Breaking Down the Disney Infinity Base

April 3, 2023

After the success of my first open source project contribution, I decided I wanted to keep it going and reverse engineer another USB gaming peripheral - the Disney Infinity Base. There was minimal open research going around online, so this was going to be a much harder challenge to overcome.

The Toys

The Disney Infinity toys came in 3 different flavours, the figures from different Disney franchises that you play as in-game, hexagonal Play Sets that unlocked levels to play, and Power Discs which either unlocked abilities for characters, or extra items that you could use when creating levels in the 'Toy Box' mode. All of the toys acted as keys, which let players access content in game, and all contained 320 byte NFC tags that stored some toy information and progression data.

There are only 5 blocks (16 bytes) which the game uses during play, which is block 0, the block where an NFC tag's UID is stored along with other manufacturer data, block 1 which contains the information about the particular toy, and then block 4, 8 and 12 each hold the toy's progression data and/or ownership data. The data is stored on the figure encrypted, and the NFC chips themselves are read/write protected by a password. This password, however, has been discovered and is an SHA1 digest of a hard coded byte prefix, the UID (7 bytes) and then another hard coded byte postfix.

After being able to read the data, the data then has to be decrypted further in order to obtain the relevant information needed to generate blank toys. This process also requires some well known decryption algorithms, firstly an SHA1 digest of hard coded bytes and the UID, then from this the first 16 bytes (out of 20) are used as a 128 bit AES key (with every 4 bytes reversed due to endianness), which can then be used to decrypt the blocks on the figure.

Following the above steps allows you to get a breakdown on some of the toy's data, and importantly you can understand what some of the data on the figure means. In block 1, the data is structured as follows:

```
Name                              :  Type   #   Extra     Series Num.    CRC32

The Incredibles - Mr. Incredible : 00 0F 42 41 0D 08 14 00 00 01 D1 1F E4 1A BC B8
Starter Play Set                 : 00 1E 84 81 0D 03 1A 00 00 01 D1 1F 7E 09 EF 97
Pieces of Eight - Ability        : 00 2D C6 CE 11 04 16 00 00 01 D1 1F CD 94 03 50
Mike's New Car - Toy (Vehicle)   : 00 3D 09 17 12 0A 15 00 00 01 D1 1F 0A 56 6B C5
```

The first 4 bytes of block one are the figure's model number, as hex. The model numbers can be found here, but characters are any number between 1000000 and 1999999, play sets are between 2000000 and 2999999, circular power discs are between 3000000 and 3999999, and hexagon power discs/items are between 4000000 and 4999999.

The next set of 4 bytes indicates the manufacture date of the particular toy, formatted as YY/MM/DD. The starter set figures I own have this set to 0D 04 1D, which is the 19th of April 2013. The most recent figures I own were manufactured in 2017, about the time when Disney Infinity was discontinued. Luckily for emulation, we can set this value to be the same for every toy, so in Dolphin I have it set to the day Disney Infinity 1.0 was released (18th of August 2013, or 0D 08 12 as YY MM DD hex)

The next 4 bytes are the series number, all series 1 toys have 00 01 D1 1F for this, and series 2 is 00 02 D1 1F, series 3 is 00 03 D1 1F. The last 4 bytes is a CRC32 checksum, calculated from the previous 12 bytes.

The Base

The Infinity Base is a USB Peripheral used in the Disney Infinity Series of games. Only the console versions used the Base, and the PC version had the same content but instead of having to buy the individual toys to unlock it, you could purchase the content as DLC. The Wii/WiiU/PS3/PS4 versions all had a Base that was interchangeable between all games and consoles, but the Xbox 360/One versions could only be used on the console the Base was purchased for. The device is defined in Dolphin using the following constructor:

```
InfinityUSB::InfinityUSB(Kernel& ios, const std::string& device_name) : m_ios(ios)
{
  m_vid = 0x0E6F;
  m_pid = 0x0129;
  m_id = (u64(m_vid) << 32 | u64(m_pid) << 16 | u64(9) << 8 | u64(1));
  m_device_descriptor = DeviceDescriptor{0x12,   0x1,    0x200, 0x0, 0x0, 0x0, 0x20,
                                         0x0E6F, 0x0129, 0x200, 0x1, 0x2, 0x3, 0x1};
  m_config_descriptor.emplace_back(ConfigDescriptor{0x9, 0x2, 0x29, 0x1, 0x1, 0x0, 0x80, 0xFA});
  m_interface_descriptor.emplace_back(
      InterfaceDescriptor{0x9, 0x4, 0x0, 0x0, 0x2, 0x3, 0x0, 0x0, 0x0});
  m_endpoint_descriptor.emplace_back(EndpointDescriptor{0x7, 0x5, 0x81, 0x3, 0x20, 0x1});
  m_endpoint_descriptor.emplace_back(EndpointDescriptor{0x7, 0x5, 0x1, 0x3, 0x20, 0x1});
}
```

The base has 2 Interrupt Endpoints - one IN and one OUT. The max packet size is 32 bytes, and all requests and responses sent are padded to 32 bytes. The game sends requests to the IN endpoint, and the Base responds via the OUT endpoint.

Transfer Commands

The game always sends a command packet with optional data attached, and the power base responds with another packet (with optional data attached). Commands beginning with 0x00, 0xAA and 0xAB are used only for the OUT endpoint, whereas commands beginning 0xFF are for the IN endpoint. The structure of command packets sent from the game (IN Endpoint) are as follows:

FF [packet length] [packet data] [checksum]

[packet length] is the length of [packet data] (not the checksum or FF).

[checksum] is the sum of all the bytes preceding it (including the FF and [packet length]).

[packet data] consists of:

[command] [sequence] [optional attached data]

[sequence] is an auto-incrementing sequence, so that the game can match up the command packet with the response it goes with.

Start-up Commands

80 - Hello. This command is the first sent to the Base, and has a consistent response - potentially based on device types.

81 - Initiate device challenge. This command is sent along with an 8 byte scrambled number, and this scrambled number is the seed for a random number generator used in future challenges. These 8 bytes are descrambled, then used as a seed, and a blank response is sent. Within Dolphin, this is implemented using the Descramble and GenerateSeed methods. The descramble algorithm is shown below:

```
u32 InfinityBase::Descramble(u64 num_to_descramble)
{
  u64 mask = 0x8E55AA1B3999E8AA;
  u32 ret = 0;

  for (int i = 0; i < 64; i++)
  {
    if ((mask & 0x8000000000000000) > 0)
    {
      ret = ((ret << 1) & 0xFFFFFFFF) | (num_to_descramble & 0x01);
    }

    num_to_descramble = num_to_descramble >> 1;
    mask = mask << 1;
  }

  return ret;
}
```

83 - Get Next Challenge Response. This command is used to obtain the next random number that the game is expecting, based on the seed generated after the challenge was initiated. If the next random number is not what the game was expecting, then it drops communication with the Base and the game needs to be restarted. In Dolphin, this is implemented using the GetNext and Scramble methods, with the Scramble method shown below. As far as I am aware, the 'garbage' param can be anything.

```
u64 InfinityBase::Scramble(u32 num_to_scramble, u32 garbage)
{
  u64 mask = 0x8E55AA1B3999E8AA;
  u64 ret = 0;

  for (int i = 0; i < 64; i++)
  {
    ret = ret << 1;

    if ((mask & 0x01) > 0)
    {
      ret |= (num_to_scramble & 0x01);
      num_to_scramble = num_to_scramble >> 1;
    }
    else
    {
      ret |= (garbage & 0x01);
      garbage = garbage >> 1;
    }

    mask = mask >> 1;
  }
  return ret;
}
```

Color Commands

90/92/93/95/96 - Color commands. These commands set the colours of lights on the Base, and there are lights in each of the toy slots. These are not implemented in Dolphin currently, but all send a blank response. Some of these commands will point to particular positions, contain colour information, and potentially time that the colour should be displayed for.

Read/Write Commands

A1 - Get Present Toys. In total, there are a total of 5 possible present figures on the Infinity Base. Player one and Player two, an ability disc (which sits underneath the character toys), and a hexagon piece (play set or item). For each toy present, the base responds with 2 bytes. The first byte is the toy's position + the order the toy was added, and the second byte appears to always be 0x09. Player one's slot position is 0x20, player two's slot position is 0x30, and the hexagon slot is 0x10. If player one's ability was added first, then player one's character was added second, and then a play set was added the response would be 0x21 0x09 0x22 0x09 0x12 0x09. the order added persists across toys, so if player one's character was removed (in the previous scenario), then it's ability disc was removed, then the character was added back, it would still respond 0x22 0x09. The relevant method within Dolphin is InfinityBase::GetPresentFigures.

A2 - Read Toy data. This command attempts to read one of the 4 important blocks of data mentioned previously, block 1, 4, 8 or 12. This command sends 3 bytes to know which toy to read from, the first byte is the order added (generated when the toy is added, and known by the game due to the A1 command), the 2nd byte is the block to read from (0, 1, 2 or 3, which corresponds with 1, 4, 8 or 12) and then 0 or 1. I believe the 3rd byte is not relevant. In the response, all 16 bytes from the request block on the toy are returned. The relevant method within Dolphin is InfinityBase::QueryBlock.

A3 - Write Toy Data. This command attempts to write a block of 16 bytes to the specified toy. The command contains the same information as the read command, which specifies which character to write to, which block to write to, and again either 0 or 1. The 16 bytes specified are encrypted, so don't need to be run through the aes key generation or SHA1 digest. The relevant method within Dolphin is InfinityBase::WriteBlock.

B4 - Get UID. This command is used to grab the 7 byte UID of the NFC chip embedded within each toy. Presumably, this is what the game bases it's future decryption off of when reading figure data, and within Dolphin (when creating blank toys) we generate this UID with a random number generator. The data sent in the request is the same as read/write, one byte specifying which toy it wants to get the ID of based on the order it was added. Relevant method within Dolphin is InfinityBase::GetFigureIdentifier.

B5 - Unknown, potentially a status command. Response is empty.

Response structure

Responses are structured as follows:

[AA | AB] [packet length] [packet data] [checksum]

The first byte is AA for responses to game command requests, and it is AB for when a toy has been added or removed.

[packet length] is the length of [packet data] (not the checksum or AA|AB).

[checksum] is the sum of all the bytes preceding it (including the AA|AB and [packet length]).

[packet data] consists of:

[sequence] [optional attached data]

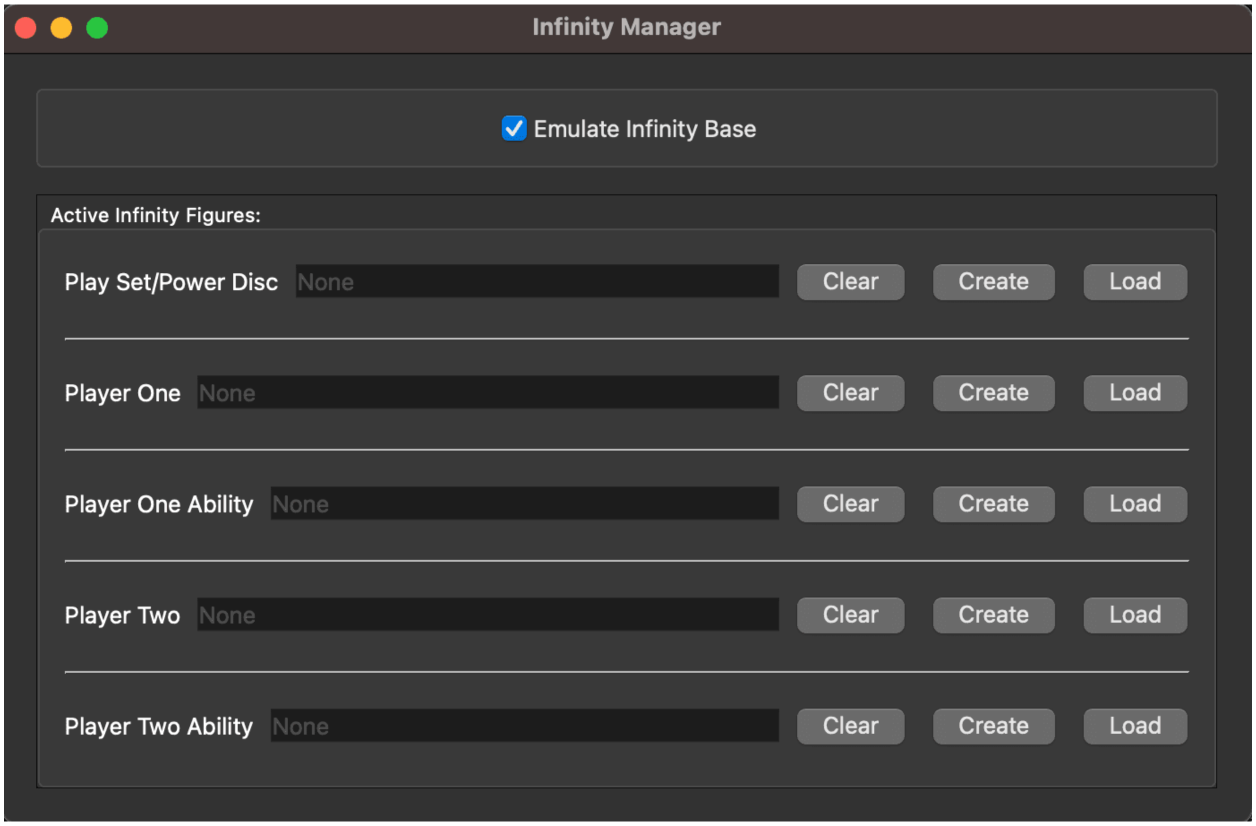
[sequence] corresponds with the matching FF command received, as noted above.

Unprompted Responses

In the case that a toy is added to or removed from the base, a response is queued up to inform the game there is a new set of data to read from. This response always starts with 0xAB as the first byte, then the packet length (4), then the packet data is the position (same as in A1 command), then 0x09, then the toy order added number, then 0x00 for a new toy being added, or 0x01 for a toy being removed. This response is added to a queue of character change responses, and the next time a response is sent a character change response is prioritised over any data requested from the game.

Creating Blank Toy Files

In terms of how this might translate to Dolphin - it is not an easy to follow process. Unfortunately due to the nature of the toys, with some that have some extra data (that 2nd section of 4 bytes in block 1), these values need to be included when creating blank toy files to be used in game. Granted - I need to do some more testing to follow up but from what I remember this data was important, and couldn't be dropped off/changed to some of the default data like other toys. I also need to investigate the option of using a generated/consistent NFC UID, but I am also not convinced that the game allows this. If I can gather any more evidence during some testing around whether the UID can be generated, and all the toys can have the same consistent 'extra' data depending on their toy type, then the list of toys stored within Dolphin can simply be a character name, an enum toy type, and a u16 character number.



The relevant methods for this are within the InfinityBaseWindow.cpp Qt class, and the Create/Load figure methods within the Infinity.cpp class. The create figure flow is also quite complex - but from the Qt Window where a User can specify what toy they would like to create, the CreateFigure method gets called, which then finds the character in the List, generates a 320 byte array, creates the 16 bytes based on the toy type to be put in block 1 of the toy file, creates a SHA1 digest based on existing Dolphin crypto SHA1 methods and the bytes specified in the Toys section of this article, big endian aligns the first 16 bytes, creates an AES key using existing Dolphin crypto methods, encrypts the character data block (block 0) and 4 blank blocks (4, 8, 12 and 13 as a backup), before writing the UID block, the encrypted character data block and the 4 encrypted blank blocks to the file created in the Qt window.

The character created is then displayed in the Qt window, with the character name appearing in a QLineEdit. The load process doesn't need to create any data, but still needs to follow the create processes' decryption in order to correctly display which character has been loaded in. The relevant functions for creating and loading (that contain the important decrypt/encrypt methods) are InfinityBase::LoadFigure and InfinityBase::CreateFigure.

Conclusions

There are definitely some things that can be optimised in terms of code that I have written - but as a whole I am super excited to be able to have researched and put together all of this work. While the Skylander Portal implementation was a rehash of existing research, an existing emulator implementation and some Dolphin extra flair, the Infinity Base is all my own code - and I am absolutely chuffed. Being able to do this project has allowed me to show off my critical thinking skills, while also being able to demonstrate the power of the Dolphin Emulator. If you read this far, feel free to reach out on my Github PR or via the Dolphin Discord Server - but otherwise check out this video I have made showcasing the tool in action!

I am also including the GenerateSeed and GetNext methods below, just because algorithms are fun:

```
void InfinityBase::GenerateSeed(u32 seed)
{
  _a = 0xF1EA5EED;
  _b = seed;
  _c = seed;
  _d = seed;

  for (int i = 0; i < 23; i++)
  {
    GetNext();
  }
}

u32 InfinityBase::GetNext()
{
  u32 a = _a;
  u32 b = _b;
  u32 c = _c;
  u32 ret = std::rotl(_b, 27);

  u32 temp = (a + ((ret ^ 0xFFFFFFFF) + 1)) & 0xFFFFFFFF;
  b = (b ^ std::rotl(c, 17)) & 0xFFFFFFFF;
  a = _d;
  c = (c + a) & 0xFFFFFFFF;
  ret = (b + temp) & 0xFFFFFFFF;
  a = (a + temp) & 0xFFFFFFFF;

  _c = a;
  _a = b;
  _b = c;
  _d = ret;

  return ret;
}
```