The Discover process example implementation is out of date with respect to the recent changes/additions to the discover process.

Recommended changes

Replace K.2 with the following text;

```c
// SASDiscoverSimulation.h

// assume the maximum number of phys in an expander device is 128
#define MAXIMUM_EXPANDER_PHYS 128

// assume the maximum number of indexes per phy is 128
#define MAXIMUM_EXPANDER_INDEXES 128

// limit to 8 initiators for this example
#define MAXIMUM_INITIATORS 8

// defines for address frame types
#define ADDRESS_IDENTIFY_FRAME 0x00
#define ADDRESS_OPEN_FRAME 0x01

// defines for SMP frame types
#define SMP_REQUEST_FRAME 0x40
#define SMP_RESPONSE_FRAME 0x41

// defines for SMP request functions
#define REPORT_GENERAL 0x00
#define REPORT_MANUFACTURER_INFORMATION 0x01
#define DISCOVER 0x10
#define REPORT_PHY_ERROR_LOG 0x11
#define REPORT_PHY_SATA 0x12
#define REPORT_ROUTE_INFORMATION 0x13
#define CONFIGURE_ROUTE_INFORMATION 0x90
#define PHY_CONTROL 0x91
#define PHY_TEST 0x92

// defines for the protocol bits
#define SATA 0x01
#define SMP 0x02
#define STP 0x04
#define SSP 0x08

// defines for open responses, arbitrary values, not defined in the spec
#define OPEN_ACCEPT 0
#define OPEN_REJECT_BAD_DESTINATION 1
#define OPEN_REJECT_RATE_NOT_SUPPORTED 2
#define OPEN_REJECT_NO_DESTINATION 3
#define OPEN_REJECT_PATHWAY_BLOCKED 4
#define OPEN_REJECT_PROTOCOL_NOT_SUPPORTED 5
#define OPEN_REJECT_RESERVE_ABANDON 6
#define OPEN_REJECT_RESERVE_CONTINUE 7
#define OPEN_REJECT_RESERVE.Initialize 8
#define OPEN_REJECT_RESERVE_STOP 9
#define OPEN_REJECT_RETRY 10
#define OPEN_REJECT_STP_RESOURCES_BUSY 11
#define OPEN_REJECT_WRONG_DESTINATION 12
#define OPEN_REJECT_WAITING_ON_BREAK 13
```
// definitions for discovery algorithm use
enum
{
    SAS_SIMPLE_LEVEL_DESCENT = 0,
    SAS_UNIQUE_LEVEL_DESCENT
};

enum
{
    SAS_10_COMPATIBLE = 0,
    SAS_11_COMPATIBLE
};

// definitions for SMP function results
enum SMPFunctionResult
{
    SMP_REQUEST_ACCEPTED = 0,       // from original example
    SMP_FUNCTION_ACCEPTED = 0,
    SMP_UNKNOWN_FUNCTION,
    SMP_FUNCTION_FAILED,
    SMP_INVALID_REQUEST_FRAME_LENGTH,
    SMP_PHY_DOES_NOT_EXIST = 0x10,
    SMP_INDEX_DOES_NOT_EXIST,
    SMP_PHY_DOES_NOT_SUPPORT_SATA,
    SMP_UNKNOWN_PHY_OPERATION,
    SMP_UNKNOWN_PHY_TEST_FUNCTION,
    SMP_UNKNOWN_PHY_TEST_FUNCTION_IN_PROGRESS,
    SMP_PHY_VACANT
};

// DeviceTypes
enum DeviceTypes
{
    NO_DEVICE = 0,
    END_DEVICE,
    EDGE_EXPANDER_DEVICE,
    FANOUT_EXPANDER_DEVICE,
    END = END_DEVICE,             // from original example
    EDGE = EDGE_EXPANDER_DEVICE,  // from original example
    FANOUT = FANOUT_EXPANDER_DEVICE // from original example
};

// RoutingAttribute
enum RoutingAttribute
{
    DIRECT = 0,
    SUBTRACTIVE,
    TABLE,

    // this attribute is a pseudo attribute, used to reflect the function
    // result of SMP_PHY_VACANT in a fabricated discover response
    PHY_NOT_USED = 15
};

// ConnectorType
enum ConnectorType
{
    UNKNOWN_CONNECTOR = 0,
    SFF_8470_EXTERNAL_WIDE,
    SFF_8484_INTERNAL_WIDE = 16,
    SFF_8482_BACKPLANE = 32,
    SATA_HOST_PLUG,
    SAS_DEVICE_PLUG,
    SATA_DEVICE_PLUG
// RouteFlag
enum DisableRouteEntry
{
    ENABLED = 0,
    DISABLED
};

// PhyLinkRate(s)
enum PhysicalLinkRate
{
    RATE_UNKNOWN = 0,
    PHY_DISABLED,
    PHY_FAILED,
    SPINUP_HOLD_OOB,
    PORT_SELECTOR_DETECTED,

    // this is a psuedo link rate, used to reflect the function
    // result of SMP_PHY_VACANT in a fabricated discover response
    PHY_DOES_NOT_EXIST,

    GBPS_1_5 = 8,
    GBPS_3_0
};

// PhyOperation
enum PhyOperation
{
    NOP = 0,
    LINK_RESET,
    HARD_RESET,
    DISABLE,
    CLEAR_ERROR_LOG = 5,
    CLEAR_AFFILITATION,
    TRANSMIT_SATA_PORT_SELECTION_SIGNAL
};

// provide the simple type definitions
typedef unsigned char byte;
typedef unsigned short word;
typedef unsigned long dword;
typedef unsigned _int64 quadword;

// the structures assume a char bitfield is valid, this is compiler
// dependent defines would be more portable, but less descriptive

// the Identify frame is exchanged following OOB, for this
// code it contains the identity information for the attached device
// and the initiator application client
struct Identify
{
    // byte 0
    byte AddressFrameType:4; // ADDRESS_IDENTIFY_FRAME
    byte DeviceType:3;     // END_DEVICE

    byte RestrictedByte0Bit7:1;

    // byte 1
    byte RestrictedByte1;

    // byte 2
    union
    {
        struct
        {
            byte RestrictedByte2Bit0:1;
        }
    }
byte SMPInitiatorPort:1;
byte STPInitiatorPort:1;
byte SSPInitiatorPort:1;
byte ReservedByte2Bit4_7:4;
};
byte InitiatorBits;
};

// byte 3
union
{
    struct
    {
        byte RestrictedByte3Bit0:1;
        byte SMPTargetPort:1;
        byte STPTargetPort:1;
        byte SSPTargetPort:1;
        byte ReservedByte3Bit4_7:4;
    }
    byte TargetBits;
};

// byte 4-11
byte RestrictedByte4_11[8];

// byte 12-19
quadword SASAddress;

// byte 20
byte PhyIdentifier;

// byte 21-27
byte ReservedByte21_27[4];

// byte 28-31
dword CRC;

// the Open address frame is used to send open requests
struct OpenAddress
{
    // byte 0
    byte AddressFrameType:4;          // ADDRESS_OPEN_FRAME
    byte Protocol:3;                  // SMP
    // STP
    // SSP
    byte InitiatorPort:1;

    // byte 1
    byte ConnectionRate:4;            // GBPS_1_5
    // GBPS_3_0
    byte Features:4;

    // byte 2-3
    word InitiatorConnectionTag;

    // byte 4-11
    quadword DestinationSASAddress;

    // byte 12-19
    quadword SourceSASAddress;

    // byte 20
    byte CompatibleFeatures;

    // byte 21
    byte PathwayBlockedCount;
// byte 22-23
word ArbitrationWaitTime;

// byte 24-27
byte MoreCompatibleFeatures[4];

// byte 28-31
dword CRC[4];

// request specific bytes for a general input function
struct SMPRequestGeneralInput
{
    // byte 4-7
dword CRC;
};

// request specific bytes for a phy input function
struct SMPRequestPhyInput
{
    // byte 4-7
    byte IgnoredByte4_7[4];

    // byte 8
    byte ReservedByte8;

    // byte 9
    byte PhyIdentifier;

    // byte 10
    byte IgnoredByte10;

    // byte 11
    byte ReservedByte11;

    // byte 12-15
dword CRC;
};

// the ConfigureRouteInformation structure is used to provide the
// expander route entry for the expander route table, it is intended
// to be referenced by the SMPRequestConfigureRouteInformation struct
struct ConfigureRouteInformation
{
    // byte 12
    byte IgnoredByte12Bit0_6:7;
    byte DisableRouteEntry:1; // if a routing error is detected
                              // then the route is disabled by
                              // setting this bit

    // byte 13-15
    byte IgnoredByte13_15[3];

    // byte 16-23
    quadword RoutedSASAddress; // identical to the AttachedSASAddress
                              // found through discovery

    // byte 24-35
    byte IgnoredByte24_35[12];

    // byte 36-39
    byte ReservedByte36_39[4];
};

// request specific bytes for SMP ConfigureRouteInformation function
struct SMPRequestConfigureRouteInformation
{  
  // byte 4-5
  byte ReservedByte4_5[2];

  // byte 6-7
  word ExpanderRouteIndex;

  // byte 8
  byte ReservedByte8;

  // byte 9
  byte PhyIdentifier;

  // byte 10-11
  byte ReservedByte10_11[2];

  // byte 12-39
  struct ConfigureRouteInformation Configure;

  // byte 40-43
  dword CRC;
};

// the PhyControlInformation structure is used to provide the expander phy control values, it is intended to be referenced by the SMPRequestPhyControl struct
struct PhyControlInformation
{
  // byte 12-31
  byte IgnoredByte12_31[20];

  // byte 32
  byte IgnoredByte32Bit0_3:4;
  byte ProgrammedMinimumPhysicalLinkRate:4;

  // byte 33
  byte IgnoredByte33Bit0_3:4;
  byte ProgrammedMaximumPhysicalLinkRate:4;

  // byte 34-35
  byte IgnoredByte34_35[2];

  // byte 36
  byte PartialPathwayTimeoutValue:4;
  byte ReservedByte36Bit4_7:4;

  // byte 37-39
  byteReservedByte37_39[3];
};

// request specific bytes for SMP Phy Control function
struct SMPRequestPhyControl
{
  // byte 4-7
  byte IgnoredByte4_7[4];

  // byte 8
  byte ReservedByte8;

  // byte 9
  byte PhyIdentifier;

  // byte 10
  byte PhyOperation;

  // byte 11
  byte UpdatePartialPathwayTimeoutValue:1;
byte ReservedByte11Bit1_7:7;

// byte 12-39
struct PhyControlInformation Control;

// byte 40-43
dword CRC;
);

// request specific bytes for SMP Phy Test function
struct SMPRequestPhyTest
{
    // byte 4-7
    byte IgnoredByte4_7[4];

    // byte 8
    byte ReservedByte8;

    // byte 9
    byte PhyIdentifier;

    // byte 10
    byte PhyTestFunction;

    // byte 11
    byte PhyTestPattern;

    // byte 12-14
    byte ReservedByte12_14[3];

    // byte 15
    byte PhyTestPatternPhysicalLinkRate:4;
    byte ReservedByte15Bit4_7:4;

    // byte 16-39
    byte ReservedByte16_39[24];

    // byte 40-43
dword CRC;
};

// generic structure referencing an SMP Request, must be initialized
// before being used
struct SMPRequest
{
    // byte 0
    byte SMPFrameType;                  // always SMP_REQUEST_FRAME

    // byte 1
    byte Function;                      // REPORT_GENERAL
                                          // REPORT_MANUFACTURER_INFORMATION
                                          // DISCOVER
                                          // REPORT_PHY_ERROR_LOG
                                          // REPORT_PHY_SATA
                                          // REPORT_ROUTE_INFORMATION
                                          // CONFIGURE_ROUTE_INFORMATION
                                          // PHY_CONTROL
                                          // PHY_TEST

    // byte 2-3
    byte ReservedByte2_3[2];

    // bytes 4-n
    union
    {
        struct SMPRequestGeneralInput ReportGeneral;
        struct SMPRequestGeneralInput ReportManufacturerInformation;
    };

struct SMPRequestPhyInput Discover;
struct SMPRequestPhyInput ReportPhyErrorLog;
struct SMPRequestPhyInput ReportPhySATA;
struct SMPRequestPhyInput ReportRouteInformation;
struct SMPRequestConfigureRouteInformation ConfigureRouteInformation;
struct SMPRequestPhyControl PhyControl;
struct SMPRequestPhyTest PhyTest;
}
Request;

// request specific bytes for SMP Report General response, intended to be
// referenced by SMPResponse
struct SMPResponseReportGeneral
{
    // byte 4-5
    word ExpanderChangeCount;

    // byte 6-7
    word ExpanderRouteIndexes;

    // byte 8
    byte ReservedByte8;

    // byte 9
    byte NumberOfPhys;

    // byte 10
    byte ConfigurableRouteTable:1;
    byte Configuring:1;
    byte ReservedByte10Bit2_7:6;

    // byte 11-18
    byte EnclosureLogicalIdentifier[8];

    // byte 19-27
    byte ReservedByte19_27[9];

    // byte 28-31
    dword CRC;
};

struct SAS11FormatReportManufacturerInformation
{
    // byte 40-47
    byte ComponentVendorIdentification[8];

    // byte 48-49
    byte ComponentID[2];

    // byte 50
    byte ComponentRevisionID;

    // byte 51
    byte Reserved;

    // byte 52-59
    byte VendorSpecific[8];
};

// request specific bytes for SMP Report Manufacturer Information response,
// intended to be referenced by SMPResponse
struct SMPResponseReportManufacturerInformation
{
    // byte 4-7
    byte IgnoredByte4_7[4];

    // byte 8
byte SAS11Format:1;
byte ReservedByte8_Bit1_7:7;

// byte 9-10
byte IgnoredByte9_10[2];

// byte 11
byte ReservedByte11;

// byte 12-19
byte VendorIdentification[8];

// byte 20-35
byte ProductIdentification[16];

// byte 36-39
byte ProductRevisionLevel[4];

union
{
    struct SAS11FormatReportManufacturerInformation SAS11;

    // byte 40-59
    byte VendorSpecific[20];
};

// byte 60-63
dword CRC;

// the Discover structure is used to retrieve expander port information
// it is intended to be referenced by the SMPResponseDiscover structure
struct Discover
{
    // byte 12
    byte ReservedByte12Bit0_3:4;
    byte AttachedDeviceType:3;
    byte IgnoredByte12Bit7:1;

    // byte 13
    byte NegotiatedPhysicalLinkRate:4;
    byte ReservedByte13Bit4_7:4;

    // byte 14
    union
    {
        struct
        {
            byte AttachedSATAHost:1;
            byte AttachedSMPInitiator:1;
            byte AttachedSTPInitiator:1;
            byte AttachedSSPInitiator:1;
            byte ReservedByte14Bit4_7:4;
        };  // end struct
        byte InitiatorBits;
    }  // end union
;

    // byte 15
    union
    {
        struct
        {
            byte AttachedSATADevice:1;
            byte AttachedSMPTarget:1;
            byte AttachedSTPTarget:1;
            byte AttachedSSPTarget:1;
            byte ReservedByte15Bit4_6:3;
        }
    }
;


byte AttachedSATAPortSelector:1;
byte TargetBits;
}

// byte 16-23
quadword SASAddress;

// byte 24-31
quadword AttachedSASAddress;

// byte 32
byte AttachedPhyIdentifier;

// byte 33-39
byte ReservedByte33_39[7];

// byte 40
byte HardwareMinimumPhysicalLinkRate:4;
byte ProgrammedMinimumPhysicalLinkRate:4;

// byte 41
byte HardwareMaximumPhysicalLinkRate:4;
byte ProgrammedMaximumPhysicalLinkRate:4;

// byte 42
byte PhyChangeCount;

// byte 43
byte PartialPathwayTimeoutValue:4;
byte IgnoredByte36Bit4_6:3;
byte VirtualPhy:1;

// byte 44
byte RoutingAttribute:4;
byte ReservedByte44Bit4_7:4;

// byte 45
byte ConnectorType:7;
byte ReservedByte45Bit7:1;

// byte 46
byte ConnectorElementIndex;

// byte 47
byte ConnectorPhysicalLink;

// byte 48-49
byte ReservedByte48_49[2];

// byte 50-51
byte VendorSpecific[2];

// byte 52-55
dword CRC;
};

// response specific bytes for SMP Discover, intended to be referenced by // SMPResponse
struct SMPResponseDiscover
{
    // byte 4-7
    byte IgnoredByte4_7;

    // byte 8
    byte ReservedByte8;
// byte 9
byte PhyIdentifier;

// byte 10
byte IgnoredByte10;

// byte 11
byte ReservedByte11;

union
    // original example used Results instead
    // of Result, this allows both
    {
        // byte 12-55
        struct Discover Results;
        struct Discover Result;
    };

// response specific bytes for SMP Report Phy Error Log, intended to be
// referenced by SMPResponse
struct SMPResponseReportPhyErrorLog
{
    // byte 4-7
    byte IgnoredByte4_7;

    // byte 8
    byte ReservedByte8;

    // byte 9
    byte PhyIdentifier;

    // byte 10
    byte IgnoredByte10;

    // byte 11
    byte ReservedByte11;

    // byte 12-15
    dword InvalidDwordCount;

    // byte 16-19
    dword DisparityErrorCount;

    // byte 20-23
    dword LossOfDwordSynchronizationCount;

    // byte 24-27
    dword PhyResetProblemCount;

    // byte 28-31
    dword CRC;
};

// this structure describes the Register Device to Host FIS defined in the
// SATA specification
struct RegisterDeviceToHostFIS
{
    // byte 24
    byte FISType;

    // byte 25
    byte ReservedByte25Bit0_5:6;
    byte Interrupt:1;
    byte ReservedByte25Bit7:1;

    // byte 26
    byte Status;
// byte 27
byte Error;

// byte 28
byte SectorNumber;

// byte 29
byte CylLow;

// byte 30
byte CylHigh;

// byte 31
byte DevHead;

// byte 32
byte SectorNumberExp;

// byte 33
byte CylLowExp;

// byte 34
byte CylHighExp;

// byte 35
byte ReservedByte35;

// byte 36
byte SectorCount;

// byte 37
byte SectorCountExp;

// byte 38-43
byte ReservedByte38_43[6];
}

// response specific bytes for SMP Report Phy SATA, intended to be
// referenced by SMPResponse
struct SMPResponseReportPhySATA
{
    // byte 4-7
    byte IgnoredByte4_7;

    // byte 8
    byte ReservedByte8;

    // byte 9
    byte PhyIdentifier;

    // byte 10
    byte IgnoredByte10;

    // byte 11
    byte AffiliationValid:1;
    byte AffiliationsSupported:1;
    byte ReservedByte11Bit2_7:6;

    // byte 12-15
    byte ReservedByte12_15[4];

    // byte 16-32
    quadword STPSASAddress;

    // byte 24-43
    struct RegisterDeviceToHostFIS FIS;
}
// byte 44-47
byte ReservedByte44_47[4];

// byte 48-55
quadword AffiliatedSTPInitiatorSASAddress;

// byte 56-59
dword CRC;
};

struct ReportRouteInformation
{
  // byte 12
  byte IgnoredByte12Bit0_6:7;
  byte ExpanderRouteEntryDisabled:1;

  // byte 13-15
  byte IgnoredByte13_15[3];

  // byte 16-23
  quadword RoutedSASAddress;

  // byte 24-35
  byte IgnoredByte24_35[12];

  // byte 36-39
  byte ReservedByte36_39[4];
};

// response specific bytes for SMP Report Route Information, intended to be
// referenced by SMPResponse
struct SMPResponseReportRouteInformation
{
  // byte 4-5
  byte IgnoredByte4_5;

  // byte 6-7
  word ExpanderRouteIndex;

  // byte 8
  byte ReservedByte8;

  // byte 9
  byte PhyIdentifier;

  // byte 10
  byte IgnoredByte10;

  // byte 11
  byte ReservedByte11;

  // byte 12-39
  struct ReportRouteInformation Result;

  // byte 40-43
  dword CRC;
};

// response specific bytes for SMP Configure Route Information,
// intended to be referenced by SMPResponse
struct SMPResponseConfigureRouteInformation
{
  // byte 4-7
  dword CRC;
};
// response specific bytes for SMP Phy Control,  
// intended to be referenced by SMPResponse
struct SMPResponsePhyControl
{
    // byte 4-7
    dword CRC;
};

// response specific bytes for SMP Phy Test,  
// intended to be referenced by SMPResponse
struct SMPResponsePhyTest
{
    // byte 4-7
    dword CRC;
};

// generic structure referencing an SMP Response, must be initialized  
// before being used
struct SMPResponse
{
    // byte 0
    byte SMPFrameType;                // always 41h for SMP responses

    // byte 1
    byte Function;

    // byte 2
    byte FunctionResult;

    // byte 3
    byte ReservedByte3;

    // bytes 4-n
    union
    {
        struct SMPResponseReportGeneral ReportGeneral;
        struct SMPResponseReportManufacturerInformation ReportManufacturerInformation;
        struct SMPResponseDiscover Discover;
        struct SMPResponseReportPhySATA ReportPhySATA;
        struct SMPResponseReportRouteInformation ReportRouteInformation;
        struct SMPResponseConfigureRouteInformation ConfigureRouteInformation;
        struct SMPResponsePhyControl PhyControl;
        struct SMPResponsePhyTest PhyTest;
    } Response;
};

// this structure is how this simulation obtains it's knowledge about the  
// initiator port that is doing the discover, it is not defined as part of  
// the standard...
struct ApplicationClientKnowledge
{
    quadword SASAddress;
    byte NumberOfPhys;
    byte InitiatorBits;
    byte TargetBits;
};

// the RouteTableEntry structure is used to contain the internal copy of  
// the expander route table
struct RouteTableEntry
{
    byte ExpanderRouteEntryDisabled;
    quadword RoutedSASAddress;
};
// the TopologyTable structure is the summary of the information gathered
during the discover process, the table presented here is not concerned
about memory resources consumed, production code would be more concerned
about specifying necessary elements explicitly

```c
struct TopologyTable
{
    // pointer to a simple list of expanders in topology
    // a walk thru this link will encounter all expanders in
    // discover order
    struct TopologyTable *Next;

    // simple reference to this device, primarily to keep identification of
    // this structure simple, otherwise, the only place the address is
    // located is within the Phy element
    quadword SASAddress;

    // information from REPORT_GENERAL
    struct SMPResponseReportGeneral Device;

    // information from DISCOVER
    struct SMPResponseDiscover Phy[MAXIMUM_EXPANDER_PHYS];

    // list of route indexes for each phy
    word RouteIndex[MAXIMUM_EXPANDER_PHYS];

    // internal copy of the route table for the expander
    struct RouteTableEntry
        RouteTable[MAXIMUM_EXPANDER_PHYS][MAXIMUM_EXPANDER_INDEXES];

    // in production code there would also be links to the necessary device
    // information like end device; vendor, model, serial number, etc.
    // the gathering of that type of information is not done here...
    //
};
```

Replace K.3 with the following text;

```c
// SASDiscoverSimulation.cpp
//
// this is a simple simulation and code implementation of the initiator
// based expander discovery and configuration

// there is no attempt to handle phy errors, arbitration issues, etc.
// production level implementation would have to handle errors appropriately

// structure names used are equivalent to those referenced in the
// SAS document

// basic assumptions
//
// 1. change primitives will initiate a rediscovery/configuration sequence
// 2. table locations for SASAddresses are deterministic for a specific
//    topology only, when the topology changes, the location of a SASAddress
//    in an ASIC table cannot be assumed
// 3. a complete discovery level occurs before the configuration of the
//    level begins, multiple passes are required as the levels of expanders
//    encountered between the initiator and the end devices is increased
// 4. configuration of a single expander occurs before proceeding to
//    subsequent expanders attached
// 5. the Attached structure is filled in following OOB and is available
//    from the initialization routines
// 6. the Iam structure is provide by the application client
```
```c
#include <malloc.h>
#include <memory.h>
#include <stdlib.h>

// include the SAS structures
#include "SASDiscoverSimulation.h"

// this defines the type of algorithm used for discover
int DiscoverAlgorithm = SAS_SIMPLE_LEVEL_DESCENT;
int SASCompatibility = SAS_11_COMPATIBLE;

// loaded by the application client, in this simulation it is provided
// in a text file, SASDeviceSetExample.ini
extern struct ApplicationClientKnowledge Iam[MAXIMUM_INITIATORS];

// obtained following OOB from the attached phy, in this simulation
// it is provided in a text file, SASDeviceSetExample.ini
extern struct Identify Attached[MAXIMUM_INITIATORS];

// buffers used to request and return SMP data
extern struct SMPRequest SMPRequestFrame;
extern struct SMPResponse SMPResponseFrame;

// resulting discover information will end up in this table
extern struct TopologyTable *SASDomain[MAXIMUM_INITIATORS];

// this is the function used to send an SMPRequest and get a response back
extern byte SMPRequest(byte PhyIdentifier,
    quadword Source,
    quadword Destination, 
    struct SMPRequest *SMPRequestFrame, 
    struct SMPResponse *SMPResponseFrame, 
    byte *OpenStatus, 
    byte Function, 
    ...);

// this function is used to output error information, it mimics fprintf
// functionality to an open trace file
extern int TracePrint(char *String, ...);

// this function gets the report general and discover information for
// a specific expander, the discover process should begin at the subtractive
// boundary and progress downstream
static struct TopologyTable *DiscoverExpander(byte PhyIdentifier, 
    quadword SourceSASAddress, 
    quadword DestinationSASAddress) 
{
    struct TopologyTable *expander = 0;
    byte phyCount = 0;
    int error = 1;

    byte openStatus = OPEN_ACCEPT;

    // get the report general information for the expander
    SMPRequest(PhyIdentifier, 
        SourceSASAddress, 
        DestinationSASAddress, 
        &SMPRequestFrame, 
        &SMPResponseFrame, 
        &openStatus, 
        REPORT_GENERAL);

    // don't worry about too much in the 'else' case for this example,
    // production code must handle
    if((openStatus == OPEN_ACCEPT) &&
```
{
    // allocate space to retrieve the expander information
    expander = (struct TopologyTable *)calloc(1, sizeof(struct TopologyTable));

    // make sure we only do this if the allocation is successful
    if(expander)
    {
        // save the address of this expander
        expander->SASAddress = DestinationSASAddress;

        // copy the result into the topology table
        memcpy((void *)&(expander->Device),
               (void *)&SMPResponseFrame.Response.ReportGeneral,
               sizeof(struct SMPResponseReportGeneral));

        // now walk through all the phys of the expander
        for(phyCount = 0;
            (phyCount < expander->Device.NumberOfPhys);
            phyCount++)
        {
            // get the discover information for each phy
            SMPRequest(PhyIdentifier,
                       SourceSASAddress,
                       DestinationSASAddress,
                       &SMPRequestFrame,
                       &SMPResponseFrame,
                       openStatus,
                       DISCOVER,
                       phyCount);

            // don't worry about the 'else' case for this example,
            // production code must handle
            if((openStatus == OPEN_ACCEPT) &&
               (SMPResponseFrame.FunctionResult == SMP_FUNCTION_ACCEPTED))
            {
                // clear the error flag
                error = 0;

                // copy the result into the topology table
                memcpy((void *)&(expander->Phy[phyCount]),
                       (void *)&SMPResponseFrame.Response.Discover,
                       sizeof(struct SMPResponseDiscover));
            }
            else if((openStatus == OPEN_ACCEPT) &&
                     (SMPResponseFrame.FunctionResult == SMP_PHY_VACANT))
            {
                struct Discover *discover;


                // clear the error flag
                error = 0;

                // set the routing attribute and link rate to indicate that
                // the phy is not being used, this keeps it from being
                // included in the routing table information, these values
                // are not defined in the spec at this time, but are listed
                // as reserved values
                discover->NegotiatedPhysicalLinkRate = PHY_DOES_NOT_EXIST;
                discover->RoutingAttribute = PHY_NOT_USED;
// copy the result into the topology table
memcpy((void *)&(expander->Phy[phyCount]),
(void *)&SMPResponseFrame.Response.Discover,
sizeof(struct SMPResponseDiscover));
}
else
{
  // if we had a problem on this link, then don't bother
  // to do anything else, production code, should be more
  // robust...
  // for this simulation example, the addresses are
  // described as strings, so we can print them out...
  // not true for production code...
  TracePrint("\n"  
  "discover error, %02Xh at %s\n",
  SMPResponseFrame.FunctionResult,
  (char *)&DestinationSASAddress);

  // something happened so just bailout on this expander
  error = 1;

  // release the memory we allocated for this...
  free(expander);
  expander = 0;
  break;
}
}
}
// the assumptions we made were exceeded, need to bump simulation
// limits...
else
{
  TracePrint("\n"  
  "report general error"
  ", NumberOfPhys %d exceeded limit %d on %s\n",
  expander->Device.NumberOfPhys,
  MAXIMUM_EXPANDER_PHYS,
  (char *)&DestinationSASAddress);
}
else
{
  // if we had a problem getting report general for this expander,
  // something is wrong, can't go any further down this path...
  // production code, should be more robust...
  // for this simulation example, the addresses are
  // described as strings, so we can print them out...
  // not true for production code...
  TracePrint("\n"  
  "report general error, open %02Xh result %02Xh at %s\n",
  openStatus,
  SMPResponseFrame.FunctionResult,
  (char *)&DestinationSASAddress);
}

// the expander pointer is the error return, a null indicates something
// bad happened...
return(expander);

// this routine searches upstream for the subtractive boundary that defines
// the edge expander device set
static
struct TopologyTable *FindBoundary(byte PhyIdentifier,
  quadword SourceSASAddress,
  struct TopologyTable *Expander,
struct TopologyTable **DeviceSet) {

    struct TopologyTable *expander = Expander;
    struct TopologyTable *nextExpander;

    struct Discover *discover;

    byte phyCount;
    int error = 0;
    int foundSubtractivePort = 0;

    quadword subtractiveSASAddress;
    byte attachedPhyIdentifier;

    // make sure the device set link is initialized
    *DeviceSet = 0;

    // the outer loop searches for subtractive phys and finds the SAS addresses
    // connected to them, it validates that the subtractive phys all resolve
    // to the same expander address, then moves upstream searching for the
    // edge expander device set boundary
    do {
        // initialize the subtractive address, a zero value is not valid
        subtractiveSASAddress = 0;
        attachedPhyIdentifier = 0;

        // walk through all the phys of this expander
        for(phyCount = 0;
            (phyCount < expander->Device.NumberOfPhys);
            phyCount++)
        {
            // this is just a pointer helper
            discover = &(expander->Phy[phyCount].Result);

            // look for phys with edge or fanout devices attached...
            if((discover->RoutingAttribute == SUBTRACTIVE) &&
                ((discover->AttachedDeviceType == EDGE_EXPANDER_DEVICE) ||
                (discover->AttachedDeviceType == FANOUT_EXPANDER_DEVICE)))
            {
                // make sure all the subtractive phys point to the same address
                // when we are connected to an expander device
                if(!subtractiveSASAddress)
                {
                    subtractiveSASAddress = discover->AttachedSASAddress;
                    attachedPhyIdentifier = discover->AttachedPhyIdentifier;
                    foundSubtractivePort = 1;
                }
            }
            // the addresses don't match... problem...
            else if(subtractiveSASAddress !=
                discover->AttachedSASAddress)
            {
                // production code needs to deal with this better, for this
                // example, the SASAddresses are assumed to strings
                // so just print out the error information
                TracePrint("\n" "topology error, diverging subtractive phys"
                ", '%s' != '%s' \n",
                (char *)&subtractiveSASAddress,
                (char *)&discover->AttachedSASAddress);
                error = 1;
                break;
            }  
        }
    }
}
// if no error, then decide if we need to go upstream or stop
if(!error)
{
    // if we have a subtractive address then go upstream to see
    // if it is part of the edge expander device set
    if(subtractiveSASAddress)
    {
        // get the discover information
        nextExpander = DiscoverExpander(PhyIdentifier,
            SourceSASAddress,
            subtractiveSASAddress);

        // if we successfully got the information from the next
        // expander then proceed upstream...
        if(nextExpander)
        {
            struct Discover *discover;
            // this is just a pointer helper
            discover = &(nextExpander->Phy[attachedPhyIdentifier].Result);

            // check to see if we are connected to the subtractive
            // port of the next expander, if we are then we have two
            // expander device sets connected together, stop here
            // and save the address of next expander in device set,
            // the return will be expander
            if(discover->RoutingAttribute == SUBTRACTIVE)
            {
                *DeviceSet = nextExpander;
                break;
            }
            // go ahead and continue upstream looking for the boundary
            else
            {
                // release the memory we allocated for this
                free(expander);

                // move upstream to the next expander
                expander = nextExpander;
            }
        }
        // if there are no more upstream expanders stop here...
        else
        {
            break;
        }
    }
    // if we did not get a subtractive address this time around then stop
    else
    {
        // if we did find a subtractive port on a previous pass,
        // then return with expander pointing to the last device
        // with the subtractive port
        if(foundSubtractivePort)
        {
            break;
        }
        // if we never found a subtractive port, then return with a
        // null indicating there are no subtractive phys, don't free
        // the memory, because it is still in use by the calling routine
        else
        {
            expander = 0;
        }
    }
}
// if there was an error make sure we return a null expander pointer
else
{
    // to get here, we had to see more than one subtractive phy that
    // connect to different SAS addresses, this is a topology error
    // do cleanup on any memory allocated if necessary
    if((expander != Expander) &&
        (expander != *DeviceSet))
    {
        // release the memory we allocated for this and make sure
        // we return a null
        free(expander);
        expander = 0;
    }
}
}
while(!error &&
    expander &&
    subtractiveSASAddress);

// on return expander should contain the subtractive boundary expander
// or a null indicating there were no subtractive phys,
// or a null indicating there was an error
return(expander);
}

// find the table structure associated with a specific SAS address
static
struct TopologyTable *FindExpander(struct TopologyTable *Expander,
    quadword SASAddress)
{
    // walk the list of expanders, when we find the one that matches, stop
    while(Expander)
    {
        // do the SASAdresses match
        if(SASAddress == Expander->SASAddress)
        {
            break;
        }

        Expander = Expander->Next;
    }

    return(Expander);
}

// this routine searches the subtractive phys for the upstream expander address
static
int UpstreamExpander(struct TopologyTable *Expander,
    quadword *SASAddress,
    byte *PhyIdentifier)
{
    struct Discover *discover;

    byte phyCount;

    int found = 0;

    // walk through all the phys of this expander, searching for subtractive
    // phys return the SASAddress and PhyIdentifier for the first subtractive
    // phy encountered, they should all be the same if they have anything
    // attached
    for(phyCount = 0;
        (phyCount < Expander->Device.NumberOfPhys);
        phyCount++)
    {
        // this is just a pointer helper

discover = &(Expander->Phy[phyCount].Result);

   // look for phys with edge or fanout devices attached...
   if ((discover->RoutingAttribute == SUBTRACTIVE) &&
   ((discover->AttachedDeviceType == EDGE_EXPANDER_DEVICE) ||
   (discover->AttachedDeviceType == FANOUT_EXPANDER_DEVICE)))
   {
      *SASAddress = discover->AttachedSASAddress;
      *PhyIdentifier = discover->AttachedPhyIdentifier;
      found = 1;
      break;
   }
}

return(found);

   // this routine determines whether a SAS address is directly attached to
   // an expander
   static
   int DirectAttached(struct TopologyTable *Expander,
                      quadword SASAddress)
   {
      int direct = 0;
      byte phyCount;

      for (phyCount = 0;
           phyCount < Expander->Device.NumberOfPhys;
           phyCount++)
      {
         // did we find the address attached locally
         if (SASAddress ==
             Expander->Phy[phyCount].Result.AttachedSASAddress)
         {
            direct = 1;
            break;
         }
      }

      return(direct);
   }

   // this routine determines whether the SAS address, can be optimized out
   // of the route table
   static
   int QualifiedAddress(struct TopologyTable *Expander,
                        byte PhyIdentifier,
                        quadword SASAddress,
                        byte RoutingAttribute,
                        byte *DisableRouteEntry)
   {
      int qualified = 1;
      word routeIndex;

      if (DiscoverAlgorithm == SAS_UNIQUE_LEVEL_DESCENT)
      {
         // leave in any entries that are direct routing attribute, assumes
         // that they are slots that will be filled by end devices, if
         // it is not direct, then filter out any empty connections,
         // connections that match the expander we are configuring
         // and connections that are truly direct attached
         if (SASAddress &&
             (SASAddress != Expander->SASAddress) &&
             (!DirectAttached(Expander,
                              SASAddress)))
         {
            if (RoutingAttribute == DIRECT)

   Page 22 of 32
else
{
    for(routeIndex = 0;
        routeIndex < Expander->Device.ExpanderRouteIndexes;
        routeIndex++)
    {
        struct RouteTableEntry *entry =
            &Expander->RouteTable[PhyIdentifier][routeIndex];

        if(entry->RoutedSASAddress == SASAddress)
        {
            qualified = 0;
            break;
        }
    }
}
}
else if(!SASAddress &&
    (RoutingAttribute == DIRECT))
{
    // if a 0 address that is direct routing, then assume it is an
    // empty slot that can be filled at any time, this keeps things
    // positionally stable for most reasonable topologies
    *DisableRouteEntry = DISABLE;
}
else
{
    qualified = 0;
}
}

return(qualified);
}

// this function is the configuration cycle from the current expander to
// the hub expander
static int ConfigureExpander(byte PhyIdentifier,
    quadword SourceSASAddress,
    struct TopologyTable *HubExpander,
    struct TopologyTable *ThisExpander)
{
    struct TopologyTable *thisExpander = ThisExpander;
    struct TopologyTable *expander = ThisExpander;
    struct TopologyTable *configureExpander;

    struct Discover *discover;

    quadword upstreamSASAddress = 0;
    byte upstreamPhyIdentifier = 0;

    byte phyIndex;
    word routeIndex;
    byte openStatus = OPEN_ACCEPT;

    int error = 0;

    do
    {
        // move upstream from here to find the expander table to configure with
        // information from "thisExpander"
if(!UpstreamExpander(thisExpander,
   &upstreamSASAddress,
   &upstreamPhyIdentifier))
{
   break;
}

if(upstreamSASAddress)
{
   // get the expander associated with the upstream address
   configureExpander = FindExpander(HubExpander,
                                      upstreamSASAddress);

   // if we found an upstream expander, then program it's route
   // table
   if(configureExpander)
   {
      byte disableRouteEntry = ENABLED;

      for(phyIndex = 0;
          phyIndex < configureExpander->Device.NumberOfPhys;
          phyIndex++)
      {
         if(configureExpander->Phy[phyIndex].Result.AttachedSASAddress ==
            thisExpander->SASAddress)
         {
            // loop through all the phys of the attached expander
            for(routeIndex = 0;
                (routeIndex <
                thisExpander->Device.NumberOfPhys) &&
                (configureExpander->RouteIndex[phyIndex] <
                configureExpander->Device.ExpanderRouteIndexes);
                routeIndex++)
         {
            discover = &(expander->Phy[routeIndex].Result);

            // assume the route entry is enabled
            disableRouteEntry = ENABLED;

            // check to see if the address needs to be configured
            // in the route table, this decision is based on the
            // optimization flag
            if(QualifiedAddress(configureExpander,
                                 phyIndex,
                                 discover->AttachedSASAddress,
                                 discover->RoutingAttribute,
                                 &disableRouteEntry))
            {
               word index = configureExpander->RouteIndex[phyIndex];

               struct RouteTableEntry *entry =
               &configureExpander->RouteTable[phyIndex][index];

               // configure the route indexes for the expander
               // with the attached address information
               SMPRequest(PhyIdentifier,
                           SourceSASAddress,
                           configureExpander->SASAddress,
                           &SMPRequestFrame,
                           &SMPResponseFrame,
                           &openStatus,
                           CONFIGURE_ROUTE_INFORMATION,
                           index,
                           phyIndex,
                           disableRouteEntry,
                           discover->AttachedSASAddress);
if((openStatus != OPEN_ACCEPT) ||
   (SMPResponseFrame.FunctionResult !=
    SMP_FUNCTION_ACCEPTED))
{
    error = 1;
    break;
}

// add the address to the internal copy of the
// route table, if successfully configured
entry->RoutedSASAddress =
    discover->AttachedSASAddress;

// increment the route index for this phy
configureExpander->RouteIndex[phyIndex]++;

// add the address to the internal copy of the
// route table, if successfully configured
entry->RoutedSASAddress =
    discover->AttachedSASAddress;

// increment the route index for this phy
configureExpander->RouteIndex[phyIndex]++;

// move upstream
thisExpander = configureExpander;
}

while(!error &&
    thisExpander &&
    upstreamSASAddress);

return(error);

// this discovers then configures as necessary the expanders it finds
// within the SAS domain that are "downstream"
static struct TopologyTable *DiscoverAndConfigure(byte PhyIdentifier,
    quadword SourceSASAddress,
    struct TopologyTable *HubExpander,
    struct TopologyTable **DeviceSet)
{
    struct TopologyTable *currentExpander = HubExpander;
    struct TopologyTable *nextExpander;
    struct Discover *currentDiscover;

    quadword sasAddress;
    byte phyIndex;

    int error = 0;

    // this is a level descent traversal with a configuration stage
    // at each transition to a new level, if a configuration is required
    // by the expander

    // the discover process moves forward through the topology, but the
    // configuration process stays anchored at the hub of the
    // topology, meaning the fanout expander, or the top most subtractive
    // edge expander device
    // this ensures that as each new expander is added to the
    // topology table list, it is in the configuration chain

    while(!error &&
        currentExpander)
    {
        // start at phy 0

phyIndex = 0;

// walk through all the phys of the current expander looking for
// new expanders to add to the topology table
do
{
  // this is just a pointer helper
  currentDiscover = &(currentExpander->Phy[phyIndex].Result);

  // look for phys with edge or fanout devices attached...
  if((currentDiscover->RoutingAttribute == TABLE) &&
      ((currentDiscover->AttachedDeviceType == EDGE_EXPANDER_DEVICE) ||
       (currentDiscover->AttachedDeviceType == FANOUT_EXPANDER_DEVICE)))
  {
    struct TopologyTable *thisExpander = currentExpander;
    struct TopologyTable *previousExpander = currentExpander;

    // check to see if we already have the address information
    // in our expander list
    while(thisExpander)
    {
      // if we do, then stop here
      if(currentDiscover->AttachedSASAddress ==
          thisExpander->SASAddress)
      {
        break;
      }

      // setup the pointer references
      previousExpander = thisExpander;
      thisExpander = thisExpander->Next;
    }

    // if we did not have the expander in our list, then get
    // the information
    if(!thisExpander)
    {
      // discover all the details about the attached expander
      // and insert into the master list
      thisExpander =
        DiscoverExpander(PhyIdentifier,
                         SourceSASAddress,
                         currentDiscover->AttachedSASAddress);

      // if we got the discover information, then add it to the
      // list
      if(thisExpander)
      {
        previousExpander->Next = thisExpander;

        // go through the configure cycle progressively ascending
        // to each expander starting at "thisExpander"
        ConfigureExpander(PhyIdentifier,
                          SourceSASAddress,
                          HubExpander,
                          thisExpander);
      }
    }
  }
}

// look for subtractive phys with edge or fanout devices attached...
else if(DeviceSet &&
  (currentDiscover->RoutingAttribute == SUBTRACTIVE) &&
  ((currentDiscover->AttachedDeviceType == EDGE_EXPANDER_DEVICE) ||
   (currentDiscover->AttachedDeviceType == FANOUT_EXPANDER_DEVICE)))
{
if (*DeviceSet == 0)
{
    struct TopologyTable *thisExpander = currentExpander;
    struct TopologyTable *previousExpander = currentExpander;

    // check to see if we already have the address information
    // in our expander list
    while (thisExpander)
    {
        // if we do, then stop here
        if (!memcmp(&currentDiscover->AttachedSASAddress,
            &thisExpander->SASAddress,
            8))
        {
            break;
        }

        // setup the pointer references
        previousExpander = thisExpander;
        thisExpander = thisExpander->Next;
    }

    // if we did not have the expander in our list, then get
    // the information
    if (!thisExpander)
    {
        // discover all the details about the attached expander
        // and insert into the master list
        thisExpander =
            DiscoverExpander(PhyIdentifier,
            SourceSASAddress,
            currentDiscover->AttachedSASAddress);

        // if we got the discover information, then set it as the
        // other device set
        if (thisExpander)
        {
            *DeviceSet = thisExpander;
        }
    }
}

    // move to the next phy on this expander
    phyIndex++;
} while (phyIndex <
    currentExpander->Device.NumberOfPhys);

    // cycle to the next expander to discover
    currentExpander = currentExpander->Next;
}

    // return the top of expander list
    return (HubExpander);
}

    // this routine will append the leaf to the tree domain
    static void ConcatenateSASDomains(struct TopologyTable *Tree,
        struct TopologyTable *Leaf)
    {
        while (Tree)
        {
            if (Tree->Next == 0)
            {
                Tree->Next = Leaf;
            }
        }
    }
break;
}
      
      Tree = Tree->Next;
    
  }
}

// validate the route table entries for all expanders
static int ValidateRouteTables(byte PhyIdentifier, 
    quadword SourceSASAddress, 
    struct TopologyTable *Expander, 
    int SASCompatibility)
{
  struct ReportRouteInformation *route;

  // buffers used to request and return SMP data
  struct SMPRequest request = { 0 },
  struct SMPResponse response = { 0 };

  byte phyIndex;
  word routeIndex;

  byte openStatus = OPEN_ACCEPT;
  int valid = 1;

  if(SASCompatibility == SAS_10_COMPATIBLE)
  {
    // this is just a pointer helper
    route = &(response.Response.ReportRouteInformation.Result);
  }

  // walk the list of expanders
  while(valid && 
    Expander)
  {
  
    if(Expander->Device.ConfigurableRouteTable)
    {
      struct RouteTableEntry *entry;

      word expanderRouteIndexes;

      _swab((char *)&Expander->Device.ExpanderRouteIndexes,
            (char *)&expanderRouteIndexes,
            sizeof(word));

      for(phyIndex = 0;
          (valid &&
            (phyIndex < Expander->Device.NumberOfPhys));
            phyIndex++)
      {
        // loop through all the phys of the expander
        for(routeIndex = 0;
            (valid &&
              ((routeIndex < 
                Expander->RouteIndex[phyIndex]) &&
                (routeIndex <
                expanderRouteIndexes))));
          routeIndex++)
        {
          openStatus = OPEN_ACCEPT;

          // report the route indexes for the expander
          SMPRequest(PhyIdentifier,
              SourceSASAddress,
              Expander->SASAddress,
if((openStatus != OPEN_ACCEPT) ||
    (response.FunctionResult !=
    SMP_FUNCTION_ACCEPTED))
{
    break;
}

entry = &(Expander->RouteTable[phyIndex][routeIndex]);

if((memcmp(&entry->RoutedSASAddress,
            &route->RoutedSASAddress,
            8)) ||
   (entry->ExpanderRouteEntryDisabled !=
    route->ExpanderRouteEntryDisabled))
{
    valid = 0;
}
}

Expander = Expander->Next;
}

return(valid);
}

// validate that the change count for the hub expander is still the same
// as when we started

static int ChangeCount(byte PhyIdentifier,
                        quadword SourceSASAddress,
                        struct TopologyTable *Expander)
{
    // buffers used to request and return SMP data
    struct SMPRequest request = { 0 };
    struct SMPResponse response = { 0 };

    int change = 0;

    byte openStatus = OPEN_ACCEPT;

    // get the report general information for the expander
    SMPRequest(PhyIdentifier,
               SourceSASAddress,
               Expander->SASAddress,
               &request,
               &response,
               &openStatus,
               REPORT_GENERAL);

    // don't worry about too much in the 'else' case for this example,
    // production code must handle
    if((openStatus == OPEN_ACCEPT) &&
       (response.FunctionResult == SMP_FUNCTION_ACCEPTED))
    {
            Expander->Device.ExpanderChangeCount)
void DeleteSASDomain(struct TopologyTable **Expander)
{
    struct TopologyTable *expander = *Expander;
    struct TopologyTable *nextExpander = 0;

    // walk the list of expanders
    while(expander)
    {
        nextExpander = expander->Next;

        free(expander);
        expander = nextExpander;
    }

    *Expander = 0;
}

// the application client for the initiator device would make a call to
// this function to begin the discover process...
// to simplify the setup for the simulation, the DiscoverProcess will get
// the Initiator number to allow multiple initiators...
void DiscoverProcess(byte Initiator,
    byte PhyIdentifier)
{
    // check to see if an expander is attached
    if((Attached[Initiator].DeviceType == EDGE_EXPANDER_DEVICE) ||
        (Attached[Initiator].DeviceType == FANOUT_EXPANDER_DEVICE))
    {
        struct TopologyTable *connectedExpander;

        // get some local variables to keep things simple
        quadword sourceSASAddress = Iam[Initiator].SASAddress;
        quadword destinationSASAddress = Attached[Initiator].SASAddress;

        // expander is attached, so begin by getting the information about
        // the connected expander
        connectedExpander = DiscoverExpander(PhyIdentifier,
            sourceSASAddress,
            destinationSASAddress);

        // make sure we get the information from the expander
        if(connectedExpander)
        {
            struct TopologyTable *thisDeviceSet;
            struct TopologyTable *attachedDeviceSet = 0;

            int redoDiscover = 0;
            int changed = 0;

            do
            {
                // go upstream on the subtractive phys until we discover that we
                // are attached to another subtractive phy or a fanout expander
                // then begin the discover process from that point, this works
                // because any new address that we find will naturally move
                // upstream due to the subtractive addressing method
            } while(redoDiscover == 0);
        }
    }
if during the discover cycle, it is determined that there are two device sets connected, then a second discover and configuration cycle is required for the other device set:

```
thisDeviceSet = FindBoundary(PhyIdentifier, sourceSASAddress, connectedExpander, &attachedDeviceSet);
```

// set the root for the domain as the subtractive boundary
```
if(thisDeviceSet)
{
    // output a little information about the subtractive boundary
    TracePrint("subtractive boundary at \%s\n", (char *)&thisDeviceSet->SASAddress);
    SASDomain[Initiator] = thisDeviceSet;
}
```

// if there was no subtractive boundary, then the root is the expander connected to the initiator
```
else
{
    // output a little information about the subtractive boundary
    TracePrint("connected expander at \%s\n", (char *)&connectedExpander->SASAddress);
    SASDomain[Initiator] = connectedExpander;
}
```

// begin the discover and configuration cycle
```
DiscoverAndConfigure(PhyIdentifier, sourceSASAddress, SASDomain[Initiator], &attachedDeviceSet);
```

// if two device sets are connected, then the attached device set has to be discovered and configured
```
if(attachedDeviceSet)
{
    // output a little information about the attached device set
    TracePrint("attached device set at \%s\n", (char *)&attachedDeviceSet->SASAddress);

    // discover and configure the attached device set
    DiscoverAndConfigure(PhyIdentifier, sourceSASAddress, attachedDeviceSet, 0);

    // put the domains together
    ConcatenateSASDomains(SASDomain[Initiator], attachedDeviceSet);
}
```

// if the change count of the top most expander is different from when we started, then the topology was not stable so do the discover again
```
changed = ChangeCount(PhyIdentifier, sourceSASAddress, SASDomain[Initiator]);
```

// if we used the route table optimization, check the route tables for each expander phy, if they are incorrect then change back to the original discover algorithm and redo discover, continue to use the original algorithm for any new discover
```
if(!changed && (DiscoverAlgorithm == SAS_UNIQUE_LEVEL_DESCENT) &&
    ``
!ValidateRouteTables(PhyIdentifier,
    sourceSASAddress,
    SASDomain[Initiator],
    SASCompatibility))

{ redoDiscover = 1;

    // if the change count of the top most expander is the same
    // as when we started the validation of the route tables
    // then the topology was stable, so change the algorithm
    // before the rediscover
    if(!ChangeCount(PhyIdentifier,
        sourceSASAddress,
        SASDomain[Initiator]))
    {
        DiscoverAlgorithm = SAS_SIMPLE_LEVEL_DESCENT;
    }

    // delete everything allocated and start over
    DeleteSASDomain(&SASDomain[Initiator]);
}

} while(redoDiscover ||
    changed);
} }