OpenLCB for MERG CBUS Modules

The aim is to provide OpenLCB protocol software for MERG CBUS hardware modules.

CBUS modules are currently sold as kits with CBUS software in the PIC. This project is to try and provide replacement software which is compatible with OpenLCB. Currently this means that users have to build the kit themselves, program the PIC chip, and then set the NodeID. All OpenLCB modules need a unique NodeID which will usually be set by the maker. This can be assigned by the kit builder from their MERG or NMRA membership number.

Every module has 2 LEDs, a Green one on pin 28, and a Yellow one on pin 27. (Some diagrams have the LEDs reversed or undefined.)
The green LED comes on when the module has initialized.
The yellow LED and green LED are on when the boot loader is being executed.
The yellow LED pulses on when any event packet or a DAA packet to this module is seen.

One of the problems of new users is getting their first event produced and acted upon. They usually have no idea whether a problem is due to a packet is not being sent or not being acted on as expected. The yellow LED flash on event is meant to help in this situation.

Many modules have switches for configuring the module, the NodeID switches on producer modules are not used and can be omitted. The small push button is also not used and can be omitted. Any pull up resistors on those switches should probably be fitted.

Consumer modules can be taught to react to events by putting them in learn mode and creating the event.

There is a plan to convert CBUS modules to 12V dc supply instead of the current confused situation. This would mean new PCB's so other changes may also take place.

Protocol

Since this is a proof of concept implementation the number are not fixed and may change as the protocol is revised. So in general the names should be used instead of the numbers.

This protocol only uses the 29 bit CAN extended header with normal frames.

The order of packets is not fixed, packets may be re-ordered by the internal buffering in the chip and so will need to be corrected by the receiver. Usually the first packet of a group arrives first.

:XpfdddsssNxx.....;

:X	USB or serial prefix of hex digit string for extended header. (S = short type not used.)
p	1 bit (most significant bit) priority, currently always 1.
f	4 bits frame type
ddd	12 bits destination node alias or broadcast message type
sss	12 bits source node alias.
N	Normal frame type, (R = RTR type not used.)
xx..	Data in hex pairs

;	end of packet string marker

Frame Type

0x0	FT_CIM0	Top 12 bits of node ID, sent to test a new alias.
0x1	FT_CIM1	2nd top 12 bits of node ID, sent to test a new alias.
0x2	FT_CIM2	3rd top 12 bits of node ID, sent to test a new alias.
0x3	FT_CIM3	lowest 12 bits of node ID, sent to test a new alias.
0x4		
0x5		
0x6		
0x7	FT_RIM	0x7FFF, sent by any node objecting to the node alias sent in a CIMn
0x8	See below	Broadcast frame type
0x9		
0xA		
0xB		
0xC		
0xD		
0xE	See below	Destination Alias Addressed
0xF	Stream	

0x8ddd Broadcast Packets

0x8000	FT_RESET	System reset, not yet implemented.
0x8001	FT_VNSN	Verify Node Serial Numbers, sent by anything that needs to know how to translate an alias to a full node id. Reply is several DAA_NSN packets.
0x8002	FT_INIT	Normal Initialization Complete, no data. Should it send the full nodeID ?
0x8003	FT_BOOT	Similar to FT_INIT but only Boot Loader Initialization Complete.
0x8004	FT_NSN	Node serial number, replaces DAA_NSN

Accessory

0x8010	FT_EVENT	8 bytes event code in the data
0x8011	FT_RFID	5 bytes RFID tag in the data

Track commands

0x8020	FT_TOF	Track Off, broadcast from CS
0x8021	FT_TON	Track On or Normal operation, broadcast from CS
0x8022	FT_ESTOP	Track Stopped (emergency stop)

0x8023	FT_CSRESET	Command station Reset, not implemented yet.
0x8024	FT_RTOF	Request Track Off, from CAB
0x8025	FT_RTON	Request Track On or Normal operation, from CAB
0x8026	FT_RESTP	Request Emergency Stop ALL

CAB commands

0x8030	FT_RLOC	Request loco info
0x8031	FT_STMOD	Request speed step change
0x8032	FT_DSPD	Set Engine Speed/Dir
0x8033	FT_DFUN	Set engine functions
0x8034	FT_PLOC	Engine report from CS
0x8035	FT_PLOCF	Engine function report from CS
0x8036	FT_KLOC	Release loco

Consist commands, not implemented yet.

0x8037	FT_PCON	Consist Engine
0x8038	FT_KCON	Remove engine from consist

DCC programming

0x8040	FT_RDCC3	Request 3 byte DCC packet
0x8041	FT_RDCC4	Request 4 byte DCC packet
0x8042	FT_RDCC5	Request 5 byte DCC packet
0x8043	FT_RDCC6	Request 6 byte DCC packet

0xEddd Destination Alias Addressed

1st byte of data has message type.

General

0x00-0xFF	DAA_DATA	Sequence number in low 4 bits of the first byte, 7 bytes of data		
0x10	DAA_ACK	Acknowledge with status in 2nd data byte.		
		0	ACK_OK	OK
		1	ACK_CRC	CRC error, no longer used
		2	ACK_TIMEOUT	time out on data transfer, 2 seconds
		3	ACK_NODATA	The requested data does not exist
		4	ACK_NOSPACE	No space to store this data
		5	ACK_ALIASERROR	Wrong Source Alias, probably 2 writes at the same time.

Program Loader

0x20	DAA_UPGSTART	Enter loader
0x21	DAA_UPGRESET	Clear program not loaded and re-boot
0x22	DAA_UPGREAD	Read 64 bytes, reply is enough DAA_DATA packets. 24 bit address in data bytes.
0x23	DAA_UPGADDR	Write 64 bytes, followed by enough DAA_DATA packets. 24 bit address in data bytes.

Events

0x30	DAA_CEERASEH	Consumer erase events, High 7 bytes of event in data.
0x31	DAA_CEERASEL	Consumer erase events, Low byte of event. If the 8 bytes of the event are all zero then erase everything
0x32	DAA_CEREADH	Consumer read events, High 7 bytes.
0x33	DAA_CEREADL	Consumer read events, Low byte, index, data length byte.
0x34	DAA_CEWRITEH	Consumer write event, High 7 bytes
0x35	DAA_CEWRITE	Consumer write event, Low byte, data length, up to 5 data bytes
0x36	DAA_PEERASE	Producer erase event, index
0x37	DAA_PEREAD	Producer read event, index
0x38	DAA_PEWRITEH	Producer write event, High 7 bytes
0x39	DAA_PEWRITEL	Producer write event, Low byte, index

Node variables

0x40	DAA_NVREAD	Read, 1 byte index
0x41	DAA_NVWRITE	Set, 1 byte index + 1 byte data
0x42	DAA_NVREPLY	Reply to read

Misc

0x50	DAA_NSN	Node serial number, use FT_NSN in new code
0x51	DAA_DEFAULT	Reset (almost) everything to default values
0x52	DAA_REBOOT	Re-boot the module, (after node ID write)

Node ID

All node id's should be unique, so that any node can be connected to all the others without causing a problem. A 48 bit node id was chosen to make this easy to achieve. For a MERG Kit, the buyer should be a MERG member, and use a node id formed as follows. The 1st byte = 3, the 2nd byte = 2, the 3rd to 5th bytes are the membership number and the last byte can have a unique value chosen by the member.

When initially programmed the boot loader initializes the node ID. However CAN only works with 2 nodes or more connected. So a USB or RS232 interface has an initial node ID = 000000000001 and other nodes have an initial node ID = 000000000002. Thus a network of 2 nodes can be constructed and the node id's changed before adding any more nodes.

If a larger network is constructed before setting the node id's, then in theory it is possible that 2

nodes will loop forever selecting new but identical aliases. If this happens then it will be necessary to manually reboot each node (by shorting the MCLR pin to ground) to get unique node aliases.

Node Alias

To get an initial node alias from the 48 bit node ID the 4 x 12 bit parts are xored together.

The 4 parts are transmitted using the FT_CIMn packets together with the alias. A 1 second time out allows any node with the same alias to send a FT_RIM to object and cause the node to generate another alias.

To get the next alias the node ID is used as the initial seed for a 48 bit random number generator, the 4 x 12 bit parts are then xored together as before.

If 2 or more nodes have the same alias after the initial set up, then all nodes that detects the error should try to obtain a new unique alias by the method above.

The proposed 48-bit Pseudo-Random Number Generator is from “A 48-Bit Pseudo-Random Number Generator”, Heidi G. Kuehn, Communications of the ACM, Volume 4, Issue 8 (August 1961), pages 350-352.

$$x_{i+1} = (2^9+1)\, x_i + c$$

where c = 29,741,096,258,473 or 0x1B0CA37A4BA9.The paper actually describes a Pseudo-Random Number Generator that uses signed arithmetic, but for our purposes the 48th “sign” bit isn't important.

Note that, unlike some other Pseudo-Random Number Generators, this one can generate a zero result, and can accept a zero seed.

Memory Layout

The memory layout here is fixed to make it possible to write the initial PC interface tool before the XML file layout is fixed.

Some information is duplicated in the XML file, which wastes space, so this will need to be re-organized when the XML side is working on the PC.

The location of the remaining information in memory will also need to be specified in the XML file.

Boot memory layout

The boot loader uses the bottom 4k of program memory.

0x000000	Reset, test the program loaded flag and execute the program or the loader
0x000008	High priority interrupt, always redirected to the program.
0x000018	Low priority interrupt, always redirected to the program.
0x000020	7 byte version info for boot loader
0x000030	jump to loader, called from the program.

0x000040	6 byte Node ID (low first), 6 byte random seed, 2 bytes alias, 50 bytes empty
0x000080	64 byte block, boot loader text string
0x0000C0	64 byte block, user identification string

Program memory layout

Program start address = 0x1000.

0x001000	redirected reset
0x001008	redirected high priority interrupt
0x001018	redirected low priority interrupt
0x001020	7 byte version info for program
0x001027	valid program loaded flag, 0xFF=no program, 0x00=valid program
0x001028	4 bytes start address of XML data description file
0x00102C	2 bytes length of XML file.
0x00102E	18 spare bytes
0x001040	64 byte block with program identification string

Note. In C18, 16 and 32 bit (2 and 4 byte) data are stored low byte first. Addresses are 24 bits. OpenLCB data is stored most significant byte first.

Program loading

There are 3 versions of the loader, one for loading from CAN interface, one for USB interface and one for loading from RS232 interface. Most modules will use the CAN version.

The PC side will only send data above the 4k boot loader area. In order to change the boot loader a program has to be loaded which will check the new code is complete and then copies the new boot loader over the old one. Finally setting the program loaded flag so it does not get executed again after a reset.

A new program is sent 64 bytes at a time. The first write should be the first 64 bytes of the new program with the valid program loaded flag set to 0xFF.

To enter the boot loader a DAA_UPGSTART is sent, after the program halts interrupts and jumps to the loader a DAA_ACK with ACK_OK is sent back to the PC.

Each 64 byte block starts with DAA_UPGADDR and is followed by 10 DAA_DATA packets. After writing the block the loader will send DAA_ACK with ACK_OK to signal that the next block can be sent. The blocks can be received in any order so they should be stored on a buffer until all the packets are received.

After all the program blocks have been successfully sent, a DAA_UPGRESET is sent from the PC to clear the valid program loaded flag and restart the program.

Event teaching from PC

All modules should respond to these packets even if they have no event data.

The packets for event teaching from the PC all use Destination Alias Addressed packets, with the first byte of the data containing a type code for the rest of the data bytes.

For general I/O modules there may be both producer and consumer events to handle, so two sets of commands are used.

Consumer events are looked up using the 8 byte event number, producer events are looked up by an index formed from the input.

Some limits for event data:
Always 8 bytes for the event number.
Maximum of 65000 events per module, but most modules use much less than this.
Maximum of 64 bytes for the event data so it can use the same buffer as the loader, most modules will only use 1 or 2 bytes.

Event Packet Types

DAA_CEERASEH	Erase consumer events, the high 7 bytes of the event code.
DAA_CEERASEL	Erase consumer events, the low byte of the event code.
DAA_CEREADH	Read consumer events, the high 7 bytes of the event code.
DAA_CEREADL	Read consumer events, the low byte, event index and data length.
DAA_CEWRITEH	Write consumer event, the high 7 bytes of the event code.
DAA_CEWRITEL	Write consumer event, Low byte and the index.
DAA_PEERASE	Erase producer event given by the index.
DAA_PEREAD	Read producer events, event index.
DAA_PEWRITEH	Write producer event, the high 7 bytes of the event code.
DAA_PEWRITEL	Write producer event, Low 1 bytes and the index.
DAA_DEFAULT	Initialize events and node variables to default values.
DAA_DATA	Up to 7 bytes of data per packet, block of up to 64 bytes
DAA_ACK	Ack with status

Set default

DAA_DEFAULT: no data.

Re-initialize the producer events and node variables to the default values.
Default producer events are the node ID followed by the switch or input number and the on or off state.
The reply should be DAA_ACK = OK, even if there is nothing to do.

Consumer Erase Event data

DAA_CEERASEH: 7 high bytes of event
DAA_CEERASEL: last byte of event

For a consumer, the node should erase all events that match the event number. If the event number is zero then it erases all events. The reply should be DAA_ACK = OK, even if there is nothing to do.

Producer Erase Event data

DAA_PEERASE: 2 byte index

For a producer, the event number is not used, the event at the index is erased. The reply is DAA_ACK = OK, or DAA_ACK = NOSPACE if the index is out of range.

Consumer Event read

DAA_CEREADH: 7 high bytes of event
DAA_CEREADL: last byte of event, 2 byte index,

If the event number is zero then read all events. The first read should have index = 0. Each read should increase the index until the error return NODATA is received. This returns one event, in DAA_CEWRITEH, DAA_CEWRITEL and a number of DAA_DATA packets. The index is used to decide which event to read if there is more than one set of event data with a matching event number. The module should send an error reply DAA_ACK = NODATA if there is no data corresponding to this index or DAA_ACK = TIMEOUT if a packet gets lost .

Producer Event read

DAA_PEREAD: 2 byte index

A producer will ignore the event number and just use the index. This returns one event, in DAA_PEWRITEH, DAA_PEWRITEL. The module should send an error reply DAA_ACK = NODATA if there is no data corresponding to this index.

Consumer Event write

DAA_EVWRITEH: 7 high bytes of event
DAA_EVWRITEL: last byte of event, 1 byte data length, 5 bytes data

The event number is never zero. If the number of bytes of data that will follow is 5 or less, then the data is in the packet. If the number of bytes is more than 5 then all the data is in data packets. The module should send DAA_ACK = OK after storing the data, or DAA_ACK = TIMEOUT if a packet gets lost, or DAA_ACK = NOSPACE if there is no space to store it.

Producer Event write

DAA_EVWRITEH: 7 high bytes of event
DAA_EVWRITEL: last byte of event, 2 bytes index

The event number is never zero. The index is use to decide which event is replaced. The module should send DAA_ACK = OK after storing the data, or DAA_ACK = TIMEOUT if a packet gets lost, or DAA_ACK = NOSPACE if there is no space to store it.

Node Variables

There can be up to 256 node variables of 1 byte each. These are used for configurable settings on the modules. E.G. input de-bounce time, servo end point.

There are 3 packet types for these:
DAA_NVREAD Read Node Variable
DAA_NVWRITE Set a Node Variable
DAA_NVREPLY Node Variable Value

Read Node Variable

DAA_NVREAD: 1 byte index

Read an node variable, one of up to 256 single bytes. The reply is either DAA_NVREPLY, or an DAA_ACK acknowledge error message ACK_NODATA if the variable is not used.

Set a Node Variable

DAA_NVWRITE: 1 byte index, 1 byte data

The reply will be an DAA_ACK acknowledge ACK_OK, or ACK_NOSPACE if the variable is not used.

Node Variable Value

DAA_NVREPLY: 1 byte index, 1 byte data

The value of an node Variable , a reply to a DAA_NVREAD.

Track Control

These commands control the supply to the track.

FT_TOF	Track Off, broadcast from CS
FT_TON	Track On or Normal operation, broadcast from CS
FT_ESTOP	Track Stopped (emergency stop)
FT_CSRESET	Command station Reset, not implemented yet.
FT_RTOF	Request Track Off, from CAB
FT_RTON	Request Track On or Normal operation, from CAB
FT_RESTP	Request Emergency Stop ALL

All these commands have no data.

Train Control

These commands are used to control DCC trains or locomotives.

FT_RLOC	Request loco info
FT_STMOD	Request speed step change
FT_DSPD	Set Engine Speed/Dir
FT_DFUN	Set engine functions
FT_PLOC	Engine report from CS
FT_PLOCF	Engine function report from CS
FT_KLOC	Release loco
FT_PCON	Consist Engine
FT_KCON	Remove engine from consist

Request loco info

FT_RLOC <DCC address high byte> <DCC address low byte>

7 bit addresses have (AddrH=0). 14 bit addresses have bits 7,8 of AddrH set to 0. The command station responds with (PLOC and PLOCF). This command is typically sent by a cab to the command station following a change of the controlled decoder address.

Request speed step change

FT_STMOD <Session> <Speed steps>

Speed steps:
Bits 0 – 1: speed mode
00: 128 speed steps
01: 14 speed steps
10: 28 speed steps with interleave steps ?
11: 28 speed steps
Bits 2 - 7: spare

Sent from a cab to the command station to change the number of speed steps used, to that supported by the decoder. The command station should remember the setting and send it in PLOC replies for this decoder.

Set Engine Speed/Dir

FT_DSPD <Session> <Speed/Dir>

Sent by a device to request an engine speed/dir change. The speed is in the DCC format for the number of speed steps. For 128 steps, bit 7 is the direction, 1 for forward. The other 7 bits are the speed, 0 is stop, 1 is emergency stop. For 14 or 28 steps, bit 6 is 1, bit 5 is the direction, bit 4 is the lsb of the 28 step speed, the other 4 bits are the speed, with 0 is stop and 1 is emergency stop.

Set engine functions

FT_DFUN <session> <function group><functions>

<function group> is 1 for FN0-FN5, 2 is FN5-FN8, 3 is FN9-FN12, 4 is FN13-FN20 and 5 is FN21-FN28.
<functions> is 5, 4 or 8 bits, one per function in DCC order. eg. FN0 is in bit 4.
Sent by a device to request an engine Fn state change.

Engine report from CS

FT_PLOC <session><DCC address high><DCC address low><flags><speed>

<session> Session number for engine assigned by the command station. This session number is used in all references to the engine until it is released.
<DCC address high> is the MS byte of the DCC address. For short addresses it is set to 0.
<DCC address low> is the LS byte of the DCC address.
<flags> number of speed steps and other flags.
bits 0-1: Speed Mode
00: 128 speed steps
01: 14 speed steps
10: 28 speed steps with interleave steps
11: 28 speed steps
Bit 2: spare
Bit 3: spare
Bit 4: Consisted
Bit 5: Consist Master
Bit 6: spare
Bit 7: In use by another cab.
<Speed> is the Speed/Direction value. Bit 7 is the direction bit and bits 0-6 are the speed value. If

the engine is consisted, this is the consist address.

A report of an engine entry sent by the command station. Usually sent in response to FT_RLOC as an acknowledgement of acquiring an engine requested by a cab. Due to the number of functions, these states are now in the PLOCF packet, which may be before or after the PLOC packet.

A report of an engine entry sent by the command station. Usually sent in response to QLOC or as an acknowledgement of acquiring an engine requested by a cab (RLOC).

Engine function report from CS

FT_PLOCF <Session><AddrH><AddrL><FN0><FN5><FN13><FN21>

<Session> Session for engine assigned by the command station. This session number is used in all references to the engine until it is released.
<AddrH> is the MS byte of the DCC address. For short addresses it is set to 0.
<AddrL> is the LS byte of the DCC address.
<FN0> is the 5 functions FN0 to FN4 in DCC bit order.
<FN5> is the 8 functions FN5 to FN12.
<FN13> is the 8 functions FN13 to FN20.
<FN21> is the 8 functions FN21 to FN28.

The PLOCF may appear before or after the PLOC, and is usually a reply to a QLOC or RLOC request.

Release loco

FT_KLOC <Session number>

Releases control of a loco or train. The entry remains in the command station until needed the command station list is full and the entry needs to be reused.

Consist Engine

FT_PCON <Session><Consist address>

Adds a decoder to a consist. If consisting is successful, an ACK is sent. If engine is consisted already, an error is reported: engine in use.

Remove engine from consist

FT_KCON <Session><Consist address>

Removes a decoder from a consist. If de-consisting is successful an ACK is sent. If engine is not consisted, an error is reported: Engine not found.

Raw DCC Commands

These are CBUS packets which contain a preformatted DCC command, typically used for accessory operation or programming on main. The last byte of each DCC packet is the error check, so it's not actually necessary as the command station can complete it.

FT_RDCC3 Request 3 byte DCC packet
FT_RDCC4 Request 4 byte DCC packet
FT_RDCC5 Request 5 byte DCC packet

FT_RDCC6 Request 6 byte DCC packet

The first data byte is the <REP> byte, it is a 6 bit count of the number of times the command is sent to the track. Bit 7 is used for an operational mode write when the command has to be sent to the same address twice. Bit 6 is used allow a deactivate command to follow an activate command. The following data bytes are the preformatted DCC command.

Embedded XML Files

More work need to be done on this but this is a start.

XML files are read from the modules so that a program running on a connected PC can display easily understandable text for the configuration.

XML files may be in plain text or in EXI (Efficient XML Interchange) format. EXI uses the first 2 bits of the file to determine if the XML file is compressed. Most files should be very small so compression may not be necessary.

<name> should be a necessary tag. Blank for unused bits ?

<default> is optional, with 0 being implied if its missing.

<bits> is optional, with 8 being implied if its missing.

Solenoid point motor module:

```
<XML>
  <NodeString>OpenLCB Output driver for CANACC4</NodeString>
  <EventData>
    <name>Event Number</name><bits>48</bits>
    <name></name><bits>5</bits>
    <name>Output</name><bits>3</bits>
  </EventData>
</XML>
```

8 R/C servo module:

```
<XML>
  <NodeString>OpenLCB Servo Output Driver</NodeString>
  <EventData>
    <name>Event Number</name><bits>48</bits>
    <name></name><bits>4</bits>
    <name>Polarity</name><bits>1</bits>
    <name>Output</name><bits>3</bits>
  </EventData>
  <NodeVariable>
    <name>Servo#0 off position</name><default>125</default>
    <name>Servo#1 off position</name><default>125</default>
    <name>Servo#2 off position</name><default>125</default>
    <name>Servo#3 off position</name><default>125</default>
    <name>Servo#4 off position</name><default>125</default>
    <name>Servo#5 off position</name><default>125</default>
    <name>Servo#6 off position</name><default>125</default>
    <name>Servo#7 off position</name><default>125</default>
    <name>Servo#0 on position</name><default>125</default>
    <name>Servo#1 on position</name><default>125</default>
    <name>Servo#2 on position</name><default>125</default>
    <name>Servo#3 on position</name><default>125</default>
    <name>Servo#4 on position</name><default>125</default>
    <name>Servo#5 on position</name><default>125</default>
    <name>Servo#6 on position</name><default>125</default>
```

```
      <name>Servo#7 on position</name><default>125</default>
    </NodeVariable>
  </XML>
```

8 input module

```
  <XML>
    <NodeString>OpenLCB Input Driver for CANACE8C</NodeString>
    <NodeVariable>
      <name>Debounce</name><default>0</default>
    </NodeVariable>
 </XML>
```

XSD

A start of on the definition of the XML file format.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema elementFormDefault="qualified"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
      <xsd:element name="XML">
            <xsd:complexType mixed="true">
                  <xsd:sequence>
                        <xsd:element ref="NodeString" minOccurs="0"
maxOccurs="1"/>
                        <xsd:element ref="NodeInfo" minOccurs="0"
maxOccurs="1"/>
                        <xsd:element ref="Events" minOccurs="0" maxOccurs="1"/>
                        <xsd:element ref="NodeVariables" minOccurs="0"
maxOccurs="1"/>
                        <xsd:element ref="Data" minOccurs="0" maxOccurs="1"/>
                  </xsd:sequence>
            </xsd:complexType>
      </xsd:element>
      <xsd:element name="NodeString">
            <xsd:simpleType>
                  <xsd:restriction base="xsd:string">
                        <xsd:maxLength value="64"/>
                  </xsd:restriction>
            </xsd:simpleType>
      </xsd:element>
      <xsd:element name="NodeInfo">
            <xsd:simpleType>
                  <xsd:restriction base="xsd:hexBinary">
                        <xsd:length value="7"/>
                  </xsd:restriction>
            </xsd:simpleType>
      </xsd:element>
      <xsd:element name="Events">
            <xsd:complexType mixed="true">
                  <xsd:sequence minOccurs="0" maxOccurs="unbounded">
                        <xsd:element ref="Name"/>
                        <xsd:element ref="Bits" minOccurs="0" maxOccurs="1"/>
                  </xsd:sequence>
            </xsd:complexType>
      </xsd:element>
      <xsd:element name="NodeVariables">
            <xsd:complexType mixed="true">
                  <xsd:sequence minOccurs="0" maxOccurs="256">
                        <xsd:element ref="Name"/>
                        <xsd:element ref="Bits" minOccurs="0" maxOccurs="1"/>
                        <xsd:element ref="Default" minOccurs="0" maxOccurs="1"/>
                  </xsd:sequence>
```

```
            </xsd:complexType>
      </xsd:element>
      <xsd:element name="Data">
            <xsd:complexType mixed="true">
                  <xsd:sequence minOccurs="0" maxOccurs="unbounded">
                        <xsd:element ref="Name"/>
                        <xsd:element ref="StartAddress"/>
                        <xsd:element ref="Length"/>
                  </xsd:sequence>
            </xsd:complexType>
      </xsd:element>
      <xsd:element name="Name" type="xsd:string"/>
      <xsd:element name="Bits" type="xsd:decimal"/>
      <xsd:element name="Default" type="xsd:string"/>
      <xsd:element name="StartAddress">
            <xsd:simpleType>
                  <xsd:restriction base="xsd:string">
                        <xsd:pattern value="0x[0-9,A-F]{6}"/>
                  </xsd:restriction>
            </xsd:simpleType>
      </xsd:element>
      <xsd:element name="Length" type="xsd:decimal"/>
</xsd:schema>
```