IEEE 1394 Configuration ROM Decoder

Bill Alexander

Short Biography

Bill Alexander is a Sr. Software Engineer at LSI Logic (formerly Symbios Logic). His current responsibilities include developing 1394 device drivers, software and utilities. He can be contacted through his web page at http://www.geocities.com/SiliconValley/Haven/4824.

History & Introduction

I started developing software in the IEEE 1394 (a.k.a. FireWire) arena about two years ago. Symbios Logic (now LSI Logic) was developing the SYM13FW500 chip, which is a 1394-to-ATA/ATAPI bridge that allows IDE drives to be connected to a system in a hot Plug-and-Play fashion. At the time there were not many 1394 development tools. The Open Host Controller Interface (OHCI) initiative was still coming up to speed and the only 1394 PCI controllers on the market were TI's TSBKPCI and Adaptec's AHA-8940. My group decided to standardize on the TI TSBKPCI because of the robust and reliable software development environment (called LynxSoft) included with the controller. The main parts of LynxSoft are a Windows 95 VxD (PCILYNX.VXD) and a DLL (CSL1394.DLL) which work together to provide developers with a powerful API in which to begin 1394 development.

One utility I wrote for the SYM13FW500 project is a configuration ROM dumper and decoder. This utility also works for any peripheral device connected to the TSBKPCI. This article describes the source code to the DUMPROM utility.

The IEEE 1394 Addressing Scheme

Before getting started with the source code, I need to explain the basic 1394 addressing scheme. 1394 is a serial bus design in which nodes participate in making up a 64-bit address space. When the bus is initialized or reset, each node is assigned a node number. A node number is used as part of the 64-bit address to access the memory space on a specific node. An address is made of three fields: bus_number (bits 54–63), node_number (bits 48-53), and address_offset (bits 0-47). Currently, nobody is routing from one 1394 bus to another, so the bus_number that you will always see is 0x3FF (local bus). Node_number ranges from 0 to 0x3F (64 nodes). 0x3F is a special node number denoting a broadcast to all nodes on a specified bus_number, so there are only 63 useable nodes possible on a single bus. Address_offset ranges from 0x0000:00000000 to 0xFFFF:FFFFFFFF which gives you up to 256 TB (terabytes) of address space for a single node. I use the ':' character to help differentiate the upper 16 bits from the lower 32-bits of the address_offset.

Inside of this 256 TB address space on a node, 1394 defines an address map for three sub-spaces: initial memory space (0x0000:00000000 to 0xFFFF:DFFFFFFF), private space (0xFFFF:E0000000 to 0xFFFF:EFFFFFFF) and register space (0xFFFF:F0000000 to 0xFFFF:FFFFFFFF). The configuration ROM is defined to start at offset 0x400 within the register space (i.e. address_offset 0xFFFF:F0000400).

From this point on, I will not deal with bus_number and node_number. When I say "address", I really mean the 48-bit "address_offset".

The IEEE 1394 Configuration ROM Format

The IEEE 1394 configuration ROM is based on another IEEE standard called 1212. The 1212 specification defines a generic control and status register (CSR) architecture for microcomputer buses. One part of the specification defines the configuration ROM definition. The 1394 configuration ROM definition is a specific implementation of 1212. The Serial Bus Protocol 2 (SBP-2) is another 1394-related specification that defines how to communicate with storage devices on a 1394 bus. The SBP-2 configuration ROM definition is a specific implementation of 1394. Figure 1 shows the relationship between the three configuration ROM specifications.

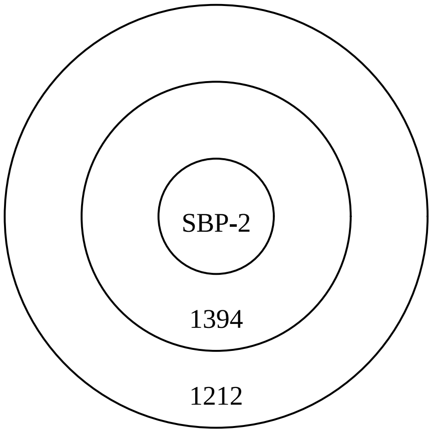


Figure 1. 1212, 1394 and SPB-2 Configuration ROM Specification Relationship.

The purpose of the configuration ROM is to provide the system with information about a "node" on the bus. The information usually consists of the global unique ID (GUID), the device type, the device name, the manufacturer name, the model number, the device's capabilities, etc. A configuration ROM is made up of a series of blocks. These blocks are in turn made up of entities which 1394 calls quadlets (4-bytes). The blocks are logically arranged in a hierarchical structure with the upper blocks referring to subsequent blocks. A general 1394 configuration ROM format is shown in figure 2.

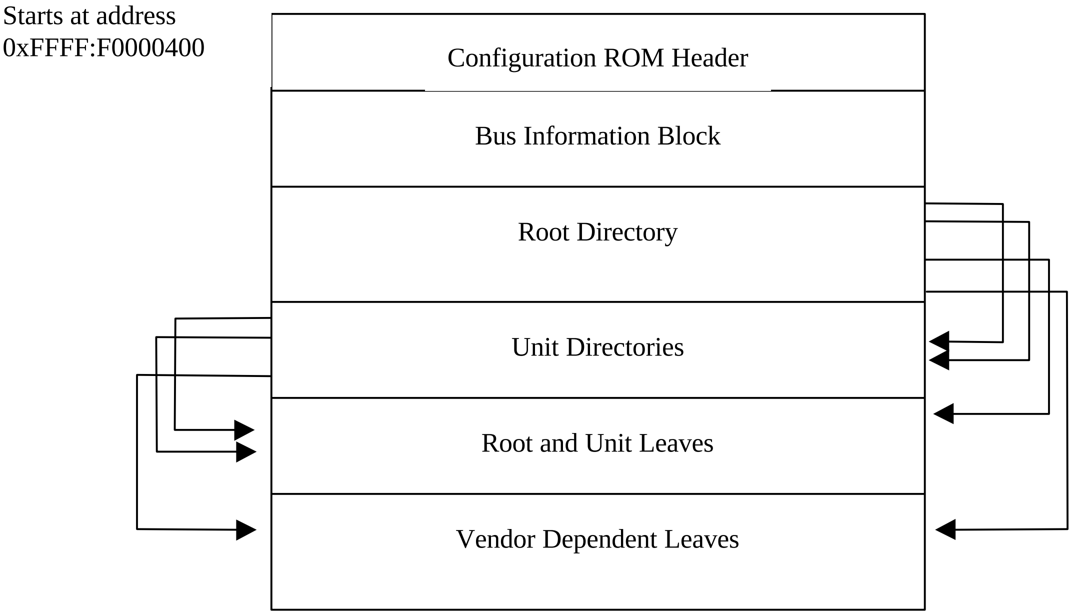


Figure 2. General IEEE 1394 Configuration ROM Format

By definition, all 1394 configuration ROMs must start at address 0xFFFF:F0000400 on the device. The first quadlet is the Configuration ROM Header structure. This structure is made up of three fields: info_length (bits 24-31), crc_length (bits 16-23) and cfg_rom_CRC (bits 0-15). The info_length field defines the length in quadlets of the Bus Information Block, which starts immediately after the header. The crc_length field indicates the number of quadlets following the header that make up the configuration ROM. The cfg_rom_CRC is a CRC-16 value calculated on the quadlets referred to by the crc_length. The Configuration ROM Header is shown in figure 3.

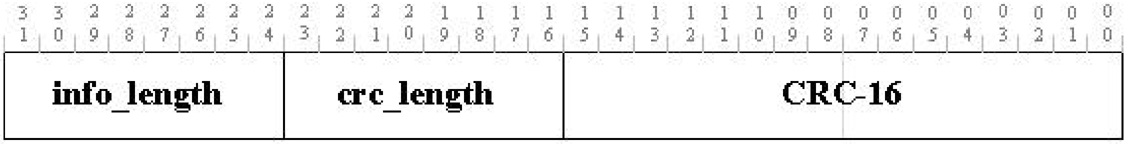


Figure 3. Configuration ROM Header.

The Bus Information Block is a four-quadlet structure. The first quadlet is the bus_name, which is always set to 0x31333934 (ASCII “1394”). The second quadlet defines the node capabilities. The last two quadlets of the Bus Information Block are made up of the node_vendor_id, chip_id_hi and chip_id_lo fields. Together there fields provide a global unique ID (GUID) for the node. The structure of the Bus Information Block is shown in Figure 4.

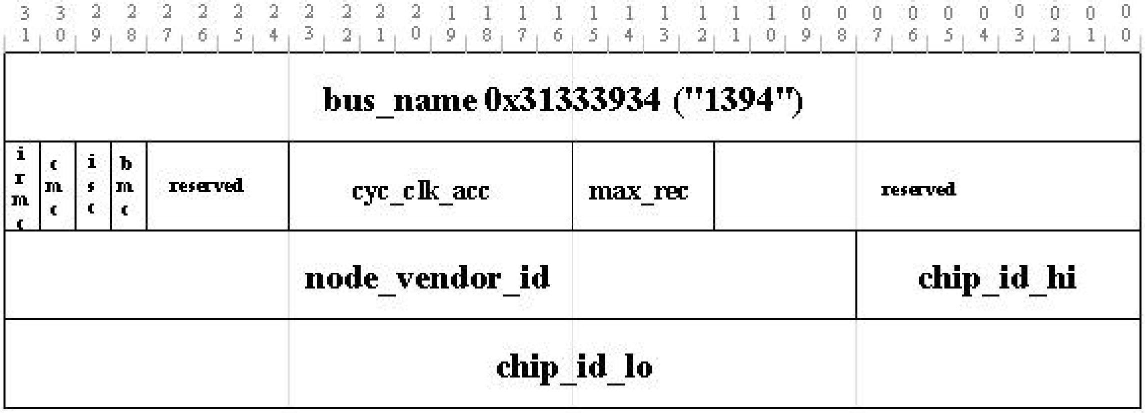


Figure 4. IEEE 1394 Bus Information Block.

The Root Directory Block immediately follows the Bus Information Block. The Root Directory is a collection of directory entries that both describe the node and refer to other directory and leaf blocks in the configuration ROM. At the beginning of the Root Directory Block (and all other blocks) is a Block Header structure. This structure is made up of two fields: length (bits 24-31) and CRC-16 (bits 0-23). The length field indicates the block's length in quadlets (excluding the header) and the CRC-16 value calculated for these quadlets. Figure 5 shows the structure of a Block Header.

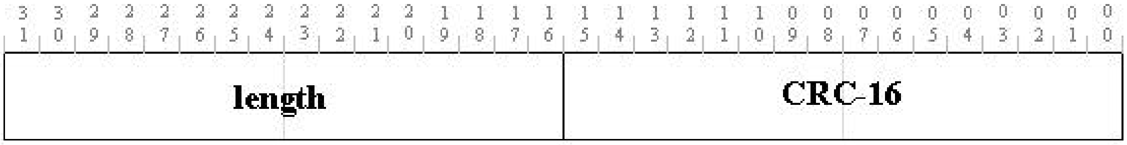


Figure 5. Block Header.

After the header, the directory entries begin. Each directory entry is a quadlet made up of three fields: key_type (bits 30-31), key_value (bits 24-29) and value (bits 0-23). The key_type is really a misnomer: it has nothing to do with the type of data being described. Rather key_type defines the way in which the data is referenced. If key_type is 0, the value field is an immediate data value. Key_types of 1, 2 and 3 turn the value field into a pointer, which is really an offset counted in quadlets. Key_type 1 means that the value field is a offset relative to the node's 1394 CSR space (i.e. address 0xFFFF:F0000000). For example, a value of 0x000001 indicates that the data is located at address 0xFFFF:F0000004 on the node. Key_type 2 makes the value field a pointer to a leaf block. The pointer is offset relative to the directory entry's position in the configuration ROM. For example, if the address of a director entry is 0xFFFF:F0000440 and the value field is 0x000002, then the leaf block is located at address 0xFFFF:F0000448. Key_type 3 is the same as key_type 2 except that the block being pointed to is a directory instead of a leaf. The key_value field dictates the kind of data being referenced. For example, key_value 0x03 means that data is the Module_Vendor_Id. Figure 5 shows the structure of a directory entry.

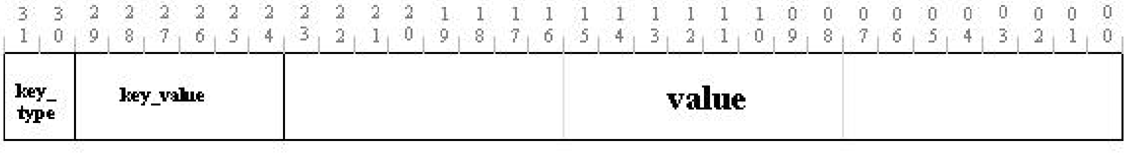


Figure 5. Directory entry structure.

One important point is that the key_value usually restricts the values of the key_type field. For instance, the key_value Module_Vendor_Id (0x03) restricts key_type to 0 (immediate data value). Logically it does not make sense to have a Mode_Vendor_Id directory. Another example is key_value Unit_Dependent_Info (0x14). Under 1212, Unit_Dependent_Info key_types can be either 2 (leaf) or 3 (directory). No immediate value is possible as defined by 1212. However, you know how specifications evolve and need dictates that we utilize available space. So under SBP-2, key_value 0x14 is redefined as Logical_Unit. If key_type is 0 (immediate), the value is a Logic_Unit_Number structure. If key_type is 3 (directory), the value field is a pointer to a Logical_Unit_Directory. Now enters the exception: if key_type is 1 (CSR offset), the value field is offset in the CSR space of the Management_Agent register -- not of a Logic_Unit.

Sometimes the key_type and key_value fields are combined and referred to as the key field. In fact the SBP-2 specification only refers to the "concatenation" of the key_type and key_value fields. It uses the combination to avoid confusion over the exceptions it defines in the key_value field. Thus for SBP-2, key

0x14 is a Logical_Unit_Number, key 0xD4 is a Logical_Unit_Directory and key 0x94 is the Management_Agent.

DUMPROM Utility Operation

The DUMPROM utility is a Win32 console application that calls TI's CLS1394.DLL. Running DUMPROM without any parameters will simply dump the configuration ROM of the first 1394 device other than the controller. Listing 1 shows the output from DUMPROM with the SYM13FW500.

```
Raw Data Dump of the Configuration ROM from device number 1 (id=00A0B800)

1394 Addr Off    Data
-------------    --------  --------  --------  --------
FFFF:F0000400    042E19A8  31333934  00FF5000  00A0B800
FFFF:F0000410    00005000  00067A52  0C0083C0  0300A0B8
FFFF:F0000420    8100000F  0400500A  81000015  D1000001
FFFF:F0000430    00067752  1200609E  13010483  5400C000
FFFF:F0000440    3A401E08  14000000  D4000001  000329A2
FFFF:F0000450    0400500A  8100000A  8200000E  0007BD5D
FFFF:F0000460    00000000  00000000  53594D42  494F5320
FFFF:F0000470    4C4F4749  432C2049  4E432E00  00044469
FFFF:F0000480    00000000  00000000  53594D42  494F5300
FFFF:F0000490    000AE09E  00000000  00000000  53594D31
FFFF:F00004A0    33465735  30302D44  49534B20  44524956
FFFF:F00004B0    45000000  00000000  00000000

Decode of the Configuration ROM from device number 1 (id=00A0B800)

1394 Addr Off Quadlet  Meaning
------------- -------- --------------------------------------------------

Confiruation ROM Header
FFFF:F0000400 042E19A8 info_length=04, crc_length=2E, rom_crc_value=19A8

Bus_Info_Block
FFFF:F0000404 31333934 bus_name=31333934 ("1394")
FFFF:F0000408 00FF5000 irmc=0, cmc=0, isc=0, bmc=0, cyc_clk_acc=FF, max_rec=5
FFFF:F000040C 00A0B800 node_vendor_id=00A0B8, chip_id_hi=00
FFFF:F0000410 00005000 chip_id_lo=00005000

Root_Directory
FFFF:F0000414 00067A52 length=0006, crc=7A52
FFFF:F0000418 0C0083C0 Node_Capabilities spt 64 fix lst drg
FFFF:F000041C 0300A0B8 Module_Vendor_Id 00A0B8
FFFF:F0000420 8100000F Textual_Descriptor leaf ind_off=00000F (FFFF:F000045C)
FFFF:F0000424 0400500A Module_Hw_Version 00500A
FFFF:F0000428 81000015 Textual_Descriptor leaf ind_off=000015 (FFFF:F000047C)
FFFF:F000042C D1000001 Unit_Directory directory ind_off=000001 (FFFF:F0000430)

Unit_Directory directory referenced from FFFF:F000042C
FFFF:F0000430 00067752 length=0006, crc=7752
FFFF:F0000434 1200609E Unit_Spec_Id 00609E
FFFF:F0000438 13010483 Unit_Sw_Version 010483
FFFF:F000043C 5400C000 Management_Agent crc_offset=00C000 (FFFF:F0030000)
FFFF:F0000440 3A401E08 Unit_Characteristics 401E08
FFFF:F0000444 14000000 Logical_Unit_Number o=0, device_type=00, lun=0000
FFFF:F0000448 D4000001 Logical_Unit_Directory directory ind_off=000001 (FFFF:F000044C)

Logical_Unit_Directory directory referenced from FFFF:F0000448
FFFF:F000044C 000329A2 length=0003, crc=29A2
FFFF:F0000450 0400500A Module_Hw_Version 00500A
FFFF:F0000454 8100000A Textual_Descriptor leaf ind_off=00000A (FFFF:F000047C)
FFFF:F0000458 8200000E Bus_Dependent_Info leaf ind_off=00000E (FFFF:F0000490)

Textual_Descriptor leaf referenced from FFFF:F0000420
FFFF:F000045C 0007BD5D length=0007, crc=BD5D
FFFF:F0000460 00000000 ....
FFFF:F0000464 00000000 ....
FFFF:F0000468 53594D42 SYMB
FFFF:F000046C 494F5320 IOS
FFFF:F0000470 4C4F4749 LOGI
FFFF:F0000474 432C2049 C, I
FFFF:F0000478 4E432E00 NC..

Textual_Descriptor leaf referenced from FFFF:F0000454
FFFF:F000047C 00044469 length=0004, crc=4469
FFFF:F0000480 00000000 ....
FFFF:F0000484 00000000 ....
FFFF:F0000488 53594D42 SYMB
FFFF:F000048C 494F5300 IOS.

Bus_Dependent_Info leaf referenced from FFFF:F0000458
FFFF:F0000490 000AE09E length=000A, crc=E09E
FFFF:F0000494 00000000 ....
FFFF:F0000498 00000000 ....
FFFF:F000049C 53594D31 SYM1
FFFF:F00004A0 33465735 3FW5
FFFF:F00004A4 30302D44 00-D
FFFF:F00004A8 49534B20 ISK
FFFF:F00004AC 44524956 DRIV
FFFF:F00004B0 45000000 E...
FFFF:F00004B4 00000000 ....
FFFF:F00004B8 00000000 ....
```

Listing 1. Output from DUMPROM for the SYM13FW500 device.

The output is divided into two parts. The first part is the raw quadlet data dump. This section shows nothing but raw 32-bit hexadecimal data. The second section is the decode of the ROM data. The decode is broken into blocks with each block delimited by a blank line and a description. The beginning of each line starts with the address followed by a quadlet value. Then the meaning of the quadlet is displayed.

DUMPROM Source Code

One of the first things that DUMPROM does is load TI's CLS1394.DLL. The program does this via the LoadLibraryAndFixPointer() function, which calls the two Win32 APIs LoadLibrary() and GetProcAddress(). The function first call LoadLibrary() to load the DLL and then uses GetProcAddress() to fetch the addresses of various functions. The functions that DUMPROM needs to call are:

cls1394Initialize()
cls1394Terminate()
cls1394CreateFile()
cls1394CloseHandle()
cls1394DeviceIoControl
GetDeviceInfo()

You may be wondering why I did not link with the CLS1394.LIB import library and avoid the hassle of LoadLibrary() and GetProcAddress(). The reason is that CLS1394.LIB and CLS1394.DLL are incompatible between versions 2.1 and 2.2. The problem stems from the fact the 2.1 and 2.2 were developed for different revisions of the Lynx hardware, namely the TSBKPCI and the TSBKPCI403. The LynxSoft 2.1 does not work with the TSBKPCI403 and vise versa. In spite of the incompatibilities, the API between the versions did not change significantly. So using the LoadLibrary() and GetProcAddress() functions allows me to build one program which will work with both versions.

Once the CLS1394.DLL is loaded, DUMPROM initializes the DLL by calling cls1394Initialize(). Next DUMPROM obtains a handle to a 1394 device. The DeviceOpen() function calls GetDeviceInfo() to get the device ID of the first device on the bus. Next, DeviceOpen() calls cls1394CreateFile() get the handle to the device.

The device handle allows you to now communicate with the device. DUMPROM needs only to read quadlets from the device, which it does through the ReadQuadlet() function. The ReadQuadlet() function is shown in Listing 2.

```
BOOL ReadQuadlet(
        cls1394HANDLE hDev,
        LARGE_INTEGER liAddress,
        QUADLET *qValue)
{
        clsAsyncRead aw;
        BOOL rc;

        // Setup request structure
        aw.DestinationAddress = &liAddress;
        aw.nNumberOfBytesToRead = sizeof(QUADLET);
        aw.nBlockSize = 0;
        aw.fulFlags = 0;
        aw.lpBuffer = qValue;

        // Call driver
        rc = lpfnCls1394DeviceIoControl(
                hDev,
                CLS_REQUEST_ASYNC_READ,
                &aw,
                sizeof(clsAsyncRead),
                0,
                0,
                0,
                0);
        return (rc);
}
```

Listing 2. The ReadQuadlet() funtion.

Before DUMPROM can display and decode the configuration ROM, it must first determine the size of the ROM and allocate a buffer to hold it in memory. The AllocateConfigRomBuffer() function performs this operation. AllocateConfigRomBuffer() reads the first quadlet in the configuration ROM, which contains the size in quadlets of the ROM (crc_length). It then allocates a buffer with the size = (crc_length+1)*sizeof(QUADLET). Next, ReadConfigRomBuffer() reads the entire ROM into the buffer. In a FOR loop, ReadConfigRomBuffer() the starts at address 0xFFFF:F0000400 and reads one quadlet at a time until it has read the ROM, incrementing the address by sizeof(QUADLET) for each iteration.

Next, DUMPROM performs the raw quadlet data dump of the buffer. There is nothing exciting here, so I'll move along to the decode process.

The DecodeConfigRom() function parses and decodes the configuration ROM buffer on quadlet at a time. To facilitate single-pass parsing, the routine allocates a table (called blockInfo) of BlockInfoT structures which has one element for every quadlet in the configuration ROM. There are three fields in this structure: BlockType, ReferringDirEntry and ReferringDirEntryIdx. The BlockType field indicates that the corresponding quadlet in the ROM is the beginning of a new block. BlockType BT_NONE (0) indicates that the quadlet is ordinary element of the current block. BlockType BT_DIRECTORY (1) indicates that the quadlet is the beginning of a directory block. BlockType BT_LEAF (2) indicates that the quadlet is the beginning of a leaf block. The ReferringDirEntry field is a copy of the directory entry which refers to this specific block. ReferringDirEntryIdx is the quadlet offset from the beginning of the ROM of the directory entry that refers to this block. The BlockInfoT definition is shown Listing 3.

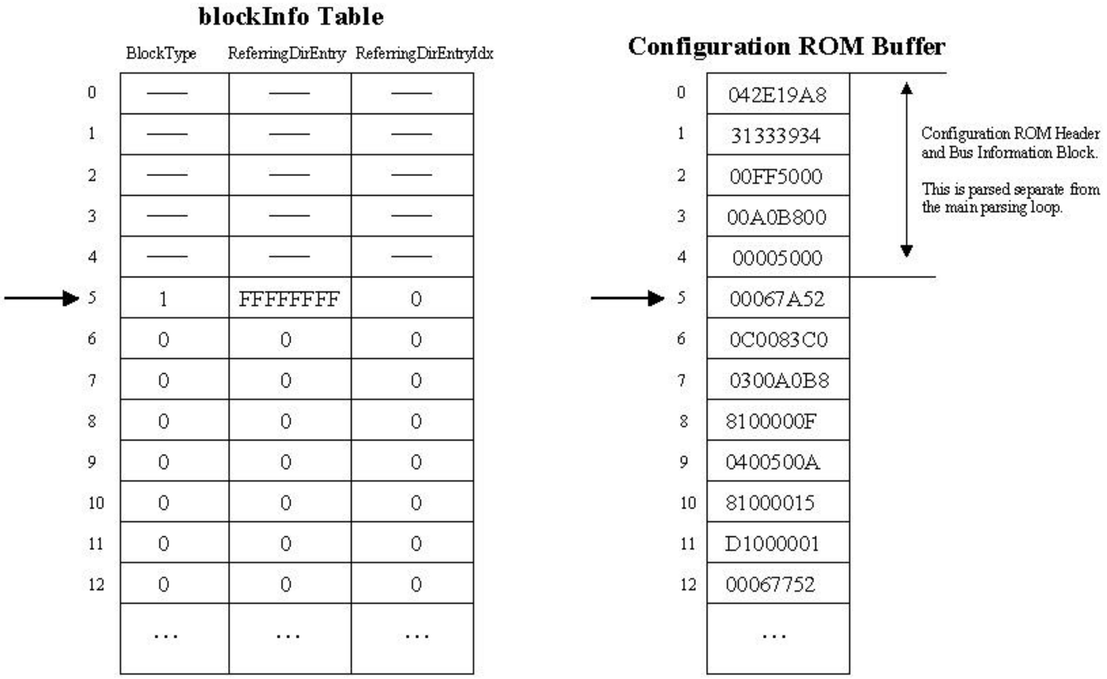
```
// Block Types
#define BT_NONE                 0
#define BT_DIRECTORY            1
#define BT_LEAF                 2

// Block Information Structuretypedef struct BlockInfoStruct
{
        DWORD           BlockType;              // NONE, DIRECTORY or LEAF
        DirEntryT       ReferringDirEntry;      // Copy of the directory entry quadlet
        DWORD           ReferringDirEntryIdx;   //Quadlet offset from 0xFFFF:F0000400
} BlockInfoT, *PBlockInfoT;
```

Listing 3. The BlockInfoT structure definition.

The idea of the blockInfo table is to record block information when references are parsed in directory blocks. Initially, the entire blockInfo table is zeroed, thus indicating that all of the quadlets are elements of the current block. The Configuration ROM Header and Bus Information Block are special exceptions to the algorithm. These are parsed up front apart from the directories that follow.

The Root Directory is where the algorithm starts its main parsing loop. Before entering the parsing loop, however, the blockInfo table entry for the Root Directory block is initialized by setting the BlockType field to BT_DIRECTORY (1) and the ReferringDirEntry field to DT_ROOT (0xFFFFFFFF). The initial blockInfo table is shown in Figure 6.



blockInfo Table

	BlockType	ReferringDirEntry	ReferringDirEntryIdx
0	——	——	——
1	——	——	——
2	——	——	——
3	——	——	——
4	——	——	——
5	1	FFFFFFFF	0
6	0	0	0
7	0	0	0
8	0	0	0
9	0	0	0
10	0	0	0
11	0	0	0
12	0	0	0
	...	...	...

Configuration ROM Buffer

0	042E19A8
1	31333934
2	00FF5000
3	00A0B800
4	00005000
5	00067A52
6	0C0083C0
7	0300A0B8
8	8100000F
9	0400500A
10	81000015
11	D1000001
12	00067752
	...

Configuration ROM Header and Bus Information Block.

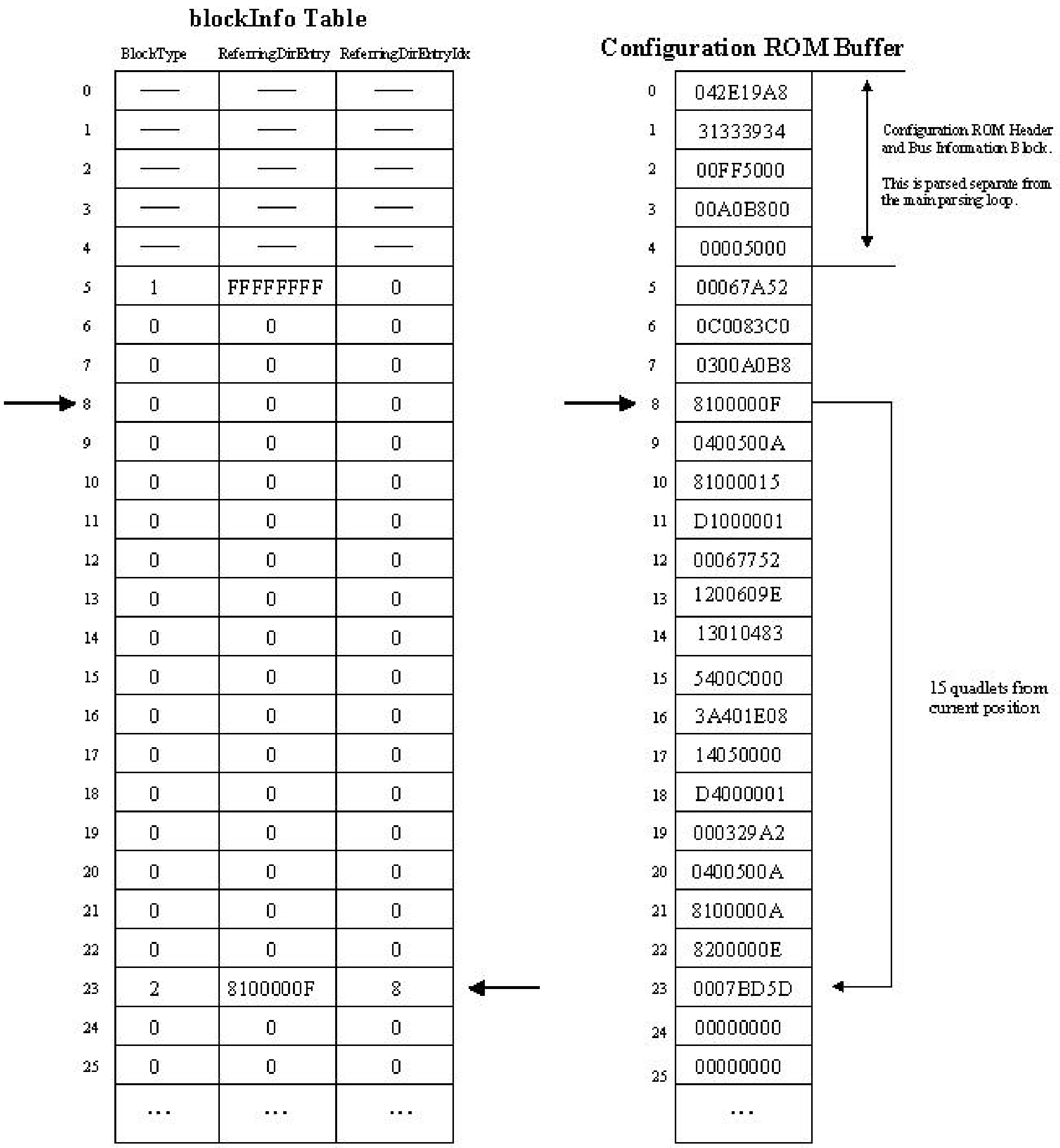
This is parsed separate from the main parsing loop.

Figure 6. Initial blockInfo Table.

The parsing loop begins at index 5. The loop examines the BlockType field and finds BT_DIRECTORY (1), which indicates the beginning of a new directory block. The header is parsed and the loop prints a new line, the block type, a back reference, the 1394 address, the quadlet value and the decode information (see Listing 2). The block type is determined by looking at the ReferringDirectoryEntry. In this case, 0xFFFFFFFF indicates the Root Directory block, which has no back reference since it is the top of the hierarchy. The decode information for a block header consists of a 16-bit length field followed by a CRC-16 field. Next, the parsing loop sets the curBlockType variable to the BlockType field. The curBlockType variable is used by the loop to keep track of the current parsing state on subsequent iterations.

On the next iteration of the parsing loop (index 6), the BlockType is now BT_NONE (0), which indicates that the current quadlet in the ROM buffer is an element of the current block. Since the curBlockType is BT_DIRECTORY, the quadlet is decoded as a directory entry. Our example in Figure 6 shows that the directory entry is 0x0C0083C0. The key_type field is 0 meaning this is an immediate value. The key_value field is 0x0C, which indicates a Node_Capabilites value. The value 0x083C0 is a bitmap of 1394 node capabilities with the spt, 64, fix, lst and drg bits set. On the next iteration (index 7), BlockType is also BT_NONE. So the directory entry 0x0300A0B8 is decoded to a Module_Vendor_Id of value 0x00A0B8. This is Symbios Logic's (now LSI Logic) registered IEEE 1394 vendor ID.

The quadlet at index 8 is a little more interesting. BlockType is still 0 and curBlockType is still BT_DIRECTORY, so this is a directory entry, but the quadlet 0x81000015 is a pointer. The key_type field is 2, so this is a leaf pointer. What kind of leaf? Key_value is 0x01 which is Textual_Descriptor. How far away from the current position? The value field is 0x00000F (15 decimal). So counting 15 quadlets from the current position, the newly detected leaf block starts at index 23. At index 23 in the blockInfo table, the parsing loop sets BlockType to BT_LEAF (2), ReferringDirEntry to 0x8100000F and ReferringDirEntryIndex to 8. Figure 7 shows the two tables at this point.



blockInfo Table

	BlockType	ReferringDirEntry	ReferringDirEntryIdx
0	——	——	——
1	——	——	——
2	——	——	——
3	——	——	——
4	——	——	——
5	1	FFFFFFFF	0
6	0	0	0
7	0	0	0
8	0	0	0
9	0	0	0
10	0	0	0
11	0	0	0
12	0	0	0
13	0	0	0
14	0	0	0
15	0	0	0
16	0	0	0
17	0	0	0
18	0	0	0
19	0	0	0
20	0	0	0
21	0	0	0
22	0	0	0
23	2	8100000F	8
24	0	0	0
25	0	0	0
	…	…	…

Configuration ROM Buffer

0	042E19A8
1	31333934
2	00FF5000
3	00A0B800
4	00005000
5	00067A52
6	0C0083C0
7	0300A0B8
8	8100000F
9	0400500A
10	81000015
11	D1000001
12	00067752
13	1200609E
14	13010483
15	5400C000
16	3A401E08
17	14050000
18	D4000001
19	000329A2
20	0400500A
21	8100000A
22	8200000E
23	0007BD5D
24	00000000
25	00000000
	…

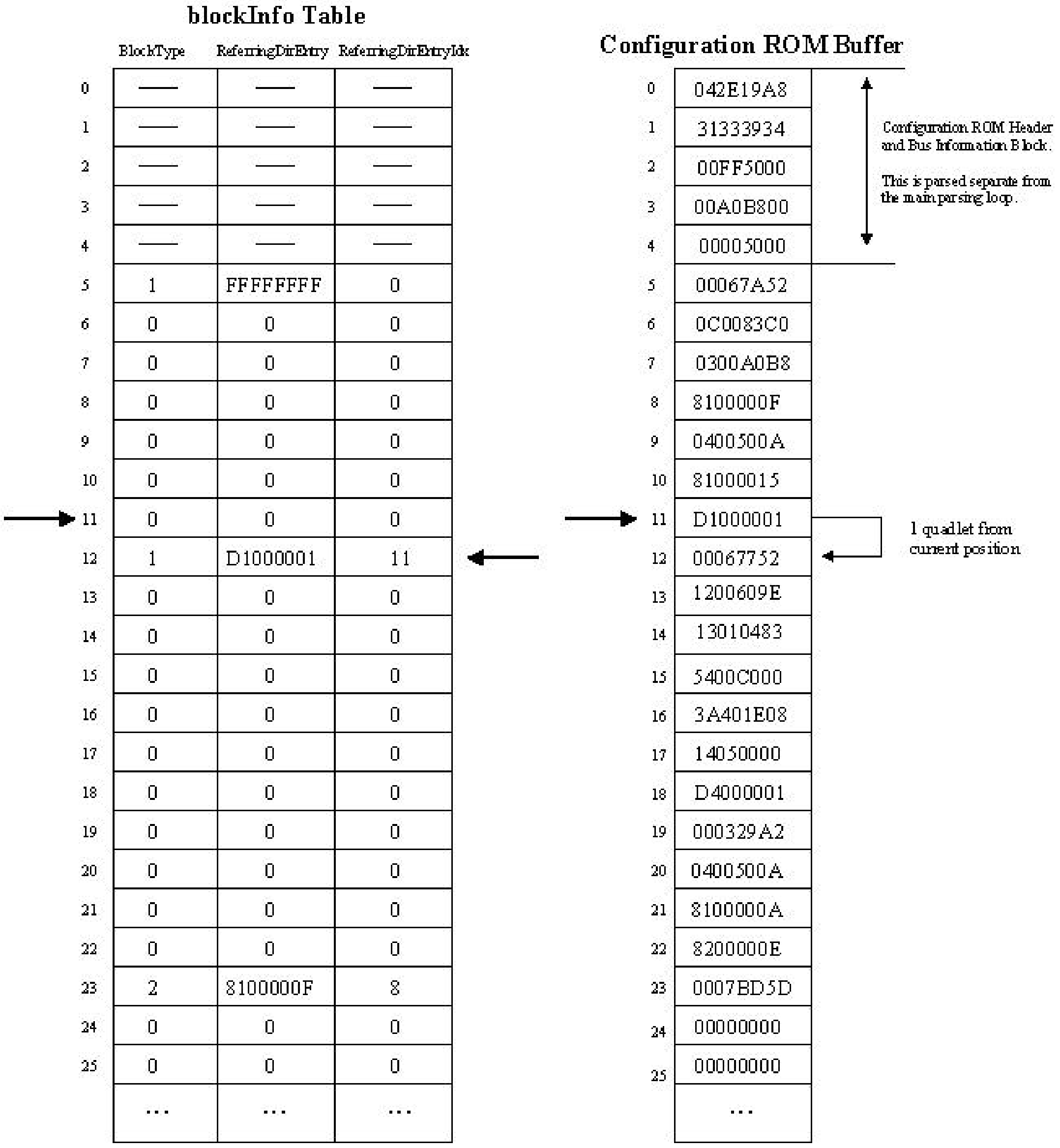
Configuration ROM Header and Bus Information Block.

This is parsed separate from the main parsing loop.

15 quadlets from current position

Figure 7. The leaf block starting at index 23 is referenced at index 8.

Parsing continues in this way one quadlet at a time. At index 11 key_type 3, key_value 0x14, value 0x000001 means a Unit_Directory is located 1 quadlet away (index 12). At index 12 in the blockInfo table, the parsing loop sets BlockType to BT_DIRECTORY (1), ReferringDirEntry to 0xD1000001 and ReferringDirEntryIdx to 11. Figure 8 shows what the tables look like at this point.



Configuration ROM Header and Bus Information Block.
This is parsed separate from the main parsing loop.
1 quadlet from current position

blockInfo Table

	BlockType	ReferringDirEntry	ReferringDirEntryIdx
0	——	——	——
1	——	——	——
2	——	——	——
3	——	——	——
4	——	——	——
5	1	FFFFFFFF	0
6	0	0	0
7	0	0	0
8	0	0	0
9	0	0	0
10	0	0	0
11	0	0	0
12	1	D1000001	11
13	0	0	0
14	0	0	0
15	0	0	0
16	0	0	0
17	0	0	0
18	0	0	0
19	0	0	0
20	0	0	0
21	0	0	0
22	0	0	0
23	2	8100000F	8
24	0	0	0
25	0	0	0
	...	...	...

Configuration ROM Buffer

0	042E19A8
1	31333934
2	00FF5000
3	00A0B800
4	00005000
5	00067A52
6	0C0083C0
7	0300A0B8
8	8100000F
9	0400500A
10	81000015
11	D1000001
12	00067752
13	1200609E
14	13010483
15	5400C000
16	3A401E08
17	14050000
18	D4000001
19	000329A2
20	0400500A
21	8100000A
22	8200000E
23	0007BD5D
24	00000000
25	00000000
	...

Figure 8. Unit Directory at index 12 referenced by directory entry at index 11.

On the next iteration, index 12, BlockType is 1. This signifies a transition from one block to another. In this case, it is another directory, the Unit_Directory. Processing of the Unit_Directory continues just as with the Root_Directory. I will now skip ahead to index 15 where the key_type is 1, meaning a pointer relative to the CSR base address 0xFFFF:F0000000. The key_value is 0x14, Management_Agent – NOT Unit_Directory! Remember that this is the SBP-2 exception to the uniform definitions of 1212. The value is 0x00C000, so the Management_Agent register is located at address 0xFFFF:F0000000 + (0x00C000 * sizeof(QUADLET)) = 0xFFFF:F0030000.

Skipping ahead to index 18, this directory entry references a Logical_Unit_Directory one quadlet away, so the parsing loop sets up the blockInfo at index 19. At index 19 a new block starts. At index 23, the Textual_Descriptor leaf block that was referenced by index 8 is finally reached. At this point, all elements in the block are parsed ASCII data even though Textual_Descriptor leaf blocks have their own structure.

Conclusion

The DUMPROM is a handy utility that allows you to examine the configuration ROM of any 1394 device. However, the program lacks some features that would make it even more useful. For instance, DUMPROM tends to be SBP-2 oriented and does not properly decode digital video information. This would be a nice enhancement since there is more interest in video on 1394 than in storage. You should be able to extend the code to include the digital video key_values with no trouble.

Another enhancement you could add is proper leaf block decoding. Currently DUMPROM decodes leaf quadlets as ASCII data. Decoding leaf blocks in their true structures would be a good improvement.

If you are interested in the 1212, 1394 or SBP-2 specifications, I recommend looking at the definitions in the source code and reading the specifications themselves.

If you are interested in developing with the TSBKPCI or TSBKPCI403, you can find more information on Texas Instruments' Web site at http://www.ti.com/sc/docs/msp/1394/evm/pci.htm and http://www.ti.com/sc/docs/msp/1394/evm/pci403.htm. You can also download the LynxSoft 2.1 and 2.2 development kits at http://www.ti.com/sc/data/msp/1394/evm/lynx21.exe and http://www.ti.com/sc/data/msp/1394/evm/lynx22.exe.

Acknowledgements

I would like to acknowledge Matt Pujol and Leo Grassens for their assistance in writing this article.

References

ISO/IEC 13212:1994(E) ANSI/IEEE Std 1212, 1994 Edition, pp 79-100.

IEEE Standard for a High Performance Serial BUS, IEEE Std 1394-1995, August 30, 1996, pp. 221-227.

ANSI T10, Project 1155D, Revision 4, May 19, 1998, pp. 43-52, 76-77 and 83-84.

Texas Instruments LynxSoft 1394 Software Application Programmer User's Guide, SLLU003, February 24, 1998, Version 2.2.

Tewell, Thomas. "FireWire: The IEEE 1394 Serial Bus," Dr. Dobbs Journal, #269, September, 1997, pp. 58-66.