DAN Framestore: Technical

Richard Black

15th September 1995

Abstract

This document is an attempt, after the fact, to document the Framestore node of the Desk Area Network. The details in this document have been derived from reverse engineering the working software, so although believed correct may contain residual errors.

1 Introduction

The DAN Framestore is a rather unusual device. Its purpose from a logical and philosophical and research point of view is described elsewhere. This document is about the grubby details.

Apart from being attached to the DAN there is another thing about the DFS (DAN Frame-Store) which is unusual. There is a per-pixel capability which indicates which owner is capable of updating that pixel. This means that the DFS is a *User Safe Device*, a device which can be accessed directly from many mutually suspicions programs without the necessity of a shared server on the data path. This is a benefit for reducing quality of service crosstalk within the system.

1.1 Hardware Components

The DFS is a standard DAN shaped board, viz. a double height extended depth eurocard (about 9" square). The major components are listed in table 1.

The PCB is four layer, with all signal traces on the top and bottom surfaces. This leads to quite a densely packed PCB. Unfortunately, due an unclear section in the Bt463 data-book, the pixel bus is connected in a moulinexed fashion. This has led to a number of differences between the two boards which have been made up, since only one of them has had the 64 patches necessary to fix this. The differences are summarised in table 2. If it is necessary to identify in software which board is in use, this can be done by using the difference in the process version register.

1.2 The Pixel bus

The VRAM has 16 bits of data and 8 bits of tag/mask per pixel. The Bt463 has a 32 bit input bus for pixel, window and overlay data. The pixel bus is four pixels wide in parallel to achieve sufficient bandwidth for a high quality display. As already mentioned, the pixel bus is connected differently on the two boards **A** and **B**. The arrangement is shown in tables 3 and 4 respectively. Unconnected inputs are grounded.

The 8 mask bits are used for:

[1] Implemented using a conventional CMOS oscillator and an MC10116 ECL buffer.

Number	Component	Purpose
12	NEC D482234LE-70 VRAM	Video, Capability, and program store.
1	Bt463 RAMDAC	Video pixel generator.
1	Bt438	Video clock distributor.
1	LM1882	Video sync generator.
1	ECL Oscillator[1]	Video clock generation.
3	Analogue gloop	Noise suppression on Red, Green and Blue.
1	Arm 710[A]	DMA Engine and miscellaneous services.
1	Oscillator	Processor and Memory clocks.
1	MACH210-12	Processor and Memory clocks.
4	MACH220-12?	Miscellaneous PALs.
4	CY7C453	Receive network interface fifos.
1	CY7C451	Transmit network interface fifo.
8	LED	Debugging usage.
5	LED	Fixed purpose indicators.

Table 1: DFS Major Components

Board	MAX Visual depth	ARM	FCLK	MCLK	Header or Empty Patch
A	15	710	50	12.5	Yes
B	8	710A	20	10	No

Table 2: DFS Differences

1. Software cursor
2. Visual selection
3. Pixel Capability

Of these, 2, and 3 must be orthogonal, 1 must use the top four bits and 1 and 2 must also be orthogonal. The design splits these functions as follows. The pixel capability uses bits 0 to 5. The software cursor uses bits 6 and 7. The visual selection therefore uses bits 4 and 5. Thus two of the bits of a window's capability are defined by the visual class of which it is a member. The software currently assigns visuals to this class as follows.

0 Pseudo colour 1

1 Pseudo colour 2

2 True colour 15 bit

3 (unassigned)

<table>
<tr><th></th><th colspan="4">Window Type</th><th colspan="28">Pixel Bit</th></tr>
<tr><th></th><th>3</th><th>2</th><th>1</th><th>0</th><th>27</th><th>26</th><th>25</th><th>24</th><th>23</th><th>22</th><th>21</th><th>20</th><th>19</th><th>18</th><th>17</th><th>16</th><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td>Mask</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>Data</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td></td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td>15bit</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td colspan="5">Blue</td><td></td><td></td><td></td><td colspan="5">Green</td><td></td><td></td><td></td><td colspan="5">Red</td></tr>
<tr><td>8bit</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td colspan="8">Pseudo Hi</td><td></td><td></td><td></td><td></td><td></td><td></td><td colspan="8">Pseudo Lo</td></tr>
</table>

Table 3: Pixel bus, board A

	Window Type				Pixel Bit																											
	3	2	1	0	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Mask	7	6	5	4	3	2	1	0																								
Data																	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
8bit																	Pseudo Hi								Pseudo Lo							

Table 4: Pixel bus, board B

Two bits for the software cursor gives three programmable colours plus see-through for the normal display.

2 Memory map

The memory map of the board is rather unusual to take care of the manner in which it is booted (see section 4). This section describes the physical memory map, the virtual memory map will be configured somewhat differently as described in section 4.

Since the ARM powers up in 26bit compatible mode, and physical address space is not at a premium, the board does not use any address bits in the range 26-31. Major decoding is done on address bits 23 to 25 as shown in table 5. This gives 8 Megabyte sections between functions.

A25..23	Address	Purpose
000	0x00000000	Registers
001	0x08000000	Pixel / Code VRAM
010	0x10000000	Mask VRAM
011	0x18000000	Pixel Transfer (PT) Fifo
100	0x20000000	Sam Transfer
101	0x28000000	PT cpu
110	0x30000000	(unused)
111	0x38000000	Registers

Table 5: Physical Memory Address Decoding

The register area (which is available at two separate locations) is further decoded by access to different registers on various devices on the board. Access methods for different registers may be different; see the relevant section. Decoding is based on address bits 9 to 11 giving the full range in a single 4K page, with 512 bytes for each register. Software should not attempt to perform sequential cycles which cross from one register to another.

2.1 Accessing the VRAM

The VRAM (both code/data and also the mask/capability) is organised in two banks, with address bit 2 being used to select between them.

Note that address bits 0 and 1 are not connected on the board, so byte cycles will in fact affect the entire word. Software should avoid performing byte **write** *cycles* (unless the author knows what the Arm is going to do and is relying on that behaviour which is considered tacky). Byte read cycles are fine because the Arm will do the extraction.

The VRAM has 16 bits of data and 8 bits of tag per pixel. The two different types are accessible at completely different addresses. Pixels are grouped together in pairs to each word.

A11..9	Address	Purpose
000	0x0000	RXFIFO
001	0x0200	LM1882
010	0x0400	BT463
011	0x0600	STATUS
100	0x0800	TXFIFO (and RXFIFO non-blocking)
101	0x0a00	CMDREG
110	0x0c00	MASK
111	0x0e00	LEDS

Table 6: Register access decoding

The system is equipped with enough RAM for one million pixels (i.e. 2 MB of memory). It is intended that this be organised as a 1024×864 display, with the additional RAM being used for software. Such an organisation leads to a 2Kbyte scan line.

Each VRAM chip provides 256×8 bits of memory. This represents 9 bits of RAS/CAS. The RAS is supplied by Arm address bits 11 to 19, and the CAS by bits 3 to 10 and bit 20. This means that there is a requirement to perform a non-sequential cycle on a 2Kbyte boundary (consistent with the size of the scan line).

Fortunately for software (e.g. stacks) the Arm MMU has a smallest protection granularity of 1Kbyte. Even though the page table entries may be for much larger granularity mappings, the Arm MMU will insert a non-sequential cycle on any access which crosses a 1Kbyte boundary to enable the check against the TLB to be performed (the TLB is not checked on other sequential accesses). This is totally undocumented behaviour of the Arm. The non-sequential cycle inserted on the inside is not removed by the write buffer.

The DFS hardware relies on this behaviour to ensure that the VRAM timings are kept. No external checking is performed on address changes.

Note that the 9 bits of column with two 32 bit wide banks leads to 4 Kbytes per VRAM row. It is quite deliberate that address bit 20 is used as the most significant bit of the column address. Since loading the SAMs with each scan line will cause a row operation it will refresh two scan line amounts of the memory. The fact that this is split on the most significant address bit for VRAM access means that (since the vertical size is greater than 512) the contiguous off-screen memory used for software is refreshed by the action of displaying the video image.

This arrangement assumes that vertical refresh is sufficiently rapid that memory will be correctly refreshed. If use of unexpected video signals is made, then the effect on RAM refresh should be carefully considered. Refresh should remain in spec right down to 50Hz PAL signals.

2.2 SAM Transfer

Software is responsible for reloading the SAMs at the end of each scan line. The hardware generates a FIQ interrupt to the CPU at the start of horizontal blanking (when the LM1882 indicates start of horizontal sync). The CPU has $1.7\mu s$ to reload the SAMs with the new scan line. This is performed by *writing* to the base address of the required scan line in the SAM transfer memory region. The data written is ignored. The operation of the write also clears the interrupt and refreshes the two scan lines worth of memory in that row.

Since handling of FIQ interrupts may be delayed due to completion of burst cycles the SAM transfer operation should be completed as soon as possible to minimise the probability of disruption on the video output. Since the operation occurs very frequently (e.g. once every $17\mu s$), the FIQ code should be written to be as fast as possible, however, it has been observed in practice that the write buffer should be *disabled* for the SAM transfer region, since otherwise the Arm could attempt to handle the interrupt again whilst the request is in the write buffer.

Software should not attempt to read from the SAM transfer region.

2.3 Registers

Some of the components attached to the board have rather peculiar or slow timings, having come from other applications (such as television sets). For such devices software may be responsible for ensuring that setup and hold times are met by performing multiple identical sequential cycles.

To reduce bus loading, the various 8 bit devices are connected to differing byte "lanes" on the bus. Similarly, where devices have internal address spaces, the address bits supplied are also different to reduce address bus loading. These must also be taken into account when performing accesses.

No other type of access should be performed to any of these registers other than those described herein, otherwise damage may result. Likewise no attempt should be made to write to read status registers as a bus clash may result.

2.3.1 Bt463

The Bt463 contains many of the control registers to do with the colour and pixel bus management. It is an eight bit wide device connected to lane 1 (data bus bits 8-15). There are only two address bits connected externally on the device. These are connected to Arm address bus lines 2 and 3. The offsets from the base of the Bt463 are thus shown in table 7.

Offset	Register
0x00	Address low
0x04	Address high
0x08	Command
0x0C	Palette

Table 7: Bt463 register addressing

Two of the four addressable registers contain must be programmed with the internal address required, and the other two allow writing to the command and palate space respectively. See the Bt463 data-sheet for further details.

Writing to the Bt463 requires an extended cycle, as the chip samples the data late with respect to the bus timings. This should be performed by executing an increment-after store-multiple instruction to the required base address, with two identical words of data.

When reading from the Bt463, again the data arrives late with respect to the address. An increment-after load-multiple instruction should be used starting at the required base address, and the second word of data loaded is considered to have more accurate data.

2.3.2 LM1882

The LM1882 is attached to lane 3 (data bits 24-31) and the two control signals DATAnADDR and HIGHnLOW are attached to address bits 8 and 7 respectively.

The device has four internal addressable ports as shown in table 8. Internal registers may be between 8 and 16 bits wide, and the chip contains support for automatic indexing of the internal space. See the LM1882 data-sheet for full details. Note that to obtain the normal 901 scan line display (for 1024×864) the VMAX register should be programmed with one greater (902) which is not made clear in that data-sheet.

No special timing requirements are needed for software access to this device.

Offset	Register
0x000	Address
0x080	Automatic
0x100	Low
0x180	High

Table 8: LM1882 register addressing

2.3.3 LEDS

The eight debugging LEDS are latched in a register which is connected to lane 1 (data bits 8 to 15). A "one" value represents "on". At reset this register is zero. Accesses to this register do not require any special software operations.

2.3.4 MASK

The Mask register must be loaded in software to allow hardware clipping to take place whilst updates to the VRAM buffer take place. It is six bits wide connected to data bits 8 to 13. Accesses to this register do not require any special software operations.

2.3.5 Command Register

The command register provides two control bits, connected to arm data bits 8 and 9. It gives control over the single cell transmit interface. The commands are pulsed outputs which are automatically clearing and do not need to be reset in software. If bit 9 is set, the transmit machinery and fifo is reset. If bit 8 is set the transmit machinery is activated for transmitting the cell in the transmit fifo to the fabric.

2.3.6 Transmit Fifo

The transmit fifo should be loaded with a fabric sized (54 byte) cell for transmission to the fabric. It is connected to lane 0 (bits 0 to 7). For performance reasons, burst cycles are supported to this device.

2.3.7 Status Register

The status register is connected to data bits 24 to 29. The meanings of the bits are described in table 9. The vgate signal comes from the LM1882 and is programmable.

Bit	Meaning
24	Receive fifo not empty
25	vgate signal
26	Horizontal sync
27	Special fifo empty bit which is always 1 (see later)
28	Last TX cell went O.K.
29	Spare (currently connected to OddnEven)

Table 9: Status register contents

2.3.8 Receive FIFO

The receive fifo is 32 bits wide. There are two ways of accessing the FIFO in the register space (and a further way in the transfer region - see later). At the RXFIFO base, accesses to the FIFO are blocking. The Arm's wait line will be held if the fifo is empty, until data is present. At the non-blocking address (equal to the TXFIFO offset), an access will return a word of data if present in the fifo, or else a status register read if there is no data.

This non-blocking access is intended for headers only. Since valid headers will never have bit 27 set, it is possible for software to determine both whether there is another cell in the FIFO, and if so to read its header, in a single access.

2.4 PT fifo

The PTfifo region is the reason for the existence of the DFS. This mode is used for transferring screen updates from the receive FIFOs to the picture VRAM. In this region the Arm is used solely as an address generator. Accesses to this region cause the Arm's databus to be disabled and the FIFOs will drive the bus instead. No read cycles should be preformed to this memory area as they may cause a bus-clash between the FIFOs and the VRAM.

For each address, the MASK VRAM is read and compared with the MASK register. If the values are equal then the pixel VRAM is updated with the video data read from the FIFOs. This is done on a pixel rather than a word basis by using different enable lines for the two pixels in each word. Because of the compare stage, there is a one cycle pipeline and so the Arm will have wait asserted for the cycle following the transfer.

For performance reasons burst sequential cycles are supported (and should be used) in this memory region.

2.5 PT cpu

The PTcpu region is the same as the PTfifo region with the exception that the Arm's data bus is not disabled, and the data supplied for comparison and selective writing is provided by the Arm and not the FIFOs. The purpose of this mode is to allow the Arm software to perform block copy operations for moving a window around the screen.

It *may* be possible to support read cycles in the PTcpu region, to allow protected read of pixels. This mode has not been tested. It is did work, software would obtain whatever was floating on the bus for pixels on which the capability did not match.

3 Network Interface

The network interface is a key component to the DFS. It performs various manipulations on the data and controls the function of the board. A cell will be taken from the fabric if the receive fifo is less than half full at the point when the port controller route byte arrives. The port controller route byte itself is otherwise ignored.

The differing action of the network interface is controlled by bits 1 to 3 of the first byte of the header proper as it arrives from the fabric. This represents VPI bits 5 to 7 for a standard header. The meaning of these bits is shown in table 10. Note that this means that it is impossible for a DFS to receive investigate fabric cells. A comparison of the DFS control bits with the UNI and NNI standard cell headers is shown in table 11.

These differing behaviours will now be described.

D3..1	Purpose
001	Boot cell
010	Reset cell
011	8-bit mode
10x	16-bit mode (by convention 100 for video and 101 for software)
others	Cell will be ignored

Table 10: Network behaviour by header bits

Octet	Bit 7	6	5	4	3	2	1	0
0 (UNI)	GFC				VPI 7–4			
0 (NNI)	VPI 11–4							
0 (DFS)					DFS			
1	VPI 3–0				VCI 15..12			
2	VCI 11–4							
3	VCI 3–0				PT			CLP

Table 11: Comparison of cell headers

3.1 Reset Mode

When a cell of this type is received, the network interface asserts board reset. All activity will cease except that if the video circuitry has previously been programmed, the Bt463 will continue to output the same SAM memory over and over again leading to a stable picture identical on each scan line. The board stays in reset and will ignore all other cells until a Boot cell is received.

3.2 Boot Mode

If a boot mode cell is received, the board is taken out of reset (if appropriate) and the cell *minus its header* it is put into the fifo. This is to allow such cells to be executed as opcodes by the CPU providing a first level bootstrap mechanism.

3.3 16-bit mode

This is the normal mode of operation. The full 52 bytes of the cell are loaded into the FIFO as 13 words. This mode should be used for software communication and for 16-bit video streams.

3.4 8-bit mode

In this mode, the header of the cell is loaded as a single word. For subsequent bytes arriving from the fabric they are loaded into two successive fifos, replicating each byte. Thus each word in the fifo contains two bytes of cell, the first byte replicate across lanes 0 and 1, and the second byte replicated across lanes 2 and 3. A 52 byte cell therefore is stored as 25 words in the fifo.

This mode should be used for 8-bit video streams.

4 Booting

When the board comes out of reset, the Arm's MMU will be disabled, and the processor will attempt to start executing opcodes from physical address 0x0. This is the location of the network interface fifo (in blocking mode). A boot cell (with no header) will be present at the head of the fifo (since the board has just come out of reset), so the Arm will start executing the data portion of the cell.

The operating system on the DAN should ensure that the primary loader is sent in boot cells to the Framestore, and that during that time all video streams are disabled.

4.1 Primary loader

The primary loader consists of three parts and must in total be less than 512 bytes.

4.1.1 Part 1

The first part must be a multiple of 12 words (currently 24 words). It puts a code on the LEDs, and then performs initiating refresh (8 refreshes in sequence) to *both* banks of the region where the secondary loader will be placed. These 8 cycles are necessary to wake the VRAM up, and get it to start working. It also (indirectly) prepares register contents for part 2.

4.1.2 Part 2

The second part of the code contains a number (currently 9) of upload blocks each of size 12 words. Each upload block contains six opcodes of the primary loader followed by six words of secondary loader. To account for the pipeline, a load multiple of the six secondary loader words is placed at opcode 4, and the store multiple at opcode 6. The other opcodes are used to update the LED status. Each upload block will therefore transfer the six words of secondary loader to the memory prepared by part 1.

4.1.3 Part 3

The third part consists of exactly 10 opcodes, which together with the AAL5 trailer make a single cell. The tenth opcode must be a jump to the secondary loader. The two words of AAL5 trailer will thus be consumed by the Arm's pipeline, but never executed. The other opcodes in the cell are used to update the LEDs and to perform another refresh to the row containing the secondary loader.

4.2 Secondary loader

The secondary loader is now running from a small area of VRAM at offset 1 Megabyte (in the middle of memory). Its objective is to suck a single AAL5 PDU out of the FIFOs and into the bottom of VRAM. The cell headers must match a well known compiled in value (which uses the 16-bit network receive mode). If a different header is received the code halts with a fault code on the LEDs.

When there are no cells to read we spin doing 8 refresh cycles to each bank (sequentially). This ensures refresh is performed frequently.

After copying a cell with the AAL5 end of PDU bit set, the secondary loader will jump to the base of VRAM (the first word of the loaded image).

4.3 Starting the running image

The code loaded by the secondary loader is the running image that manages and runs the DFS. It is a single (large!) AAL5 PDU.

When it is started by the secondary loader it is running in physical memory at VRAM address. For efficiency of interrupt dispatching, however, it is primarily intended to operate at virtual address zero and so is linked at that address. The initial assembler must therefore initialise the MMU tables whilst using position independent code.

The very first thing that the code does is ensure that the trailing pointer left by the secondary loader is consistent with the length in the trailing cell. This at least ensures that all of the cells made it. Then it copies the MMU tables (which only need 256 bytes since we remain in 26bit mode) from to a reserved location and initialises the MMU. The virtual address space is shown in table 12.

Virtual	Physical	Size (MB)	Cache	Write Buffer	Purpose
0x00000000	0x00800000	2	YES	YES	Memory as DRAM
0x00800000	0x00800000	2	NO	YES	VRAM as uncached VRAM
0x01000000	0x01000000	2	NO	YES	Mask VRAM access
0x01800000	0x01800000	2	NO	YES	PTfifo access
0x02000000	0x02000000	2	NO	NO	SAM transfer
0x02800000	0x02800000	2	NO	YES	PTcpu access
0x03800000	0x00000000	1	NO	YES	Registers (including FIFO)

Table 12: Virtual address space

Once the MMU is ready, the code prepares a stack in the DRAM area and jumps to itself running in that area (at its linked address). It then calls some C code which performs a 32bit CRC on the running image (including the AAL5 trailer). The result is checked and the system halts if the checksum fails. Finally the `main` function is entered, and the booting process is complete.

4.4 Initialisation

The priority of the initialisation code is to get memory refresh working. All that is needed for this is that the LM1882 be programmed. This is performed first followed by enabling of FIQ (horizontal sync) interrupts. (The initial position of the screen is set to be 30 lines before vertical sync; this is the settling time of the clock circuitry). The initialisation code then initialises the Bt463, clears the mask ram, and displays the initial picture. Finally IRQs are enabled and the code enters the idle loop.

4.5 VRAM Memory Usage

The 2 Megabytes of 32-bit wide memory is used for various purposes. Table 13 shows the division of usage of the memory.

4.6 Software cursor updates

The software cursor update is triggered by the FIQ code when a vertical flyback is detected (and the FIQ must reload its screen base register). If the system is idling (in SVR mode) the FIQ will transmute the CPU into IRQ mode as it returns and cause the cursor code to be entered at a point where FIQs are enabled and the return from the IRQ will be to the SVR idle loop. If IRQs

Start(K)	Size(K)	Usage
0	$128 - \delta$	Program code and data etc.
$128 - \delta$	δ	Supervisor stack
128	0.25	MMU Table
128.25	117.75	Free
246	74	Vertical Flyback scan lines
320	1728	Visible scan lines
2048		

Table 13: Memory usage

are already active, the FIQ invents an IRQ stack frame and forces a recursive IRQ to the same cursor software. Return in this case will be top the interrupted (network) IRQ.

Software cursor updates should take less than one frame time otherwise it will be entered recursively and a crash will result. To avoid flicker, the cursor update should aim to be finished before the end of vertical blanking.

5 Paint methods

The general operation of the video or pixel painting machinery is that the hardware generates a low priority IRQ interrupt when the receive FIFOs are non-empty. An outer loop tests for data in the fifo and dispatches to a per-VCI painting routine with a state record for that VCI (having first initialised the mask register).

The inner routine stays active whilst there are cells at the head of the FIFOs that are of its VCI. If the header differs it will return the new header, if the fifo goes empty it returns zero. It transfers the data to the video memory using the PTfifo region.

5.1 AVA painting methods

There are currently two painting methods, both of which expect to work on AAL5 packets of AVA standard video. Both of these methods are aggressively optimised and achieve substantially in excess of 95% of the theoretical maximum bus bandwidth.

AVA video packets are composed of a number of 8×8 pixel *tiles*. The number of tiles per packet is called the packing factor. Tiles are sent as a sequence of 8 *slits*, where each slit is a horizontal row of eight pixels. Following the video data is a trailer which includes (amongst other things) the X and Y co-ordinates (in tiles).

For both painting methods the assumption is made that the connection is of sufficiently low loss that:

- The length need not be checked provided resynchronisation occurs on the next cell with the AAL5 bit set and that if a non-end cell is received when unexpected then it is discarded.
- The CRC need not be checked since it is video rather than program data.
- The X and Y co-ordinates of one PDU are sufficiently good predictors of the correct place to put the next data to arrive.

The assumption about low loss should be correct for data arriving at a DAN node. The network interface node should have checked it all, but this has not currently been implemented.

5.1.1 8-bit painting

For 8 bit video there are 6 slits per cell (with 0, 2 or 4 slits in the trailer cell). The VCI state record deals in the number of pairs of slits remaining in the PDU. The data of each cell will have been replicated to make 24 words. Each pair of slits can be transferred as two sets of four words to addresses in subsequent scan lines. In the write shadow of this transfer the code updates the number of pairs of slits remaining and updates the transfer pointer either another scan line, or -7 scan lines plus 8 pixels, as appropriate.

For the trailing cell, the code uses conditional instructions to transfer the remaining slits (which must progress strictly vertically) and then reads the X and Y co-ordinates and dumps the rest of the cell.[2]

5.1.2 16-bit painting

The painting of 16bit video is similar except that there are only 3 slits of data per cell (0, 1, or 2 slits in the trailing cell), therefore the pointer adjustment must be performed per slit rather than per pair of slits. Of course there is only half as much data to transfer per cell because it has not been replicated by the network module.

5.2 Nemesis painting method

There is a prototype paint method for painting updates from the Nemesis WS windows system. This method is neither finished not optimised and is not described here.

6 Acknowledgements

The excellent design and implementation of the DAN Framestore was performed by Ian Pratt Ian.Pratt@cl.cam.ac.uk. The author was involved in advisory and software roles only.

[2]This is done by transferring the data to a reserved off-screen area.