M PROTOCOL AGENT

NAMING CONVENTIONS

m_	precedes all global agent variables, structures and functions
bm_	precedes all global variables, structures and functions used only by the Bus Master
M_	precedes all agent constants
BM_	precedes all constants used only by Bus Master
_s	attached to the end of variable, signifies the variable is static

All variables, structures and functions are lower case. All constants and macros are upper case. Local variables do not follow naming conventions.

HOW THE AGENT WORKS

The M protocol agent is called by invoking the function *m_agent()*. This function is a state machine that handles interaction between the M protocol port and the application. The state machine is used as a method of keeping track of what needs to be done next without waiting for something to happen. If the agent has nothing to do, it will return control to the application as soon as possible. Calling the *m_agent* function allows the agent processing time and must be done in a timely manner.

The majority of the agent's time is spent parsing data from the M protocol port of the device. The application is expected to place data received from the M protocol port into a ring buffer via a serial interrupt. The ring buffer is then analyzed by the agent through a call to *m_look_for_message( ).*

If the agent is not in control of the bus, the agent waits for the bus token **-** it's turn to control the bus. Once the agent has determined it has the token, it will scan the databases for changes. If it finds any elements that need to be read or written, it will form a message to one device, otherwise it will simply pass the token in a "null" message. If the message is a write, it will pass the token with the message, otherwise if it is a read or a write with verify on the local bus, the agent will wait for a reply, then pass the token upon receipt of any message without the grant bit set. If the grant bit is set it is considered an error where the Bus Master was forced to take over the bus.

Any time a message is sent external to the local bus the token is passed since there is no guarantee of a timely response when dealing with anything outside of the local bus. If there are no requests by the application, the token is simply passed on to the next device in a null message.

The agent handles all communication with the M protocol port and interfaces with an application using the element states associated with each element of a database. Element states enable the application to tell the agent which elements of a database need to be read or written. In turn, the agent tells the application which elements have been written, which are waiting for a response or verification, and which have failed to be read or written.

USING THE M PROTOCOL AGENT IN YOUR APPLICATION

The M protocol agent must be set up for each application. It must be told which databases are being used and how to access these databases. While the structure of each database and the access of each element within the structure of the database are pre-defined in the agent, it is still necessary to provide the agent with pointers to the databases in memory. The agent must also be told which databases it has and how many of each type it must deal with. Each database is placed in an array containing pointers to instances of the database with an associated four byte network address. This allows the agent to relate a database with a particular device on the network. Databases belonging to the agent are included in this structure and have an address matching the agent's address.

It is the responsibility of the application to set up the database pointers and addresses in the structures mentioned above. For this reason, the application may refer to the databases explicitly by name. The agent will not have this luxury, as the databases are defined on an application by application basis (except for database 0).

The following is a list of items that must be done for each application:
In your main code:

1. Call *m_agent( )* from the main loop.
2. Initialize *agent_address.device* (perhaps from a dip switch). The upper three address bytes are defaulted to 0xFF (invalid). The Bus Master will provide the upper 3 bytes of address when it turns the device on.
3. CHECK ALL DATABASES FOR CHANGE OF STATE FROM THE AGENT and clear the element state when done or you will not receive any new data in that element.

In agent.h:

1. Define M_VERSION and M_REVISION to the version and revision levels of your application. These values are placed in database 0 and may be accessed by the outside world along with the finished good product number m_fingood[ ].
2. Make sure the definitions for M_BYTE, M_WORD and M_DWORD are accurate for your processor, they should be 8, 16 and 32 bits respectively.
3. Define M_BIG_ENDIAN if the architecture is big endian, otherwise do not define this constant.
4. For each database there is a conditional compiler option that must be set if the database type is to be used. The options are M_USE_DATABASEx where x is the number of the database. For instance all applications must have at least one instance of the agent database, database 0, therefore every application has M_USE_DATABASE0 defined as default.
5. The agent needs to be told the M_TOTAL_NUMBER_OF DATABASES which is used for indexing through the database array.
6. Next, for each instance of a database an external declaration for the database should be present for global access to the database. This declaration allows all files with agent.h included to access the database. Here's an example:
   ```
   extern  struct  m_db0_type  m_agent_db;
   ```
7. Adjust M_MAX_NUMBER_DATA_FIELDS. This is the size of the buffer containing header information for each data field (contains database, element and length). This constant limits the number of separate fields allowed by the agent in the outgoing data section.
8. Adjust M_MAX_DATA_COUNT. This is the size of the buffer for the data section without the header information of each field. This is the raw data being sent. The header information contains pointers into this structure to associate the raw data with the header.
9. M_MAX_TX_BUF is based upon the last two values since only one message may be prepared to be sent at a time. Make sure this amount of RAM is acceptable. Do the same for M_MAX_RX_BUF.

In agentdat.c:

1. Each instance of each database must be declared here. The following must be declared for the example above:
   struct m_db0_type m_agent_db;
2. Update *m_copyright[ ]* and *m_fingood[ ]*. These are copied to database 0 and are accessible along with the application's version and revision. It is suggested that these arrays be made ROM instead of RAM to save space. For purposes of being generic to different compilers, these were defined as data.

In agentdb.c:

1. There is an array of structures named *m_db_array[]*. Each element of the array contains a network address, a database type, a number of elements in the database, and a pointer to a database of the database type. The network address is used by the agent to determine which

database to use when speaking with a particular device on the network. The network address and pointer to the database must be provided by the application. While it is possible to change these during execution of the program, it is important to initialize this structure. A good location for this is in *init_database* ( ). The following example is for database 0 and similar code should be in all applications:

```
m_db_array[0].address.network = agent_network_address;
m_db_array[0].address.hub = agent_hub_address;
m_db_array[0].address.bus = agent_bus_address;
m_db_array[0].address.device = agent_local_address;
m_db_array[0].db_type = M_DATABASE0;
m_db_array[0].number_elements = M_NUMBER_ELEMENTS_DB0;
m_db_array[0].db = &m_agent_db;    /* point to first instance of database 0 */
```

where the addresses are provided by the application and is the address of the application in the case of the agent database. Note that the index in the above example happens to be 0, this is not relevant to the type of the database. It is suggested that databases that will be used heavily be placed nearest to the beginning of the database array.

In comm.c:

1. All routines in this file are left as examples and should be changed for your hardware platform.

In agent.c

1. The macros INTS_ON and INTS_OFF are used to turn interrupts on and off. These should be filled in with appropriate statements for your hardware configuration.

There are several functions that must be changed for each hardware configuration. These routines deal with the uart and hardware interrupts. The following is a list of the routines that need to be looked at closely and changed for hardware differences:

Function Name	File	Reason
tx_m_message( )	agent.c	enables transmit in uart and transmit interrupt, disables receive interrupt if Bus Master
m_put_data( )	agent.c	INTS_ON and INTS_OFF are hardware dependent
m_get_data( )	agent.c	INTS_ON and INTS_OFF
enable_m_rx( )	busmast.c	deals directly with bits in uart (Bus Master only)
disable_m_rx( )	busmast.c	deals directly with bits in uart (Bus Master only)

ADDRESSING

Database 0, which is a required database for each application, contains a structure of type m_network_address which contains the four bytes of network address. The upper three bytes of the network address are provided by the Bus Master and are write only. The upper three bytes are write only to prevent a device from reading these bytes into their own database 0 and destroying their own addressing. The lower byte or device address is provided by the application, perhaps by a dip switch, and is read only. The information in Database 0 is used for communication. The working network address is called *m_agent_address* and is updated from Database 0 when Database 0 is changed through the network.

DATABASE STRUCTURE

All databases have the same initial structure in the agent, meaning that the first few entries are common across databases. The first byte of each database is a *cos* or change of state byte. One bit within this byte is used to tell the agent that an element state in the database has changed and the agent needs to scan through the database. Another bit in the *cos* byte is used to tell the application that the agent has changed an element state and that the application needs to scan through the database. The remaining six bits of the *cos* byte are undefined.

After the *cos* byte is an array of varying size called *m_element_states*. There is one byte in the array for every element in the database containing the current state of the element. Using this commonality between databases, the agent may look through the element states of databases in the *m_db_array[]* structure using a cast to database 0 to the pointer in the database array. Database 0 is known to exist in all applications so the structure is always known. Because of this method of looking at element states, the database array must also contain the number of elements the database being pointed to contains and the type of the database.

Besides being in every agent, database 0 is special in other ways. There is always an instance of database 0 which has been named *m_agent_db* in the agent code. All writes and reads made to *m_agent_db* are completely handled by the agent. It is okay to have another instance of database 0, but this other instance will not be managed by the agent in the same way, but rather it will be treated as any other remote database. The *m_agent_db* is used to provide error codes, in essence giving the protocol more commands (acknowledge and negative acknowledge in *error_code* with locations of where errors occurred in *error_string*). The address portion of database 0 is used for communication purposes and must be kept separately from the database as well to allow the outside world to change the upper three bytes of the address.

The Bus Master also handles a special database 1 called *bus_master_db* in much the same way an agent handles *m_agent_db*. The Bus Master is a specialized agent, since the Bus Master code becomes integrated with the agent code rather than simply calling agent routines, but the integration is minimized and clearly separated with *if defines*. Database 1 may also have other remote instances, just like database 0, but neither will they be managed by the Bus Master.

It should be pointed out that all database elements that are not M_READ_ONLY can be written over by another device on the network, unless the element state of the element is kept in a state to prevent this from happening, which means that there is no history for the data. If a history of the data is required, it is up to the application to provide.

Note: The agent does not allow multiple instances of local databases. In other words, if a unit has a local database 2, it cannot have another local database 2 because the address for both databases is the same, making the target database for a message either or both. When a message comes in and a database of a particular type is found in the *m_db_array[ ]* with the correct address, this is the only database that will be accessed for that message (this could be changed, but so far we have not thought of a reason to do so or how the conflict should be handled). The same would hold true for remote databases of the same type with the same address, but since there are no local multiples with the same address, this is not a problem. For now, the rule is the first database found that matches is used.

COORDINATION BETWEEN AGENT AND APPLICATION

Every element in a database has an element state associated with it. Each database contains an array named *m_element_states[ ]*. The state of an element may be found by indexing into this array by the number of the element. If any of the element states in a database change, the *cos* byte for the database is set to indicate a change in the database. There are two *cos* bits in each database, one to notify the agent that there is something in the database for the agent's attention called M_COS_FOR_AGENT and one to notify the application that there is something in the database for the application's attention called M_COS_FOR_APP.

The following element states are defined:

NAME OF STATE	IDENTIFIER	DESCRIPTION
M_NORMAL_STATE	0	Normal - no action required.
M_AGENT_CHANGED	1	Agent wrote into the element.
M_APP_WRITE_REQ	2	Application wishes agent to send this element.
M_APP_WRITE_REQ_W_ACK	3	Application wishes agent to send this element and requires and acknowledgement for receiver.
M_PENDING_WRITE	4	Agent has written to remote database and is waiting for an acknowledge.
M_WRITE_DONE	5	Write to remote element is completed.
M_WRITE_FAILED	6	Write to this remote element failed.
M_TEMP_WRITE_FAILED	7	Write to this remote element failed temporarily (element state would not allow a write).
M_APP_READ_REQ	8	Application wishes agent to request this element from remote database.
M_PENDING_READ	9	Agent has asked remote for information and is waiting for a response.
M_READ_DONE	10	Read from remote element is completed.
M_READ_FAILED	11	Read from this remote element failed.
M_TEMP_READ_FAILED	12	Read from this remote element failed temporarily (element state would not allow a read).
M_APP_LOCK	13	Application has locked this element from being read or written.

If the application wishes to write a particular element, the application must change the associated element state to M_APP_WRITE_REQ (if the application wants to verify the write it may make it M_APP_WRITE_REQ_W_ACK) and the agent must be notified that the database has something in it for its attention by the application setting M_COS_FOR_AGENT in the associated database. When the agent sees the M_COS_FOR_AGENT, it will scan through the database and find the M_APP_WRITE_REQ and form a message to the remote device. The element state will then be changed to M_WRITE_DONE and the M_COS_FOR_AGENT will be cleared. The agent will then set M_COS_FOR_APP so that the application will be notified that the write is done.

ELEMENT STATES (an example)

There are two examples provided with the released source code. KEYBOARD translates D protocol input from serial port 0 to M protocol on serial port 1. PTZ takes M protocol from serial port 1 and translates it to D protocol on serial port 0. Simply define KEYBOARD or PTZ in agent.h to create the application. These examples run as is on the CM9760-ALM board which contains 2 serial ports. Serial port 0 may be configured for EIA232 or EIA422. Serial port 1 may be configured for EIA422 or EIA485 (and may also be used as single wire 485 which is perfect for M protocol).

Either one of the applications, KEYBOARD or PTZ, may also be a Bus Master by simply defining M_BUS_MASTER in agent.h along with KEYBOARD or PTZ, however serial port 0 will be used for the applications and will not be available for diagnostics as when M_BUS_MASTER is defined by itself. The diagnostics are text driven commands that allow read and write commands to be sent, shows which devices are active or failed, and displays when a read or write comes across the bus.

The dip switches on the CM9760-ALM board are used as follows:

Dip Switch 1:

Baud0	Baud0	Baud1	Baud1	Baud1	db addr	db addr	db addr	EIA232	Diag
00 = 2400, 01 = 4800, 10 = 9600, 11 = 19200		000 = 2400, 001 = 4800, 010 = 9600, 011 = 19200, 100 = 38400, 101 = 57600, 110 = 115200, 111 = 230400			This is the address of all remote databases. PTZ_ADDR in code. This is how the KEYBOARD knows which address the PTZ is at, therefore the PTZ address must be in range 0 - 7.			If ON port is EIA232, else EIA422.	If ON diag mode is on in Bus Master.

Baud1 is for port 1, which is the M protocol port. Port 0 is used for diagnostics or serial communications in D protocol.

For Dip Switch 2, levers 1-8 are the device's local address, levers 9-10 are unused. For the PTZ, this local address is also the address of it's local pt_db and lens_db unless the PTZ is a Bus Master, which means the local databases will have address 0.

KEYBOARD

In the example code, serial0.c contains a function called D_to_M_agent( ) which is called if KEYBOARD is defined. This function translates serial input in D protocol to M protocol. This is a good example of one way communication (only writes occur from the KEYBOARD to the PTZ). Each time a command is interpreted from D protocol an element of the database is involved. The D_to_M_agent( ) function changes the database element itself, then changes the element state and finally changes the cos byte for the database. Here's an example where a preset set command from the D protocol is being translated to M protocol:

```
case CMD_PRS_SET:
        pt_db.preset_save = cmd_data_s[3];          /* change db value */
        /* tell agent to send it */
        pt_db.element_state[M_EL_PRESET_SAVE] = M_APP_WRITE_REQ;
        pt_db.cos |= M_COS_FOR_AGENT;               /* tell agent to look at db */
        break;
```

Note that the function does not pay any attention to what the database values currently are or what the element state is before the write occurs - this is okay since this application does not care. Your application may need to be more careful. You may wish to verify that the element state is not M_AGENT_CHANGED before writing to it, otherwise you will lose the changes made by the agent to the element that you may have requested. How you handle the element states and values within the database will depend upon your application and is left up to you.

PTZ

The other side of this translator is called PTZ, which translates M protocol to D protocol. Continuing with the above example where a preset is set ; once the agent in the PTZ application receives a write to its corresponding *pt_db* through the M protocol, the agent will first check the element state of the preset set element of *pt_db*. If the element is in a state where the agent may write to the element, then the agent performs the write to the database, changes the element state appropriately, and marks the database so that the application knows an element has been changed by the agent. This takes place in the function write_data_parser( ).

Since PTZ has been defined, the function M_to_D( ), found in serial0.c, will be called. This function looks for the changes in the cos byte of each database to indicate changes for the application. If any database needs the attention of the application then this function will scan the element state of each element. The following code detects a change to the preset set element of the *pt_db* and forms the D protocol command to send out:

```
if (pt_db.cos & M_COS_FOR_APP)
{
        /* check each element of this instance */
        for (elmnt = 0; elmnt < M_NUMBER_ELEMENTS_DB4; elmnt++)
        {
                /* if the agent has changed this element */
                if (pt_db.element_state[elmnt] == M_AGENT_CHANGED)
                {
                        switch (elmnt)
                        {
                                ….
                                case M_EL_PRESET_SAVE:
                                        cmd1 = 0x00;
                                        cmd2 = CMD_PRS_SET;
                                        cmd3 = 0x00;
                                        cmd4 = pt_db.preset_save;
                                        something_changed |= 0x02;
                                        break;
                        }
                }
        }
        pt_db.cos &= ~COS_FOR_APP;
}
```

HANDLING ERRORS IN THE AGENT

When receiving data, the agent may write into a database element as long as the element state is M_NORMAL_STATE, M_PENDING_WRITE, or M_PENDING_READ, all other states of the element will cause an error code to be sent back to the sender of the message. A read into a database is allowed as long as the element state is not M_APP_LOCK, which will also cause an error code to be sent back. Note that in truth, error messages are only sent back if the Read bit in the command byte of the original message is set, signifying that a response is expected. If a write command is rejected and the read bit is not set, then the message simply disappears.

The error message is handled by writing to Database 0 of the sender. Database 0, which is a required database for each device, contains a field called *error_code*. The *error_code* field is used to receive acknowledgements and negative acknowledgements. For a list of error codes see the Millennium System Protocol Database manual.

When the application marks an element to be read, it changes the element state to be M_APP_READ_REQ. The agent scans the database, finds the element state requiring it’s attention, forms a message, and changes the element state to M_PENDING_READ. When the response to the read request is written back, M_PENDING_READ transitions to M_READ_DONE. Each time the agent changes the element state, the application is notified of the change through the change of state bit in the database. Likewise, during a write operation, if the error code is an acknowledgement (80 00), then all states in the M_PENDING_WRITE state transition to M_WRITE_DONE.

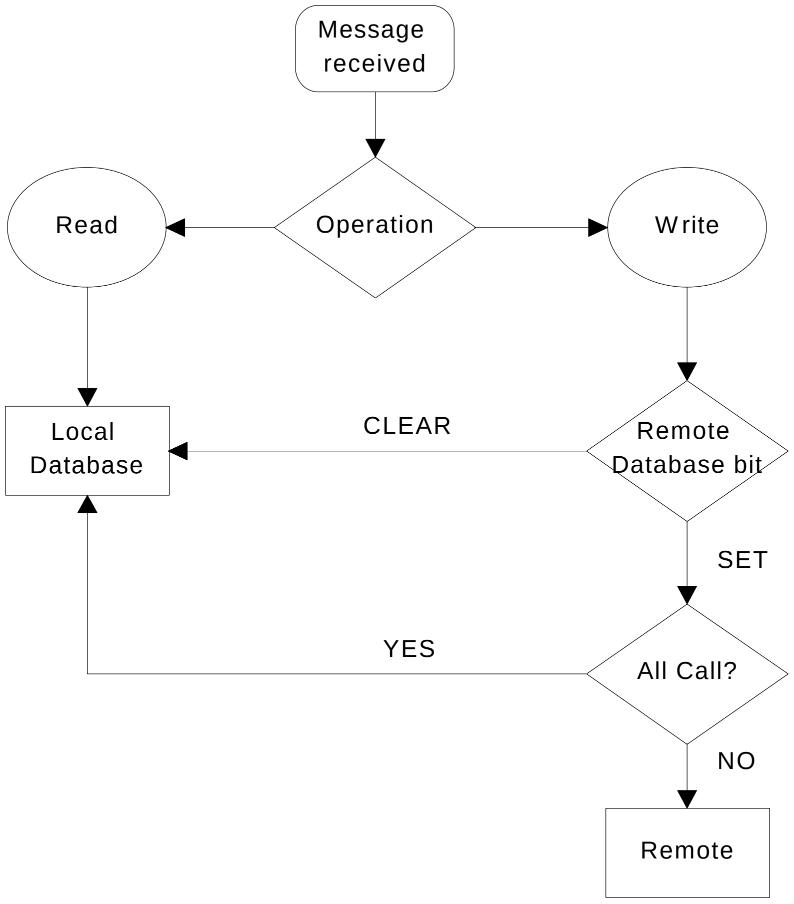
However, if the agent has elements in PENDING states and receives an error code, the agent must decide what to do with the elements now that an error has occurred. When an error is received, the highest bit of the error code is set for an error during a write operation, and clear for an error that occurred during a read operation. This tells the agent which elements to look at; elements in the M_PENDING_WRITE or M_PENDING_READ state. The agent then searches through databases associated with the source address for PENDING elements of the correct type. When a PENDING element is found, the table below displays the actions of the agent. The error string element provides additional information in the form of where the error took place (database number and element number). The information in error string is only expected to be sent for certain errors, for example, a general error does not provide this information and may not be able to since the message may have not been formed correctly.

ERROR CODE	OPERATION	ELEMENT STATE	NEW ELEMENT STATE
M_NO_ERROR	Read	M_PENDING_ READ	M_READ_DONE
M_NO_ERROR	Write	M_PENDING_ WRITE	M_WRITE_DONE
M_GENERAL_ERROR	Read	M_PENDING_ READ	M_READ_FAILED
M_GENERAL_ERROR	Write	M_PENDING_ WRITE	M_WRITE_FAILED
M_WRONG_DEVICE_TYPE	Read or Write	M_PENDING_ READ	M_READ_FAILED
M_WRONG_DEVICE_TYPE	Read or Write	M_PENDING_ WRITE	M_WRITE_FAILED
M_INVALID_OPERATION	Read	M_PENDING_ READ	Failed Element -> M_READ_FAILED Not Failed Element -> M_APP_READ_REQ
M_INVALID_OPERATION	Write	M_PENDING_ WRITE	Before Failed Element -> M_WRITE_DONE Failed Element -> M_WRITE_FAILED After Failed Element -> M_APP_WRITE_REQ_W_ACK
M_TEMP_INVALID_ OPERATION	Read	M_PENDING_ READ	Failed Element -> M_TEMP_READ_FAILED Not Failed Element -> M)APP_READ_REQ
M_TEMP_INVALID_ OPERATION	Write	M_PENDING_ WRITE	Before Failed Element -> M_WRITE_DONE Failed Element -> M_TEMP_WRITE_FAILED After Failed Element -> M_APP_ WRITE_REQ_W_ACK
M_ELEMENT_DOES_NOT_ EXIST	Read	M_PENDING_ READ	Failed Element -> M_READ_FAILED Not Failed Element -> M_APP_READ_REQ
M_ELEMENT_DOES_NOT_ EXIST	Write	M_PENDING_ WRITE	Before Failed Element -> M_WRITE_DONE Failed Element -> M_WRITE_FAILED After Failed Element -> M_APP_ WRITE_REQ_W_ACK
M_DEVICE_FAILED	Read or Write	M_PENDING_ READ	M_READ_FAILED
M_DEVICE_FAILED	Read or Write	M_PENDING_ WRITE	M_WRITE_FAILED
M_DEVICE_FAILED	Read or Write	M_APP_READ_ REQ	M_READ_FAILED
M_DEVICE_FAILED	Read or Write	M_APP_WRITE _REQ	M_WRITE_FAILED
M_DEVICE_FAILED	Read or Write	M_APP_WRITE _REQ_W_ACK	M_WRITE_FAILED

NOTE: It is the responsibility of the application to clear any element state that is in a state other than those handled by the agent (APP requests or PENDING states are handled by the agent). Also, if something is PENDING and no response ever comes in for that element, it will remain PENDING unless the application does something with it since there is no timeout for a PENDING element (it is not feasible to have a timer for each element).

RECEIVING A MESSAGE

A device may contain both local and remote databases. Local databases belong to the device itself, while remote databases are images of databases on other devices and are used to communicate with the other devices. All databases have a network address associated with them. When a message is received with the Remote Database Bit set the agent knows that the message deals with the database corresponding to the source address of the message, otherwise the destination address is used to find the database. If the message is an all call, it is always destined for the unit's own database so the agent's own address is used to find the correct database. The following graphically portrays which database a message is written to:



Once again, a Local database is defined as a database that is "owned" by a particular application while a Remote database belongs to a device outside of the application somewhere on the network and is nothing more than a reflection of that remote device's database. Remote database are used to communicate with the remote device. The agent associates databases with network addresses in order to form messages to send through the network, and the application uses the element states of the database to inform the agent that it wants a particular element sent.

The following table shows which database writes are valid. Reads may only be performed upon a device’s local database. LOCAL refers to a database belonging to the application with the agent in question, REMOTE refers to a database belonging to another application somewhere on the network.

FROM	TO	ACTION
LOCAL	LOCAL	Application writes to its own database.
LOCAL	REMOTE	If an application asks the agent to send an element or elements of a local database, the agent must send the message as all call since no destination address is present. In the case of a read operation where the remote queries a local database the return address is provided in the protocol message and the write is valid. In both cases the remote database bit is set in the response to indicate the information is associated with the sender's address.
REMOTE	LOCAL	Application informs agent to send an element or elements. Agent knows where to send information since the destination address will be equal to the Remote database's address.
REMOTE	REMOTE	Not allowed.

If an application changes the element state of any element in a Local database to APP_WRITE_REQ then the agent will send the element out in an all call message. This is implied since the agent does not have a destination address to write the data.

CONTROL BITS

The following table shows combinations of control bits, whether or not they are valid on both the local bus and when sent outside the local bus. If the combination is valid, the returned control byte is shown in the same order.

Control Byte						
Read	Write	Grant	Remote Bit	Local Bus	Response X = invalid	Description
1	0	0	0	1	0101	Immediate response read of destination's local DB
1	0	0	0	0	X	Network cannot immediately respond to a read
1	0	0	1	1	X	Cannot read a remote DB (no network address)
1	0	0	1	0	X	Cannot read a remote DB
1	0	1	0	1	X	Cannot delay a read on a local bus. Device must be able to respond immediately to a read since there is no buffering.
1	0	1	0	0	0101 Delayed	Delayed response read of destination's local DB. This is OK since the device may respond immediately (grant bit on different bus is ignored).
1	0	1	1	1	X	Cannot read a remote DB. Cannot delay a read on a local bus.
1	0	1	1	0	X	Cannot read a remote DB
1	1	0	0	1	0101	Immediate response write of destination's local DB w/verify
1	1	0	0	0	X	Network cannot immediately respond to a write w/verify
1	1	0	1	1	0100	Write to destination's remote DB w/verify
1	1	0	1	0	X	Network cannot immediately respond to a write w/verify
1	1	1	0	1	X	Cannot delay a read on a local bus.
1	1	1	0	0	0101 Delayed	Delayed response write of destination's local DB w/verify
1	1	1	1	1	X	Cannot delay a read on a local bus.
1	1	1	1	0	0100 Delayed	Delayed response write of destination's remote DB w/verify
0	0	0	0	1	Reserved	00 in read/write reserved
0	0	0	0	0	Reserved	00 in read/write reserved
0	0	0	1	1	Reserved	00 in read/write reserved
0	0	0	1	0	Reserved	00 in read/write reserved
0	0	1	0	1	Reserved	00 in read/write reserved
0	0	1	0	0	Reserved	00 in read/write reserved
0	0	1	1	1	Reserved	00 in read/write reserved
0	0	1	1	0	Reserved	00 in read/write reserved
0	1	0	0	1	Next Message	Local DB write message
0	1	0	0	0	Next Message	Local DB write message
0	1	0	1	1	Next Message	Remote DB write message
0	1	0	1	0	Next Message	Remote DB write message
0	1	1	0	1	Next Device	Local DB write message with grant
0	1	1	0	0	Next Device	Local DB write message with grant
0	1	1	1	1	Next Device	Remote DB write message with grant
0	1	1	1	0	Next Device	Remote DB write message with grant

Note:
A "null" message, useful for passing the token, will always have the write bit set by default. The agent knows the message is null since there is no data section. While having the write bit set in a null message is a contradiction, it leaves open '00' in the command bits (read and write bits) for future use.

SPECIAL DATA TYPES IN THE AGENT

There are several data types handled by the agent that need further explanation. These are Bits, Strings and Word Groups. This section details how the agent implements each data type.

Keep in mind that for each database type there is a structure of type element_access which is an array of descriptions of each element within the database type. The element_access array is indexed into by the element number. This description contains the type of the element, an offset from the database in memory, and additional information for special data types called type_info. Type_info contains the bit mask for bits, the length for strings and the length in words for word groups. Since type_info is a single byte, the length of strings and word groups is limited.

Bits

The protocol writes bits by sending a bit mask of the affected bits followed by the values of the bits. If only one bit is to be sent, this means that two bytes must be sent in the protocol. However, eight bit elements may also be changed in a single write.

For example if the Bus Master wishes to tell device one to be on-line, it must set the device_status bit of the remote unit's database 0. This message would appear as follows:

Preamble		Control	Dest	Source	Count	DB	EL	Length	Mask	Data	CKSM	
0xFF	0xFF	0x0C	0x01	0x00	0x05	0x00	0x05	0x02	0x80	0x80	?	?

Note that the above message includes an acknowledge in the control byte since both the read and write bits are set.

When reading bits, the protocol requires that the entire byte be sent back. The message written back will have the mask set for each valid bit within the byte. Note that if an element is read that is in the middle of a set of bits within a byte, the entire byte is sent back but the bit mask will not include those elements within the byte that are before the element requested.

Read request of bits from database 0 starting at Cycle Power:

Preamble		Control	Dest	Source	Count	DB	EL	Length	CKSM	
0xFF	0xFF	0x08	0x01	0x00	0x05	0x00	0x06	0x01	?	?

The response from device one should look something like this:

Preamble		Control	Dest	Source	Count	DB	EL	Length	Mask	Data	CKSM	
0xFF	0xFF	0x24	0x01	0x00	0x05	0x00	0x05	0x02	0x60	0x80	?	?

Note that the control byte in the response includes the remote database bit. The mask includes elements six and seven which are Cycle Power and Reset. The data, 0x80 shows that the device is on line and has not been told to Cycle Power or Reset. The on line portion is sent with the data even though it was not requested, but it will not be written to the database since the mask does not show that bit to be valid. Cycle Power and Reset bits in the database will be set to zero.

Strings

Strings are always preceded by a length. This means it is not necessary to write the entire string. However it is necessary to include the entire possible length of the string when requesting the string. This is needed in order for the data section parser to know if any data beyond the string is being requested. For example, if the Bus Master wished to request the Fin_Good_Num from device three, the message would look like this:

Preamble		Control	Dest	Source	Count	DB	EL	Length	CKSM	
0xFF	0xFF	0x08	0x03	0x00	0x03	0x00	0x0D	0x10	?	?

If there was a one byte element beyond the Fin_Good_Num in the database (which there is not) we could read it by changing the length to 0x11. Since the length of the string is part of the string it is included in the length when requesting or writing the string.

Word Groups

Word groups are preceded by an offset in words (one byte) and a count in words (one byte). Unlike a string, the agent treats this as a separate word from the data, not as part of the data. The definition of a word group in the agent database is very strict. The offset_length part of the word group should be a word in memory that in contiguous with the word group itself which should be defined as an array of words. The status_timeout of the Bus Master Database is the only example presently. Note in the definition of db1_type, the offset_length is defined separately from the word group status_timeout. This is done for ease of indexing in the status_timeout array. If someone wants to read the status and timeout of device zero, the offset would be zero and the length would be one (word). For instance, if a device (say a hub at address 6) wanted to read the status information from the bus master for devices 4 through 5 (devices start at 0), the message would be as follows:

Preamble		Control	Dest	Source	Count	DB	EL	Length	Offset	Len	cksm	
0xFF	0xFF	0x08	0x00	0x06	0x05	0x01	0x02	0x06	0x04	0x02	?	?

Note the Length field value of 0x06. This is the number of bytes we are requesting from the word group including the offset and length bytes. This number is calculated by multiplying the length portion of the offset and length word by two and adding two for the offset and length bytes that will be returned. If the length were greater than six this would be requesting data beyond the word group (another element) as well as the word group.

The response:

Preamble		Control	Dest	Source	Count	DB	EL	Len	Offset	Len	Data				cksm	
0xFF	0xFF	0x24	0x06	0x00	0x09	0x01	0x02	0x06	0x04	0x02	0x80	0x0a	0x80	0x0a	?	?

Whenever a change is made to a word within a word group, not including the offset-length word, the MSB of the word group is set to indicate that the word has been changed by the agent. The element state of the entire word group is also changed. This completes communication from the agent to the application.

For communication from the application to the agent, the offset-length word is used to inform the agent which parts of the word group to send. The element state is still used to inform the agent of the need to send some or all of the word group element.

Important note:
In order for the application to tell the agent which part of the word group to send, the application must alter the offset and length bytes of the word group. There is no default that writes the entire word group when it is requested to be sent. Whatever values are left in offset and length will be the ones used to determine which part of the word group to write.

AGENT RULES AND REQUIREMENTS

- Agent will set grant bit for any message written outside the local bus be it read or write.
- The preamble is not included in the checksum.
- Check sum is calculated on the compressed data.
- Length of the data field is in bytes not element numbers.
- Count is count of compressed data.
- The application is not involved in the response process. If a write with acknowledgement occurs, the agent performs the write regardless of the value (no range limits in agent) and responds whether or not the write was performed.

THE BUS MASTER (A SPECIALIZED AGENT)

The Bus Master bypasses the *m_agent( )* loop of the agent for expediency. The Bus Master is a specialized agent with its own state machine, which is responsible for keeping the bus going at all times. It must keep track of timeout values for devices on its bus, time out devices that fail to respond, substitute messages for devices that have failed, and pass the token to devices where a gap in active devices on the bus exists. The Bus Master is also responsible for activating and deactivating devices on the bus.

If the device to be created is going to be a Bus Master, the following need to be done in each application above and beyond the requirements stated above for an agent:

In your main code:
1. Call *bus_master( )* instead of *m_agent( )* from the main loop.
2. Initialize *bus_master_db.address*. The full network address should be available to the Bus Master since it must supply this information to the devices on it's bus.

In a timer interrupt:
1. Decrement *bm_poll_timer* every millisecond if not 0.
2. Decrement *bm_message_timer* every millisecond if not 0.

Communication routines:
1. Make sure *m_tx_in_prog* is handled correctly. In the example code it is set in *tx_m_message( )* and cleared in *serial1_tei( )*. This means that the flag is set during the entire time of transmission and when the last bit is transmitted the flag is cleared. The routine *serial1_txi( )* pulls data from the transmit buffer and places it in the uart, during which time the Bus Master is waiting in M_ST_WAIT_TRANSMIT. When the flag *m_tx_in_prog* is cleared, the Bus Master sets the *bm_message_timer*. The receiver is disabled during transmit so that we do not receive our own transmission. If we receive our own transmission, it will be parsed by the *m_look_for_message( )* routine which resets *bm_message_timer* to one character time (thinking that the message we are timing out has started to arrive).

In agent.h:
1. BUS_MASTER must be defined. This will define M_USE_DATABASE1 and make M_NUMBER_INSTANCES_DB0 = 1, but will not increment M_TOTAL_NUMBER_OF_DATABASES (which is left to you).

2. Make sure M_ONE_CHARACTER_TIME is set to something reasonable, based upon baud rate preferrably. This constant is used by the receive data parser *m_look_for_message( )* to reset the *bm_message_timer* to time out in between bytes of a message.
3. Make sure BM_LOCAL_TIMEOUT is set to something reasonable. This is the amount of dead time allowed on the bus for each device before it must start transmitting, then M_ONE_CHARACTER_TIME takes over for in between bytes. M_LOCAL_TIMEOUT is written into the bus master's database *status_timeout* for each device, however the *status_timeout* element of the bus master's database may be changed by the outside world.

The database of the bus master contains status information for each device on the bus called *status_timeout* which is a Word Group. Each word represents information for the corresponding device number. The low byte of each word is the timeout. The high byte of each word is status information. Under normal conditions the bus master will start up with all status bits equal to zero except its own. The bus master will then poll each device and if it finds a device active it will set the appropriate status bit in its database. However, the status information may be changed externally. The following table describes the actions carried out by the bus master when it is time for the bus master to poll and when the status information for a particular device changes externally.

STATUS BITS BEFORE POLL				ACTION BY BUS MASTER ON POLL	TEMP STATUS BITS				STATUS BITS IF POLL SUCCEEDS				STATUS BITS IF POLL FAILS			
LA	LF	NE	LO		LA	LF	NE	LO	LA	LF	NE	LO	LA	LF	NE	LO
0	0	0	0	Send Device off, Listen only off	0	0	0	0								
1	x	0	0	Poll with Listen Only off	0	x	0	0	1	0	0	0	0	x	0	0
x	1	0	0	Poll with Listen Only off	0	1	0	0	1	0	0	0	0	1	0	0
x	x	0	1	Poll with Listen Only on	0	x	0	1	1	0	0	1	0	x	0	0
x	x	1	x	If Not Exist is set, clear all other bits Send Device off, Listen only off	0	0	1	0								

Where "LA" is Local Active, "LF" is Local Failed, "NE" is Not Exist and "LO" is Listen Only. Local Active means that the device is on line and is expected to get the token when it's turn occurs. Local Failed means that a device that was on line and failed to take the token or respond to queries 3 times in a row. Local Active and Local Failed are considered read only, even though the outside world can manipulate them. Not Exist is used to keep a device from responding or being polled. Listen only is used to keep a device from getting the token, yet the device is allowed to respond to queries from other devices.

As it can be seen from the table, anytime Not Exist is set externally, all other bits are set to zero since they are not considered valid. Anytime Listen Only is set externally, Local Active and Local Failed are cleared.

In the non bus master agent, there are two bits corresponding to Local Active and Listen Only in database 0.

LOCAL ACTIVE (DEVICE ON)	LISTEN ONLY	ACTION ON RECEIVE WRITE	ACTION ON RECEIVE READ	ACTION ON APPLICATION REQUEST TO WRITE
0	0	Perform write	No response	Fail Write
0	1	Perform write	Respond	Fail Write
1	0	Perform write	Respond	Write
1	1	Perform write	Respond	Fail Write

BUS MASTER RULES AND REQUIREMENTS

- The bus master must know it's full network address. This can be provided by three 8 lever dip switches (or a user interface) since it is always device address 0.
- The bus master must provide the upper 3 bytes of network address to each unit when it turns a unit on line.
- Bus master does not timeout messages outside of the local bus.
- If the grant bit is on and failure occurs or the next device to receive the token is failed, the Bus master must substitute the source address of device that has failed for its own address in the null message so that the next device knows it is its turn to take the token.
- If the grant bit is off and failure occurs the Bus master must substitute a message for the failed device. The Bus master must use the destination address of the last message as the source address so that the device with the token knows which address to associate failures with and will be able take any outstanding element states (PENDING) to the address and place them in the FAILED element state.
- The bus master keeps a single variable for the highest address it has seen talking on the bus. It also keeps 3 bits for each possible address on the local bus that is the counter of the number of times in a row that device has failed. If the counter of any device reaches three a message is sent to a provided address indicating the failure.
- A device added to the system always comes up as inactive. It must wait for the bus master to "poll" it. The bus master will therefore "poll" for an added device occasionally. The poll includes a write to the *Device On* element of database 0, a write to the *Listen Only* element of database 0, and a write to the upper 3 bytes of database 0 to provide the full network addressing. The read bit is set for acknowledgement of the command. If the command is successful, the device is added to the list of active devices in the bus master database.
- When the bus master comes up, it assumes control of the bus. This might be a problem if we wish to have a redundant bus master?

APPLICATION REQUIREMENTS

- The application must clear all element states and database change of state bits that are not handled by the agent.
- Initialize databases.
- Initialize agent address.

VARIABLE AND STRUCTURE DESCRIPTIONS

BUSMASTER ONLY

<u>bus_master_db</u>	An instance of database 1. Necessary for Bus Master.
<u>bm_message_timer</u>	A timer used by the Bus Master to time out "dead time" on the bus. Dead time is the amount of time devices have to begin sending data on the bus when they receive the token or need to respond to a read request.
<u>bm_poll_timer</u>	A timer used by the Bus Master. When the timer reaches 0 and the Bus Master has the token, it will initiate a "poll".

AGENT

<u>agent_db</u>	An instance of database 0. All applications should have this database as it is used in acknowledgements by the agent.
m_agent_address	The agent's working network address. Database 0 contains this information for communication.

m_current_msg	Contains control, source, destination, and count of last message seen on bus. Also updated in transmit routines since data that we transmit is no longer parsed by our own receive.
m_data_section_begin	Pointer into m protocol receive ring buffer to beginning of data section (after count byte).
m_data_section_end	Pointer into m protocol receive ring buffer to end of data section (before checksum).
m_dbX_access[ ]	Where X is the type of database, this array contains access information for each element of the database type. This information can be used for all databases of the same type no matter how many instances of the database exist.
m_loop_state	Current processing state of agent.
m_message_ready	Flag to indicate that a message has been prepared and is waiting for transmit.
m_rx_buf	Receive ring buffer for M protocol port.
m_rx_head	Pointer into m protocol receive ring buffer indicating where data should be added to the ring buffer and where the last data was added in the ring buffer.
m_rx_tail	Pointer into M protocol receive ring buffer indicating data that has not yet been processed. This is incremented after data is parsed.
m_tx_buf	Transmit ring buffer for M protocol port.
m_tx_head	Pointer into M protocol transmit ring buffer indicating where data should be added into the ring buffer and where the last data was added in the ring buffer.
m_tx_in_prog	Flag to indicate transmission is occurring. Also used to initiate tranmit interrupt.
m_tx_tail	Pointer into M protocol transmit ring buffer indicating data that has not yet been transmitted.
m_next_loop_state	Used to transition the agent to a different processing state.
m_outgoing_data[ ]	An array of raw data ready for transmit. Used in conjunction with out_data_header[ ].
m_outgoing_data_head	A pointer into the outgoing_data array indicating where data should be added and the last position where data was added.
m_outgoing_data_tail	A pointer into the outgoing_data array indicating last processed data.
m_out_data_header[ ]	Used in buffering data for transmit. This is an array of header information for fields to be transmitted.
m_out_data_header_index	Index into out_data_header array. Used to indicate position where next field should be added. Subtract one for current field.
m_update_address_from_db0	A flag that indicates that the network address in database 0 has been changed by another device on the network and it is time to update our working address.

LIST OF STATIC VARIABLES

outgoing_database_s	In buffer_outgoing_data( ) remembers which database we are buffering for the next call to the function.
outgoing_element_s	In buffer_outgoing_data( ) remembers which element we are buffering for the next call to the function.
last_bitmask_position_ptr_s	In *m_buffer_outgoing_data*( ) remembers position of last bit mask in case the next call to the function buffers another bit within the same byte.
compress_count_s	In *m_compress_outgoing_data*( ) this counts the number of 0xFF bytes sent to the function in a row after do_compress is set.
do_compress_s	In *m_compress_outgoing_data*( ) this signals that we have received a 0xFF and must start counting them for compression.
parser_state_s	Serial data from the M protocol port is parsed one byte at a time, this variable keeps the state of the parser *m_look_for_message*( ) through each call to the function.

msg_in_prog_s	Message information for data that is currently being parsed from the M protocol port by *m_look_for_message*( ).
rcvd_data_count_s	In *m_look_for_message*( ) this is the actual count of data received which is compared with count of data byte (or word - may be extended) so that we know when to stop looking for data section.
c0_s	In *m_look_for_message*( ) this is the received checksum that is updated after every byte is received.
c1_s	Same as C0, second byte of checksum.
run_length_next_s	Used to decompress during parsing in *m_look_for_message*( ).
run_length_s	Used to decompress during parsing in *m_look_for_message*( ).
c0_tx_s	Checksum that is calculated each call to prep_send_m_message( ).
c1_tx_s	Same as C0_TX, second byte of checksum.
tx_cntrl_s	Remembers control byte information for message being sent out in *m_prep_send_m_message*( ).
decompress_next_s	When parsing data section, this signals we have received a 0xFF and must decompress information on the next call to *m_get_next_data_byte*().
decompress_length_s	When parsing data section, this is the number of 0xFF we must return in decompressing the data section.
next_state_after_tx_wait_s	In the Bus Master this is the next state to transition to after leaving state M_ST_TX_WAIT.

STRUCTURE DESCRIPTIONS

m_dbX_type	Actual definition of database type X, where X is the type of the database.
m_db_instance	An element of db_array[ ]. Defines an instance of a database by providing a network address, database type, number of elements and pointer to the database in memory.
m_element_access	Contains element type, an offset for the element from the beginning of the database, and a bit mask if the element is a bit or string length if the element is a string.
m_outgoing_header	Contains header information for a field of the data section with pointer into the outgoing data buffer to which data is associated with the field.
m_message_info	Contains message information such as control byte, destination. source and count of bytes in message.
m_network_address	Four bytes of network address.