Nisha SpareTable Specification

Preliminary

Version 0.1-0

Oct. 3, 1984

```
Structure ::= (<Fence {bytes/$00:03; length = 4}>
               <RunNumber {bytes/$04:07; length = 4}>
               <Format_Offset {byte/$08; length = 1}>
               <Format_InterLeave {byte/$09; length = 1}>
               <HeadPtr_Array {bytes/$0A:49; length = 64}>
               <SpareCount {byte/$4A; length = 1}>
               <BadBlockCount {byte/$4B; length = 1}>
               <BitMap {bytes/$4C:55; length = 10}>
               <Heap {bytes/$56:185; length = 304}>
               <InterLeave_Map {bytes/$186:1A5; length = NumberOfSectors}>
               <CheckSum {bytes/$1A6:1A7; length = 2}>
               <Fence {bytes/$1A8:1AB; length = 4}>
               <Zone_Table {bytes/$1AC:1C3; length = 24}>
               <Fence {bytes/$200:203; length = 4}> )
```

$1C4 : 6 bytes for LiIA ~~personality~~
$1CA : personality

```
Fence ::= (<$F0> <$78> <$3C> <$1E> )
```

```
RunNumber ::= <32-bit integer>
```

This integer is incremented once each time the spare table is written to to the disk.Because two copies are kept on the the disk, the RunNumber is used to indicate which is the more recent of the two, should both copies not be updated.

```
Format_Offset ::= <0..NumberOfSectors>
```

Format_Offset is the number of physical sectors there are from index mark until logical sector 0. On Nisha, this value will always be $00 indicating that logical sector 0 always follow the index mark.

```
Format_InterLeave ::= <0..6>
```

This number is the interleave factor for this disk and is used in calculating where each of the logical sectors are relative to actual sector locations. On Nisha, this value will always be $01 indicating a physical interleave of 2:1.

```
HeadPtr_Array ::= <ARRAY[0..127] of HeadPtr

   HeadPtr ::= <Nil+Ptr>
                    Nil ::= <$80 {if Nil the end-of-chain}>
```

```
                              Ptr ::= <$00..$7F {address of next element}>
                                          A Ptr is a 7-bit structure that 'points' to
a
                                               specific location within the Heap. To
arrive
                                           at the actual index value within the Heap,
the
                                               Ptr must first be multiplied by 4 {the
length
                                          of each element}.
```

When a disk is formatted and being written to for the first time, each logical block is assigned the first available physical block on the disk. Therefore you would expect that LogicalBlock(0) would occupy PhysicalBlock(0), L(1) --> P(1), etc. There are instances, however, when a block of data must be relocated to another space on the disk that does not follow the original progression (for example, the original space was defective). In order to 'find' these relocated blocks in the future a record must be kept as to where all these relocated blocks have been put. This record takes the form of 128 linked lists having the form:

HeadPtr[n] --> LinkedList[n], where n ::= [0..127]

The algorithm for deciding whether or not a logical block has been relocated is to extract bits 10:15 from the LogicalBlockNumber and use it as an index into the HeadPtrArray:

```
    IF (HeadPtr[LogicalBlockNumber/bits 10:15].Nil)
      THEN LogicalBlock has not been relocated
      ELSE use HeadPtr[].Ptr to begin searching the chain for a matching
             element {refer to the structure of ListElement for more detail}
        IF no matching ListElement
          THEN LogicalBlock has not been relocated
               ELSE the element position in the Heap corresponds to the new
physical
                block location
```

```
  SpareCount ::= <$00..$4B>

  BadBlockCount ::= <$00..$4B>

  BitMap ::= <ARRAY[$00..$4B] of Bits>
                         The bit map is used to keep a record of which spare blocks
are
                      occupied.

  Heap ::= <ARRAY[$00..$4B] of ListElement>

      ListElement ::= (<Nil+Used+Useable+Spr_Type+Data_Type>
                          <Token>
```

```
                    <Ptr>)

          Used ::= <$40>
          Useable ::= <$20>
          Spr_Type ::= <Spare|BadBlock>
             Spare ::= <$10>
             BadBlock ::= <$00>
          Data_Type ::= <Data|SpareTable>
             Data ::= <$02>
             SpareTable ::= <$08>

          Token ::= <Bits 0:9 of LogicalBlock>

InterLeave_Map ::= <ARRAY[0..15] of [0..NumberOfSectors]>
```

The InterLeave_Map is used to logical re-interleave the drive so that Widget can be run optimally on any system without having different manufacturing or formatting processes.

```
Check_Sum ::= <sum of all bytes in the spare table from the first fence to
                    beginning of this structure, in MOD-65536 arithmetic>


Zone_Table ::= <ARRAY[0..NumberOfZones] of Zone_Element>

        Zone_Element ::= <Offset_Direction+Offset_Magnitude>
```

The Zone_Table is used in improving the performance of the positioning system. In the case where the drive develops a non-negligible amount of fine positioning offset (as in the case of high or low temperature) seeks with manual offset are used. The amount to manually offset (and direction) is kept in the Zone_Table. Refer to the ***Nisha Operation Summary*** Specification for further details.

Finding the SpareTable on Nisha

The SpareTable on Nisha is allowed to occupy any of the blocks reserved for sparing {there is nothing unique about the media location that the SpareTable is written at and therefore is subject to the same probabilities for defects and handling errors as any other block of data}. Because of this, when the drive is powered up the SpareTable can not be counted on to be residing in any specific location and must be searched for.

SpareBlocks are located every 512 physical blocks apart, beginning with physical block 512. Because Nisha has 2 tracks per surface and 32 sectors per track, SpareBlocks can easily be located by noting that they reside on sector 0, {the sector following Index} head 0 of every eigth track. To actually find the SpareTable, each spare block must be read to see if it is one of the SpareTable blocks; in order to not confuse a block of data with a SpareTable block several data structures need to match:

1. the fence residing at address 0:4
2. the fence residing at address $1A8:1AB
3. the fence residing at address $200:203
4. the checksum must match

After all SpareTables have been found {it is quite possible to have more than 2 copies of legitimate SpareTables on the disk at any given point in time; the RunNumber decides which is the most current} the 2 that are themost current are updated {RunNumber incremented by one, and Write/Verified to the disk}. In the case where only one SpareTable is found, then a second one is generated with the same RunNumber.

```
Find_SpareTable;
 begin
  while Not_All_SpareBlocks_Have_Been_Searched do begin
    read a spareblock using full recovery methods
    if The_Block_Can_Be_Read then
       begin
         if (bytes 0:4 = fence) and
            (bytes $1A8:1AB = fence) and
            (bytes $200:203 = fence) and
            (bytes $1A6:1A7 = calculated check sum) then
          begin
            this block is a SpareTable
            increment the count of SpareTables found
            if (more than one SpareTable has been found) and
               (the RunNumber of the last SpareTable found is greater
                then the RunNumber of the previous SpareTable)
              then latest SpareTable found is valid and should replace
                   any previous version found
          end {if-then}
       end {if-then}
    end {while-do}
    if At_Least_One_Copy_Of_The_SpareTable_Has_Been_Found
      then increment the RunNumber and Write/Verify back to the disk
      else the drive should not be used
 end.
```

Updating the SpareTable Structure

The SpareTable keeps track of two data structures: SpareBlocks and BadBlocks (SpareBlocks are sectors that have been remapped, while BadBlocks are sectors whose data can not be recovered and are logged until the next write to that sector when they will become candidates for remapping). By definition a SpareTable is a SpareBlock, a SpareBlock may be either of type UserData or SpareTableData, and a BadBlock is always of type UserData and never occupies an actual sector (it is just logged in the SpareTable).
Basically, the overall structure of the SpareTable is that of a series of singly linked lists. These linked lists are pointed to by a HeadPtr which is kept in an 64 element array of HeadPtrs. To locate a block within the SpareTable an algorithm is used consisting of finding the correct linked list to search (indexing to the proper HeadPtr) and then matching DataType (SpareTable vs. UserData), SpareType (SpareBlock vs. BadBlock), Useable (has the block been remapped several times?), and a portion of the block number being searched for. Once a block has been found in the SpareTable, its new physical address is a function of its position within the SpareTable (i.e., if it occupied the 1st location in the SpareTable then its address would be 1x512, the 2nd position --> 2x512, etc.). Adding and deleting elements from the SpareTable is a matter of manipulating the linked lists: lists that have thier last element deleted must change the HeadPtr for that list to reflect an empty list; BadBlocks can be deleted while SpareBlocks are never deleted so removing an element from the middle of a list must be handled; and of course adding onto the end of a list is common.

Ex.

Lets assume that Logical Block $32AA is to be searched for in the SpareTable.

```
Function SearchSpareTable(LogicalBlock : BlockNumber;

var
 PhysicalBlock : BlockNumber;
 IsData : Boolean;
 var IsSpare : Boolean{output}) : Boolean

begin
index := most significant 6 bits of LogicalBlock
if HeadPtr[index].NIL then
   PhysicalBlock := LogicalBlock + (LogicalBlock DIV 512)
   if (PhysicalBlock DIV 512) <> (LogicalBlock DIV 512)
   then PhysicalBlock := PhysicalBlock + 1
   SearchSpareTable := false  {LogicalBlock not found}
else
   ptr := HeadPtr[index].ptr * 4 {calculate effective address within
                                   Heap}
   if ptr^.Useable and
     ((ptr^.Data_Type = UserData) = IsData) and
```

```
  (ptr^.Token = bits 0:9 of LogicalBlock) then
     PhysicalBlock := (index+1)*512
     if (ptr^.Spr_Type = Spare)
        then IsSpare := true
        else IsSpare := false
SearchSpareTable := true {LogicalBlock found}
```