBit(byte&32 );
          Status |= VoidSendBit(byte&16 );        Status |= VoidSendBit(byte&8  );        Status |= VoidSendBit(byte&4  );
          Status |= VoidSendBit(byte&2  );        Status |= VoidSendBit(byte&1  );        Status |= VoidReadBit(ack);
          if(ack==1) Status |= BAD_NACK;
          return Status;
};

BYTE      CI2C::VoidSendBit( BYTE bit, BOOL Forced ) // Send one data bit, clean or not
{// Assert(SCL==0)
          WSetSCL(1);                   // SCL forced low optionnal
          Wait();                       // Delay optionnal
          WSetSDA(1);                   // update SDA
          Wait();                       // wait half period

          if(!WaitSCLHigh()) return SCL_TIMEOUT;  // Here is the delay for clock stretching
          WaitHalfClock();    // Wait SCL
          // if(GetSDA()!=bit) return ARLOSS;
          WSetSCL(1);
          Wait();
          return 0;
};

BYTE      CI2C::VoidReadBit(BYTE& Bit)  // Read one bit on I2C
{         WSetSCL(1);                   // SCL forced low
          Wait();                       // Delay
          WSetSDA(1);                   // Raise SDA, for the other guy to handle it low or not
          Wait();
          WaitSCLHigh();                // Here is the delay for clock stretching ***
          WaitHalfClock();    // Wait SCL
          Bit = WGetSDA();    // Sampling SDA
          WSetSCL(1);                   // Add-on
          Wait();
          return 0;
};

WORD      CI2C::WGetVendorID()          { return GetVendorID(); };

WORD      CI2C::WGetDeviceID()          { return GetDeviceID(); };
```

23. APPENDIX G - DDC2Bi Basic Function Implementation

Note: All Appendix are not part of the standard.

This source code is extracted from SGS-THOMSON demonstration tools (P&D and DCP). They can be reused for quick implementation of I^2C bus on graphic host systems. This code is for guidance only, and is not part of the standard. Code written in Microsoft Visual C++ 4.0 and tested on various Monitors. (see VESA FTP server)
This code create a global object of class CAB and named AB.
This code uses the resources of the I^2C global object which is the lower layer S/W interface.

```
/////////////////////////////////
// ab.h : header file (DDC2Bi functions)

#define NOT_SUPPORTED        128
#define OTHER_ERROR 128
// Do we have to specify the number of communication retrials in case of failures?

/////////////////////////////////////////////////
// CAB
class CAB : public CObject
{
public:
CAB() { LinkEnabled = FALSE; };

///////////////////
// ACCESS BUS LAYER MANAGEMENT COMMUNICATION
BYTE      GetSelfTestReport( BYTE& Flags, BYTE *ptr, BYTE& Length);            // Application Test

//////////////////////////////
// CMD functions
// VCP manipulation functions
BYTE      GetVCP( BYTE VCP, WORD& Cur, WORD& Max, BOOL& Temporary );  // We have to check it exists!
BYTE      SetVCP( BYTE VCP, WORD& Cur );
BYTE      ResetVCP( BYTE VCP );
BYTE      EnVCP( BYTE VCP, BOOL Enable );                  // Enable/Disable VCP feature
BYTE      SaveCurrentSettings();

////////////////////////////////
// MORE FRIENDLY EXTRACTED INFO FROM CAPABILITY
BYTE      GetID( CString& ProtRev, CString& ModuleRev, CString& VendorName, CString& ModuleName, DWORD& DevNb );
// This is just the ID String extracted for immediate reuse without headaches

BYTE      GetProt( CString& Prot );
BYTE      GetType( CString& Type );
BYTE      GetPrev( CString& Prev );
BYTE      GetEDID( BYTE* edid, WORD& Length );
BYTE      GetVDIF( BYTE* edid, WORD& Length );
BYTE      GetCmds( CString& Cmds );
BYTE      GetVCP( CString& Vcp );
BYTE      GetVCPNames( CString& VcpNames );

// SYSTEM CONFIGURATION LAYER COMMAND SET
// IT SIMPLY UPDATE THE OBJECT INTERNAL VARIABLES
BYTE      Reset();                      // Send reset message
BYTE      GetID();                      // Get ID String
BYTE      GetCapability();              // Get Capability String (limited to 8 kb buffer)
BYTE      IsPresent();                                    // Presence Check (ACK 6E/6F)
BYTE      EnableAppReport(BOOL Enable = TRUE);    // Enable application report

// VERY LOW LEVEL COMMANDS
BOOL      IsAck( BYTE I2C_Adr );        // Check if a guy is acknowledging the selected I2C address
BOOL      GetPotential( BOOL& DDC2B, BOOL& DDC2AB, BOOL& SEB );
BYTE      SendAbMsg( BYTE *ptr, BYTE L = 0x81);            // Data tranfer
BYTE      GetAnswer(BYTE *ptr, BYTE& Length);              // Get answer from device

// Error management
CString GetErrorMsg( BYTE Status );     // Translate all status flags for user string
void      Wait( BYTE msec );
void      NineStops();

// For configuration
BYTE      ConfigureDevice();  // Send all the monitor access bus system commands to configure it
BOOL      IsEnabled();                  // Tells the application if a display device is present
public:
BYTE      Msg[512];           // Shared Buffer for transmit or receive data

// Internal Backup data for working operation
BYTE      IdStr[64];                    // ID String (fixed size)
BYTE      CapaStr[9000];      // Capability String
WORD      CapaLgth;           // the size of the capability string

// VCP Remote Information
BOOL      LinkEnabled;        // The access Bus application layers are working
BYTE      Delay;                        // A latency programmable delay
BOOL      IsVCPSupported(BYTE VCP);     // Is a VCP control available (getVCP check way)
};
```

```
/////////////////////////////////////////
// ab.cpp : implementation file

#include "stdafx.h"
#include "ab.h"

#include "i2c.h"
extern CI2C         I2C;

#define delay 20

#ifdef _DEBUG
#undef THIS_FILE
static char BASED_CODE THIS_FILE[] = __FILE__;
#endif

///////////////////
// ACCESS BUS LAYER MANAGEMENT COMMUNICATION
// GENERIC ACCESS BUS COMMANDS

BYTE      CAB::IsStillThere()                               // Check if the display is still there (EDID ID identical and 6E
{
          if(!IsEnabled()) return OTHER_ERROR;

          Msg[0] = 0xF1;
          BYTE Length = 0x81;
          BYTE Status = SendAbMsg( Msg, Length );
          if(Status!=0) return Status;

          Wait(delay);

          BYTE BlkLgth;
          Status = GetAnswer( Msg, BlkLgth );
          if(Status != 0) return Status;

          for( int i = 0; i<28; i++ ) if(IdStr[i] != Msg[i+1]) return OTHER_ERROR;
          return 0;
};

//////////////////////////////
// CMD functions
// VCP manipulation functions
BYTE      CAB::GetVCP( BYTE VCP, WORD& Cur, WORD& Max, BOOL& Temporary )
{         if(!IsEnabled()) return OTHER_ERROR;

          Msg[0] = 0x01;
          Msg[1] = VCP;       // VCP code
          BYTE Length = 0x82;
          BYTE Status = SendAbMsg( Msg, Length );
          if(Status !=0 ) return Status;

          Wait(delay);

          BYTE BlkLgth;
          Status = GetAnswer( Msg, BlkLgth );
          if(Status != 0 ) return Status;

          // I should check the op-code normally... let's do it later (I've written the reminder!)
          Max = Msg[4] * 256 + Msg[5];
          Cur = Msg[6] * 256 + Msg[7];
          Temporary = (Msg[3]!=0);

          if(Msg[1] != 0) { Max = Cur = 0;        return OTHER_ERROR; };
          return 0;
};

BYTE      CAB::SetVCP( BYTE VCP, WORD& Cur )
{         if(!IsEnabled()) return OTHER_ERROR;

          Msg[0] = 0x03;
          Msg[1] = VCP;       // VCP code
          Msg[2] = HIBYTE( Cur );
          Msg[3] = LOBYTE( Cur );

          BYTE Length = 0x84;
          return SendAbMsg( Msg, Length );
};

BYTE      CAB::ResetVCP( BYTE VCP )
{         if(!IsEnabled()) return OTHER_ERROR;

          Msg[0] = 0x09;
          Msg[1] = VCP;       // VCP code

          BYTE Length = 0x82;
          BYTE Status;
          Status = SendAbMsg( Msg, Length );
          if(Status != 0 ) return Status;

          // New add-on not tested
          BYTE BlkLgth;
          Status = GetAnswer( Msg, BlkLgth );
          return Status;
};

BYTE      CAB::EnVCP( BYTE VCP, BOOL Enable )               // Enable/Disable VCP feature
{         if(!IsEnabled()) return OTHER_ERROR;
```

```
          if( Enable ) Msg[0] = 0x0B; else Msg[0] = 0x0A;
          Msg[1] = VCP;       // VCP code

          BYTE Length = 0x82;
          BYTE status;
          status = SendAbMsg( Msg, Length );

          // New add-on not tested: if VCP Reply to discard
          BYTE BlkLgth;
          status = GetAnswer( Msg, BlkLgth );
          return status;
};

BYTE      CAB::SaveCurrentSettings()
{         if(!IsEnabled()) return OTHER_ERROR;
          Msg[0] = 0x0C;

          BYTE Length = 0x81;
          return SendAbMsg( Msg, Length );
};


////////////////////////////////
// SYSTEM CONFIGURATION LAYER COMMAND SET

BYTE      CAB::GetID()                            // Get ID String
{
          Msg[0] = 0xF1;
          BYTE Length = 0x81;
          BYTE Status = SendAbMsg( Msg, Length );
          if(Status!=0) return Status;

          Wait(delay);

          BYTE BlkLgth;
          Status = GetAnswer( Msg, BlkLgth );
          if(Status != 0) return Status;

          for( int i = 0; i<28; i++ ) IdStr[i] = Msg[i+1];  // copy this stuff
          return 0;
};

BYTE      CAB::GetCapability()          // Get Capability String (limited to 8 kb buffer)
{
          BYTE Status, BlkLgth, i,test;
          BYTE Length = 0x83;
          WORD Ofst = 0x0000;

          CapaLgth = 0;       // restart from scratch

          for(      test=0;   ; Ofst += BlkLgth, test++ )
          {
                    Msg[0] = 0xF3;
                    Msg[1] = HIBYTE(Ofst);
                    Msg[2] = LOBYTE(Ofst);
                    Length = 0x83;

                    Status = SendAbMsg( Msg, Length );
                    if(Status != 0) return Status;                    // Error management...

                    if(test==200) return 128;
// check the loop: overflow watchdog used! If the capability string exceed 200 segments, stop it!

                    Wait(delay);        // typical wait for message to be ready!
                    Status = GetAnswer( Msg, BlkLgth );
                    if(Status != 0) return Status;
                    BlkLgth &= 0x7F;
                    BlkLgth -= 3;       // Number of data bytes for the capability string
                    if(BlkLgth==0) { CapaLgth = Ofst; break;};        // finished!
                    for( i=0; i<BlkLgth; i++ ) CapaStr[Ofst+i] = Msg[i+3];
                    Wait(delay);        // To be sure...
          };
          return 0;
};

BYTE      CAB::EnableAppReport(BOOL Enable)       // Enable application report
{
          Msg[0] = 0xF5;
          if(Enable)          Msg[1] = 0x01; else Msg[1] = 0x00;      // Enable/Disable application report
          BYTE Length = 0x82;
          BYTE Status = SendAbMsg( Msg, Length );
          if(Status==0) LinkEnabled = Enable;
          return Status;
};

BYTE      CAB::GetSelfTestReport( BYTE& Flags, BYTE *ptr, BYTE& Length)         // Application Test
{
          Msg[0] = 0xB1;
          Length = 0x81;
          BYTE Status = SendAbMsg( Msg, Length );
          if(Status !=0 ) return Status;


          Wait(delay);
          BYTE BlkLgth;
          Status = GetAnswer( ptr, BlkLgth );
          if(Status != 0 ) return Status;
          Length = BlkLgth & 0x7F;      // All data message including flags
          Flags = ptr[1];               // Self Test Result

          // I should check the op-code normally... let's do it later (I've written the reminder!)
          return 0;
};

////////////////////////////////
// VERY LOW LEVEL COMMANDS
BOOL      CAB::GetPotential( BOOL& DDC2B, BOOL& DDC2AB, BOOL& SEB )
{
          DDC2B     = IsAck(0xA0) & IsAck(0xA1);
          DDC2AB    = IsAck(0x6E);
          SEB       = IsAck(0x6F);
          return 0;
};

BOOL      CAB::IsAck( BYTE I2C_Adr )    // Check if a guy is acknowledging the selected I2C address
{
          BYTE Status, buf;
          WORD Length = 0;
          Status = I2C.Transfer( I2C_Adr, Length, &buf, TRUE );       // Data tranfer
          return (Status==0); // Check if ok
};

BYTE      CAB::SendAbMsg( BYTE *ptr, BYTE L)      // Data tranfer
{         // Translate for Access Bus slave mode functions
          return I2C.AbTransfer( 0x6E, ptr, L, 0x51);       // Data tranfer
};

BYTE      CAB::GetAnswer(BYTE *ptr, BYTE& Length)           // Get answer from device
{         // The goal of this function is to read the Ab message from the 6F device
          return I2C.AbTransfer( 0x6F, ptr, Length, 0x51 ); // Data tranfer in proprietary mode
};

////////////////////////////////
// Error Mangagement
CString CAB::GetErrorMsg( BYTE Status ) // Translate all status flags for user string
{
          CString Txt = "Errors: ";
          if( Status & SCL_TIMEOUT) Txt += "SCL_TIMEOUT ";
          if( Status & BAD_NACK) Txt += "BAD_NACK ";
          if( Status & NOT_IDLE) Txt += "NOT_IDLE ";
          if( Status & SLVADR_NACK) Txt += "SLVADR_NACK ";
          if( Status & ARLOSS) Txt += "ARLOSS ";
          if( Status & BAD_CHECKSUM) Txt += "BAD_CHECKSUM ";
          if( Status & BAD_MESSAGE) Txt += "BAD_MESSAGE ";
          if( Status & 128) Txt += " OTHER_ERROR";
          if(Status==0) Txt = "No errors";
          return Txt;
};

// seems working not that bad
void CAB::Wait( BYTE msec )
{
          DWORD Time0 = GetCurrentTime();
          for( ; (GetCurrentTime() - Time0) < msec; );
};

void CAB::NineStops()         { I2C.NineStop(); };

BOOL      CAB::IsEnabled()              // Tells the application if a display device is present
{ return LinkEnabled; };

BOOL CAB::IsVCPSupported(BYTE VCP)
{
          WORD Cur,Max;
          BOOL Temp;
          if(GetVCP(VCP, Cur, Max, Temp)!=0 ) return FALSE; // If error: unsupported
          if(Max==0) return FALSE;
          return TRUE;        // yes, this VCP definitely exist!
};
```

24. APPENDIX H - Other source code files available

Note: All Appendix are not part of the standard.

Here is a list of existing source code available in the VESA FTP server:

I2C.CPP	The I^{2}C Bus Class described above
AB.CPP	The higher level communication interface object, descibed above
EDID.CPP	The EDID data editor object (1.0 and 1.1)
COLORTUNE.CPP	The White Color Interpolation Formulas
DCP.EXE	Display Control Pannel Debugging Tool for Windows 95/Menphis
PND.EXE	Plug and Display End User Utility for Windows 95/Memphis
DDC.DLL	The driver interface used by above applications to drive S3D Virge chipset
DDC.C	The DLL source code. (prototype)

25. APPENDIX I - Host S/W driver implementation

Note: All Appendix are not part of the standard.

A discussion is currently on going to define the best S/W architecture on the host operating system in order to have the possibility to have multiple applications sharing the same I^{2}C bus without conflict, and be H/W independent. This topic is not part of this standard.

The communication can be split in different layers.
The lowest layer is directly driving the H/W and implement the functions described in the VBE-SCI specification.
Higher layers must deal with multiple I^{2}C bus on the same graphic hosts and also multiple graphic host support
Hihest layer will interface with multiple Applications.
Then, any application using the I^{2}C bus will call the highest S/W layer.

***** End of Standard *****