Two UEFI Screenshot Utilities Added To UEFI-Utilities-2019

Finnbarr P. Murphy
(fpm@fpmurphy.com)

I have had a very rudimentary UEFI utility for taking screenshots from within the UEFI shell since circa 2012. Recently, I decided to update and polish this utility. The end result is two seperate screenshot capturing utilities – a UEFI shell command line utility (*ScreenShot.efi*) and a UEFI driver-based screenshot utility (ScreenshotDriver.efi.) These are both available on GitHub. in my UEFI-Utilities-2019 repository.

In this post, I describe the key functions (AKA components) of these two utilities. Note that this post loads slowly because of the inclusion of some fullscreen BMP images generated by the utilities – so apologies in advance.

This is the key snippet of code in both utilities:

```
Status = ShowStatus( Gop, Yellow, StartX, StartY, Width, Height );
gBS->Stall(500*1000);
Status = SnapShot( Gop , StartX, StartY, Width, Height );
if (EFI_ERROR(Status)) {
    Status = ShowStatus( Gop, Red, StartX, StartY, Width, Height );
} else {
    Status = ShowStatus( Gop, Lime, StartX, StartY, Width, Height );
}
```

This code displays 4 small 10 pixel squares (size is configurable) designating the top left, top right, bottom left and bottom right of the rectangular area of the screen which will be snapshot-ed. Initially the squares are yellow. They change to green if the screenshot is successfully saved to a file, otherwise they change to red to denote that an error occurred and no scrrenshot was saved.

Here is the source code for *ShowStatus* function:

```
EFI_STATUS
ShowStatus( EFI_GRAPHICS_OUTPUT_PROTOCOL *Gop,
            UINT8 Color,
            UINTN StartX,
            UINTN StartY,
            UINTN Width,
            UINTN Height )
{
    EFI_GRAPHICS_OUTPUT_BLT_PIXEL Square[STATUS_SQUARE_SIDE * STATUS_SQUARE_SIDE];
    EFI_GRAPHICS_OUTPUT_BLT_PIXEL BackupTL[STATUS_SQUARE_SIDE * STATUS_SQUARE_SIDE];
    EFI_GRAPHICS_OUTPUT_BLT_PIXEL BackupTR[STATUS_SQUARE_SIDE * STATUS_SQUARE_SIDE];
    EFI_GRAPHICS_OUTPUT_BLT_PIXEL BackupBL[STATUS_SQUARE_SIDE * STATUS_SQUARE_SIDE];
    EFI_GRAPHICS_OUTPUT_BLT_PIXEL BackupBR[STATUS_SQUARE_SIDE * STATUS_SQUARE_SIDE];
    // set square color
    for (UINTN i = 0 ; i < STATUS_SQUARE_SIDE * STATUS_SQUARE_SIDE; i++) {
        Square[i].Blue = EfiGraphicsColors[Color].Blue;
        Square[i].Green = EfiGraphicsColors[Color].Green;
        Square[i].Red = EfiGraphicsColors[Color].Red;
        Square[i].Reserved = 0x00;
    }
    Width = Width - STATUS_SQUARE_SIDE -1;
```



```
    Height = Height - STATUS_SQUARE_SIDE -1;
    // backup current squares
    Gop->Blt(Gop, BackupTL, EfiBltVideoToBltBuffer, StartX, StartY, 0, 0, STATUS_SQUARE_SI
DE, STATUS_SQUARE_SIDE, 0);
    Gop->Blt(Gop, BackupTR, EfiBltVideoToBltBuffer, StartX + Width, StartY, 0, 0, STATUS_S
QUARE_SIDE, STATUS_SQUARE_SIDE, 0);
    Gop->Blt(Gop, BackupBL, EfiBltVideoToBltBuffer, StartX, StartY + Height, 0, 0, STATUS_
SQUARE_SIDE, STATUS_SQUARE_SIDE, 0);
    Gop->Blt(Gop, BackupBR, EfiBltVideoToBltBuffer, StartX + Width, StartY + Height, 0, 0,
 STATUS_SQUARE_SIDE, STATUS_SQUARE_SIDE, 0);
    // draw status square
    Gop->Blt(Gop, Square, EfiBltBufferToVideo, 0, 0, StartX, StartY, STATUS_SQUARE_SIDE, S
TATUS_SQUARE_SIDE, 0);
    Gop->Blt(Gop, Square, EfiBltBufferToVideo, 0, 0, StartX + Width, StartY, STATUS_SQUARE
_SIDE, STATUS_SQUARE_SIDE, 0);
    Gop->Blt(Gop, Square, EfiBltBufferToVideo, 0, 0, StartX, StartY + Height, STATUS_SQUAR
E_SIDE, STATUS_SQUARE_SIDE, 0);
    Gop->Blt(Gop, Square, EfiBltBufferToVideo, 0, 0, StartX + Width, StartY + Height, STAT
US_SQUARE_SIDE, STATUS_SQUARE_SIDE, 0);
    // wait 500ms
    gBS->Stall(500*1000);
    // restore squares from backups
    Gop->Blt(Gop, BackupTL, EfiBltBufferToVideo, 0, 0, StartX, StartY, STATUS_SQUARE_SIDE,
 STATUS_SQUARE_SIDE, 0);
    Gop->Blt(Gop, BackupTR, EfiBltBufferToVideo, 0, 0, StartX + Width, StartY, STATUS_SQUA
RE_SIDE, STATUS_SQUARE_SIDE, 0);
    Gop->Blt(Gop, BackupBL, EfiBltBufferToVideo, 0, 0, StartX, StartY + Height, STATUS_SQU
ARE_SIDE, STATUS_SQUARE_SIDE, 0);
    Gop->Blt(Gop, BackupBR, EfiBltBufferToVideo, 0, 0, StartX + Width, StartY + Height, ST
ATUS_SQUARE_SIDE, STATUS_SQUARE_SIDE, 0);
    return EFI_SUCCESS;
}
```

As you can see, it makes extensive use of the GOP (Graphic Output Protocol) *blt* (block transfer) functionality to copy data between the video framebuffer and allocated temporary memory using the *EfiBltVideoToBltBuffer* and *EfiBltBufferToVideo* parameters.

Here is the source code for the function that takes the actual screenshot:

```
EFI_STATUS
SnapShot( EFI_GRAPHICS_OUTPUT_PROTOCOL *Gop,
          UINTN                        StartX,
          UINTN                        StartY,
          UINTN                        Width,
          UINTN                        Height )
{
    EFI_GRAPHICS_OUTPUT_BLT_PIXEL *BltBuffer = NULL;
    EFI_STATUS Status = EFI_SUCCESS;
    UINTN      ImageSize;
    ImageSize = Width * Height;
    // allocate memory for snapshot
    BltBuffer = AllocateZeroPool( sizeof(EFI_GRAPHICS_OUTPUT_BLT_PIXEL) * ImageSize );
    if (BltBuffer == NULL) {
        Print(L"ERROR: BltBuffer. No memory resources\n");
        return EFI_OUT_OF_RESOURCES;
    }
    // take screenshot
    Status = Gop->Blt( Gop, BltBuffer, EfiBltVideoToBltBuffer, StartX, StartY, 0, 0, Width,
 Height, 0 );
    if (EFI_ERROR(Status)) {
        Print(L"ERROR: Gop->Blt [%d]\n", Status);
        FreePool( BltBuffer );
        return Status;
    }
```



```
    PrepareBMPFile( BltBuffer, Width, Height );
    return Status;
}
```

Again, note the use of GOP *blt* with the *EfiBltVideoToBltBuffer* parameter to capture the contents of the designated screen area as a 32-bit bitmap into *BltBuffer*.

The next task after capturing a bitmap of the designated screen area is to create the contents of the BMP file in memory before writting it out to disk. This is done in the *PrepareBMPFile* function as shown below:

```
EFI_STATUS
PrepareBMPFile( EFI_GRAPHICS_OUTPUT_BLT_PIXEL *BltBuffer,
                UINT32 Width,
                UINT32 Height )
{
    EFI_GRAPHICS_OUTPUT_BLT_PIXEL *Pixel;
    BMP_IMAGE_HEADER  *BmpHeader;
    EFI_STATUS        Status = EFI_SUCCESS;
    EFI_TIME          Time;
    CHAR16            FileName[40];
    UINT8             *FileData;
    UINTN             FileDataLength;
    UINT8             *ImagePtr;
    UINT8             *ImagePtrBase;
    UINTN             ImageLineOffset;
    UINTN             x, y;
    ImageLineOffset = Width * 3;
    if ((ImageLineOffset % 4) != 0) {
        ImageLineOffset = ImageLineOffset + (4 - (ImageLineOffset % 4));
    }
    // allocate buffer for data
    FileDataLength = sizeof(BMP_IMAGE_HEADER) + Height * ImageLineOffset;
    FileData = AllocateZeroPool( FileDataLength );
    if (FileData == NULL) {
        FreePool( BltBuffer );
        Print(L"ERROR: AllocateZeroPool. No memony resources\n");
        return EFI_OUT_OF_RESOURCES;
    }
    // fill header
    BmpHeader = (BMP_IMAGE_HEADER *)FileData;
    BmpHeader->CharB = 'B';
    BmpHeader->CharM = 'M';
    BmpHeader->Size = (UINT32)FileDataLength;
    BmpHeader->ImageOffset = sizeof(BMP_IMAGE_HEADER);
    BmpHeader->HeaderSize = 40;
    BmpHeader->PixelWidth = Width;
    BmpHeader->PixelHeight = Height;
    BmpHeader->Planes = 1;
    BmpHeader->BitPerPixel = 24;
    BmpHeader->CompressionType = 0;
    BmpHeader->XPixelsPerMeter = 0;
    BmpHeader->YPixelsPerMeter = 0;
    // fill pixel buffer
    ImagePtrBase = FileData + BmpHeader->ImageOffset;
    for (y = 0; y < Height; y++) {
        ImagePtr = ImagePtrBase;
        ImagePtrBase += ImageLineOffset;
        Pixel = BltBuffer + (Height - 1 - y) * Width;
        for (x = 0; x < Width; x++) {
            *ImagePtr++ = Pixel->Blue;
            *ImagePtr++ = Pixel->Green;
            *ImagePtr++ = Pixel->Red;
            Pixel++;
```



```
        }
    }
    FreePool(BltBuffer);
    Status = gRT->GetTime(&amp;Time, NULL);
    if (!EFI_ERROR(Status)) {
        UnicodeSPrint( FileName, 62, L"screenshot-%04d%02d%02d-%02d%02d%02d.bmp",
                       Time.Year, Time.Month, Time.Day, Time.Hour, Time.Minute, Time.Secon
d );
    } else {
        UnicodeSPrint( FileName, 30, L"screenshot.bmp" );
    }
    SaveImage( FileName, FileData, FileDataLength );
    FreePool( FileData );
    return Status;
}
```

Finally the in-memory copy of the BMP file is written out to disk in the current directory by the *SaveImage* function:

```

EFI_STATUS
SaveImage( CHAR16 *FileName,
           UINT8  *FileData,
           UINTN  FileDataLength )
{
    SHELL_FILE_HANDLE FileHandle = NULL;
    EFI_STATUS        Status = EFI_SUCCESS;
    CONST CHAR16      *CurDir = NULL;
    CONST CHAR16      *PathName = NULL;
    CHAR16            *FullPath = NULL;
    UINTN             Length = 0;
    CurDir = gEfiShellProtocol->GetCurDir(NULL);
    if (CurDir == NULL) {
        Print(L"ERROR: Cannot retrieve current directory\n");
    return EFI_NOT_FOUND;
    }
    PathName = CurDir;
    StrnCatGrow(&amp;FullPath, &amp;Length, PathName, 0);
    StrnCatGrow(&amp;FullPath, &amp;Length, L"\\", 0);
    StrnCatGrow(&amp;FullPath, &amp;Length, FileName, 0);
#ifdef DEBUG
    Print(L"FullPath: [%s]\n", FullPath);
#endif
    Status = gEfiShellProtocol->OpenFileByName( FullPath,
                                                &amp;FileHandle,
                                                EFI_FILE_MODE_READ | EFI_FILE_MODE_WRITE |
 EFI_FILE_MODE_CREATE);
    if (EFI_ERROR(Status)) {
        Print(L"ERROR: OpenFileByName [%s] [%d]\n", FullPath, Status);
        return Status;
    }
    // BufferSize = FileDataLength;
    Status = gEfiShellProtocol->WriteFile( FileHandle,
                                           &amp;FileDataLength,
                                           FileData );
    gEfiShellProtocol->CloseFile( FileHandle );
    if (EFI_ERROR(Status)) {
        Print(L"ERROR: Saving image to file: %x\n", Status);
    } else {
        Print(L"Successfully saved image to %s\n", FullPath);
    }
    return Status;
}
```

Could I have directly accessed the screen framebuffer instead of using the functionality provided

by GOP's *blt*? Absolutely. However I felt that using the GOP *blt* functionality was probably the more portable approach.

I also decided to see if I could convert *ScreenShot* into a UEFI driver so that a user could take multiple screenshots using a hotkey combination.

Here is the basic framework for the *ScreenshotDriver* UEFI driver:

```
#define SCREENSHOTDRIVER_VERSION 0x1
EFI_DRIVER_BINDING_PROTOCOL gScreenshotDriverBinding = {
    ScreenshotDriverBindingSupported,
    ScreenshotDriverBindingStart,
    ScreenshotDriverBindingStop,
    SCREENSHOTDRIVER_VERSION,
    NULL,
    NULL
};
GLOBAL_REMOVE_IF_UNREFERENCED
EFI_COMPONENT_NAME2_PROTOCOL gScreenshotDriverComponentName2 = {
    (EFI_COMPONENT_NAME2_GET_DRIVER_NAME) ScreenshotDriverComponentNameGetDriverName,
    (EFI_COMPONENT_NAME2_GET_CONTROLLER_NAME) ScreenshotDriverComponentNameGetControllerNa
me,
    "en"
};
GLOBAL_REMOVE_IF_UNREFERENCED
EFI_UNICODE_STRING_TABLE mScreenshotDriverNameTable[] = {
    { "en", (CHAR16 *) L"ScreenshotDriver" },
    { NULL , NULL }
};
GLOBAL_REMOVE_IF_UNREFERENCED
EFI_DRIVER_DIAGNOSTICS2_PROTOCOL gScreenshotDriverDiagnostics2 = {
    ScreenshotDriverRunDiagnostics,
    "en"
};
GLOBAL_REMOVE_IF_UNREFERENCED
EFI_DRIVER_CONFIGURATION2_PROTOCOL gScreenshotDriverConfiguration2 = {
    ScreenshotDriverConfigurationSetOptions,
    ScreenshotDriverConfigurationOptionsValid,
    ScreenshotDriverConfigurationForceDefaults,
    "en"
};
EFI_STATUS
ScreenshotDriverConfigurationSetOptions( EFI_DRIVER_CONFIGURATION2_PROTOCOL       *This,
                                         EFI_HANDLE                               Controll
erHandle,
                                         EFI_HANDLE                               ChildHan
dle,
                                         CHAR8                                    *Languag
e,
                                         EFI_DRIVER_CONFIGURATION_ACTION_REQUIRED *ActionR
equired )
{
    return EFI_UNSUPPORTED;
}
EFI_STATUS
ScreenshotDriverConfigurationOptionsValid( EFI_DRIVER_CONFIGURATION2_PROTOCOL *This,
                                           EFI_HANDLE                          ControllerHa
ndle,
                                           EFI_HANDLE                         ChildHandle )
{
    return EFI_UNSUPPORTED;
}
EFI_STATUS
ScreenshotDriverConfigurationForceDefaults( EFI_DRIVER_CONFIGURATION2_PROTOCOL       *This,
                                            EFI_HANDLE                                Contr
```



```
ollerHandle,
                                        EFI_HANDLE                              Child
Handle,
                                        UINT32                                  Defau
ltType,
                                        EFI_DRIVER_CONFIGURATION_ACTION_REQUIRED *Acti
onRequired )
{
    return EFI_UNSUPPORTED;
}
EFI_STATUS
ScreenshotDriverRunDiagnostics( EFI_DRIVER_DIAGNOSTICS2_PROTOCOL *This,
                                EFI_HANDLE                      ControllerHandle,
                                EFI_HANDLE                      ChildHandle,
                                EFI_DRIVER_DIAGNOSTIC_TYPE      DiagnosticType,
                                CHAR8                           *Language,
                                EFI_GUID                        **ErrorType,
                                UINTN                           *BufferSize,
                                CHAR16                          **Buffer )
{
    return EFI_UNSUPPORTED;
}
//
//  Retrieve user readable name of the driver
//
EFI_STATUS
ScreenshotDriverComponentNameGetDriverName( EFI_COMPONENT_NAME2_PROTOCOL *This,
                                        CHAR8                   *Language,
                                        CHAR16                  **DriverName )
{
    return LookupUnicodeString2( Language,
                                 This->SupportedLanguages,
                                 mScreenshotDriverNameTable,
                                 DriverName,
                                 (BOOLEAN)(This == &amp;amp;gScreenshotDriverComponentName2) )
;
}
//
//  Retrieve user readable name of controller being managed by a driver
//
EFI_STATUS
ScreenshotDriverComponentNameGetControllerName( EFI_COMPONENT_NAME2_PROTOCOL *This,
                                            EFI_HANDLE                  ControllerHan
dle,
                                            EFI_HANDLE                  ChildHandle,
                                            CHAR8                       *Language,
                                            CHAR16                      **ControllerN
ame )
{
    return EFI_UNSUPPORTED;
}
//
//  Start this driver on Controller
//
EFI_STATUS
ScreenshotDriverBindingStart( EFI_DRIVER_BINDING_PROTOCOL *This,
                              EFI_HANDLE                  Controller,
                              EFI_DEVICE_PATH_PROTOCOL    *RemainingDevicePath )
{
    return EFI_UNSUPPORTED;
}
//
//  Stop this driver on ControllerHandle.
//
EFI_STATUS
ScreenshotDriverBindingStop( EFI_DRIVER_BINDING_PROTOCOL *This,
                             EFI_HANDLE                  Controller,
                             UINTN                       NumberOfChildren,
```



```
                              EFI_HANDLE                 *ChildHandleBuffer )
{
    return EFI_UNSUPPORTED;
}
//
//  See if this driver supports ControllerHandle.
//
EFI_STATUS
ScreenshotDriverBindingSupported( EFI_DRIVER_BINDING_PROTOCOL *This,
                                  EFI_HANDLE                  Controller,
                                  EFI_DEVICE_PATH_PROTOCOL    *RemainingDevicePath )
{
    return EFI_UNSUPPORTED;
}
```

Note that I only used version 2 of the *ComponentName, Diagnostics* and *Configuration* driver options.

Here is the source code for the driver entry point:

```

FI_STATUS
EFIAPI
ScreenshotDriverEntryPoint( EFI_HANDLE ImageHandle,
                            EFI_SYSTEM_TABLE *SystemTable )
{
    EFI_GUID gEfiSimpleTextInputExProtocolGuid = EFI_SIMPLE_TEXT_INPUT_EX_PROTOCOL_GUID;
    EFI_STATUS                       Status;
    EFI_KEY_DATA                     SimpleTextInExKeyStroke;
    EFI_SIMPLE_TEXT_INPUT_EX_PROTOCOL *SimpleTextInEx;
    // install driver model protocol(s).
    Status = EfiLibInstallAllDriverProtocols2( ImageHandle,
                                               SystemTable,
                                               &amp;gScreenshotDriverBinding,
                                               ImageHandle,
                                               NULL,
                                               &amp;gScreenshotDriverComponentName2,
                                               NULL,
                                               &amp;gScreenshotDriverConfiguration2,
                                               NULL,
                                               &amp;gScreenshotDriverDiagnostics2 );
    if (EFI_ERROR(Status)) {
        DEBUG((DEBUG_ERROR, "EfiLibInstallDriverBindingComponentName2 [%d]\n", Status));
        return Status;
    }
    // set key combination to be LEFTCTRL+LEFTALT+F12
    SimpleTextInExKeyStroke.Key.ScanCode = SCAN_F12;
    SimpleTextInExKeyStroke.Key.UnicodeChar = 0;
    SimpleTextInExKeyStroke.KeyState.KeyShiftState = EFI_SHIFT_STATE_VALID | EFI_LEFT_CONT
ROL_PRESSED | EFI_LEFT_ALT_PRESSED;
    SimpleTextInExKeyStroke.KeyState.KeyToggleState = 0;
    Status = gBS->HandleProtocol( gST->ConsoleInHandle,
                                  &amp;gEfiSimpleTextInputExProtocolGuid,
                                  (VOID **) &amp;SimpleTextInEx );
    if (EFI_ERROR (Status)) {
        DEBUG((DEBUG_ERROR, "SimpleTextInputEx handle not found via HandleProtocol\n"));
        return Status;
    }
    // register key notification function
    Status = SimpleTextInEx->RegisterKeyNotify( SimpleTextInEx,
                                                &amp;SimpleTextInExKeyStroke,
                                                TakeScreenShot,
                                                &amp;SimpleTextInExHandle );
    if (EFI_ERROR (Status)) {
        DEBUG((DEBUG_ERROR, "SimpleTextInEx->RegisterKeyNotify returned %d\n", Status));
```



```
        return Status;
    }
    return EFI_SUCCESS;
}
```

From the above code you can see that the hotkey combination to take a screenshot is *LEFTCTRL+LEFTALT+F12*. Note that the order of passing parameters to *EfiLibInstallAllDriverProtocols2* is critical; *NULL* indicates that a specific driver protocol is not supported.

Here is the code that unloads the *ScreenshotDriver*:

```
EFI_STATUS
EFIAPI
ScreenshotDriverUnload( EFI_HANDLE ImageHandle )
{
    EFI_GUID gEfiSimpleTextInputExProtocolGuid = EFI_SIMPLE_TEXT_INPUT_EX_PROTOCOL_GUID;
    EFI_SIMPLE_TEXT_INPUT_EX_PROTOCOL *SimpleTextInEx;
    EFI_STATUS Status = EFI_SUCCESS;
    UINTN      Index;
    UINTN      HandleCount = 0;
    EFI_HANDLE *HandleBuffer = NULL;
    Status = gBS->HandleProtocol( gST->ConsoleInHandle,
                                  &amp;gEfiSimpleTextInputExProtocolGuid,
                                  (VOID **) &amp;SimpleTextInEx );
    if (EFI_ERROR (Status)) {
        DEBUG((DEBUG_ERROR, "SimpleTextInputEx handle not found via HandleProtocol\n"));
        return Status;
    }
    // unregister key notification function
    Status = SimpleTextInEx->UnregisterKeyNotify( SimpleTextInEx,
                                                  SimpleTextInExHandle );
    if (EFI_ERROR (Status)) {
        DEBUG((DEBUG_ERROR, "SimpleTextInEx->UnregisterKeyNotify returned %d\n", Status));
        return Status;
    }
    // get list of all the handles in the handle database.
    Status = gBS->LocateHandleBuffer( AllHandles,
                                      NULL,
                                      NULL,
                                      &amp;HandleCount,
                                      &amp;HandleBuffer );
    if (EFI_ERROR (Status)) {
        DEBUG((DEBUG_ERROR, "LocateHandleBuffer [%d]\n", Status));
        return Status;
    }
    for (Index = 0; Index < HandleCount; Index++) {
        Status = gBS->DisconnectController( HandleBuffer[Index],
                                            ImageHandle,
                                            NULL );
    }
    if (HandleBuffer != NULL) {
        FreePool( HandleBuffer );
    }
    // uninstall protocols installed in the driver entry point
    Status = gBS->UninstallMultipleProtocolInterface( ImageHandle,
                                                      &amp;gEfiDriverBindingProtocolGuid,
 &amp;gScreenshotDriverBinding,
                                                      &amp;gEfiComponentName2ProtocolGuid,
 &amp;gScreenshotDriverComponentName2,
                                                      &amp;gEfiDriverConfiguration2Protoc
olGuid, gScreenshotDriverConfiguration2,
                                                      &amp;gEfiDriverDiagnostics2Protocol
Guid, &amp;gScreenshotDriverDiagnostics2,
```



```
                                                    NULL );
    if (EFI_ERROR (Status)) {
        DEBUG((DEBUG_ERROR, "UninstallMultipleProtocolInterfaces returned %d\n", Status));
        return Status;
    }
    return EFI_SUCCESS;
}
```

Note that the order in which you pass the protocol parameter pairs to *UninstallMultipleProtocolInterface* is also critical. It must be *Binding*, followed by *ComponentName*, followed by *Configuration*, followed by *Diagnostics*.

Here is the *INF* for this driver:

```
[Defines]
  INF_VERSION                    = 1.25
  BASE_NAME                      = ScreenshotDriver
  FILE_GUID                      = 4ea87c57-7795-5ded-1055-747010f3ce51
  MODULE_TYPE                    = UEFI_DRIVER
  VERSION_STRING                 = 0.9
  ENTRY_POINT                    = ScreenshotDriverEntryPoint
  UNLOAD_IMAGE                   = ScreenshotDriverUnload
  VALID_ARCHITECTURES            = X64

[Packages]
  MdePkg/MdePkg.dec
  MdeModulePkg/MdeModulePkg.dec

[LibraryClasses]
  UefiBootServicesTableLib
  UefiRuntimeServicesTableLib
  UefiDriverEntryPoint
  DebugLib
  PrintLib
  ShellLib
  BaseLib
  BaseMemoryLib
  UefiLib

[Sources]
  ScreenshotDriver.c
  ScreenshotDriver.h

[Packages]
  MdePkg/MdePkg.dec
  ShellPkg/ShellPkg.dec

[Depex]
  gEfiGraphicsOutputProtocolGuid AND
  gEfiSimpleTextInputExProtocolGuid

[Pcd]
```

Note that the module type is defined as *UEFI_DRIVER*, the *ENTRY_POINT* and *UNLOAD_IMAGE* keywords are mandatory as is the *Depex* section.

Here is a BMP file produced by *ScreenShot* when fullscreen (the default) was specified:

Here is a BMP file produced by *ScreenshotDriver* when loaded and the LEFTCTRL+LEFTALT+F12 hotkey combination pressed:

Here is a BMP file produced by *ScreenShot* when a specific rectangular part of the screen was specified:

I plan to enhance the functionality of these two utilities over time, especially better driver support. One feature I would like to add is the ability to generate a compressed BMP file. Another feature I am considering is support for generating a JPEG-encoded file. If you have other feature requests, please let me know.

Meanwhile, enjoy!

