



Designing and Implementing RemoteFX Display Drivers in Windows Embedded Compact 7

Writers: Rohit Talwar, Ryan Wike

Technical Reviewer: Windows Embedded Compact RDP Team

Applies To: Windows Embedded Compact 7

Published: December 2011

Abstract

When you implement Microsoft RemoteFX on a Windows Embedded Compact 7 enabled thin-client device, you have a choice between decompressing the display data by using either software decoding or hardware-accelerated decoding. If you decide to implement hardware-accelerated decoding, you must develop a RemoteFX display driver. This paper outlines the process that an OEM driver developer uses to implement a RemoteFX display driver. It includes:

- How to choose between window-frame or full-screen-only display modes.
- Implementation details for both modes, including registry settings and code examples.
- A listing of the graphics device interface (GDI) escape sequences that must be supported.
- Display driver design guidelines.
- A code example that shows how to extract information from the RemoteFX display data.

For an introduction to the RemoteFX architecture and implementation recommendations, we recommend [Getting Started with Remote FX in Windows Embedded Compact 7](http://go.microsoft.com/fwlink/?LinkID=215085) (<http://go.microsoft.com/fwlink/?LinkID=215085>).

Contents

| | |
|-----------------------------------------------------------------------------|----|
| Introduction to Developing a RemoteFX-Capable Display Driver..... | 4 |
| RemoteFX Display Driver Overview..... | 5 |
| Using a Coprocessor for Hardware Acceleration..... | 6 |
| Display Driver Design Guidelines..... | 7 |
| Display Driver Design Choice: Window-Frame or Full-Screen Display Mode..... | 7 |
| Performance Optimization for the RemoteFX Thin-Client Display Driver | 9 |
| Implementing a RemoteFX Display Driver | 10 |
| RemoteFX Window-Frame and Full-Screen Display Modes..... | 12 |
| RemoteFX Window-Frame Display Mode..... | 12 |
| RemoteFX Full-Screen Display Mode | 13 |
| DrvEscape Function..... | 14 |
| ESCAPE_GET_CAPABILITIES..... | 17 |
| ESCAPE_DEC3..... | 20 |
| ESCAPE_COPY_2BMP..... | 28 |
| ESCAPE_EXIT_CA..... | 32 |
| Extracting Rectangle Information from RemoteFX Display Data..... | 33 |
| Conclusion | 38 |
| Additional Resources | 39 |

Introduction to Developing a RemoteFX-Capable Display Driver

OEMs, especially driver developers in OEMs, may require guidelines and code examples for designing and implementing a RemoteFX display driver on a Windows Embedded Compact 7 (Compact 7) thin client. When you implement the RemoteFX display driver, you perform an essential step in deploying RemoteFX on thin-client devices. RemoteFX technology uses the hardware on servers that run Windows Server 2008 R2 with Service Pack 1 (SP1) to efficiently encode display data before the data is sent to the Compact 7 thin client. The display data is decoded on the client by the display driver before being displayed.

When you implement RemoteFX on thin clients, an important part of your deployment design is the type of decoding you choose to use. RemoteFX on Compact 7 supports client-based hardware-accelerated decoding and also supports software decoding. Hardware-accelerated decoding offers the best performance, especially on low-end hardware. We recommend that you use hardware-accelerated decoding on your thin clients. If you decide to build a thin client that uses hardware-accelerated decoding to decompress RemoteFX compressed data, you also need to develop a RemoteFX display driver.

RemoteFX hardware-accelerated decoding offloads data decompression to a coprocessor, which reduces the workload on the CPU of the client device. By using hardware-accelerated decoding, you can develop a lower-cost Compact 7 thin client that uses less powerful hardware compared to the typical thin clients that are used with a remote server. A RemoteFX-enabled thin-client device that runs on Compact 7 also provides users with a full remote experience, such as high-quality full-screen video and the Windows Aero theme.

RemoteFX with Compact 7 requires that you run Windows Server 2008 R2 with SP1 on the server and Compact 7 on the thin-client device. For specific hardware and software requirements for the client and server and for an introduction to using RemoteFX with Compact 7, you should first read [Getting Started with Remote FX in Windows Embedded Compact 7](http://go.microsoft.com/fwlink/?LinkID=215085) (<http://go.microsoft.com/fwlink/?LinkID=215085>).

After you review the hardware and software requirements and the introductory article, continue by reviewing this article, which describes how to design and implement a RemoteFX display driver on a Compact 7 thin client. The following list provides a brief roadmap to the information in this article:

- An overview of implementing a Compact 7 RemoteFX display driver in [RemoteFX Display Driver Overview](#).
- Information about how to use hardware-accelerated decoding in display drivers in [Using a Coprocessor for Hardware Acceleration](#).
- An introduction to the two display-mode choices and guidance about performance optimization in [Display Driver Design Guidelines](#).
- Implementation details, including the applicable escape sequences, display-mode flow charts, and several code examples in [Implementing a RemoteFX Display Driver](#).

RemoteFX Display Driver Overview

RemoteFX-enabled display drivers, which OEMs create and load on their client devices, connect the Compact 7 operating system to the display hardware. After you load your drivers, they are directly called by the [Graphics, Windowing, and Events Subsystem \(GWES\)](http://go.microsoft.com/fwlink/?LinkId=219552) (<http://go.microsoft.com/fwlink/?LinkId=219552>), which is the interface between the end user, your applications, and the OS.

You can develop a RemoteFX display driver in either of two modes: window-frame display mode or full-screen-only display mode. A window-frame display driver shares the display surface with the RDP client session windows and other local applications. A full-screen-only display driver makes board development and prototyping faster and also reduces CPU load at run time. For more information about the two display modes, see [Display Driver Design Choice: Window-Frame or Full-Screen Display Mode](#) later in this article.

When you develop a RemoteFX-enabled display driver, the RDP client, which is a component on your device that receives RemoteFX packets from the server, needs to communicate directly with your display driver in order to access the capabilities of the decoding and display hardware. The RDP client processes these packets and then sends the data to the RemoteFX-enabled display driver that you created and loaded.

RemoteFX uses a set of escape sequences together with the [DrvEscape](http://go.microsoft.com/fwlink/?LinkId=218939) (<http://go.microsoft.com/fwlink/?LinkId=218939>) function to enable the RDP client to communicate directly with the display driver and to offload display data to specialized coprocessor hardware, such as an application-specific integrated circuit (ASIC) or a digital signal processor (DSP). For more information about coprocessor hardware, see [Using a Coprocessor for Hardware Acceleration](#) later in this article.

The RDP client interacts with the RemoteFX display driver via well-defined interfaces that use the set of Graphics Device Interface (GDI) escape sequences previously mentioned. You need to use the GDI escape sequence interface to implement your thin-client RemoteFX display driver. In addition to supporting the typical graphics-rendering operations, the display driver must also support the applicable escape sequences. For more information about the escape sequences, including which escape sequences are used for each of the two display-driver modes, see [Implementing a RemoteFX Display Driver](#) later in this article.

The [ExtEscape](http://go.microsoft.com/fwlink/?LinkId=218940) (<http://go.microsoft.com/fwlink/?LinkId=218940>) function is used to pass the GDI escape sequences from the RDP client to the display driver. The RDP client calls the [ExtEscape](http://go.microsoft.com/fwlink/?LinkId=218940) (<http://go.microsoft.com/fwlink/?LinkId=218940>) function that is defined in Compact 7 and passes in an escape sequence and the RemoteFX-encoded display data. The [ExtEscape](http://go.microsoft.com/fwlink/?LinkId=218940) (<http://go.microsoft.com/fwlink/?LinkId=218940>) function then passes the escape sequence and display data to the [DrvEscape](http://go.microsoft.com/fwlink/?LinkId=218939) (<http://go.microsoft.com/fwlink/?LinkId=218939>) function in the display driver. The capabilities of the [DrvEscape](http://go.microsoft.com/fwlink/?LinkId=218939) (<http://go.microsoft.com/fwlink/?LinkId=218939>) function must be implemented by the OEM. The [DrvEscape](http://go.microsoft.com/fwlink/?LinkId=218939) (<http://go.microsoft.com/fwlink/?LinkId=218939>) function processes the escape sequence and display data and returns a value to the [ExtEscape](#)

(<http://go.microsoft.com/fwlink/?LinkID=218940>) function, which then passes the value to the RDP client.

Using a Coprocessor for Hardware Acceleration

Client CPU performance is enhanced by using a coprocessor for hardware-accelerated decoding of RemoteFX data. Dedicated coprocessor hardware, such as an ASIC, can perform decoding of RemoteFX data. The coprocessor requires a less powerful CPU and less memory on your thin client. You can obtain significant performance improvements over CPU-based software decoding when you combine coprocessor offloading with the high-compression ratio of RemoteFX.

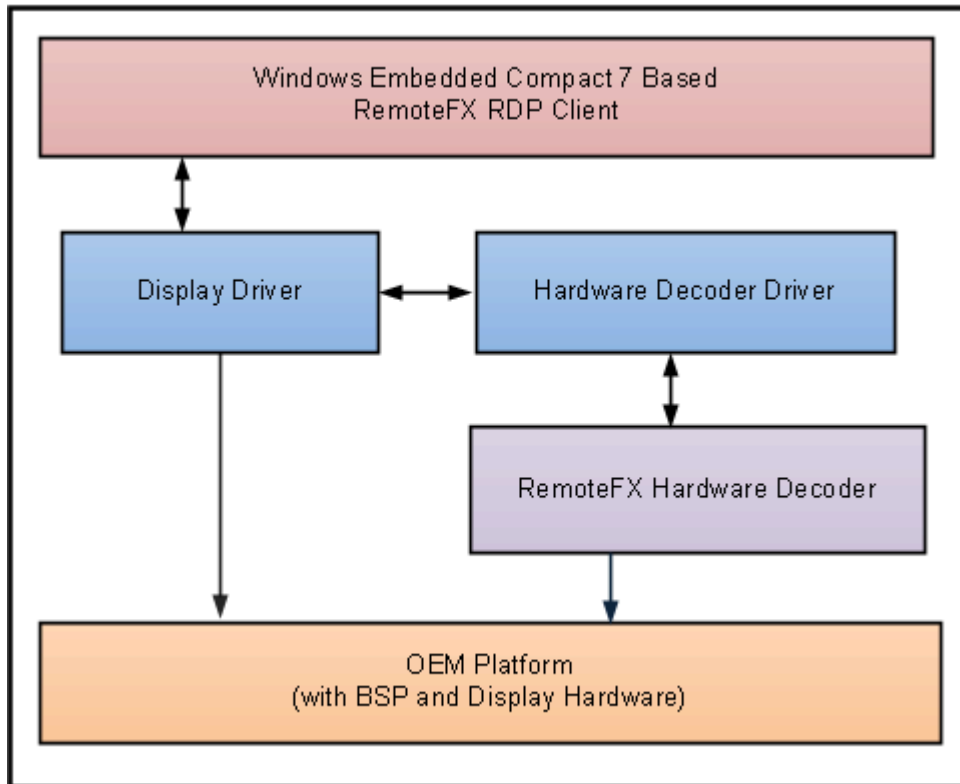
If you decide to build a thin client that uses coprocessor offloading to decompress RemoteFX compressed data, you also need to develop a display driver. Your display driver must support a specified set of RemoteFX interfaces and the driver must communicate with hardware that is capable of processing RemoteFX data.

When you use coprocessor offloading, the RDP client receives RemoteFX packets from the server, processes these packets, and then sends the data to a RemoteFX-enabled display driver. The display driver is responsible for processing the RemoteFX data and making the processed data available to the hardware decoder coprocessor. The RDP client then determines where to display the data on a rendering surface. The RDP client interacts with a display driver via a set of well-defined interfaces that use Graphics Device Interface (GDI) escape sequences.

The following diagram shows the relationships among the different elements that are required to create a RemoteFX-enabled thin client that uses hardware decoding. It also shows the dependent components that you need to create, such as the display driver and the hardware decoder driver that work with a RemoteFX-enabled client.

The RemoteFX-encoded data stream from the server is passed down through the RDP stack, sent to the hardware decoder for decoding, and then displayed through the display controller.

Figure 1: RemoteFX Thin Client Architecture with Hardware Decoder



System integrators who want to develop RemoteFX hardware or to implement the RemoteFX algorithm in a coprocessor need to obtain a RemoteFX Partner license. Partners who are interested in obtaining this information are encouraged to contact their Technical Account Manager to learn more about becoming a RemoteFX Partner. If you are not a partner, see the [Microsoft RemoteFX](http://go.microsoft.com/fwlink/?LinkId=215879) (<http://go.microsoft.com/fwlink/?LinkId=215879>) site for information about joining the partner program.

Display Driver Design Guidelines

Compact 7 uses a display driver model for hardware acceleration of RemoteFX because during a RemoteFX session, the RDP client might request decoding for large amounts of display data. When Compact 7 uses the display driver to process data, any decompressed data can be sent directly to a display controller, without being sent through numerous memory-transfer operations.

Display Driver Design Choice: Window-Frame or Full-Screen Display Mode

You can develop a RemoteFX display driver by using either of the following methods:

- As a full-featured display driver that supports RDP window-frame mode, where the display surface is shared between the RDP window and other local applications.

Figure 2: Window-Frame Display Mode



- As a full-screen-only display driver (only used for hardware-assisted decoding). You set full-screen mode in the registry of the client device by using the **FullScreenRFXOnly** subkey. A full-screen-only hardware-assisted display driver makes board development and prototyping faster because display data can be sent directly to the frame buffer from the decode buffer (see figure 5). A full-screen-only display driver also reduces CPU load at run time.

Figure 3: Full-Screen Display Mode

Performance Optimization for the RemoteFX Thin-Client Display Driver

To optimize the performance of your RemoteFX display driver, consider the following:

- Do not use a software Graphics Primitive Engine (GPE) because it puts more load on the CPU. You can obtain optimal display performance when the video card is capable of supporting the full set of Graphics Device Interface (GDI) raster operations on Compact 7.
- Use direct memory access (DMA) whenever possible to move the decoded data in video memory.
- Use a hardware-accelerated BitBlt operation.

Implementing a RemoteFX Display Driver

Display drivers connect the Windows Embedded Compact 7 (Compact 7) operating system to the display hardware. These drivers, which OEMs create and load on their client devices, are directly called by the [Graphics, Windowing, and Events Subsystem \(GWES\)](http://go.microsoft.com/fwlink/?LinkId=219552) (<http://go.microsoft.com/fwlink/?LinkId=219552>).

When you develop a RemoteFX-enabled driver, the RDP client, which is a component on your device that receives RemoteFX packets from the server, needs to communicate directly with your display driver in order to access the capabilities of the decoding and display hardware. This feature requires OEMs to create drivers for interacting with their hardware.

RemoteFX uses a set of escape sequences together with the [DrvEscape](http://go.microsoft.com/fwlink/?LinkId=218939) (<http://go.microsoft.com/fwlink/?LinkId=218939>) function to enable the RDP client to communicate directly with the display driver and to offload the data to specialized coprocessor hardware, such as an application-specific integrated circuit (ASIC) or digital signal processor (DSP).

The RDP client interacts with a display driver via a set of well-defined interfaces that use GDI escape sequences. You need to use the GDI escape sequence interface to implement your thin-client RemoteFX display driver. In addition to supporting the typical graphics-rendering operations, the display driver must support the applicable RemoteFX escape sequences that are shown in Table 1.

The RDP client invokes the Compact 7–defined [ExtEscape](http://go.microsoft.com/fwlink/?LinkId=218940) (<http://go.microsoft.com/fwlink/?LinkId=218940>) function to pass an escape sequence and RemoteFX-encoded data stream to a RemoteFX-enabled display driver. [ExtEscape](http://go.microsoft.com/fwlink/?LinkId=218940) (<http://go.microsoft.com/fwlink/?LinkId=218940>), in turn, passes those parameters to the display driver's [DrvEscape](http://go.microsoft.com/fwlink/?LinkId=218939) (<http://go.microsoft.com/fwlink/?LinkId=218939>) function. [DrvEscape](http://go.microsoft.com/fwlink/?LinkId=218939) processes the escape code and returns a value to [ExtEscape](http://go.microsoft.com/fwlink/?LinkId=218940) (<http://go.microsoft.com/fwlink/?LinkId=218940>), which then returns the value to the application.

Which escape sequences you use in your driver depend on the display mode that you choose for your client device. For more information about choosing a display mode, see [Display Driver Design Choice: Window-Frame or Full-Screen Display Mode](#) earlier in this article.

Some escape sequences are used by both display modes. Others apply to just one display mode. The following table includes a “Display Mode” column that shows you the escape sequences that are required by each display. When an escape sequence is required by both types of display modes, the column indicates “Both.”

Table 1: Escape Sequences That the Display Driver Must Support in Each Display Mode

| Escape Sequence | Value | Display Mode | Description |
|-----------------|---------------------|--------------|-------------------------------------------------------------------|
| QUERYESCSUPPORT | QUERYESCSUPPORT (8) | Both | Queries whether the driver supports a particular escape function. |

| Escape Sequence | Value | Display Mode | Description |
|-----------------------------------------|------------------|-------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| | | | QUERYESCSUPPORT is the only predefined value. |
| ESCAPE_GET_CAPABILITIES | 0x20103 (131331) | Both | Allows the codec library in the RDP client to query the hardware capabilities (for example, the supported tile size) of the display driver. |
| ESCAPE_DEC3 | 0x20101 (131329) | Both | Allows the codec library to decode compressed data into the video memory buffer and optionally, to transfer decoded data back to system memory. |
| ESCAPE_COPY_2BMP | 0x20102 (131330) | Window-frame only | Allows the codec library to copy decoded data directly to the shadow bitmap. |
| ESCAPE_EXIT_CA | 0x20199 (131481) | Full-screen only | When full-screen mode is enabled and the RemoteFX session ends, the RDP client sends the ESCAPE_EXIT_CA escape sequence to the display driver. The display driver uses this notification to change back to non-RemoteFX display mode. |

The following sections provide flow charts for the two display-modes and code examples for implementing a RemoteFX display driver:

- [RemoteFX Window-Frame and Full-Screen Display Modes](#)
- [DrvEscape Function](#)
- [ESCAPE_GET_CAPABILITIES](#)

- [ESCAPE_DEC3](#)
- [ESCAPE_COPY_2BMP](#)
- [ESCAPE_EXIT_CA](#)
- [Extracting Rectangle Information from RemoteFX Display Data](#)

RemoteFX Window-Frame and Full-Screen Display Modes

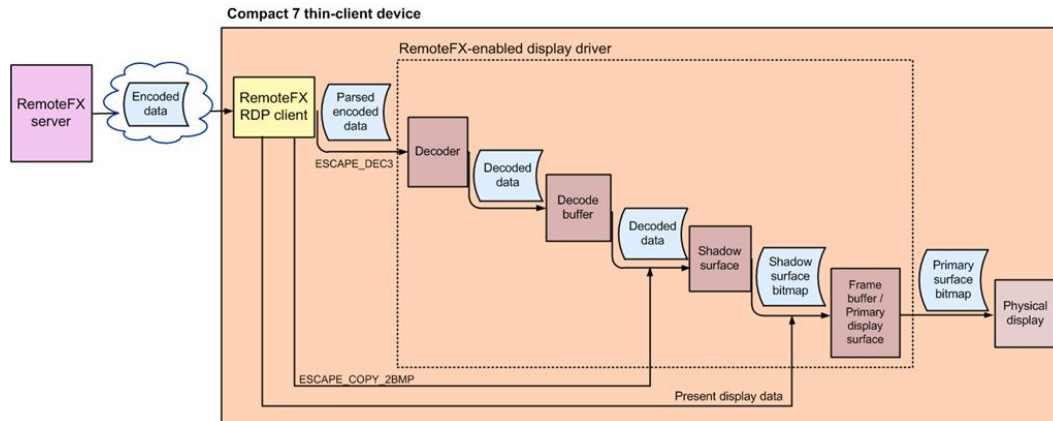
You can develop a RemoteFX display driver in window-frame display mode or in full-screen-only display mode. A window-frame display driver shares the display surface with the RDP client session windows and other local applications. A full-screen-only display driver makes board development and prototyping faster and also reduces CPU load at run time.

RemoteFX Window-Frame Display Mode

If you want your application to use window-frame display mode, your driver must be designed to share the display surface with the RDP client session windows and other local applications. The process that is used by RemoteFX hardware-accelerated decoding to offload coprocessor work from the RDP client to the Windows server follows this sequence:

1. On initialization, the RDP client gets the hardware capabilities from the display driver by using the [ESCAPE_GET_CAPABILITIES](#) escape sequence and then negotiates RDP capabilities from the server.
2. At run time, the RDP client receives and parses the RemoteFX tile set data from the server.
3. The client offloads the encoded tile set data to the display driver for decoding by using the [ESCAPE_DEC3](#) escape sequence.
4. The decoder processes the data and outputs the decoded data to the decode buffer.
5. The RDP client uses the [ESCAPE_COPY_2BMP](#) escape sequence to move decoded data from the decode buffer to the shadow surface. Hardware acceleration is recommended to transfer the decoded bitmap to the shadow surface.
6. The RDP client presents the shadow bitmap content into the frame buffer, which is then output to the physical display. Hardware-accelerated BitBlt is recommended to transfer the shadow bitmap to the frame buffer.

The following figure represents this process in a diagram.

Figure 4: Window-Frame Display Mode RemoteFX Hardware Offload Flow Chart

RemoteFX Full-Screen Display Mode

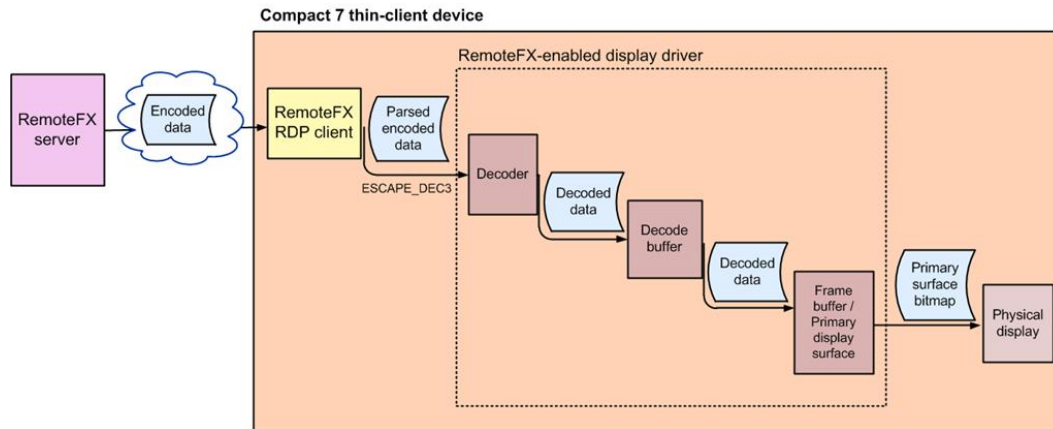
In Compact 7, OEMs have the option of implementing full-screen display mode with RemoteFX. A full-screen-only display driver makes board development and prototyping faster and also reduces CPU load at run time because decoded data is written from the decode buffer directly to the frame buffer.

If you want to use full-screen display mode for your application, the process that is used by RemoteFX hardware-accelerated decoding to offload coprocessor work from the RDP client to the Windows server follows this sequence:

1. On initialization, the RDP client gets the hardware capabilities from the display driver by using the [ESCAPE_GET_CAPABILITIES](#) escape sequence and then negotiates RDP capabilities from the server.
2. At run time, the RDP client receives and parses the RemoteFX tile set data from the server.
3. The client offloads the encoded tile set data to the display driver for decoding by using the [ESCAPE_DEC3](#) escape sequence.
4. The decoder processes the data and outputs the decoded data to the decode buffer.
5. The display driver moves the decoded data from the decode buffer to the frame buffer.

The following figure represents this process in a diagram.

Figure 5: Full-Screen Display Mode RemoteFX Hardware Offload Flow Chart



You enable full-screen display mode by setting the **FullScreenRFXOnly** registry key on the client device. To enable RemoteFX full-screen display mode, create the following registry entry in the Rdp.reg file:

```
[HKEY_LOCAL_MACHINE\Software\Microsoft\Terminal Server Client]
"FullScreenRFXOnly"=dword:1
```

If your RemoteFX implementation uses hardware-accelerated decoding, **FullScreenRFXOnly** disables the Graphics Device Interface (GDI), which is only used for window-frame display mode. The blue connection bar is also hidden.

When **FullScreenRFXOnly** is enabled, the RDP client does not call the **ESCAPE_COPY_2BMP** escape sequence, the RDP client does not present any RemoteFX data to the display screen, and it is the responsibility of the display driver to display the decoded RemoteFX data. For more information and for code examples, see [Extracting Rectangle Information from RemoteFX Display Data](#).

DrvEscape Function

The RDP client calls the [ExtEscape](http://go.microsoft.com/fwlink/?LinkId=218940) (http://go.microsoft.com/fwlink/?LinkId=218940) function that is defined in Compact 7 and passes in an escape sequence and the RemoteFX-encoded display data. The **ExtEscape** function then passes the escape sequence and display data to the **DrvEscape** function in the display driver.

The **DrvEscape** function is a display driver function that allows applications to access the capabilities of a particular device, such as hardware-accelerated decoding; these device capabilities are not necessarily available within the application or the Compact 7 operating system. The **DrvEscape** function must be implemented by an OEM in order to access the hardware-accelerated decoding capabilities of the device. For more information about this function, see [DrvEscape](http://go.microsoft.com/fwlink/?LinkId=218939) (http://go.microsoft.com/fwlink/?LinkId=218939) on MSDN.

The *iEsc* parameter of the **DrvEscape** function queries whether the driver supports a particular escape function. **QUERYESCSUPPORT** is the only predefined value.

DrvEscape should return a nonzero value for any supported escape sequences when *iEsc* is set to **QUERYESCSUPPORT**. In this case, *pvIn* points to an escape function number. During initialization and before offloading of the RemoteFX-encoded data stream through the **ExtEscape** function, the RDP client makes multiple **QUERYESCSUPPORT** calls to verify whether an escape sequence is supported by the display driver. The following code example shows how to implement the **DrvEscape** function in the display driver.



Important

For readability, the following code example does not contain security or error handling. Do not use the following code in a production environment.

```
ULONG DrvEscape(
    SURFOBJ * pso,
    ULONG     iEsc,
    ULONG     cjIn,
    void      * pvIn,
    ULONG     cjOut,
    void      * pvOut
)
{
    if (iEsc == QUERYESCSUPPORT)
    {
        if ( *(DWORD*)pvIn == ESCAPE_GET_CAPABILITIES
            || *(DWORD*)pvIn == ESCAPE_DEC3
            || *(DWORD*)pvIn == ESCAPE_COPY_2BMP )
        {
            // The escape functions are supported.
            return 1;
        }
    }
    else if (iEsc == ESCAPE_GET_CAPABILITIES)
    {
        int RetVal = 0;

        RetVal = esc_get_capabilities(pso, iEsc, cjIn, pvIn, cjOut, pvOut);

        return RetVal;
    }
}
```

```

    }

    else if (iEsc == ESCAPE_DEC3)
    {
        int    RetVal = 0;

        RetVal = esc_dec3(pso,iEsc,cjIn,pvIn,cjOut,pvOut);

        return RetVal;
    }

    else if (iEsc == ESCAPE_COPY_2BMP)
    {
        int    RetVal = 0;

        RetVal = esc_copy_2bmp(pso,iEsc,cjIn,pvIn);

        return RetVal;
    }

    return 0;
}

```

The Graphics Device Interface (GDI) passes data directly from the RDP client to the display driver, which means that the **DrvEscape** function must validate all input arguments. The following list shows what this function must verify:

- The *iEsc* parameter value represents a valid query.
- The *cjIn* parameter value is valid for the specified query.
- The *pvlIn* parameter contents are valid for the specified query.
- The *cjOut* parameter contents are valid for the specified query.

Driver-specific escape sequences may conflict with those that are used in other display drivers; so the display driver needs to validate the escape parameters before processing the escape data. One way to do that is for the driver to validate the input block size, output block size, and the input block parameters.

For added security, drivers should include a specific value that OEMs must set in every input block to ensure that the input block is from a trusted source. In RemoteFX, this security mechanism is achieved by using an escape header data structure. The escape headers in RemoteFX always contain a magic value that corresponds to 0xBEEFABCE.

In addition to providing a driver that has the usual display capabilities, as an OEM, you need to implement the following escape sequences and functions in your display driver to access the capabilities of the hardware:

- **Window-frame mode display drivers** must support the **QUERYESCSUPPORT**, [ESCAPE_GET_CAPABILITIES](#), [ESCAPE_DEC3](#), and [ESCAPE_COPY_2BMP](#) escape sequences in the **DrvEscape** function.

- **Full-screen mode display drivers** must implement the **QUERYESCSUPPORT**, [ESCAPE_GET_CAPABILITIES](#), [ESCAPE_DEC3](#), and [ESCAPE_EXIT_CA](#) escape sequences in the **DrvEscape** function as well as implement functionality for [Extracting Rectangle Information from RemoteFX Display Data](#).

The following sections introduce these escape sequences and provide code examples for how to use them.

ESCAPE_GET_CAPABILITIES

The **ESCAPE_GET_CAPABILITIES** escape sequence allows the RDP client to query the capabilities of the hardware. Currently, only the RemoteFX hardware tile size is used. Some bytes are reserved for future use. The following code example shows the structure declaration that is required for the **escape_get_capabilities** function.

Important

For readability, the following code examples do not contain security or error handling. Do not use the following code examples in a production environment.

```
#define DRAW_ESCAPE_CODE_BEGIN          0x20000
#define ESCAPE_MAGIN_IN                 0xBEEFABCE
#define ESCAPE_CODE_BEGIN               (DRAW_ESCAPE_CODE_BEGIN+0x100)

struct _hdr
{
    ULONG magic;
    // const: 0xBEEFABCE
    ULONG size;
    ULONG code;
};

#define ESCAPE_GET_CAPABILITIES         (ESCAPE_CODE_BEGIN+3)
struct esc_get_capabilities_in
{
    struct _hdr hdr;
};
struct esc_get_capabilities_out
{
    ULONG tile_size;
```

```

        ULONG reserved[2];
    };

```

The following code example shows how **ESCAPE_GET_CAPABILITIES** should be implemented in a display driver. The function validates the escape parameters that the escape header received from the RDP client to ensure that the input buffer is from a trusted source.

```

BOOL esc_get_capabilities(
    SURF_OBJ*    pso,
    ULONG        iEsc,
    ULONG        cjIn,
    PVOID        pvIn,
    ULONG        cjOut,
    PVOID        pvOut
)
{
    struct esc_get_capabilities_in* p_in = NULL;
    struct esc_get_capabilities_out* p_out = NULL;
    BOOL status = FALSE;
    int tileSize = 64;
    do {

        if (!pso)
        {
            break;
        }

        if (
            (!pvIn)
            ||
            (cjIn != sizeof(*p_in)))
        {
            break;
        }

        p_in = (struct esc_get_capabilities_in*)pvIn;
    } while (FALSE);
}

```

```

    if (
        (p_in->hdr.code != iEsc)
        ||
        (p_in->hdr.magic != ESCAPE_MAGIN_IN)
        ||
        (p_in->hdr.size != sizeof(*p_in)))
    {
        break;
    }

    if (
        (!pvOut)
        ||
        (cjOut != sizeof(*p_out)))
    {
        break;
    }

    p_out = (struct esc_get_capabilities_out*)pvOut;
    p_out->tile_size = tileSize;
    //setting tile size to 64
    status = TRUE;

} while (0);

return status;

}

```

The *pvIn* parameter is a pointer to the input data for this escape call and consists of escape header information, which is defined in the **esc_get_capabilities_in** struct. The *pvOut* parameter is a pointer to the output buffer for this escape call and consists of the tile size and reserved bytes, which are defined in the **esc_get_capabilities_out** struct.

The **tile_size** member variable is set to 64 and passed back to the RDP client through the [ExtEscape](http://go.microsoft.com/fwlink/?LinkID=218940) (<http://go.microsoft.com/fwlink/?LinkID=218940>) function.

**Caution**

The RDP client sets the tile size internally during decoder initialization. The RemoteFX codec algorithm currently supports 64 x 64 tile sizes. Setting an unsupported tile size in the display driver may cause the RemoteFX session to not work correctly.

ESCAPE_DEC3

The **ESCAPE_DEC3** escape sequence offloads the RemoteFX display data to the specialized hardware decoder for decoding and optionally, transfers decoded data back to system memory. The decoded data is handled in hardware, such as in video memory, because the data is no longer in system memory. We do not recommend the option of transferring decoded data back to system memory because the memory transfer causes slower performance.

The following code example shows the structure declarations that are required for the **ESCAPE_DEC3** escape sequence.

**Important**

For readability, the following code examples do not contain security or error handling. Do not use the following code examples in a production environment.

```
#define ESCAPE_DEC3                                (ESCAPE_CODE_BEGIN+1)

struct ct_rect
{
    LONG left;
    LONG top;
    LONG right;
    LONG bottom;
};

typedef struct ct_rect CRect;

struct Rect
{
    short llX;
    short llY;
    short urX;
    short urY;
};
```

```
typedef struct
{
    int size;
    BYTE buffer;
}
Buffer1D;

struct esc_dec3_in
{
    struct _hdr    hdr;
    UCHAR*        comp_data;
    ULONG         comp_data_size;
    struct ct_rect* rects;
    ULONG         rect_size;
    UCHAR*        uncomp_data;
    ULONG         uncomp_data_size;
};

struct esc_dec3_out
{
    struct dec3_out ioctl1_dec3_out;
    ULONG         channelId;
    ULONG         nBytesConsumed;
};

struct __IndexBuffer
{
    // const: 0xABCDDCB1
    ULONG    marker;
    ULONG    width;
    ULONG    height;
    PVOID    tileset;
    PVOID*   tileBuffer;
}
```

```

    PULONG   tileBufferLength;
}

typedef struct tagTS_GFX_RECT {
    LONG left;
    LONG top;
    LONG right;
    LONG bottom;
    } TS_GFX_RECT, *PTS_GFX_RECT;

// Refer to the RemoteFX protocol specification for more details related to the
// following structures.
typedef struct TagTS_RFX_CODEC_QUANT
{
    // Level-3 quantization factor.
    BYTE LL3 : 4;
    BYTE LH3 : 4;
    BYTE HL3 : 4;
    BYTE HH3 : 4;

    // Level-2 quantization factor.
    BYTE LH2 : 4;
    BYTE HL2 : 4;
    BYTE HH2 : 4;

    // Level-1 quantization factor.
    BYTE LH1 : 4;
    BYTE HL1 : 4;
    BYTE HH1 : 4;
}
TS_RFX_CODEC_QUANT;

typedef struct TagTS_RFX_TILE

```

```

{
    // H/w Tile Marker:0x90ff
    USHORT      marker;

    // 0xa
    USHORT      size

    // The X-index of the encoded tile in the screen tile grid.
    BYTE        xIdx;

    // The Y-index of the encoded tile in the screen tile grid.
    BYTE        yIdx;

    // Specifies the size, in bytes, of the Y-Data field of YUVData.
    USHORT      YLen;

    // Specifies the size, in bytes, of the U-Data field of YUVData.
    USHORT      ULen;

    // Specifies the size, in bytes, of the V-Data field of YUVData.
    USHORT      VLen;

    // The start of the encoded data for the YUV-component of the tile.
    BYTE        YUVData[1];
} TS_RFX_TILE;

typedef struct TagTS_RFX_TILESET
{
    // Block Type WBT_EXTENSION: 0xCCC7
    USHORT      blockType;
    UINT        blockLen;

    // const: 1

```

```

    BYTE                codecId;

    BYTE                channelId;

    // 0xCAC2

    USHORT              subtype;

    USHORT              idx;

    USHORT              lt : 1;

    USHORT              flags : 3;

    USHORT              cct : 2;

    USHORT              xft : 4;

    USHORT              et : 4;

    USHORT              qt : 2;

    BYTE                numQuant;

    BYTE                tileSize;

    USHORT              numTiles;

    ULONG               tilesDataSize;

    TS_RFX_CODEC_QUANT   quantVals;

    TS_RFX_TILE          tiles[1];
} TS_RFX_TILESET;

```

The following code example shows how the **ESCAPE_DEC3** sequence is implemented in a display driver. The **esc_dec3** function validates the escape parameters and the escape header received from the RDP client to ensure that the input buffer is from a trusted source.

```

#define OUTRECTS_BUFFER_SIZE_DFLT 128

// Global variable declaration

Buffer1D _OutBuf;

Rect      *_pOutRects;

int        _cOutRects;

int        _cOutRectsSize;

int        Width;

int        Height;

int        TileSize;

```

```
PTS_GFX_RECT pDestFrame;
```

```

BOOL esc_dec3(
    SURFOBJ*    pso,
    ULONG        iEsc,
    ULONG        cjIn,
    PVOID        pvIn,
    ULONG        cjOut,
    PVOID        pvOut
)
{
    struct esc_dec3_in*    p_in                = NULL;
    struct esc_dec3_out*    p_out                = NULL;
    struct _IndexBuffer*    _InBuf                = NULL;
    struct ct_rect*    rects                = NULL;
    BOOL                    status                = FALSE;
    ULONG                    bytesConsumed        = 0;

    int nChannelID;
    int cbConsumed = 0;

    do {
        if (!pso)
        {
            break;
        }

        if (
            (!pvIn)
            ||
            (cjIn != sizeof(*p_in))
        )
        {
            break;
        }

        p_in = (struct esc_dec3_in*)pvIn;
    }

```

```

if (
    (p_in->hdr.code != iEsc)
    ||
    (p_in->hdr.magic != ESCAPE_MAGIN_IN)
    ||
    (p_in->hdr.size != sizeof(*p_in)))
{
    break;
}

if (
    (!pvOut)
    ||
    (cjOut != sizeof(*p_out)))
{
    break;
}

p_out = (struct esc_dec3_out*)pvOut;

Buffer1D InBuf;

_InBuf = (struct _IndexBuffer*)p_in->comp_data;

p_out->nBytesConsumed = p_in->comp_data_size;
p_out->ioc1_dec3_out.cx = _InBuf->width;
p_out->ioc1_dec3_out.cy = _InBuf->height;
//During initialization, allocate memory as shown below
pDestFrame = (PTS_GFX_RECT)malloc(sizeof PTS_GFX_RECT);
pDestFrame->left =0;
pDestFrame->top =0;
pDestFrame->right =0;
pDestFrame->bottom=0;

```

```

    TS_RFX_TILESET *tileSet = reinterpret_cast<TS_RFX_TILESET*>(_InBuf-
>tileset);

    // Call ScanSyncFrameBeginBlock function only when registry
    // FullScreenRFXOnly is set.
    // Refer to the Full Screen Mode section for more details.
    ScanSyncFrameBeginBlock((BYTE *)_InBuf->tileset);
    InBuf.buffer = (BYTE *)_InBuf->tileset;
    InBuf.size = tileSet->tilesDataSize;
    _pOutRects = (Rect *)malloc(OUTRECTS_BUFFER_SIZE_DFLT *
sizeof(Rect));

    _cOutRectsSize = OUTRECTS_BUFFER_SIZE_DFLT;
    Width = _InBuf->width;
    Height = _InBuf->height;
    TileSize = tileSet->tileSize;

    for(ULONG i=0; i< tileSet->numTiles; i++)
    {
        BYTE *buf = (BYTE*)_InBuf->tileBuffer[i];
        TS_RFX_TILE *pTile = reinterpret_cast<TS_RFX_TILE*>(buf);
        int xIdx = pTile->xIdx;
        int yIdx = pTile->yIdx;

        int lftX = xIdx * TileSize;
        int lftY = yIdx * TileSize;
        int rgtX = lftX + TileSize;
        int rgtY = lftY + TileSize;

        p_in->rects[i].left = lftX;
        p_in->rects[i].top = lftY;
        p_in->rects[i].right = rgtX;
        p_in->rects[i].bottom = rgtY;
    }

```

```

        // Allocate buffer for the rectangle list returned by the decoder
        _OutBuf.size = Width * Height * 32 / 8;
        _OutBuf.buffer = BYTE(malloc(_OutBuf.size));
        Buffer1D *pOutBuf = (_OutBuf.buffer) ? &_OutBuf : NULL;

//TO DO: The encoded data will be decoded using the specialized hardware decoder
//such as ASIC or DSP. The Decode function below needs to be implemented by the
//decoder driver, which will interface with specialized hardware.

//The decoder driver will drive the decoding process, off-loading the data to
//specialized hardware. It communicates with the hardware through the display
//driver interface and transfers compressed tiles into the decoder.

        Decode(InBuf, _cOutRectsSize, _pOutRects, _cOutRects, cbConsumed,
               nChannelID, Width, Height, pOutBuf);

        p_out->iocctl_dec3_out.rect_count = tileSet->numTiles;
        status = TRUE;

    } while (0);
    return status;
}

```

The encoded data is decoded by using a specialized hardware decoder, such as ASIC or a digital signal processor (DSP). The **Decode** function in the previous example needs to be implemented by a decoder driver, which interfaces with specialized hardware. The decoder driver drives the decoding process, off-loading the data to specialized hardware. The decoder driver communicates with the hardware through the display driver interface and transfers compressed tiles into the decoder. Direct memory access (DMA) is recommended to transfer the decoded data in video memory without overloading the processor. If you do not use DMA, the processor copies all the data from the source to the destination, making the processor unavailable for other tasks.

ESCAPE_COPY_2BMP

The **ESCAPE_COPY_2BMP** sequence is sent by the RDP client immediately after the **ESCAPE_DEC3** sequence to copy *dirty rectangles* from the decoder surface to the shadow bitmap. A *dirty rectangle* is a list of changed rectangles on the screen. A shadow bitmap is a device-compatible bitmap that is created by the display driver during the initialization of the RemoteFX component. The shadow surface

represents the video memory that belongs to the full physical screen and can be used as an off-screen buffer to update the primary display surface. Calling the **ESCAPE_COPY_2BMP** invokes the RDP client to blit the dirty rectangles to the correct location in a temporary frame buffer and then to blit all the updated rectangles to an output frame buffer.

The **ESCAPE_COPY_2BMP** call relies on the process of transferring the bitmaps, first to an off-screen graphical frame buffer (or surface), before they are rendered on the screen by the RDP client. The RDP client is responsible for copying the contents of the shadow bitmap to the destination (video memory). To achieve optimal performance from RemoteFX, we recommend that your display driver supports video memory and hardware-accelerated blitting operations, which are considerably faster than software.

If you do not implement hardware-accelerated graphics, Compact 7 uses the Graphics Device Interface (GDI) to draw UI objects pixel-by-pixel onto the primary display surface, performing all the operations in software and resulting in reduced performance on the client.

The following code example shows the structure declaration that is required for this escape function.



Important

For readability, the following code examples do not contain security or error handling. Do not use the following code examples in a production environment.

```
typedef struct RECT {
    LONG left;
    LONG top;
    LONG right;
    LONG bottom;
};

struct copy_rectangle_2bmp
{
    ULONG transX;
    ULONG transY;
    ULONG rect_count;
    RECT* rects;
};

struct esc_copy_rectangle_2bmp_in
{
    struct _hdr hdr;
    struct copy_rectangle_2bmp copy2bmp;
```

```

HBITMAP hbitmap;

};

```

The following code example shows how the **ESCAPE_COPY_2MP** sequence is implemented in the display driver. The **esc_copy_2bmp** function validates the escape parameters and escape header that are received from the RDP client to ensure that the input buffer is from a trusted source.

```

BOOL esc_copy_2bmp(
    SURFOBJ*    pso,
    ULONG       iEsc,
    ULONG       cjIn,
    PVOID       pvIn
)
{
    struct esc_copy_rectangle_2bmp_in*    p_in        = NULL;
    BOOL    status                        = FALSE;
    do {
        if (
            (!pvIn)
            ||
            (cjIn != sizeof(*p_in)))
        {
            break;
        }
        p_in = (struct esc_copy_rectangle_2bmp_in*)pvIn;

        if (
            (p_in->hdr.code != iEsc)
            ||
            (p_in->hdr.magic != ESCAPE_MAGIN_IN)
            ||
            (p_in->hdr.size != sizeof(*p_in)))
        {
            break;
        }
    }
}

```

```

HBITMAP hBitmap = p_in->hbitmap;

// TO DO: The SetBitmapBits API is provided just for reference here. The
// display driver should implement the functionality to map the hbitmap
// to the surface created by the display driver. Use the bitmap handle
// to move decoded data from the decode buffer to the shadow surface.

SetBitmapBits(hBitmap, _OutBuf.size, _OutBuf.buffer);

status = TRUE;

} while (0);

return status;

}

```

The RDP client uses the **ESCAPE_COPY_2BMP** escape sequence to move decoded data from the decode buffer to the shadow surface. Hardware acceleration is recommended in transferring the decoded bitmap to the shadow surface. The RDP client is responsible for presenting the shadow bitmap content into the primary display surface.

The **SetBitmapBits** function is a high-level API that is used in the preceding code example to move the bits from the decode buffer to the shadow surface. The display driver implements the functionality to map the **hbitmap** to the surface that is created by the display driver. The bitmap handle is used to move decoded data from the decode buffer to the shadow surface. We recommend that the display driver also implements hardware-accelerated BitBlt to transfer the shadow bitmap to the primary display surface.

The Remote Desktop Virtualization Host (RD Virtualization Host) and Remote Desktop Session Host (RD Session Host) send different types of rectangle coordinates to the Compact 7 device in the RemoteFX-encoded data stream. In the case of the RD Virtualization Host, each decode call on the RDP client provides the driver with the list of all dirty rectangles and the exact rectangle coordinates of the rectangles. These coordinates, for example, 240, 320, 264, or 356, are used to determine where the bitmap needs to be placed on the screen.

In the case of the RD Session Host, each decode call provides only one dirty rectangle with coordinates starting from (0,0). These coordinates effectively describe the size of the bitmap that needs to be updated on the screen. To correctly place the bitmap on the screen, a destination rectangle is required to calculate the dirty region. This information is sent to the display driver in the **transX** and **transY** members of the **copy_rectangle_2bmp** structure. If the display driver implements the **ESCAPE_COPY2_BMP** sequence, the bitmap is placed on the screen by using the **transX** and **transY** coordinates together with **rect_count** and **rects** information.

Note

The previous sample display driver does not calculate the dirty region because the **SetBitmapBits** function does not need that information when the device is connected to the RD Virtualization Host. You may need to add additional functionality to meet the requirements of your specific device.

ESCAPE_EXIT_CA

If the RemoteFX session ends while the device is in full-screen display mode, the RDP client sends the **ESCAPE_EXIT_CA** escape sequence to the display driver. The display driver uses this notification to return to non-RemoteFX display mode. The following code example demonstrates the **ESCAPE_EXIT_CA** escape sequence.



Note

In window-frame display mode, there is no equivalent to the **ESCAPE_EXIT_CA** escape sequence.



Important

For readability, the following code example does not contain security or error handling. Do not use the following code in a production environment.

```
#define ESCAPE_EXIT_CA                                (ESCAPE_CODE_BEGIN+0x99)

ULONG DrvEscape(
    SURFOBJ * pso,
    ULONG     iEsc,
    ULONG     cjIn,
    void      * pvIn,
    ULONG     cjOut,
    void      * pvOut
)
{
    if (iEsc == QUERYESCSUPPORT)
    {
        if ( *(DWORD*)pvIn == ESCAPE_GET_CAPABILITIES
            || *(DWORD*)pvIn == ESCAPE_DEC3
            || *(DWORD*)pvIn == ESCAPE_COPY_2BMP )
        {
            // The escape functions are supported.
            return 1;
        }
        if ( *(DWORD*)pvIn == ESCAPE_EXIT_CA )
        {

```

```

        // TO DO: Switch to normal display mode when FullScreenRFXOnly
        // is set.
        return 1;

    }

    else
    {
        // The escape isn't supported.
        return 0;
    }
}
}

```

Extracting Rectangle Information from RemoteFX Display Data

In full-screen mode, the **ESCAPE_COPY2_BMP** escape sequence is not sent to the display driver; therefore, the display driver must extract the destination rectangle information from the RemoteFX-encoded display data. The destination rectangle information is present before the start of each frame in the RemoteFX-encoded data stream. The destination rectangle information is also present before the **sync** message, which is the first message in a RemoteFX-encoded data stream.

The following code example demonstrates how the destination rectangle information can be extracted from the RemoteFX-encoded display data.



Important

For readability, the following code example does not contain security or error handling. Do not use the following code in a production environment.

```

const int WF_MAGIC = 0xCACCACCA;

enum cmdType
{
    CMDTYPE_FRAME_MARKER = 0x04,
    CMDTYPE_STREAM_SURFACE_BITS = 0x06
}

```

```
};

enum BlockType
{
    WBT_SYNC = 0xCCC0,
    WBT_CODEC_VERSIONS = 0xCCC1,
    WBT_CHANNELS = 0xCCC2,
    WBT_CONTEXT = 0xCCC3,
    WBT_FRAME_BEG = 0xCCC4,
    WBT_FRAME_END = 0xCCC5,
    WBT_REGION = 0xCCC6,
    WBT_EXTENSION = 0xCCC7,
    WBT_LAST = 0xCCC8
};

typedef struct tagTS_DEST_RECT
{
    UINT16  destLeft;
    UINT16  destTop;
    UINT16  destRight;
    UINT16  destBottom;
}TS_DEST_RECT, *PTS_DEST_RECT;

typedef struct tagTS_BITMAP_DATA_EX
{
    UINT8    Bpp;
    UINT8    reserved1;
    UINT8    reserved2;
    UINT8    CodecID;
    UINT16    Width;
    UINT16    Height;
    UINT32    bitmapDataLength;
}TS_BITMAP_DATA_EX , *PTS_BITMAP_DATA_EX;
```

```
typedef struct tagTS_SURFCMD_SET_SURF_BITS
{
    UINT16          CmdType;
    TS_DEST_RECT     destRect;
    TS_BITMAP_DATA_EX bitmapHeader;
} TS_SURFCMD_SET_SURF_BITS;
```

```
typedef struct TAGTS_RFX_BLOCKT
{
    USHORT          blockType;
    UINT            blockLen;
} TS_RFX_BLOCKT;
```

```
typedef struct TagTS_RFX_SYNC
{
    TS_RFX_BLOCKT block;
    UINT          magic;
    USHORT        version;
} TS_RFX_SYNC;
```

```
typedef struct TagTS_RFX_FRAME_BEGIN
{
    TS_RFX_BLOCKT block;
    BYTE          codecId;
    BYTE          channelId;
    UINT          frameIdx;
    USHORT        numRegions;
} TS_RFX_FRAME_BEGIN;
```

```
typedef struct TagTS_RFX_CHANNELS
{
    TS_RFX_BLOCKT block;
```

```

    BYTE          numChannels;

    BYTE          channelId;

    USHORT        width;

    USHORT        height;
} TS_RFX_CHANNELS;

// The function GetDestRectangle fills in the pDestFrame structure from
// destination rectangle information.
void GetDestRectangle( BYTE *mDataBuf)
{
    BYTE *dataBuf = mDataBuf;

    dataBuf = dataBuf - sizeof(TS_SURFCMD_SET_SURF_BITS) + sizeof(short);

    PTS_DEST_RECT Rect = (PTS_DEST_RECT) (dataBuf);

    pDestFrame->left = Rect->destLeft;

    pDestFrame->top = Rect->destTop;

    pDestFrame->right = Rect->destRight;

    pDestFrame->bottom = Rect->destBottom;
}

// The function ScanSyncFrameBeginBlock extracts destination rectangle information
// present before the start of each Frame. For the first message in any encoded
// stream, this information is present before sync message.
void ScanSyncFrameBeginBlock(BYTE *mDataBuf )
{
    BYTE *dataBuf = mDataBuf;

    for (int i=0;
        (*dataBuf != CMDTYPE_STREAM_SURFACE_BITS)  &&
        (*(dataBuf - 2* sizeof(int)) != CMDTYPE_FRAME_MARKER); i++, dataBuf--)
    {
        TS_RFX_BLOCKT *block = (TS_RFX_BLOCKT*)dataBuf;

        switch (block->blockType)
        {
            case WBT_SYNC:

```

```

    {
        TS_RFX_SYNC *ss = (TS_RFX_SYNC*)dataBuf;
        if (ss->block.blockType == WBT_SYNC &&
            ss->magic == WF_MAGIC)
        {
            GetDestRectangle(dataBuf);
        }
    }
    break;

case WBT_FRAME_BEG:
    {
        TS_RFX_FRAME_BEGIN *fb = (TS_RFX_FRAME_BEGIN*)dataBuf;
        BYTE *mBuf = dataBuf - sizeof (TS_RFX_CHANNELS);
        TS_RFX_CHANNELS *ch = (TS_RFX_CHANNELS*)mBuf;
        if (fb->block.blockType == WBT_FRAME_BEG &&
            ch->block.blockType != WBT_CHANNELS )
        {
            GetDestRectangle(dataBuf);
        }
    }
    break;
}

}

// The _cOutRects and _pOutRects out params of decode call provide the number
// of rects and co-ordinates of each rect, for example, _pOutRects can
// be 0,0, 34, 26. To calculate the dirty, or changed, region after decoding is
// done, combine the decoder output with the destination rectangle information as
// follows to place the decoded bitmap on the screen.

for (int i = 0; i < _cOutRects; i++)

```

```

{
    TS_GFX_RECT DirtyRect = {
        pDestFrame->left + _pOutRects-> llX,
        pDestFrame->top + _pOutRects-> llY,
        pDestFrame->left + _pOutRects-> urX,
        pDestFrame->top + _pOutRects-> urY,
    };
}

```

Conclusion

Driver developers at OEMs who want to deploy RemoteFX on thin-client devices receive an implementation roadmap, including guidelines for making design choices and code examples for building a RemoteFX display driver.

The first task for implementing RemoteFX on thin clients that run Compact 7 is to choose between software decoding and hardware-accelerated decoding for decompressing display data on remote clients. Software decoding does not require specialized hardware, but hardware-accelerated decoding typically offers the best performance, especially on low-end hardware. If you decide to use software decoding, you do not need to develop a RemoteFX display driver. We recommend that you use hardware decoding on your thin clients.

RemoteFX technology uses the hardware on servers that run Windows Server 2008 R2 with SP1 to efficiently encode display data before the data is sent to Compact 7 thin clients. Display data is decoded on your clients by the display driver before it is displayed. When you start designing your implementation, it's a good idea to review [Getting Started with Remote FX in Windows Embedded Compact 7](http://go.microsoft.com/fwlink/?LinkID=215085) (<http://go.microsoft.com/fwlink/?LinkID=215085>), which outlines the benefits of implementing RemoteFX and details the hardware and software requirements.

In Compact 7, RemoteFX hardware-accelerated decoding offloads data decompression to a coprocessor, which reduces the workload on the CPUs of your client devices. By using hardware-accelerated decoding, you develop Compact 7 thin clients that use less powerful hardware compared to the typical thin clients that are used with a remote server. With Compact 7, you also use the new RemoteFX enhancements for RDP to build low-cost RemoteFX-enabled thin-client devices.

If you decide to build a thin client that uses coprocessor offloading to decompress RemoteFX compressed data, you also need to develop a display driver. You can develop a RemoteFX display driver in window-frame display mode or in full-screen-only display mode. A window-frame display driver shares the display surface between the RDP client session windows and other local applications. A full-screen-only display driver makes board development and prototyping faster and also reduces CPU load at run time.

RemoteFX uses a set of escape sequences together with the [DrvEscape](http://go.microsoft.com/fwlink/?LinkID=218939) (<http://go.microsoft.com/fwlink/?LinkID=218939>) function to enable the client to communicate directly with the display driver and to offload display data to specialized coprocessor hardware. You need to use the escape sequence interface to implement your thin-client RemoteFX display driver.

In the final section, you receive several code examples that get you started coding your RemoteFX display driver and help you deploy RemoteFX more quickly.

Additional Resources

- [Windows Embedded Compact 7 Documentation](http://go.microsoft.com/fwlink/?LinkId=190787) (<http://go.microsoft.com/fwlink/?LinkId=190787>)
- [Microsoft RemoteFX](http://go.microsoft.com/fwlink/?LinkID=215879) (<http://go.microsoft.com/fwlink/?LinkID=215879>)
- [Getting Started with RemoteFX in Windows Embedded Compact 7](http://go.microsoft.com/fwlink/?LinkID=215085) (<http://go.microsoft.com/fwlink/?LinkID=215085>)
- [DrvEscape](http://go.microsoft.com/fwlink/?LinkID=218939) (<http://go.microsoft.com/fwlink/?LinkID=218939>)
- [RemoteFX Technet Wiki](http://go.microsoft.com/fwlink/?prd=11857&pver=7.1&sbp=Windows%20Embedded%20Compact&plcid=0x409&clcid=0x409&ar=Windows%20Embedded%20Compact%20TechNet&sar=RemoteFX%20TechNet%20Wiki) (<http://go.microsoft.com/fwlink/?prd=11857&pver=7.1&sbp=Windows Embedded Compact&plcid=0x409&clcid=0x409&ar=Windows Embedded Compact TechNet&sar=RemoteFX Technet Wiki>)

This document is provided “as-is.” Information and views expressed in this document, including URL and other Internet Web site references, may change without notice. You bear the risk of using it.

This document does not provide you with any legal rights to any intellectual property in any Microsoft product. You may copy and use this document for your internal, reference purposes.

© 2011 Microsoft. All rights reserved.