# BIOS Functions

The BIOS includes several System Call Functions which can be accessed by SWI instructions. Incoming parameters are usually passed through registers R0,R1,R2,R3. Outgoing registers R0,R1,R3 are typically containing either garbage, or return value(s). All other registers (R2,R4-R14) are kept unchanged.

**Caution**
When invoking SWIs from inside of ARM state specify SWI NN*10000h, instead of SWI NN as in THUMB state.

**Overview**


**All Functions Described**


**RAM Usage, BIOS Dumps**


**How BIOS Processes SWIs**
SWIs can be called from both within THUMB and ARM mode. In ARM mode, only the upper 8bit of the 24bit comment field are interpreted.
Each time when calling a BIOS function 4 words (SPSR, R11, R12, R14) are saved on Supervisor stack (_svc). Once it has saved that data, the SWI handler switches into System mode, so that all further stack operations are using user stack.
In some cases the BIOS may allow interrupts to be executed from inside of the SWI procedure. If so, and if the interrupt handler calls further SWIs, then care should be taken that the Supervisor Stack does not overflow.

# BIOS Function Summary

```
GBA  NDS7 NDS9 DSi7 DSi9 Basic Functions
00h  00h  00h  -    -    SoftReset
01h  -    -    -    -    RegisterRamReset
02h  06h  06h  06h  06h  Halt
03h  07h  -    07h  -    Stop/Sleep
04h  04h  04h  04h  04h  IntrWait       ;DSi7/DSi9: both bugged?
05h  05h  05h  05h  05h  VBlankIntrWait ;DSi7/DSi9: both bugged?
06h  09h  09h  09h  09h  Div
07h  -    -    -    -    DivArm
08h  0Dh  0Dh  0Dh  0Dh  Sqrt
09h  -    -    -    -    ArcTan
```

```
0Ah  -    -    -    -    ArcTan2
0Bh  0Bh  0Bh  0Bh  0Bh  CpuSet
0Ch  0Ch  0Ch  0Ch  0Ch  CpuFastSet
0Dh  -    -    -    -    GetBiosChecksum
0Eh  -    -    -    -    BgAffineSet
0Fh  -    -    -    -    ObjAffineSet
GBA  NDS7 NDS9 DSi7 DSi9 Decompression Functions
10h  10h  10h  10h  10h  BitUnPack
11h  11h  11h  11h  11h  LZ77UnCompReadNormalWrite8bit   ;"Wram"
12h  -    -    -    -    LZ77UnCompReadNormalWrite16bit  ;"Vram"
-    -    -    01h  01h  LZ77UnCompReadByCallbackWrite8bit
-    12h  12h  02h  02h  LZ77UnCompReadByCallbackWrite16bit
-    -    -    19h  19h  LZ77UnCompReadByCallbackWrite16bit (same as above)
13h  -    -    -    -    HuffUnCompReadNormal
-    13h  13h  13h  13h  HuffUnCompReadByCallback
14h  14h  14h  14h  14h  RLUnCompReadNormalWrite8bit     ;"Wram"
15h  -    -    -    -    RLUnCompReadNormalWrite16bit    ;"Vram"
-    15h  15h  15h  15h  RLUnCompReadByCallbackWrite16bit
16h  -    16h  -    16h  Diff8bitUnFilterWrite8bit       ;"Wram"
17h  -    -    -    -    Diff8bitUnFilterWrite16bit      ;"Vram"
18h  -    18h  -    18h  Diff16bitUnFilter
GBA  NDS7 NDS9 DSi7 DSi9 Sound (and Multiboot/HardReset/CustomHalt)
19h  08h  -    08h  -    SoundBias
1Ah  -    -    -    -    SoundDriverInit
1Bh  -    -    -    -    SoundDriverMode
1Ch  -    -    -    -    SoundDriverMain
1Dh  -    -    -    -    SoundDriverVSync
1Eh  -    -    -    -    SoundChannelClear
1Fh  -    -    -    -    MidiKey2Freq
20h  -    -    -    -    SoundWhatever0
21h  -    -    -    -    SoundWhatever1
22h  -    -    -    -    SoundWhatever2
23h  -    -    -    -    SoundWhatever3
24h  -    -    -    -    SoundWhatever4
25h  -    -    -    -    MultiBoot
26h  -    -    -    -    HardReset
27h  1Fh  -    1Fh  -    CustomHalt
28h  -    -    -    -    SoundDriverVSyncOff
29h  -    -    -    -    SoundDriverVSyncOn
2Ah  -    -    -    -    SoundGetJumpList
GBA  NDS7 NDS9 DSi7 DSi9 New NDS Functions
-    03h  03h  03h  03h  WaitByLoop
-    0Eh  0Eh  0Eh  0Eh  GetCRC16
-    0Fh  0Fh  -    -    IsDebugger
-    1Ah  -    1Ah  -    GetSineTable
-    1Bh  -    1Bh  -    GetPitchTable (DSi7: bugged)
-    1Ch  -    1Ch  -    GetVolumeTable
-    1Dh  -    1Dh  -    GetBootProcs (DSi7: only 1 proc)
-    -    1Fh  -    1Fh  CustomPost
GBA  NDS7 NDS9 DSi7 DSi9 New DSi Functions (RSA/SHA1)
-    -    -    20h  20h  RSA_Init_crypto_heap
-    -    -    21h  21h  RSA_Decrypt
-    -    -    22h  22h  RSA_Decrypt_Unpad
-    -    -    23h  23h  RSA_Decrypt_Unpad_GetChunk04
-    -    -    24h  24h  SHA1_Init
-    -    -    25h  25h  SHA1_Update
-    -    -    26h  26h  SHA1_Finish
-    -    -    27h  27h  SHA1_Init_update_fin
-    -    -    28h  28h  SHA1_Compare_20_bytes
-    -    -    29h  29h  SHA1_Random_maybe
GBA  NDS7 NDS9 DSi7 DSi9 Invalid Functions
2Bh+ 20h+ 20h+ -    -    Crash (SWI xxh..FFh do jump to garbage addresses)
-    xxh  xxh  -    -    Jump to 0   (on any SWI numbers not listed above)
-    -    -    12h  12h  No function (ignored)
-    -    -    2Bh  2Bh  No function (ignored)
```

```
-    -    -    40h+ 40h+ Mirror      (SWI 40h..FFh mirror to 00h..3Fh)
-    -    -    xxh  xxh  Hang        (on any SWI numbers not listed above)
```

Invalid NDS functions: NDS7 SWI 01h, 02h, 0Ah, 16h-19h, 1Eh, and NDS9 SWI 01h, 02h, 07h, 08h, 0Ah, 17h, 19h-1Eh will jump to zero (ie. to the NDS7 reset vector, or to NDS9 unused (usually PU-locked ITCM) memory, which will be both redirected to the debug handler, if any).
Invalid DSi functions: DSi9 SWI 00h, 07h-08h, 0Ah, 0Fh, 17h, 1Ah-1Eh, 2Ah, 2Ch-3Fh do hang in endless loop.

# BIOS Differences between GBA and NDS functions

**Differences between GBA and NDS BIOS functions**
- SoftReset uses different addresses
- SWI numbers for Halt, Stop/Sleep, Div, Sqrt have changed
- Halt destroys r0 on NDS9, IntrWait bugged on NDS9
- CpuFastSet allows 4-byte blocks (nice), but...
- CpuFastSet works very SLOW because of a programming bug (uncool)
- Some of the decompression functions are now using callbacks
- SoundBias uses new delay parameter

And, a number of GBA functions have been removed, and some new NDS functions have been added, see:
BIOS Function Summary

# BIOS Arithmetic Functions

Div
DivArm
Sqrt
ArcTan
ArcTan2

**SWI 06h (GBA) or SWI 09h (NDS7/NDS9/DSi7/DSi9) - Div**
Signed Division, r0/r1.

```
r0  signed 32bit Number
r1  signed 32bit Denom
```

Return:

```
r0  Number DIV Denom ;signed
r1  Number MOD Denom ;signed
r3  ABS (Number DIV Denom) ;unsigned
```

For example, incoming -1234, 10 should return -123, -4, +123.
The function usually gets caught in an endless loop upon division by zero.
Note: The NDS9 and DSi9 additionally support hardware division, by math coprocessor, accessed via I/O Ports, however, the SWI function is a raw software division.

**SWI 07h (GBA) - DivArm**
Same as above (SWI 06h Div), but incoming parameters are exchanged, r1/r0 (r0=Denom, r1=number). For compatibility with ARM's library. Slightly slower (3 clock cycles) than SWI 06h.

**SWI 08h (GBA) or SWI 0Dh (NDS7/NDS9/DSi7/DSi9) - Sqrt**
Calculate square root.

```
r0   unsigned 32bit number
```

Return:

```
r0   unsigned 16bit number
```

The result is an integer value, so Sqrt(2) would return 1, to avoid this inaccuracy, shift left incoming number by 2*N as much as possible (the result is then shifted left by 1*N). Ie. Sqrt(2 shl 30) would return 1.41421 shl 15.
Note: The NDS9 and DSi9 additionally support hardware square root calculation, by math coprocessor, accessed via I/O Ports, however, the SWI function is a raw software calculation.

**SWI 09h (GBA) - ArcTan**
Calculates the arc tangent.
```
  r0   Tan, 16bit (1bit sign, 1bit integral part, 14bit decimal part)
```
Return:
```
  r0   "-PI/2<THETA/<PI/2" in a range of C000h-4000h.
```
Note: there is a problem in accuracy with "THETA<-PI/4, PI/4<THETA".

**SWI 0Ah (GBA) - ArcTan2**
Calculates the arc tangent after correction processing.
Use this in normal situations.
```
  r0   X, 16bit (1bit sign, 1bit integral part, 14bit decimal part)
  r1   Y, 16bit (1bit sign, 1bit integral part, 14bit decimal part)
```
Return:
```
  r0   0000h-FFFFh for 0<=THETA<2PI.
```

# BIOS Rotation/Scaling Functions

BgAffineSet
ObjAffineSet

**SWI 0Eh (GBA) - BgAffineSet**
Used to calculate BG Rotation/Scaling parameters.
```
  r0   Pointer to Source Data Field with entries as follows:
        s32  Original data's center X coordinate (8bit fractional portion)
        s32  Original data's center Y coordinate (8bit fractional portion)
        s16  Display's center X coordinate
        s16  Display's center Y coordinate
        s16  Scaling ratio in X direction (8bit fractional portion)
        s16  Scaling ratio in Y direction (8bit fractional portion)
        u16  Angle of rotation (8bit fractional portion) Effective Range 0-FFFF
  r1   Pointer to Destination Data Field with entries as follows:
        s16  Difference in X coordinate along same line
        s16  Difference in X coordinate along next line
        s16  Difference in Y coordinate along same line
        s16  Difference in Y coordinate along next line
        s32  Start X coordinate
        s32  Start Y coordinate
  r2   Number of Calculations
```
Return: No return value, Data written to destination address.

**SWI 0Fh (GBA) - ObjAffineSet**
Calculates and sets the OBJ's affine parameters from the scaling ratio and angle of rotation.
The affine parameters are calculated from the parameters set in Srcp.
The four affine parameters are set every Offset bytes, starting from the Destp address.
If the Offset value is 2, the parameters are stored contiguously. If the value is 8, they match the structure of OAM.
When Srcp is arrayed, the calculation can be performed continuously by specifying Num.
```
  r0   Source Address, pointing to data structure as such:
        s16  Scaling ratio in X direction (8bit fractional portion)
        s16  Scaling ratio in Y direction (8bit fractional portion)
        u16  Angle of rotation (8bit fractional portion) Effective Range 0-FFFF
  r1   Destination Address, pointing to data structure as such:
```

```
       s16  Difference in X coordinate along same line
       s16  Difference in X coordinate along next line
       s16  Difference in Y coordinate along same line
       s16  Difference in Y coordinate along next line
  r2   Number of calculations
  r3   Offset in bytes for parameter addresses (2=continuous, 8=OAM)
```

Return: No return value, Data written to destination address.

For both Bg- and ObjAffineSet, Rotation angles are specified as 0-FFFFh (covering a range of 360 degrees), however, the GBA BIOS recurses only the upper 8bit; the lower 8bit may contain a fractional portion, but it is ignored by the BIOS.

# BIOS Decompression Functions

BitUnPack
Diff8bitUnFilter
HuffUnComp
LZ77UnComp
RLUnComp

**Decompression Read/Write Variants**

```
  ReadNormal:      Fast (src must be memory mapped)
  ReadByCallback:  Slow (src can be non-memory, eg. serial Firmware SPI bus)
  Write8bitUnits:  Fast (dest must support 8bit writes, eg. not VRAM)
  Write16bitUnits: Slow (dest must be halfword-aligned) (for VRAM)
```

**BitUnPack - SWI 10h (GBA/NDS7/NDS9/DSi7/DSi9)**

Used to increase the color depth of bitmaps or tile data. For example, to convert a 1bit monochrome font into 4bit or 8bit GBA tiles. The Unpack Info is specified separately, allowing to convert the same source data into different formats.

```
  r0  Source Address      (no alignment required)
  r1  Destination Address (must be 32bit-word aligned)
  r2  Pointer to UnPack information:
       16bit  Length of Source Data in bytes     (0-FFFFh)
       8bit   Width of Source Units in bits      (only 1,2,4,8 supported)
       8bit   Width of Destination Units in bits (only 1,2,4,8,16,32 supported)
       32bit  Data Offset (Bit 0-30), and Zero Data Flag (Bit 31)
      The Data Offset is always added to all non-zero source units.
      If the Zero Data Flag was set, it is also added to zero units.
```

Data is written in 32bit units, Destination can be Wram or Vram. The size of unpacked data must be a multiple of 4 bytes. The width of source units (plus the offset) should not exceed the destination width.
Return: No return value, Data written to destination address.

**Diff8bitUnFilterWrite8bit (Wram) - SWI 16h (GBA/NDS9/DSi9)**
**Diff8bitUnFilterWrite16bit (Vram) - SWI 17h (GBA)**
**Diff16bitUnFilter - SWI 18h (GBA/NDS9/DSi9)**

These aren't actually real decompression functions, destination data will have exactly the same size as source data. However, assume a bitmap or wave form to contain a stream of increasing numbers such like 10..19, the filtered/unfiltered data would be:

```
  unfiltered:   10  11  12  13  14  15  16  17  18  19
  filtered:     10  +1  +1  +1  +1  +1  +1  +1  +1  +1
```

In this case using filtered data (combined with actual compression algorithms) will obviously produce better compression results.
Data units may be either 8bit or 16bit used with Diff8bit or Diff16bit functions respectively.

```
  r0  Source address (must be aligned by 4) pointing to data as follows:
       Data Header (32bit)
```

```
        Bit 0-3   Data size (must be 1 for Diff8bit, 2 for Diff16bit)
        Bit 4-7   Type (must be 8 for DiffFiltered)
        Bit 8-31  24bit size after decompression
      Data Units (each 8bit or 16bit depending on used SWI function)
        Data0          ;original data
        Data1-Data0    ;difference data
        Data2-Data1    ;...
        Data3-Data2
        ...
  r1  Destination address
```

Return: No return value, Data written to destination address.

**HuffUnCompReadNormal - SWI 13h (GBA)**
**HuffUnCompReadByCallback - SWI 13h (NDS/DSi)**

The decoder starts in root node, the separate bits in the bitstream specify if the next node is node0 or node1, if that node is a data node, then the data is stored in memory, and the decoder is reset to the root node. The most often used data should be as close to the root node as possible. For example, the 4-byte string "Huff" could be compressed to 6 bits: 10-11-0-0, with root.0 pointing directly to data "f", and root.1 pointing to a child node, whose nodes point to data "H" and data "u".

Data is written in units of 32bits, if the size of the compressed data is not a multiple of 4, please adjust it as much as possible by padding with 0.

Align the source address to a 4Byte boundary.

```
  r0  Source Address, aligned by 4, pointing to:
       Data Header (32bit)
         Bit0-3   Data size in bit units (normally 4 or 8)
         Bit4-7   Compressed type (must be 2 for Huffman)
         Bit8-31  24bit size of decompressed data in bytes
       Tree Size (8bit)
         Bit0-7   Size of Tree Table/2-1 (ie. Offset to Compressed Bitstream)
       Tree Table (list of 8bit nodes, starting with the root node)
        Root Node and Non-Data-Child Nodes are:
         Bit0-5   Offset to next child node,
                  Next child node0 is at (CurrentAddr AND NOT 1)+Offset*2+2
                  Next child node1 is at (CurrentAddr AND NOT 1)+Offset*2+2+1
         Bit6     Node1 End Flag (1=Next child node is data)
         Bit7     Node0 End Flag (1=Next child node is data)
        Data nodes are (when End Flag was set in parent node):
         Bit0-7   Data (upper bits should be zero if Data Size is less than 8)
       Compressed Bitstream (stored in units of 32bits)
         Bit0-31  Node Bits (Bit31=First Bit)  (0=Node0, 1=Node1)
  r1  Destination Address
  r2  Callback temp buffer      ;\for NDS/DSi "ReadByCallback" variants only
  r3  Callback structure        ;/(see Callback notes below)
```

Return: No return value, Data written to destination address.

**LZ77UnCompReadNormalWrite8bit (Wram) - SWI 11h (GBA/NDS7/NDS9/DSi7/DSi9)**
**LZ77UnCompReadNormalWrite16bit (Vram) - SWI 12h (GBA)**
**LZ77UnCompReadByCallbackWrite8bit - SWI 01h (DSi7/DSi9)**
**LZ77UnCompReadByCallbackWrite16bit - SWI 12h (NDS), SWI 02h or 19h (DSi)**

Expands LZ77-compressed data. The Wram function is faster, and writes in units of 8bits. For the Vram function the destination must be halfword aligned, data is written in units of 16bits.

CAUTION: Writing 16bit units to [dest-1] instead of 8bit units to [dest] means the reading from [dest-1] won't work, ie. the "Vram" function works only with disp=001h..FFFh, but not with disp=000h.

If the size of the compressed data is not a multiple of 4, please adjust it as much as possible by padding with 0.

Align the source address to a 4-Byte boundary.

```
  r0  Source address, pointing to data as such:
       Data header (32bit)
         Bit 0-3   Reserved
         Bit 4-7   Compressed type (must be 1 for LZ77)
         Bit 8-31  Size of decompressed data
```

```
      Repeat below. Each Flag Byte followed by eight Blocks.
      Flag data (8bit)
        Bit 0-7   Type Flags for next 8 Blocks, MSB first
      Block Type 0 - Uncompressed - Copy 1 Byte from Source to Dest
        Bit 0-7   One data byte to be copied to dest
      Block Type 1 - Compressed - Copy N+3 Bytes from Dest-Disp-1 to Dest
        Bit 0-3   Disp MSBs
        Bit 4-7   Number of bytes to copy (minus 3)
        Bit 8-15  Disp LSBs
  r1  Destination address
  r2  Callback parameter        ;\for NDS/DSi "ReadByCallback" variants only
  r3  Callback structure        ;/(see Callback notes below)
```

Return: No return value.

**RLUnCompReadNormalWrite8bit (Wram) - SWI 14h (GBA/NDS7/NDS9/DSi7/DSi9)**
**RLUnCompReadNormalWrite16bit (Vram) - SWI 15h (GBA)**
**RLUnCompReadByCallbackWrite16bit - SWI 15h (NDS7/NDS9/DSi7/DSi9)**
Expands run-length compressed data. The Wram function is faster, and writes in units of 8bits. For the Vram function the destination must be halfword aligned, data is written in units of 16bits.
If the size of the compressed data is not a multiple of 4, please adjust it as much as possible by padding with 0.
Align the source address to a 4Byte boundary.

```
  r0  Source Address, pointing to data as such:
      Data header (32bit)
        Bit 0-3   Reserved
        Bit 4-7   Compressed type (must be 3 for run-length)
        Bit 8-31  Size of decompressed data
      Repeat below. Each Flag Byte followed by one or more Data Bytes.
      Flag data (8bit)
        Bit 0-6   Expanded Data Length (uncompressed N-1, compressed N-3)
        Bit 7     Flag (0=uncompressed, 1=compressed)
      Data Byte(s) - N uncompressed bytes, or 1 byte repeated N times
  r1  Destination Address
  r2  Callback parameter        ;\for NDS/DSi "ReadByCallback" variants only
  r3  Callback structure        ;/(see Callback notes below)
```

Return: No return value, Data written to destination address.

**NDS/DSi Decompression Callbacks**
On NDS and DSi, the "ReadByCallback" variants are reading source data from callback functions (rather than directly from memory). The callback functions may read normal data from memory, or from other devices, such like directly from the gamepak bus, without storing the source data in memory. The downside is that the callback mechanism makes the function very slow, furthermore, NDS7/NDS9 SWI 12h, 13h, 15h are using THUMB code, and variables on stack, alltogether that makes the whole shit very-very-very slow.

```
  r2 = user defined callback parameter (passed on to Open function)
       (or, for Huffman: pointer to temp buffer, max 200h bytes needed)
  r3 = pointer to callback structure
```

Callback structure (five 32bit pointers to callback functions)

```
  Open_and_get_32bit (eg. LDR r0,[r0], get header)
  Close              (optional, 0=none)
  Get_8bit           (eg. LDRB r0,[r0])
  Get_16bit          (not used)
  Get_32bit          (used by Huffman only)
```

All functions may use ARM or THUMB code (indicated by address bit0). The current source address (r0) is passed to all callback functions. Additionally, the initial destination address (r1), and a user defined parameter (r2) are passed to the Open function. For Huffman r2 must point to a temp buffer (max 200h bytes needed, internally used by the SWI function to make a copy of the huffman tree; needed for random-access to the tree, which wouldn't work with the sequentially reading callbacks).
All functions have return values in r0. The Open function normally returns the first word (containing positive length and type), alternatively it may return a negative error code to abort/reject decompression. The Close function, if it is defined, should return zero (or any positive value), or a negative errorcode. The other functions return raw data, without errorcodes. The SWI returns the length of decompressed data, or the signed errorcode

from the Open/Close functions.

## BIOS Memory Copy

CpuFastSet
CpuSet

**SWI 0Ch (GBA/NDS7/NDS9/DSi7/DSi9) - CpuFastSet**
Memory copy/fill in units of 32 bytes. Memcopy is implemented as repeated LDMIA/STMIA [Rb]!,r2-r9 instructions. Memfill as single LDR followed by repeated STMIA [Rb]!,r2-r9.
After processing all 32-byte-blocks, the NDS/DSi additonally processes the remaining words as 4-byte blocks.
BUG: The NDS/DSi uses the fast 32-byte-block processing only for the first N bytes (not for the first N words), so only the first quarter of the memory block is FAST, the remaining three quarters are SLOWLY copied word-by-word.
The length is specifed as wordcount, ie. the number of bytes divided by 4.
On the GBA, the length should be a multiple of 8 words (32 bytes) (otherwise the GBA is forcefully rounding-up the length). On NDS/DSi, the length may be any number of words (4 bytes).

```
  r0    Source address        (must be aligned by 4)
  r1    Destination address   (must be aligned by 4)
  r2    Length/Mode
          Bit 0-20  Wordcount (GBA: rounded-up to multiple of 8 words)
          Bit 24    Fixed Source Address (0=Copy, 1=Fill by WORD[r0])
```

Return: No return value, Data written to destination address.

**SWI 0Bh (GBA/NDS7/NDS9/DSi7/DSi9) - CpuSet**
Memory copy/fill in units of 4 bytes or 2 bytes. Memcopy is implemented as repeated LDMIA/STMIA [Rb]!,r3 or LDRH/STRH r3,[r0,r5] instructions. Memfill as single LDMIA or LDRH followed by repeated STMIA [Rb]!,r3 or STRH r3,[r0,r5].
The length must be a multiple of 4 bytes (32bit mode) or 2 bytes (16bit mode). The (half)wordcount in r2 must be length/4 (32bit mode) or length/2 (16bit mode), ie. length in word/halfword units rather than byte units.

```
  r0    Source address        (must be aligned by 4 for 32bit, by 2 for 16bit)
  r1    Destination address   (must be aligned by 4 for 32bit, by 2 for 16bit)
  r2    Length/Mode
          Bit 0-20  Wordcount (for 32bit), or Halfwordcount (for 16bit)
          Bit 24    Fixed Source Address (0=Copy, 1=Fill by {HALF}WORD[r0])
          Bit 26    Datasize (0=16bit, 1=32bit)
```

Return: No return value, Data written to destination address.

Note: On GBA, NDS7 and DSi7, these two functions will silently reject to do anything if the source start or end addresses are reaching into the BIOS area. The NDS9 and DSi9 don't have such read-proctections.

## BIOS Halt Functions

Halt
IntrWait
VBlankIntrWait
Stop/Sleep
CustomHalt

**SWI 02h (GBA) or SWI 06h (NDS7/NDS9/DSi7/DSi9) - Halt**
Halts the CPU until an interrupt request occurs. The CPU is switched into low-power mode, all other circuits (video, sound, timers, serial, keypad, system clock) are kept operating.
Halt mode is terminated when any enabled interrupts are requested, that is when (IE AND IF) is not zero, the

GBA locks up if that condition doesn't get true. However, the state of CPUs IRQ disable bit in CPSR register, and the IME register are don't care, Halt passes through even if either one has disabled interrupts.
On GBA and NDS7/DSi7, Halt is implemented by writing to HALTCNT, Port 4000301h. On NDS9/DSi9, Halt is implemted by writing to System Control Coprocessor (mov p15,0,c7,c0,4,r0 opcode), this opcode hangs if IME=0.
No parameters, no return value.
(GBA/NDS7/DSi7: all registers unchanged, NDS9/DSi9: R0 destroyed)

**SWI 04h (GBA/NDS7/NDS9/DSi7/DSi9) - IntrWait ;DSi7/DSi9=bugged?**
Continues to wait in Halt state until one (or more) of the specified interrupt(s) do occur. The function forcefully sets IME=1. When using multiple interrupts at the same time, this function is having less overhead than repeatedly calling the Halt function.

```
r0    0=Return immediately if an old flag was already set (NDS9: bugged!)
      1=Discard old flags, wait until a NEW flag becomes set
r1    Interrupt flag(s) to wait for (same format as IE/IF registers)
r2    DSi7 only: Extra flags (same format as DSi7's IE2/IF2 registers)
```

Caution: When using IntrWait or VBlankIntrWait, the user interrupt handler MUST update the BIOS Interrupt Flags value in RAM; when acknowleding processed interrupt(s) by writing a value to the IF register, the same value should be also ORed to the BIOS Interrupt Flags value, at following memory location:

```
Host     GBA (16bit)  NDS7 (32bit)  NDS9 (32bit)  DSi7-IF2 (32bit)
Address  [3007FF8h]   [380FFF8h]    [DTCM+3FF8h]  [380FFC0h]
```

NDS9: BUG: No Discard (r0=0) doesn't work. The function always waits for at least one IRQ to occur (no matter which, including IRQs that are not selected in r1), even if the desired flag was already set. NB. the same bug is also found in the GBA/NDS7 functions, but it's compensated by a second bug, ie. the GBA/NDS7 functions are working okay because their "bug doesn't work".
Return: No return value, the selected flag(s) are automatically reset in BIOS Interrupt Flags value in RAM upon return.
DSi9: BUG: The function tries to enter Halt state via Port 4000301h (which would be okay on ARM7, but it's probably ignored on ARM9, which should normally use CP15 to enter Halt state; if Port 4000301h is really ignored, then the function will "successfully" wait for interrupts, but without actually entering any kind of low power mode).
DSi7: BUG: The function tries to wait for IF and IF2 interrupts, but it does accidently ignore the old IF interrupts, and works only with new IF2 ones.

**SWI 05h (GBA/NDS7/NDS9/DSi7/DSi9) - VBlankIntrWait ;DSi7/DSi9=bugged?**
Continues to wait in Halt status until a new V-Blank interrupt occurs.
The function sets r0=1 and r1=1 (plus r2=0 on DSi7) and does then execute IntrWait (SWI 04h), see IntrWait for details.
No parameters, no return value.

**SWI 03h (GBA) - Stop**
Switches the GBA into very low power mode (to be used similar as a screen-saver). The CPU, System Clock, Sound, Video, SIO-Shift Clock, DMAs, and Timers are stopped.
Stop state can be terminated by the following interrupts only (as far as enabled in IE register): Joypad, Game Pak, or General-Purpose-SIO.
"The system clock is stopped so the IF flag is not set."
Preparation for Stop:
Disable Video before implementing Stop (otherwise Video just freezes, but still keeps consuming battery power). Possibly required to disable Sound also? Obviously, it'd be also recommended to disable any external hardware (such like Rumble or Infra-Red) as far as possible.
No parameters, no return value.

**SWI 07h (NDS7/DSi7) - Sleep**
No info, probably similar as GBA SWI 03h (Stop). Sleep is implemented for ARM7 only, not for ARM9. But maybe the ARM7 function does stop <both> ARM7 and ARM9 (?)

**SWI 27h (GBA) or SWI 1Fh (NDS7/DSi7) - CustomHalt (Undocumented)**
Writes the 8bit parameter value to HALTCNT, below values are equivalent to Halt and Stop/Sleep functions, other values reserved, purpose unknown.

```
  r2  8bit parameter (GBA: 00h=Halt, 80h=Stop) (NDS7/DSi7: 80h=Halt, C0h=Sleep)
```

No return value.

# BIOS Reset Functions

SoftReset
RegisterRamReset
HardReset

**SWI 00h (GBA/NDS7/NDS9) - SoftReset**
Clears 200h bytes of RAM (containing stacks, and BIOS IRQ vector/flags), initializes system, supervisor, and irq stack pointers, sets R0-R12, LR_svc, SPSR_svc, LR_irq, and SPSR_irq to zero, and enters system mode.
Note that the NDS9 stack registers are hardcoded (the DTCM base should be set to the default setting of 0800000h). The NDS9 function additionally flushes caches and write buffer, and sets the CP15 control register to 12078h.

```
  Host  sp_svc    sp_irq    sp_sys    zerofilled area       return address
  GBA   3007FE0h  3007FA0h  3007F00h  [3007E00h..3007FFFh]  Flag[3007FFAh]
  NDS7  380FFDCh  380FFB0h  380FF00h  [380FE00h..380FFFFh]  Addr[27FFE34h]
  NDS9  0803FC0h  0803FA0h  0803EC0h  [DTCM+3E00h..3FFFh]   Addr[27FFE24h]
```

The NDS7/NDS9 return addresses at [27FFE34h/27FFE24h] are usually containing copies of Cartridge Header [034h/024h] entry points, which may select ARM/THUMB state via bit0. The GBA return address 8bit flag is interpreted as 00h=8000000h (ROM), or 01h-FFh=2000000h (RAM), entered in ARM state.
Note: The reset is applied only to the CPU that has executed the SWI (ie. on the NDS, the other CPU will remain unaffected).
Return: Does not return to calling procedure, instead, loads the above return address into R14, and then jumps to that address by a "BX R14" opcode.

**SWI 01h (GBA) - RegisterRamReset**
Resets the I/O registers and RAM specified in ResetFlags. However, it does not clear the CPU internal RAM area from 3007E00h-3007FFFh.

```
  r0  ResetFlags
       Bit   Expl.
       0     Clear 256K on-board WRAM  ;-don't use when returning to WRAM
       1     Clear 32K on-chip WRAM    ;-excluding last 200h bytes
       2     Clear Palette
       3     Clear VRAM
       4     Clear OAM              ;-zerofilled! does NOT disable OBJs!
       5     Reset SIO registers    ;-switches to general purpose mode!
       6     Reset Sound registers
       7     Reset all other registers (except SIO, Sound)
```

Return: No return value.
Bug: LSBs of SIODATA32 are always destroyed, even if Bit5 of R0 was cleared.
The function always switches the screen into forced blank by setting DISPCNT=0080h (regardless of incoming R0, screen becomes white).

**SWI 26h (GBA) - HardReset (Undocumented)**
This function reboots the GBA (including for getting through the time-consuming nintendo intro, which is making the function particularly useless and annoying).
Parameters: None. Return: Never/Reboot.
Execution Time: About 2 seconds (!)

# BIOS Misc Functions

GetBiosChecksum
WaitByLoop
GetCRC16
IsDebugger
GetSineTable
GetPitchTable
GetVolumeTable
CustomPost
GetBootProcs

**SWI 0Dh (GBA) - GetBiosChecksum (Undocumented)**
Calculates the checksum of the BIOS ROM (by reading in 32bit units, and adding up these values). IRQ and FIQ are disabled during execution.
The checksum is BAAE187Fh (GBA and GBA SP), or BAAE1880h (DS in GBA mode, whereas the only difference is that the byte at [3F0Ch] is changed from 00h to 01h, otherwise the BIOS is 1:1 same as GBA BIOS, it does even include multiboot code).
Parameters: None. Return: r0=Checksum.

**SWI 03h (NDS7/NDS9/DSi7/DSi9) - WaitByLoop**
Performs a "LOP: SUB R0,1 / BGT LOP" wait loop, the loop is executed in BIOS memory, which provides reliable timings (regardless of the memory waitstates & cache state of the calling procedure). Intended only for short delays (eg. flash memory programming cycles).

```
  r0  Delay value (should be in range 1..7FFFFFFFh)
```

Execution Time: NDS7: R0*4 cycles, plus some overload on SWI handling.
Execution Time: NDS9: R0*2 (cache on), or R0*8 (cache off), plus overload.
Note: Both NDS7 and NDS9 timings are counted in 33.51MHz units.
Return: No return value.

**SWI 0Eh (NDS7/NDS9) - GetCRC16**

```
  r0  Initial CRC value (16bit, usually FFFFh)
  r1  Start Address   (must be aligned by 2)
  r2  Length in bytes (must be aligned by 2)
```

CRC16 checksums can be calculated as such:

```
  val[0..7] = C0C1h,C181h,C301h,C601h,CC01h,D801h,F001h,A001h
  for i=start to end
    crc=crc xor byte[i]
    for j=0 to 7
      crc=crc shr 1:if carry then crc=crc xor (val[j] shl (7-j))
    next j
  next i
```

Return:

```
  r0  Calculated 16bit CRC Value
```

Additionally, if the length is nonzero, r3 contains the last processed halfword at [addr+len-2]. Unlike most other NDS7/DSi7 SWI functions (which do reject reading from BIOS memory), this allows to dump the NDS7/DSi7 BIOS (except for the memory region that is locked via BIOSPROT Port 4000308h).

**SWI 0Fh (NDS7/NDS9) - IsDebugger**
Detects if 4MB (normal) or 8MB (debug version) Main RAM installed.
Caution: Fails on ARM9 when cache is enabled (always returns 8MB state).
Return: r0 = result (0=normal console 4MB, 1=debug version 8MB)
Destroys halfword at [27FFFFAh] (NDS7) or [27FFFF8h] (NDS9)!

The SWI 0Fh function doesn't work stable if it gets interrupted by an interrupt which is calling SWI 0Fh, which would destroy the above halfword scratch value (unless the IRQ handler has saved/restored the halfword).

**SWI 1Ah (NDS7/DSi7) - GetSineTable**

```
  r0  Index (0..3Fh) (must be in that range, otherwise returns garbage)
```

Return: r0 = Desired Entry (0000h..7FF5h) ;SIN(0 .. 88.6 degrees)*8000h

**SWI 1Bh (NDS7/DSi7) - GetPitchTable (DSi7: bugged)**

```
  r0  Index (0..2FFh) (must be in that range, otherwise returns garbage)
```

BUG: DSi7 accidently reads from SineTable instead of PitchTable, as workaround for obtaining PitchTable values, one can set "r0=(0..2FFh)-46Ah" on DSi.
Return: r0 = Desired Entry (0000h..FF8Ah) (unsigned)

**SWI 1Ch (NDS7/DSi7) - GetVolumeTable**

```
  r0  Index (0..2D3h) (must be in that range, otherwise returns garbage)
```

Return: r0 = Desired Entry (00h..7Fh) (unsigned)

**SWI 1Fh (NDS9/DSi7) - CustomPost**

Writes to the POSTFLG register, probably for use by Firmware boot procedure.

```
  r0  32bit value, to be written to POSTFLG, Port 4000300h
```

Return: No return value.

**SWI 1Dh (NDS7/DSi7) - GetBootProcs**

Returns addresses of Gamecart boot procedure/interrupt handler, probably for use by Firmware boot procedure. Most of the returned NDS7 functions won't work if the POSTFLG register is set.
The return values are somewhat XORed by each other (on DSi7 most of the values are zero; which does rather negate the XORing effect, and, as a special gimmick, one of the zero values is XORed by incoming r2).

# BIOS Multi Boot (Single Game Pak)

MultiBoot

**SWI 25h (GBA) - MultiBoot**

This function uploads & starts program code to slave GBAs, allowing to launch programs on slave units even if no cartridge is inserted into the slaves (this works because all GBA BIOSes contain built-in download procedures in ROM).
However, the SWI 25h BIOS upload function covers only 45% of the required Transmission Protocol, the other 55% must be coded in the master cartridge (see Transmission Protocol below).

```
  r0  Pointer to MultiBootParam structure
  r1  Transfer Mode (undocumented)
       0=256KHz, 32bit, Normal mode    (fast and stable)
       1=115KHz, 16bit, MultiPlay mode (default, slow, up to three slaves)
       2=2MHz,   32bit, Normal mode    (fastest but maybe unstable)
  Note: HLL-programmers that are using the MultiBoot(param_ptr) macro cannot
  specify the transfer mode and will be forcefully using MultiPlay mode.
```

Return:

```
  r0  0=okay, 1=failed
```

See below for more details.

**Multiboot Parameter Structure**

Size of parameter structure should be 4Ch bytes (the current GBA BIOS uses only first 44h bytes though). The following entries must be set before calling SWI 25h:

```
  Addr Size Name/Expl.
  14h  1    handshake_data (entry used for normal mode only)
  19h  3    client_data[1,2,3]
```

```
1Ch  1    palette_data
1Eh  1    client_bit (Bit 1-3 set if child 1-3 detected)
20h  4    boot_srcp  (typically 8000000h+0C0h)
24h  4    boot_endp  (typically 8000000h+0C0h+length)
```

The transfer length (excluding header data) should be a multiple of 10h, minimum length 100h, max 3FF40h (ca. 256KBytes). Set palette_data as "81h+color*10h+direction*8+speed*2", or as "0f1h+color*2" for fixed palette, whereas color=0..6, speed=0..3, direction=0..1. The other entries (handshake_data, client_data[1-3], and client_bit) must be same as specified in Transmission Protocol (see below hh,cc,y).

**Multiboot Transfer Protocol**

Below describes the complete transfer protocol, normally only the Initiation part must be programmed in the master cartridge, the main data transfer can be then performed by calling SWI 25h, the slave program is started after SWI 25h completion.

The ending handshake is normally not required, when using it, note that you will need custom code in BOTH master and slave programs.

```
Times  Send   Receive  Expl.
----------------------Required Transfer Initiation in master program
...    6200   FFFF     Slave not in multiplay/normal mode yet
1      6200   0000     Slave entered correct mode now
15     6200   720x     Repeat 15 times, if failed: delay 1/16s and restart
1      610y   720x     Recognition okay, exchange master/slave info
60h    xxxx   NN0x     Transfer C0h bytes header data in units of 16bits
1      6200   000x     Transfer of header data completed
1      620y   720x     Exchange master/slave info again
...    63pp   720x     Wait until all slaves reply 73cc instead 720x
1      63pp   73cc     Send palette_data and receive client_data[1-3]
1      64hh   73uu     Send handshake_data for final transfer completion
----------------------Below is SWI 25h MultiBoot handler in BIOS
DELAY  -      -        Wait 1/16 seconds at master side
1      llll   73rr     Send length information and receive random data[1-3]
LEN    yyyy   nnnn     Transfer main data block in units of 16 or 32 bits
1      0065   nnnn     Transfer of main data block completed, request CRC
...    0065   0074     Wait until all slaves reply 0075 instead 0074
1      0065   0075     All slaves ready for CRC transfer
1      0066   0075     Signalize that transfer of CRC follows
1      zzzz   zzzz     Exchange CRC must be same for master and slaves
----------------------Optional Handshake (NOT part of master/slave BIOS)
...    ....   ....     Exchange whatever custom data
```

Legend for above Protocol

```
y     client_bit, bit(s) 1-3 set if slave(s) 1-3 detected
x     bit 1,2,or 3 set if slave 1,2,or 3
xxxx  header data, transferred in 16bit (!) units (even in 32bit normal mode)
nn    response value for header transfer, decreasing 60h..01h
pp    palette_data
cc    random client_data[1..3] from slave 1-3, FFh if slave not exists
hh    handshake_data, 11h+client_data[1]+client_data[2]+client_data[3]
uu    random data, not used, ignore this value
```

Below automatically calculated by SWI 25h BIOS function (don't care about)

```
llll  download length/4-34h
rr    random data from each slave for encryption, FFh if slave not exists
yyyy  encoded data in 16bit (multiplay) or 32bit (normal mode) units
nnnn  response value, lower 16bit of destadr in GBA memory (00C0h and up)
zzzz  16bit download CRC value, must be same for master and slaves
```

Pseudo Code for SWI 25h Transfer with Checksum and Encryption calculations

```
if normal_mode    then c=C387h:x=C37Bh:k=43202F2Fh
if multiplay_mode then c=FFF8h:x=A517h:k=6465646Fh
m=dword(pp,cc,cc,cc):f=dword(hh,rr,rr,rr)
for ptr=000000C0h to (file_size-4) step 4
  c=c xor data[ptr]:for i=1 to 32:c=c shr 1:if carry then c=c xor x:next
  m=(6F646573h*m)+1
  send_32_or_2x16 (data[ptr] xor (-2000000h-ptr) xor m xor k)
next
```

```
  c=c xor f:for i=1 to 32:c=c shr 1:if carry then c=c xor x:next
  wait_all_units_ready_for_checksum:send_32_or_1x16 (c)
```

Whereas, explained: c=chksum,x=chkxor,f=chkfin,k=keyxor,m=keymul

**Multiboot Communication**
In Multiplay mode, master sends 16bit data, and receives 16bit data from each slave (or FFFFh if none). In Normal mode, master sends 32bit data (upper 16bit zero, lower 16bit as for multiplay mode), and receives 32bit data (upper 16bit as for multiplay mode, and lower 16bit same as lower 16bit previously sent by master). Because SIODATA32 occupies same addresses as SIOMULTI0-1, the same transfer code can be used for both multiplay and normal mode (in normal mode SIOMULTI2-3 should be forced to FFFFh though). After each transfer, master should wait for Start bit cleared in SIOCNT register, followed by a 36us delay.
Note: The multiboot slave would also recognize data being sent in Joybus mode, however, master GBAs cannot use joybus mode (because GBA hardware cannot act as master in joybus mode).

**Multiboot Slave Header**
The transferred Header block is written to 2000000-20000BFh in slave RAM, the header must contain valid data (identically as for normal ROM-cartridge headers, including a copy of the Nintendo logo, correct header CRC, etc.), in most cases it'd be recommended just to transfer a copy of the master cartridges header from 8000000h-80000BFh.

**Multiboot Slave Program/Data**
The transferred main program/data block is written to 20000C0h and up (max 203FFFFh) in slave RAM, note that absolute addresses in the program must be then originated at 2000000h rather than 8000000h. In case that the master cartridge is 256K or less, it could just transfer a copy of the whole cartridge at 80000C0h and up, the master should then copy & execute its own ROM data into RAM as well.

**Multiboot Slave Extended Header**
For Multiboot slaves, separate Entry Point(s) must be defined at the beginning of the Program/Data block (the Entry Point in the normal header is ignored), also some reserved bytes in this section are overwritten by the Multiboot procedure. For more information see chapter about Cartridge Header.

**Multiboot Slave with Cartridge**
Beside for slaves without cartridge, multiboot can be also used for slaves which do have a cartridge inserted, if so, SELECT and START must be kept held down during power-on in order to switch the slave GBA into Multiboot mode (ie. to prevent it from starting the cartridge as normally).
The general idea is to enable newer programs to link to any existing older GBA programs, even if these older programs originally didn't have been intended to support linking.
The uploaded program may access the slaves SRAM, Flash ROM, or EEPROM (if any, allowing to read out or modify slave game positions), as well as cartridge ROM at 80000A0h-8000FFFh (the first 4KBytes, excluding the nintendo logo, allowing to read out the cartridge name from the header, for example).
The main part of the cartridge ROM is meant to be locked out in order to prevent software pirates from uploading "intruder" programs which would send back a copy of the whole cartridge to the master, however, for good or evil, at present time, current GBA models and GBA carts do not seem to contain any such protection.

**Uploading Programs from PC**
Beside for the ability to upload a program from one GBA to another, this feature can be also used to upload small programs from a PC to a GBA. For more information see chapter about External Connectors.

**Nintendo DS**
The GBA multiboot function requires a link port, and so, works on GBA and GBA SP only. The Nintendo DS in GBA mode does include the multiboot BIOS function, but it won't be of any use as the DS doesn't have a link port.

# BIOS Sound Functions

MidiKey2Freq
SoundBias
SoundChannelClear
SoundDriverInit
SoundDriverMain
SoundDriverMode
SoundDriverVSync
SoundDriverVSyncOff
SoundDriverVSyncOn
SoundWhatever0..4
SoundGetJumpList

**SWI 1Fh (GBA) - MidiKey2Freq**
Calculates the value of the assignment to ((SoundArea)sa).vchn[x].fr when playing the wave data, wa, with the interval (MIDI KEY) mk and the fine adjustment value (halftones=256) fp.

```
  r0  WaveData* wa
  r1  u8 mk
  r2  u8 fp
```

Return:

```
  r0  u32
```

This function is particularly popular because it allows to read from BIOS memory without copy protection range checks. The formula to read one byte (a) from address (i, 0..3FFF) is:
a = (MidiKey2Freq(i-(((i AND 3)+1)OR 3), 168, 0) * 2) SHR 24

**SWI 19h (GBA) or SWI 08h (NDS7/DSi7) - SoundBias**
Increments or decrements the current level of the SOUNDBIAS register (with short delays) until reaching the desired new level. The upper bits of the register are kept unchanged.

```
  r0   BIAS level (0=Level 000h, any other value=Level 200h)
  r1   Delay Count (NDS/DSi only) (GBA uses a fixed delay count of 8)
```

Return: No return value.

**SWI 1Eh (GBA) - SoundChannelClear**
Clears all direct sound channels and stops the sound.
This function may not operate properly when the library which expands the sound driver feature is combined afterwards. In this case, do not use it.
No parameters, no return value.

**SWI 1Ah (GBA) - SoundDriverInit**
Initializes the sound driver. Call this only once when the game starts up.
It is essential that the work area already be secured at the time this function is called.
You cannot execute this driver multiple times, even if separate work areas have been prepared.

```
  r0  Pointer to work area for sound driver, SoundArea structure as follows:
       SoundArea (sa) Structure
        u32    ident      Flag the system checks to see whether the
                          work area has been initialized and whether it
                          is currently being accessed.
        vu8    DmaCount   User access prohibited
        u8     reverb     Variable for applying reverb effects to direct sound
        u16    d1         User access prohibited
        void   (*func)()  User access prohibited
        int    intp       User access prohibited
        void*  NoUse      User access prohibited
        SndCh  vchn[MAX]  The structure array for controlling the direct
                          sound channels (currently 8 channels are
                          available). The term "channel" here does
```

```
                    not refer to hardware channels, but rather to
                    virtual constructs inside the sound driver.
 s8     pcmbuf[PCM_BF*2]
SoundChannel Structure
 u8         sf     The flag indicating the status of this channel.
                   When 0 sound is stopped.
                   To start sound, set other parameters and
                   then write 80h to here.
                   To stop sound, logical OR 40h for a
                   release-attached off (key-off), or write zero
                   for a pause. The use of other bits is
                   prohibited.
 u8         r1     User access prohibited
 u8         rv     Sound volume output to right side
 u8         lv     Sound volume output to left side
 u8         at     The attack value of the envelope. When the
                   sound starts, the volume begins at zero and
                   increases every 1/60 second. When it
                   reaches 255, the process moves on to the
                   next decay value.
 u8         de     The decay value of the envelope. It is
                   multiplied by "this value/256" every 1/60
                   sec. and when sustain value is reached, the
                   process moves to the sustain condition.
 u8         su     The sustain value of the envelope. The
                   sound is sustained by this amount.
                   (Actually, multiplied by rv/256, lv/256 and
                   output left and right.)
 u8         re     The release value of the envelope. Key-off
                   (logical OR 40h in sf) to enter this state.
                   The value is multiplied by "this value/256"
                   every 1/60 sec. and when it reaches zero,
                   this channel is completely stopped.
 u8         r2[4]  User access prohibited
 u32        fr     The frequency of the produced sound.
                   Write the value obtained with the
                   MidiKey2Freq function here.
 WaveData*  wp     Pointer to the sound's waveform data. The waveform
                   data can be generated automatically from the AIFF
                   file using the tool (aif2agb.exe), so users normally
                   do not need to create this themselves.
 u32        r3[6]  User access prohibited
 u8         r4[4]  User access prohibited
WaveData Structure
 u16   type    Indicates the data type. This is currently not used.
 u16   stat    At the present time, non-looped (1 shot) waveform
               is 0000h and forward loop is 4000h.
 u32   freq    This value is used to calculate the frequency.
               It is obtained using the following formula:
               sampling rate x 2^((180-original MIDI key)/12)
 u32   loop    Loop pointer (start of loop)
 u32   size    Number of samples (end position)
 s8    data[]  The actual waveform data. Takes (number of samples+1)
               bytes of 8bit signed linear uncompressed data. The last
               byte is zero for a non-looped waveform, and the same
               value as the loop pointer data for a looped waveform.
```

Return: No return value.

**SWI 1Ch (GBA) - SoundDriverMain**

Main of the sound driver.

Call every 1/60 of a second. The flow of the process is to call SoundDriverVSync, which is explained later, immediately after the V-Blank interrupt.

After that, this routine is called after BG and OBJ processing is executed.

No parameters, no return value.

**SWI 1Bh (GBA) - SoundDriverMode**
Sets the sound driver operation mode.

```
r0  Sound driver operation mode
     Bit    Expl.
     0-6    Direct Sound Reverb value (0-127, default=0) (ignored if Bit7=0)
     7      Direct Sound Reverb set (0=ignore, 1=apply reverb value)
     8-11   Direct Sound Simultaneously-produced (1-12 channels, default 8)
     12-15  Direct Sound Master volume (1-15, default 15)
     16-19  Direct Sound Playback Frequency (1-12 = 5734,7884,10512,13379,
            15768,18157,21024,26758,31536,36314,40137,42048, def 4=13379 Hz)
     20-23  Final number of D/A converter bits (8-11 = 9-6bits, def. 9=8bits)
     24-31  Not used.
```

Return: No return value.

**SWI 1Dh (GBA) - SoundDriverVSync**
An extremely short system call that resets the sound DMA. The timing is extremely critical, so call this function immediately after the V-Blank interrupt every 1/60 second.
No parameters, no return value.

**SWI 28h (GBA) - SoundDriverVSyncOff**
Due to problems with the main program if the V-Blank interrupts are stopped, and SoundDriverVSync cannot be called every 1/60 a second, this function must be used to stop sound DMA.
Otherwise, even if you exceed the limit of the buffer the DMA will not stop and noise will result.
No parameters, no return value.

**SWI 29h (GBA) - SoundDriverVSyncOn**
This function restarts the sound DMA stopped with the previously described SoundDriverVSyncOff.
After calling this function, have a V-Blank occur within 2/60 of a second and call SoundDriverVSync.
No parameters, no return value.

**SWI 20h..24h (GBA) - SoundWhatever0..4 (Undocumented)**
Whatever undocumented sound-related BIOS functions.

**SWI 2Ah (GBA) - SoundGetJumpList (Undocumented)**
Receives pointers to 36 additional sound-related BIOS functions.

```
r0  Destination address (must be aligned by 4) (120h bytes buffer)
```

## BIOS SHA1 Functions (DSi only)

SHA1_Init(struct)
SHA1_Update(struct,src,srclen)
SHA1_Finish(dst,struct)
SHA1_Init_Update_Finish(dst,src,srclen)
SHA1_Init_Update_Finish_Mess(dst,dstlen,src,srclen)
SHA1_Compare_20_Bytes(src1,src2)
SHA1_Default_Callback(struct,src,len)

**SWI 24h (DSi9/DSi7) - SHA1_Init(struct)**
Initializes a 64h-byte structure for SHA1 calculations:

```
[struct+00h] = 67452301h      ;\
[struct+04h] = EFCDAB89h      ;
[struct+08h] = 98BADCFEh      ; initial SHA1 checksum value
[struct+0Ch] = 10325476h      ;
```

```
  [struct+10h] = C3D2E1F0h      ;/
  [struct+14h] = 00000000h ;lsw ;\total len in bits, initially zero
  [struct+18h] = 00000000h ;msw ;/
  [struct+1Ch] = uninitialzed   ;-buffer for incomplete fragment (40h bytes)
  [struct+5Ch] = 00000000h      ;-incomplete fragment size
  if [struct+60h] = 00000000h then [struct+60h] = SHA1_Default_Callback
```

Observe that the incoming [struct+60h] value should be 00000000h, otherwise the default callback isn't installed (using a different callback doesn't make too much sense, and it's probably not done by any DSi programs) (the callback feature might be intended to mount hardware accelleration, or to hook, customize, encrypt, or replace the SHA1 functionality).

### SWI 25h (DSi9/DSi7) - SHA1_Update(struct,src,srclen)

This function should be placed between Init and Finish. The Update function can be called multiple times if the source data is split into separate blocks. There's no alignment requirement (though the function works faster if src is 4-byte aligned).

```
  [struct+14h]=[struct+14h]+len*8  ;64bit value ;-raise total len in bits
  if [struct+5Ch]<>0 and [struct+5Ch]+len>=40h  ;\
    for i=[struct+5Ch] to 3Fh                   ; merge old incomplete chunk
      [struct+1Ch+i]=[src], src=src+1, len=len-1; with new data and process it
    SHA1_Callback(struct,struct+1Ch,40h)        ; (if it gives a full chunk)
    [struct+5Ch]=0                              ;/
  if len>=40h then                              ;\process full 40h-byte chunks
    SHA1_Callback(struct,src,len AND NOT 3Fh)   ; (if src isn't 4-byte aligned
    src=src+(len AND NOT 3Fh)                   ; then the DSi BIOS internally
    len=len AND 3Fh                             ;/copies all chunks to struct)
  if len>0 then                                 ;\
    for i=[struct+5Ch] to [struct+5Ch]+len-1    ; memorize remaining bytes
      [struct+1Ch+i]=[src], src=src+1, len=len-1; as incomplete chunk
      [struct+5Ch]=[struct+5Ch]+1               ;/
```

### SWI 26h (DSi9/DSi7) - SHA1_Finish(dst,struct)

```
  [total_len]=bswap8byte([struct+14h]) ;get total len in bits in big-endian
  SHA1_Update(struct,value_80h,1)                           ;append end byte
  while [struct+5Ch]<>38h do SHA1_Update(struct,value_00h,1) ;append padding
  SHA1_Update(struct,total_len,8)                           ;append 64bit len
  [struct+14h]=bswap8byte([total_len]) ;restore total len, exclude above update
  [dst+00h]=bswap([struct+00h]  ;msw   ;\
  [dst+04h]=bswap([struct+04h]         ; store SHA1 result at dst
  [dst+08h]=bswap([struct+08h]         ; (in big-endian)
  [dst+0Ch]=bswap([struct+0Ch]         ;
  [dst+10h]=bswap([struct+10h]  ;lsw   ;/
```

### SHA1_Default_Callback(struct,src,len)

```
  for j=1 to len/40h
    a=[struct+0], b=[struct+4], c=[struct+8], d=[struct+0Ch], e=[struct+10h]
    for i=0 to 79
      if i=0..15  then w[i] = bswap([src]), src=src+4
      if i=16..79 then w[i] = (w[i-3] xor w[i-8] xor w[i-14] xor w[i-16]) rol 1
      if i=0..19  then f=5A827999h + e + (d xor (b and (c xor d)))
      if i=20..39 then f=6ED9EBA1h + e + (b xor c xor d)
      if i=40..59 then f=8F1BBCDCh + e + ((b and c) or (d and (b or c)))
      if i=60..79 then f=CA62C1D6h + e + (b xor c xor d)
      e=d, d=c, c=(b ror 2), b=a, a=f + (a rol 5) + w[i]
    [struct+0]=[struct+0]+a, [struct+4]=[struct+4]+b, [struct+8]=[struct+8]+c
    [struct+0Ch]=[struct+0Ch]+d, [struct+10h]=[struct+10h]+e
```

### SWI 27h (DSi9/DSi7) - SHA1_Init_Update_Finish(dst,src,srclen)

```
  [struct+60h]=00000000h  ;want Init to install the default SHA1 callback
  SHA1_Init(struct)
  SHA1_Update(struct,src,srclen)
  SHA1_Finish(dst,struct)
```

Always returns r0=1.

**SWI 29h (DSi9/DSi7) - SHA1_Init_Update_Finish_Mess(dst,dstlen,src,srclen)**

```
  if dst=0 then exit(r0=1)  ;uh, that's same return value as when okay
  if src=0 and srclen<>0 then exit(r0=0)
  [struct+60h]=00000000h                ;\
  SHA1_Init(struct)                     ; first compute normal SHA1
  SHA1_Update(struct,src,srclen)        ; (same as SHA1_Init_Update_Finish)
  SHA1_Finish(first_sha1,struct)        ;/
 @@lop1:
  i=13h  ;start with LSB of big-endian 20-byte value  ;\increment SHA1 value
 @@lop2:                                              ; by one (with somewhat
  [first_sha1+i]=[first_sha1+i]+1, i=i-1               ; uncommon/bugged carry-
  if i>=0 and [first_sha1+i+1]=01h then goto @@lop2   ;/out to higher bytes)
  SHA1_Update(struct,first_sha1,14h)    ;\compute 2nd SHA1 across 1st SHA1,
  SHA1_Finish(second_sha1,struct)       ;/done without re-initializing struct
  for i=0 to min(14h,dstlen)-1, [dst]=[second_sha1+i], dst=dst+1
  dstlen=dstlen-min(14h,dstlen)
  if dstlen<>0 then goto @@lop1 else exit(r0=1)
```

**SHA1_Init_Update_Finish_HMAC(dst,key,src,srclen)**

```
  if len(key)>40h then key=SHA1(key) ;convert LONG keys to 14h-bytes length
  if len(key)<40h then zero-pad key to 40h-bytes length
  for i=0 to 3Fh, [inner_key+i]=[key+i] xor 36h ;\
  [struct+60h]=00000000h                        ;
  SHA1_Init(struct)                             ; compute 1st SHA1
  SHA1_Update(struct,inner_key,40h)             ; across inner key and data
  SHA1_Update(struct,src,srclen)                ;
  SHA1_Finish(first_sha1,struct)                ;/
  for i=0 to 3Fh, [outer_key+i]=[key+i] xor 5Ch ;\
  [struct+60h]=00000000h                        ;
  SHA1_Init(struct)                             ; compute final SHA1
  SHA1_Update(struct,outer_key,40h)             ; across outer key and 1st SHA1
  SHA1_Update(struct,first_sha1,14h)            ;
  SHA1_Finish(dst,struct)                       ;/
```

**SWI 28h (DSi9/DSi7) - SHA1_Compare_20_Bytes(src1,src2)**

Out: r0=1=match, r0=0=mismatch/error (error occurs if src1=0 or src2=0).

## BIOS RSA Functions (DSi only)

RSA_Init_crypto_heap(heap_nfo,heap_start,heap_size)
RSA_Decrypt(heap_nfo,struct,dest4)
RSA_Decrypt_Unpad(heap_nfo,dst,src,key)
RSA_Decrypt_Unpad_GetChunk04(heap_nfo,dst,src,key)

RSA is important because the DSi cartridge header and system files do contain RSA signatures. Which makes it impossible to create unlicensed software (without knowing Nintendo's private key).

**SWI 20h (DSi9/DSi7) - RSA_Init_crypto_heap(heap_nfo,heap_start,heap_size)**

Initializes the heap for use with SWI 21h..23h. heap_nfo is a 0Ch-byte structure, which gets set to:

```
  [heap_nfo+0] = heap_start (rounded-up to 4-byte boundary)
  [heap_nfo+4] = heap_end   (start+size, rounded-down to 4-byte boundary)
  [heap_nfo+8] = heap_size  (matched to above rounded values)
```

heap_start should point to a free memory block which will be used as heap, heap_size should be usually 1000h.

**SWI 21h (DSi9/DSi7) - RSA_Decrypt(heap_nfo,ptr_nfo,len_dest)**

```
  [ptr_nfo+0] = dst (usually 7Fh bytes, max 80h bytes)
```

```
  [ptr_nfo+4] = src (80h bytes)
  [ptr_nfo+8] = key (80h bytes)
```

This is a subfunction for SWI 22h/23h. It's returning raw decrypted data without unpadding (aside from stripping leading 00h bytes; most (or all) signatures are containing one leading 00h byte, so the returned [len_dest] value will be usually 7Fh).
Return value (r0) is: 0=failed, 1=okay. The length of the decrypted data is returned at [len_dest].

**SWI 22h (DSi9/DSi7) - RSA_Decrypt_Unpad(heap_nfo,dst,src,key)**
Same as SWI 21h, and additionally removes the "01h,min eight FFh,00h" padding. src,dst,key should be 80h-bytes. The output at dst can be theoretically max 80h-bytes (or shorter due to removed padding). In practice, the DSi is often using only the first 14h-bytes at dst (aka the last 14h-bytes from src) as SHA1 or SHA1-HMAC value (RSA SHA1). Return value (r0) is: 0=failed, 1=okay.

**SWI 23h (DSi9/DSi7) - RSA_Decrypt_Unpad_GetChunk04(heap_nfo,dst,src,key)**
Same as SWI 22h, but with some extra processing (DER related?) on the final decrypted & unpadded data:
The data must consist of five chunks (with IDs 30h,30h,06h,05h,04h), the last chunk (with ID=04h) must be 14h bytes in size, and the 14h-byte chunk data is then copied to dst. The other four chunks must exist, but their content is just skipped. Handling of chunks bigger than 7Fh looks quite weird/bugged.

**Unencrypted Signatures**
Emulators can hook the RSA BIOS functions to copy unencrypted signatures as-is (instead of trying to decrypt them). That allows to create "authentic" files that are fully compatible with the DSi firmware, and which would be actually working on real hardware (when knowing the private key).
Unencrypted 80h-byte signatures should be stored in following format (as defined in RFC 2313):

```
  00h   1      "00" Leading zero (00h)
  01h   1      "BT" Block type (always 01h on DSi)
  02h   8+n    "PS" Padding (FFh-filled, min 8 bytes, usually 69h bytes on DSi)
  0Ah+n 1      "00" Padding end (00h)
  0Bh+n 75h-n  "D"  Data (max 75h bytes, usually a 14h-byte SHA1 value on DSi)
```

If the hooked BIOS function sees a RSA source to contain "00h, 01h, at least eight FFh, followed by 00h", then it could treat it as unencrypted data (it's nearly impossible that an encrypted signature could contain those values).

**Key.Bit0**
The DSi BIOS contains two different RSA core modes (either one of them is used, depending on whether BIT0 of the FIRST BYTE of the "key" is set or cleared). The purpose & difference between those two modes is unknown. Also, dunno if that BIT0 thing is something common in the RSA world?

**DSi Public RSA Keys (public keys for decryption/verification)**

```
  ARM9BIOS:FFFF87F4h (ROM Key #0)  System Menu (Launcher)
  ARM9BIOS:FFFF8874h (ROM Key #1)  System Fun Tools and Wifi Firmware
  ARM9BIOS:FFFF88F4h (ROM Key #2)  System Base Tools (Settings, Shop)
  ARM9BIOS:FFFF8974h (ROM Key #3)  DSiWare and DSi ROM Cartridges
  ARM9BIOS:FFFF89F4h (ROM Key #4)  Unknown (probably a RSA key)
  ARM9BIOS:FFFF8A74h (ROM Key #5)  Unknown (probably a RSA key)
  ARM9BIOS:FFFF8AF4h (ROM Key #6)  Unknown (probably a RSA key)
  ARM9BIOS:FFFF8B74h (ROM Key #7)  Unknown (probably a RSA key)
  ARM9BIOS:FFFF9920h (ROM Stuff)   Unknown (maybe not a RSA key)
  Launcher (BA,F1,98,A4,...)       HWINFO_S.dat (with RSA-SHA1-HMAC)
  Launcher (9F,80,BC,5F,...)       Version Data and TWLFontTable.dat
  Launcher (E5,1C,BF,C7,...)       Unknown (some alternate/unused RSA key)
  Launcher (9E,C1,CC,C0,...)       Unknown (some RSA key for "DL")
  TWL_FIRM (F1,F5,1A,FF,...)       eMMC Boot Info
  Unknown                          DS Cart Whitelist (probably has RSA)
  Unknown                          HWID.sgn (is that RSA, too?)
  Unknown                          Newer NDS ROM Cartridges (have RSA, too?)
  Unknown                          Public key for Flipnote .ppm files
  Unknown                          ...?
```

**DSi Private RSA Keys (keys for encryption/signing)**
Nintendo's private keys should be known only by Nintendo. The console may have a few "own" private keys (eg. for sending signed data to Nintendo, or for storing signed data on SD card).

```
Unknown                          Private key for Flipnote .ppm files
Unknown                          ...?
```

# BIOS RAM Usage

Below contains info about RAM contents at cartridge boot time (as initialized by the BIOS/Firmware), plus info about RAM locations used by IRQ handlers and SWI functions.

**GBA BIOS RAM Usage**
Below memory at 3007Fxxh is often accessed directly, or via mirrors at 3FFFFxxh.

```
3000000h 7F00h User Memory and User Stack              (sp_usr=3007F00h)
3007F00h A0h   Default Interrupt Stack (6 words/time)  (sp_irq=3007FA0h)
3007FA0h 40h   Default Supervisor Stack (4 words/time) (sp_svc=3007FE0h)
3007FE0h 10h   Debug Exception Stack (4 words/time)    (sp_xxx=3007FF0h)
3007FF0h 4     Pointer to Sound Buffer (for SWI Sound functions)
3007FF4h 3     Reserved (unused)
3007FF7h 1     Reserved (intro/nintendo logo related)
3007FF8h 2     IRQ IF Check Flags (for SWI IntrWait/VBlankIntrWait functions)
3007FFAh 1     Soft Reset Re-entry Flag (for SWI SoftReset function)
3007FFBh 1     Reserved (intro/multiboot slave related)
3007FFCh 4     Pointer to user IRQ handler (to 32bit ARM code)
```

**NDS BIOS RAM Usage**
Below memory at 27FFxxxh is mirrored to 23FFxxxh (on retail consoles with 4MB RAM), however, it should be accessed via address 27FFxxxh (for compatibility with debug consoles with 8MB RAM). Accessing it via mirrors at 2FFFxxxh is also valid (this is done by DSi enhanced games; even when running in non-DSi mode; this allows DSi games to use the same memory addresses in NDS and DSi mode).

```
2000000h ...   ARM7 and ARM9 bootcode can be loaded here (2000000h..23BFDFFh)
2400000h ...   Debug bootcode can be loaded here (2400000h..27BFDFFh)
23FEE00h 168h  Fragments of NDS9 firmware boot code
27FF800h 4     NDS Gamecart Chip ID 1
27FF804h 4     NDS Gamecart Chip ID 2
27FF808h 2     NDS Cart Header CRC (verified)            ;hdr[15Eh]
27FF80Ah 2     NDS Cart Secure Area CRC (not verified ?) ;hdr[06Ch]
27FF80Ch 2     NDS Cart Missing/Bad CRC (0=Okay, 1=Missing/Bad)
27FF80Eh 2     NDS Cart Secure Area Bad (0=Okay, 1=Bad)
27FF810h 2     Boot handler task number (usually FFFFh at cart boot time)
27FF812h 2     Secure disable (0=Normal, 1=Disable; Cart[078h]=BIOS[1088h])
27FF814h 2     SIO Debug Connection Exists (0=No, 1=Yes)
27FF816h 2     RTC Status?                 (0=Okay, 1=Bad)
27FF818h 1     Random RTC    ;random LSB from SIO debug detect handshake
27FF819h 37h   Zerofilled by firmware
27FF850h 2     NDS7 BIOS CRC (5835h)
27FF860h 4     Somewhat copy of Cart[038h], nds7 ram addr (?)
27FF864h 4     Wifi FLASH User Settings Bad (0=Okay, 1=Bad)
27FF868h 4     Wifi FLASH User Settings FLASH Address (fmw[20h]*8)
                 maybe recommended to use above RAM cell instead FLASH entry?
27FF86Ch 4     Whatever (seems to be zero at cart boot time)
27FF870h 4     Whatever (seems to be zero at cart boot time)
27FF874h 2     Wifi FLASH firmware part5 crc16 (359Ah) (fmw[026h])
27FF876h 2     Wifi FLASH firmware part3/part4 crc16 (fmw[004h] or ZERO)
                 Above is usually ZERO at cart boot (set to fmw[004h] only
                 when running pictochat, or maybe also when changing user
                 settings)
27FF878h 08h   Not used
```

```
27FF880h 4     Message from NDS9 to NDS7  (=7 at cart boot time)
27FF884h 4     NDS7 Boot Task (also checked by NDS9) (=6 at cart boot time)
27FF888h ..    Whatever (seems to be zero at cart boot time)
27FF890h 4     Somewhat boot flags (somewhat B0002A22h)
                 bit10 part3/part4 loaded/decoded (bit3 set if bad crc)
                 bit28 part5 loaded/decoded with good crc
27FF894h 36Ch  Not used (zero)
27FFC00h 4     NDS Gamecart Chip ID 1   (copy of 27FF800h)
27FFC04h 4     NDS Gamecart Chip ID 2   (copy of 27FF804h)
27FFC08h 2     NDS Cart Header CRC      (copy of 27FF808h)
27FFC0Ah 2     NDS Cart Secure Area CRC (copy of 27FF80Ah)
27FFC0Ch 2     NDS Cart Missing/Bad CRC (copy of 27FF80Ch)
27FFC0Eh 2     NDS Cart Secure Area Bad (copy of 27FF80Eh)
27FFC10h 2     NDS7 BIOS CRC (5835h)    (copy of <27FF850h>)
27FFC12h 2     Secure Disable           (copy of 27FF812h)
27FFC14h 2     SIO Debug Exist          (copy of 27FF814h)
27FFC16h 1     RTC Status?              (<8bit> copy of 27FF816h)
27FFC17h 1     Random 8bit              (copy of <27FF818h>)
27FFC18h 18h   Not used (zero)
27FFC30h 2     GBA Cartridge Header[BEh], Reserved
27FFC32h 3     GBA Cartridge Header[B5h..B7h], Reserved
27FFC35h 1     Whatever flags ?
27FFC36h 2     GBA Cartridge Header[B0h], Maker Code
27FFC38h 4     GBA Cartridge Header[ACh], Gamecode
27FFC3Ch 4     Frame Counter (eg. 00000332h in no$gba with original firmware)
27FFC40h 2     Boot Indicator (0001h=normal; required for some NDS games)
27FFC42h 3Eh   Not used (zero)
27FFC80h 70h   Wifi FLASH User Settings (fmw[newest_user_settings])
27FFCF0h 10h   Not used (zero)
27FFDxxh ..    NDS9 Debug Exception Stack (stacktop=27FFD9Ch)
27FFD9Ch 4     NDS9 Debug Exception Vector (0=None)
27FFDA0h ..    ...
27FFE00h 170h  NDS Cart Header at 27FFE00h+0..16Fh
27FFF70h ..    Not used (zerofilled at cart boot time)
27FFFF8h 2     NDS9 Scratch addr for SWI IsDebugger check
27FFFFAh 2     NDS7 Scratch addr for SWI IsDebugger check
27FFFFCh ..    ...
27FFFFEh 2     Main Memory Control (on-chip power-down I/O port)
DTCM+3FF8h 4   NDS9 IRQ IF Check Bits (hardcoded RAM address)
DTCM+3FFCh 4   NDS9 IRQ Handler (hardcoded RAM address)
37F8000h FE00h ARM7 bootcode can be loaded here (37F8000h..3807DFFh)
380F700h 1D4h  Fragments of NDS7 firmware boot code
380F980h 4     Unknown/garbage (set to FBDD37BBh, purpose unknown)
                 NOTE: Cooking Coach is doing similar crap at 37FCF1Ch ?!?!
380FFC0h 4     DSi7 IRQ IF2 Check Bits (hardcoded RAM address) (DSi only)
380FFDCh ..    NDS7 Debug Stacktop / Debug Vector (0=None)
380FFF8h 4     NDS7 IRQ IF Check Bits (hardcoded RAM address)
380FFFCh 4     NDS7 IRQ Handler (hardcoded RAM address)
---
summary of nds memory used at cartridge boot time:
(all other memory zero-filled unless containing cartridge data)
37F8000h..3807E00h  ;cartridge area (nds7 only)
2000000h..23BFE00h  ;cartridge area (nds9 and nds7)
2400000h..27BFE00h  ;cartridge area (debug ver)
23FEE00h..23FEF68h  ;fragments of NDS9 firmware boot code
27FF800h..27FF85Fh  ;various values (from BIOS boot code)
27FF860h..27FF893h  ;various values (from Firmware boot code)
27FFC00h..27FFC41h  ;various values (from Firmware boot code)
27FFC80h..27FFCE6h  ;firmware user settings
27FFE00h..27FFF6Fh  ;cart header
380F700h..380F8D4h  ;fragments of NDS7 firmware boot code
380F980h            ;set to FBDD37BBh
---
register settings at cartridge boot time:
nds9 r0..r11     = zero
```

```
nds9 r12,r14,r15 = entrypoint
nds9 r13         = 3002F7Ch (!)
nds9 r13_irq     = 3003F80h
nds9 r13_svc     = 3003FC0h
nds9 r14/spsr_irq= zero
nds9 r14/spsr_svc= zero
---
nds7 r0..r11     = zero
nds7 r12,r14,r15 = entrypoint
nds7 r13         = 380FD80h
nds7 r13_irq     = 380FF80h
nds7 r13_svc     = 380FFC0h
nds7 r14/spsr_irq= zero
nds7 r14/spsr_svc= zero
---
Observe that SWI SoftReset applies different stack pointers:
Host  sp_svc    sp_irq    sp_sys    zerofilled area       return address
NDS7  380FFDCh  380FFB0h  380FF00h  [380FE00h..380FFFFh]  Addr[27FFE34h]
NDS9  0803FC0h  0803FA0h  0803EC0h  [DTCM+3E00h..3FFFh]   Addr[27FFE24h]
```

**DSi BIOS RAM**

Not too much known for sure (because DSi exploits may change RAM before allowing to execute custom diagnostics tools, and decrypting/emulating DSi firmware isn't yet possible).

Parts of memory are probably same as on NDS (but moved from 27FFxxxh to 2FFFxxxh). Some additional DSi specific memory areas:

```
2000400h 128h  System Settings from TWLCFGn.dat file (bytes 088h..1AFh)
2FF80xxh
2FF82xxh
2FF83xxh
2FF89xxh
2FF8Axxh
2FF8Bxxh
2FF8Cxxh
2FF8Dxxh       ... Wifi MAC address, channel mask, etc.
2FF8Fxxh
2FF90xxh
2FF91xxh
2FF9208h       FBDD37BBh (that odd "garbage" value occurs also on NDS)
2FFA1xxh
2FFA2xxh
2FFA5xxh
2FFA6xxh
2FFA680h 12    02FD4D80h,00000000h,00001980h
2FFA68Ch ..    Zerofilled
2FFC000h 1000h DSi Full Cart Header (same as at 2FFE000h)
2FFD000h 7B0h  Zerofilled
2FFD7B0h 8+1   Version Data Filename (eg. 30,30,30,30,30,30,30,34,00)
2FFD7B9h 1     Version Data Region   (eg. 50h="P"=Europe)
2FFD7BAh 1     Unknown  (00)   ;bit0 = warmboot-flag-related
2FFD7BBh 1     Unknown  (00)
2FFD7BCh 15+1  eMMC CID (dd,ss,ss,ss,ss,03,4D,30,30,46,50,41,00,00,15), 00
2FFD7CCh 15+1  eMMC CSD (40,40,96,E9,7F,DB,F6,DF,01,59,0F,2A,01,26,90), 00
2FFD7DCh 4     eMMC OCR (80,80,FF,80)
2FFD7E0h 8     eMMC ?   (00,04,00,00,00,00,00,00)
2FFD7E8h 2     eMMC RCA (01,00)
2FFD7EAh 2     eMMC ?   (01,00)
2FFD7ECh 14    eMMC ?   (00,00,00,00,00,00,00,00,00,09,00,00,00,01)
2FFD7FAh 2     eMMC ?   maybe Port 4004828h setting?  (E0,40)
2FFD7FCh 4     eMMC ?   (00,00,01,00)
2FFD800h 1     Unknown 05h (maybe number of IDs at 2FFD850h?)
2FFD801h 2Fh   Zerofilled
2FFD830h 1     Unknown 1Fh
2FFD831h 1Fh   Zerofilled
2FFD850h 5x8   Five Title IDs (ROM Cart, and HNBP, HNDA, HNEA, HNGP) why?
```

```
2FFD878h 788h  Zerofilled
2FFE000h 1000h DSi Full Cart Header (additionally to short headers)
2FFF000h 0Ch   Zerofilled
2FFF00Ch 4     ? 0000007Fh
2FFF010h 4     ? 550E25B8h
2FFF014h 4     ? 02FF4000h
2FFF018h A68h  Zerofilled
2FFFA80h 160h  Short Cart header (same as at 2FFFE00h)
2FFFBE0h 20h   Zerofilled
```

Below resembles NDS area at 27FFC00h (with added/removed stuff)...

```
2FFFC00h 4     NDS Gamecart Chip ID
2FFFC04h 20h   Zerofilled
2FFFC24h 5     ? (04 00 73 01 03)
2FFFC29h 7     Zerofilled
2FFFC30h 12    GBA Cartridge Header (FF FF FF FF FF 00 FF FF FF FF FF FF)
2FFFC3Ch 4     Frame Counter maybe? (eg. 1F 01 00 00 in cooking coach)
2FFFC40h 2     Boot Indicator (0001h=normal; required for some NDS games)
2FFFC42h 3Eh   Not used (zero)
2FFFC80h 70h   Wifi FLASH User Settings (fmw[newest_user_settings])
2FFFCF0h 4     ?           (3D 00 01 6E)
2FFFCF4h 6     MAC Address (00 23 CC xx xx xx) (fmw[036h])
2FFFCFAh 6     ?           (41 10 00 00 00 00)
2FFFD00h 68h   Zerofilled
2FFFD68h 4     Bitmask for Supported Languages (3Eh for Europe);\
2FFFD6Ch 4     Unknown (00,00,00,00)                           ; from
2FFFD70h 1     Console Region (0=JP,1=US,2=EU,3=AU,4=CHN,5=KOR); HWINFO_S.dat
2FFFD71h 12    Serial/Barcode (ASCII, 11-12 characters)        ;
2FFFD7Dh 3     ? (00 00 3C)                                    ;/
2FFFD80h 0Ch   Zerofilled
2FFFD8Ch 10h   ARM9 debug exception stack (stacktop 2FFFD9Ch)
2FFFD9Ch 4     ARM9 debug exception vector (020D3E64h)
2FFFDA0h 4     02F80000h                                ;\
2FFFDA4h 4     02FFA674h                                ;
2FFFDA8h 4     00000000h zero                           ; start addresses?
2FFFDACh 4     01FF86E0h itcm?                          ;
2FFFDB0h 4     027C00C0h                                ;
2FFFDB4h 4     02FFF000h                                ;
2FFFDB8h 4     03040000h wram?                          ;
2FFFDBCh 4     03800000h wram?                          ;
2FFFDC0h 4     0380C3B4h wram?                          ;/
2FFFDC4h 4     02F80000h                                ;\
2FFFDC8h 4     02FFC000h ptr to DSi Full Cart Header    ;
2FFFDCCh 4     00000000h zero                           ; end addresses?
2FFFDD0h 4     02000000h ram bottom?                    ; (for above nine
2FFFDD4h 4     027C0780h                                ; start addresses)
2FFFDD8h 4     02FFF680h                                ;
2FFFDDCh 4     03040000h wram?                          ;
2FFFDE0h 4     03800000h wram?                          ;
2FFFDE4h 4     0380F780h wram?                          ;/
2FFFDE8h 4     RTC Date at Boot (BCD) (yy,mm,dd,XX) (XX=maybe day-of-week?)
2FFFDECh 4     RTC Time at Boot (BCD) (hh,ss,mm,0) (hh.bit6=maybe PM or 24h?)
2FFFDF0h 4     Initial ARM7 Port 4004008h bits (13FBFB06h)
2FFFDF4h 1     Initial ARM7 Port 40040xxh bits (C4h)
2FFFDF5h 1     Initial ARM7 Port 400400xh bits (F0h)
2FFFDF6h 2+2   Zerofilled
2FFFDFAh 1     Warmboot Flag (bptwl[70h] OR 80h, ie. 80h=cold or 81h=warm)
2FFFDFBh 1     01h
2FFFDFCh 4     Pointer to TWLCFGn.dat (usually 2000400h) (or 0=2000400h)
2FFFE00h 160h  Short Cart header (unlike NDS, only 160h, not 170h)
2FFFF60h A0h   Zerofilled
380F010h 10h   AES key for dev.kp (E5,CC,5A,8B,...) (optional/for launcher)
xxxxxxxh ?     ARM7i and ARM9 bootcode can be loaded WHERE and WHERE?
cart_header[1D4h] base address where various structures and parameters...?
```

Initial state after DSi BIOS ROM bootcode (when starting eMMC bootcode) requires only a few memory blocks in ITCM, ARM7 WRAM, and AES keyslots:

```
1FFC400h 400h  BIOS Keys from FFFF87F4h (C3 02 93 DE ..) Whatever, 8x80h RSA?
1FFC800h 80h   BIOS Keys from FFFF9920h (30 33 26 D5 ..) Whatever
1FFC880h 14h   Whatever, should/may be zerofilled?
1FFC894h 1048h BIOS Keys from FFFF99A0h (99 D5 20 5F ..) Blowfish/NDS-mode
1FFD8DCh 1048h BIOS Keys from FFFFA9E8h (D8 18 FA BF ..) Blowfish/unused?
3FFC400h 200h  BIOS Keys from 00008188h (CA 13 31 79 ..) Whatever, 32x10h AES?
3FFC600h 40h   BIOS Keys from 0000B5D8h (AF 1B F5 16 ..) Whatever, AES?
3FFC640h 14h   Whatever, should/must be zerofilled?
3FFC654h 1048h ROM:0000C6D0h (59 AA 56 8E ..) Blowfish/DSi-mode
3FFD69Ch 1048h ROM:0000D718h (54 86 13 3B ..) Blowfish/unused?
3FFE6E4h 44h   eMMC info (to be relocated to 2FFD7BCh, see there for details)
4004450h 8     AES Key0.X ("Nintendo") for modcrypt
4004480h 10h   AES Key1.X (CPU/Console ID and constants) for dev.kp and Tad
40044xxh ?     AES Key2... (?)
40044E0h 1Ch   AES Key3.X/Y (CPU/Console ID and constants) for eMMC
```

# BIOS Dumping

**BIOSes**

```
GBA BIOS 16K  (fully dumpable)
NDS7 BIOS 16K (fully dumpable)
NDS9 BIOS 4K  (fully dumpable)
DSi7 BIOS 64K (about 41K dumpable)
DSi9 BIOS 64K (about 41K dumpable)
DSiWifi BIOS 80K on older DSi (fully dumpable)
DSiWifi BIOS Unknown size on newer DSi (probably fully dumpable)
```

**GBA BIOS**

Contains SWI Functions and Bootcode (for starting cartridges, or booting via Serial Port). The GBA BIOS can be read only by opcodes executed in BIOS area, for example, via the MidiKey2Freq function (most other SWI Functions (like CpuSet) are refusing source addresses within BIOS area).

**NDS BIOSes**

Contains SWI Functions and Bootcode (for booting from SPI Bus Firmware FLASH memory). The NDS9 BIOS can be dumped without restrictions (eg. via CpuSet, or via LDR opcodes in RAM). The NDS7 BIOS has same restrictions as GBA, ie. reading works only by BIOS opcodes, and not by functions like CpuSet. The GetCRC16 functions does work though (at least for memory at 1204h..3FFFh). As an additional obstacle, memory at 0000h..1203h can be dumped only by opcodes within 0000h..1203h (that memory does mainly contain data, but some of the data values can serve as THUMB LDR opcodes). For details see:
DS Memory Control - BIOS
Note: DSi consoles are containing a copy of the NDS BIOSes, but with BIOSPROT set to 0020h (even when running in NDS mode), so the first 20h bytes of the DSi's NDS7 BIOS aren't dumpable (except via tracing, see below), that 20h bytes should be just same as on original NDS7 though.

**DSi BIOSes - Lower 32K-halves (SWI Functions)**

The lower 32K of DSi9 doesn't have any restricions. The lower 32K of DSi7 has similar restrictions as NDS7, but with BIOSPROT set to 0020h (instead of 1204h), this is making it more easy to dump memory at 0020h..7FFFh (eg. via GetCRC16), but makes it impossible to dump the exception vectors at 0000h..001Fh, however, they can be deduced by tracing (with timer IRQs):

```
ROM:00000000h  EA000006  b 20h   ;dsi7_reset_vector
ROM:00000004h  EA000006  b 24h   ;dsi7_undef_handler
ROM:00000008h  EA00001F  b 8Ch   ;dsi7_swi_handler
ROM:0000000Ch  EA000004  b 24h   ;dsi7_prefetch_abort_handler
ROM:00000010h  EA000003  b 24h   ;dsi7_data_abort_handler
ROM:00000014h  EAFFFFFE  b 14h   ;reserved_vector
```

```
ROM:00000018h  EA000013  b 6Ch   ;dsi7_irq_handler
ROM:0000001Ch  EA000000  b 24h   ;dsi7_fiq_handler
```

Aside from branch opcodes, above could theoretically contain ALU opcodes that modify R15 (but that would be very unlikely, and would make no difference).

**DSi BIOSes - Upper 32K-halves (Bootcode, for booting from eMMC memory)**

The upper 32K of the DSi9 and DSi7 BIOSes are locked at some point during booting, and there's no known way to dump them directly. However, portions of that memory are relocated to RAM/TCM before locking, and that relocated copies can be dumped.

On a DSi, the following DSi ROM data can be dumped (via Main Memory hacks, ie. with complex external hardware soldered to the mainboard):

```
ROM:FFFF87F4h / TCM:1FFC400h (400h)  (C3 02 93 DE ..) Whatever, 8x80h RSA?
ROM:FFFF9920h / TCM:1FFC800h (80h)   (30 33 26 D5 ..) Whatever
ROM:FFFF99A0h / TCM:1FFC894h (1048h) (99 D5 20 5F ..) Blowfish/NDS-mode
ROM:FFFFA9E8h / TCM:1FFD8DCh (1048h) (D8 18 FA BF ..) Blowfish/unused?
ROM:00008188h / RAM:3FFC400h (200h)  (CA 13 31 79 ..) Whatever, 32x10h AES?
ROM:0000B5D8h / RAM:3FFC600h (40h)   (AF 1B F5 16 ..) Whatever, "common key"?
ROM:0000C6D0h / RAM:3FFC654h (1048h) (59 AA 56 8E ..) Blowfish/DSi-mode
ROM:0000D718h / RAM:3FFD69Ch (1048h) (54 86 13 3B ..) Blowfish/unused?
```

On a 3DS, the following "DSi ROM data" can be dumped from the 2470h-byte DSi key area in 3DS memory at ARM9 ITCM 01FFD000h..01FFF46F (via 3DS exploits that are capable of executing code on ARM9 side):

```
ROM:FFFF87F4h / 3DS:01FFD000h  200h  RSA key 0..3
ROM:00008308h / 3DS:01FFD200h  80h   some AES keys
ROM:FFFF9920h / 3DS:01FFD280h  80h   whatever
ROM:0000B5D8h / 3DS:01FFD300h  40h   AES keys and values (common etc)
ROM:?         / 3DS:01FFD340h  A0h   misc "Nintendo" string etc.
ROM:0000C6D0h / 3DS:01FFD3E0h  1048h Blowfish for DSi-mode
ROM:FFFF99A0h / 3DS:01FFE428h  1048h Blowfish for DS-mode
```

The 3DS does have only half of the DSi keys (the extra keys might be used for DSi debug version, but aren't need for normal DSi software).

The 40h-byte area for ROM:0000B5D8h can be fully dumped from 3DS ITCM, the same vales should also exist in DSi ITCM, but the DSi zerofills a 10h-byte fraction of that area after initialization, and it doesn't seem be possible to read that values via Main Memory hacks (most of the missing values can be found in AES keyslots though).

The A0h-byte area is found only in 3DS ITCM, it should also exist somewhere in DSi ROM, but isn't relocated to DSi ITCM (however, the relevant values can be found in AES keyslots, eg. the "Nintendo" string).

**Checksums for BiosDSi.rom (20000h bytes)**

```
Offset Size  CRC32
00000h 8000h 5434691Dh               ;\
08000h 188h  ?                       ;
08188h 180h  E5632151h  (not 3ds)    ;
08308h 80h   64515306h               ;
08388h 3250h ?                       ;
0B5D8h 20h   85BE2749h               ; ARM7
0B5F8h 10h   180DF59Bh  (3ds only)   ;
0B608h 10h   E882B9A9h               ;
0B618h 10B8h ?                       ;
0C6D0h 1048h 3B5CDF06h               ;
0D718h 1048h 5AC363F9h  (not 3ds)    ;
0E860h 18A0h ?                       ;/
10000h 8000h 11E7C1EAh               ;\
18000h 7F4h  ?                       ;
187F4h 200h  4405D4BAh               ;
189F4h 200h  2A32F2E7h  (not 3ds)    ;
18BF4h D2Ch  ?                       ; ARM9
19920h 80h   2699A10Fh               ;
199A0h 1048h A8F58AE7h               ;
1A9E8h 1048h E94759ACh  (not 3ds)    ;
1BA30h 45D0h ?                       ;/
?      A0h   180DF59Bh  (3ds only)   ;-whatever, "Nintendo" string etc.
```

Checksums for the 'whole' 20000h-byte file (with unknown/missing areas zerofilled):

```
180DF59Bh (tcm/ram dump)  (missing 10h bytes)
03A21235h (3ds dump)      (missing 180h+200h+1048h+1048h bytes)
CDAA8FF6h (combined dump) (missing only the unknown "?" areas)
```

**DSiWifi BIOS**

The Wifi BIOS can be dumped by using the WINDOW_DATA register via SDIO CMD53.

**Further DSi BIOSes**

The DSi cameras and several other I2C/SPI devices are probably having BIOS ROMs, too. Unknown if/how that ROMs are dumpable.