Mark DiVecchio
markd@silogic.com
V2.02 11 Sep 2019
http://www.silogic.com/trains/RTC/Engine%20RAM%20Mapping.pdf

Web pages:

http://www.silogic.com/trains/RTC_Running.html
http://www.silogic.com/trains/OOK_Radio_Support.html
http://www.silogic.com/trains/ADPCM.html
http://www.silogic.com/trains/RFID.html
http://www.silogic.com/trains/RTC_Control_Language.html

Refer to this doc available on the RTC_Running page: QUERY command uses.rtf

http://www.silogic.com/trains/RTC/QUERY%20command%20uses.rtf

Here I will write up what I have been able to figure out about the RAM memory in each engine. This was all done by sending commands to the TIU and examining the responses. As I had stated before and as appears on my web pages: "**I figured this out just by looking at the RS-232 stream over the radio. No code disassembly, no logic analyzers, no opening up of Remotes or TIU's.**" Eric Linz gave me some ideas when it came to decoding the responses from the DOD, DCH and DTO commands.

As I saw when working on the Sound File with my ADPCM program, multi-byte data in the RAM is stored big-endian (sure sign of using Apple computers rather than Windows computers to develop sound files). This means that in multi-byte data, the lowest address contains the most significant byte of the data.

The C++ code for the Remote Train Control is released under the GNU General Public License, v3 or later. If you want the entire code base, it is available on my RTC_Running web page. The latest version does not get on the web page for while after the binary release so that I can fix any bugs that might show up. At any time, though, if you want the latest code, send me an email.

How to get information from the engine.

1. Use remote commands:

DOD	Odometer
DCH	Chronometer
DTO	Trip Odometer
DTV	Track Voltage
?	Battery Status
FRM	Maintenance
I0	Lists engine known by the TIU

Ixxx Lists the name of the engine and the soft keys that it supports

2. Read the RAM directly (this is NEW!)

The “q” or Query command is used by the remote commands. I found that this command directly accesses the RAM in the engine. It appears to be a 1024 byte RAM (addresses 0x0000 to 0x03FF)

Lengthy study of the q commands as used by the remote, helped me understand the rather unusual address mapping from the q command to actual RAM addresses. Look at the next figure.

A q command consists 16 bits in 4 hex digits broken into these fields:

2 bits – selects the output byte position in the 4 byte response word of the addressed byte.
*1 bit – low order bit of the actual RAM address.
4 bits – all 1’s
*2 bits – inverted 2 high order bits of the actual RAM address
*7 bits – next 7 bits of the RAM address

The *2 bits + *7 bits + *1 bit make up the 10 bit RAM address for the 1024 byte RAM.

Here is my C code #define for the translation of a 10 bit RAM address to the 4 hex digit ‘q’ command:

```
//---------------------------------------------------------------------------
//
// Convert a RAM address (0 - 0x3FF) and byte number (0-3) into a "q" command address
//
#define RAMTOQ(adr, b) ( (b << 14) | ((adr & 0x0001) << 13 | (adr & 0xFFE) >> 1) ^ 0x1F80)
//---------------------------------------------------------------------------
```

For example, what I later describe as the Scale Factor, uses the following three q commands:

q9F84
qFF84
q

Each one of those commands returns 4 bytes. Only two bytes are meaningful here, the 2nd byte as selected by the high order bits of the ‘9’ and the 3rd byte as selected by the high order two bits of the ‘F’. Each command prepares a one new byte of information to be sent by the Engine. The next ‘q’ command actually fetches that byte.

So when the final empty ‘q’ command is sent, the Scaling Factor value appears in byte 2 and byte 3 of the 4 byte response:

```
    // pull out the scale factor
    // 965A 99A6 5555 5555 5559 5559 A596 6566 555A
    //   87   D9   00   00   04   04   E1   31   05
    //                                 ^^^^^^^^^   Scale Factor
```

I’ll talk later about what this Scale Factor value means.

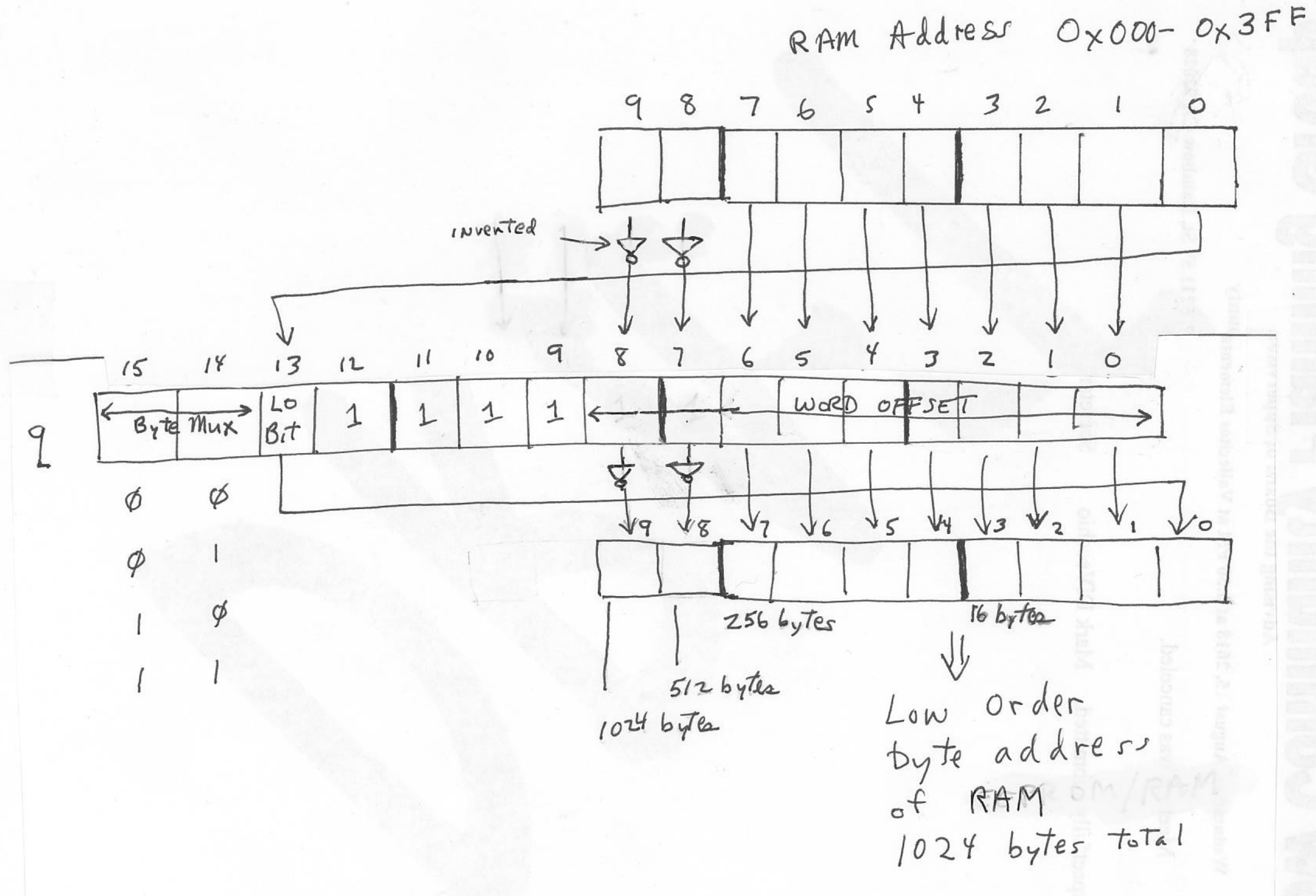
The DTO command uses these five q commands:

q1F9E
q7F9E
q9F9F

```
qFF9F
q

      // pull out the value
      // 965A 99A6 5555 5555 5559 5559 A596 6566 555A
      //   87   D9   00   00   04   04   E1   31   05
      //                          ^^^^^^^^^^^^^^^^^^^   Trip Odometer Mileage
```

These five q commands return 4 bytes.



9Feb19 MARK DIVECCHIO

Once I figured out this address translation, I added a Dump routine to my RTC program which can dump any number of 16 byte lines from the engine RAM memory. Here is a dump of the first 16 lines of RAM from 20-20246-1 GP-7 Diesel Engine P&LE #1500 PS3:

```
Offset(h)  00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
           -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
0x00000000 B1 62 55 D5 02 9A 01 F8 54 AB 78 F1 00 0E 64 E9
0x00000010 7F BC 7F BC 7F BC 7F BC 00 00 00 03 00 1A 00 01
0x00000020 00 01 21 05 A2 6F 4C CC 05 A8 01 6A 0E 4F 00 11
0x00000030 00 00 00 00 00 00 B7 05 28 CD 91 E1 00 00 00 00
0x00000040 00 00 02 96 00 00 00 00 00 00 00 00 0F E9 00 EE
0x00000050 00 00 FF FF 00 00 47 21 00 00 00 01 00 05 00 00
0x00000060 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 01
0x00000070 00 00 80 00 01 B4 01 2C 0A 23 00 00 00 00 02 9B
0x00000080 00 00 00 00 00 00 00 00 00 E4 00 55 80 00 00 00
0x00000090 02 58 00 00 00 00 00 00 00 00 00 0A A0 02 9B
0x000000A0 00 00 00 00 00 00 00 01 C3 00 00 F8 FE 00 00
0x000000B0 00 05 00 02 00 00 00 00 00 00 02 00 05 02 20
0x000000C0 00 11 80 00 00 00 0B 63 00 0D FF FF 00 00 00 00
0x000000D0 00 00 00 00 00 00 00 06 7F FF 02 5D 23 80 00 00
0x000000E0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x000000F0 03 70 04 20 00 00 00 00 02 92 E9 B9 00 00 4B CE
```

How did I know this was right? I took responses from the DOD, DCH and DTO commands and searched this RAM for the values returned. I could see that the DOD value was at address 0x34 to 0x37. Looking at the ‘q’ commands for DOD, I could see that the addresses contained in there mapped to 0x34 to 0x37.

I did more experimenting, manually translating of the ‘q’ commands and dumping of memory of other engines and I found in the RAM:

DOD	0x34-0x37
DCH	0x38-0x3B
DTO	0x3C-0x3F
DCS engine number	0x0D
Scaling Factor	0x08-0x09

Now I started changing settings in the engine and then dumping out the low part of RAM. I looked for changes and found them.

I found 16 bit values for all of the volume settings. I found two bytes that are bit mapped to indicate which lights/features were on or off. I found the engine speed as requested by the ‘s’ command and I found the instantaneous engine speed (as the engine is accelerating or decelerating to the requested speed).

Look at the next figure which shows the RAM mapping that I have found so far.

ENGINE RAM MAPPING

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00									Scaling Factor					Eng #?	Power on Startup Master Vol	
01	Eng Vol		Acc Vol		Horn Vol		Bell Vol									
02			?	Eng Type	Status 2	Status 1										
03					DOD				DCH				DTO			
04													DTV			
05																
06	PS3 Speed ACTUAL	PS2 speed	PS2/PS3 speed GOAL					Dir								
07																
08			Battery													
09																
0A																

Vol 0 - 32700
each 1% is 327

other mappings
battery - 0x82→0x83
FRM - 0x324→0x325

unknown: chuff rate
acc/deacc rates
Dieral Rev Level

13 Feb 2019 MARK DIVECCHIO

Lights/features Status

The next drawing shows the bit mapped status for lights/features in the engine. Note that I’ve not figured out what all of the bits mean. When the remote reads up this information, it reads 4 bytes of information from RAM 0x22 thru 0x25. The status bytes really only use bytes at RAM 0x24 to 0x25 so you could just read up 2 bytes.

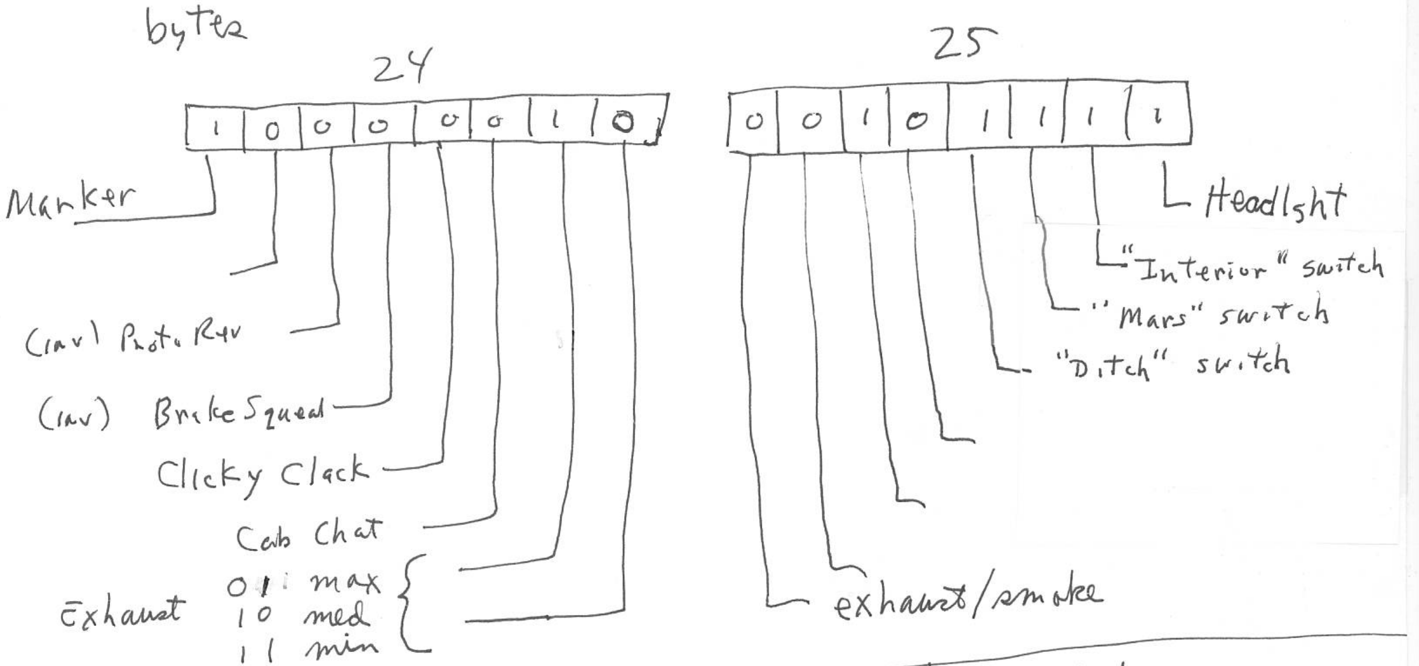
As the drawing shows, the byte at RAM 0x23 tells whether the engine is steam or diesel. I don’t know what the byte at 0x22 means.

9 Feb 19 Mark DiVecchio

bytes	22	unknown
	23	0x00 = steam 0x05 = diesel 0x85 = diesel
	24	status
	25	status

q 1F91
q 7F91
q 9F92
q FF92
q

bytes



ENG		byte 22	23	24	25
1	#2808	22	05	82	07
2	#211	20	00	80	07
3	#896	20	05	82	0F
4	#1208	20	05	82	03
5	#9378	20	05	82	47
6	#9401	20	05	82	03
7	#1501	21	05	82	4F

ENG		byte 22	23	24	25
8	#1561	00	05	02	03
9	#1556	20	05	82	2F
10					
11	#9060	21	00	82	5B
12	#2057	22	05	82	25
13	#1500	21	05	82	4B
14	#2060				
15	#2061	22	05	82	07
16					

Decodings that I've figured out.

These decodings are guesses by me and a few others. They seem to match the remote fairly closely but they are not exact. I believe that they are more accurate than the remote as I compared my results to actual physical distances and wall clock time.

DCH

The chronometer value is 4 bytes. I looked at the chronometer value over 25 minutes. I divided the difference between the beginning value and ending value. This gave me a figure of 31.3197 ticks of DCH per second. Here is the decoding equation:

```
// Send DCH ‘q’ command sequence
float t = DCHbyteResp[5] << 24 | DCHbyteResp[6] << 16 |
      DCHbyteResp[7] << 8 | DCHbyteResp[8];
t /= 31.3197;     // estimated ticks per second
const int tt = static_cast<float> (t);
const unsigned int hours = (tt/(60 * 60));
const unsigned int minutes = (tt/60) % 60;
const unsigned int seconds = tt % 60;
```

I can't figure out where the 31.3197 ticks/sec comes from. Maybe some divide down from a clock frequency in the engine.

DTO

Trip odometer command uses the Scale Factor ‘q’ command sequence followed by the DTO ‘q’ command sequence. I laid out a 1/10 scale mile length of track on my layout and ran an engine over that length of track noting the value of DTO before and after. By trial & error, I found that this equation accurately gave me a trip odometer value of 1/10 Smile.

```
// Send Scale Factor ‘q’ command sequence
fSF = FactorResp[7] << 8 | FactorResp[8];

// Send DTO ‘q’ command sequence
const unsigned long lVal = DTObyteResp[5] << 24 | DTObyteResp[6] << 16 |
      DTObyteResp[7] << 8 | DTObyteResp[8];
const float DTO = (lVal)/(fSF * 11.0);
```

I can't explain why the DTO adjustment factor is 11.

DOD

Odometer command uses the Scale Factor ‘q command sequence followed by the DOD ‘q’ command sequence. I got a hint from Eric Linz about this decoding. He uses the scale factor in the same way but multiplies the result by 48.0. Eric liked the 48 because it is related to 1/48 scale of O gauge. I found that 48.0 did not give an accurate result. Trial & error gave me 46.5.

```
// Send Scale Factor ‘q’ command sequence
fSF = FactorResp[7] << 8 | FactorResp[8];

// Send DOD ‘q’ command sequence
const unsigned long lVal = DODbyteResp[5] << 24 | DODbyteResp[6] << 16 |
      DODbyteResp[7] << 8 | DODbyteResp[8];
const float DOD = (46.5 * lVal)/(fSF);
```

I can't explain why the DOD adjustment factor is 46.5 but Stan on the OGRForum noted that
    11.0 * 46.5 = 511.5
which might be 512.0 (2^9) if I knew those adjustment factors more accurately.

Volume

The volume values are stored in units of 327. Off is indicated by a value of 0. 100% volume is a value of 32700 (0x7FBC).

In the Sound file where the initial sound volumes are stored, I see the 100% volume set to 0x7FFF or 32767. Once the user lowers the volume, it cannot be raised above 32700 (until I figure out a way to write to the RAM).

Engine Speed

There are two values, the set speed and the instantaneous speed:

Set Engine Speed is the speed requested by the last ‘s’ command. It is stored in RAM locations 0x62 and 0x63. Take this binary value and divide it by 8 to get actual speed in Smph.

Instantaneous Engine Speed is the actual speed of the engine on the way to or from the requested speed. This value seems to be different between PS2 and PS3 engines

In PS2: engines, it seems to be stored in RAM locations 0x60 and 0x61. Take this binary value and divide it by 8 to get instantaneous speed in Smph.

In PS3 engines, it seems to be stored in RAM location 0x60. Take this binary value as the instantaneous speed in Smph.

Detecting if the engine is PS2 or PS3.

To get the instantaneous engine speed, you have to know if the engine is PS2 or PS3. I don’t have any idea if there is a bit in RAM that tells us this.

I dumped out about a dozen engines and compared the RAM contents between them. I could not find a particular bit or byte to determine the PS level.

But I was able to find, among my engines, that byte 0x50 is zero in every PS3 engine and non-zero in every PS2 engine. I don’t know what this byte means or if my assumption is correct over all engines. That remains to be seen. But, at least, with this, I can decode the instantaneous speed correctly for my engines.

Note About the Sound File

It appears that the first 104 bytes of the Sound File are transferred to the first 104 bytes (0x00 thru 0x67) of RAM when a new Sound File is loaded. These bytes seem to serve as initial values for engine capabilities. For example, the scale factor used by DTO and DOD processing at RAM 0x08 -0x09 and the 5 volume levels set at 100+%.

Here is some code that might be useful:

```
//---------------------------------------------------------------------------
//
// Convert a RAM address (0 - 0x3FF) and byte number (0-3) into a "q" command address
//
#define RAMTOQ(adr, b) ( (b << 14) | ((adr & 0x0001) << 13 | (adr & 0xFFE) >> 1) ^ 0x1F80)
//---------------------------------------------------------------------------
//---------------------------------------------------------------------------
byte __fastcall TMainForm::FetchByte(int addr, byte Eng, byte TIU)
{
//
// Fetch one byte of RAM memory
//
assert(addr <= 0x03FF);
byte FBResp[CMDMAX];
const unsigned int TranslatedAddr = RAMTOQ(addr, 0);
char cmd[15];
snprintf(cmd, sizeof cmd, "q%04X", TranslatedAddr);
if (!Query(cmd, FBResp, Eng, TIU, RemoteNo)) return 0;
if (Query("q", FBResp, Eng, TIU, RemoteNo)) {
      return FBResp[5];
      }
return 0xFF;
}
//---------------------------------------------------------------------------
unsigned short __fastcall TMainForm::FetchWord(int addr, byte Eng, byte TIU)
{
//
// Fetch two bytes of RAM memory
// addr points to the lowest byte address
//
assert(addr <= 0x03FF);
byte FBResp[CMDMAX];
unsigned int TranslatedAddr;
char cmd[15];

TranslatedAddr = RAMTOQ(addr, 0);
snprintf(cmd, sizeof cmd, "q%04X", TranslatedAddr);
if (!Query(cmd, FBResp, Eng, TIU, RemoteNo)) return 0;
addr++;
TranslatedAddr = RAMTOQ(addr, 1);
snprintf(cmd, sizeof cmd, "q%04X", TranslatedAddr);
if (!Query(cmd, FBResp, Eng, TIU, RemoteNo)) return 0;

if (Query("q", FBResp, Eng, TIU, RemoteNo)) {
      return FBResp[5] << 8 | FBResp[6];
      }
return 0xFFFF;
}
//---------------------------------------------------------------------------
unsigned long __fastcall TMainForm::FetchInt(int addr, byte Eng, byte TIU)
{
//
// Fetch four bytes of RAM memory
// addr points to the lowest byte address
//
assert(addr <= 0x03FF);
byte FBResp[CMDMAX];
unsigned int TranslatedAddr;
char cmd[15];

TranslatedAddr = RAMTOQ(addr, 0); // BYTE0 | ((addr & 0x0001) << 13 | (addr & 0xFFE) >> 1) ^
0x1F80;
snprintf(cmd, sizeof cmd, "q%04X", TranslatedAddr);
if (!Query(cmd, FBResp, Eng, TIU, RemoteNo)) return 0;
addr++;
TranslatedAddr = RAMTOQ(addr, 1); // BYTE1 | ((addr & 0x0001) << 13 | (addr & 0xFFE) >> 1) ^
0x1F80;
snprintf(cmd, sizeof cmd, "q%04X", TranslatedAddr);
if (!Query(cmd, FBResp, Eng, TIU, RemoteNo)) return 0;
addr++;
TranslatedAddr = RAMTOQ(addr, 2); // BYTE2 | ((addr & 0x0001) << 13 | (addr & 0xFFE) >> 1) ^
0x1F80;
snprintf(cmd, sizeof cmd, "q%04X", TranslatedAddr);
if (!Query(cmd, FBResp, Eng, TIU, RemoteNo)) return 0;
addr++;
TranslatedAddr = RAMTOQ(addr, 3); // BYTE3 | ((addr & 0x0001) << 13 | (addr & 0xFFE) >> 1) ^
0x1F80;
snprintf(cmd, sizeof cmd, "q%04X", TranslatedAddr);
if (!Query(cmd, FBResp, Eng, TIU, RemoteNo)) return 0;

if (Query("q", FBResp, Eng, TIU, RemoteNo)) {
      return FBResp[5] << 24 | FBResp[6] << 16 | FBResp[7] << 8 | FBResp[8];
      }
return 0xFFFFFFFF;
}
//---------------------------------------------------------------------------
void __fastcall TMainForm::DumpMemoryButton1Click(TObject *Sender)
{
// Dump Engine memory from 0x00 to 0x6F
// if TIU Port is not connected fail
if (!IsTIUConnected()) {
      ShowMessage(ConnectMsg);
      return;
    }
if (!DEBUGGING) return;
if (FirstLine->ValueAsInt > LastLine->ValueAsInt) return;  // FirstLine must always be <=
LastLine
//
// Dumps from FIRSTLINE to LASTLINE (line = 16 bytes) of either EEPROM or a RAM memory.
// I suspect it is RAM memory as the DCH time is there (at 0x0038) and it is
// incrementing. You really can't do that with a block erase EEPROM.
//
String PLine = "Offset(h)  00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F";

DB->Debug_Print(PLine, clGreen);
    CRCBuffer->SelAttributes->Color = clGreen;
    CRCBuffer->Lines->Append(PLine);
    CRCBuffer->Perform(EM_SCROLL,SB_LINEDOWN,0);     // remain at end of CRC Buffer
PLine = " ";
DB->Debug_Print(PLine, clGreen);
    CRCBuffer->SelAttributes->Color = clGreen;
    CRCBuffer->Lines->Append(PLine);
    CRCBuffer->Perform(EM_SCROLL,SB_LINEDOWN,0);     // remain at end of CRC Buffer

for (int i = FirstLine->ValueAsInt ; i <= LastLine->ValueAsInt ; i++) {
      PLine  = Format("0x0000%s", ARRAYOFCONST((IntToHex((i * 0x10), 4))));
      for (int j = 0x00 ; j < 0x10 ; j = j + 0x04) {
      unsigned int B = FetchInt((i * 0x10) + j, EngineNo, TIUNo);
            PLine = PLine + Format(" %s %s %s %s", ARRAYOFCONST((
            IntToHex((int)(B >> 24) & 0xFF, 2),
            IntToHex((int)(B >> 16) & 0xFF, 2),
            IntToHex((int)(B >>  8) & 0xFF, 2),
            IntToHex((int)(B      ) & 0xFF, 2)
            )));
      }
    DB->Debug_Print(PLine, clGreen);
    CRCBuffer->SelAttributes->Color = clGreen;
    CRCBuffer->Lines->Append(PLine);
    CRCBuffer->Perform(EM_SCROLL,SB_LINEDOWN,0);     // remain at end of CRC Buffer
    PLine = "";
    }
DB->Debug_Print(PLine, clGreen);
CRCBuffer->SelAttributes->Color = clGreen;
CRCBuffer->Lines->Append("");
CRCBuffer->Perform(EM_SCROLL,SB_LINEDOWN,0);   // remain at end of CRC Buffer
}
//---------------------------------------------------------------------------
```