
# Programmer's Handbook

*(this book is based on 2dfx v1.0.1 firmware and Enterprise PCB v7 / TVC PCB v1)*

# Contents

# Contents (continued)

# Contents (continued)

# Acknowledgements

I would like to extend my heartfelt thanks to **Noel Persa** ***(geco)*** for his constructive comments and suggestions. From a user's and developer's point of view, he followed the project all the way from the first moving worm appearing on the screen to 2dfx becoming a complex graphics system.

I would also like to thank the **OpenAI 5.5** and **OpenAI 5.6 Sol** models, ***(CsetPeti)***, for turning the ideas and the envisioned architecture into working code for the microcontroller inside 2dfx, optimised about as far as circumstances would reasonably allow.

My thanks also go to the designers of the Enterprise and the Videoton TVC, in particular **Nicholas Toop** and **David Allen Woodfield**, for having the foresight to leave open the possibility that one day the two machines might use external colour inputs.

I thank **my father** for buying me my Enterprise computer at the spring BNV fair in 1988, and **László Szabó**, who kept it safe in his attic for almost twenty-five years before it eventually found its way back to me in 2022.

**István Matusa** ***(Tutus)*** and **Zoltán Németh** ***(ZozoSoft)*** took part in the project in a more indirect way: over the past forty years they have kept the club and magazine built around the Enterprise computer alive, and to this day they remain the driving forces behind this small but enthusiastic and remarkably persistent community. Thank you!

**Without you, 2dfx would not exist today.**

*August 2026*

# General information

# WARNING

**Use 2dfx at your own risk.**

I have tried to design the hardware so that it is difficult to cause any particular trouble during normal use, but it is still hardware connected directly to the computer's expansion connector. Incorrect connection, inserting the card in the wrong orientation or position, or using it together with an unsuitable expansion card can cause serious damage.

In the worst case, not only 2dfx but the computer itself, another expansion card, or even several of these devices at once may be damaged. **I accept no responsibility for damage of this kind.**

The most important rule is this: **always connect 2dfx to the computer while the computer is switched off, and disconnect it only while the computer is switched off.** Before powering on, check once more that the connector is facing the right way and is in the correct position.

# Other expansion cards using external colour control

2dfx uses the **/EXTC** and **EC0-3** signals intended for external colour control on the Enterprise and the TVC. On neither machine are these lines like a conventional three-state data bus, where several devices can simply "let go" of the bus whenever they have nothing to do on it. It is better to think of them as signals such as the Z80 **/INT**, **/NMI** or **/WAIT** inputs: if several pieces of hardware want to use the same line, they must be connected in a way that allows them to cooperate electrically.

2dfx is, of course, designed for this in hardware. The **/EXTC** and **EC0-3** signals do not go directly from the MCU pins to the computer; they pass through suitable open-collector-style drivers, with pull-up and series resistors. In other words, 2dfx does not try to drive these lines "by force" against another device.

If your computer also contains another card that uses the **/EXTC** and **EC0-3** signals, then before using the two devices together it is definitely worth checking that card's documentation or schematic, or asking its designer whether the expansion is intended to share these signals with other hardware.

If another card drives these lines directly from a conventional push-pull output, or if, for example, a 5 V signal can reach a component that is not designed to tolerate it, then that card must not be used at the same time as 2dfx.

In such a case the two expansions may end up electrically fighting each other. The result may be rather more serious than simply "it does not work": one of the cards can even be physically damaged. And just because two expansions work perfectly on their own does not mean at all that they are safe to use side by side.

So, if you are not certain that another card using external colour control handles sharing the **/EXTC** and **EC0-3** lines correctly, do not use the two cards at the same time.

# Technical background

On both the Enterprise and the Videoton TVC, the visible picture is generated by dedicated video circuitry. One thing the two machines have in common is that, unlike some other computers of the era – for example the Commodore 64, MSX machines or the Ataris – they have no hardware sprite support.

The designers did not, however, close off this possibility forever. On both machines they brought the pixel clock, the horizontal and vertical synchronisation signals, and five inputs intended for the so-called external colour inputs out to the expansion connector.

Using the external colour inputs (**/EXTC**, **EC0-3**), the machine's internal video circuitry can be instructed to display the colour corresponding to the four-bit value present on the external inputs instead of its own pixel currently being drawn. This gives sixteen possible colours, provided that the fifth input, **/EXTC**, is at its active-low level. The really special part of this arrangement is that the external colour appears independently of the computer's current video mode and colour mode*. In other words, whatever mode the machine is using, the colour of the current elementary pixel can be overridden from outside with one of the values of a sixteen-colour palette.
** There is one restriction on the Videoton TVC (GRAPHICS 2), but we will come back to that later.*

External hardware can follow exactly which pixel is currently being drawn by using the synchronisation signals and the pixel clock. On the Enterprise, one television line consists of 912 pixel clocks; on the TVC, it consists of 800. These are, of course, theoretical pixel counts: the entire television line is not visible, only a portion of it – the visible image area.

In the PAL system, one field consists of 312.5 television lines, so the theoretical resolution that matters to 2dfx is **912×312** on the Enterprise and **800×312** on the TVC. These figures define the coordinate system in which 2dfx operates.

# What is 2dfx?

2dfx is external graphics hardware that uses the pixel clock and synchronisation signals coming from the computer, and directly controls the **/EXTC** and **EC0-3** inputs, to change the colour of any pixel on the current screen. Its operating principle is simple: 2dfx maintains two counters. One follows the current line (Y), the other the current pixel within that line (X). The X counter restarts at every horizontal sync pulse, and the Y counter at every vertical sync pulse. The X counter is incremented by the pixel clock, while the Y counter is incremented by the line-sync signal, again one by one. From these, the 2dfx microcontroller (MCU) knows exactly where the computer currently is while drawing the picture, and sets the external colour inputs accordingly.

The microcontroller inside 2dfx runs a fairly complex piece of software that extends the graphics capabilities of the Enterprise and the Videoton TVC. Much like a modern GPU, it uses a large collection of built-in algorithms to take pixel-by-pixel calculations away from the Z80.

The hardware comes with a simple and easy-to-follow programming interface. The user program running on the computer does not have to send data for every single pixel in every single frame; instead, it describes the graphics it wants to display using higher-level commands and parameters. The 2dfx renderer then makes those elements visible by changing the appropriate pixels externally.

2dfx also contains **8 MB** of internal RAM. This storage is fully available to the user as the **Resource RAM (RRAM)**. It can hold image data, text, sprites, tiles, fill patterns and any other data needed for drawing.

This handbook is intended to help you understand how all this works and make full use of the 2dfx user interface.

The latest version of this handbook, together with other 2dfx-related material (firmware, online editors, purchasing information), can be found at **https://2dfx.net**.

# General operation and limitations of 2dfx

Among 1980s microcomputers that implemented any form of hardware sprite handling, the hardware itself determined what the user could do. On the Commodore 64, for example, the limit became a maximum of eight simultaneous 24×21-pixel sprites.

2dfx is not a similar fixed-function hardware product with a predetermined set of operating rules, nor does it run exactly the same software for exactly the same amount of time to produce every frame regardless of its contents.

Unlike such fixed hardware solutions, 2dfx offers enormous flexibility. Its main rule is that **it may perform as much graphics work, of whatever complexity, as the user can dream up, but it has one display refresh (one frame) in which to calculate those graphics objects, which in our case is 20 ms**. All graphics rendering must fit within this time budget if we want smooth, continuous movement of sprites and other graphical elements. Expressed the other way round, **this means a 50 fps refresh rate**: 2dfx has to redraw the graphics objects, text and sprites completely fifty times per second. At first this may sound rather startling to a programmer accustomed to the TVC or Enterprise, but in most cases the microcontroller working inside 2dfx handles this dense refresh schedule with ease.

*As a general rule, the larger an object is, the longer it takes to draw*, although thanks to the many thoroughly optimised drawing routines inside 2dfx, render time is not necessarily directly proportional to either the number or the size of the displayed objects. Some jobs go through so-called fast paths that require considerably less mathematical work, while others are, by their nature, more calculation-intensive and therefore slower.

Consequently, most limits found in 2dfx are not really hard physical limits; in many cases the main aim of capping parameter values was simply to keep the user interface manageable. For example, 2dfx can handle and display "only" 64 different sprites at the same time – and in the overwhelming majority of cases it will happily do so. What cannot be guaranteed is that all 64 can actually be drawn smoothly within the 20 ms interval (render time). Once again, the practical limit depends on the size and complexity of the sprites and other graphics objects involved. Fortunately the MCU is very fast and the code has been through no shortage of optimisation, so ordinary tasks fit comfortably within the available time. There can, however, be special cases where this is no longer true.

With 2dfx, the proof of the pudding really is in the eating: the user may dream up graphics of any complexity, huge sprites, moving full-screen backgrounds and so on, and programming/testing will reveal whether drawing them really fits within the render time. 2dfx also provides visual help with this. On the one hand, its dedicated output can be connected to an oscilloscope to measure the actual render time precisely; on the other, while displaying graphics, 2dfx can warn the programmer that rendering has overflowed the 20 ms budget by placing a huge flashing exclamation mark in the middle of the screen. This direct visual warning can, of course, be enabled and disabled in software. Even a render-time overrun is not otherwise a disaster. 2dfx does not freeze, reboot, crash or start drawing random rubbish; animations or moving graphics simply become visibly jerky, and only for as long as the render-time overrun actually persists.

The descriptions of the Easter Eggs near the end of the book also include the measured render time for each Egg on both machine types. These figures may provide a useful reference when experimenting with how far the limits of 2dfx can be pushed.

# Geometry

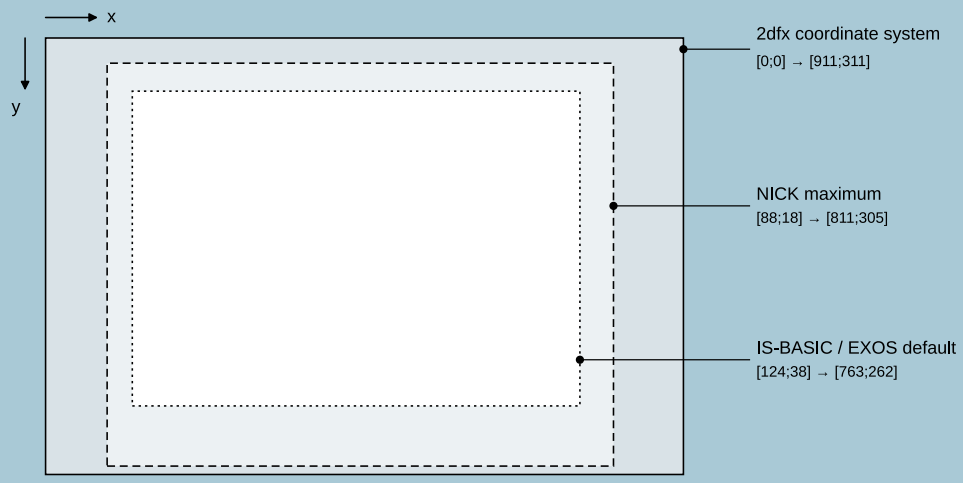
The 2dfx graphics coordinate system is defined by the theoretical pixel counts mentioned earlier – as a reminder, **912×312** on the Enterprise and **800×312** on the TVC. Because this includes more than just the visible image area, we can create objects that lie partly or entirely outside the screen. This is particularly useful, for example, if we want a sprite to glide in from outside the visible area, or if we create pixel-precise scrolling text that begins in the invisible region and runs out beyond the opposite edge.

An important geometric characteristic of both machines is that the pixel aspect ratio is approximately **2:1** (slightly less on the TVC). If we want a shape to appear genuinely square on the screen, its horizontal size in pixels should therefore be roughly twice its vertical size.

Neither the Enterprise nor the TVC reports what resolution the screen is currently using, how wide the visible area is, how many lines are visible, or how large the border is. The programmer must therefore position content drawn by 2dfx in accordance with the screen dimensions currently configured on the machine. 2dfx itself always works in the coordinate system described above.

# Enterprise geometry

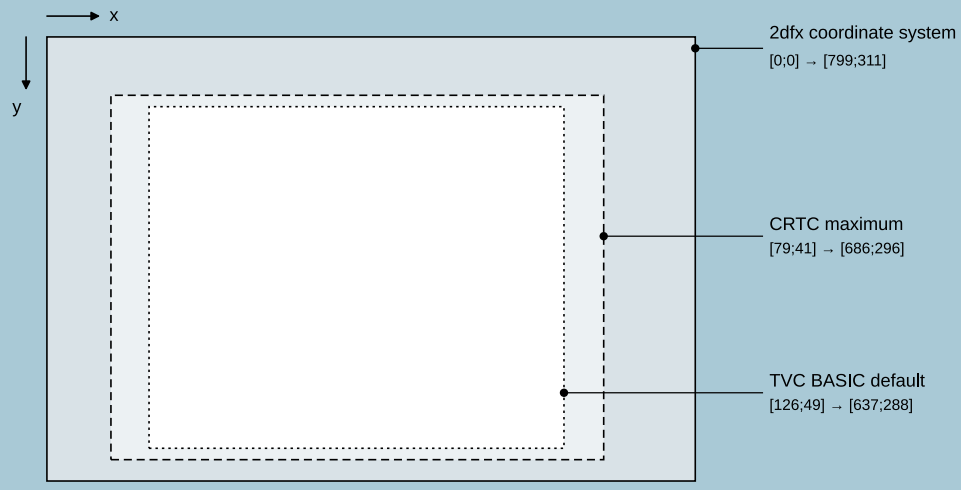
On the Enterprise, the diagram below shows the practical layout of the coordinate system:




On the Enterprise, the size and position of the visible image area can be configured by programming the NICK video chip. The largest available screen area appears between the coordinates shown in the diagram above. If we do not modify NICK's LPT and instead use the default EXOS / IS-BASIC screen dimensions, the actually visible area falls within a smaller range (see the diagram). The area outside the screen is part of the border; NICK does not allow this region to be overridden through the external colour inputs.

# Videoton TVC geometry

The Videoton TVC coordinate system is as follows:



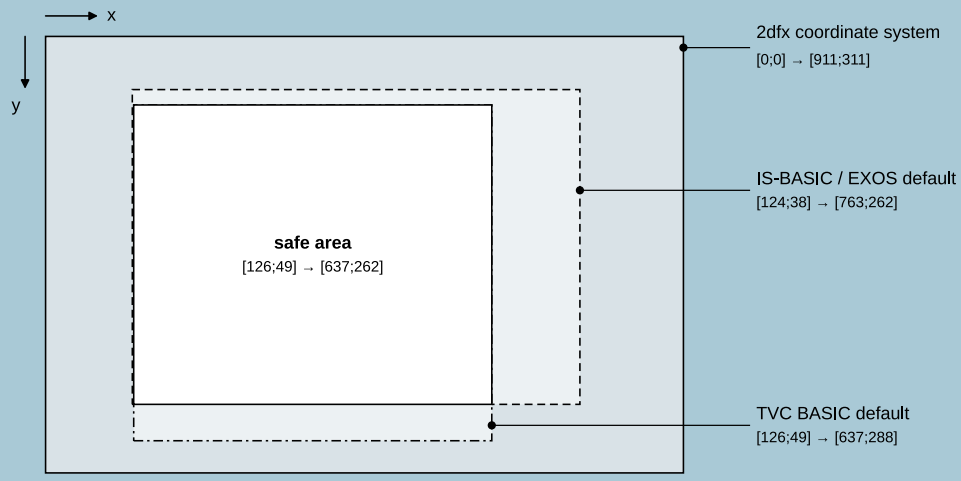

On the Videoton TVC, a Motorola 6845 CRTC controls the timing of the on-board video circuitry. Compared with the Enterprise NICK chip, this circuit offers simpler possibilities; the TVC's largest still-visible resolution can be **608×256** pixels.

On the TVC it is particularly important that the full horizontal resolution is available only in GRAPHICS 2 mode because of the machine's motherboard design. In any other mode, the TVC considers only every second theoretical pixel arriving from the external colour inputs and displays it two pixels wide. 2dfx receives no feedback about this –the TVC provides no such information– so it always sends consecutive pixels at the same rate. How they are interpreted for display is then entirely up to the TVC. One of the Easter Eggs is dedicated specifically to demonstrating this motherboard-level hardware limitation.

On the TVC, too, the area outside the visible image remains in the border colour. During border time, the external colour inputs cannot be used to change the colour of those pixels.

# Cross-platform development

Because the default layouts of the two machines differ, cross-platform development can be easier if we choose a coordinate region that lies within the default visible area (IS-BASIC or TVC BASIC) on both machines. The safe region in which graphics can be placed can be worked out at a glance from the previous two diagrams, but to save everyone the arithmetic, here is a diagram of the safe area:




It is also worth noting that, for cross-platform development, the graphics elements, sprites and bitmaps drawn by 2dfx only need to be prepared for one machine. 2dfx will draw them in exactly the same way on the Enterprise as on the TVC, and vice versa. What does require attention is colour setup: on the Enterprise side, the fixed TVC palette must be mapped by programming NICK appropriately if, for example, a 2dfx application is being ported from the TVC to the Enterprise.

Note: the pixel aspect ratio is slightly different on the two machines because their pixel clocks differ. On the Enterprise one pixel lasts about 70 ns, while on the TVC it is exactly 80 ns. In practice this means that, viewed on the same monitor, a 2dfx pixel is about 14% wider on the TVC than on the Enterprise. Even so, this is unlikely to cause a noticeable difference in proportions when comparing the pictures from the two machines.

# Colour handling

As we already know, both the Enterprise and the TVC provide five input lines that can be used to change the colour of a pixel.

The **EC0–3** lines define a four-bit number: the colour index from the current sixteen-colour palette that we want to display at the given pixel.

The **/EXTC** input line controls, on both the Enterprise and the TVC, whether the colour index arriving on the **EC0–3** inputs actually overrides the current pixel; in other words, it is effectively the "transparency switch" for that pixel. When it is 0, the machine displays the colour index arriving on the **EC0–3** lines at that pixel; when it is 1, the computer displays the pixel generated by its own video circuitry.*

** On the Enterprise there are also two more exotic configuration possibilities thanks to NICK's capabilities, but they are outside the scope of this book. They can be learned from the NICK documentation and do not change the operating principle of 2dfx.*

All five input lines described above have to be controlled – which, for 2dfx, would mean storing five bits per pixel. The earliest experiments started from exactly this approach, but because of the storage complexity and the associated extra rendering overhead, the final version of 2dfx uses hardware comparator-based transparency control instead.

In practice this means that we can – and must – select one of the sixteen possible colour indices for 2dfx. A hardware component on the 2dfx circuit board, independent of the microcontroller, compares that value with the colour index currently coming from 2dfx. If the two values differ, the hardware comparator signals through the **/EXTC** input that the computer should display the colour generated by 2dfx. If they match, it instructs the computer to ignore the colour on the **EC0–3** lines and display the pixel generated by the computer itself.

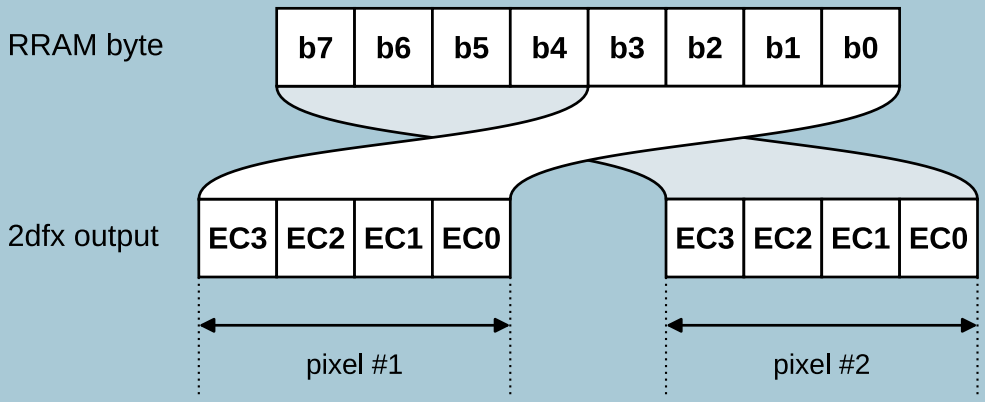
The consequence is that **when using 2dfx, one colour must be designated – and sacrificed – so that 2dfx can provide the computer with a transparency signal.** On the TVC, a practical choice is the "second black"; on the Enterprise, for example, one of the BIAS colours (8–15). Regardless of the active 2dfx profile (Enterprise or TVC), the power-on default is colour index **zero**. A single command lets us change it to any other index at any time while the system is running.

2dfx also has a special ENGINE_ON_FORCED mode. This overrides the hardware colour comparator and holds the **/EXTC** input continuously at 0. In this mode we have to give up graphics and text generated by the computer itself, because 2dfx continuously tells the computer to display the colour information arriving through the external colour inputs, completely overriding its own picture.

Since 2dfx tries to cover as broad a range of graphics tasks as possible with its command set, it is quite likely that we will need the computer's own graphics capabilities only minimally – or perhaps not at all.

Now that transparency is clear, we really should talk about the **EC0–3** lines themselves, so that colour handling does not spring any surprises on the 2dfx programmer.

In its own memory, 2dfx stores the colours of two pixels in one byte. The relationship between this representation and the values on the **EC0–3** lines is shown below; we will return to it when discussing Resource RAM.




As can be seen, whichever machine we use, the 2dfx RAM stores the colour indices of two consecutive pixels, and those values are placed on the **EC0–3** lines in sequence without alteration. This was designed deliberately so that we do not waste microcontroller time continuously rearranging bits before displaying each pixel on the **EC0–3** outputs. That leaves more resources available for rendering.

In the internal 4bpp pixel format used by 2dfx, the lower four bits deliberately contain the colour index of the left-hand pixel and the upper four bits the colour index of the right-hand pixel. At first this may look backwards compared with familiar storage formats, but it directly matches the way the 2dfx output interface works: framebuffer data is sent to the PIO by 32-bit DMA, and the PIO shifts right, always outputting the low four bits to the EC0-3 lines first. The framebuffer can therefore be displayed directly, without any intermediate nibble rearrangement, which also improves processing speed.

The 2dfx online graphics editors already export the artwork in this format by default.

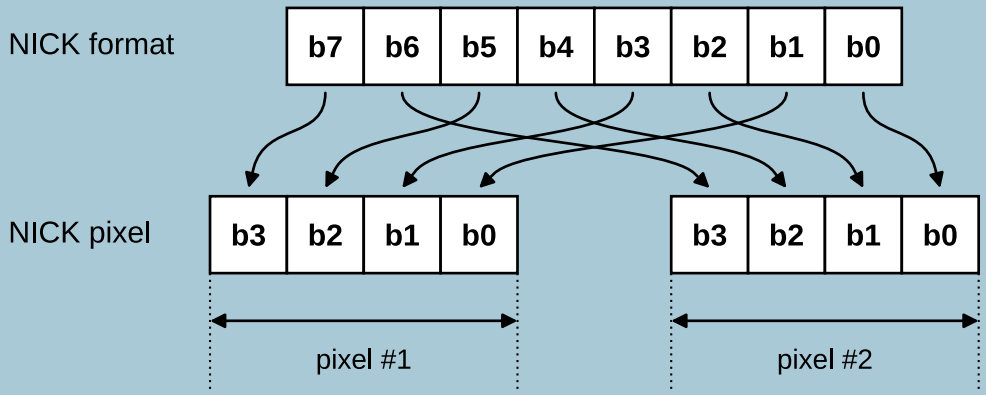
# Enterprise colour handling

On the Enterprise, the relationship between colours and the **EC0–3** lines is fairly straightforward. The Enterprise's current sixteen-colour palette* maps directly, one-to-one, to the values of the **EC0–3** lines.

** The lower eight colours of the palette (0–7) can be freely selected from the 256 available colours – even on a line-by-line basis if an appropriate LPT is created – while the upper eight colours can be changed for the whole picture by adjusting NICK's BIAS value.*

So, if we want a pixel to use colour 7 of the current palette, all we have to do is set the value of that pixel in 2dfx RAM to **0111b**.

Simple as this storage scheme is to follow, it is completely alien to programmers brought up on the Enterprise, because in 16-colour mode NICK represents the colours of two pixels in its own video RAM as follows:

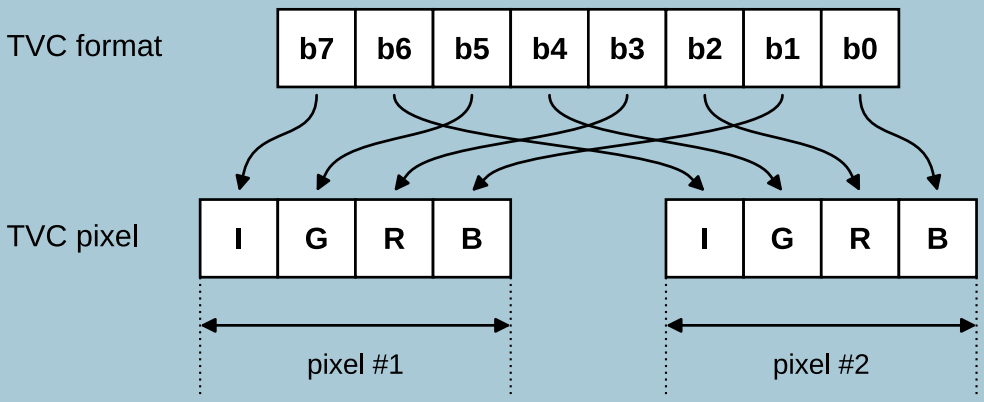



Existing sprite data or images in NICK format would therefore have to be converted into the internal 2dfx format before 2dfx could display them correctly. Before the dear programmer clutches at their heart, some reassurance: 2dfx takes care of this. Alongside **UPLOAD_RAW** (the command used to upload data already in 2dfx format), it also provides **UPLOAD_NICK**. The latter lets image data originally produced in NICK format be uploaded without any user-side conversion; after the upload, 2dfx performs the NICK→2dfx format conversion itself, byte by byte. Detailed information about these commands can be found in the **Structure and use of Resource RAM (RRAM)** chapter.

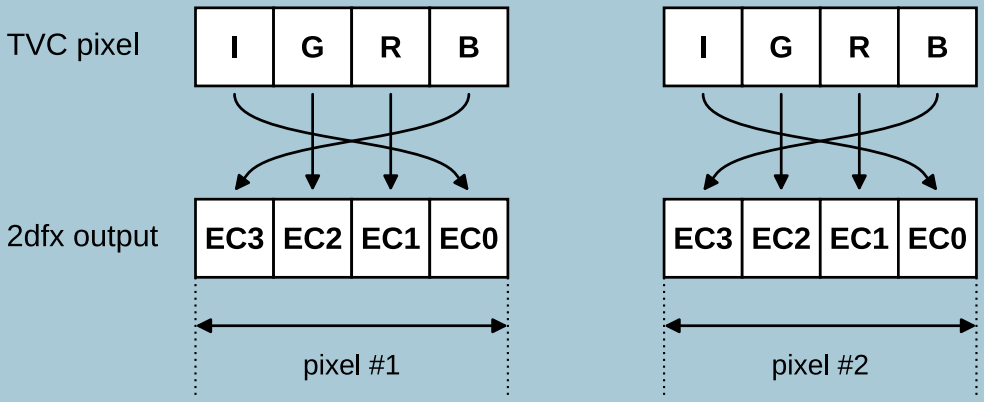
# Videoton TVC colour handling

Although the TVC has only sixteen colours, and those colours cannot even be changed, its designers still managed to make the way pixel colours are stored in video RAM thoroughly interesting – just as on the Enterprise.

In 16-colour mode, the storage of adjacent pixel colours bears an uncanny resemblance to the Enterprise NICK format, namely:

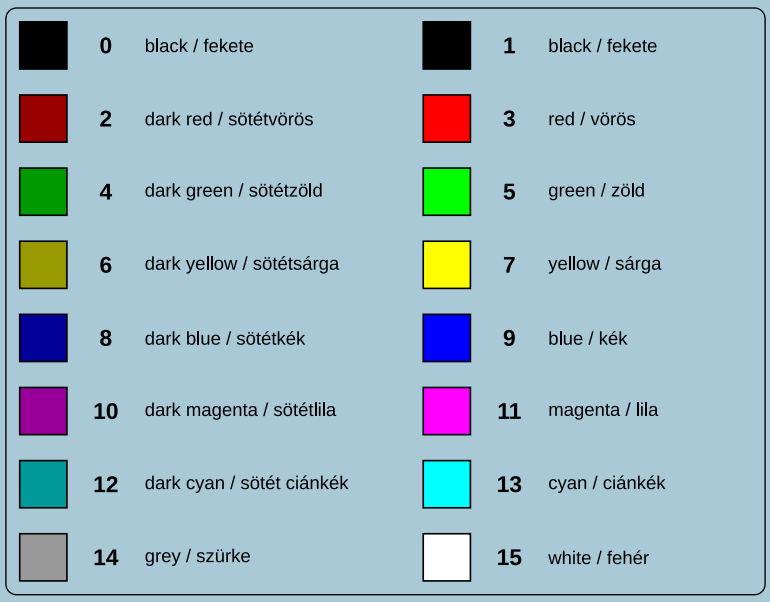



But the shuffling does not stop there: compared with the already rather scrambled video-RAM representation, the designers also rearranged the values on the **`EC0-3`** lines. Here it is:




As can be seen, the individual bits are assigned completely differently even from the TVC video-RAM format: instead of IGRB, the EC3-0 inputs expect BGRI.

This arrangement does, however, lead to a fixed mapping (because the sixteen-colour palette cannot be changed). To display TVC graphics correctly, data placed in 2dfx RRAM must therefore be prepared according to the following colour table:



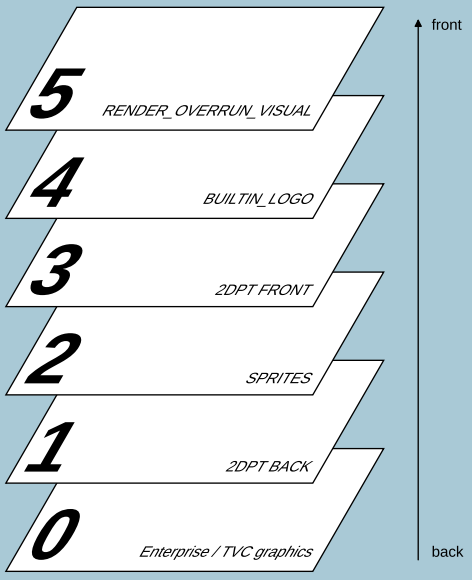

There is no reason to panic here either. Firstly, the format above is compulsory for colour-correct image generation only when using **`UPLOAD_RAW`**. Secondly, the built-in **`UPLOAD_TVC`** command can upload graphics prepared in the 16-colour video-RAM format described in the TVC manual, after which 2dfx automatically performs the necessary TVC→2dfx conversion on the uploaded data. The **Structure and use of Resource RAM (RRAM)** chapter explains the relevant commands and their use in more detail.

In addition, as mentioned earlier, the 2dfx online graphics editors export finished graphics directly in 2dfx format by default, so they can be used immediately with **`UPLOAD_RAW`** (see later).

Knowing the colour table above is nevertheless important when choosing the transparent colour: it matters which colour index we decide to sacrifice in order to provide transparency.

# Layering

2dfx builds the picture as **five** layers placed on top of one another. A layer with a higher number always has higher priority and therefore appears above the layers below it:




The **zeroth layer** is the picture generated by the Enterprise or TVC itself: text, graphics, border, everything the machine would draw on its own. 2dfx modifies this base picture by using the external colour inputs.

To understand the remaining layers, it is worth taking a brief look at how the 2dfx graphics system evolved.

The original aim of the project was to display hardware sprites while requiring as little work as possible from the Z80 CPU. The application does not have to spend time drawing the sprites in software; it merely has to set the appropriate 2dfx parameters. Sprites have priority according to their number: sprite 0 is at the bottom and sprite 63 at the top. If visible, non-transparent pixels of two sprites overlap, the pixel belonging to the higher-numbered sprite is displayed at that position. The sprites form the **second layer**.

During development it became clear that the MCU inside 2dfx was capable of considerably more. It can handle not only sprites, but also generate graphics across the machines' full theoretical horizontal and vertical resolution, right down to individual pixels, using any of the sixteen available colours.

Two 2D drawing layers were therefore added alongside the sprites: *2DPT back* and *2DPT front*. Each is redrawn every frame from a playlist of up to 256 elementary graphics operations. 2DPT back is placed behind the sprites, while 2DPT front is placed in front of them. Later chapters describe in detail the drawing facilities provided by these layers. *2DPT back* is therefore the **first layer**, and *2DPT front* the **third layer**.

**SHOW_BUILTIN_LOGO** can also appear as a separate layer. It draws or hides the built-in 2dfx logo at any position on the screen without requiring the user to upload separate logo data. In the menu of a game or demo, for example, it works nicely as a “powered by 2dfx” badge. Its use is, of course, entirely optional. The logo has a fixed size of **206×79** pixels and is the sole element of the **fourth layer**.

The highest-priority **fifth layer** is a particularly useful form of feedback during development. If drawing 2DPT back/front and processing the sprites does not fit within the time available for the current frame – 20 ms in 50 fps mode, or 40 ms in 25 fps mode – 2dfx can display a continuously colour-changing exclamation mark when **SET_RENDER_OVERRUN_VISUAL** is enabled. This indicates that the graphics workload exceeds the renderer’s current time budget, which in practice may result in jerkier frame changes. The use of this feature is covered in more detail later.

# Communicating with 2dfx

On both the Enterprise and the Videoton TVC, 2dfx communicates through just two Z80 I/O ports. On both machines these are fixed at **F8h (248)** and **F9h (249)**.

The commands and parameters are identical regardless of which machine is being used. This makes it easier to port applications between the Enterprise and the TVC: graphics objects drawn by 2dfx appear using the same commands and in the same coordinate system.

When porting, of course, the different visible image areas of the two machines, the starting position of the picture and the interpretation of colours must all be taken into account. These are covered in detail in the **Geometry** and **Colour handling** chapters.

The **F8h (248)** port is both a *command and status port*: it can be written and read. **F9h (249)** is the *data port*, which is write-only and is used to send the parameters of the current command to 2dfx. For example, sending a command with three parameters looks like this:

```
OUT 248, aaa          ! aaa = command code
OUT 249, bbb          ! bbb = value of the first parameter
OUT 249, ccc          ! ccc = value of the second parameter
OUT 249, ddd          ! ddd = value of the third parameter
```

The effect of sprite and 2DPT graphics commands always appears in the next frame. This prevents half-updated object parameters from being used while rendering, and lets us prepare changes for the next picture at any time within a frame without any special software timing. The changes become active at the next vertical synchronisation.

From this point on, I use the following table format to present commands and their parameters:

command name

command code, hexadecimal

command code, decimal

**SET_RASTER_INT1..4 (54h..57h / 84..87)**

Up to four raster interrupts can be configured with this command. Using the synchronisation signals, 2dfx tracks which television line the Enterprise or TVC is currently displaying. When the configured line is reached, the corresponding raster-interrupt bit in the status register is set to 0. Whenever any raster interrupt occurs, 2dfx activates the computer's /INT input in hardware, requesting an interrupt from the Z80. The default interrupt mode on both Enterprise and TVC is IM1, in which the Z80 jumps to address 0038h.

To disable a particular raster interrupt completely, set its line value to zero.

**description** what the command does

Data bits

| Parameter name | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description |
|---|---|---|---|---|---|---|---|---|---|
| line_high | 0 | 0 | 0 | 0 | 0 | 0 | 0 | a | raster line value |
| line_low | • | • | • | • | • | • | • | • | raster line value |

**parameter structure** the bits of every parameter byte

**a** the upper, ninth bit of the raster line
**•** the lower eight bits of the raster line
**0** must be set to zero

**symbols** meaning of the marks used in the table

WARNING! Triggering a raster interrupt is not a cycle-exact timing signal at the start of a line. The 2dfx MCU sets the indication when it has reached the configured line and is processing that event. In practice this normally happens shortly after horizontal sync, but it cannot be guaranteed to occur exactly at the beginning of the line. If a program needs precise timing within a particular line, it is advisable to configure the raster interrupt for the previous line and then use timing inside the Z80 interrupt routine to reach the required point. In practice, the indication arrives no later than roughly 14 usec after horizontal sync.

**remark** important practical note

# The status register

Reading port **F8h (248)** returns the value of the 2dfx status register. **An active indication is represented by 0 (zero)!** The bits of the register have the following meanings:

| bit | name | description |
|---|---|---|
| bit7 | COLLISION_DETECTED | At least one pair of visible pixels belonging to two active sprites overlapped in the previous frame. Collision detection has a resolution of two pixels horizontally and one pixel vertically. |
| bit6 | 2DFX_PRESENT | After initialisation is complete, 2dfx keeps this bit fixed at 0. This allows the user program to check whether a 2dfx card is present in the system. (See also: Initialisation chapter.) |
| bit5 | RENDER_OVERRUN | Active if 2dfx could not complete the current rendering task within the time available for the frame. It can be cleared with the CLEAR_RENDER_OVERRUN command. |
| bit4 | RASTER_INT4 | The fourth raster interrupt has occurred. |
| bit3 | RASTER_INT3 | The third raster interrupt has occurred. |
| bit2 | RASTER_INT2 | The second raster interrupt has occurred. |
| bit1 | RASTER_INT1 | The first raster interrupt has occurred. |
| bit0 | RENDER_TIME | Used for measuring render time with an oscilloscope. It is also brought out to the solder point marked RTIME on the 2dfx circuit board. It is active while the renderer is calculating the data for a frame. After issuing SYS_RESET, its value remains 0 while the command is running, then changes to 1. |

The status port is also brought out on the 2dfx circuit board to eight green LEDs and the same number of solder pads, making measurements with a logic analyser or oscilloscope easier.

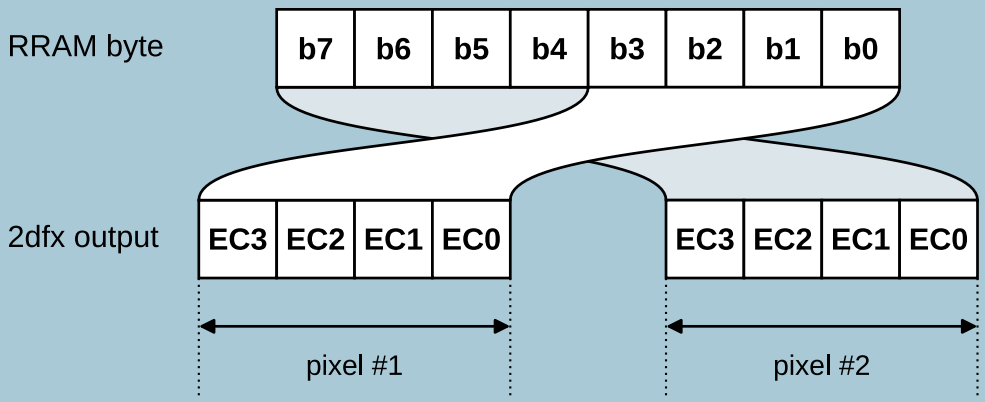
More detailed information about the status bits can be found in the descriptions of the commands or events associated with them.

# Detailed command reference

# Structure and use of Resource RAM (RRAM)

The 2dfx card provides 8 MB of RAM for storing data required for display. This may contain, for example, sprite bitmaps, character sets, tiles, tilemap data, strings to be displayed, and any other data used by the drawing subsystems. This memory area is called **Resource RAM**, or **RRAM** for short.

For efficient display, the internal 2dfx pixel format stores the colour indices of two adjacent pixels in one byte:




In other words, one byte of RRAM contains the colour indices of two neighbouring pixels: the future values of the output **`EC0-3`** signals.

Fonts are also stored in RRAM. Since font data does not store a colour for every pixel, but only whether that pixel is visible, it uses a 1bpp format. FONT-based text drawn by 2dfx can therefore use only one colour at a time. Naturally, that colour can be chosen freely for each string, but not independently within a single glyph image.

The RRAM address range is **`000000h–7FFFFFh`**. The entire area is available to the user, and data may be uploaded freely starting at any address. Addresses are 23 bits wide, and all 2dfx subsystems expect them as three bytes in *little-endian* order: first the lower eight bits, then the middle eight bits, and finally the upper seven bits. The most significant bit of the third byte must be 0.

If an upload runs past address **`7FFFFFh`**, 2dfx automatically wraps back to **`000000h`**. The byte following **`7FFFFFh`** is therefore written at the beginning of RRAM, and the address counter continues increasing from there *(address wrap)*.

The **`UPLOAD_XX`** commands can upload at most 65536 bytes, or 64 kB, in a single transaction. This does not limit the total size of a graphics object; it simply means that larger data must be transferred to RRAM in several consecutive uploads.

The upload commands behave rather like LDIR on the Z80: if the number of bytes to upload is **zero**, the command expects **65536** bytes of data. Otherwise it expects exactly the number specified by the two-byte length value.

Use **UPLOAD_RAW** when uploading data already in the native 2dfx format, when sending font data, or for any other data (for example a string) that must be stored byte-for-byte in RRAM exactly as uploaded.

The Enterprise NICK chip and the various image converters store pixel data in a different order, so 2dfx must also be able to handle that format.

The same is true of the Videoton TVC, whose sixteen-colour pixel data is also stored differently from the internal 2dfx format. In addition, the wiring of the **EC0-3** colour inputs on the TVC motherboard does not match the logic of its conventional video modes: EC0 is connected to I, EC1 to R, EC2 to G and EC3 to B.

For this reason, **UPLOAD_NICK** and **UPLOAD_TVC** convert every uploaded data byte from the NICK or TVC format, respectively, into the internal 2dfx pixel format during the upload. By contrast, **UPLOAD_RAW** stores the data in RRAM byte-for-byte without conversion.

## UPLOAD_RAW (80h / 128)

Uploads data to RRAM. The parameters specify the RRAM start address and the number of bytes to upload, followed by exactly that many data bytes. The uploaded data is stored in RRAM unchanged.

| | Data bits | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Parameter name | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description |
| RRAM_address_low | • | • | • | • | • | • | • | • | RRAM start address bits 7..0 |
| RRAM_address_mid | • | • | • | • | • | • | • | • | RRAM start address bits 15..8 |
| RRAM_address_high | 0 | • | • | • | • | • | • | • | RRAM start address bits 22..16 |
| length_low | • | • | • | • | • | • | • | • | number of bytes to upload, bits 7..0 |
| length_high | • | • | • | • | • | • | • | • | number of bytes to upload, bits 15..8 |
| payload_0000h | • | • | • | • | • | • | • | • | first uploaded data byte |
| payload_0001h | • | • | • | • | • | • | • | • | second uploaded data byte |
| payload_yyyyh | • | • | • | • | • | • | • | • | uploaded data byte yyyyh |

**0** must be set to zero

WARNING! If the number of bytes to upload (length_low, length_high) is set to zero, the command expects exactly 65536 bytes of payload data.

# UPLOAD_NICK (81h / 129)

Uploads data to RRAM. The parameters specify the RRAM start address and the number of bytes to upload, followed by exactly that many data bytes. Before writing to RRAM, the NICK bit rearrangement is performed (see the Colour handling chapter).

| | Data bits | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Parameter name | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description |
| RRAM_address_low | • | • | • | • | • | • | • | • | RRAM start address bits 7..0 |
| RRAM_address_mid | • | • | • | • | • | • | • | • | RRAM start address bits 15..8 |
| RRAM_address_high | 0 | • | • | • | • | • | • | • | RRAM start address bits 22..16 |
| length_low | • | • | • | • | • | • | • | • | number of bytes to upload, bits 7..0 |
| length_high | • | • | • | • | • | • | • | • | number of bytes to upload, bits 15..8 |
| payload_0000h | • | • | • | • | • | • | • | • | first uploaded data byte |
| payload_0001h | • | • | • | • | • | • | • | • | second uploaded data byte |
| payload_yyyyh | • | • | • | • | • | • | • | • | uploaded data byte yyyyh |

**0** must be set to zero

WARNING! If the number of bytes to upload (length_low, length_high) is set to zero, the command expects exactly 65536 bytes of payload data.

# UPLOAD_TVC (82h / 130)

Uploads data to RRAM. The parameters specify the RRAM start address and the number of bytes to upload, followed by exactly that many data bytes. Before writing to RRAM, the TVC bit rearrangement is performed (see the Colour handling chapter).

| | Data bits | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Parameter name | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description |
| RRAM_address_low | • | • | • | • | • | • | • | • | RRAM start address bits 7..0 |
| RRAM_address_mid | • | • | • | • | • | • | • | • | RRAM start address bits 15..8 |
| RRAM_address_high | 0 | • | • | • | • | • | • | • | RRAM start address bits 22..16 |
| length_low | • | • | • | • | • | • | • | • | number of bytes to upload, bits 7..0 |
| length_high | • | • | • | • | • | • | • | • | number of bytes to upload, bits 15..8 |
| payload_0000h | • | • | • | • | • | • | • | • | first uploaded data byte |
| payload_0001h | • | • | • | • | • | • | • | • | second uploaded data byte |
| payload_yyyyh | • | • | • | • | • | • | • | • | uploaded data byte yyyyh |

**0** must be set to zero

WARNING! If the number of bytes to upload (length_low, length_high) is set to zero, the command expects exactly 65536 bytes of payload data.

**2dfx also includes a built-in ZX1 decompressor.** Pre-compressed ZX1 data packages can therefore be uploaded to RRAM with **`UPLOAD_RAW`** and then decompressed into the desired destination area. **`UNZX1_RAW`** performs byte-exact decompression, while **`UNZX1_NICK`** and **`UNZX1_TVC`** also perform the appropriate bit rearrangement on the affected RRAM area after decompression.

For example, if an Enterprise-format graphics object is compressed with ZX1 on the development machine, the compressed data must be uploaded with **`UPLOAD_RAW`**. The **`UNZX1_NICK`** command can then decompress it and perform the NICK→2dfx format conversion.

The **Upload guide** chapter provides a comprehensive table of the possible upload and decompression variants.

After uploading ZX1-compressed data, decompression must always be performed as a separate step. The 2dfx graphics subsystems interpret the selected RRAM area as raw data; they have no way of knowing that it actually contains a compressed data package.

**Warning! While UNZX1 decompression is in progress, the current rendering work is suspended. The last completed frame remains visible on screen until the operation finishes. For this reason, larger uploads/decompressions are best performed when the application starts, between game levels, or, in demos, during transitions between scenes.**

## UNZX1_RAW (83h / 131)

Decompresses ZX1-compressed data previously uploaded to RRAM with UPLOAD_RAW, starting at the destination address. No bit rearrangement is performed during decompression.

| | Data bits | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Parameter name | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description |
| RRAM_src_addr_low | • | • | • | • | • | • | • | • | RRAM start address of the compressed data, bits 7..0 |
| RRAM_src_addr_mid | • | • | • | • | • | • | • | • | RRAM start address of the compressed data, bits 15..8 |
| RRAM_src_addr_high | 0 | • | • | • | • | • | • | • | RRAM start address of the compressed data, bits 22..16 |
| RRAM_dst_addr_low | • | • | • | • | • | • | • | • | RRAM destination address for decompression, bits 7..0 |
| RRAM_dst_addr_mid | • | • | • | • | • | • | • | • | RRAM destination address for decompression, bits 15..8 |
| RRAM_dst_addr_high | 0 | • | • | • | • | • | • | • | RRAM destination address for decompression, bits 22..16 |

**0** must be set to zero

WARNING! ZX1-compressed data should be uploaded only with UPLOAD_RAW so that its contents are preserved byte-for-byte.

# UNZX1_NICK (84h / 132)

Decompresses ZX1-compressed data previously uploaded to RRAM with UPLOAD_RAW, starting at the destination address. Before each data byte is stored, the NICK bit rearrangement is performed during decompression.

| | Data bits | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Parameter name | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description |
| RRAM_src_addr_low | • | • | • | • | • | • | • | • | RRAM start address of the compressed data, bits 7..0 |
| RRAM_src_addr_mid | • | • | • | • | • | • | • | • | RRAM start address of the compressed data, bits 15..8 |
| RRAM_src_addr_high | 0 | • | • | • | • | • | • | • | RRAM start address of the compressed data, bits 22..16 |
| RRAM_dst_addr_low | • | • | • | • | • | • | • | • | RRAM destination address for decompression, bits 7..0 |
| RRAM_dst_addr_mid | • | • | • | • | • | • | • | • | RRAM destination address for decompression, bits 15..8 |
| RRAM_dst_addr_high | 0 | • | • | • | • | • | • | • | RRAM destination address for decompression, bits 22..16 |

**0** must be set to zero

WARNING! ZX1-compressed data should be uploaded only with UPLOAD_RAW so that its contents are preserved byte-for-byte. If the original source data (before compression) was in NICK format, use this command for decompression.

# UNZX1_TVC (85h / 133)

Decompresses ZX1-compressed data previously uploaded to RRAM with UPLOAD_RAW, starting at the destination address. Before each data byte is stored, the TVC bit rearrangement is performed during decompression.

| | Data bits | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Parameter name | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description |
| RRAM_src_addr_low | • | • | • | • | • | • | • | • | RRAM start address of the compressed data, bits 7..0 |
| RRAM_src_addr_mid | • | • | • | • | • | • | • | • | RRAM start address of the compressed data, bits 15..8 |
| RRAM_src_addr_high | 0 | • | • | • | • | • | • | • | RRAM start address of the compressed data, bits 22..16 |
| RRAM_dst_addr_low | • | • | • | • | • | • | • | • | RRAM destination address for decompression, bits 7..0 |
| RRAM_dst_addr_mid | • | • | • | • | • | • | • | • | RRAM destination address for decompression, bits 15..8 |
| RRAM_dst_addr_high | 0 | • | • | • | • | • | • | • | RRAM destination address for decompression, bits 22..16 |

**0** must be set to zero

WARNING! ZX1-compressed data should be uploaded only with UPLOAD_RAW so that its contents are preserved byte-for-byte. If the original source data (before compression) was in TVC format, use this command for decompression.

# Global commands affecting general operation

The global commands affect the general operation of 2dfx. They do not belong to a particular sprite or drawing object; instead they affect the graphics engine as a whole by setting the operating mode, transparent colour, rendering cadence and various operating indications.

## SYS_RESET (8Fh / 143)

Resets the application and graphics state of 2dfx, effectively performing a soft reset. It clears the contents of RRAM, the SPBs, both 2DPTs and the DEFINE_XX resource definitions, and also restores settings including SET_FPS, SET_TRANSPARENT_COLOR and the raster interrupts to their defaults. It does not change the selected engine mode (OFF/NORMAL/FORCED). While SYS_RESET is running, bit 0 of the status register is 0 and changes to 1 when SYS_RESET has finished. Always wait for this before issuing another command.

the command has no **F9h (249)** parameter

## SET_ENGINE_MODE (50h / 80)

Selects between the three main operating modes of 2dfx.

In ENGINE_OFF, the renderer does not start new frames and 2dfx displays no graphics of its own, but communication remains active: the hardware continues to receive and process commands.
ENGINE_ON is the normal operating mode. Here 2dfx takes control of a pixel's colour only where it draws non-transparent content; for a transparent pixel, the Enterprise or TVC picture remains visible.
In ENGINE_ON_FORCED mode, 2dfx keeps /EXTC continuously active, so the machine's own picture cannot show through: 2dfx determines the colour of every pixel.

| | Data bits | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Parameter name | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description |
| mode | • | • | • | • | • | • | • | • | eight-bit value selecting the operating mode |

- 0 = ENGINE_OFF
- 1…254 = ENGINE_ON
- 255 = ENGINE_ON_FORCED

## SET_TRANSPARENT_COLOR (40h..4Fh / 64..79)

Selects the colour index that 2dfx will treat as transparent from this point on. If a sprite or another graphics command that performs transparency testing contains a pixel of this colour, 2dfx does not draw it; instead, by leaving /EXTC inactive, it tells the computer to keep the machine's own picture at that position. For sprites, collision detection likewise considers only visible, non-transparent pixels. The lower four bits of the command code already specify the transparent colour index, so the command requires no additional parameter.

the command has no **F9h (249)** parameter
The power-on default is colour index zero.

## SET_FPS (51h / 81)

Sets the 2dfx scene refresh cadence. The default is 0, meaning 50 fps operation: 2dfx produces a new frame on every vertical synchronisation, giving the renderer 20 ms per frame. If a program, game or demo does not need such a fast refresh, or the graphics workload is too complex for the 20 ms budget, SET_FPS can be set to 1 to select 25 fps. In this mode rendering may take up to 40 ms, and the completed image remains visible for two 50 Hz video frames.

| | Data bits | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Parameter name | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description |
| mode | 0 | 0 | 0 | 0 | 0 | 0 | 0 | a | selects the operating mode |

**a** 0 = 50 fps
1 = 25 fps

**0** must be set to zero

## SET_RASTER_INT1..4 (54h..57h / 84..87)

Up to four raster interrupts can be configured with this command. Using the synchronisation signals, 2dfx tracks which television line the Enterprise or TVC is currently displaying. When the configured line is reached, the corresponding raster-interrupt bit in the status register is set to 0. Whenever any raster interrupt occurs, 2dfx activates the computer's /INT input in hardware, requesting an interrupt from the Z80. The default interrupt mode on both Enterprise and TVC is IM1, in which the Z80 jumps to address 0038h.

To disable a particular raster interrupt completely, set its line value to zero.

| | Data bits | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Parameter name | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description |
| line_high | 0 | 0 | 0 | 0 | 0 | 0 | 0 | a | raster line value |
| line_low | • | • | • | • | • | • | • | • | raster line value |

**a** the upper, ninth bit of the raster line

**•** the lower eight bits of the raster line

**0** must be set to zero

WARNING! Triggering a raster interrupt is not a cycle-exact timing signal at the start of a line. The 2dfx MCU sets the indication when it has reached the configured line and is processing that event. In practice this normally happens shortly after horizontal sync, but it cannot be guaranteed to occur exactly at the beginning of the line. If a program needs precise timing within a particular line, it is advisable to configure the raster interrupt for the previous line and then use timing inside the Z80 interrupt routine to reach the required point. In practice, the indication arrives no later than roughly 14 usec after horizontal sync.

## CLEAR_RASTER_INT (58h / 88)

Clears all currently active raster-interrupt requests, sets all four interrupt bits in the status register to 1, and thereby deactivates the Z80 /INT line as well.

the command has no **F9h (249)** parameter

WARNING! The raster-interrupt indication is sticky. Once triggered, the Z80 /INT input remains active until the user program issues CLEAR_RASTER_INT.

# SET_RENDER_OVERRUN_VISUAL (52h / 82)

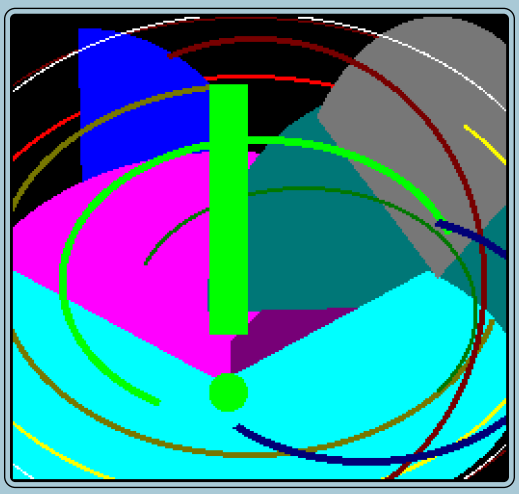
A command intended mainly for development. 2dfx continuously measures how long the current frame takes to produce. The available time is 20 ms in 50 fps mode and 40 ms in 25 fps mode. If rendering exceeds the current time budget even once – for example because of many large sprites and complex 2DPT back/front drawing – the RENDER_OVERRUN status bit changes to 0 and remains there as a sticky flag. The bit can of course be read by the user program through the status register, but during development it may be more convenient to enable the visual indication: 2dfx then draws a large exclamation mark on the topmost layer, changing colour from frame to frame.

Enabling SET_RENDER_OVERRUN_VISUAL is generally not recommended in a released user program. Its power-on default is 0, meaning disabled.

| | Data bits | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Parameter name | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description |
| mode | 0 | 0 | 0 | 0 | 0 | 0 | 0 | a | selects the operating mode |

**a** 0 = disabled
1 = enabled

**0** must be set to zero



# CLEAR_RENDER_OVERRUN (53h / 83)

Performs two actions: it clears the RENDER_OVERRUN status bit, restoring it to 1, and, if the visual overrun indication is enabled, removes the exclamation mark from the screen. The clear naturally applies only to overruns that have already occurred. If the next frame again exceeds the render budget, the status bit returns to 0 and, with SET_RENDER_OVERRUN_VISUAL enabled, the exclamation mark reappears as well.

the command has no **F9h (249)** parameter

# SHOW_BUILTIN_LOGO (90h / 144)

Draws a fixed, correctly proportioned 206×79-pixel 2dfx logo at any X/Y coordinate on the screen. The logo appears on the fourth layer, above 2DPT back, the sprites and 2DPT front. It uses three logical colours: a transparent background, the main colour of the 2dfx lettering, and the highlight colour of the curved X stroke. On the Enterprise, 2dfx does not know NICK's current palette, so the command must specify which colour indices are to be used for the two visible logo colours. The TVC has a fixed sixteen-colour palette, but for a common cross-platform command format the colour-index byte must still be supplied there; the TVC environment ignores it.

The built-in logo is useful in game or demo menus when you want to show that a program was made for 2dfx.

| | Data bits | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Parameter name | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description |
| highbits | x | x | x | x | x | a | a | b | upper bits of the coordinates |
| X_low | • | • | • | • | • | • | • | • | lower eight bits of the X coordinate |
| Y_low | • | • | • | • | • | • | • | • | lower eight bits of the Y coordinate |
| color_mapping | c | c | c | c | d | d | d | d | colour indices used by the logo |

**a** the upper two bits of the X coordinate
**b** the upper bit of the Y coordinate
**c** the colour index used for white in the 2dfx logo
**d** the colour index used for red in the 2dfx logo
**x** these bits are ignored by 2dfx


# HIDE_BUILTIN_LOGO (91h / 145)

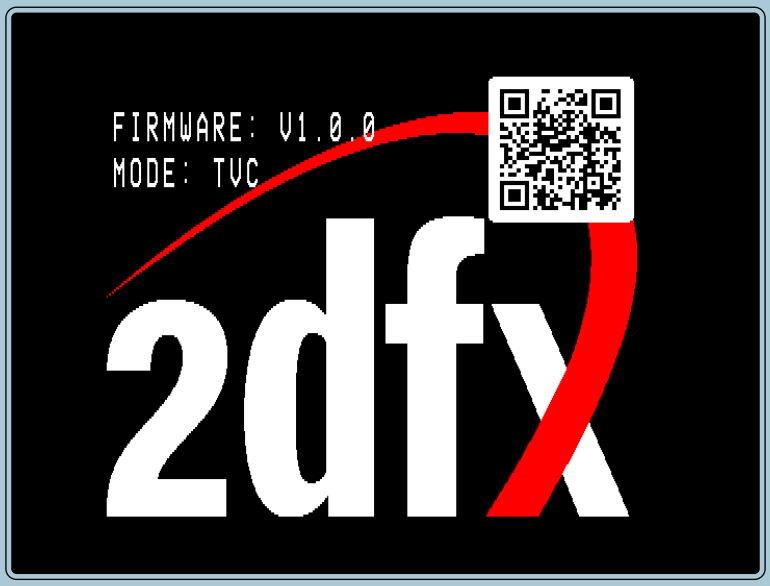
Hides the previously displayed 2dfx logo.

the command has no **F9h (249)** parameter

# SHOW_FW (8Eh / 142)

Displays the version of the firmware currently running on 2dfx, the operating mode, and the firmware download link. To leave this display, the computer must be reset.

the command has no **F9h (249)** parameter



# The sprite subsystem and its use

The most fundamental function of 2dfx is hardware sprite handling. The system can display 64 sprites at once, in either the visible or invisible area of the screen, with transparency taken into account.

A sprite may be up to `912×312` pixels in size.

Simple transformations can also be requested for sprites during rendering: **Xflip** (horizontal flip), **Yflip** (vertical flip), **Xdouble** (horizontal doubling) and **Ydouble** (vertical doubling). These operations may be combined freely.

**2dfx also provides basic collision indication.** For each sprite it can be selected independently whether it participates in collision detection. The result is written to the most significant bit of the status register during rendering and remains stable there until the next completed frame. The test is not based on bounding rectangles: a collision occurs only if at least one pair of visible, non-transparent pixels from two collision-enabled sprites actually overlaps.

Sprites are described by nine-byte **Sprite Parameter Blocks**, or **SPBs** for short. Most 2dfx commands require every parameter to be supplied explicitly, but SPB writing is different: it is enough to send only as many bytes as are minimally necessary for the desired change. The SPB is therefore arranged so that frequently modified data – such as the RRAM start address and X/Y position – comes first, while values that change less often, such as width and height, come later. Modified SPB data becomes active at the next vertical synchronisation, preventing half-updated parameters from causing display errors during rendering.

The sprite is selected by the command byte itself. Commands **00h–3Fh**, that is **0–63**, begin modifying the SPB of the sprite with the corresponding number.

A sprite may have an odd horizontal width. In that case the data must be uploaded to RRAM so that the upper four bits of the byte containing the final pixel no longer belong to the sprite; the renderer ignores them. It is still a good idea to place the transparent colour index there, even though the hardware no longer cares about that value.

Sprite data must be stored linearly by rows in RRAM. The start address points to the byte containing the first two pixels at the sprite's upper-left corner; the remaining pixels of the first row follow, then the second row, and so on. The storage required by a sprite is ((W+1) / 2) * H bytes, including the unused half-byte at the end of each row when the width is odd.

2dfx does not animate sprites by itself, but the start address of the graphics data in RRAM can be changed freely in the SPB. The same sprite can therefore display a different animation frame in the next video frame. This is the same principle familiar from classic eight-bit hardware sprite systems.

Sliding a sprite onto the screen is straightforward: begin drawing it outside the visible region, then change its X and/or Y coordinate from frame to frame. Because of the 2dfx coordinate system, the sprite can of course exist even while it is not yet visible.

The SPBs are stored in the internal RAM of 2dfx and do not consume space in RRAM.

The sprite subsystem uses just one command.

# ALTER_SPB (00h..3Fh / 0..63)

Modifies the SPB selected by the command byte. This is the only 2dfx command that does not require every parameter to be supplied each time; after transmission, parameter bytes are written immediately to the corresponding bytes of the SPB descriptor stored inside 2dfx.

| | Data bits | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Parameter name | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description |
| highbits | a | b | c | c | d | e | e | f | control bits and upper bits of the X, Y, W and H parameters |
| X_low | • | • | • | • | • | • | • | • | lower eight bits of the X coordinate |
| Y_low | • | • | • | • | • | • | • | • | lower eight bits of the Y coordinate |
| RRAM_address_low | • | • | • | • | • | • | • | • | RRAM start address bits 7..0 |
| RRAM_address_mid | • | • | • | • | • | • | • | • | RRAM start address bits 15..8 |
| RRAM_address_high | 0 | • | • | • | • | • | • | • | RRAM start address bits 22..16 |
| transform_flags | g | h | i | j | 0 | 0 | 0 | 0 | transformation flags |
| W_low | • | • | • | • | • | • | • | • | lower eight bits of width W (in pixels) |
| H_low | • | • | • | • | • | • | • | • | lower eight bits of height H (in pixels) |

**a** SPRITE_ENABLE bit; if 1, the sprite is displayed, if 0, it is not
**b** COLLISION_ENABLE bit; if 1, the sprite participates in collision detection
**c** the upper two bits of the X coordinate
**d** the upper bit of the Y coordinate
**e** the upper two bits of the width W
**f** the upper bit of the height H
**g** Xdouble bit; if 1, every sprite pixel is doubled horizontally during rendering
**h** Ydouble bit; if 1, every sprite pixel is doubled vertically during rendering
**i** Xflip bit; if 1, the sprite is mirrored horizontally during rendering
**j** Yflip bit; if 1, the sprite is mirrored vertically during rendering
**0** must be set to zero

# Introducing the 2DPT and how it works

Alongside sprites, 2dfx can also perform both simple and complex 2D graphics tasks. These elements may appear behind the sprites (*2DPT back*) or in front of them (*2DPT front*).

**2DPT** stands for **2D Parameter Table**. Both the *back* and *front* layers have a 256-element playlist. The elements of a playlist are slots: compartments into which the user can place one elementary graphics command each. Slots are numbered **0–255**.

The renderer processes the 2DPT back playlist first, from slot 0 up to at most slot 255. The sprites follow, and then the 2DPT front playlist is processed in the same way. 2dfx stores the contents of both 2DPTs, so the complete table does not have to be uploaded again for every frame; only the slots that actually change need to be modified.

2dfx performs no collision detection between the 2DPTs and the sprites.

The 2DPTs can be used to create full-resolution playfields, HUD elements, information bars and other graphics layers using any of the available colours.

Every 2DPT command has two variants: one for the back playlist and one for the front playlist. The most significant bit of the command code selects the destination: 0 modifies 2DPT back, while 1 modifies 2DPT front. In the detailed command descriptions, the first command code always refers to use in the back 2DPT and the second to use in the front 2DPT.

The 2DPTs reside in the internal RAM of 2dfx. They cannot be accessed directly by the user program and do not consume space in RRAM.

During initialisation, every slot in both 2DPTs is filled with a **`RET`** command.

# General 2DPT commands

The following commands are not direct drawing instructions. They perform a particular general task and/or define general behaviour for subsequent slots in the 2DPT.

## 2DPT - NOP (60h, E0h / 96, 224)

The NOP command does absolutely nothing. During 2DPT playback, the renderer simply continues processing with the next slot in the playlist. It can be useful for reserving positions for slots that will be populated later.

| | Data bits | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Parameter name | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description |
| slot_number | • | • | • | • | • | • | • | • | 2DPT slot number (0…255) |

## 2DPT - RET (7Fh, FFh / 127, 255)

The RET command stops further processing of the 2DPT, telling the renderer that the current 2DPT has ended and that subsequent slots should be ignored.

| | Data bits | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Parameter name | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description |
| slot_number | • | • | • | • | • | • | • | • | 2DPT slot number (0…255) |

it is advisable to use RET after the last active 2DPT command to avoid executing tasks left in any old slots

## 2DPT - SET_COLOR (61h, E1h / 97, 225)

Sets the ink and paper colour indices to be used by subsequent 2DPT commands.

| | Data bits | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Parameter name | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description |
| slot_number | • | • | • | • | • | • | • | • | 2DPT slot number (0…255) |
| colour_index | a | a | a | a | b | b | b | b | see explanation |

**a** the colour index of the ink colour

**b** the colour index of the paper colour

the default is colour index 0Fh (15) for the ink colour and the transparent colour index for the paper colour

# 2DPT - SET_XY_OFFSET (6Eh, EEh / 110, 238)

Offsets the origins (X/Y, X0/Y0 and CX/CY) of subsequent drawing commands, as well as the X1/Y1 values of LINE and ERASE_LINE, by the signed number of pixels specified by the parameters. The offset may also be negative.

This makes it possible to shift the coordinates of all subsequent drawing operations together; it is useful, for example, for moving a multi-layer group of graphics objects as one. The setting remains in effect until the next SET_XY_OFFSET command, or, if there is none, until the end of the 2DPT.

| | Data bits | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Parameter name | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description |
| slot_number | • | • | • | • | • | • | • | • | 2DPT slot number (0…255) |
| X_offset | • | • | • | • | • | • | • | • | offsets the X coordinates of subsequent commands by the specified signed integer number of pixels (-128…127) (00h = 0; 7Fh = 127; 80h = -128; FFh = -1) |
| Y_offset | • | • | • | • | • | • | • | • | offsets the Y coordinates of subsequent commands by the specified signed integer number of pixels (-128…127) (00h = 0; 7Fh = 127; 80h = -128; FFh = -1) |

WARNING! The initial SET_XY_OFFSET value is X=0 and Y=0 at the beginning of 2DPT playlist playback. The command does not use relative values: the parameters of two consecutive SET_XY_OFFSET commands are not added together. Each is always interpreted relative to the absolute X=0 and Y=0 origin, so a later command replaces the earlier offset.

# 2DPT - SKIP_NEXT (6Fh, EFh / 111, 239)

Skips the specified number of subsequent 2DPT slots so that they do not participate in drawing the screen. It is effectively the equivalent of the Z80 JR opcode, except that it can jump only forwards.

If the supplied value would move execution beyond slot 255, processing of the 2DPT ends.

This is particularly useful for graphics elements that should appear only at certain times while the commands that follow must always run. In such a case it is convenient to make the 2DPT “jump over” them; when the temporary commands are needed, only this SKIP_NEXT slot has to be replaced with NOP.

| | Data bits | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Parameter name | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description |
| slot_number | • | • | • | • | • | • | • | • | 2DPT slot number (0…255) |
| skip_cmd_no | • | • | • | • | • | • | • | • | skips the specified number of subsequent slot commands |

# 2DPT graphics primitives

The graphics primitives draw points, lines, rectangles, ellipses and various forms of these shapes. Each primitive occupies a single 2DPT slot.

The list of graphics primitives and their detailed parameters is shown in the tables below. For every 2DPT command, exactly the number of parameters specified in that command's description must be supplied.

## 2DPT - PLOT (62h, E2h / 98, 226)

Draws a one-pixel point in the current ink colour at the coordinate pair supplied as parameters.

| | Data bits | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Parameter name | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description |
| slot_number | • | • | • | • | • | • | • | • | 2DPT slot number (0…255) |
| highbits | x | x | a | a | b | x | x | x | see explanation |
| X_low | • | • | • | • | • | • | • | • | lower eight bits of the X coordinate |
| Y_low | • | • | • | • | • | • | • | • | lower eight bits of the Y coordinate |

**a** the upper two bits of the X coordinate
**b** the upper bit of the Y coordinate
**x** these bits are ignored by 2dfx

WARNING! A known processing peculiarity of NICK on the Enterprise is that it cannot display a single isolated non-transparent pixel immediately after transparent pixels. ENGINE_ON_FORCED is an exception because in that mode transparency is never actually signalled to NICK.

## 2DPT - LINE (63h, E3h / 99, 227)

Draws a line of the specified thickness in the current ink colour between the two coordinate pairs supplied as parameters. For diagonal lines, the Bresenham line algorithm is used to calculate the pixels.

| | Data bits | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Parameter name | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description |
| slot_number | • | • | • | • | • | • | • | • | 2DPT slot number (0…255) |
| highbits | a | a | b | b | c | d | d | e | see explanation |
| X0_low | • | • | • | • | • | • | • | • | lower eight bits of the start point X coordinate |
| Y0_low | • | • | • | • | • | • | • | • | lower eight bits of the start point Y coordinate |
| X1_low | • | • | • | • | • | • | • | • | lower eight bits of the end point X coordinate |
| Y1_low | • | • | • | • | • | • | • | • | lower eight bits of the end point Y coordinate |

**a** the desired line thickness - 1
**b** the upper two bits of the start point X coordinate
**c** the upper bit of the start point Y coordinate
**d** the upper two bits of the end point X coordinate
**e** the upper bit of the end point Y coordinate

## 2DPT - ERASE_LINE (64h, E4h / 100, 228)

Draws a line of the specified thickness in the current paper colour between the two coordinate pairs supplied as parameters. For diagonal lines, the Bresenham line algorithm is used to calculate the pixels.

| | Data bits | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Parameter name | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description |
| slot_number | • | • | • | • | • | • | • | • | 2DPT slot number (0…255) |
| highbits | a | a | b | b | c | d | d | e | see explanation |
| X0_low | • | • | • | • | • | • | • | • | lower eight bits of the start point X coordinate |
| Y0_low | • | • | • | • | • | • | • | • | lower eight bits of the start point Y coordinate |
| X1_low | • | • | • | • | • | • | • | • | lower eight bits of the end point X coordinate |
| Y1_low | • | • | • | • | • | • | • | • | lower eight bits of the end point Y coordinate |

**a** the desired line thickness - 1
**b** the upper two bits of the start point X coordinate
**c** the upper bit of the start point Y coordinate
**d** the upper two bits of the end point X coordinate
**e** the upper bit of the end point Y coordinate

## 2DPT - BUCKET_FILL (65h, E5h / 101, 229)

Starting from the colour index found at the supplied X/Y coordinate, fills the connected area of that same colour with the current ink colour. The fill propagates through connected pixels whose colour index matches the index at the starting X/Y coordinate, and stops at pixels of a different colour. A typical use is filling the inside of a closed shape. On larger or more complicated areas this operation can be fairly time-consuming, so for filling rectangles, rounded rectangles or ellipses it is better to use the corresponding dedicated FILL_XX commands, which are specifically optimised for speed.

| | Data bits | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Parameter name | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description |
| slot_number | • | • | • | • | • | • | • | • | 2DPT slot number (0…255) |
| highbits | x | x | a | a | b | x | x | x | see explanation |
| X_low | • | • | • | • | • | • | • | • | lower eight bits of the fill sampling-point X coordinate |
| Y_low | • | • | • | • | • | • | • | • | lower eight bits of the fill sampling-point Y coordinate |

**a** the upper two bits of the fill sampling point X coordinate
**b** the upper bit of the fill sampling point Y coordinate
**x** these bits are ignored by 2dfx

WARNING! BUCKET_FILL uses an internal, finite work stack to keep track of areas that still need to be processed. Its size is deliberately limited, so the fill operation can never cause a stack overflow or another memory error in the MCU. On a highly fragmented fill area with many branches or regions connected by narrow passages, however, this internal work stack may become full. In that case the operation still terminates safely, but part of the area may remain unfilled. If this happens, simply issue BUCKET_FILL again at an X/Y coordinate inside one of the remaining unfilled regions.

# 2DPT - RECT (66h, E6h / 102, 230)

Draws a rectangle with the specified line thickness. Its outline is drawn in the ink colour selected by SET_COLOR. If the paper colour specified by SET_COLOR is not the transparent colour, the interior of the rectangle is filled with the paper colour. If the paper colour is the transparent colour index, only the outline is drawn.

| | Data bits | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Parameter name | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description |
| slot_number | • | • | • | • | • | • | • | • | 2DPT slot number (0…255) |
| highbits | a | a | b | b | c | d | d | e | see explanation |
| X_low | • | • | • | • | • | • | • | • | lower eight bits of the upper-left corner X coordinate |
| Y_low | • | • | • | • | • | • | • | • | lower eight bits of the upper-left corner Y coordinate |
| W_low | • | • | • | • | • | • | • | • | lower eight bits of width W |
| H_low | • | • | • | • | • | • | • | • | lower eight bits of height H |

**a** the desired line thickness - 1
**b** the upper two bits of the upper-left corner X coordinate
**c** the upper bit of the upper-left corner Y coordinate
**d** the upper two bits of the width (W) in pixels
**e** the upper bit of the height (H) in pixels

line thickness takes the 2:1 pixel aspect ratio into account, so vertical sides are twice as thick as horizontal ones

# 2DPT - FILL_RECT (67h, E7h / 103, 231)

Draws a rectangle in the current ink colour and fills it with the same ink colour.

| | Data bits | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Parameter name | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description |
| slot_number | • | • | • | • | • | • | • | • | 2DPT slot number (0…255) |
| highbits | x | x | a | a | b | c | c | d | see explanation |
| X_low | • | • | • | • | • | • | • | • | lower eight bits of the upper-left corner X coordinate |
| Y_low | • | • | • | • | • | • | • | • | lower eight bits of the upper-left corner Y coordinate |
| W_low | • | • | • | • | • | • | • | • | lower eight bits of width W |
| H_low | • | • | • | • | • | • | • | • | lower eight bits of height H |

**a** the upper two bits of the upper-left corner X coordinate
**b** the upper bit of the upper-left corner Y coordinate
**c** the upper two bits of the width (W) in pixels
**d** the upper bit of the height (H) in pixels
**x** these bits are ignored by 2dfx

# 2DPT - ROUND_RECT (68h, E8h / 104, 232)

Draws a rounded rectangle with the specified line thickness. Its outline is drawn in the ink colour previously selected by SET_COLOR. If the paper colour specified by SET_COLOR is not the transparent colour, the interior of the rectangle is filled with the paper colour. If the paper colour is the transparent colour index, only the outline is drawn. Each corner is represented by a quarter ellipse according to the RX and RY radii.

| | Data bits | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Parameter name | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description |
| slot_number | • | • | • | • | • | • | • | • | 2DPT slot number (0…255) |
| highbits | a | a | b | b | c | d | d | e | see explanation |
| X_low | • | • | • | • | • | • | • | • | lower eight bits of the upper-left corner X coordinate |
| Y_low | • | • | • | • | • | • | • | • | lower eight bits of the upper-left corner Y coordinate |
| W_low | • | • | • | • | • | • | • | • | lower eight bits of width W |
| H_low | • | • | • | • | • | • | • | • | lower eight bits of height H |
| X_radius | • | • | • | • | • | • | • | • | horizontal radius (RX) |
| Y_radius | • | • | • | • | • | • | • | • | vertical radius (RY) |

**a** the desired line thickness - 1
**b** the upper two bits of the upper-left corner X coordinate
**c** the upper bit of the upper-left corner Y coordinate
**d** the upper two bits of the width (W) in pixels
**e** the upper bit of the height (H) in pixels

# 2DPT - FILL_ROUND_RECT (69h, E9h / 105, 233)

Draws a rounded rectangle in the current ink colour and fills it with the same ink colour. Each corner is represented by a quarter ellipse according to the RX and RY radii.

| | Data bits | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Parameter name | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description |
| slot_number | • | • | • | • | • | • | • | • | 2DPT slot number (0…255) |
| highbits | x | x | a | a | b | c | c | d | see explanation |
| X_low | • | • | • | • | • | • | • | • | lower eight bits of the upper-left corner X coordinate |
| Y_low | • | • | • | • | • | • | • | • | lower eight bits of the upper-left corner Y coordinate |
| W_low | • | • | • | • | • | • | • | • | lower eight bits of width W |
| H_low | • | • | • | • | • | • | • | • | lower eight bits of height H |
| X_radius | • | • | • | • | • | • | • | • | horizontal radius (RX) |
| Y_radius | • | • | • | • | • | • | • | • | vertical radius (RY) |

**a** the upper two bits of the upper-left corner X coordinate
**b** the upper bit of the upper-left corner Y coordinate
**c** the upper two bits of the width (W) in pixels
**d** the upper bit of the height (H) in pixels
**x** these bits are ignored by 2dfx

# 2DPT - ELLIPSE (6Ah, EAh / 106, 234)

Draws an ellipse with the specified line thickness. Its outline is drawn in the ink colour selected by SET_COLOR. If the paper colour specified by SET_COLOR is not the transparent colour, the interior of the ellipse is filled with the paper colour. If the paper colour is the transparent colour index, only the outline is drawn. The parameters are the ellipse centre (CX/CY), horizontal radius (RX) and vertical radius (RY).

| | Data bits | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Parameter name | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description |
| slot_number | • | • | • | • | • | • | • | • | 2DPT slot number (0…255) |
| highbits | a | a | b | b | c | d | d | e | see explanation |
| CX_low | • | • | • | • | • | • | • | • | lower eight bits of the centre X coordinate |
| CY_low | • | • | • | • | • | • | • | • | lower eight bits of the centre Y coordinate |
| X_radius | • | • | • | • | • | • | • | • | lower eight bits of horizontal radius RX |
| Y_radius | • | • | • | • | • | • | • | • | lower eight bits of vertical radius RY |

**a** the desired line thickness - 1
**b** the upper two bits of the centre X coordinate
**c** the upper bit of the centre Y coordinate
**d** the upper two bits of the horizontal radius (RX)
**e** the upper bit of the vertical radius (RY)

# 2DPT - FILL_ELLIPSE (6Bh, EBh / 107, 235)

Draws an ellipse in the current ink colour and fills it with the same ink colour. The parameters are the ellipse centre (CX/CY), horizontal radius (RX) and vertical radius (RY).

| | Data bits | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Parameter name | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description |
| slot_number | • | • | • | • | • | • | • | • | 2DPT slot number (0…255) |
| highbits | x | x | a | a | b | c | c | d | see explanation |
| CX_low | • | • | • | • | • | • | • | • | lower eight bits of the centre X coordinate |
| CY_low | • | • | • | • | • | • | • | • | lower eight bits of the centre Y coordinate |
| X_radius | • | • | • | • | • | • | • | • | lower eight bits of the horizontal radius |
| Y_radius | • | • | • | • | • | • | • | • | lower eight bits of the vertical radius |

**a** the upper two bits of the centre X coordinate
**b** the upper bit of the centre Y coordinate
**c** the upper two bits of the horizontal radius (RX)
**d** the upper bit of the vertical radius (RY)
**x** these bits are ignored by 2dfx

# 2DPT - ARC (6Ch, ECh / 108, 236)

Draws an arc of the specified line thickness in the current ink colour. The parameters are similar to those of an ellipse: centre (X/Y), horizontal radius (RX) and vertical radius (RY), with the addition of the start_angle and end_angle parameters.

The one-byte angle parameters divide the full 360° circle into equal steps. Key values are: 00h (0) – 0°, 12 o'clock; 40h (64) – 90°, 3 o'clock; 80h (128) – 180°, 6 o'clock; C0h (192) – 270°, 9 o'clock. Because of this mapping, FFh (255) is not the same as 0°.

If the two angle parameters are identical, no arc is drawn.

| | Data bits | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Parameter name | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description |
| slot_number | • | • | • | • | • | • | • | • | 2DPT slot number (0…255) |
| highbits | a | a | b | b | c | d | d | e | see explanation |
| CX_low | • | • | • | • | • | • | • | • | lower eight bits of the centre X coordinate |
| CY_low | • | • | • | • | • | • | • | • | lower eight bits of the centre Y coordinate |
| X_radius | • | • | • | • | • | • | • | • | lower eight bits of the horizontal radius |
| Y_radius | • | • | • | • | • | • | • | • | lower eight bits of the vertical radius |
| start_angle | • | • | • | • | • | • | • | • | start angle value |
| end_angle | • | • | • | • | • | • | • | • | end angle value |

**a** the desired line thickness - 1
**b** the upper two bits of the centre X coordinate
**c** the upper bit of the centre Y coordinate
**d** the upper two bits of the horizontal radius (RX)
**e** the upper bit of the vertical radius (RY)

WARNING! ARC is one of the most computationally expensive primitives. In some cases it is worth replacing it with a pre-drawn shape or bitmap uploaded to RRAM.

# 2DPT - PIE (6Dh, EDh / 109, 237)

Draws a pie slice in the current ink colour. The parameters are similar to those of an ellipse: centre (X/Y), horizontal radius (RX) and vertical radius (RY), with the addition of the start_angle and end_angle parameters. The endpoints corresponding to the two angles are connected to the centre to form the pie slice, which is then filled with the ink colour.

The one-byte angle parameters divide the full circle into equal steps. Key values are: 00h (0) – 0°, 12 o'clock; 40h (64) – 90°, 3 o'clock; 80h (128) – 180°, 6 o'clock; C0h (192) – 270°, 9 o'clock. Because of this mapping, FFh (255) is not the same as 0°.

If the two angle parameters are identical, nothing is drawn.

| | Data bits | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Parameter name | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description |
| slot_number | • | • | • | • | • | • | • | • | 2DPT slot number (0…255) |
| highbits | x | x | a | a | b | c | c | d | see explanation |
| CX_low | • | • | • | • | • | • | • | • | lower eight bits of the centre X coordinate |
| CY_low | • | • | • | • | • | • | • | • | lower eight bits of the centre Y coordinate |
| X_radius | • | • | • | • | • | • | • | • | lower eight bits of the horizontal radius |
| Y_radius | • | • | • | • | • | • | • | • | lower eight bits of the vertical radius |
| start_angle | • | • | • | • | • | • | • | • | start angle value |
| end_angle | • | • | • | • | • | • | • | • | end angle value |

**a** the upper two bits of the centre X coordinate
**b** the upper bit of the centre Y coordinate
**c** the upper two bits of the horizontal radius (RX)
**d** the upper bit of the vertical radius (RY)
**x** these bits are ignored by 2dfx

WARNING! PIE is one of the most computationally expensive primitives. In some cases it is worth replacing it with a pre-drawn shape or bitmap uploaded to RRAM.

# 2DPT resource-based commands

The 2DPT supports not only directly drawn primitives, but also so-called resource-based commands.

These commands perform more complex tasks and assume that the required data has already been placed in RRAM. This may look slightly indirect at first, but the logic is simple: first we upload and define the resource, and later the 2DPT command merely refers to it.

2dfx supports four kinds of resource-based 2DPT command. The following subsections introduce them.

# 2DPT BLIT – fast drawing of bitmaps and backgrounds

The **BLIT** command is used to draw a predefined rectangular image area quickly. It copies image data stored from a specified RRAM start address, with width W and height H, to the specified X/Y position on the screen. Its operation is similar in many ways to a sprite, but for speed it follows simpler rules: it handles whole bytes only, which means an even number of horizontal pixels, and aligns the destination X coordinate to an even value, also taking into account the offset set by **SET_XY_OFFSET**. It can also optionally perform transparency testing: either every pixel is copied, including pixels containing the transparent colour index, or transparency is tested in the same way as for sprites and the bitmap is copied accordingly. Naturally, the latter is slower than the former.

The most obvious use of **BLIT** is to display a prepared image or background element that has already been uploaded to RRAM.

To use **BLIT**, the image data must first be uploaded to RRAM using the usual UPLOAD commands. **DEFINE_BITMAP** then assigns it a logical identifier and specifies the bitmap's start address, width and height. A later 2DPT **BLIT** command refers only to this identifier and needs to specify only where the bitmap should appear on the screen. If **BLIT** receives an odd X coordinate, it automatically aligns it to an even value; in other words, the lowest bit is ignored.

A **BLIT** command may be placed in any 2DPT slot. Up to 256 bitmaps can be defined at a time, and any of them may be reused while the program is running.

# DEFINE_BITMAP (5Ch / 92)

Assigns a logical identifier to a bitmap previously uploaded to RRAM. The 2DPT BLIT command uses this identifier to obtain the bitmap's RRAM start address, width and height. Up to 256 bitmap definitions can exist at the same time, and any of them may be reused while the system is running.

| | Data bits | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Parameter name | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description |
| bitmap_id | • | • | • | • | • | • | • | • | requested bitmap identifier (0..255) |
| RRAM_address_low | • | • | • | • | • | • | • | • | RRAM start address bits 7..0 |
| RRAM_address_mid | • | • | • | • | • | • | • | • | RRAM start address bits 15..8 |
| RRAM_address_high | 0 | • | • | • | • | • | • | • | RRAM start address bits 22..16 |
| highbits | x | x | x | x | x | a | a | b | see explanation |
| W_low | • | • | • | • | • | • | • | x | lower eight bits of bitmap width W – 2dfx ignores the lowest bit |
| H_low | • | • | • | • | • | • | • | • | lower eight bits of bitmap height H |

**a** the upper two bits of the width (W) in pixels
**b** the upper bit of the height (H) in pixels
**x** these bits are ignored by 2dfx
**0** must be set to zero

# 2DPT - BLIT (71h, F1h / 113, 241)

Copies a bitmap previously uploaded to RRAM and identified by DEFINE_BITMAP to the specified screen coordinate (X/Y). BLIT performs transparency testing only when requested. With transparency testing disabled, pixels are copied according to their stored colour indices and overwrite the pixels in the destination area. When transparency testing is enabled, pixels are drawn in the same way as for sprites. This makes the copy operation somewhat slower.

| | Data bits | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Parameter name | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description |
| slot_number | • | • | • | • | • | • | • | • | 2DPT slot number (0…255) |
| highbits | a | x | b | b | c | x | x | x | see explanation |
| X_low | • | • | • | • | • | • | • | • | lower eight bits of the bitmap upper-left X coordinate on screen |
| Y_low | • | • | • | • | • | • | • | • | lower eight bits of the bitmap upper-left Y coordinate on screen |
| bitmap_id | • | • | • | • | • | • | • | • | bitmap identifier (0..255) assigned by DEFINE_BITMAP |

**a** enable transparency testing when set to 1
**b** the upper two bits of the upper-left corner X coordinate
**c** the upper bit of the upper-left corner Y coordinate
**x** these bits are ignored by 2dfx
**0** must be set to zero

WARNING! For speed, BLIT copies only an even number of pixels and performs transparency testing only when explicitly requested. Otherwise every pixel of the bitmap is copied unchanged into its bounding rectangle.

# 2DPT FONT – displaying text and strings

2dfx can also display arbitrary text. Characters are drawn using the currently selected foreground colour, that is, the ink colour set by **SET_COLOR**.

2dfx contains no built-in character set, but up to 64 different fonts can be defined. A font may contain 128 character shapes. The 1bpp pattern data for the characters must first be uploaded to RRAM using the appropriate UPLOAD command.

In a font's 1bpp bitmap, the most significant bit represents the leftmost pixel and the least significant bit the rightmost pixel.

Character cells are built from basic 8×8-pixel units and may be horizontal and vertical multiples of that size. It is important to note that this does not mean automatic scaling: a larger cell size requires more genuinely stored pixels for each character. A character can therefore be as large as 128×128 pixels if that is what the font definition describes.

In RRAM, the font data begins with the definition of character 0, followed in order by character 1, character 2, and so on up to character 127. A complete 8×8 font requires 128 * 8 bytes, or 1024 bytes, while a 64×32 font requires 128 * 8 * 32 bytes, or 32768 bytes.

The **DEFINE_FONT** resource command associates font data already uploaded to RRAM with a logical font identifier, and also specifies the size of the character cells. It is sensible to issue this command immediately after the upload, so that later **DRAW_TEXT** commands can simply refer to the prepared character set.

To display text, the character string itself must also be uploaded to RRAM using **UPLOAD_RAW**. A string may contain at most 255 characters; for a shorter string, a terminating byte with the value **80h (128)** must mark the end. Characters that can actually be displayed lie in the range **00h–7Fh (0–127)**. There is no separate command for uploading strings, and it is important that they enter RRAM as RAW data so that 2dfx does not rearrange their bits.

The **DRAW_TEXT** command may be placed in any 2DPT slot. Its parameters specify the font identifier to use, the screen position of the first character, and the RRAM start address of the string to display.

**DRAW_TEXT** reads the string from RRAM byte by byte and continues drawing characters until it encounters the 80h terminator byte or reaches the maximum of 255 characters.

# DEFINE_FONT (5Bh / 91)

Associates a font bitmap already uploaded to RRAM with a logical font identifier. Up to 64 font identifiers can exist at the same time, and any of them may be redefined while the system is running. The definition also specifies the maximum size of one character image in the font (up to 128×128 pixels). A font may define 128 characters.

| | Data bits | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Parameter name | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description |
| font_id | 0 | 0 | • | • | • | • | • | • | requested font identifier (0…63) |
| RRAM_address_low | • | • | • | • | • | • | • | • | RRAM start address bits 7..0 |
| RRAM_address_mid | • | • | • | • | • | • | • | • | RRAM start address bits 15..8 |
| RRAM_address_high | 0 | • | • | • | • | • | • | • | RRAM start address bits 22..16 |
| width_height_units | a | a | a | a | b | b | b | b | see explanation |

**a** character width in units of eight pixels – 1 (0 = 8 pixels, 1 = 16 pixels, etc.)

**b** character height in units of eight pixels – 1 (0 = 8 pixels, 1 = 16 pixels, etc.)

**0** must be set to zero

# 2DPT - DRAW_TEXT (70h, F0h / 112, 240)

Displays text starting at the specified RRAM address using the font selected by its font identifier, beginning at the supplied screen coordinate (X/Y). Characters are drawn in the current ink colour. If the current paper colour is not the transparent colour, the area behind the characters is filled with the paper colour. The text stored in RRAM is read and displayed until the first 80h (128) terminator byte, or up to 255 characters if no terminator is encountered.

| | Data bits | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Parameter name | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description |
| slot_number | • | • | • | • | • | • | • | • | 2DPT slot number (0…255) |
| highbits | x | x | a | a | b | x | x | x | see explanation |
| X_low | • | • | • | • | • | • | • | • | lower eight bits of the upper-left corner X coordinate |
| Y_low | • | • | • | • | • | • | • | • | lower eight bits of the upper-left corner Y coordinate |
| font_id | 0 | 0 | • | • | • | • | • | • | selected font identifier |
| RRAM_address_low | • | • | • | • | • | • | • | • | RRAM start address of the stored text, bits 7..0 |
| RRAM_address_mid | • | • | • | • | • | • | • | • | RRAM start address of the stored text, bits 15..8 |
| RRAM_address_high | 0 | • | • | • | • | • | • | • | RRAM start address of the stored text, bits 22..16 |

**a** the upper two bits of the start point X coordinate

**b** the upper bit of the start point Y coordinate

**x** these bits are ignored by 2dfx

**0** must be set to zero

# 2DPT PATTERN_FILL – filling rectangles with patterns

With 2dfx, rectangular areas can also be filled with a repeating pattern. A fill pattern is a fixed 16×8-pixel, 4bpp image element and therefore occupies exactly 64 bytes. Several patterns may be stored consecutively in RRAM; together they form a pattern table.

The first step is therefore to upload the image data for the patterns to RRAM. The **DEFINE_PATTERN** resource command then specifies the start address of the pattern table and assigns it a table_id. Up to 64 different table_id values can be defined at the same time.

A pattern table may contain up to 256 consecutive patterns. The **PATTERN_FILL** command therefore selects two things: first the table defined under the desired table_id, and then, within that table, the specific pattern numbered 0..255 using pattern_number. The address of the selected pattern follows from the table's start address and the pattern number; the programmer does not have to create a separate resource definition for every single pattern.

**PATTERN_FILL** may be placed in any slot of 2DPT back or front. Starting at the specified X/Y coordinate, the command fills a rectangle of width W and height H by repeating the selected 16×8 pattern. The pattern repeats automatically both horizontally and vertically until the entire specified area has been filled.

The rectangle being filled does not have to be an exact multiple of the 16×8 pattern. If the final repetition would extend beyond the specified W×H area, 2dfx draws only the pixels that fall inside the rectangle. Clipping at the edge of the framebuffer does not change the pattern phase either: the pattern remains aligned to the originally specified X/Y starting point.

**PATTERN_FILL** does not perform transparency testing: every pixel of the selected pattern is drawn.

# DEFINE_PATTERN (5Dh / 93)

Assigns a logical identifier to fixed 16×8 fill patterns already uploaded to RRAM. One identifier may refer to up to 256 different consecutive 16×8 patterns in RRAM. Up to 64 pattern definitions can exist at the same time, and any of them may be reused while the system is running.

| | Data bits | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Parameter name | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description |
| table_id | 0 | 0 | • | • | • | • | • | • | fill-pattern table identifier |
| RRAM_address_low | • | • | • | • | • | • | • | • | RRAM start address of the fill patterns, bits 7..0 |
| RRAM_address_mid | • | • | • | • | • | • | • | • | RRAM start address of the fill patterns, bits 15..8 |
| RRAM_address_high | 0 | • | • | • | • | • | • | • | RRAM start address of the fill patterns, bits 22..16 |

**0** must be set to zero

# 2DPT - PATTERN_FILL (72h, F2h / 114, 242)

Fills the rectangle specified by this command using the selected fill pattern previously uploaded to RRAM and identified with DEFINE_PATTERN. The command is optimised for speed, so only even X coordinates and even W widths are permitted; 2dfx ignores the lowest bit of both parameters. Transparency is not handled: pixels are copied from RRAM one by one.

| | Data bits | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Parameter name | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description |
| slot_number | • | • | • | • | • | • | • | • | 2DPT slot number (0…255) |
| highbits | x | x | a | a | b | c | c | d | see explanation |
| X_low | • | • | • | • | • | • | • | • | lower eight bits of the upper-left X coordinate of the rectangle to fill on screen |
| Y_low | • | • | • | • | • | • | • | • | lower eight bits of the upper-left Y coordinate of the rectangle to fill on screen |
| W_low | • | • | • | • | • | • | • | x | lower eight bits of the width of the rectangle to fill – 2dfx ignores the lowest bit |
| H_low | • | • | • | • | • | • | • | • | lower eight bits of the height of the rectangle to fill |
| table_id | 0 | 0 | • | • | • | • | • | • | fill-pattern table identifier |
| pattern_number | • | • | • | • | • | • | • | • | number of the fill pattern to use |

**a** the upper two bits of the upper-left corner X coordinate
**b** the upper bit of the upper-left corner Y coordinate
**c** the upper two bits of the width (W) in pixels
**d** the upper bit of the height (H) in pixels
**0** must be set to zero
**x** these bits are ignored by 2dfx

# 2DPT TILEMAP – drawing tile-based playfields

At first sight, **TILEMAP** may be the most complex of the 2DPT command structures, but it is also one of the most useful. It can build complete levels or larger playfield sections from predefined repeating images – tiles. The complete level may be much larger than the visible image area; **TILEMAP** always draws a selected window of it, the so-called viewport. This window can be moved simply by changing the parameters of a single 2DPT command.

The tilemap system consists of three main commands. Two of these are resource-based definitions, meaning that they set up the data to be used globally, independently of the current rendering operation.

Tiles are fixed 16×8-pixel image elements. One tile occupies 64 bytes in RRAM, and the tiles follow one another consecutively. **DEFINE_TILESET** specifies the logical identifier of the tileset and the RRAM address at which its tile graphics begin. One TILESET can contain up to 256 different tiles, and up to 64 TILESETs can be defined at the same time. A level structure within one TILESET can therefore use at most 256 tile types, but several TILESETs may appear on screen if several **TILEMAP** commands are placed in the 2DPT.

Before using **DEFINE_TILESET**, the tile image data must be uploaded to RRAM using one of the usual UPLOAD/UNZX1 workflows. From the start address, the tiles follow at a fixed stride: the address of the tile with identifier tile_id is tileset_base + tile_id * 64. Later tilemap data refers to tiles by numbers from 00h–FFh, that is 0–255.

**DEFINE_TILEMAP** describes a complete map. It can contain up to 256×256 tiles, so it may be thought of as the entire playfield. Before it can be used, the map cells must be uploaded to RRAM: each byte gives the number of the tile at the corresponding position. The map data occupies W×H bytes stored row by row, with each byte containing the 0..255 identifier of the tile used in that cell.

**TILEMAP** itself is the drawing command placed in the 2DPT. It determines which part of the previously defined map appears on the screen. Its parameters specify the tilemap identifier to use, the upper-left X/Y coordinate of the viewport on the screen, the X/Y position of the upper-left source tile within the map, and the number of tiles to draw horizontally and vertically.

The first step of the workflow is to upload the tile graphics to RRAM. The **DEFINE_TILESET** command then associates the start address of the uploaded graphics data with a logical tileset identifier.

The second step is to upload the complete map to RRAM, where each byte represents the number of the tile used in that cell. **DEFINE_TILEMAP** associates this with a tilemap identifier; the full width and height of the map are specified here as well.

The final step is display. The **TILEMAP** command can be written into any 2DPT slot and specifies which tilemap to use, where on the screen to draw it, and how large a visible window to display. The size of the viewport is given not in pixels but as a number of tiles horizontally and vertically.

# DEFINE_TILESET (5Eh / 94)

Associates a tileset of 16×8-pixel tiles previously uploaded to RRAM with an identifier called tileset_id. The tileset may contain 256 different tile patterns.

The tile patterns must be stored consecutively and contiguously in RRAM. Each tile occupies 64 bytes, so a complete tileset is 16 kB. Up to 64 tilesets can be defined at the same time.

| | Data bits | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Parameter name | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description |
| tileset_id | 0 | 0 | • | • | • | • | • | • | tileset identifier (0…63) |
| RRAM_address_low | • | • | • | • | • | • | • | • | RRAM start address of the tile images, bits 7..0 |
| RRAM_address_mid | • | • | • | • | • | • | • | • | RRAM start address of the tile images, bits 15..8 |
| RRAM_address_high | 0 | • | • | • | • | • | • | • | RRAM start address of the tile images, bits 22..16 |

**0** must be set to zero

# DEFINE_TILEMAP (5Fh / 95)

Defines up to 32 different tilemaps, each containing at most 256×256 tiles, identified by tilemap_id.

The map data must first be uploaded to RRAM as W * H bytes stored row by row, where each byte contains the identifier of the tile used in the corresponding map cell.

A map can be associated with one tileset at a time. Several tilemap identifiers may use the same tileset_id.

| | Data bits | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Parameter name | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description |
| tilemap_id | 0 | 0 | 0 | • | • | • | • | • | identifier of the tilemap to define (0..31) |
| RRAM_address_low | • | • | • | • | • | • | • | • | RRAM start address of the map data, bits 7..0 |
| RRAM_address_mid | • | • | • | • | • | • | • | • | RRAM start address of the map data, bits 15..8 |
| RRAM_address_high | 0 | • | • | • | • | • | • | • | RRAM start address of the map data, bits 22..16 |
| W_cell | • | • | • | • | • | • | • | • | horizontal size of the map in tiles – 1 |
| H_cell | • | • | • | • | • | • | • | • | vertical size of the map in tiles – 1 |
| tileset_id | 0 | 0 | • | • | • | • | • | • | tileset identifier (0…63) |

**0** must be set to zero

# 2DPT - TILEMAP (73h, F3h / 115, 243)

Draws a selected part of a previously defined and uploaded tilemap, identified by tilemap_id, onto the screen.

The command specifies the tilemap_id to use, the screen coordinates (X/Y) of the drawing origin, the position of the upper-left source tile within the map (in tile units), and the number of tiles to draw horizontally and vertically.

2dfx copies tile pixels to the destination area byte-for-byte without transparency testing.

| | Data bits | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Parameter name | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description |
| slot_number | • | • | • | • | • | • | • | • | 2DPT slot number (0…255) |
| highbits | a | a | a | a | a | b | b | c | see explanation |
| X_low | • | • | • | • | • | • | • | • | lower eight bits of the upper-left corner X coordinate |
| Y_low | • | • | • | • | • | • | • | • | lower eight bits of the upper-left corner Y coordinate |
| X_source | • | • | • | • | • | • | • | • | X position of the first source tile in the map, measured in tiles |
| Y_source | • | • | • | • | • | • | • | • | Y position of the first source tile in the map, measured in tiles |
| C_draw | • | • | • | • | • | • | • | • | number of tiles to draw horizontally |
| R_draw | • | • | • | • | • | • | • | • | number of tiles to draw vertically |

**a** number of the tilemap (tilemap_id) to use (0..31)

**b** the upper two bits of the upper-left corner X coordinate

**c** the upper bit of the upper-left corner Y coordinate

# Using 2dfx in practice

# Initialisation

When the computer is powered on, or when the computer's **RESET** button is pressed, 2dfx enters an inactive sleep mode. None of the status bits returns a valid value, and 2dfx does not respond to ordinary commands.

In sleep mode, 2dfx recognises and executes only the **SET_ENGINE_MODE** command. When user software starts, 2dfx must therefore be woken up and put to work by selecting either **ENGINE_ON** or **ENGINE_ON_FORCED**:

```
OUT 248, 80
OUT 249, 1          ! normal mode
```

or

```
OUT 248, 80
OUT 249, 255        ! forced mode
```

The difference between the two modes is explained in the description of the **SET_ENGINE_MODE** command.

After waking, 2dfx runs its internal initialisation routines. **In the worst case this can take two seconds.**

After that, we can read the status register and test its sixth bit (**AND 40h**) to verify that a 2dfx card is actually present in the machine and is operational: for an initialised, working 2dfx, the status bit will be **zero**.

**During the roughly two-second initialisation period, 2dfx does not communicate or execute commands. The user program must wait, either with a timed delay or by repeatedly reading the status register (with a suitable timeout to avoid an infinite loop if no 2dfx is installed) and checking for the sixth bit to become zero.**

It is, of course, possible that 2dfx is already active when our program starts because a previous user program has used it. We can determine this by reading the status register (**F8h**) and examining its sixth bit (**AND 40h**). If the bit is zero, 2dfx is operational. In that case, to clear any data left behind by the previous program in RRAM, the 2DPTs and the SPBs, issue **SYS_RESET** and wait until bit zero of the status register (**AND 01h**) changes to 1.

# Recommended workflow

When the user program starts, the first step is to perform the initialisation procedure described in the previous chapter.

If the presence check succeeds, set the required global parameters: the transparent colour, FPS, and any other settings that should differ from their power-on defaults. Likewise, if raster interrupts are going to be used, it is sensible to set up the Z80 interrupt-handling routine in the user program at this stage.

The next sensible step is to load the contents of RRAM. This can take some time for larger amounts of data, so it is worth giving the user some feedback while it happens. The built-in 2dfx logo is useful here, for example (see **SHOW_BUILTIN_LOGO**), because apart from a few parameter bytes it requires no separate data upload.

Once RRAM has been loaded, set up the definitions used by resource-based commands: fonts, bitmaps, tilemaps and any other **DEFINE_XX**-style resources. Later 2DPT commands can then refer to ready-to-use resources.

The 2DPT back and front playlists can be built next. A useful trick, especially when programming from BASIC, is to leave slot 0 alone temporarily, or set it only at the very end. At power-on and after **SYS_RESET**, every 2DPT slot contains a **RET** command. If the renderer finds **RET** in slot 0, it immediately stops processing that playlist. This avoids partially built graphics appearing while the 2DPT is being populated. Once the other slots have been set, change slot 0 to **NOP** or to the desired first graphics command as the final step; in the next frame, the renderer will see the complete playlist.

The same idea can be used when preparing SPBs. Load the SPBs of the sprites used by the program in their complete nine-byte form, but leave the SPRITE_ENABLE bit of the first byte at 0. Later it is enough to resend just the first SPB byte with SPRITE_ENABLE set to 1, and the sprite appears immediately with all of its previously prepared parameters.

At this point most of the initialisation is complete. The user program can concentrate on gameplay, demo logic or application logic, while 2dfx takes care of a substantial part of the graphics work.

When the program exits normally, it is good practice to issue **SET_ENGINE_MODE** with a zero parameter, that is ENGINE_OFF.

# Upload guide

The chapter on using RRAM explains that 2dfx supports several upload and decompression modes. The overview table below helps choose the correct combinations.

The table contains all possible combinations. **`TILEMAP`** map data, FONT character-set data, and text/string data for **`DRAW_TEXT`** must always be RAW data.

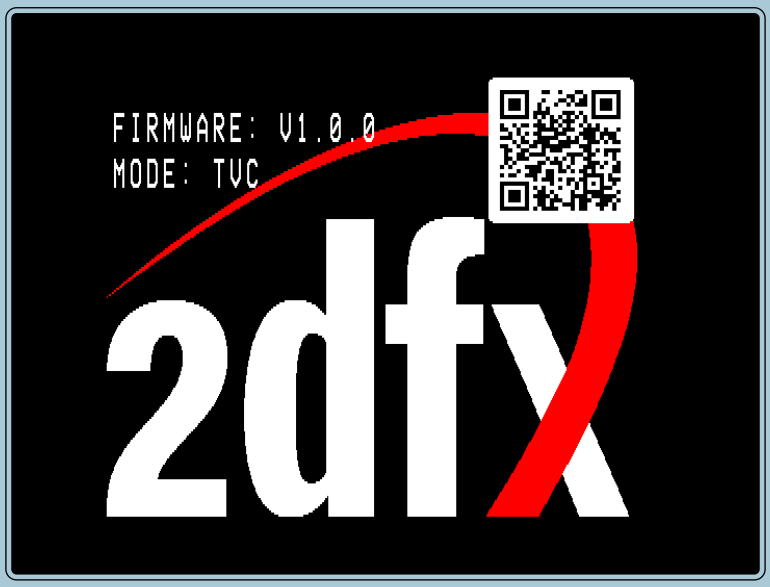
| source data type | source format | compressed? | upload command | decompression command |
|---|---|---|---|---|
| image data for sprites, BLIT and PATTERN_FILL; TILESET tile image data | 2dfx internal | no | UPLOAD_RAW | – |
| image data for sprites, BLIT and PATTERN_FILL; TILESET tile image data | TVC | no | UPLOAD_TVC | – |
| image data for sprites, BLIT and PATTERN_FILL; TILESET tile image data | NICK | no | UPLOAD_NICK | – |
| TILEMAP map data; FONT character-set data; text/string for DRAW_TEXT | 2dfx internal | no | UPLOAD_RAW | – |
| image data for sprites, BLIT and PATTERN_FILL; TILESET tile image data | 2dfx internal | yes | UPLOAD_RAW | UNZX1_RAW |
| image data for sprites, BLIT and PATTERN_FILL; TILESET tile image data | TVC | yes | UPLOAD_RAW | UNZX1_TVC |
| image data for sprites, BLIT and PATTERN_FILL; TILESET tile image data | NICK | yes | UPLOAD_RAW | UNZX1_NICK |
| TILEMAP map data; FONT character-set data; text/string for DRAW_TEXT | 2dfx internal | yes | UPLOAD_RAW | UNZX1_RAW |

# Firmware update

Although testing consumed most of the development time, **2dfx is most definitely not guaranteed to be bug-free**. Smaller or larger bugs may still turn up, and I will of course do my best to fix them based on user feedback. Naturally, the same applies to this handbook.

The latest version of the 2dfx operating firmware is always available at
**`https://2dfx.net/firmware/latest`**.

2dfx can display the version of the firmware currently running by means of the **`SHOW_FW`** command. After the command is issued, the screen shows the 2dfx logo, the current firmware version, the operating mode (TVC or Enterprise), and a QR code pointing to the download link above.




After issuing **`SHOW_FW`**, the TVC or Enterprise must be reset before normal operation can resume.

The 2dfx cards for the TVC and the Enterprise run the same operating software (unified firmware) and use the same drawing routines, so there is no separate TVC or Enterprise firmware version.

Compare the software version displayed by **SHOW_FW** with the version number shown on the website. If they differ, the 2dfx firmware can be updated in the following very simple way:

Download the file with the .uf2 extension from the website, then connect your computer (PC or Mac) to the USB-C socket on 2dfx using an ordinary USB cable. Next switch off the TVC or Enterprise, then switch it on again while holding down the single push-button on the 2dfx circuit board. An RP2350 drive will shortly appear on the PC; at that point, release the button. Simply copy the downloaded .uf2 file onto the RP2350 drive. The firmware update takes place automatically after the copy. The whole operation takes no more than a few seconds; the RP2350 drive then disappears and 2dfx resets itself to its power-on state, now running the new firmware. After performing the initialisation steps, the new version can be verified with **SHOW_FW**.

It is possible that the computer reports an error while copying. In that case, repeat the power-off/power-on procedure and try copying the .uf2 file again. Experience shows that after a few tens of seconds of inactivity, the RP2350 drive does not always accept a copy operation.

# Command summary table

## RRAM commands

| command code hex/dec | command name | function |
|---|---|---|
| 80h / 128 | UPLOAD_RAW | uploads data to RRAM byte-for-byte, without post-processing |
| 81h / 129 | UPLOAD_NICK | uploads data to RRAM with NICK bit rearrangement during upload |
| 82h / 130 | UPLOAD_TVC | uploads data to RRAM with TVC bit rearrangement during upload |
| 83h / 131 | UNZX1_RAW | ZX1-decompresses data previously uploaded to RRAM, without post-processing |
| 84h / 132 | UNZX1_NICK | ZX1-decompresses data previously uploaded to RRAM with NICK bit rearrangement |
| 85h / 133 | UNZX1_TVC | ZX1-decompresses data previously uploaded to RRAM with TVC bit rearrangement |

## Sprite command

| command code hex/dec | command name | function |
|---|---|---|
| 00h..3Fh / 0..63 | ALTER_SPB | modifies the SPB of the sprite selected by the command code |

# 2dfx general and system commands

| command code hex/dec | command name | function |
|---|---|---|
| 40h..4Fh / 64..79 | SET_TRANSPARENT_COLOR | the lower four bits of the command code select the transparent colour index – default: 0 |
| 50h / 80 | SET_ENGINE_MODE | selects the operating mode of the 2dfx renderer; this is the first step of initialisation! |
| 51h / 81 | SET_FPS | sets the 2dfx scene rate to 50 fps or 25 fps – default: 50 fps |
| 52h / 82 | SET_RENDER_OVERRUN_VISUAL | enables the visual render-overrun indicator (exclamation mark) – default: disabled |
| 53h / 83 | CLEAR_RENDER_OVERRUN | clears the latched render-overrun indication |
| 54h..57h / 84..87 | SET_RASTER_INTx | enables or disables any of the four raster interrupts – default: all four disabled |
| 58h / 88 | CLEAR_RASTER_INT | clears the latched raster-interrupt indication(s); it does not clear the interrupt configuration itself, only the indication! |
| 8Eh / 142 | SHOW_FW | displays the current firmware version and operating mode – the only way to exit is to reset the computer! |
| 8Fh / 143 | SYS_RESET | warm reset; clears RRAM, the SPBs, the 2DPT back/front tables and DEFINE_XX resource definitions; resets SET_FPS, SET_TRANSPARENT_COLOR and raster-interrupt settings to their defaults; preserves the engine mode |
| 90h / 144 | SHOW_BUILTIN_LOGO | displays the 2dfx logo at the specified X/Y coordinates, at a fixed size of 206×79 pixels, above every drawing layer |
| 91h / 145 | HIDE_BUILTIN_LOGO | removes the logo displayed by the previous command from the screen |

## 2DPT resource definition commands

| command code hex/dec | command name | function |
|---|---|---|
| 5Bh / 91 | DEFINE_FONT | assigns 1bpp font data uploaded to RRAM to a font_id |
| 5Ch / 92 | DEFINE_BITMAP | assigns a bitmap uploaded to RRAM to a bitmap_id |
| 5Dh / 93 | DEFINE_PATTERN | assigns a fill-pattern set uploaded to RRAM to a table_id |
| 5Eh / 94 | DEFINE_TILESET | assigns a set of tile images uploaded to RRAM to a tileset_id |
| 5Fh / 95 | DEFINE_TILEMAP | assigns a tilemap uploaded to RRAM to a tilemap_id |

## 2DPT back general commands

| command code hex/dec | command name | function |
|---|---|---|
| 60h / 96 | NOP | does nothing; placeholder |
| 61h / 97 | SET_COLOR | selects the ink and paper colours used by subsequent commands |
| 6Eh / 110 | SET_XY_OFFSET | adds signed X/Y values to the starting coordinates used by subsequent 2DPT commands |
| 6Fh / 111 | SKIP_NEXT | 2dfx skips the next x slots specified by the command parameter and does not execute them |
| 7Fh / 127 | RET | when this command is reached, 2dfx stops 2DPT processing regardless of whether further commands remain in the playlist |

## 2DPT back graphics primitives

| command code hex/dec | command name | function |
|---|---|---|
| 62h / 98 | PLOT | draws a point in the ink colour at the X/Y coordinates |
| 63h / 99 | LINE | draws a line in the ink colour, with the specified line thickness, between X0/Y0 and X1/Y1 |
| 64h / 100 | ERASE_LINE | draws a line in the paper colour, with the specified line thickness, between X0/Y0 and X1/Y1 |
| 65h / 101 | BUCKET_FILL | fills an area with the ink colour from the X/Y coordinate, following adjacent pixels of the colour found there until a different colour is reached |
| 66h / 102 | RECT | draws a rectangle in the ink colour and fills it with the paper colour if that is not the transparent colour, using the specified line thickness, from the X/Y starting coordinates with width W and height H |
| 67h / 103 | FILL_RECT | draws and fills a rectangle in the ink colour from the X/Y starting coordinates with width W and height H |
| 68h / 104 | ROUND_RECT | draws a rounded rectangle in the ink colour, bounded by quarter ellipses with radii RX and RY, then fills it with the paper colour if that is not the transparent colour, using the specified line thickness, from the X/Y starting coordinates with width W and height H |
| 69h / 105 | FILL_ROUND_RECT | draws and fills a rounded rectangle in the ink colour, bounded by quarter ellipses with radii RX and RY, from the X/Y starting coordinates with width W and height H |
| 6Ah / 106 | ELLIPSE | draws an ellipse centred at CX/CY with radii RX and RY in the ink colour, then fills it with the paper colour if that is not the transparent colour, using the specified line thickness |
| 6Bh / 107 | FILL_ELLIPSE | draws and fills an ellipse centred at CX/CY with radii RX and RY in the ink colour |
| 6Ch / 108 | ARC | draws an arc in the ink colour, with the specified line thickness, along an ellipse centred at CX/CY with radii RX and RY between start_angle and end_angle |
| 6Dh / 109 | PIE | draws a pie slice along an ellipse centred at CX/CY with radii RX and RY between start_angle and end_angle, connects the start and end points to the centre, and fills the interior with the ink colour |

## 2DPT back resource commands

| command code hex/dec | command name | function |
|---|---|---|
| 70h / 112 | DRAW_TEXT | draws the string stored in RRAM (up to 255 characters at a time) using the character set selected by font_id, in the ink colour from the specified X/Y starting coordinates; if the paper colour is not the transparent colour, the area behind it is filled with the paper colour |
| 71h / 113 | BLIT | draws the bitmap selected by bitmap_id from the X/Y coordinates, with or without transparency testing |
| 72h / 114 | PATTERN_FILL | fills a rectangle of width W and height H from the X/Y coordinates using pattern_number from the fill-pattern set selected by table_id |
| 73h / 115 | TILEMAP | draws a map area C_draw tiles wide and R_draw tiles high from the X/Y starting coordinates, beginning at tile X_source/Y_source in the tilemap selected by tilemap_id |

## 2DPT front general commands

| command code hex/dec | command name | function |
|---|---|---|
| E0h / 224 | NOP | does nothing; placeholder |
| E1h / 225 | SET_COLOR | selects the ink and paper colours used by subsequent commands |
| EEh / 238 | SET_XY_OFFSET | adds signed X/Y values to the starting coordinates used by subsequent 2DPT commands |
| EFh / 239 | SKIP_NEXT | 2dfx skips the next x slots specified by the command parameter and does not execute them |
| FFh / 255 | RET | when this command is reached, 2dfx stops 2DPT processing regardless of whether further commands remain in the playlist |

## 2DPT front graphics primitives

| command code hex/dec | command name | function |
|---|---|---|
| E2h / 226 | PLOT | draws a point in the ink colour at the X/Y coordinates |
| E3h / 227 | LINE | draws a line in the ink colour, with the specified line thickness, between X0/Y0 and X1/Y1 |
| E4h / 228 | ERASE_LINE | draws a line in the paper colour, with the specified line thickness, between X0/Y0 and X1/Y1 |
| E5h / 229 | BUCKET_FILL | fills an area with the ink colour from the X/Y coordinate, following adjacent pixels of the colour found there until a different colour is reached |
| E6h / 230 | RECT | draws a rectangle in the ink colour and fills it with the paper colour if that is not the transparent colour, using the specified line thickness, from the X/Y starting coordinates with width W and height H |
| E7h / 231 | FILL_RECT | draws and fills a rectangle in the ink colour from the X/Y starting coordinates with width W and height H |
| E8h / 232 | ROUND_RECT | draws a rounded rectangle in the ink colour, bounded by quarter ellipses with radii RX and RY, then fills it with the paper colour if that is not the transparent colour, using the specified line thickness, from the X/Y starting coordinates with width W and height H |
| E9h / 233 | FILL_ROUND_RECT | draws and fills a rounded rectangle in the ink colour, bounded by quarter ellipses with radii RX and RY, from the X/Y starting coordinates with width W and height H |
| EAh / 234 | ELLIPSE | draws an ellipse centred at CX/CY with radii RX and RY in the ink colour, then fills it with the paper colour if that is not the transparent colour, using the specified line thickness |
| EBh / 235 | FILL_ELLIPSE | draws and fills an ellipse centred at CX/CY with radii RX and RY in the ink colour |
| ECh / 236 | ARC | draws an arc in the ink colour, with the specified line thickness, along an ellipse centred at CX/CY with radii RX and RY between start_angle and end_angle |
| EDh / 237 | PIE | draws a pie slice along an ellipse centred at CX/CY with radii RX and RY between start_angle and end_angle, connects the start and end points to the centre, and fills the interior with the ink colour |

## 2DPT front resource commands

| command code hex/dec | command name | function |
|---|---|---|
| F0h / 240 | DRAW_TEXT | draws the string stored in RRAM (up to 255 characters at a time) using the character set selected by font_id, in the ink colour from the specified X/Y starting coordinates; if the paper colour is not the transparent colour, the area behind it is filled with the paper colour |
| F1h / 241 | BLIT | draws the bitmap selected by bitmap_id from the X/Y coordinates, with or without transparency testing |
| F2h / 242 | PATTERN_FILL | fills a rectangle of width W and height H from the X/Y coordinates using pattern_number from the fill-pattern set selected by table_id |
| F3h / 243 | TILEMAP | draws a map area C_draw tiles wide and R_draw tiles high from the X/Y starting coordinates, beginning at tile X_source/Y_source in the tilemap selected by tilemap_id |

## Easter Egg and demo commands

| command code hex/dec | command name | function |
|---|---|---|
| C0h / 192 | EGG_C0 | firmware demo: "What shall I call you?" |
| C1h / 193 | EGG_C1 | firmware demo: transformation benchmark |
| C2h / 194 | EGG_C2 | firmware demo: animated 2DPT demo |
| C3h / 195 | EGG_C3 | firmware demo: text handling and display |
| C4h / 196 | EGG_C4 | firmware demo: BLIT demo |
| C5h / 197 | EGG_C5 | firmware demo: PATTERN_FILL demo |
| C6h / 198 | EGG_C6 | firmware demo: mini TILEMAP demo |
| C7h / 199 | EGG_C7 | firmware demo: playable classic Pong |
| C8h / 200 | EGG_C8 | firmware demo: WOW Pong |
| C9h / 201 | EGG_C9 | firmware demo: Worm invasion! |
| CAh / 202 | EGG_CA | firmware demo: Boulder Dash by 2dfx |
| CBh / 203 | EGG_CB | firmware demo: render-overrun demo |
| CCh / 204 | EGG_CC | firmware demo: diagnostic tests |
| CDh / 205 | EGG_CD | firmware demo: full-screen BLIT |
| CEh / 206 | EGG_CE | firmware demo: The Prison |
| CFh / 207 | EGG_CF | firmware demo: TVC external-colour sampling test |
| D0h / 208 | EGG_D0 | firmware demo: TVC refresh stress test |
| D1h / 209 | EGG_D1 | firmware demo: coordinates of the visible screen area |
| D2h / 210 | EGG_D2 | firmware demo: animated 2DPT and SET_XY_OFFSET demo |

# Built-in firmware demos (Easter Eggs)

# Built-in firmware demos (Easter Eggs)

At first, demos – or Easter Eggs – were added to 2dfx purely for experimentation, measurement and demonstration. In reality, each of them was created to test a particular 2dfx feature or group of features, but I eventually left them in the code because simply watching them already gives a fairly good picture of what 2dfx can do and how computationally demanding those features are. They therefore also provide a useful point of reference for how hard 2dfx can be pushed.

The descriptions of the more interesting Eggs/demos include a measured render time for both Enterprise and TVC environments. The two values differ somewhat for several reasons: the Enterprise has a higher horizontal resolution, so there are more pixels that can be made visible, and a significant number of the demos were adapted to the different default screen layouts of the two machines (IS-BASIC versus TVC BASIC), which can also change the amount of computation required. The percentage shown in parentheses next to a render time is the actual share of the 20 ms time budget that is being used.

Render times are shown as minimum and maximum values when the visible content changes over time. In demo C2, for example, the displayed circles continuously grow and shrink, producing a measurable difference in processing time. The changing render times illustrate a fundamental principle of 2dfx rather well: the cost of drawing operations depends not only on how many there are, but also on how complex they are.

When idle (after initialisation or **`SYS_RESET`**), the renderer consumes about 128 us of the available 20 or 40 ms time budget on both machines. The render times listed for the demos include this overhead.

Render duration can also be observed with an oscilloscope. The 2dfx PCB has a dedicated RTIME solder pad and a neighbouring GND pad. Connecting these to a scope shows the actual render time directly: at the beginning of frame generation RTIME goes to 0 (0 V), and when generation of the frame finishes it returns to 1 (3.3 V). The time spent at zero is the image-generation time itself – the render time.

The demos run in 50 fps mode at the maximum visible horizontal resolution. Before using them on a Videoton TVC – or even while they are running – switch the machine to GRAPHICS 2 mode. This is not necessary on the Enterprise, where the maximum horizontal resolution is available in every video mode.

Enterprise IS-BASIC is not exactly drowning in colours. To make the firmware demos a little more spectacular on screen, it is worth changing the current palette as shown below before starting an Egg with the appropriate OUT command – but do not forget to initialise 2dfx first!

```
SET £102:PALETTE 0,73,75,219,146,182,109,255
OUT 248, xxx  ! xxx = Egg command code
```

The screenshots of the Eggs (with the exception of C0) were taken on a Videoton TVC.

# EGG_C0 (C0h / 192) – "What shall I call you?"

I created this demo specifically as a visual attraction for the Enterprise Club meeting held on 29 May 2026. After demonstrating the prototype board and how it worked, the final act was essentially “all right, let’s give the baby a name”. This was the moment when the Esteemed Audience learned that the board was called **2dfx**. The demo consists of five main scenes:

1. chaos – The individual letters of the ENTERPRISE logo bounce around the screen independently for roughly a minute and a half. Each letter is a separate sprite, but only the images of the distinct letters are stored in RRAM: E, N, T, R, P, I and S. Each sprite is 52 pixels high, with widths varying between 14 and 34 pixels. To display them with the correct proportions, Xdouble is enabled for every sprite, so every pixel is doubled horizontally. Six coloured stripes are also visible; these are sprites as well, displayed at 48x6 pixels, again with Xdouble enabled.

When a letter sprite reaches the right or bottom edge of the screen, it goes through an animation made from 11 horizontal and 11 vertical pre-generated movement phases. The sprite “folds up”, after which the same sequence is played in reverse, but depending on the situation the Xflip or Yflip transformation bit is also enabled in that sprite’s SPB, and the transformed sprite continues on its way across the screen. In other words, the image data for, say, an upside-down flying letter is not stored separately in RRAM; the renderer performs the transformation on the fly according to the transformation bit set in the SPB. Animation phases are changed by rewriting the RRAM start address of the sprite image data while the demo is running.

The relationship between the rightmost coloured stripe and the letter T is also worth watching: collision detection is enabled for these two sprites. If visible pixels of the two sprites overlap – in other words, if they collide – a coloured stripe flashes in the centre of the lower third of the screen and remains there for as long as the collision state persists. This stripe is another sprite. The Egg’s internal logic continuously checks whether a collision is present; when one occurs, it merely enables the SPRITE_ENABLE bit in that sprite’s SPB, and disables it again when the collision ends.

2. assembly – All letters move above the coloured stripes and assemble into the ENTERPRISE logo.

3. turning back – Any sprite that was left in an Xflip or Yflip-transformed state when the logo was assembled turns itself back in place by replaying the animation phases described in the chaos scene. For letter shapes that are symmetrical around the relevant axis (for example “I”), that particular reversal is skipped.

4. 2dfx introduces itself – About ten seconds after the third scene, the 2dfx logo drops into view with simulated gravity and bouncing, then comes to rest at a fixed Y position. The logo is 515×198 pixels and is uploaded separately; this is not the built-in **`SHOW_BUILTIN_LOGO`**.

5. rainbow – The coloured stripe sprites beneath the ENTERPRISE logo begin alternating. This is achieved simply by changing the X positions of the stripes in their respective SPBs.

Summary: a striking animation can be built with 2dfx by changing little more than SPBs. Once the animation phases have been stored in RRAM, the ten independently moving and smoothly turning letters, the six coloured stripes, the collision indication and the large falling logo are all controlled solely by modifying the relevant SPBs.


## Render times:

| scene | EP render time | TVC render time |
|---|---|---|
| chaos | 2.5 ms (12.5%) | 2.42 ms (12.1%) |
| final scene, everything in place | 2.97 ms (14.85%) | 2.93 ms (14.65%) |

# EGG_C1 (C1h / 193) – transformation benchmark

This demo puts all 64 available sprites on screen using SPBs. Its purpose was to demonstrate sprite transformations and document the differences in render time that they cause.

The two painting-like rectangles (with more than a passing resemblance to Piet Mondrian and Victor Vasarely) are built from data calculated offline and compressed with ZX1. They are enormous: 540×230 pixels each. Even so, they occupy relatively little space when compressed. Storing and uploading an uncompressed 540×230-pixel image would require 62100 bytes. By contrast, the actual compressed size of the Mondrian rectangle is 170 bytes (!), while the Vasarely one is 25851 bytes. The demo uploads these to RRAM using the internal equivalent of **`UPLOAD_RAW`** (which uses the same F8h command API, so it behaves exactly as if the data had been uploaded from an Enterprise or TVC), then decompresses them into RRAM using **`UNZX1_RAW`**.

All 62 smiley SPBs use the same 16×16-pixel sprite image, uploaded to RRAM once with **`UPLOAD_RAW`**. A 16×16 smiley occupies only 128 bytes in RRAM, so for this small graphic I deliberately did not use ZX1 compression. For interest, the actual smiley data can be compressed with ZX1 to 76 bytes in the Enterprise version and 84 bytes in the TVC version.

The demo consists of three main scenes:

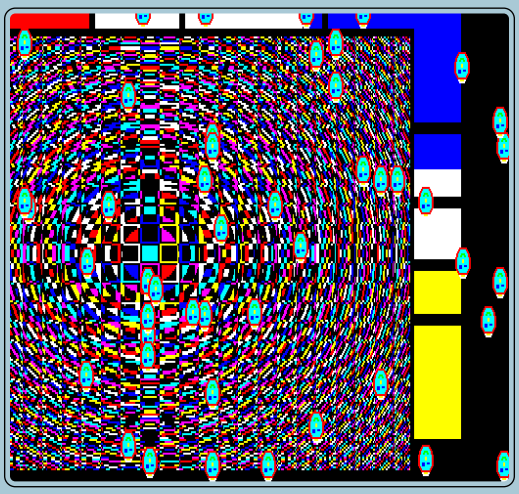
1. the sprites arrive – First the two 540×230-pixel paintings appear, followed one by one by the 62 16×16 smileys.

2. demonstrating transformations with the smiley sprites

Here the visible image is altered by using the various transformation flags in the SPBs. No other operation is performed on the sprite data stored in RRAM itself. The two large sprites are deliberately left untransformed.

3. moving sprites – The two large sprites bounce around within a defined area of the screen, while the 62 smileys move in randomly chosen directions and at random speeds; when one leaves the screen, it re-enters from the opposite side. A few sprites may appear to stand still, and that is genuinely down to chance: when their random X and Y speeds were generated, both happened to be zero. Meanwhile the transformation phases from the second scene are applied to the smileys continuously in a loop.

Summary: all 64 sprites can be visible and moving on screen, and even with different transformations applied at the same time the renderer does not necessarily reach the critical 20 ms render time.



## Render times:

| transformation | EP render time | TVC render time |
|---|---|---|
| normal – transformation bits disabled | 5.81 ms (29.05%) | 4.52 ms (22.6%) |
| X double | 6.32 ms (31.6%) | 5 ms (25%) |
| Y double | 6.12 ms (30.6%) | 4.75 ms (23.75%) |
| X+Y double | 7.28 ms (36.4%) | 5.92 ms (29.6%) |
| X flip | 5.92 ms (29.6%) | 4.64 ms (23.2%) |
| Y flip | 5.72 ms (28.6%) | 4.44 ms (22.2%) |
| X+Y flip | 8.82 ms (44.1%) | 7.54 ms (37.7%) |
| X double + X flip | 6.32 ms (31.6%) | 5.02 ms (25.1%) |
| Y double + Y flip | 8.82 ms (44.1%) | 7.54 ms (37.7%) |
| X+Y double + X+Y flip | 7.42 ms (37.1%) | 6.04 ms (30.2%) |

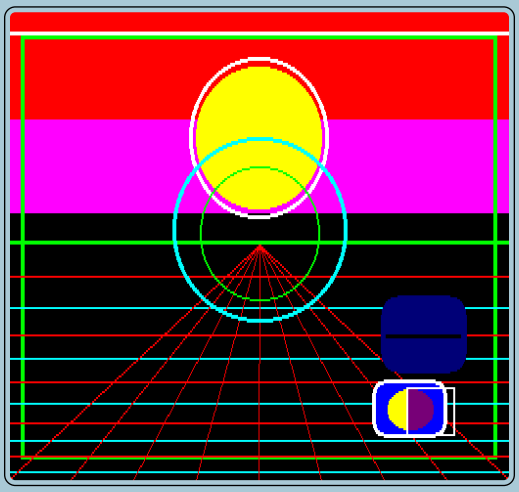
# EGG_C2 (C2h / 194) – animated 2DPT demo

This demo contains no sprites at all. It uses only the 256-slot 2DPT back playlist and, from the complete command set, only the general **SET_COLOR** and **RET** commands plus graphics primitives. Drawing the screen uses a total of 73 2DPT slots: 36 **SET_COLOR** commands, one **RET**, and the rest **FILL_RECT**, **LINE**, **RECT**, **ELLIPSE**, **ROUND_RECT**, **FILL_ROUND_RECT**, **ERASE_LINE** and **BUCKET_FILL**.

After every frame refresh, the demo rebuilds the complete 2DPT table – although in reality there is no need to do so. It would be enough to overwrite only the slots whose coordinates change (for example the moving line or the moving rectangle at the bottom) and/or whose radii change (the two circles at the top). Every 20 ms the renderer starts from an empty screen and walks through the 2DPT slots until it encounters a **RET** instruction.

The variation in render time is caused by the circles changing size while the demo is running.

Summary: even without sprites or resource-based 2DPT commands, a complex playfield can be constructed. A complete game screen for an Elite-style game, for example, could be drawn simply by changing drawing parameters.



## Render times:

| parameter | EP render time | TVC render time |
|---|---|---|
| minimum | 0.95 ms (4.75%) | 0.87 ms (4.35%) |
| maximum | 1.18 ms (5.9%) | 1.11 ms (5.55%) |

# EGG_C3 (C3h / 195) – text handling and display

This demo uses five complete fonts at the same time, each containing 128 character images, through the resource-based 2DPT commands **DEFINE_FONT** and **DRAW_TEXT**: a narrow 8×8-pixel font, a normal 16×8-pixel font, a wide 32×8-pixel font, a tall 16×16-pixel font and a large 32×16-pixel font.

Each font is stored offline in the MCU code with ZX1 compression. When the demo starts, it uploads them to RRAM using **UPLOAD_RAW**, decompresses them (**UNZX1_RAW**), then defines them for 2dfx using **DEFINE_FONT**. It also uploads the strings themselves to RRAM and displays them using **DRAW_TEXT**. While running, the demo modifies only the piece of text containing the frame counter in RRAM (by uploading a new value); everything else, including the entire 2DPT table, remains unchanged.

The demo uses 24 2DPT slots to draw the screen itself (the rectangle, the line inside it and **SET_COLOR**), together with **DRAW_TEXT** commands referring to the different font definitions and strings in RRAM.

Summary: with 2dfx support, drawing text – or displaying labels and HUD elements in a game – requires very little rendering resource on the 2dfx side, while displaying strings from the Enterprise or TVC side is also a very simple process.


## Render times:

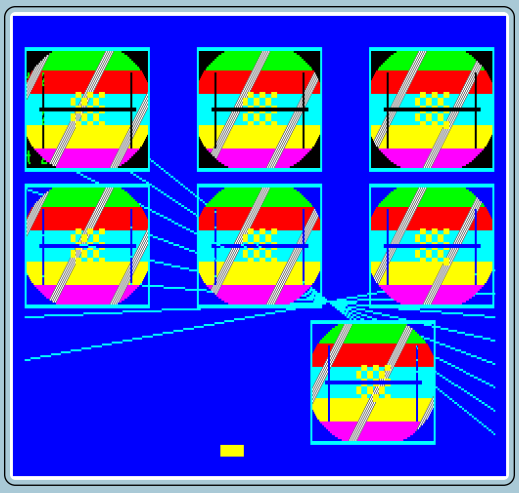
| EP render time | TVC render time |
|---|---|
| 1.32 ms (6.6%) | 1.31 ms (6.55%) |

# EGG_C4 (C4h / 196) – BLIT demo

This demo shows the operations that can be performed with **BLIT**. The 128×64-pixel bitmap object used by the demo has 64 animation phases generated offline in advance. They are stored in the MCU code as a single ZX1-compressed animation bank; the bank is transferred to RRAM using the internal equivalents of **UPLOAD_RAW** and **UNZX1_RAW**, where it occupies 256 kB after decompression. And, just to make the amount of loaded data even larger, the same 64 phases are stored in three additional banks, each shifted by 16 phases. This has no practical benefit whatsoever; it merely demonstrates how easily a large amount of data can be loaded into 2dfx. Together, the four banks occupy 1 MB of RRAM.

Drawing is performed through the 2DPT in two different ways. In the first row, the **BLIT** objects are placed on screen without transparency testing. Because they use different source banks, the moving stripes can clearly be seen at different phases relative to one another. In the second row, the same objects are drawn with transparency testing, while the horizontally moving copy in the bottom row is also displayed using transparent **BLIT**.

While the demo is running, the **BLIT** commands in the 2DPT do not change, except for the X coordinate of the moving copy in the bottom row. Animation is achieved mainly by changing the bitmap's RRAM start address in the **DEFINE_BITMAP** descriptor once per frame, so that it always points to the current 4 KiB animation phase. No new bitmap data is uploaded or decompressed while the visible animation is running.



## Render times:

| EP render time | TVC render time |
|---|---|
| 1.77 ms (8.85%) | 1.63 ms (8.15%) |

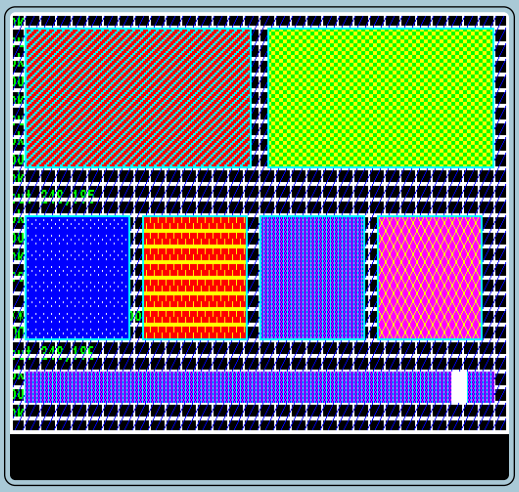
# EGG_C5 (C5h / 197) – PATTERN_FILL demo

This demo shows the 2DPT **PATTERN_FILL** command in action. It creates and uses eight 16×8-pixel fill patterns. The large background area, the two wide rectangles, the three smaller rectangles and the wide strip at the very bottom all use these eight patterns. A smaller 16×16-pixel **FILL_RECT** also moves continuously along the bottom of the screen.

Internally, the eight fixed-size patterns are stored consecutively in RRAM as a single 512-byte pattern bank, made available to the 2DPT through one **DEFINE_PATTERN** descriptor. The various independent **PATTERN_FILL** commands select whichever pattern they currently need from this bank.

The large background pattern changes more slowly, while the pattern in the wide bottom strip changes more quickly, and the solid 16x16-pixel rectangle travels along that lower strip. The demo rebuilds the background 2DPT list on every refresh.

Summary: pattern filling is very useful for drawing things such as floors, backgrounds for status displays, or other repeating surfaces, while both the graphics resource size and the processing time required remain minimal.



## Render times:

| EP render time | TVC render time |
|---|---|
| 2.16 ms (10.8%) | 1.73 ms (8.65%) |

# EGG_C6 (C6h / 198) – mini TILEMAP demo

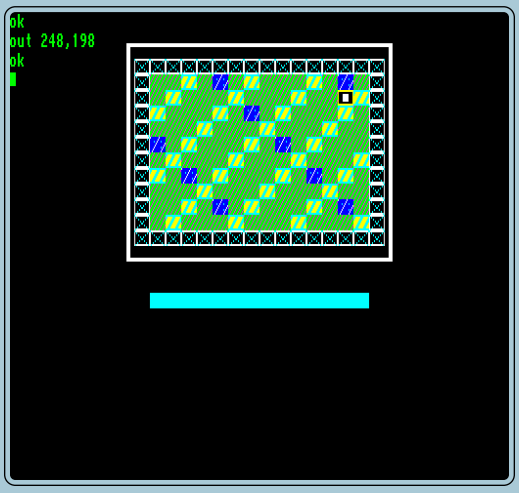
This demo shows the 2DPT **TILEMAP** command in operation. The displayed playfield consists of 16×12 tiles, each 16×8 pixels. A frame surrounds the tilemap, with a coloured strip underneath; these are drawn using simple 2DPT graphics primitives.

The demo reserves space for a total of eight different 16×8-pixel tiles in the tileset. Each tile occupies 64 bytes, so the complete eight-tile tileset uses 512 bytes of RRAM.

In the 16×12-cell tilemap, each cell needs only one byte specifying which numbered tile appears at that position. The complete playfield description therefore occupies 192 bytes in RRAM.

The tilemap command structure is used here through the same three commands described earlier: the tileset itself is defined with **DEFINE_TILESET**, while the map is defined with **DEFINE_TILEMAP**. The complete 16×12 TILEMAP is then placed on screen with a single 2DPT slot command.

While the demo is running, those three commands are set up only once. During animation, the 192-byte playfield description is regenerated (by changing the tile number at the required positions) and uploaded again to RRAM. This could obviously be made more efficient by overwriting only the few bytes that actually change, but for simplicity all 192 bytes are uploaded again. During the next render, the same single 2DPT **TILEMAP** command automatically draws the new contents.




## Render times:

| EP render time | TVC render time |
|---|---|
| 0.47 ms (2.35%) | 0.4 ms (2%) |

# EGG_C7 (C7h / 199) – playable classic Pong

This demo is a simple playable Pong game showing how a few sprites and the 2DPT can be used together to build an almost complete game. The two paddles and the ball are three separate sprites; the playfield, centre line and scores are produced with 2DPT commands.

The three sprites are simple rectangles: the paddles are 12×48 pixels and the ball is 12×6 pixels. Their data is stored offline and uploaded to RRAM when the demo starts using the internal equivalent of **UPLOAD_RAW** (612 bytes in total).

The sprite data does not change while the game is running; only the drawing position (X and Y coordinates) is changed in the SPBs.

The playfield is made from a few simple 2DPT commands using **RECT** and **FILL_RECT**. The score digits themselves are also built from **FILL_RECT** rectangles placed appropriately for the current value.

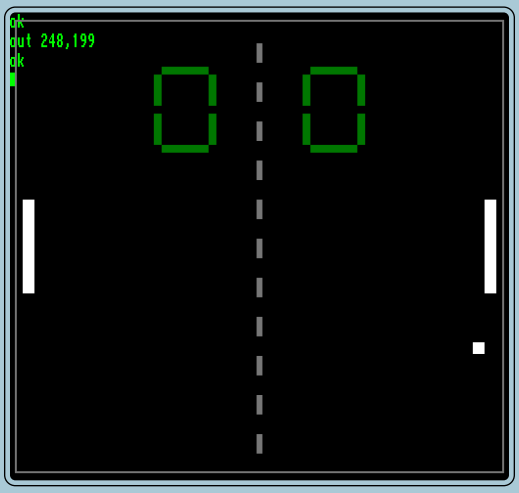
On every frame refresh the demo calculates the new positions of the ball and paddles and updates the three SPBs, while also rebuilding the entire background 2DPT playlist. This could be simplified by rewriting only the slots corresponding to score values that actually change. The demo does not use the collision detection provided by 2dfx; the game logic calculates collisions from the rectangles itself and changes the X/Y movement of the ball accordingly.

When one player scores (the ball leaves the other side of the playfield), the playfield is shaken. This is done by changing **SET_XY_OFFSET**, which is much simpler than recalculating and uploading the X and Y values of every affected 2DPT slot. Because **SET_XY_OFFSET** applies only to the 2DPT playlist, the sprites visibly do not shake together with the playfield.

A short BASIC program makes Pong playable on either an Enterprise or a TVC. While the demo is running, changing particular bits of port F9h tells it to start a new game or move a given paddle in a particular direction. The movement signal must be cleared when the key is released! Here is the BASIC program for the Enterprise that provides the controls:

```
100 PROGRAM "pong.bas"
110 SET KEY RATE 1
120 SET KEY DELAY 1
130 SET KEY CLICK OFF
140 SET £102:PALETTE 0,73,75,219,146,182,109,255
150 OUT 248,80
160 OUT 249,1
170 WAIT DELAY 2
180 OUT 248,199
190 LET A$=INKEY$
200 IF A$="q" THEN OUT 249,8
210 IF A$="a" THEN OUT 249,4
220 IF A$="p" THEN OUT 249,2
230 IF A$="l" THEN OUT 249,1
240 IF A$="n" THEN OUT 249,128
250 IF A$="" THEN OUT 249,0
260 GOTO 190
```

Summary: this demo is a simple example of how little data actually has to change to produce a game screen. Once the three sprite images have been placed in RRAM, gameplay changes little more than a few coordinates and the score; 2dfx recreates the playfield each frame using only a handful of 2DPT commands.




## Render times:

| EP render time | TVC render time |
|---|---|
| 0.32 ms (1.58%) | 0.31 ms (1.56%) |

# EGG_C8 (C8h / 200) – WOW Pong

This demo is a visually turbocharged version of the classic Pong from C7. The gameplay is identical to classic Pong, but it is deliberately stuffed with all sorts of additional graphical elements that are easy to produce with the 2DPT. I call it a visual experiment; in practice it is almost spectacularly unplayable – although we could equally call it advanced-level Pong…

Alongside the three sprites, this version also uses a font, tilemap, patterns, text and various 2D graphics primitives. It can be played with the same BASIC program described for C7, for anyone who happens to enjoy a challenge…

One part of the background is an animated tilemap. On the Enterprise it is 39×22 cells, and on the TVC 30×24 cells; each cell is still a single byte containing the number of the tile to display at that position. The associated tileset uses eight tiles. The contents of the tilemap are regenerated on every frame refresh. Along with the frame and the horizontal and vertical "decorations", the positions of certain tiles also change, so the background appears to be in constant motion. The tileset itself is of course not uploaded again; only the map, a few hundred bytes in size, changes. The **DEFINE_TILESET** and **DEFINE_TILEMAP** descriptors continue to point to the same RRAM areas throughout.

The demo also uses four **PATTERN_FILL** patterns. These form the animated strips at the top and bottom, as well as the panels around the scores and in the centre. The selected patterns change over time, adding still more movement to the background without requiring new graphics data to be uploaded while the game is running.

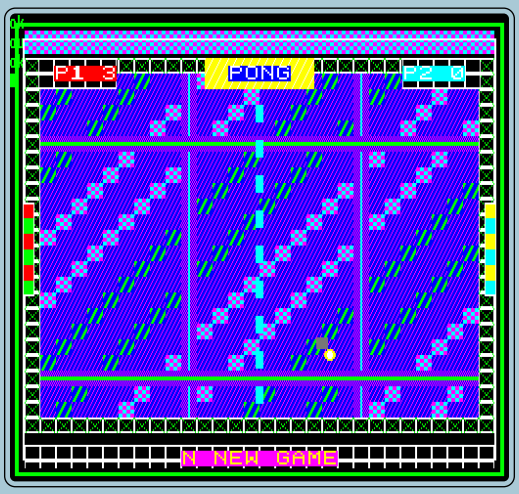
A complete 16x8-pixel font is stored in RRAM for the labels. Each character occupies eight bytes, so the complete font uses 1024 bytes. The strings "PONG", "P1 0", "P2 0" and "N NEW GAME" live in separate RRAM areas. When a point is scored, there is no need to create a new **DRAW_TEXT** command in the 2DPT; only the digit character inside the P1 or P2 string is rewritten in RRAM, while the existing **DRAW_TEXT** continues to refer to the same string.

Alongside **TILEMAP** and **PATTERN_FILL**, the 2DPT is built from **RECT**, **FILL_RECT** and **DRAW_TEXT** instructions. These draw the playfield borders, centre line, labels and other details, as well as the simple "trail" effect visible behind the ball.

After a point is scored, this demo shakes the picture too, but in a somewhat more spectacular fashion than C7. The background drawn by the 2DPT and the sprites receive two separate shake animations. The background coordinates receive a common offset through **SET_XY_OFFSET**, while the SPB coordinates of the paddles and ball change independently by slightly different amounts. The result is that the whole scene shakes together without moving like one rigid picture.

On every 50 Hz refresh, the game calculates the new game state, creates and uploads the current tilemap, updates the score string if necessary, rebuilds the background 2DPT, and finally sets the current SPBs of the three sprites. The larger graphics resources – the sprites, font, tileset and pattern bank – remain unchanged in RRAM throughout.

Summary: compared with C7, this demo makes it very clear that the same extremely simple game logic can be surrounded by a much more complex screen without requiring the computer itself to draw anything pixel by pixel.




## Render times:

| EP render time | TVC render time |
|---|---|
| 1.56 ms (7.8%) | 1.42 ms (7.1%) |

# EGG_C9 (C9h / 201) – Worm invasion!

This demo uses sprites exclusively – but wastes very little time before using almost all of them. First the heads of the worms appear one by one along the left, right and bottom edges of the screen; after a short pause, all 63 invade the display. Finally the chief worm arrives from the right. It uses the 64th sprite slot, so at that point all 64 available sprites are active.

A normal worm is 64×16 pixels, so one animation phase occupies 512 bytes, and there are eight different phases. One complete animation in a particular colour therefore uses 4 kB of RRAM. The demo prepares the same eight phases for all 15 usable colours – colour index 0 remains transparent – giving a total of 60 kB of animation data for the normal worms. This does not mean that the 63 sprites each have their own graphics data! All 63 worms share the same 15 sets of coloured animation data in RRAM. For each individual worm, only the RRAM start address in the SPB changes according to its current colour and animation phase. Every worm receives one of the 15 colours at random, and their animation speeds may differ as well. There is no separately stored mirrored artwork for worms moving from right to left either. They use the same image data, with the Xflip transformation enabled in the SPB.

The chief worm appearing as sprite 64 uses the same animation, but its phases are stored in RRAM already enlarged to four times the size. One phase is therefore 256×64 pixels, or 8192 bytes, and all eight phases together occupy 64 kB. In addition, Xdouble and Ydouble are enabled in its SPB during display, so the sprite actually visible on screen becomes 512×128 pixels. Since the chief worm travels from right to left, Xflip is enabled in its SPB as well.

Summary: the demo demonstrates resource optimisation with sprites. Sixty-three independently moving objects with different colours, animation speeds and directions do not necessarily require 63 separate animation sets. The image data is shared; while the demo runs, essentially only the address, position and transformation fields in the SPBs change.


## Render times:

| scene | EP render time | TVC render time |
|---|---|---|
| sixty-three worms running around | 2.11 ms (10.55%) | 2.07 ms (10.35%) |
| the whole team together | 4.21 ms (21.05%) | 3.97 ms (19.85%) |

# EGG_CA (CAh / 202) – Boulder Dash by 2dfx

This demo is an extended version of the simple C6 **TILEMAP** demonstration. Here we move around a level much larger than the visible screen with smooth, pixel-by-pixel scrolling. The player character remains in the centre while the complete world moves behind it.

The complete level consists of 160×100 tiles, for a total of 16,000 tilemap cells. Since each cell is one byte, the complete level description itself occupies 16,000 bytes of RRAM. The tileset contains 29 different 16×8-pixel tiles and occupies 1856 bytes in total.

The graphical objects are actually built from 2×2 tiles, so one level element is effectively 32×16 pixels. This is how the ground, wall, brick, boulder, diamond and exit are constructed.

The complete level (**DEFINE_TILEMAP**) is created and uploaded to RRAM once when the demo starts. During scrolling, the tilemap data itself never changes and is never uploaded again. The visible window contains 38×25 tiles at once on the Enterprise and 30×26 on the TVC.

The key to smooth scrolling is that the camera position (viewport) is tracked not only in tiles but also in pixels. Two pieces of information are calculated from the current X and Y position: which tile should be the first source cell for the **TILEMAP** command, and how many pixels of offset we currently have within that tile.

Until the next 16-pixel tile boundary is reached, the starting tile remains the same, while the X drawing coordinate of the complete **TILEMAP** is shifted a few pixels to the left. When the tile boundary is crossed, the source tile X coordinate increases by one and the within-tile pixel offset starts again. The same thing happens vertically, except that tiles are 8 pixels high. There is one small trick: if the tilemap is shifted 15 pixels to the left, for example, part of the next tile is already visible at the right edge. The demo therefore always draws one extra column and one extra row. Thus, for a 38×25 visible area on the Enterprise it actually draws 39×26 tiles, while for the TVC's 30×26 area it draws 31×27.

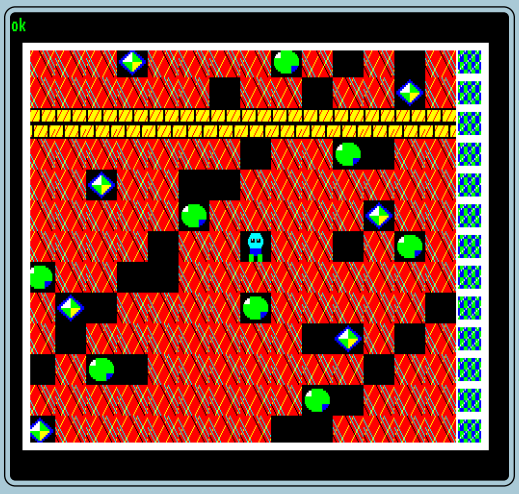
The unnecessarily drawn edges are masked by four 2DPT **FILL_RECT** slots, and a **RECT** slot then draws a frame around the visible window. From the outside we therefore see only a fixed-size viewport, while behind it the **TILEMAP** draws a slightly larger area and moves within it.

The player is a separate 32×16-pixel sprite occupying 256 bytes. It remains at the same screen position throughout; the impression of movement is actually created by the tilemap scrolling behind it.

On every 50 Hz refresh, the demo rebuilds the background 2DPT list with the current **TILEMAP** source and destination coordinates. The **DEFINE_TILESET** and **DEFINE_TILEMAP** descriptors, however, are set only once when the demo starts because the underlying RRAM data does not change. The important point is that the 16 kB level data does not have to be touched at all while scrolling.

The camera route deliberately approaches the right and bottom edges of the level as well, making it visible that if **TILEMAP** is configured so that the drawn region would extend beyond the original map, the empty area does not show the opposite side of the level or random garbage; the MCU handles this "overhang" correctly.

Summary: the most important lesson of this demo is that smooth scrolling of a large tilemap does not require the level data itself to be moved or reloaded. It is enough to change the source cell and screen position of the **`TILEMAP`** command and draw one extra row and column around the edges. A level containing thousands of cells can remain unchanged in RRAM throughout, without the Enterprise or TVC having to upload it again every time.




## Render times:

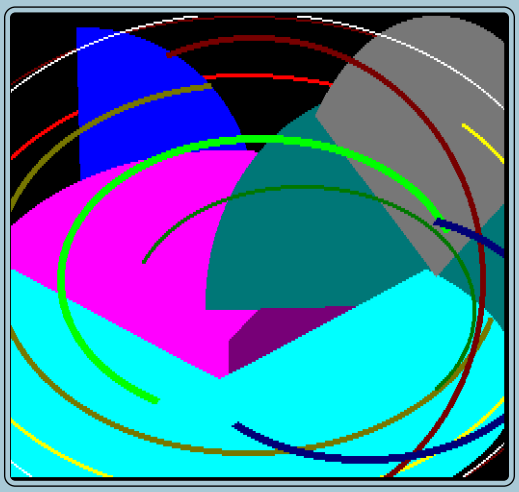
| EP render time | TVC render time |
|---|---|
| 1.64 ms (8.2%) | 1.46 ms (7.3%) |

# EGG_CB (CBh / 203) – render overrun demo

This demo deliberately builds a 2DPT whose rendering time comfortably exceeds the available 20 ms budget in 50 fps mode. Along with two large **ELLIPSE** primitives, it draws six filled **PIE** shapes and eight **ARC** primitives of different line thicknesses on top of one another – the latter two are the most computationally expensive primitives.

The main purpose is not to demonstrate **ARC** and **PIE** themselves, but to show what happens when the renderer does not finish a frame in time.

When a render overrun occurs, bit 5 of the status register (**AND 20h**) changes to zero and remains in that state until cleared with **CLEAR_RENDER_OVERRUN**. If the next display refresh arrives while the previous rendering operation is still in progress, the previously completed framebuffer remains on screen.

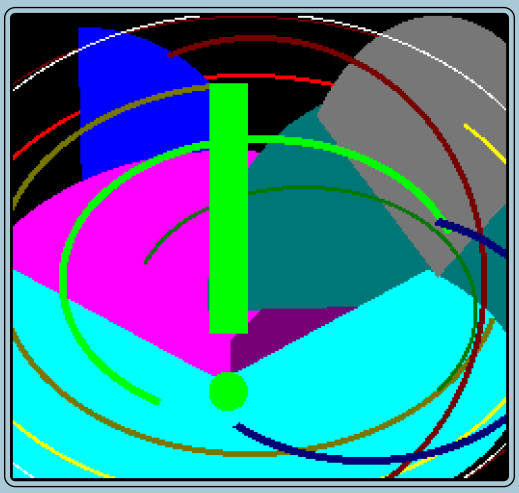


The CB demo also makes it easy to try the **SET_RENDER_OVERRUN_VISUAL** command in practice.

After starting the demo, enable the visual indication:

```
OUT 248, 82
OUT 249, 1
```

A flashing, colour-changing exclamation mark appears in the centre of the screen above the normal graphics layers.



The sticky overrun state can be cleared with **CLEAR_RENDER_OVERRUN**, but because the demo is still running and continuously drawing the same image, the exclamation mark reappears in the next frame.

## Render times:

| EP render time | TVC render time |
|---|---|
| 43.1 ms (215.5%) | 43.1 ms (215.5%) |

# EGG_CC (CCh / 204) – diagnostic tests

This demo consists of five diagnostic tests, each running for roughly 20 seconds. These are not visual showpieces; they are intended to verify correct operation of the renderer, RRAM addressing, sprite transformations, clipping and the different resource types. Render time is not listed separately here because it is not relevant.

23-bit RRAM addressing and cache check – The first scene displays sprites placed at deliberately widely separated RRAM addresses, including the **000000h**, **010000h**, **100000h**, **200000h**, **400000h** and **7FF000h** regions. Several sprites also use addresses with identical low bits but different upper address portions. This verifies that the renderer and internal sprite cache really use the complete 23-bit address and do not confuse resources separated by as much as several megabytes.

Sprite transformations – In this scene, the same directional marker sprite is displayed in ten copies using the X/Y double, X/Y flip and combined transformations also used in C1. Because the graphic is asymmetric, any incorrect mirroring or enlargement is immediately visible.

Clipping – The third section places a 96×64-pixel sprite partly outside the left, right, top and bottom edges of the renderable area, and also uses an Xdouble version. The purpose is to verify that sprite clipping works correctly for both normal and transformed objects, and that the renderer does not read outside the RRAM region belonging to the object.

Resource handling – Here **PATTERN_FILL**, **TILEMAP**, two **BLIT** operations and two sprites all run at the same time; one of the sprites uses the X+Y double transformation. This allows the interaction of different resource descriptors and rendering modes to be checked within a single frame.

ZX1 decompression – The final test uploads ZX1-compressed sprite data into one region of RRAM, clears the destination region first, then starts **UNZX1_RAW**. The sprite SPB already points to the decompression destination address, so the image appears when the decompression operation running on MCU core1 has completed. This simultaneously checks ZX1 decompression and verifies that data produced by core1 becomes correctly visible to the renderer.

# EGG_CD (CDh / 205) – full-screen BLIT

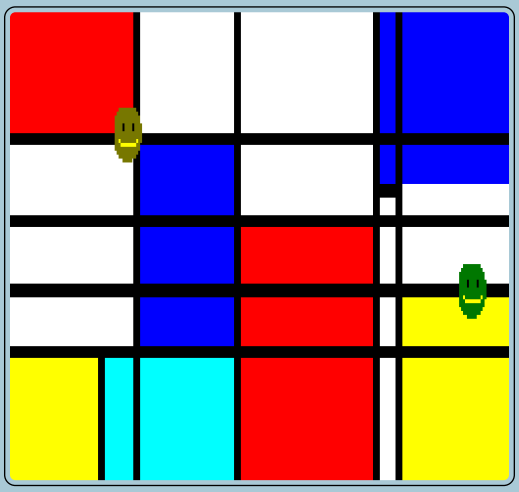
This demo was created to measure the rendering speed of **BLIT**. A single bitmap fills almost the entire 2dfx render area: 744×288 pixels on the Enterprise and 704×256 pixels on the TVC. That corresponds to 107136 bytes of image data on the Enterprise and 90112 bytes on the TVC.

The Mondrian-like bitmap is generated once when the demo starts and uploaded to RRAM. The image does not use colour index 0, the transparent index. After one **DEFINE_BITMAP** command, the demo issues a single 2DPT **BLIT** command. Neither the bitmap itself nor the **BLIT** command changes while the demo is running.

The demo consists of two scenes of twenty seconds each, alternating continuously:

Full-screen **BLIT** – In this part, only the large bitmap **BLIT** loads the renderer, allowing the time required to copy a bitmap approaching full-framebuffer size to be measured directly.

Full-screen **BLIT** + two sprites – Here the full-screen **BLIT** remains unchanged, but two 16×16-pixel smiley sprites are added. Xdouble and Ydouble are enabled on both, so they appear at 32×32 pixels. This makes it easy to measure how much two sprites add to the render time on top of the already large **BLIT**.



## Render times:

| scene | EP render time | TVC render time |
|---|---|---|
| full-screen BLIT | 2.22 ms (11.1%) | 1.86 ms (9.3%) |
| full-screen BLIT + two sprites | 2.29 ms (11.45%) | 1.94 ms (9.7%) |

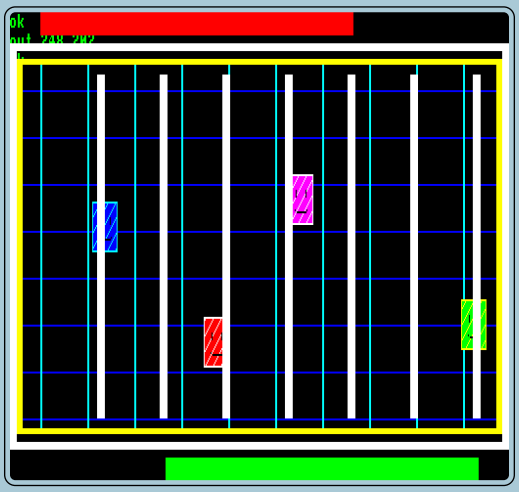
# EGG_CE (CEh / 206) – The Prison

This demo demonstrates the complete 2dfx layer order: 2DPT back, sprites, 2DPT front.

Four 32×32-pixel prison inmates – cheerful ones, admittedly – bounce around the screen. Behind them, 2DPT back draws the grid and frame; in front of them, the vertical bars, outer frame and two HUD strips are drawn in 2DPT front.

The image data for the four sprites is uploaded to RRAM once when the demo starts. While it runs, only the X/Y coordinates in the SPBs change as the sprites bounce around the screen. 2DPT back is also built only once because the grid visible behind the sprites is static. 2DPT front, on the other hand, is rebuilt on every frame refresh because the vertical bars are constantly moving. When a sprite passes behind one of these bars, 2DPT front simply draws over it, while the background grid lines (2DPT back) always remain behind the sprites.

Summary: the demo shows that graphics drawn in front of sprites do not require additional sprites or modified sprite images. A game can use the same technique for a HUD, window frame, gate, bars, foreground scenery or any other masking graphic.



## Render times:

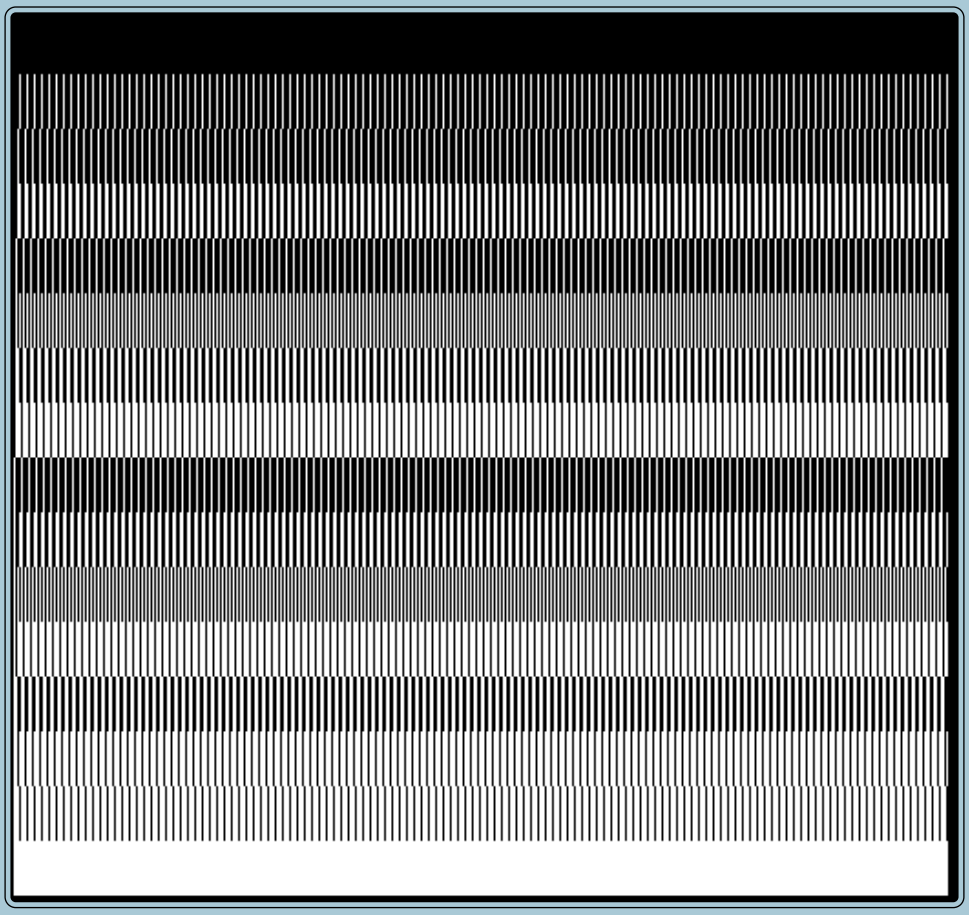
| EP render time | TVC render time |
|---|---|
| 1.04 ms (5.2%) | 1 ms (5%) |

# EGG_CF (CFh / 207) – TVC external-colour sampling test

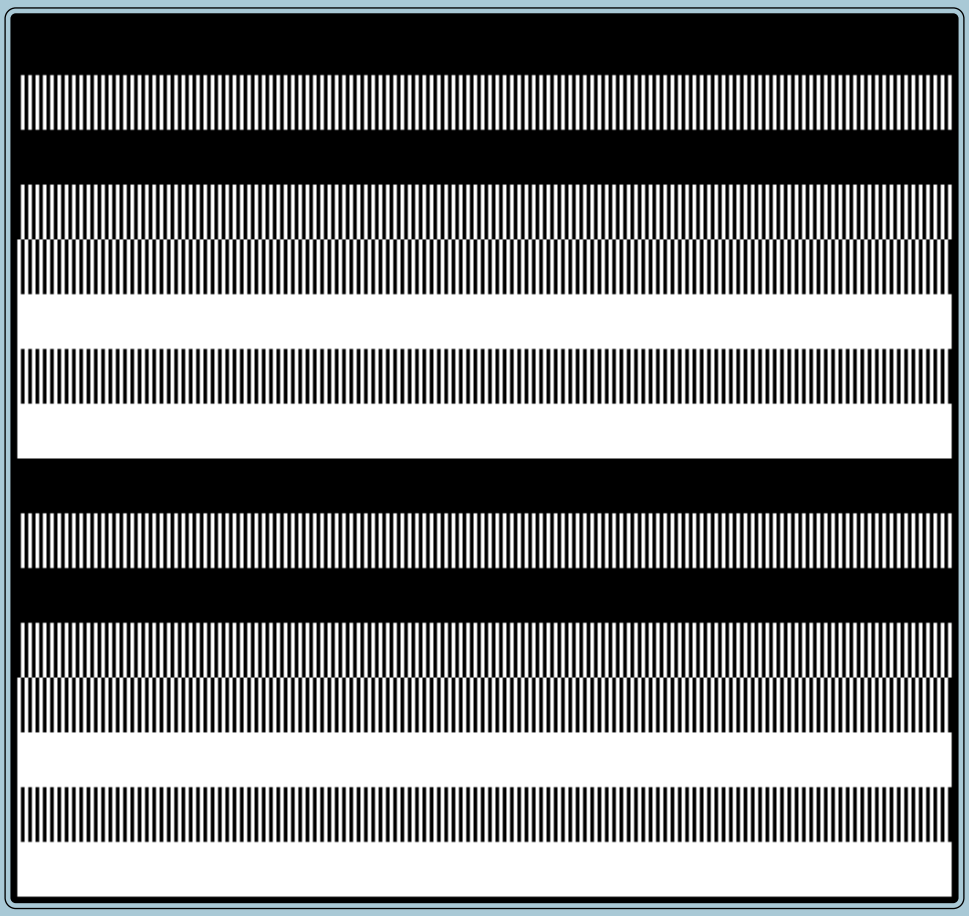
This demo can be used only on the TVC and demonstrates an important limitation in the way the TVC processes its external colour inputs. Render time is not important here.

2dfx changes the EC0..EC3 external colour signals toward the TVC at a 12.5 MHz rate, thereby reaching the maximum possible horizontal resolution. The TVC, however, samples this full rate only in GRAPHICS 2 mode. In GRAPHICS 4 and GRAPHICS 16 modes, the video system samples the EC0..EC3 inputs at only half that rate – in other words, it considers only every second 2dfx pixel.

The demo displays 16 horizontal bands showing different four-pixel bit patterns from 0 to 15, with the pattern itself changing from band to band. In the first band every pixel has value 0; in the second, groups of three 0-valued pixels followed by one 15-valued pixel repeat, and so on. In GRAPHICS 2 mode all 16 bands are displayed correctly and are clearly distinguishable from one another.



In the other video modes, however, the effect of half-rate sampling is clearly visible. A rapidly alternating pattern such as 1010 is sent correctly by 2dfx, but the TVC samples only every second pixel. As a result, the original pattern of some bands disappears completely; for example, an alternating 0101 pattern may appear as a solid white 1111 block.



The 604×240-pixel test image is created in RRAM as a bitmap and placed on screen with a single 2DPT **`BLIT`** command.

The demo makes it very clear why GRAPHICS 2 mode must be selected on the TVC when using 2dfx if the maximum horizontal resolution is required.

# EGG_D0 (D0h / 208) – TVC refresh stress test

This is a simple stress test available only on the TVC. It deliberately changes a large portion of the screen on every single 50 Hz refresh; render time is irrelevant here.

It alternates between two tests: in one, the complete 704×256-pixel render area flashes between two different colours; in the other, a vertical boundary between black and white areas moves rapidly from side to side.

My aim was simply to make consecutive frames differ as dramatically as possible, so that any frame mixing or other display or synchronisation problem becomes immediately obvious.

# EGG_D1 (D1h / 209) – coordinates of the visible screen area

This demo was created to compare the 2dfx X/Y coordinate system with the screen area that is actually visible.

First, a one-pixel-wide vertical line travels across the complete X range and then returns. A horizontal line then sweeps through all Y coordinates in both directions. Neighbouring lines beside the bright measuring line make the position easier to follow on a television or other display.

A four-digit display in the centre of the screen continuously shows the exact coordinate currently being tested in the 2dfx coordinate system.

Each coordinate remains on screen for four frames, with a short pause at the endpoints. This makes it possible to observe where a particular X or Y value appears and when it disappears from the screen.

When planning a user program, this demo can be used to reveal the individually programmed visible screen area on an Enterprise or TVC. We can then give 2dfx graphics coordinates that really do lie inside – or deliberately outside – the visible image area. I used this demo to produce the diagrams of exact visible coordinates found in the **Geometry** chapter of this handbook.

# EGG_D2 (D2h / 210) – animated 2DPT and SET_XY_OFFSET demo

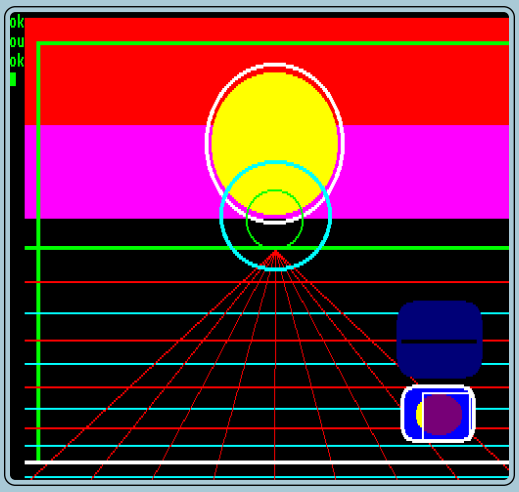
This demo uses the C2 2DPT-primitives demonstration unchanged, except that the complete graphics continuously moves around the screen.

The first element of the 2DPT is a **SET_XY_OFFSET** command. Every position-dependent 2DPT slot that follows it – lines, rectangles, ellipses, fills and so on – automatically receives the same X/Y offset. The coordinates of the individual graphics commands themselves are therefore not modified.

The X and Y offsets change independently: horizontally over a range of ±24 pixels and vertically over ±10 lines. The internal animations visible in C2 – for example the moving horizontal line, changing-size circles and moving rectangle at the bottom – continue to work exactly as before. **SET_XY_OFFSET** simply adds one further common offset on top of them.

The render times of C2 and D2 can be compared directly because the graphical content of the two demos is identical; D2 adds only the operation of **SET_XY_OFFSET**. The table clearly shows that **SET_XY_OFFSET** adds practically nothing to the render time – its overhead is almost zero.

Summary: **SET_XY_OFFSET** is useful for moving an entire screen region or HUD, implementing camera shake and transitions, or any situation in which several 2DPT elements should move together within one common coordinate system.



## Render times in the D2 demo:

| parameter | EP render time | TVC render time |
|---|---|---|
| minimum | 0.96 ms (4.8%) | 0.87 ms (4.35%) |
| maximum | 1.2 ms (6%) | 1.11 ms (5.55%) |

## Render times in the C2 demo:

| parameter | EP render time | TVC render time |
|---|---|---|
| minimum | 0.95 ms (4.75%) | 0.87 ms (4.35%) |
| maximum | 1.18 ms (5.9%) | 1.11 ms (5.55%) |


*entire hardware design: Károly Vaczkó*
*software architecture, command interface design: Károly Vaczkó*
*MCU code was programmed by ChatGPT 5.5 Plus then ChatGPT 5.6 Sol, and*
*it went through several further sanity checks by Anthropic's Fable 5*
*handbook was written by: Károly Vaczkó*

*August 2026*