Apploader

Contents

Gamecube apploader reverse engineering for CubeDocumented project

This reversing is not pretending to be compilable, and can be used for education purposes only. I dont hold any responsibility for damage of your Gamecube clock settings or memory card saves or else, produced from using code/information represented in this document.

Documented by Andrei Chestakov (org). Version 1.1. (26 Apr 2005) Additions and corrections by tmbinc.

Overview

Complete Luigi's Apploader. Reverse work by org <kvzorganic_at_mail_dot_ru> Other games seems to contain the same code with minor changes due to revision, so you may think, that apploader is the same for all games. Except Datel's FreeLoader and ActionReplay DVDs, which are using custom apploader binary (simplified Nintendo's apploader).

Apploader is a small program layer between GC bootrom and main application. Since main application's executable binary format may vary, apploader is needed to boot executable in main memory, using bootrom's low-level DVD read routines. In practice, all known game apploaders are using Nintendo's DOL format for executable files.

Format of apploader binary is simple. First goes apploader header, and then actual apploader code.

Apploader header

```
typedef struct
{
   char    revision[16];       // Revision compile date (example "2001/12/17")
   u32     entryPoint;         // Apploader's entrypoint
   s32     size;               // Size of apploader code
   s32     trailerSize;        // Size of safe trailer
```

```
    u8      padding[4];         // zeroes
} LDRImage;
```

Full apploader size is sum of size and trailerSize, rounded up to 32 bytes. Trailer is need for future apploader versions.

Apploader is located at 0x2440 offset on DVD and loaded by bootrom at 0x81200000 address. Then bootrom jumps to apploader's entrypoint:

```
void AplEntry(
   void    (**Init)(void (*OSReportCallback)(char *msg, ...)),
   BOOL    (**Main)(void **addr, s32 *length, s32 *offset),
   void*   (**Close)())
{
   // Return location of apploader's Init, Main and Close routines.
   *Init  = AplInit;
   *Main  = AplMain;
   *Close = AplClose;
}
```

After executing of entrypoint, bootrom will call in order: apploader's Init, then Main, then Close. Main is called in loop, until it will not return 0.

PART I. Apploader Init

Init call is clearing apploader's internal data, and saving there callback of OSReport call (which is located inside IPL). OSReport is used to output some debug messages, during apploader runtime.

Apploader's internal data:

```
   // Some DVD transfers require multiple times to load.
   BOOL        SecondTimeForThePart;
   void*       SecondTimeAddr;
   s32         SecondTimeLength;
   s32         SecondTimeOffset;
   int         SecondTimePart;

   DVDBB2      bb2;        // DVD boot block
   DolImage    DolImage;   // main DOL executable header

   int         Section;    // DOL section
   int         Part;       // Apploader's step
   u32         +0x148      // ?? only cleared in Init

   // OSReport routine itself is placed somewhere in IPL
   void        (*OSReport)(char *msg, ...);

#pragma pack(32)
   // First 32 bytes of DVDBI2 will be loaded here. This is a bit hacky.
   // We dont use whole DVDBI2 (its too big), to save some memory.
   // 32 bytes - because its minimum DVD DMA transfer size (?)
   s32         debugMonSize;
   s32         simMemSize;
   u32         argOffset;
   u32         debugFlag;
   int         FSTHigh;    // 1, if FST is resided on DVD above DOL
   u8          dummy[12];  // ending bytes
#pragma pack(0)

void AplInit(void (*OSReportCallback)(char *msg, ...))
{
   // Clear some structures
   memset(&bb2, 0, sizeof(DVDBB2));
   memset(&DolImage, 0, sizeof(DolImage));

   Section = 0;
   Part = 0;
   [+0x148] = 0;

   OSReport = OSReportCallback;    // Save report callback

   // Print apploader version info
   OSReport("Apploader Initialized.  $ Revision: 28 $\n");
   OSReport("This Apploader built %s %s\n", __DATE__, __TIME__);
}
```

PART II. Apploader Main

Main is called in loop by IPL, until all data not loaded. IPL is using Main output parameters, to load data from DVD. IPL keeps loading new data, until Main will not return 0, signaling that all data loaded. Just read through comments to understand code.

```
BOOL AplMain(void **addr, s32 *length, s32 *offset)
{
   OSLoMem *LoMem = OSPhysicalToCached(0x0000);

   switch(Part)
   {
       // Load BB2 structure.
       case 0:
       case 1:
           *addr = &bb2;
           *length = sizeof(DVDBB2);
           *offset = DVD_BB2_OFFSET;

           Part = 2;
           DCInvalidateRange(*addr, *length);
           break;

       // Check FST length and load first 32 bytes of BI2 structure.
       case 2:
           u32 FSTLength = bb2.FSTLength;
           u32 FSTMaxLength = bb2.FSTMaxLength;

           if(FSTLength > FSTMaxLength)
           {
               OSReport(
                   "APPLOADER ERROR >>> FSTLength(%d) in BB2 is greater "
                   "than FSTMaxLength(%d)\n", FSTLength, FSTMaxLength );
               PPCHalt();
           }

           *addr   = &debugMonSize;
           *length = 32;
           *offset = DVD_BI2_OFFSET;

           Part = 3;
           DCInvalidateRange(*addr, *length);
           break;

       // Setup some OS lomem variables by BI2 values, and check memory configuration.
       // Copy DVD BI2 in main memory.
       case 3:
           __OSDebugMonSize = debugMonSize;
           __OSDebugMon = OSRoundDown32B(__OSPhysMemSize - __OSDebugMonSize);
           __OSSimMemSize   = simMemSize;

           // Check debug monitor size alignment
           if(__OSDebugMonSize & 31)
           {
               OSReport(
                   "APPLOADER ERROR >>> Debug monitor size (%d) should "
                   "be a multiple of 32\n", __OSDebugMonSize );
               PPCHalt();
           }

           // Check console simulated memory size alignment
           if(__OSSimMemSize & 31)
           {
               OSReport(
                   "APPLOADER ERROR >>> simulated memory size (%d) "
                   "should be a multiple of 32\n", __OSSimMemSize );
               PPCHalt();
           }

           if(__OSSimMemSize == 0)
           {
               __OSSimMemSize = __OSPhysMemSize;
           }

           if(__OSSimMemSize < __OSPhysMemSize)
           {
               if(__OSDebugMonSize >= (__OSPhysMemSize - __OSSimMemSize))
```

```
            {
                OSReport(
                    "APPLOADER ERROR >>> [Physical memory size(0x%x)] "
                    "- [Console simulated memory size(0x%x)]\n"
                    "APPLOADER ERROR >>> must be greater than debug
                    "monitor size(0x%x)\n",
                    __OSPhysMemSize, __OSSimMemSize, __OSDebugMonSize );
                PPCHalt();
            }

            // Move down FST address
            bb2.FSTAddress = OSRoundDown32B(__OSSimMemSize - bb2.FSTMaxSize);
        }
        else
        {
            if(__OSSimMemSize == __OSPhysMemSize)
            {
                bb2.FSTAddress = OSRoundDown32B(__OSDebugMon - bb2.FSTMaxSize);
            }
            else
            {
                OSReport(
                    "APPLOADER ERROR >>> Physical memory size is 0x%x bytes.\n"
                    "APPLOADER ERROR >>> Console simulated memory size "
                    "must be smaller than or equal to the Physical memory size\n",
                    __OSPhysMemSize, __OSSimMemSize );
                PPCHalt();
            }
        }

        // Load whole BI2 in main memory. Reside it before FST.
        __OSBootInfo2 = bb2.FSTAddress - OS_BI2_SIZE;

        *addr   = __OSBootInfo2;
        *length = OS_BI2_SIZE;
        *offset = DVD_BI2_OFFSET;

        Part = 4;
        DCInvalidateRange(*addr, *length);
        break;

    // Load FST, only if FST placed before boot file.
    case 4:
        FSTHigh = bb2.bootFilePosition < bb2.FSTPosition;
        if(FSTHigh)
        {
            // Continue on part 5. Load FST later.
            Part = 5;
        }
        else
        {
            *addr   = bb2.FSTAddress;
            *length = OSRoundUp32B(bb2.FSTLength);
            *offset = bb2.FSTPosition;

            Part = 5;

            if(*addr > 0x81700000)
            {
                OSReport(
                    "APPLOADER ERROR >>> Illegal FST "
                    "destination address! (0x%x)\n", *addr );
                PPCHalt();
            }

            DCInvalidateRange(*addr, *length);
            LoadBigData(addr, length, offset, &Part);
            break;
        }

    // Load main DOL header.
    case 5:
        *addr   = &DolImage;
        *length = sizeof(DolImage);
        *offset = bb2.bootFilePosition;

        Part = 6;
        DCInvalidateRange(*addr, *length);
        break;

    // Check out DOL properties. Clear BSS.
    case 6:
        s32 dolSize = DOLSize();

        // Halt apploader, if DOL size exceeds limit.
        if( (dolSize > __OSBootInfo2->dolLimit) && __OSBootInfo2->dolLimit)
        {
            OSReport(
                "APPLOADER ERROR >>> Total size of text/data sections "
                "of the dol file are too big (%d(0x%08x) bytes). "
                "Currently the limit is set as %d(0x%08x) bytes\n",
                dolSize, __OSBootInfo2->dolLimit );
            PPCHalt();
        }

        // Save DOL size in OS globals
        if(FSTHigh) __OSDOLSize = dolSize + OSRoundUp32B(bb2.FSTLength);
        else __OSDOLSize = dolSize;

        // Address limit?
        #define DOL_ADDRESS_LIMIT   0x80700000  // production boards
        #define DOL_ADDRESS_LIMITD  0x81200000  // development boards
        if(!(OSGetConsoleType() & OS_CONSOLE_DEVELOPMENT))
        {
            if(AddressLimit(DOL_ADDRESS_LIMIT) == FALSE)
            {
                OSReport(
                    "APPLOADER ERROR >>> One of the sections in the dol file "
                    "exceeded its boundary. All the sections should not exceed "
                    "0x%08x (production mode).\n", DOL_ADDRESS_LIMIT );
                PPCHalt();
            }
        }
        else
        {
            // Check twice : first for production mode, then for development.

            if(AddressLimit(DOL_ADDRESS_LIMIT) == FALSE)
            {
                OSReport(
                    "APPLOADER WARNING >>> One of the sections in the dol file "
                    "exceeded its boundary. All the sections should not exceed "
                    "0x%08x in production mode.\n", DOL_ADDRESS_LIMIT );
                // Do not halt apploader (show only warning).
            }

            if(AddressLimit(DOL_ADDRESS_LIMITD) == FALSE)
            {
                OSReport(
                    "APPLOADER ERROR >>> One of the sections in the dol file "
                    "exceeded its boundary. All the sections should not exceed "
                    "0x%08x (development mode).\n", DOL_ADDRESS_LIMITD );
                PPCHalt();
            }
        }

        // Clear BSS
        memset(DolImage.bss, 0, DolImage.bssLen);
        DCFlushRange(DolImage.bss, DolImage.bssLen);

        Part = 7;

    // Load text sections.
    case 7:
        if(Section<DOL_MAX_TEXT)
        {
            if(DolImage.textData[Section])
            {
                *addr   = DolImage.text[Section];
                *length = OSRoundUp32B(DolImage.textLen[Section]);
                *offset = bb2.bootFilePosition + DolImage.textData[Section];

                if( (*addr + *length) > 0x81200000 )
                {
                    OSReport(
                        "APPLOADER ERROR >>> Too big text segment! (0x%x - 0x%x)\n",
                        *addr, *addr + *length );
                    PPCHalt();
                }

                Section++;
                DCInvalidateRange(*addr, *length);
                break;
            }
        }
        else
        {
            Section = 0;
            Part = 8;
        }

    // Load data sections.
    case 8:
        if(Section<DOL_MAX_DATA)
        {
            if(DolImage.dataData[Section])
            {
                *addr   = DolImage.data[Section];
                *length = OSRoundUp32B(DolImage.dataLen[Section]);
                *offset = bb2.bootFilePosition + DolImage.dataData[Section];

                if( (*addr + *length) > 0x81200000 )
                {
                    OSReport(
                        "APPLOADER ERROR >>> Too big data segment! (0x%x - 0x%x)\n",
                        *addr, *addr + *length );
                    PPCHalt();
                }

                Section++;
                DCInvalidateRange(*addr, *length);
                break;
            }
        }
        else
        {
            Section = 0;
            Part = 9;
        }

    // Load FST, only if FST placed after boot file.
    case 9:
        if(FSTHigh)
        {
            *addr   = bb2.FSTAddress;
            *length = OSRoundUp32B(bb2.FSTLength);
            *offset = bb2.FSTPosition;

            Part = 10;

            if(*addr > 0x81700000)
            {
                OSReport(
                    "APPLOADER ERROR >>> Illegal FST "
                    "destination address! (0x%x)\n", *addr );
                PPCHalt();
            }

            DCInvalidateRange(*addr, *length);
            LoadBigData(addr, length, offset, &Part);
            break;
        }
        else
        {
            // Continue on part 10.
            Part = 10;
        }

    // Init misc. low memory variables. Continue on part 11.
    case 10:
        InitLoMem(LoMem);
        Part = 11;

    // Read button state of 4th controller and save it in OS global.
    // It's for testing pre-master dvds (NR Discs). But as NR discs gets
    // written without change to the dvdroms the retail apploaders contains
    // it, too. If you plug in a special device (dunno what it is exactly -
    // controller with special id? some bit set?) the apploader starts making
    // a crc of the whole disc and compare that to a crc stored on memory card.
    case 11:
        PADStatus padst;
        PadRead(PAD_CHAN3, &padst);
        __OSPADButton = padst.button;

        // All data loaded. End of apploader.
        return FALSE;

    // This part will be called, if DVD need to load more, than 64 sectors.
    case 12:
        ASSERT(SecondTimeForThePart == TRUE);
        LoadBigData(addr, length, offset, &Part);
        break;

    // Unknown part. Abort data loading.
    default:
        return FALSE;
}
```

```
   // We need to load more data. Tell IPL to do that.
   return TRUE;
}
```

PART III. Apploader Close

Close call is executed after apploader's Main.

```
void* AplClose(void)
{
   // Return entrypoint of main DOL executable to IPL
   return DolImage.entry;
}
```

Now IPL jumps to DOL's entrypoint (__start).

Appendix A: Helper calls

Calculate full size of DOL text and data sections

```
s32 DOLSize(void)
{
   DolImage *dol = &DolImage;
   s32 totalBytes = 0, i;

   for(i=0; i<DOL_MAX_TEXT; i++)
   {
       if(dol->textData[i])
       {
           // Aligned to 32 byte boundary
           totalBytes += OSRoundUp32B(dol->textLen[i]);
       }
   }

   for(i=0; i<DOL_MAX_DATA; i++)
   {
       if(dol->dataData[i])
       {
           // Aligned to 32 byte boundary
           totalBytes += OSRoundUp32B(dol->dataLen[i]);
       }
   }

   return totalBytes;
}
```

Return FALSE, if any DOL section, including BSS exceed address limit

```
BOOL AddressLimit(u32 limit)
{
   DolImage *dol = &DolImage;
   s32 i, end;

   // Text sections
   for(i=0; i<DOL_MAX_TEXT; i++)
   {
       if(dol->textData[i])
       {
           end = dol->text[i] + dol->textLen[i];

           if( ((dol->text[i] < 0x81100000) || (end > 0x81300000)) && (end > limit) )
           {
               return FALSE;
           }
       }
   }

   // Data sections
   for(i=0; i<DOL_MAX_DATA; i++)
   {
       if(dol->dataData[i])
       {
           end = dol->data[i] + dol->dataLen[i];

           if( ((dol->data[i] < 0x81100000) || (end > 0x81300000)) && (end > limit) )
           {
               return FALSE;
           }
       }
   }

   // BSS
   end = dol->bss + dol->bssLen;
   if( ((dol->bss < 0x81100000) || (end > 0x81300000)) && (end > limit) )
   {
       return FALSE;
   }

   return TRUE;
}
```

Some big transfers requires multiple load times for single apploader part (Limit is set to 64 DVD sectors - WHY?).

```
void LoadBigData(void **addr, s32 *length, s32 *offset, int *part)
{
    if(SecondTimeForThePart)
    {
        SecondTimeForThePart = FALSE;

        *addr   = SecondTimeAddr;
        *length = SecondTimeLength;
        *offset = SecondTimeOffset;
        *part   = SecondTimePart;
    }
    else
    {
        if(*length > 64*2048)       // 64 DVD sectors
        {
            // Initiate multiple DVD load.
            SecondTimeForThePart = TRUE;

            SecondTimeAddr = *addr + 64*2048;
            SecondTimeLength = *length - 64*2048;
            SecondTimeOffset = *offset + 64*2048;
            SecondTimePart = *part;

            *length = 64*2048;
            *part = 12;
        }
    }
}
```

Init some OS low memory variables

```
void InitLoMem(OSLoMem *LoMem)
{
    LoMem->bi.magic = OS_BOOTINFO_MAGIC;    // 0xD15EA5E
    LoMem->bi.version = 1;
```

```
    // Clear debug interface (db/DBInterface.h)
    LoMem->db.bPresent =
    LoMem->db.exceptionMask =
    LoMem->db.ExceptionDestination =
    LoMem->db.exceptionReturn = NULL;
}
```

Read PAD input buffer (buttons state). If PAD is not polled, return zeroes

```
void PadRead(int port, u32 *buf)
{
    // If controller polling enabled
    if(SI_POLL & (0x80 >> port))
    {
        // Read SI input buffer
        buf[0] = SI_CHAN_INBUFH(port);
        buf[1] = SI_CHAN_INBUFL(port);
    }
    else
    {
        buf[0] = buf[1] = 0;
    }
}
```

Appendix B: DVD structures

```
#define DVD_BB2_OFFSET      0x420
#define DVD_BI2_OFFSET      0x440

#define OS_BI2_SIZE         0x2000  // Size of BI2 structure

// DVD Boot Block
typedef struct
{
   u32     bootFilePosition;       // offset of main DOL executable
   u32     FSTPosition;            // offset of primary FST
   u32     FSTLength;              // primary FST size (in bytes)
   u32     FSTMaxLength;           // size of biggest additional FST
   u32     FSTAddress;             // FST address in main memory
   u32     userPosition;
   u32     userLength;
   u8      padding0[4];            // reserved, should be 0
} DVDBB2;
```

FST is File String Table. User area on DVD is where all files are placed. On some DVDs, user area is configured wrong. To know correct user area, you should parse whole FST and find min and max file offsets.

```
// DVD Boot Info
typedef struct
{
   s32     debugMonSize;           // size of debug monitor
   s32     simMemSize;             // simulated memory size (in bytes)
   u32     argOffset;              // command line arguments
   u32     debugFlag;              // debug present (set to 3, if CodeWarrior on GDEV)
   u32     TRKLocation;            // target resident kernel location
   s32     TRKSize;                // size of TRK
   u32     countryCode;            // country code
   u32     ?       +1C
   u32     ?       +20
   u32     ?       +24
   u32     dolLimit;               // maximum total size of DOL text/data sections (0 - unlimited)
   u32     ?       +2C

   u8      padded0[OS_BI2_SIZE - 0x30]; // reserved, should be 0
} DVDBI2;

Somewhere in +1C, +20 or +24 must present :
u32     longFileNameSupport;    // 1: use long file names, 0: use 8.3 names
u32     padSpec;                // controller version for PADSetSpec
```

Appendix C: Used OS low memory variables

```
s32     __OSPhysMemSize     AT_ADDRESS(OS_BASE_CACHED | 0x0028);
s32     __OSDebugMonSize    AT_ADDRESS(OS_BASE_CACHED | 0x00E8);
void*   __OSDebugMon        AT_ADDRESS(OS_BASE_CACHED | 0x00EC);
s32     __OSSimMemSize      AT_ADDRESS(OS_BASE_CACHED | 0x00F0);
DVDBI2* __OSBootInfo2       AT_ADDRESS(OS_BASE_CACHED | 0x00F4);
s32     __OSDOLSize         AT_ADDRESS(OS_BASE_CACHED | 0x30D4);
u16     __OSPADButton       AT_ADDRESS(OS_BASE_CACHED | 0x30E4);
```

Strange Things

What's [+0x148] in apploader data? Its only cleared in Init, but not used.

Why FST loading is so strange : if FST placed above DOL, then its loaded after DOL, otherwise its loaded before DOL.

```
if(FSTHigh) __OSDOLSize = dolSize + OSRoundUp32B(bb2.FSTLength);
else __OSDOLSize = dolSize;
```

Why so?

Why FST is loaded in pieces, if its greater than 64 DVD sectors, but big DOL sections are not loaded in same way?

Some code is placed after apploader data. Is it garbage or what?

ToDo

Need more BI2 details.

Get Zelda apploader and crosscheck it with current reversing. Maybe some interesting things will be discovered.

Count how many apploader revisions are present today.

Source

This information was found at: [1] (http://dolwin.emulation64.com/docs/Apploader.txt)