Nisha Operation Summary

Preliminary

Written by Rodger Mohme
Orchard ParkWay MS-190 x4879

Version 0.3-0

Dec. 3, 1984

TABLE of CONTENTS

INTRODUCTION

This document is intended to specify the recommended use of the 20Mbyte Nisha Winchester disk drive as defined by its designers. Furthermore, the Nisha FST {Final System Test} is designed to qualify the drive based on the criteria put forth within the following pages.

The Nisha disk drive is a microprocessor controlled Winchester device that is composed of mainly two (2) units: the servo processor/control, and the read/write channel. The servo processor is communicated with via a serial link operating at 58.6k baud and is used to position the read/write heads over a specific area of the disk. Both course and fine positioning are controlled by this mechanism. The read/write channel controls the head signals via a parallel interface with the host. All data into and out of the read/write channel is clocked NRZ.

This operation description is written in a short-hand psuedo code where a particular procedure/function fragment is defined with its inputs, outputs, and any functional results. No attempt has been made to specify actual structures {except the sparetable} or any controller specific parameters.

SUPPORT PROCEDURES

- o SendSrvo
- o RcvSrvo
- o SrvoCmnd
- o SrvoStat
- o UpDate_Current_Cylinder
- o Restore
- o ServoOk
- o ServoRecovery
- o DoAutoOffset

```
SendSrvo(Inputs: Difcnth, Difcntl, Statreg, Cntreg
         Outputs: None
         Result: BOOLEAN, True if successful)

1: CheckByte := Ones_Complement(Difcnth+Difcntl+Statreg+Cntreg)
   set watch dog timer for 1 second
   LOOP {send Difcnth, DifCntl, Statreg, Cntreg}
      DO nothing WHILE NOT(SioRdy) AND watch dog timer hasn't expired
      WHEN SioRdy THEN send a byte to the servo, beginning with Difcnth
   UNTIL (all input bytes + checkbyte) are sent to servo OR timeout
   reset watch dog timer
   IF timeout
      THEN SendSrvo := False
      ELSE
         wait for 250 usec    {allow servo to check the checkbyte}
         IF NOT(SioRdy)
            THEN SendSrvo := True
            ELSE SendSrvo := False

RcvSrvo(Inputs: None
        Outputs: 4 bytes of servo status need to be stored somewhere
        Result: BOOLEAN, True if successful)

1: set watch dog timer for 1 second
   LOOP
      DO nothing WHILE waiting for a byte from UART AND  NOT(Timeout)
      receive byte
   UNTIL (4 status bytes + checkbyte) are received from servo OR timeout
   reset watch dog timer
   IF timeout
      THEN SendSrvo := False
      ELSE
         IF Calculated checkbyte = received checkbyte)
            THEN SendSrvo := True
            ELSE SendSrvo := False
```

```
SrvoCmnd(Inputs: Difcnth, Difcntl, Statreg, Cntreg
          Outputs: None
          Result: BOOLEAN, True if successful)

 1: Retries : = 0
    WHILE (Retries < 10) AND NOT(SendSrvo(Difcnth, Difcntl, Statreg, Cntreg)) DO
       Retries = Retries + 1
    IF (Retries = 10)
       THEN
          Servo Initialization      {re-init the servo processor}
          IF SendSrvo(40, 00, 00, 00) {data recal}
             THEN IF SendSrvo(DifCnth, DifCntl, Statreg, Cntreg)
                     THEN SrvoCmnd : = True
                     ELSE SrvoCmnd : = False
             ELSE SrvoCmnd : = False
       ELSE SrvoCmnd : = False

 SrvoStat(Inputs: Cntreg
          Outputs: Status  {4 bytes}
          Result: BOOLEAN, True if successful)

 1: Retries : = 0
    WHILE (Retries < 4) AND NOT(SrvoCmnd(0, 0, 0, Cntreg) follwed by an immediate
                                RcvSrvo(Status)) DO
       Retries = Retries + 1
    IF (Retries = 4)
       THEN
          Servo Initialization      {re-init the servo processor}
          IF SendSrvo(40, 00, 00, 00) {data recal}
             THEN IF (SendSrvo(0, 0, 0, Cntreg), RcvSrvo(Status))
                     THEN SrvoStat : = True
                     ELSE SrvoStat : = False
             ELSE SrvoStat : = False
       ELSE SrvoStat : = True
```

```
GoodHdr(Inputs: none
        Outputs: Actual_Cylinder
        Result: BOOOLEAN, True if successful)

        1: TempBool := ReadHeader(RwStat, RwErrCount)
           IF (Data.HiCylinder = NOT(Data. InvertHiCylinder)) AND
              (Data.LoCylinder = NOT(Data. InvertLoCylinder))
              (Data.HeadSector = NOT(Data. InvertHeadSector))
              THEN
                 Actual_Cylinder := Data.Cylinder
                 GoodHdr := True
              ELSE
                 Repeat the above sequence 8 times or until successful. Wait for
                 100 msec before each retry to try to minimize external
                 noise/vibration, and to get a new sector each time.
                 IF all retries fail
                    THEN GoodHdr := Unsuccessful

UpDate_Current_Cylinder(Inputs: none
                        Outputs: Current_Cylinder
                        Result: BOOLEAN, True if successful)

  1: IF GoodHdr(Actual_Cylinder)
        THEN
           Current_Cylinder := Actual_Cylinder
           UpDate_Current_Cylinder := Successful
        ELSE UpDate_Current_Cylinder := Unsuccessful
```

```
Restore(Inputs: RecalType {either DataRecal or BrakeRelease value}
                ReadHeader {boolean, indicates whether to update actual position}
        Outputs: None
        Result: BOOLEAN, True if successful)

    Constants:
        DataRecal = $40
        BrakeRelease = $70
        Max_Drive_Cylinders = 636
        Recal_Cylinder = 600

1: Temp : = SendSrvo(RecalType, 0, 0, 0) {assume recals work}
   IF (RecalType = BrakeRelease)
      THEN Current_Cylinder : = Max_Drive_Cylinders
      ELSE Current_Cylinder : = Recal_Cylinder
   Set watch dog timer for 2 seconds
   DO nothing WHILE NOT(ServoReady) AND NOT(Timeout)
   IF Timeout
      THEN Restore : = Unsuccessful
      ELSE
         IF ReadHeader
            THEN IF UpDate_Current_Cylinder
                    THEN Restore : = Successful
                    ELSE Restore : = Unsuccessful
            ELSE Restore : = Unsuccessful
```

```
ServoOk(Inputs: none
         Outputs: None
         Result: BOOLEAN, True if successful)

1: IF ServoError {hardware line on Nisha interface}
     THEN
        IF ServoReady {hardware line on Nisha interface}
           THEN Temp := SrvoStat(1, DontCare) {just read a status to clear error}
           ELSE Temp := Restore(DataRecal)
     ELSE IF NOT(ServoReady)
           THEN
              Servo_Initialization
              Restore(DataRecal)
   IF ServoReady AND NOT(ServoError)
      THEN ServoOk := Successful
      ELSE ServoOk := Unsuccessful
```

```
ServoRecovery(Inputs: none
              Outputs: None
              Result: BOOLEAN, True if successful)

1: Retries := 8
   ServoRecovery := Unsuccessful
   REPEAT
      IF ServoOk
         THEN
            IF UpDate_Current_Cylinder
               THEN
                  IF Current_Cylinder = Cylinder
                     THEN ServoRecovery := successful
                     ELSE Seek(Cylinder, Head, Sector, Offset_On) {reseek back to
                                          the same location}
         ELSE Retries := Retries - 1
   UNTIL (Retries = 0) OR ServoRecovery is successful
```

```
DoAutoOffset(Inputs: none
             Outputs: None
             Result: none)
  Constants:
    Access_Offset = $90
    Offset_Auto = $40

1: SelectHead(1)  {do all servoing off of head 1}
    Temp : = SrvoCmnd(Access_Offset, 0, Offset_Auto, 0)
    Temp : = SrvoStat(1, SrvoStatus)  {get direction and magnitude of offset}
    Off_Dir : = (SrvoStatus[byte1]/bit5)
    Off_Mag : = NOT(SrvoStatus[byte1]/bits0:4) {magnitude comes back inverted}
    Off_DirMag : = Invert(Off_Dir) merged with Off_Mag {this value is kept in the
                zone table for manual offsetting; the value to offset will be the
                opposite of the direction just read in and the same magnitude}
    SpareTable.ZoneTable[Cylinder DIV 32] : = Off_DirMag {update zone table}
```

SEEKING

```
Seek(Inputs: Cylinder, Head, Sector {new disk location}
             WithOffset : BOOLEAN  {is seek with auto offset?}
     Outputs: none
     Result: none  {if it fails then positioning system is non_functional})

   Constants:
     Access = $80
     Access_Offset = $90
     Forward = $08
     Reverse = $00
     Offset_Auto = $40
     Offset_Manual = $80


1: REPEAT
       Select_Head(1) {do all servoing stuff on head 1}
       IF ServoOk
          THEN
             Seek_Magnitude := Current_Cylinder - Cylinder
             IF (Seek_Magnitude < 0)
                THEN
                   Seek_Magnitude := -1 * Seek_Magnitude
                   Seek_Direction := Forward {towards ID}
                ELSE Seek_Direction := Reverse {away from ID}
             IF (Seek_Maginitude <> 0)
                THEN
                   IF WithOffset
                      THEN
                         Difcnth := (Access_Offset + Seek_Direction +
                                     Seek_Magnitude DIV 256)
                         Difcntl := Seek_Magnitude MOD 256
                         StatReg := Offset_Auto
                      ELSE
                         IF the magnitude of SpareTable.ZoneTable[Cylinder DIV
                            32] is greater than 10
                            THEN
                               Off_Mag  :=  magnitude  of  SpareTable.ZoneTable
                                            [Cylinder DIV 32]
                               Off_Dir  :=  direction  of  SpareTable.ZoneTable
                                            [Cylinder DIV 32]
                                Difcnth := (Access_Offset + Seek_Direction +
                                            Seek_Magnitude DIV 256)
                               Difcntl := Seek_Magnitude MOD 256
                               StatReg := Offset_Manual+Off_Dir+Off_Mag
                            ELSE
                               Difcnth := (Access + Seek_Direction +
                                            Seek_Magnitude DIV 256)
                               Difcntl := Seek_Magnitude MOD 256
                               StatReg := 0
                   IF  SendSrvo(Difcnth, Difcntl, Statreg, 0)
```

```
                    THEN Seek : = Successful
                    ELSE Seek : = Unsuccessful
               ELSE Seek : = Successful
        ELSE Seek : = Unsuccessful
UNTIL either the seek is successful OR is retried 4 times
IF seek is unsuccessful
   THEN
      Servo Initialization
      Restore(DataRecal, ReadHeader)
      Try the code inside the REPEAT UNTIL loop once more
IF seek is unsuccessful
   THEN Non_Functional_Drive
   ELSE
      Current_Cylinder : = Cylinder
      Select_Head(Head)
      Current_Sector : = Sector
      IF   (Current_Cylinder   <   PcRwi_Cylinder)   {check   for   changing
                              WritePreComp/ReducedWriteCurent}
         THEN PcRwi : = Inactive {active low}
         ELSE PcRwi : = Active
      IF Not a zero track seek
         THEN SeekCount : = SeekCount + 1 {keep track for arm sweeping, refer
                           to ArmSweep in the MISCELLANEOUS section}
```

RESET/POWER—ON {start up condiserations}

o Drive SpinUp and Motor speed checking
o Servo Initialization
o Brake release
o Servo Test
o Sector count
o Read/Write checking

Drive SpinUp and Motor speed checking

```
1: IF time from index_mark to index_mark is 21 usec +/- 5 usec
       THEN Servo Initialization
       ELSE
          wait for 20 seconds
          IF time from index_mark to index_mark is 21 usec +/- 5 usec
             THEN Servo Initialization {continue}
             ELSE Non_Functional_Drive(Motor is at the incorrect speed)
```

Servo Initialization

```
1: assert Servo Reset for 1000 usecs
   wait for 1 second to allow the servo to complete its init code
   IF SendSrvo(00, 00, 00, 01) {arbitrary, non-destructive command to test servo}
                               {processors ability to communicate}
   THEN IF RcvSrvo            { don't care what response means, just that bytes}
                               {come back and that the checkbyte is correct}
          THEN Release Brake {continue}
          ELSE Non_Function_Drive(servo will not respond after reset)
```

Release Brake

```
1: IF Restore(ReleaseBrake, DoNotReadHeader)     {send brake release command}
     THEN Servo Test {continue}
     ELSE Non_Functional_Drive
```

Servo Test

```
1: IF NOT(Restore(DataRecal, DoNotReadHeader))    {send data recal command}
     THEN Non_Functional_Drive
     ELSE
        Seek($0069, 00, 00, NoOffset) {see if servo will go thru motions of a seek,
                                       if this fails, it will fail in Seek}
        SectorCount {continue}
```

SectorCount

```
1:  IF  NOT(NumberOfSectors = 32) {count the number of sectors between one
                                   rotation of Index mark}
     THEN Non_Functional_Drive
     ELSE
        ReadWrite Test
```

ReadWrite Test

Constants:
SlfTstTrack = 612

```
1: IF NOT(Restore(DataRecal, ReadHeader))
     THEN Either Non_Functional_Drive OR drive has not been formatted
     ELSE
        Seek(SlfTstTrack, 0, 0, AutoOffset) {if this fails it will be in Seek}
        1a:

1a: DataPattern : = $39
   IF Write
      THEN
         zero memory buffer area where data is to be read
         IF Read
            THEN IF DataPattern : = $39
                  THEN Find_SpareTable {refer to the section on SPARETABLE
                                        MANIPULATION}
                  ELSE 1b:
      ELSE 1b:

1b: Repeat 1a: for all sectors on the track or until successful
   IF not successful
      THEN Non_Functional_Drive
      ELSE Find_SpareTable {this can fail in Find_SpareTable, if not then enter
                        an idle state and begin taking commands from the
                        system}
```

SPARETABLE MANIPULATION

- GetNewSpare
- Add Spare
- Delete BadBlock
- Spare
- Search SpareTable
- UpDate SpareTable
- Find SpareTable

```
GetNewSpare(Inputs: LogicalBlkNum : 17 bits
              Outputs: SpareNum
              Result: none)
  Constants:
    MaxSpares = 76

  1: SpareNum := LogicalBlkNum DIV 512 {spare are 512 blocks apart}
     IF SpareTable.BitMap[SpareNum] {bitmap is active high if filled}
        THEN
           Done := False
           NoHighs := False
           NoLows := False
           LoTemp := SpareNum {search for the next closest spare location}
           HiTemp := SpareNum
           REPEAT
              LoTemp := LoTemp - 1
              HiTemp := HiTemp + 1
              IF (LoTemp < 0) THEN NoLows := True
              IF (HiTemp >= MaxSpares) THEN NoHighs := True
              IF NOT(NoHighs) AND NOT(SpareTable.BitMap[HiTemp])
                 THEN
                    SpareNum := HiTemp
                    Done := True
                 ELSE
                    IF NOT(NoLows) AND NOT(SpareTable.BitMap[LoTemp])
                       THEN
                          SpareNum := LoTemp
                          Done := True
              IF NoHighs AND NoLows THEN Done := True
           UNTIL Done
           IF NoHighs AND NoLows THEN Non_Functional_Drive
```

```
Add_Spare(Inputs: LogicalBlkNum : 17 bits,
                  SpareNum : Byte,
                  IsSpare : BOOLEAN {true is a spare, false if a badblock}
          Outputs: none
          Result: none)

 Constants:
  NIL = $80
  Used = $40
  Useable = $20
  Spare_Type = $10
  BadBlock_Type = $00
  User_Type = $02
  SprTbl_Type = $08

 1: IF SpareTable.HeadPtr[LogicalBlkNum DIV 1024].NIL
       THEN
          SpareTable.HeadPtr[LogicalBlkNum DIV 1024]. NIL := False
          SpareTable.HeadPtr[LogicalBlkNum DIV 1024].Ptr := SpareNum
       ELSE
          search for the end of the list
          remove NIL flag from last element of list
          ptr of last element of list := SpareNum
    SpareTable.Heap[4*SpareNum] := NIL+Used+Useable
    IF IsSpare
       THEN SpareTable.Heap[4*SpareNum] := Spare_Type
       ELSE SpareTable.Heap[4*SpareNum] := BadBlock_Type
    IF LogicalBlkNum is a User_Block {as opposed to a sparetable block}
       THEN SpareTable.Heap[4*SpareNum] := User_Type
       ELSE SpareTable.Heap[4*SpareNum] := SprTbl_Type
    SpareTable.Heap[(4*SpareNum)+1)] := bits 9:8 of LogicalBlkNum
    SpareTable.Heap[(4*SpareNum)+2] := bits 7:0 of LogicalBlkNum
    SpareTable.BitMap[SpareNum] := True
```

```
Delete_BadBlock(Inputs: LogicalBlkNum : 17 bits
                Outputs: none
                Result: none)

 1: Temp := LogiccalBlkNum DIV 1024
    IF SpareTable.Heap[4*Temp].NIL
       THEN {badblock element is first on list}
          SpareTable.HeadPtr[Temp].NIL := True
          SpareTable.HeadPtr[Temp].Ptr := 0
       ELSE
          search list for bad block element, Temp1 := sparenum of element just
          before the badblock element, Temp := sparenum of badblock element
          IF the badblock element is the last element of the list
             THEN SpareTable.Heap[(4*Temp1)].Nil := True
             ELSE SpareTable.Heap[(4*Temp1)+4)].Ptr :=
                                  SpareTable.Heap[4*(Temp+4)].Ptr
    SpareTable.Heap[4*Temp] := $FF {mark the heap that a badblock was here}
    SpareTable.Heap[(4*Temp)+1] := $FF
    SpareTable.Heap[(4*Temp)+2] := $FF
    SpareTable.Heap[(4*Temp)+3] := $FF
```

```
Spare(Inputs: LogicalBlkNum : 17 bits,
              IsSpare : BOOLEAN {true is a spare, false if a badblock}
      Outputs: none
      Result: none)

 1: IF NOT(IsSpare)
       THEN
          IF block to be spared is already a badblock
             THEN
                Delete_BadBlock(LogicalBlkNum)
                SpareTable.BadBlockCount := SpareTable.BadBlockCount - 1
       Done := False
       LoopCount := 10
       ReadErrs := 0
       REPEAT
          IF NOT(Write)
             THEN Done := True {exit loop on any write error}
             ELSE
                IF NOT(Read)
                   THEN
                      IF RwStat.HeaderErr
                         THEN Done := True {exit if any header errors}
                         ELSE ReadErrs := ReadErrs + RdErrCnt
          LoopCount := LoopCount - 1
       UNTIL Done OR (LoopCount = 0)
       IF ((LoopCount <> 0) OR (ReadErrs > 3))
          THEN
             IF NOT(IsSpare) AND NOT(block is already a bad block)
                THEN
                   GetNewSpare(LogicalBlkNum, SpareNum)
                   AddSpare(LogicalBlkNum, SpareNum)
                   SpareTable.BadBlockCount := SpareTable.BadBlockCount + 1
                ELSE
                   IF IsSpare
                      THEN
                         Done := False
                         REPEAT
                             IF block to be spared is already spared
                                THEN
                                   Temp := sparenum of spared block heap element
                                   SpareTable.Heap[4*Temp].Useable := False
                            SpareTable.SpareCount := SpareTable.SpareCount + 1
                            GetNewSpare(LogicalBlkNum, SpareNum)
                            AddSpare(LogicalBlkNum, SpareNum)
                            PBlk := SpareNum * 512
                            SprCyl := PBlk DIV 64
                            SprHd := PBlk MOD 64 DIV 32
                            SprSctr := PBlk MOD 64 MOD 32
                            Seek(SprCyl, SprHd, SprSctr, OffSet_On)
                            IF Write AND Read THEN Done := True
                         UNTIL Done
       UpDate_SpareTable
```

```
Search_SpareTable(Inputs: LogicalBlkNum : 17 bits
                  Outputs: IsSpare, valid only if Result is true
                  Result: BOOLEAN, True if spare or badblock)
  Constants:
    NIL = $80

  1: Temp : = LogicalBlkNum DIV 1024
     IF SpareTable.HeadPtr[Temp].NIL
        THEN Search_SpareTable : = False
        ELSE
           Found : = False
           Done : = False
           Temp : = SpareTable.HeadPtr[Temp].Ptr
           REPEAT
              IF SpareTable.Heap[4*Temp].Useable AND
                   (SpareTable.Heap[(4*Temp)+1)] = LogicalBlkNum/bits 9:8) AND
                   (SpareTable.Heap[(4*Temp)+2)] = LogicalBlkNum/bits 7:0)
                 THEN
                    Done : = True
                    Found : = True
                    IsSpare : = SpareTable.Heap[4*Temp].Spr_Type
                 ELSE
                    IF SpareTable.Heap[4*Temp].NIL
                       THEN Done : = True
                       ELSE Temp : = SpareTable.Heap[(4*Temp)+3]
           UNTIL Done
           IF Found
              THEN Search_SpareTable : = True
              ELSE Search_SpareTable : = False
```

```
UpDate_SpareTable(Inputs: none
                  Outputs: none
                  Result: none)

 1: SpareTable.RunNumber := SpareTable.RunNumber + 1
    calculate a new checksum by summing all bytes between the start of the
          sparetable and the checksum (without carry) into a 16-bit integer.
          Place the new checksum into the SpareTable
    Done := False
    SprBlk := 0
    REPEAT
       move the SpareTable into the sequencer write buffer and place a fence at
          data locations 512:515 (fence = $F0783C1E)
       IF (SprBlk = 0)
          THEN LBlk := $32AB
          ELSE LBlk := $6553
       PBlk := Search_SpareTable(LBlk, IsSpare)
       SprCyl := PBlk DIV 64
       SprHd := PBlk MOD 64 DIV 32
       SprSctr := PBlk MOD 64 MOD 32
       Seek(SprCyl, SprHd, SprSctr, OffSet_On)
       IF Write AND Read
          THEN
             SprBlk := SprBlk + 1
             IF (SprBlk > 2) THEN Done := True
          ELSE Spare_Block(LBlk, True)
    UNTIL Done
```

```
Find_SpareTable(Inputs: none
                Outputs: none
                Result: none)

  1: clear SpareTable.ZoneTable to all zeros
     LoopCount := 76 {search all spare blocks for both spare tables}
     SprTblCount := 0
     CurrentRunNumber := 0
     REPEAT
        PBlk := 512 * LoopCount {start at ID, works towards OD}
        SprCyl := PBlk DIV 64
        SprHd := PBlk MOD 64 DIV 32
        SprSctr := 0
        Seek(SprCyl, SprHd, SprSctr, OffSet_On)
        IF Read OR if read error is not header error and RdErrCnt < 10
           THEN
              IF there is a fence at databuffer[512:515] AND
              there is a fence at databuffer[0:3] AND
              there is a fence at databuffer[424:427] AND
              the checksum over the databuffer matches the checksum in the
               databuffer
                 THEN
                    IF (the runnumber in the data buffer >= CurrentRunNumber)
                       THEN
                          move the databuffer to the spare table buffer
                          CurrentRunNumber := SpareTable.RunNumber
                          SprTblCount := SprTblCount + 1
     UNTIL (LoopCount = 0)
     IF (SprTblCount = 0)
        THEN Non_Functional_Drive
        ELSE UpDate_SpareTable
     clear SpareTable.ZoneTable to all zeros
```

MISCELLANEOUS

o Arm Sweep
o Parking
o FreeProcess/SelfTest

Arm_Sweep(Inputs: none
Outputs: none
Result: none)

{Am Arm_Sweep is to be performed every 2000 seeks {or so} to keep the rotary arm bearings lubricated. It is not critical that it occur *exactly* on the 2000th seek, but is absolutely essential to the performance and reliability of the drive. The Nisha Final System Test checks to see an Arm_Sweep is needed after every Logical Command from the host system.}

1: Seek($0069, 0, 0, Offset_Off)
Seek($026C, 0, 0, Offset_Off)
Seek(Cylinder, Head, Sector, Offset_On)

Parking(Inputs: none
Outputs: none
Result: none)

{Parking refers to the action of removing the drives r/w heads from the medias data area while idle to minimize the chances of contact between the heads and media that could destroy data or the drive. Nisha is equiped with a hardware parking device {the brake} that will automatically move the heads to the park position when power is removed from the drive; however, it is still recommended that the drive be 'parked' when idle. the Nisha Final System Test will park the drive after 8 seconds of no activity.}

1: Seek($027C, 0, 0, Offset_Off)

FreeProcess/SelfTest

It is recommended while the Nisha is idle that each of its selftest routines be performed continuously at regular intervals to minimize the chances that a drive failure will not be detected until a users data is involved. The Nisha Final System Test performs the following tests {in this order} every 8 seconds that the Nisha is idle {except for the first 8 seconds, when it parks}:

1. RamTest {test buffers that data is read/written from/to}
2. MotorSpeed Test
3. Sector Count Test
4. ReadWrite Test {incorporates the Servo Test}