HDMI – Hacking Displays Made Interesting

Andy Davis
Research Director
andy.davis@ngssecure.com

An NGS Secure Research Publication

9 March 2012

http://www.ngssecure.com

Table of Contents

1. Introduction

Picture this scene, which incidentally happens thousands of times every day all around the world: Someone walks into a meeting room, sees a video cable and plugs it into their laptop. The other end of the cable is out of sight – it just disappears through a hole in the table. What is it connected to? Presumably the video projector bolted to the ceiling, but can it be trusted to *just* display their PowerPoint presentation?

In this paper I will explain the circumstances in which display devices send data to their connected host and show that this data could potentially contain threats (which could compromise a laptop for example). I will describe video protocol data-structures, data-sequences and practical challenges. I will also explain how to build a hardware-based fuzzer, provide some example firmware fuzzing code, and describe some interesting findings from the fuzzing which has been undertaken so far.

This paper discusses the security of video drivers which interpret and process data supplied to them by external displays, projectors and KVM switches. It covers all the main video standards, including VGA, DVI, HDMI and DisplayPort.

This is a relatively new area of research and there is more research that could be performed in this area, so by summarising and sharing these resources, it is hoped that this will enable others to more quickly discover and investigate potential threats.

2. History and overview of video display and interface standards

There have been many display standards developed over the years stretching back to the 1970's and probably earlier. Video display standards typically include information such as:

- Screen resolutions
- Colour modes and palette
- Refresh rates

Whereas interface standards are more likely to define:

- Transmission protocols
- Compression techniques
- Encryption schemes

Before discussing the security implications of display technologies it is worth spending some time covering the history of the main video and interface standards that are still in use today.

2.1.   VGA – Video Graphics Array

The term VGA (Video Graphics Array) originally related to the display technology implemented on IBM PCs in the late 1980's. As with subsequent video standards, the name has become synonymous with both the video protocol standard and the physical connector type. Hence people talk about "VGA connectors" meaning the 15-pin D-type interface that everyone is familiar with:

Figure 1: 15-pin D-type female (VGA socket on a PC)

VGA is an analogue video standard – the separate Red, Green and Blue analogue video signals are sent along with horizontal sync[1] and vertical sync[2] signals. There are also four pins that were originally reserved to provide monitor identification data to the host machine; these were known as ID0, ID1, ID2 and ID3. The pins ID0-ID2 were attached to resistors to pull one or more of them to ground. These three bits allowed for the definition of up to seven monitor types, with all open meaning "no monitor present". Only three monitor types were ever defined:

- monochrome with a resolution of less than 1024×768 (NC, NC, GND)
- colour with a resolution of less than 1024×768 (NC, GND, NC)
- colour with 1024×768 (GND, GND, NC)

Pin ID3 was unused. The implementation of the VESA (Video Electronics Standards Association)[3] DDC (Display Data Channel)[4] standard changed all this (more on DDC later).

2.2.   DVI – Digital Visual Interface

As its name implies, DVI is a digital interface standard. It was developed by the Digital Display Working Group (DDWG)[5] to replace the VGA interface, which is viewed as an outdated legacy standard.

Figure 2: DVI socket on a PC   Figure 3: mini DVI socket on a Mac

Uncompressed video signal data is transmitted using TMDS (Transition Minimised Differential Signalling)[5] to reduce noise. For backward-compatibility DVI also includes analogue pins to transmit R,G,B and sync data (a la VGA). There are many more technical details associated with the DVI standard[6], however, these will not be discussed here. From a security perspective, the important thing is that DVI also supports DDC for display identification and capability advertisement.

2.3.　HDMI - High-Definition Multimedia Interface

The most recent well-known video interface standard is HDMI, which is not only used for in the world of IT, but more commonly in consumer electronics for transmitting video and audio data between devices such as Blu-ray players and flat screen TVs.
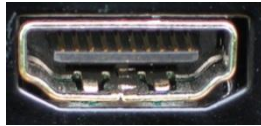
Figure 4: HDMI "type A" socket
(Most commonly seen on consumer devices)

HDMI transmits encrypted uncompressed digital video and audio data (using TMDS like DVI), it supports DDC for display identification and capability advertisement, but also it introduces a number of new technologies, which are potentially interesting from a security perspective; these include:

- CEC – Consumer Electronics Control[7]
- HDCP - High-bandwidth Digital Content Protection[8]
- HEC – HDMI Ethernet Channel[9]

All these technologies will be discussed in more detail later in the paper.

2.4.　DisplayPort

The DisplayPort standard was developed by VESA to complement HDMI. The specification is effectively a royalty-free equivalent to HDMI (the HDMI royalty fee is US$0.04 per device and has an annual fee of US$10,000 for high-volume manufacturers).

Figure 5: DisplayPort socket
(The DP++ logo indicates dual mode i.e. compatible with DVI/HDMI displays)

DisplayPort uses packet-based data transmission (like Ethernet) and supports DDC and HDCP, in addition to DPCP (DisplayPort Content Protection)[10]. However, DisplayPort does not natively support CEC.

Now the basics have been covered, it's time to discuss how security fits into this discussion

3.　A video interface is an output isn't it?

…well yes… and no

The DDC (Display Data Channel) protocol has already been mentioned with respect to all the video interface standards. DDC enables a connected display to communicate its supported display modes to the adapter and to enable the host device to adjust various monitor parameters to ensure the best

video output is displayed. When a monitor is connected to a PC or a Mac there is a short delay before the video is displayed – during this delay data is being provided by the display via DDC to facilitate the "plug-and-play" approach to computing. The display's capabilities are transmitted in a 128-byte block called an EDID (Extended Display Identification Data)[11] structure.

3.1.   DDC v1

When discussing VGA, I mentioned that there are a number of reserved pins (ID0-ID3). DDC1 used a low-speed unidirectional serial protocol to continuously send the EDID block via ID1 (the data was clocked using the vertical sync line). However, very few vendors adopted DDC1.

3.2.   DDC v2

There are a number of sub-versions of DDC2, however the most commonly implemented is DDC2B, which uses I²C (Inter-Integrated Circuit)[12] – a two-wire serial protocol widely used for communication between chips on circuit boards. The ID1 pin is used for the data line (called SDA) and ID3 is used for the clock (called SCL). Pin 9 on a VGA 15-pin connector was previously used as a mechanical key, but for DDC2B it supplies +5V to power up the E^2PROM in the monitor that contains the EDID block and therefore, the EDID data can be read even if the monitor is powered off.

The SDA and SCL pins are important to understand because they are present on (and DDC2B is supported by) all the video interfaces discussed in this paper.

3.3.   E-DDC - Enhanced Display Data Channel

The E-DDC [13] protocol utilises a segment pointer which enables up to 32K bytes of display information to be retrieved using the E-EDID (Enhanced EDID)[14] standard.

Although the standard EDID block is 128 bytes in length, standards exist for "extension blocks", each of which are also 128 bytes in length. DDC2 allowed one EDID block followed by one extension block to be transmitted by a display. However, with E-DID up to 32K bytes can be addressed, which equates to 256 x 128 byte blocks.

So, if displays are providing all this data to the hosts to which they are connected then the video drivers running on the hosts will need to parse this data and any software that parses data is potentially vulnerable to software flaws that could result in security vulnerabilities.

3.4.   The EDID structure

So what is contained in an EDID structure? The first block (initial 128 bytes) is called the VESA block – much of the data must reside in specific positions within the block, but later in the block different descriptors can be used depending on requirements.

3.4.1.   Header

The first eight bytes are a fixed value header:

```
00 FF FF FF FF FF FF 00
```

3.4.2.   Vendor and product information

The next ten bytes contain information such as the manufacturer ID and product code, a serial number and the year of manufacture.

3.4.3.   EDID version and revision

The version of EDID this block conforms to – at the time of writing the most recent version is EDID v1.4.

3.4.4.   Video input definition

The first value is a bitmap, which specifies data such as digital or analogue signal, compatibility with TMDS (if digital) and separate sync supported (if analogue). Then there are two bytes representing the maximum horizontal and vertical image size (in centimetres).

3.4.5.   Display transfer characteristics

Two bytes represent the display gamma and a supported features bitmap

3.4.6.   Colour characteristics

The 10-bit CIE xy co-ordinates[15] for red, green, blue, and white

3.4.7.   Established timings

A three-byte bitmap represents common timing modes

3.4.8.   Standard timing information

Standard timings are timings that are recognised by VESA through the VESA Discrete Monitor Timing[16] or Generalised Timing Formula[17] standards.

3.4.9.   Descriptor blocks

There is space within the VESA block for up to four descriptor blocks. These descriptors can be:

- Detailed Timing Descriptor
- Other Monitor Descriptor
- Monitor Range Limits Descriptor
- Additional White Point Descriptor

3.4.10.   Extension flag

The number of extension blocks to follow. This is set to zero if there are no extension blocks

3.4.11.   Checksum

The one-byte sum of all 128 bytes in the EDID block must equal zero.

There are many utilities available to interrogate and decode EDID data; an example is shown in figure 6.
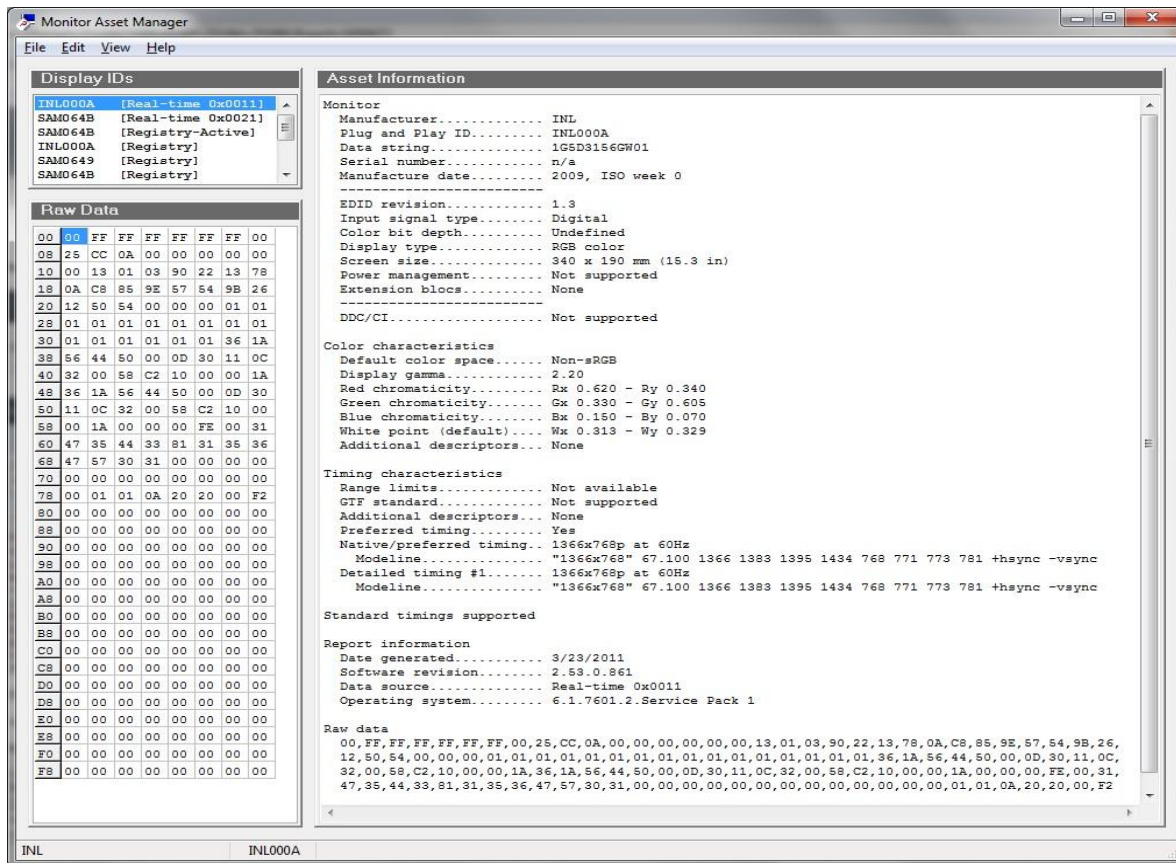
Figure 6: Raw EDID data on the left and decoded data on the right

3.5. EDID extension blocks

There are a number of extension blocks that can be implemented in addition to the VESA block, the most common of which is the CEA block that primarily contains additional timing information. Examples of some of the extension block types include:

- Video Timing Block Extension[18]
- Display Information Extension[19]
- Localised String Extension[20]
- Block Map[21]

However, when extension blocks are used, there are a number of rules that must be followed:

- The VESA block (Block 0) is always required
- At least one CEA extension block is required
- If more than one extension block is used they must all be the same EDID version

Furthermore, if more than two blocks (including the VESA + CEA blocks) are used then a "Block Map" block is required to define the blocks after the VESA block.

So as can be seen from the data discussed so far there is the possibility of identifying logic errors in the way this data is parsed and subsequently processed, but there is also the potential for buffer overflows based on the data supplied in the Localised String Extension block.

3.5.1. Localised string extension

This is an *"…optional extension to the base Enhanced EDID data structure. This structure provides data in the form of character strings used to describe the monitor. These strings are used to supplement or replace similar strings in the base EDID by providing more complete string descriptions and/or string descriptions using different languages or localizations".* [22]

The interesting elements within the block are stored in the "String Table"; these are:

- Manufacturer name string
- Manufacturer name string length
- Model name string
- Model name string length
- Serial number string
- Serial number string length

Therefore, if the parsing code has been insecurely written then a buffer overflow could result from malicious data supplied in this block. Figure 7 shows an example of a Localised string extension block.

Byte # (decimal)	Byte # (hex)	Field Name and *Comments*	Value (hex)	Value (binary)
00	00h	Extension ID	50h	01010000
01	01h	LS-Ext Version number	01h	00000001
02	02h	LS-Ext Revision number	00h	00000000
03	03h	Unicode Version – Major/Minor	32h	00110010
04	04h	Unicode Version – Update	00h	00000000
05	05h	String Table Size	28h	00101000
06	06h	String Table Header – UTF 8	00h	00000000
07	07h	Language ID structure – Default Neutral String Table	00h	00000000
08	08h		00h	00000000
09	09h		00h	00000000
10	0Ah		00h	00000000
11	0Bh	Manufacturer Name Data Length	0Eh	00001110
12	0Ch	'D'	44h	01000100
13	0Dh	'i'	69h	01101001
14	0Eh	's'	73h	01110011
15	0Fh	'p'	70h	01110000
16	10h	'l'	6Ch	01101100
17	11h	'a'	61h	01100001
18	12h	'y'	79h	01111001
19	13h	's'	73h	01110011
20	14h	','	2Ch	00101100
21	15h	' '	20h	00100000
22	16h	'I'	49h	01001001
23	17h	'n'	6Eh	01101110
24	18h	'c'	63h	01100011
25	19h	'.'	2Eh	00101110
26	1Ah	Model Name String Length	06h	01100000
27	1Bh	'F'	46h	01000110
28	1Ch	'C'	43h	01000011
29	1Dh	'1'	31h	00110001
30	1Eh	'9'	39h	00111001
31	1Fh	'0'	30h	00110000
32	20h	'1'	31h	00110001
33	21h	Serial Number Data String Length	0Ch	00001100
34	22h	'0'	30h	00110000
35	23h	'3'	33h	00110011
36	24h	'2'	32h	00110010
37	25h	'5'	35h	00110101
38	26h	'-'	2Dh	00101101
39	27h	'N'	4Eh	01001110
40	28h	'C'	43h	01000011
41	29h	'-'	2Dh	00101101
42	2Ah	'P'	50h	01010000
43	2Bh	'R'	52h	01010010
44	2Ch	'-'	2Dh	00101101
45	2Dh	'0'	30h	00110000
46	2Eh	String Table Size	1Bh	00011011

Figure 7: Part of an example Localised String Extension block

4. How KVM solutions affect EDID

When displays are connected to a host via a KVM (Keyboard, Video, and Mouse) switch, EDID data may be adversely affected based on how the data is processed by the device. There are three possible scenarios:

- No support – the KVM switch cannot handle the data and therefore, the host will not be able to determine any capabilities about the connected display. In some cases the host may assume that a generic monitor is attached and use "safe" settings.
- Fake EDID – the KVM switch generates the EDID data, which may not be appropriate for the connected display
- Pass-through – the KVM switch communicates with the display in order to obtain the EDID data and then sends this on to the host. However, in some circumstances this may confuse either the host by causing it to re-detect the display or confuse the display resulting in it entering or exiting from power-save mode.

Therefore, when performing any security testing against the processing of EDID data it is important that no KVM switch is present between the "display" and the target host.

5. Fuzzing EDID data

In order to fuzz EDID it is necessary to be able to emulate a display device to the extent that "valid" albeit malformed EDID data can be provided to the video display interface on a host via the E-DDC protocol using I^{2}C.

The easiest way to implement this is using a microcontroller that supports I^{2}C. In our research the Arduino Duemilanove microcontroller board was used, as I^{2}C is supported by default in its "Wire" library. The remaining challenges would be:

- Replicating the E-DDC protocol
- Repeatedly emulating the attachment and detachment of the "display"
- Iterating through the different elements of the EDID blocks

5.1. Replicating the E-DDC protocol

The "Wire" library provided with the Arduino only required minimal modification to support E-DDC. The changes required were as follows:

Within "wire.h" change:
```
25: #define BUFFER_LENGTH 64
```

to:

```
25: #define BUFFER_LENGTH 128
```

Within "utility/twi.h" change:

```
36: #define TWI_BUFFER_LENGTH 64
```

to:

```
36: #define TWI_BUFFER_LENGTH 128
```

The internal Arduino pull-up resistors need to be disabled in "utility/twi.cpp" (external 4.7KΩ pull-up resistors should be used instead):

```
70:    ////sbi(PORTC, 4);
71:    ////sbi(PORTC, 5);
```

Within the fuzzer the emulated display needs to be configured as an I^{2}C slave:

```
#define EDID_SLAVE 0x50
Wire.begin(EDID_SLAVE);
```

Interrupt handlers need to be configured for incoming messages and when data is wanted:

```
Wire.onReceive (receiveEvent);  // interrupt handler for incoming messages
Wire.onRequest (requestEvent);  // interrupt handler for when data is wanted
```

5.2.   Emulating the attachment and detachment of the "display"

This can easily be achieved with DVI, HDMI and DisplayPort interfaces as they all have a "hotplug" pin. If the pin is high (+5V) then the host assumes that a display is connected and if it's low (GND) then the host assumes the display has been disconnected. This can be performed programmatically on the Arduino as follows:

```
int hotplugpin=2; // use digital pin 2
digitalWrite(hotplugpin, LOW); // unplug the cable
digitalWrite(hotplugpin, HIGH); // plug in cable
```

Therefore, the "display" can be virtually unplugged and then plugged back in for each fuzz test case. However, a solution has not yet been developed for the VGA interface (where no hotplug detect pin exists).

5.3.   Iterating through the EDID elements

This was easily achieved by looping through any bytes of interest within each EDID block and modifying them based on different test cases.
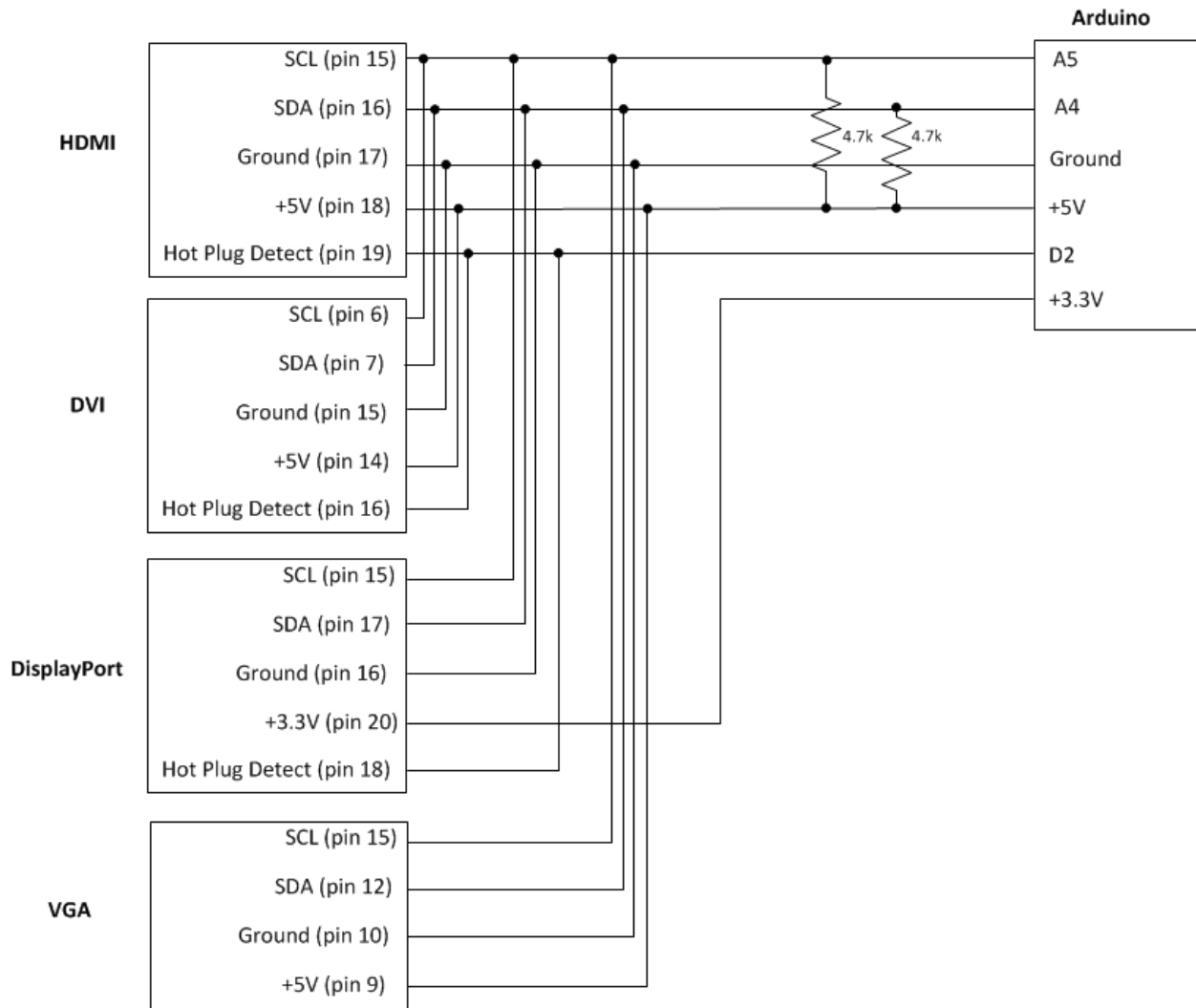
5.4. Fuzzer circuit diagram

Figure 8: Connectivity between an Arduino Duemilanove and the various display interfaces for fuzzing

5.5. The fuzzer firmware

The full source code of the fuzzer is included in Section 10 and can also be downloaded from the following URL:

http://www.ngssecure.com/research/research-overview/Public-Tools.aspx

5.6.   Findings so far

There have been a number of interesting findings so far, which imply that further investigation may yield more results. Hopefully the public release of this technique will result in a far greater range of products being tested.

5.6.1.   BlackBerry PlayBook

The original requirement to investigate EDID security arose from a separate research project looking at the BlackBerry PlayBook and the need to test the security of all available interfaces. The PlayBook has a microHDMI[23] port for mirroring the screen on an external display. After manually checking for buffer overflows associated with the Localised string extension block, the fuzzer was launched against the VESA block, using a brute-force approach to iterate through every value of every byte in the block and monitoring the state of the host. After a certain point, the host stopped responding to EDID requests and after further investigation the issue was identified. As a patch and associated advisory has not yet been released for this issue the details must remain confidential. However, the line of code that triggered the bug looked similar to this:

```
for (x = 0; x < 6; y++) {
…
}
```

N.B. The code wasn't written by RIM, but instead formed part of an open source library.

Needless to say, the "y++" should have been "x++" and as a result caused an infinite loop. The ability to perform a local Denial of Service attack against the HDMI port of a BlackBerry PlayBook is a pretty low impact discovery; however this bug subsequently triggered another bug which killed the system logger daemon and therefore, just by plugging in a "rogue display" an attacker could stop all logging prior to attempting other nefarious activities that would not be recorded in the system log.

5.6.2.   NVIDIA Windows 7 driver

A number of crashes associated with *nvlddmkm.sys* –the Windows Vista / Windows 7 kernel mode driver have been observed, however due to the complexities involved, none of these have yet been fully investigated.

6. CEC - Consumer Electronics Control

The intention of CEC is to enable users to control two or more HDMI-connected devices using a single remote control. The protocol also enables devices to control each other without user intervention. The architecture of CEC is an inverted tree with the "root" node (display) at the top, "branch" nodes (video switches) below and connected to "leaf" nodes (devices such as Blu-ray players). The concept is that the user should only need to perform the minimal number of tasks and let the "system" do the rest e.g. increasing the volume on the TV remote would identify which connected products need to be communicated with to ensure the volume of the current video /audio source increases.

CEC is a one-wire bidirectional serial bus that uses the AV.link[24] protocol for communication. Up to ten AV devices can be connected and the topology of a connected system is auto-discovered by the protocol. The messages that can be sent can be either devices-specific or general e.g. power control. Some examples of CEC commands are:

- One Touch Play: the device will become active source when playback starts
- System Standby: switches all connected devices to standby
- Preset Transfer: transfers the tuner channel setup to another TV set
- One Touch Record: start recording immediately
- Timer Programming: allow one device (e.g. a TV set or HTPC) to set the timer programming of another (e.g. a PVR,/DVR or DVD-recorder)
- System Information: checks all components for bus addresses and configuration
- Deck Control: playback control
- Tuner Control: control the tuner of another device
- OSD Display: use the OSD of the TV set to display text
- Device Menu Control: use the menus of another device
- Routing Control: control the switching of signal sources
- Remote Control Pass Through: pass through remote control commands
- Device OSD Name Transfer: transfer the preferred device names to the TV set

6.1. Block-level Protocol

Bits are grouped into 10-bit header and data blocks. Both header and data blocks include 8-bits of data along with EOM and ACK bits. The EOM bit signals the final block in a message. A '0' indicates that one or more blocks follow and a '1' indicates the message is complete. When a single follower provides an ACK to an initiator, it does so by "overriding" the output from the initiator (i.e. by pulling the bus to a logical '0' while the Initiator sends a "passive" logical '1'). Broadcast messages have special rules for handling simultaneous ACKs from multiple devices. Here, the logic is reversed and a group of followers ACK by not "overriding" the initiator (i.e. by allowing the Initiator to send a "passive" logical '1').

CEC devices have both physical and logical addresses. Normally, upon each hot-plug, each CEC source obtains a physical address by reading the EDID of the sink it is attached to. The physical address of each CEC device is expressed as four numbers and indicates where it is relative to the "root" display, whose address is always fixed at 0.0.0.0. For example, a source attached to input #1 of the "root" display, will have a physical address of 1.0.0.0 . Each CEC device also obtains a logical address - reflecting its product type – by negotiating with other CEC devices in the system. For example, the first set-top-box in the system is always given the logical address 3. Header blocks contain the 4-bit logical address of the Initiator and 4-bit logical address of the Destination in their data bit field.

6.2. Frame-level Protocol

CEC messages are sent using frames. Each CEC frame consists of a start bit, a header block and possibly data blocks. As an example, a message from a source device to a TV might display a text message on screen (On Screen Display – OSD). Such a message begins with a start bit, followed by a header block (with proper initiator /destination addresses), followed by data blocks containing an opcode 0x64 <set OSD string> and parameters to control the duration time and the text to be displayed. Each 10-bit block

(except the last one) will have the EOM set to '0', while the last block will have it set to '1'. Each block sent by an Initiator must have its ACK bit "overridden" by the destination device. If the destination is address 15, the message is deemed a "broadcast" and all devices may ACK by not overriding the Initiator's '1'.

Reliable communication is provided via frame retransmissions. If any block in a frame is not acknowledged or other bus errors exist, initiators will sense the condition and may retransmit up to five times. When destination devices withhold their ACKs, initiators retransmit. Since the CEC bus is a single wire, bus arbitration is very important. The CEC specification calls for a signal free time before sending. To allow other devices a chance to send, the time for a current initiator to send another frame is longer than that of a new initiator that wants to send a frame, and signal-free times for retransmissions are the shortest. If multiple devices try to send a message at the same time, a priority scheme is used to give a single initiator precedence.

(The CEC technical information was obtained from a paper written by Quantum data[25])

6.3.   Fuzzing CEC

As CEC is a feature rich protocol it could potentially yield some interesting security vulnerabilities in different implementations. There is an Arduino library[26] for communicating with HDMI CEC equipment; the library requires the circuit in Figure 9.
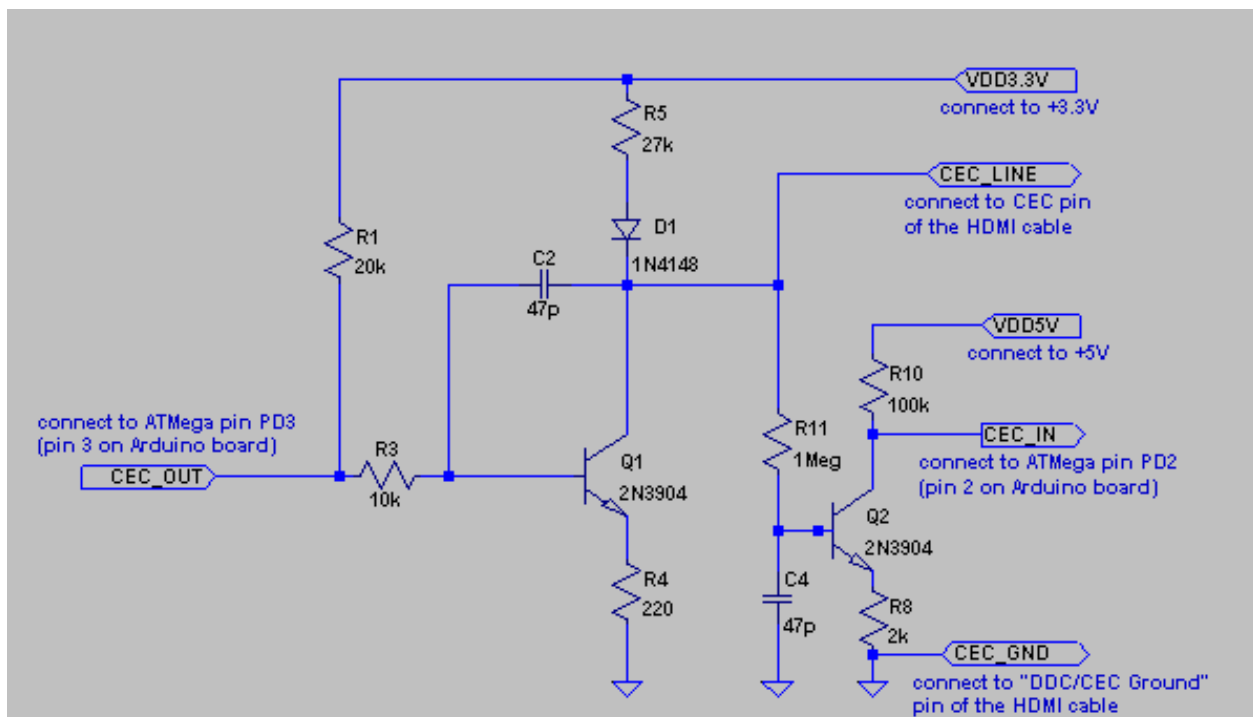

Figure 9: Circuit for interfacing an Arduino microcontroller to CEC

There are also commercial CEC emulation products, such as the "USB-CEC Adapter"[27] from Pulse Eight (this uses libCEC[28] to generate and capture valid CEC traffic):

Figure 10: USB-CEC Adapter (Pulse Eight)

There is also, the HDMI-CEC to USB and RS-232 bridge/converter[29], from RainShadow Technology.

An NGS Secure CEC fuzzer (based on the Pulse Eight product and libCEC) is currently in development, but is not yet ready for public release.

7. HEC - HDMI Ethernet Channel

HDMI Ethernet Channel was introduced in HDMI v1.4, it consolidates video, audio, and data streams into a single HDMI cable. The Ethernet channel is physically combined with audio return functionality onto a single pair of wires (or pins HEAC+ and HEAC-) on the HDMI connector. This single pair of wires can carry Ethernet only, audio only, or both simultaneously. The primary intention of HEC is to reduce the amount of cables required to connect AV devices together.

With more and more AV devices with embedded OS's communicating with the Internet and with each other using protocols such as UPnP[30], DLNA[31] and LiquidHD[32] this will result in these mini networks becoming targets for malware which may result in them joining botnets in the future.

8. HDCP - High-bandwidth Digital Content Protection

High-bandwidth Digital Content Protection (HDCP) is a form of digital copy protection developed by Intel Corporation[33] to prevent copying of digital audio and video content as it travels across connections. It is included in this paper for completeness.

HDCP is designed to prevent encrypted video and audio content from being displayed on unauthorised (non-HDCP compliant) devices, which could potentially copy the digital content to a non-encrypted form. Before sending data, a transmitting device e.g. a Blu-ray player checks that the receiver is authorised to receive it (is HDCP compliant). If so, the transmitter encrypts the data to prevent eavesdropping as it is communicated to the receiver.

HDCP uses three cryptographic mechanisms:

- Authentication prevents non-licensed devices from receiving content

- Encryption of the data prevents eavesdropping of information and man-in-the-middle attacks
- Key revocation prevents devices that have been compromised from receiving data.

Although this research did not result in any new techniques for attacking HDCP, in September 2010, an HDCP master key that allows for the generation of valid device keys was publicly disclosed[34] which effectively renders the key revocation feature of HDCP useless. There is anecdotal evidence[35] that the key was reverse engineered rather than leaked, as vendors in the business of making HDCP-compatible devices, who have access to many individual device keys, would be able to reconstruct the master key by analysing "mathematical similarities" in the individual device keys.

9. Conclusions and further research

Many people are completely unaware that video displays send data which is subsequently processed by the connected device (laptop, PC, tablet-computer) and this research has shown that this received-data can potentially contain threats.

As users demand more and more "seamless" functionality in a plug-and-play world there will be a greater need for bi-directional data to be flowing in A/V links between devices. This paper has discussed the EDID data that displays send to communicate their capabilities to a host, the CEC protocol used in HDMI to allow A/V components to communicate with each other about configuration actions that need to be performed and HEC which provides Ethernet access via the HDMI interface. All of these technologies provide scope for an attacker to exploit any vulnerabilities present in the software that processes these protocols.

Further areas for research for NGS Secure include:

- The investigation of more creative datasets of EDID block combinations using the Block Map standard
- Adding more "intelligence" to the EDID fuzzer so that it's more than just a "bit flipper"
- Identifying a technique to emulate a VGA cable insertion/removal
- Completing the CEC fuzzing project and using it to test a representative sample of hosts

This paper provides a useful introduction to the technologies involved in video interfacing, the potential for security vulnerabilities and possible ways to test for their presence. However, this introduction has only scratched the surface of these technologies and how they may potentially be attacked. By releasing details of the EDID fuzzer and upcoming CEC fuzzer it is hoped that this research has raised awareness of some of the issues and will encourage others to perform research in this area.

10. Fuzzer Firmware Source Code

```
//////////////////////////////////////////////////////////////////////////////
// EDID fuzzer v1.0
// By Andy Davis, NGS Secure
// March 2012
//
// This is a very simple example of a fuzzer to iterate through the bytes within an
// EDID block and send the EDID data emulating the connection of a display device
//
// This code has been successfully tested on the Arduino Duemilanove microcontroller
//
// Note: Internal pull-up resistors should be disabled in twi.cpp:
// //sbi(PORTC, 4);
// //sbi(PORTC, 5);
//
// 4.7k external pull-up resistors should be used
//
//////////////////////////////////////////////////////////////////////////////

#include <Wire.h>

#define EDID_SLAVE 0x50

int hotplugpin=2;
int i=0;
int a=0;
int count=1;
int z = 0;

// EDID v1.3 block

byte EDIDArray[128]={
0x00,0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,0x00,0x34,0xA9,0x01,0xC3,0x01,0x01,0x01,0x01,
0x00,0x14,0x01,0x03,0x80,0x00,0x00,0x78,0x0A,0xDA,0xFF,0xA3,0x58,0x4A,0xA2,0x29,
0x17,0x49,0x4B,0x00,0x00,0x00,0x01,0x01,0x01,0x01,0x01,0x01,0x01,0x01,0x01,0x01,
0x01,0x01,0x01,0x01,0x01,0x01,0x02,0x3A,0x80,0xd0,0x72,0x38,0x2d,0x40,0x10,0x2c,
0x45,0x80,0xba,0x88,0x21,0x00,0x00,0x1E,0x02,0x3A,0x80,0x18,0x71,0x38,0x2D,0x40,
0x58,0x2C,0x45,0x00,0xBA,0x88,0x21,0x00,0x00,0x1E,0x00,0x00,0x00,0xFC,0x00,0x50,
0x61,0x6E,0x61,0x73,0x6F,0x6E,0x69,0x63,0x2D,0x54,0x56,0x0A,0x00,0x00,0x00,0xFD,
0x00,0x17,0x3D,0x0F,0x44,0x0F,0x00,0x0A,0x20,0x20,0x20,0x20,0x20,0x20,0x01,0x53
};

// EDID v1.4 block
/*
byte EDIDArray[128]={
0x00,0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,0x00,  //EDID header
0x04,0x43, //Manufacturer's Code Name
0x08,0xF2, //ID Product Code
0x01,0x00,0x00,0x00, //ID Serial Number
0x10, //Week of Manufacture is 16
0x11, //Year of Manufacture is 2007
0x01, //"1" EDID Version Number
0x04, //"4" EDID Revision Number
0xA2, //Digital Video Input using HDMI-a, 8 Bits per Primary Colour
0x79, //Aspect Ratio is 16 : 9 AR in Landscape
0x44, //Aspect Ratio (Landscape) Flag
0x78, //Display Gamma is 2.20
0x1E, //Feature Support Byte
0xEE,0x91,0xA3,0x54,0x4C,0x99,0x26,0x0F,0x50,0x54, //Display x,y Chromaticity Coordinates
0xFF, //Established Timings I
0xEF, //Established Timings II
0x80, //Manufacturer's Timings
0x81,0x99,0x81,0x80,0x81,0x59,0x81,0x40,0x61,0x59,0x4B,0xC0,0x45,0x59,0x31,0x59,      //Standard
Timings
```

```
//First 18 Byte Data Block
0x66,0x21, //Pixel Clock
0x50,0xB0,0x51, //Horizontal Addressable Video
0x00,0x1B,0x30, //Vertical Addressable Video
0x40,0x70,0x36,0x00, //Horizontal Front Porch
0xBE,0xAB,0x42, //Horizontal Addressable Image Size
0x00,0x00, //Horizontal Border Size
0x1E, //Timing is Non-Interlaced Video

//Second 18 Byte Data Block
0x01,0x1D, //Pixel Clock
0x00,0x72,0x51, //Horizontal Addressable Video
0xD0,0x1E,0x20, //Vertical Addressable Video
0x6E,0x28,0x55,0x00, //Horizontal Front Porch
0xBE,0xAB,0x42,
0x00,0x00, //Horizontal Border Size
0x1E, //Timing is Non-Interlaced Video

//Third 18 Byte Data Block
0x00,0x00,0x00, //Established Timings III Descriptor
0xF7, //Established Timings III Block Tag
0x00, //padding
0x0A,0xF7,0x0F,0x03,0x87,0xC0,0x00,0x00,0x00,0x00,0x00,0x00,0x00, //VESA DMT Standard Version #10

//Fourth 18 Byte Data Block
0x00,0x00,0x00, //Monitor Name ASCII Descriptor
0xFC, //Monitor Name Tag
0x00, //padding
0x41,0x42,0x43,0x20,0x50,0x4C,0x41,0x35,0x35,0x0A,0x20,0x20,0x20,
0x01, //Extension flag
0x0A}; //Checksum


//EIA/CEA-861 extension block
/*
byte EDIDArray2[128]={
0x02,0x03,0x22,0x72,0x50,0x9F,0x90,0x14,0x05,0x20,0x13,0x04,0x12,0x03,0x11,0x02,
0x16,0x07,0x15,0x06,0x01,0x23,0x09,0x07,0x01,0x68,0x03,0x0C,0x00,0x30,0x00,0xB8,
0x26,0x0F,0x01,0x1D,0x80,0xD0,0x72,0x1C,0x16,0x20,0x10,0x2C,0x25,0x80,0xBA,0x88,
0x21,0x00,0x00,0x9E,0x01,0x1D,0x80,0x18,0x71,0x1C,0x16,0x20,0x58,0x2C,0x25,0x00,
0xBA,0x88,0x21,0x00,0x00,0x9E,0x01,0x1D,0x00,0xBC,0x52,0xD0,0x1E,0x20,0xB8,0x28,
0x55,0x40,0xBA,0x88,0x21,0x00,0x00,0x1E,0x01,0x1D,0x00,0x72,0x51,0xD0,0x1E,0x20,
0x6E,0x28,0x55,0x00,0xBA,0x88,0x21,0x00,0x00,0x1E,0x8C,0x0A,0xD0,0x90,0x20,0x40,
0x31,0x20,0x0C,0x40,0x55,0x00,0xBA,0x88,0x21,0x00,0x00,0x18,0x00,0x00,0x00,0xEA
};
*/

//Localised string extension block

byte EDIDArray2[128]={
0x50, //Extension ID
0x01, //LS-Ext Version number
0x00, //LS-Ext Revision number
0x32, //Unicode Version – Major/Minor
0x00, //Unicode Version – Update

0x28, //String table size
0x00, //String Table Header – UTF 8
0x00, //Language ID structure – Default Neutral String Table
0x00,
0x00,
0x00,
0x0e, //Manufacturer name length
0x44,0x69,0x73,0x70,0x6C,0x61,0x79,0x73,0x2C,0x20,0x49,0x6E,0x63,0x2E,
0x06, //Model Name String Length
0x46,0x43,0x31,0x39,0x30,0x31,
```

```
0x0C, //Serial Number Data String Length
0x30,0x33,0x32,0x35,0x2D,0x4E,0x43,0x2D,0x50,0x52,0x2D,0x30,

0x1B, //String Table Size
0x01, //String Table Header - UTF 16BE
0x00, //Language ID Structure – JPN/392
0xC4,
0x2A,
0x0E,
0x14, //Manufacturer ID Name Length
//0x14, //Manufacturer ID Name Length
0xC7,0x30,0xA3,0x30,0xB9,0x30,0xD7,0x30,0xEC,0x30,0xA4,0x30,0x2A,0x68,0x0F,0x5F,
0x1A,0x4F,0x3E,0x79,
0x00, //Model Name String Length
0x00, //Serial Number String Length

 //Padding...
0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
0x00,0x00,0x00,0x00,

0x5A, //Checksum
};

// Display Information Extension DI-EXT block
/*
byte EDIDArray2[128]={
0x40 ,  //Block Header: Value must be "40h".(See Section 3.1.1)
0x01 ,  //DI-EXT Version Number: 1 (See Section 3.1.2)
0x05 ,  //Digital Visual Interface (DVI) for Consumer Electronics
0x80 ,  //EIA0xCEA 8610xA has a Letter Designation for the Version.
0x41 ,  //ASCII Code indicates letter "A" is the Version. (See Section 3.2.2)
0x00 ,  //There is no Revision Number. (See Section 3.2.2)
0x00,
0xDC ,  //'11011100' Digital Interface Data Format Description: (See Section 3.2.3)
0x24 ,  //Digital interface uses the Standard Data Format: 24-Bit MSB-Aligned
0x19 ,  //Minimum Pixel Clock Frequency is 25 MHz. (See Section 3.2.4)
0x70 ,  //Maximum Pixel Clock Frequency is 112 MHz (0070h).
0x00 ,  //(LSB first) (See Section 3.2.4)
0x00 ,  //This is a Single Link DVI monitor. There is no Crossover Frequency.
0x00 ,  //(LSB first) (See Section 3.2.4)
0x00 ,  //Sub-Pixel Layout is not defined. Display is a CRT. (See Section 3.3.1)
0x02 ,  //Sub-Pixel Configuration is "Stripe". CRT is an Aperture Grill.
0x00 ,  //Sub-Pixel Shape is not defined. Display is a CRT. (See Section 3.3.1)
0x1F ,  //Horizontal Pixel Pitch is 0.31 mm. (See Section 3.3.2)
0x00 ,  //Vertical Pixel Pitch is 0.0 mm CRT is an Aperture Grill(See Section 3.3.2)
0x26 ,  //'00100110' Major Display Device Characteristics: (See Section 3.3.3)
0x01 ,  //'00000001' Miscellaneous Display Capabilities: (See Section 3.4.1)
0x00 ,  //'00000000' Frame Rate Conversion: (See Section 3.4.2)
0x00 ,  //Vertical Frame Rate Conversion Frequency: (See Section 3.4.2)
0x00 ,  //'0000h' => Vertical Frame Rate Conversion Frequency is not available.
0x00 ,  //Horizontal Frame Rate Conversion Frequency: (See Section 3.4.2)
0x00 ,  //'0000h' => Horizontal Frame Rate Conversion Frequency is not available.
0x42 ,  //'01000010' Display0xScan Orientation: (See Section 3.4.3)
0x01 ,  //'01h' => Default Color0xLuminance Decoding is BGR (additive colour).
0x03 ,  //'03h' => Preferred Color0xLuminance Decoding is Yxx per (SMPTE
0x8F ,  //'10001111' Color0xLuminance Decoding Capabilities include:
0xE0 ,  //'11100000' • BGR (additive colour)
0x00 ,   //'00000000' Monitor Colour Depth: (See Section 3.4.5)
0x08 ,  //BGR Monitor Colour Depth is 8 bits for colour blue on Sub-Channel 0.
0x08 ,  //BGR Monitor Colour Depth is 8 bits for colour green on Sub-Channel 1.
0x08 ,  //BGR Monitor Colour Depth is 8 bits for colour red on Sub-Channel 2.
0x08 ,  //YCrCb Monitor Colour Depth is 8 bits for Cb Sub-Channel 0.
0x08 ,  //YCrCb Monitor Colour Depth is 8 bits for Y on Sub-Channel 1.
0x08 ,  //YCrCb Monitor Colour Depth is 8 bits for Cr on Sub-Channel 2.
0xF0 ,  //'11110000' Aspect Ratio Conversion: The display supports Full Mode ,  //Zoom Mode,
```

```
0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,

0x53};
*/

void setup()
  {
  pinMode(hotplugpin, OUTPUT);  // initialize as output
  digitalWrite(hotplugpin, LOW); // unplug the cable

  Wire.begin(EDID_SLAVE);        // join i2c bus, slave mode

  Wire.onReceive (receiveEvent);  // interrupt handler for incoming messages
  Wire.onRequest (requestEvent);  // interrupt handler for when data is wanted

  Serial.begin(9600);              // start serial for output - the output is displayed via the
built-in Arduino serial interface

  digitalWrite(hotplugpin, HIGH); // plug in cable
  }

void loop()
  {

  byte y = 0;
  int x = 0;
  byte temp = 0;

  while (z < 128)
    {
    y = 0;
    while (y < 1)
      {
      temp = EDIDArray[z]; // save off current value

      EDIDArray[z] = 0xff; // modify each value in turn to be 0xff (use y to iterate through each
value per byte)

      digitalWrite(hotplugpin, LOW); // unplug the cable
      delay (10);
      digitalWrite(hotplugpin, HIGH); // plug in cable
      delay (1500);
      Serial.print(z);
      Serial.print(":");
      x = y;
      Serial.print(x);
      Serial.print("\n");

      EDIDArray[z] = temp; // restore value to structure
      y = y + 1;
      }
    z = z + 1;
    }

  }

void receiveEvent(int howMany)
  {

  while(Wire.available()) // loop through all but the last
    {
```

```
    int x = Wire.receive();    // receive byte as an integer
    Serial.println(x);         // print the integer
    }
  int x = Wire.receive();    // receive byte as an integer

  }

void requestEvent()
  {

  ////// fix up checksums /////

  int c = 0;
  int d = 0;
  byte csum = 0;
  for (int x = 0; x < 127; x++)
    csum += EDIDArray[x];
  c = csum;
  d = 256 - c;
  EDIDArray[127] = d;

  c = 0;
  d = 0;
  csum = 0;
  for (int x = 0; x < 127; x++)
    csum += EDIDArray2[x];
  c = csum;
  d = 256 - c;
  EDIDArray2[127] = d;

  if (count==1)
    {
    Wire.send(EDIDArray,128);          // !!Need to change BUFFER_LENGTH to 128 in wire.h and
TWI_BUFFER_LENGTH to 128 in twi.h
    count = 2;
    }
  else
    {
    Wire.send(EDIDArray2,128);
    count = 1;
    }

  }
```

11. References and further reading

1 - http://en.wikipedia.org/wiki/Vertical_sync#Horizontal_synchronization

2 - http://en.wikipedia.org/wiki/Vertical_sync#Vertical_synchronization

3 - http://www.vesa.org/

4 - http://www.vesa.org/vesa-standards/standards-summaries/

5 - http://www.ddwg.org/

6 - http://www.ddwg.org/lib/dvi_10.pdf

7 - http://xtreamerdev.googlecode.com/files/CEC_Specs.pdf

8 - http://en.wikipedia.org/wiki/High-bandwidth_Digital_Content_Protection

9 - http://www.hdmi.org/manufacturer/hdmi_1_4/hec.aspx

10 - http://en.wikipedia.org/wiki/DisplayPort

11 - http://en.wikipedia.org/wiki/Extended_display_identification_data

12 - http://www.i2c-bus.org/

13 - ftp://ftp.cis.nctu.edu.tw/pub/csie/Software/X11/private/VeSaSpEcS/VESA_Document_Center_Monitor_Interface/EDDCv1r1.pdf

14 - http://en.wikipedia.org/wiki/Extended_display_identification_data

15 - http://en.wikipedia.org/wiki/CIE_1931_color_space

16 - ftp://ftp.cis.nctu.edu.tw/pub/csie/Software/X11/private/VeSaSpEcS/VESA_Document_Center_Monitor_Interface/DMTv1r11.pdf

17 - ftp://ftp.cis.nctu.edu.tw/pub/csie/Software/X11/private/VeSaSpEcS/VESA_Document_Center_Monitor_Interface/gtfv1_1.pdf

18 - ftp://ftp.cis.nctu.edu.tw/pub/csie/Software/X11/private/VeSaSpEcS/VESA_Document_Center_Monitor_Interface/VTBextRa.pdf

19 - ftp://ftp.cis.nctu.edu.tw/pub/csie/Software/X11/private/VeSaSpEcS/VESA_Document_Center_Monitor_Interface/DIEXT.pdf

20 - ftp://ftp.cis.nctu.edu.tw/pub/csie/Software/X11/private/VeSaSpEcS/VESA_Document_Center_Monitor_Interface/LSExtRa.pdf

21 - ftp://ftp.cis.nctu.edu.tw/pub/csie/Software/X11/private/VeSaSpEcS/VESA_Document_Center_Monitor_Interface/EEDIDrAr2.pdf

22 - ftp://ftp.cis.nctu.edu.tw/pub/csie/Software/X11/private/VeSaSpEcS/VESA_Document_Center_Monitor_Interface/LSExtRa.pdf

23 - http://www.hdmi.org/manufacturer/hdmi_1_4/micro_connector.aspx

24 - http://en.wikipedia.org/wiki/AV.link

25 - http://www.hdmi.org/pdf/whitepaper/DesigningCECintoYourNextHDMIProduct.pdf

26 - http://code.google.com/p/cec-arduino/

27 - http://www.pulse-eight.com/store/products/104-usb-hdmi-cec-adapter.aspx

28 - https://github.com/Pulse-Eight/libcec

29 - http://rainshadowtech.com/default_files/HDMICECUSB.htm

30 - http://www.upnp.org/

31 - http://www.dlna.org/

32 - http://en.wikipedia.org/wiki/LiquidHD

33 - http://en.wikipedia.org/wiki/High-bandwidth_Digital_Content_Protection

34 - http://arstechnica.com/tech-policy/news/2010/09/intel-confirms-the-hdcp-key-is-real-can-now-be-broken-at-will.ars

35 - http://www.wired.com/threatlevel/2010/09/intel-threatens-consumers/