PC Control of MTH Engines by Serial Connection to the TIU

14. Sep. 2020

Mark Divecchio
“San Diego Mark” on the OGR Forums
markd@silogic.com

June 2019 – updates to format of LashUp commands in V5.0 of DCS.

PC Control of MTH Engines by Serial Connection to the TIU

To me, MTH makes the best engines and rolling stock for my railroads of choice, the Pittsburgh & Lake Erie Railroad and the Aliquippa & Southern Railroad (where my grandfather worked). MTH has many many engines and dozen of pieces of rolling stock for these railroads. My layout uses only MTH's DCS.

DCS has been out for almost 15 years. I've been waiting for a way to control my layout using my PC. I've always used Windows PC's so this effort was done originally using XP and more recently Windows 7 and Windows 10.

A few years ago, Mike Hewett presented a PC interface to the tethered mode of operation of DCS. He showed how to sniff out the RS-232 packets running between the Remote and the TIU, how to save those packets and how to later transmit those packets to the TIU from the PC.

Mike presented his findings in three videos which he produced around May of 2011. Look at those three videos before you continue with my description.

Chapter 1 https://www.youtube.com/watch?v=MxlUb-YccZw
Chapter 2 https://www.youtube.com/watch?v=IBrhLSVHjIo
Chapter 3 https://www.youtube.com/watch?v=oqaeeR3pgPw

Look at this OGR Forum thread for a followup:

http://ogrforum.ogaugerr.com/topic/wonderful-gift-dcs-to-pc-control?reply=5371679459030161#5371679459030161

Mike made more progress as shown in this video from 2013

https://www.youtube.com/watch?v=Ug5CSZFwo-c

PC Control of MTH Engines by Serial Connection to the TIU

14. Sep. 2020

I think that I first saw what Mike did around October of 2011 as my earliest date stamps on files that I've saved are from that date.

Mike's methodology was to record the packets sent by the Remote when each key on the Remote was pressed. Without regard to the contents of the packets, he saved them in a file. Then, later, his PC program could read up those saved packets and send them to the TIU. He created a very nice touch screen interface and he could run his DCS trains from his PC.

I contacted Mike back then and he sent me copies of his program and I was able to build up his interface to the TIU, sniff out the needed packets and I had a way to control my trains from my PC.

This worked up to a point. When I added a new engine number, I had to run the packet sniffer again and pick up the packets needed for the new engine number. Mike's program only captured a subset of the many, many types of commands that could be sent to the TIU. Mike did not read back responses from the TIU or process any of those returned packets.

I was looking for something more. I needed to understand the protocol over the tether cable.

With a lot of effort, I was able to understand almost all of the communications between the Remote and TIU. I am now able to create packets to control the DCS engines. The packets are complete with correct addressing, command syntax and CRC.

I figured this out by examination of the packets that I could sniff using Mike's original RS-232 interface design and the port settings that he found. Without Mike's insights into the RS-232 data stream, I don't think that would have been able to get a foothold into this protocol.

So again, I figured this out just by looking at the RS-232 stream over the tether cable. No code disassembly, no logic analyzers, no opening up of Remotes or TIU's.

I made up a video (Screen Capture) of the operation of my RTC or Remote Train Control program. This video demonstrates part of its use:

`http://www.silogic.com/trains/RTC_Running.html`

- ## The Encoded Data

Mike had designed an interface between the PC and the TIU where he could sniff out the packets that the Remote sent to the TIU. I built up that interface and got to work.

Here are some of the packets from the Remote to the TIU. These were from the startup sequence:

```
5555595665AA656955665565A555A95A4D
555559566599656556655565559A669A4D
5555595665A6656656996965666A566655656556 9A654D
5555595665A6666A56996965666A55695565A5959A564D
5555595665A6556A556565AA59554D
5555595665A66566559A65666665566 95565A9595A554D
```

PC Control of MTH Engines by Serial Connection to the TIU

14. Sep. 2020

This was certainly encoded in some form or another. I know a little bit about this stuff so I started guessing

It looks like a DC balanced code. DC balanced codes have a place in data transmission over wire. A long wire is like a big capacitor. As you put a voltage on it, it starts to store that voltage. Not good for a wire carrying a data signal because eventually the data signal becomes swamped by the built up voltage (the DC component). This happens if you have more '1's than '0's or the opposite.

So what that means is that you want the signal to have the same number of '1's as '0's. It looked very much like that was what we had. Look at the digits in the encoded packets. There are:

hex	binary
5	0101
9	1001
A	1010
6	0110

Except for the odd '4D' at the end, the packet was made up entirely of these 4 characters. Notice that each of them had the same number of '1's and '0's.

I searched on the Internet and found a patent. Look at the patent.

> DC balanced 4B/8B binary block code for digital data communications
> United States Patent 5625644

Look at the 4 codedigits: 5 (0101), 6 (0110), 9 (1001) and A (1010) described in this patent. We have exactly these 4 digits in our codewords.

The requirements set out in the patent to pick the codedigits:

1. Two digits make 8 bits - select code words with 4 1's and 4 0's.
2. Code words with no more than two consecutive 1's or 0's.
3. Select code words with '01' or "10" at each end.

The patent shows control word "4D" for end of transmission which was a pretty good indication that I was on the right track since I saw the same thing. Read in the patent about why "4D" is still a good codeword even though it does not meet all of the requirements.

There are 16 combinations (codewords) of the four codedigits. So I thought that there would be a one to one mapping of the 16 combinations to the hex digits 0-F.

PC Control of MTH Engines by Serial Connection to the TIU

14. Sep. 2020

I was pretty sure that the mapping outlined in the DC Balanced Code patent was not the one used, so I picked the simplest mapping. This was a VERY lucky guess – as you will see below, I hit on the correct mapping on my first try. Use the four codedigits and put them in ascending value mapped to binary in ascending value:

8 bit (hex)	4 bit (binary)
55	0000
56	0001
59	0010
5A	0011
65	0100
66	0101
69	0110
6A	0111
95	1000
96	1001
99	1010
9A	1011
A5	1100
A6	1101
A9	1110
AA	1111

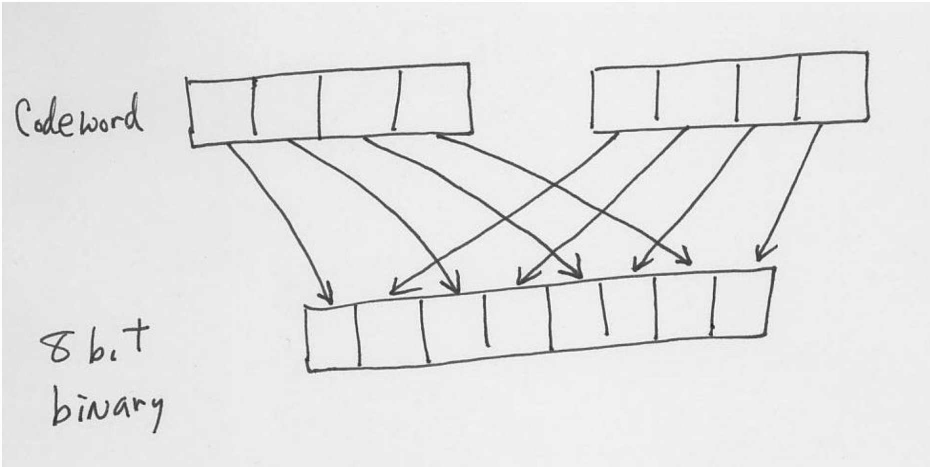
I looked at the Engine Number field as identified by Mike Hewett in his video. I sent the same command to several engines and looked at the serial data. Per Barry's book, engine 1 on the remote is called "DCS Engine" 2. I expected to see a binary 0000 0010 in there:

Engine Number	16 Serial Bits (in hex)	Mapping to 8 bit Binary
2	5655	0001 0000
3	5656	0001 0001
4	5559	0000 0010
5	555A	0000 0011
6	5659	0001 0010
15	5A5A	0011 0011
16	5565	0000 0100
17	5566	0000 0101

This looked interesting. But the result of the mapping didn't quite look like an engine number. I stared at it for a while (weeks!) and noticed that if I swapped around the bits in a certain way, I could get the binary value that I wanted.

PC Control of MTH Engines by Serial Connection to the TIU

14. Sep. 2020



Repeating the above table with a new column:

Engine Number	16 Serial Bits (in hex)	Mapping to 8 bit Binary	8 Bit Binary Swapped
2	5655	0001 0000	0000 0010
3	5656	0001 0001	0000 0011
4	5559	0000 0010	0000 0100
5	555A	0000 0011	0000 0101
6	5659	0001 0010	0000 0110
15	5A5A	0011 0011	0000 1111
16	5565	0000 0100	0001 0000
17	5566	0000 0101	0001 0001

What was this screwy bit swapping encoding? An Internet search led to these web pages:

Z-order Curve

and

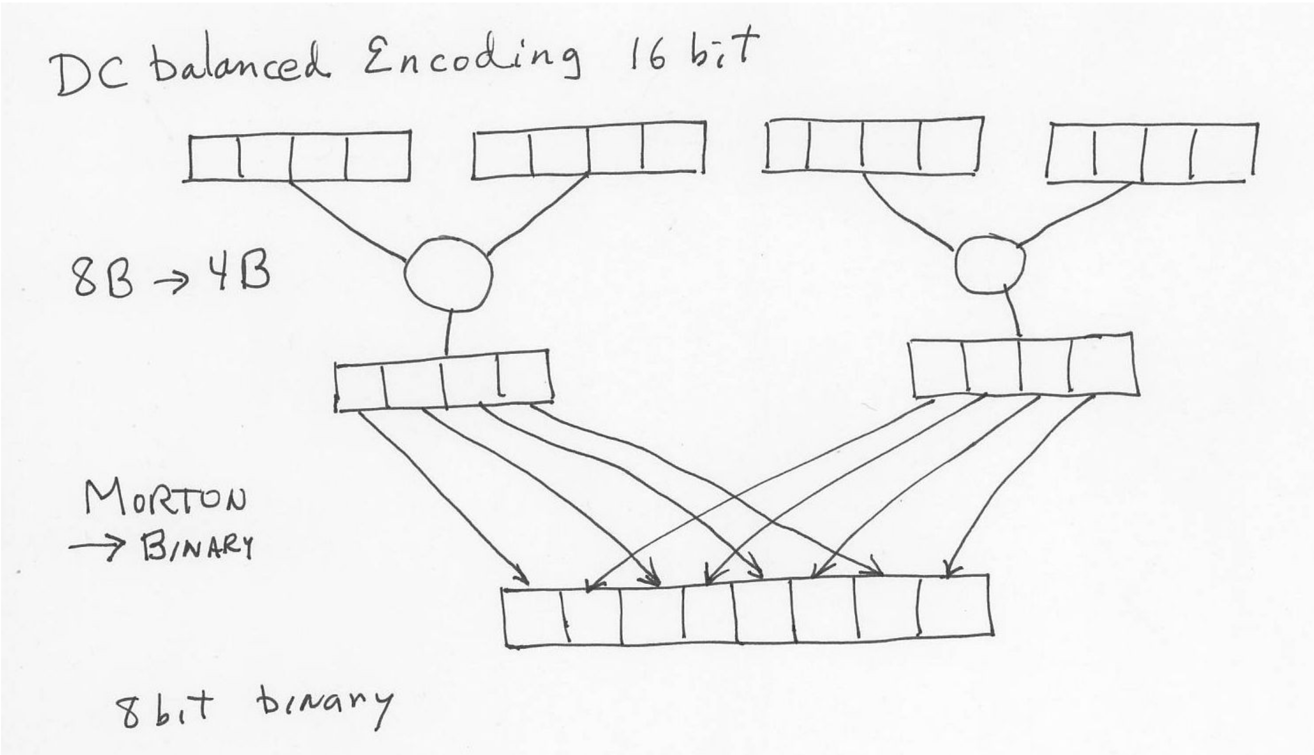
Decoding Morton Codes

As the articles describe, these codes are used to map a 2 dimensional space into a 1 dimensional space to quickly estimate the physical distance between two points. I can't even guess why this encoding into Morton Codes was used here.

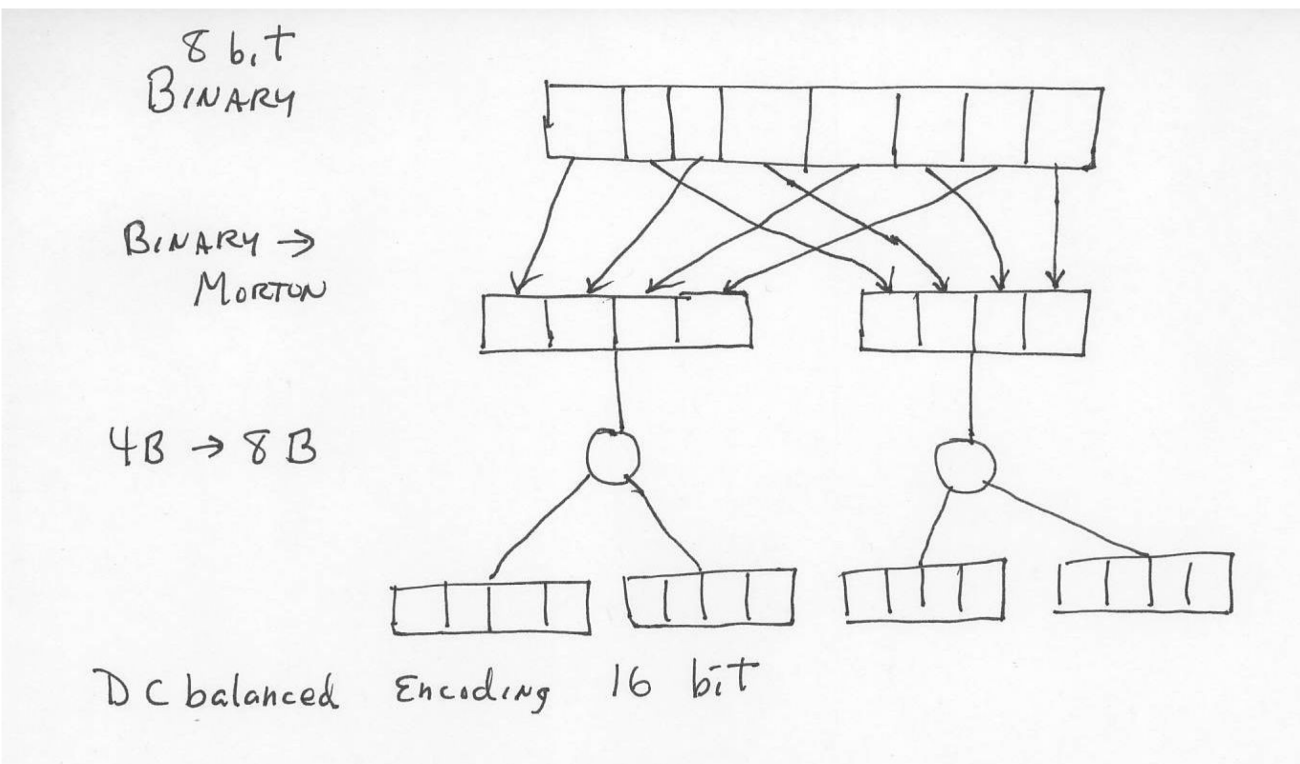
PC Control of MTH Engines by Serial Connection to the TIU

14. Sep. 2020

Here is a summary of the encoding from Encoded to 8 bit binary:



Then here is the reverse from 8 bit binary to Encoded:



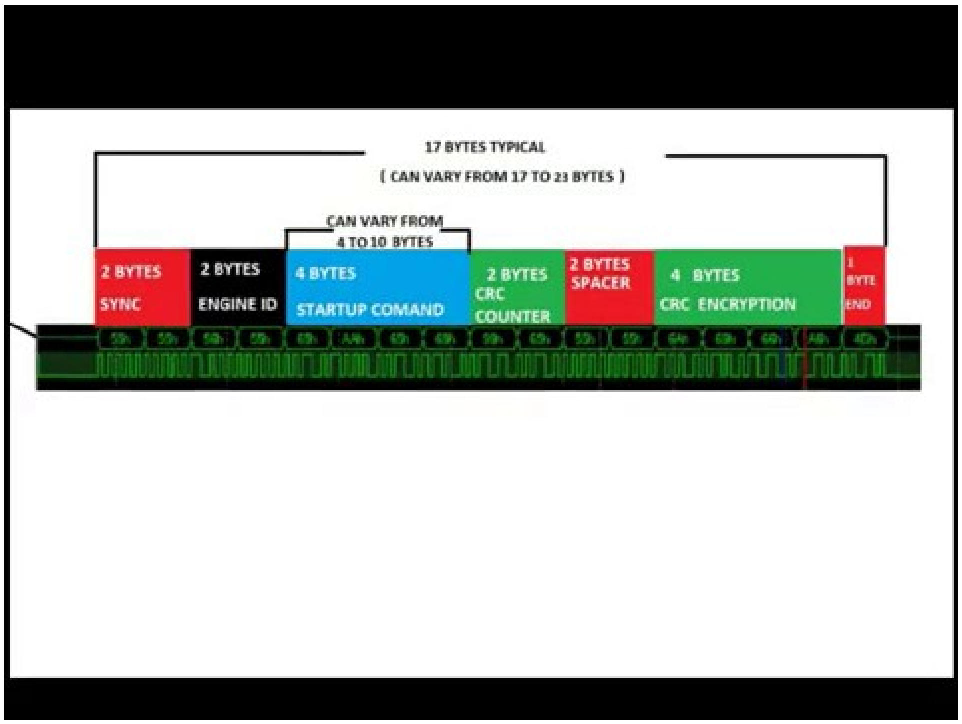
PC Control of MTH Engines by Serial Connection to the TIU

14. Sep. 2020

- ## Understanding the Decoded Packets

This seemed right. Could I verify it by checking another part of the command that I understood?

Mike Hewett identified the command portion of the serial stream (from his video):



I picked an engine and increased the speed by 1 Smph at a time and recorded the commands

So it started to make sense. Look at the commands below. The first codeword was 5555 – which I could convert to binary as 0000 0000. I could see where that could be the Sync byte also as identified by Mike. The next byte is the engine number in binary, here it is 5A55 which is DCS engine 10 (remote engine 9 which is the engine number that I was using).

The next codeword is the same for all of these. Maybe its a command. It is 66A6 which decodes to hex 0x73. It took me a few seconds to think “ASCII” and when I looked up that value it was an 's' character – maybe a “speed' command. Where the commands actually in ASCII?

The next codeword was 6566 or 0x31 – ASCII '1'. For 1 Smph. So I could see the speed increasing steadily up to ASCII '9'. At this point, the command length increased by 4 codedigits or one 8 bit binary decoded byte. I'd bet this is because the ASCII became '1' followed by '0' for 10 Smph. I confirmed this by looking at commands where the speed went from 99 to 100 Smph and I saw the same thing happen.

PC Control of MTH Engines by Serial Connection to the TIU

14. Sep. 2020

Speed 1 to 10 Smph

```
55555A5566A66566966A56666A9999564D
------------  first 12 codedigits are the same - Sync, Engine Number and
------------  something - which turned out to be 's'
            ---- “s1” command
55555A5566A6666599655666A6A695A94D
            ---- “s2” command
55555A5566A66666996656666956996654D
            ----
55555A5566A665699A65566659A96A664D
            ----
55555A5566A6656A9A6656666A6669AA4D
            ----
55555A5566A66669996956666AA9959A4D
            ----
55555A5566A6666A996A566659669656 4D
            ----
55555A5566A669659A6956665666A9994D
            ----
55555A5566A669669A6A566665A9AA554D
            ----
55555A5566A665666565A55556665A5A59964D
            --------  “s10” command
```

So I had figured out the format of the first part of the command.

I turned my attention to the byte that Mike identified as a “CRC Counter”. Well it was easy now, I could use my decoding algorithm on that byte. I saw that it was a counter but after looking at a series of commands, I could see that it was just a counter. It counted up to 0xFF and then restarted at 0x00. (I learned much latter that the TIU would ignore commands that duplicated a counter value that was just received. Probably caused by the remote resending a command that it thought the TIU had not received.)

On to the next codeword – the one identified by Mike as a “Spacer”. In the speed commands above, it is codeword 5666 or databyte 0x13. It seemed to never change as I looked at more commands. That is, never until I changed the remote number and later, the TIU number. This byte is made up of two pieces of information. The high order nybble is the TIU number (0 to 4 which corresponds to TIU 1 to 5), the low order nybble is the remote number (0 to 15).

So in my databyte of 0x13, I was using Remote 3 sending commands to TIU 2. In Mike's command example from his video, he saw codeword 5555 which decodes to 0x00 – TIU 1 and Remote 0 (the default values for those devices from the factory).

PC Control of MTH Engines by Serial Connection to the TIU

14. Sep. 2020

- ## The last step – the CRC

On to the last 2 codeword. Were they a CRC as identified by Mike?

I looked at hundreds of commands. These 2 codeword changed (what seemed like) randomly. A sure sign that they were CRC. I considered they might be a simple Checksum but one bit changes in the message caused complete perturbation of the these codeword as you would expect of a CRC and not of a Checksum.

I was pretty sure this was a CRC.

I spent a lot of time on this over several years. I'm crazy, sorry. I generated spreadsheets and wrote "C" language programs to generate and check CRC's. I tried dozens of CRC algorithms with no luck.

I found one document on the web by Ross Williams, titled "A Painless Guide to CRC Error Detection Algorithms"-- it wasn't.

But it helped in one way – the paper describes what the author named the RockSoft™ Model Algorithm for classifying CRC algorithms. Read that part because when I talk about the CRC below, it will use terms from that paper.

I found a useful utility named RevEng . This utility could calculate dozens of different CRC algorithms. For me, it confirmed over and over that I hadn't found the right CRC algorithm.

I found another great web page by Greg Ewing:

http://www.cosc.canterbury.ac.nz/greg.ewing/essays/CRC-Reverse-Engineering.html

Greg presented a lot of insights in his paper. I won't repeat what he wrote except to summarize and explain my results using his method. So follow along with his paper as I present my data.

He showed that CRC's "obey a kind of superposition principle. You can think of the CRC as being made up of the exclusive-or of a set of component CRCs, each of which depends on just one bit in the message.

He showed where, for the case of XorIn = XorOut = 0x0000, that:

C1 xor C2 = M1 xor M2
where: Cx = CRC Mx
Mx = equal length messages

He called "M1 xor M2" a difference message.

Greg wrote: "*Then I got into a conversation with Patrick Maupin, who suggested a test that might help to clarify whether it was a true CRC or not. Due to the superposition principle, if changing a message by XORing it with a bit pattern B1 causes its CRC to change by C1, and another bit pattern B2 causes the CRC to change by C2, then XORing the message with (B1 xor B2) should change the CRC by (C1 xor C2). If that doesn't happen, the algorithm can't be an ordinary CRC algorithm.*"

PC Control of MTH Engines by Serial Connection to the TIU

14. Sep. 2020

Using these 4 commands from the remote to TIU bit stream:

```
                                        CRC
                                     LSB  MSB
   ---- ---- ---- ---- ---- ---- ---- ----
M0 0x00 0x09 0x69 0x31 0x00 0x03 0xBA 0xDE
M1 0x00 0x09 0x69 0x31 0x01 0x03 0x01 0xC2
M2 0x00 0x09 0x69 0x31 0x02 0x03 0xCC 0xE7
M3 0x00 0x09 0x69 0x31 0x03 0x03 0x77 0xFB
   ---- ---- ---- ---- ---- ---- ---- ----
```

Then,

```
                                        CRC
                                     LSB  MSB
   ---- ---- ---- ---- ---- ---- ---- ----
M0 0x00 0x09 0x69 0x31 0x00 0x03 0xBA 0xDE
B1 0x00 0x00 0x00 0x00 0x01 0x00
   ------------ XOR ------------
M1 0x00 0x09 0x69 0x31 0x01 0x03 0x01 0xC2
```

C1 = 0xDEBA xor 0xC201 = 0x1CBB

and another bit pattern B2 causes the CRC to change by C2,

```
                                        CRC
                                     LSB  MSB
   ---- ---- ---- ---- ---- ---- ---- ----
M0 0x00 0x09 0x69 0x31 0x00 0x03 0xBA 0xDE
B2 0x00 0x00 0x00 0x00 0x02 0x00
   ------------ XOR ------------
M2 0x00 0x09 0x69 0x31 0x02 0x03 0xCC 0xE7
```

C2 = 0xDEBA xor 0xE7CC = 0x3976

C1 xor C2 = 0x1CBB xor 0x3976 = 0x25CD

then XORing the packet with B3 = (B1 xor B2) should change the CRC by C3 = (C1 xor C2). If that doesn't happen, the algorithm can't be an ordinary CRC algorithm.

```
                                        CRC
                                     LSB  MSB
   ---- ---- ---- ---- ---- ---- ---- ----
B1 0x00 0x00 0x00 0x00 0x01 0x00
B2 0x00 0x00 0x00 0x00 0x02 0x00
   ------------ XOR ------------
```

```
B3 0x00 0x00 0x00 0x00 0x03 0x00

M0 0x00 0x09 0x69 0x31 0x00 0x03 0xBA 0xDE
B3 0x00 0x00 0x00 0x00 0x03 0x00
   ------------ XOR ------------
M3 0x00 0x09 0x69 0x31 0x03 0x03 0x77 0xFB
```

C3 = 0xDEBA xor 0xFB77 = 0x25CD

Does C3 == 0x25CD == C1 XOR C2 == 0x25CD ? **Yes!**

In the case of this serial stream, it happens so it must be an ordinary CRC algorithm.

Greg wrote: "*Now consider two CRC values obtained from two 1-bit messages, where the 1 bits are in adjacent positions. The resulting CRCs will differ by just one shift-xor cycle. To be precise, if C1 corresponds to the message with a 1 in position i, and C2 corresponds to the message with a 1 in position i+1, then C1 is derived from applying one shift-xor cycle to C2. (If this seems backwards, it's because the further the 1 bit is from the end of the message, the more shift-xor cycles get applied to the CRC.)*

There are two possibilities. If the leading bit of C2 (the one about to be shifted out) is 0, then C1 will be equal to C2 shifted by one place. If it is 1, then C2 will be equal to C1 shifted one place and xored with the polynomial."

I constructed several difference messages that differed in only one bit using the data captured over the serial port:

```
                                       CRC
                                    LSB  MSB
    ---- ---- ---- ---- ---- ---- ---- ----
M0  0x00 0x09 0x69 0x31 0x00 0x03 0xBA 0xDE
M1  0x00 0x09 0x69 0x31 0x01 0x03 0x01 0xC2
    ---- ---- ---- ---- ---- ---- ---- ----
D01    0    0    0    0 0x01    0 0xBB 0x1C

    ---- ---- ---- ---- ---- ---- ---- ----
M0  0x00 0x09 0x69 0x31 0x00 0x03 0xBA 0xDE
M2  0x00 0x09 0x69 0x31 0x02 0x03 0xCC 0xE7
    ---- ---- ---- ---- ---- ---- ---- ----
D02    0    0    0    0 0x02    0 0x76 0x39

    ---- ---- ---- ---- ---- ---- ---- ----
M0  0x00 0x09 0x69 0x31 0x00 0x03 0xBA 0xDE
M3  0x00 0x09 0x69 0x31 0x04 0x03 0x56 0xAC
    ---- ---- ---- ---- ---- ---- ---- ----
D04    0    0    0    0 0x04    0 0xEC 0x72

    ---- ---- ---- ---- ---- ---- ---- ----
M0  0x00 0x09 0x69 0x31 0x00 0x03 0xBA 0xDE
```

```
m8   0x00 0x09 0x69 0x31 0x08 0x03 0x62 0x3B
     ---- ---- ---- ---- ---- ---- ---- ----
D08     0    0    0    0 0x08    0 0xD8 0xE5

     ---- ---- ---- ---- ---- ---- ---- ----
M0   0x00 0x09 0x69 0x31 0x00 0x03 0xBA 0xDE
M10  0x00 0x09 0x69 0x31 0x10 0x03 0x1B 0x1D
     ---- ---- ---- ---- ---- ---- ---- ----
D10     0    0    0    0 0x10    0 0xA1 0xC3

     ---- ---- ---- ---- ---- ---- ---- ----
M0   0x00 0x09 0x69 0x31 0x00 0x03 0xBA 0xDE
M20  0x00 0x09 0x69 0x31 0x20 0x03 0xE9 0x51
     ---- ---- ---- ---- ---- ---- ---- ----
D20     0    0    0    0 0x20    0 0x53 0x8f

     ---- ---- ---- ---- ---- ---- ---- ----
M0   0x00 0x09 0x69 0x31 0x00 0x03 0xBA 0xDE
M40  0x00 0x09 0x69 0x31 0x40 0x03 0x0D 0xC8
     ---- ---- ---- ---- ---- ---- ---- ----
D40     0    0    0    0 0x40    0 0xB7 0x16

     ---- ---- ---- ---- ---- ---- ---- ----
M0   0x00 0x09 0x69 0x31 0x00 0x03 0xBA 0xDE
M80  0x00 0x09 0x69 0x31 0x80 0x03 0xD4 0xF3
     ---- ---- ---- ---- ---- ---- ---- ----
D80     0    0    0    0 0x80    0 0x6e 0x2d
```

PC Control of MTH Engines by Serial Connection to the TIU

14. Sep. 2020

I came up with these CRC values for the difference messages with the byte shown:

```
Byte      CRC (with byte swap to MSB-LSB order)
-----     ----
01        1CBB
02        3976    LSB of CRC = 0
04        72EC    LSB of CRC = 0
08        E5D8    LSB of CRC = 0
10        C3A1    LSB of CRC = 1
20        8F53    LSB of CRC = 1
40        16b7    LSB of CRC = 1
80        2d6e    LSB of CRC = 0
```

Greg explained that when the LSB is a 0, the preceding CRC is one bit right shift of this CRC. If the LSB is a 1 then the preceding CRC is the current CRC XOR'ed with the polynomial and then right shifted.

```
Byte      CRC       shift     xor      Previous CRC
-----     ----      ----      ----     ----
01        1CBB
02        3976      1CBB      0000     1CBB
04        72EC      3976      0000     3976
08        E5D8      72EC      0000     72EC
10        C3A1      61D0      8408     E5D8
20        8F53      47A9      8408     C3A1
40        16b7      0B5B      8408     8F53
80        2d6e      16B7      0000     16b7
```

This told me that the polynomial was 0x8408.

The next sentence paraphrases what Greg wrote: "*The shifting direction indicates that the ReflectOut parameter should be True, since shifting to the right is equivalent to using the canonical left-shifting version of the algorithm with the polynomial 0x1021 and then reflecting the resulting CRC. It is notable that 0x1021 is one of the standard 16-bit polynomials -- the one that is called "CRC-16-CCITT"and also known as "KERMIT".*"

So Greg's paper has helped me get pretty far along with my analysis. It sure looks like its a CRC and its most likely CRC-16-CCITT or KERMIT (at least if init = XorOut = 0x0000).

But, of course, when I tried running my messages through RevEng and asking it to use the KERMIT algorithm, the actual CRC's from the messages did not match what RevEng generated.

For example, using D01 message:

```
d:>reveng -c -m kermit 000000000100 returns d819
```

Greg's analysis of his problem indicated that there might some bytes included in the CRC calculation that are not apparent. In Greg's case, he wrote : "*I came up with the following idea. Start by initialising* *the register with the polynomial -- this corresponds to the state just after encountering the 1 in a 1-bit difference message. Then run the algorithm and count the number of steps required before the known CRC value is reached. Assuming it was eventually reached, that would tell me how many 0 bits following the 1 were included in the CRC.*"

So, I said maybe this would work for me. So I added 00 bytes to the end of message D01:

```
d:>reveng -c -m kermit 00000000010000 returns dc5a
```

```
d:>reveng -c -m kermit 0000000001000000 returns bb1c
```

WOW, 0xbb1c is the CRC value that I get when I xor the two actual CRC's from M0 and M1. (I can add 0x00 bytes to the difference message if the actual bytes in the actual messages are the same and are 0x00 after they are xor'ed into the difference message. And I need to remember that this analysis still requires that init and XorOut are 0x0000 and that might not be case in the actual algorithm.

So I spent about a year (I am really crazy) trying to find the value of the extra bytes. I wrote a program which tried all combinations of extra bytes but I their values jumped all over. I'm going to leave out everything I did trying to find the extra bytes. It was a dead end for me.

< **Deleted here – a year of dead ends looking for 'extra bytes' as described by Greg.**>

But I finally figured it out! This was November of 2014. Follow on....

Maybe I still didn't have the right data bytes in the right position or maybe the assumption that init = XorOut = 0x0000 was not true.

So I decided to try more messages. I made up a bunch of messages where I could control the value of the 2nd byte (the Engine number). Here are the messages that I was able to generate:

									CRC	
									LSB	MSB
0x00	0x02	0x77	0x32				0x97	0x13	0x22	0x8D
0x00	0x04	0x77	0x32				0x97	0x13	0x14	0xE8
0x00	0x08	0x77	0x32				0x97	0x13	0x78	0x22
0x00	0x10	0x77	0x32				0x97	0x13	0xB1	0xBE
0x00	0x20	0x77	0x32				0x97	0x13	0x32	0x8F
0x00	0x40	0x77	0x32				0x97	0x13	0x34	0xEC
0x00	0x06	0x77	0x32				0x97	0x13	0x06	0xCB
0x00	0x0A	0x77	0x32				0x97	0x13	0x6A	0x01
0x00	0x12	0x77	0x32				0x97	0x13	0xA3	0x9D
0x00	0x22	0x77	0x32				0x97	0x13	0x20	0xAC
0x00	0x42	0x77	0x32				0x97	0x13	0x26	0xCF

PC Control of MTH Engines by Serial Connection to the TIU

14. Sep. 2020

And here are the difference messages:

```
                                                     CRC
                                                   LSB  MSB
     ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ----
M02a 0x00 0x02 0x77 0x32                0x97 0x13 0x22 0x8D
M03a 0x00 0x03 0x77 0x32                0x97 0x13 0xAB 0x9C
     ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ----
D01a 0x00 0x01 0x00 0x00                0x00 0x00 0x89 0x11

                                                 LSB  MSB
   ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ----
   0x00 0x04 0x77 0x32                0x97 0x13 0x14 0xE8
   0x00 0x06 0x77 0x32                0x97 0x13 0x06 0xCB
   ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ----
   0x00 0x02 0x00 0x00                0x00 0x00 0x12 0x23

                                                 LSB  MSB
   ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ----
   0x00 0x02 0x77 0x32                0x97 0x13 0x22 0x8D
   0x00 0x06 0x77 0x32                0x97 0x13 0x06 0xCB
   ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ----
   0x00 0x04 0x00 0x00                0x00 0x00 0x24 0x46

                                                 LSB  MSB
   ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ----
   0x00 0x02 0x77 0x32                0x97 0x13 0x22 0x8D
   0x00 0x0A 0x77 0x32                0x97 0x13 0x6A 0x01
   ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ----
   0x00 0x08 0x00 0x00                0x00 0x00 0x48 0x8C

                                                 LSB  MSB
   ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ----
   0x00 0x02 0x77 0x32                0x97 0x13 0x22 0x8D
   0x00 0x12 0x77 0x32                0x97 0x13 0xA3 0x9D
   ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ----
   0x00 0x10 0x00 0x00                0x00 0x00 0x81 0x10

                                                 LSB  MSB
   ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ----
   0x00 0x02 0x77 0x32                0x97 0x13 0x22 0x8D
   0x00 0x22 0x77 0x32                0x97 0x13 0x20 0xAC
   ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ----
   0x00 0x20 0x00 0x00                0x00 0x00 0x02 0x21

                                                 LSB  MSB
   ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ----
   0x00 0x02 0x77 0x32                0x97 0x13 0x22 0x8D
   0x00 0x42 0x77 0x32                0x97 0x13 0x26 0xCF
   ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ----
```

```
   0x00 0x40 0x00 0x00                  0x00 0x00 0x04 0x42
```

Then I did the same analysis on the CRC words and came up with the same polynomial as before: 0x8408 or when shifted the other direction, 0x1021

```
Byte     CRC (with byte swap to MSB-LSB order)
-----    ----
0x01     1189
0x02     2312    LSB of CRC = 0
0x04     4624    LSB of CRC = 0
0x08     8C48    LSB of CRC = 0
0x10     1081    LSB of CRC = 1
0x20     2102    LSB of CRC = 0
0x40     4204    LSB of CRC = 0
```

```
Byte     CRC    shift     XOR    Previous CRC
-----    ----    ----    ----    ----
0x01     1189
0x02     2312    1189    0000    1189
0x04     4624    2312    0000    2312
0x08     8C48    4624    0000    4624
0x10     1081    0840    8408    8C48
0x20     2102    1081    0000    1081
0x40     4204    2102    0000    2102
```

Here are all of the RockSoft™ CRC models with this poly, maybe I needed to think past KERMIT:

width=16 poly=0x1021 init=0x0000 refin=false refout=false xorout=0x0000 name="XMODEM"
width=16 poly=0x1021 init=0x0000 refin=true refout=true xorout=0x0000 name="KERMIT"
width=16 poly=0x1021 init=0x1d0f refin=false refout=false xorout=0x0000 name="CRC-16/AUG-CCITT"
width=16 poly=0x1021 init=0x89ec refin=true refout=true xorout=0x0000 name="CRC-16/TMS37157"
width=16 poly=0x1021 init=0xb2aa refin=true refout=true xorout=0x0000 name="CRC-16/RIELLO"
width=16 poly=0x1021 init=0xc6c6 refin=true refout=true xorout=0x0000 name="CRC-A"
width=16 poly=0x1021 init=0xffff refin=false refout=false xorout=0x0000 name="CRC-16/CCITT-FALSE"
width=16 poly=0x1021 init=0xffff refin=false refout=false xorout=0xffff name="CRC-16/GENIBUS"
width=16 poly=0x1021 init=0xffff refin=true refout=true xorout=0x0000 name="CRC-16/MCRF4XX"
width=16 poly=0x1021 init=0xffff refin=true refout=true xorout=0xffff name="X-25"

I noticed something with this newest set of messages. The two extra bytes were not needed to get the correct CRC for the difference messages if I just used the first byte and left off the other 0x00 bytes. Using RevEng on message D01a:

```
D:>reveng -s -w 16 018911 width=16 poly=0x1021 init=0x0000 refin=true
refout=true xorout=0x0000 check=0x2189 name="KERMIT"
```

Then I realized that if I feed the original D01 message into the CRC generator in <u>reverse byte order</u>, my original D01 message <u>generated the correct CRC</u>.

```
D:>reveng -c -m kermit 01000000 bb1c
```

PC Control of MTH Engines by Serial Connection to the TIU

14. Sep. 2020

So these bytes that I thought were extra bytes were really just bytes of the message but the message was to be processed in reverse byte order.

Could this be true?

I took M02a and M03a, and fed them into RevEng in reverse byte order:

```
D:>reveng -s -w 16 1397327702228d

width=16 poly=0x1021 init=0xffff refin=true refout=true xorout=0x0000
check=0x6f91 name="CRC-16/MCRF4XX"

D:>reveng -s -w 16 1397327703ab9c

width=16 poly=0x1021 init=0xffff refin=true refout=true xorout=0x0000
check=0x6f91 name="CRC-16/MCRF4XX"
```

And what fell out amazed me, init was all ones and the algorithm had the same poly that I previously found. The algorithm was from MCRF4xx communications (which now I see makes sense because the RS-232 line that I'm tracing replaced an RF connection).

MCRF4xx is an RFID protocol. Google it.

The message is taken as one long set of bits so that the last byte is processed first and the first byte is processed last. The 0x00 (sync) byte at the beginning of message is not used. With init = 0xFFFF, the MCRF4xx algorithm will detect extra or missing leading zeros.

I tried this algorithm on a couple dozen of the messages and it gets the correct CRC every time.

Here is a snipit of code that I found at: https://gist.github.com/aurelj/270bb8af82f65fa645c1 originally posted by “aurelj”:

```
#include <stdint.h>
#include <stddef.h>

uint16_t crc16_mcrf4xx(uint16_t crc, uint8_t *data, size_t len)
    {
    if (!data || len > 1) ^ 0x8408;
            else crc = (crc >> 1);
            }
        }
    return crc;
    }
```

Disclamer – I've not used the above code.

A web page on the RevEng site, points to some documents about MCRF4xx: http://reveng.sourceforge.net/crc-catalogue/16.htm.

I don't use the code snippet shown above but rather in my program, I use C code developed by Ross Williams:

```
/******************************************************************************/
/*                                                                            */
/* Author : Ross Williams (ross@guest.adelaide.edu.au.).                      */
/* Date   : 3 June 1993.                                                      */
/* Status : Public domain.                                                    */
/*                                                                            */
/* Description : This is the implementation (.c) file for the reference       */
/* implementation of the Rocksoft™ Model CRC Algorithm. For more              */
/* information on the Rocksoft™ Model CRC Algorithm, see the document         */
/* titled "A Painless Guide to CRC Error Detection Algorithms" by Ross        */
/* Williams (ross@guest.adelaide.edu.au.). This document is likely to be in   */
/* "ftp.adelaide.edu.au/pub/rocksoft".                                        */
/*                                                                            */
/* Note: Rocksoft is a trademark of Rocksoft Pty Ltd, Adelaide, Australia.    */
/*                                                                            */
/******************************************************************************/
```

I've sent emails to both Ross and Greg thanking them for their help.

- ## The Interface

Mike's interface between the Remote and TIU allowed me to monitor the packets sent between the Remote and TIU and to send packets to the TIU. You can see Mike's design in his three videos.

I designed a different interface that let me send packets to the TIU and read up the response from the TIU.

You can see the complete design and variations by some of my alpha testers on my web page:

http://www.silogic.com/trains/RTC_Running.html

- ## Responses from the TIU

Once I could control trains, I started to look at the responses from the TIU. The responses used the same 4B-8B DC balanced code with the intermediate Morton Numbers. That meant that I could understand the bytes returned.

The overall format was very different from the packets sent to the TIU. Seems odd but maybe it prevents packets being mistaken for each other.

The first response from the TIU are these 3 bytes which is the TIU saying that it got the command (I call this the “OK” response):

```
55554D
  00
```

This might be a sync byte and an End of Transmission character.

PC Control of MTH Engines by Serial Connection to the TIU

14. Sep. 2020

For most commands, the TIU sends 8 encoded bytes or 4 binary bytes:

```
6696A5A65A5A5565
  63  F1  0F  10
```

It seems that the TIU->Remote channel protocol is not as “nice” as the other way. You have to know how many characters are returned by each command to know if you got them all. For example, most commands, as shown above, return 4 bytes.

For the 4 byte commands (which I call “ACK” or “NACK”):

1-2. the first two bytes are the CRC. Same encoding - MCRF4XX.
3. Next byte is the TIU/Remote byte.
    For TIU V4.30 : Same encoding as the Remote to TIU command. Here TIU 1 and Remote 15.
    For TIU V5.00 : The TIU number is returned as zero sometimes.
4. Some kind of acknowledgment byte. A “00” here might be an indication that the command failed. 0x10, 0x11 and 0x12 indicate a success. TIU V5.0 seems to have made this more consistent, 0x1X indicates success with the second nybble being the TIU number that sent the response.

Some commands return longer responses. Here is a response from the 'x' command which returns 6 bytes:

```
A95555555A5A555555565555
  A8  00  0F  00  01  00
```

1-2. First two bytes are CRC
3. TIU/Remote
4. 0x00 more bytes coming
5. Byte Count of following data
6. One byte of data, in this case : Number of AIU, in binary

Here is a response to the 'q' command which returns 9 bytes:

```
9559A5A555655555555955555555A6599565
  84  F0  10  00  04  00  00  A6  90
59A5595A566655555559555555555AA6A9AA
  58  0D  13  00  04  00  00  5B  FD
```

1-2. First two bytes are CRC
3. TIU/Remote (here its TIU 2 and Remote 0)
4. 0x00 more bytes coming
5. Byte Count of following data
6-9. 32 bits of data – (not to be discussed here but look at the web page which describes how the ‘q’ command is used to read out the RAM in the engine)

If the 'q' command fails, you only get 4 bytes back with a 0x00 error indication in byte 4.

PC Control of MTH Engines by Serial Connection to the TIU

14. Sep. 2020

Maybe the response from the TIU is reversed from that sent by the Remote.

Command from Remote to TIU:

1. sync 0x5555 - always needed at the front
2. engine #
3. command
4. data bytes 0-X
5. sequence #
6. TIU/Remote
7. Two CRC bytes
8. 0x4D – EOT byte

The response from TIU to Remote:

1. sync 0x5555 - always needed at the front
2. 0x4D – EOT byte
3. Two CRC bytes
4. TIU/Remote
5. no sequence # - instead its some kind of command status flag
   0x00 command failed ? – maybe engine not on track – NACK
   0x01 with v5.00, this seems to be engine not on track
   0x1X for command accepted and acted on - ACK
       X = TIU Number sending the response (0-4) (not yet sure about this)
   0x00 indicating data byte count and data bytes follow for commands that return data.
6. data byte count
7. data bytes 0-X

The bytes have to be swapped around in the typical unusual manner for the MCRF4XX CRC:

```
55554D            -- these do not participate in the CRC
A9A99595566655555559555555556555955
  FC  C0  13  00  04  00  00  02  08

c:\> reveng -c -m CRC-16/MCRF4XX 13000400000208c0fc

reveng returns 0000 indicating a good CRC

==============================================================================
5555555A69A5AA5A5A5A6596996A4D
  00  05  78  AF  0F  61  9D
       'x'         'a'
55554D
  00
A95555555A5A555555565555
  A8  00  0F  00  01  00      -- this returns the number of AIU, 01 is byte count
-------------------------------------
5555555A6556A5655A5A99A956A94D
  00  05  21  B0  0F  DC  56
       '!'             'V'
```

```
55554D
  00
66A599695A5A5555555965656566966665555
  72  9C  0F  00  04  30  34  33  00-- four bytes of data returned  30 34 33 00
 'r'                 '0' '4' '3'    -- ASCII for TIU Version number (here its 4.3 )
-------------------------------------

55555A55559965A69A656656A9996A564D
  00  0A  44  71  9A  23  EC  2B
         'D' 'q'     '#'     '+'
55554D
  00

A9695A6A665655555655555595655
  BC  1F  23  00  02  04  02        -- two bytes returned  0x04 and 0x02
         '#'
-------------------------------------
With no engines on track, we get a "00" response

5555555A65AA6569965555559A5959654D
  00  05  75  34  82  00  8E  18
         'u' '4'
55554D
  00
A5A5A96555555555
  F0  B8  00  00        -- last "00" is failure response
                        -- >reveng -c -m CRC-16/MCRF4XX 0000B8F0  - returns 0000
-------------------------------------
55555A5566A66565AA66566695AA9A964D
  00  0A  73  30  BB  13  D5  CB
         's' '0'
55554D
  00
9AA699A556565566
  DB  D8  03  11        -- example of "11" response
-------------------------------------
55555A556599656596565555AA9556664D
  00  0A  64  30  83  00  EA  13
         'd' '0'
55554D
  00
A595696655555565
  E0  39  00  10        -- example of "10" response
                        -- reveng -c -m CRC-16/MCRF4XX 001039e0  - returns 0000
-------------------------------------
```

I've made some progress in understanding the responses from the TIU.

PC Control of MTH Engines by Serial Connection to the TIU
14. Sep. 2020

Here are some responses from the TIU:

```
---------------------
Default: This response comes back from commands that don't return any data. Every response
(not just this one) begins with 55554D.

55554D
  00
        ↓↓↓↓ TIU/Remote
A595696655555565  -     A success response
  E0  39  00  10
---------------------
"q"   - query command

55554D
  00

595666655555555555595555555555555A
  09  32  00  00  04  00  00  00  05

byte count is 4, the four bytes of data make a 32 bit word  0x00000005
---------------------
"q"   - query command

55554D
  00
        ↓↓↓↓ TIU/Remote
AA696956556555555595555656599A6596A59
  BE  29  10  00  04  00  64  A6  2E

byte count is 4, data is 0x0064A62E
---------------------
"I0"  - Interrogate Engines

55554D
  00

6656AA9A55555555595A555555555555555555555555555555555555555555555556
  23  EF  00  00  0D  00  00  00  00  00  00  00  00  00  00  00  00  01

byte count is 13 (0x0D), the 13 data bytes indicate which engines are powered on.
---------------------
"I1"  -     Interrogate Engine 0

55554D
  00

A5A5A96555555555
  F0  B8  00  00        -- I think this is a failure response
---------------------
"x"   -     Number of AIU command

55554D
  00
        ↓↓↓↓ TIU/Remote
66699AA56A5A555555565556
  36  DA  2F  00  01  01      -     one AIU

byte count is 1, data byte is 01 for one AIU present
```

```
---------------------
"x"   -     Number of AIU command

55554D
  00
        ↓↓↓↓ TIU/Remote
6A96A5565A6A555555565555
  6B  A1  1F  00  01  00      -     no AIU
---------------------
"!"   -     TIU Version Number command

55554D
  00
        ↓↓↓↓ TIU/Remote
55A569556A5A555555596565656A65656565
  50  28  2F  00  04  30  35  30  30      - version 5.00

byte count is 04, four data bytes are the version number in ASCII
---------------------
"!"   -     TIU Version Number command

55554D
  00
        ↓↓↓↓ TIU/Remote
969A55AA5A6A555555596565656966665555
  C7  55  1F  00  04  30  34  33  00      - version 4.30

byte count is 04, four data bytes are the version number in ASCII
---------------------
"m4"  -     Startup Command

55554D
  00
        ↓↓↓↓ TIU/Remote
5595A6965A5A5665
  40  E3  0F  12  - response from TIU 3

response is an ACK
---------------------
"m4"  -     Startup Command

55554D
  00
        ↓↓↓↓ TIU/Remote
A5A6A56555555566
  F1  B0  00  11  - response from TIU 2
```

response is an ACK

PC Control of MTH Engines by Serial Connection to the TIU

14. Sep. 2020

```
"I5"  -     Interrogate Engine 5, I generally call this the "Ixxx" command.

55554D
  00

999959565555555556A5555A5596665956A6655556A656AA65666665656565656555666566566666565656965AA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA5555555555555555555555555555555555555555556556595AA6A55
95555555555555555555555559A99696569A555555A5AA9A955555555555555555555
  CC  09  00  00  52  05  41  26  53  20  53  57  31  32  30  30  20  23  31  32  30  38
FF  FF  FF  FF  FF  FF  FF  FF  FF  FF  FF  FF  FF  FF  FF  FF  FF  FF  FF  FF  FF  FF  FF
FF  FF  FF  FF  FF  FF  FF  FF  FF  00  00  00  00  00  00  00  00  00  00  02  60  BF  40
00  00  00  00  00  00  CE  38  78  00  0F  FC  00  00  00  00  00
```

The byte count is 82 (0x52), the data bytes present a 16 byte ASCII engine name followed by 32 bytes of 0xFF, followed by 26 bytes of hotkey information follow by 5 bytes of unknown information. See spreadsheet "Command ('Ixxx') Response.xls" for details.

PC Control of MTH Engines by Serial Connection to the TIU

14. Sep. 2020

- ## The Command List

I used Mike's interface to enable me to look at packets that the Remote sent to the TIU. I started pushing buttons on the Remote and looking at the ASCII commands generated.

Since I knew the commands were ASCII, I could figure out the meaning of most of the commands and their responses.

I've made up names for each command, listed in column one of the table. That listing appears in another document:

http://www.silogic.com/trains/RTC/Remote%20to%20TIU%20Command%20Set.pdf

- ## LashUp Commands DCS versions 2-3-4

The LashUp commands use a non-ASCII format. I will explain a little more about the LashUp commands.

1. In LashUp command packets, the DCS engine# is **always** 0x66 (DCS #102).

2. A LashUp string is first created which lists all of the engines in the LashUp. This is a binary string (and thus different from other commands which are all ASCII). The string contains up to 10 binary numbers which are the DCS engine numbers. The first number is the head engine, the last number is the tail engine, any in between numbers are middle engines. For example, for a two engine LashUp consisting of engine 4 and engine 10, the string would be:

0x05 0x0B	Using DCS #5 and #11 (Engines #4 and #10).

If the engine is run in reverse in the LashUp, turn on the high order bit in the engine number:

0x05 0x8B	Using DCS #5 and #11 with engine #10 running in reverse.

There can be up to 8 middle engines. For example if we add engine #7 as a middle engine, running in reverse, the string would be:

0x05 0x88 0x8B	Using DCS #5, #8 and #11 with engines #7 and #10 running in reverse.

Then add a 0xFF byte to the end of the string to indicate the end of the LashUp:

0x05 0x88 0x8B 0xFF

3. Once you create this string, you need to create the LashUp. Do this with the “U” command so that the command part of the packet is:

“U” 0x05 0x88 0x8B 0xFF

To this, prepend the sync byte and the LashUp DCS #102 (0x66) and append the sequence number, the TIU/Remote byte, the CRC, and the EOT byte:

0x00 0x66 “U” 0x05 0x88 0x8B 0xFF “sequence#” “TIU/Remote#” “CRC Hi” “CRC Lo” 0x4D

Here is an actual example of a “U” command as created by the Remote. Then this can be sent to the TIU to create the LashUp:

```
5555669955AA555A5A55AAAA959955556699 9AAA4D
  00  66  55  05  0A  FF  C4  00  66  DF          Lashup of Engines 4 and 9
          'U'
```

This example shows:

1. Sync byte of 00
2. DCS Engine # of 0x66 or 102
3. the ‘U’ command
4. the two engines in the LashUp, DCS #5 and #10 (Engine numbers #4 and #9).
5. the end of LashUp indicator 0xFF
6. 0xC4 was the sequence number of this packet
7. 0x00 which is the TIU#/Remote# (TIU #1 and Remote #0)
8. 0x66 and 0xDF – the calculated CRC

4. CRC is calculated the same way as any other command. The CRC covers the bytes of the command packet from the DCS Engine #102 (0x66) through the TIU#/Remote# byte. Don't forget – feed the command to the CRC algorithm in reverse byte order.

5. Once the LashUp is created, you can use it with any valid command. There is a slight difference here from the ‘U’ command in that a “,” (a comma) or 0x2C needs to be inserted between the command letters and the LashUp string. So for example, a startup command to the LashUp created above would be:

```
5555669965AA65696959555A5A55AAAA959A555565A696564D
  00  66  75  34  2C  05  0A  FF  C5  00  71  83
         'u' '4' ','                            startup LashUp 4 and 9
```

This example shows:

1. Sync byte of 00
2. DCS Engine # of 0x66 or 102
3. the ‘u4’ command
4. 0x2C – a comma ‘,’
5. the two engines in the LashUp, DCS #5 and #10 (Engine numbers #4 and #9).
6. the end of LashUp indicator 0xFF
7. 0xC5 was the sequence number of this packet
8. 0x00 which is the TIU#/Remote# (TIU #1 and Remote #0)
9. 0x71 and 0x83 – the calculated CRC

All of the commands to LashUps follow this format, including the ‘u5’ shutdown. You only have to send one command to the TIU. The TIU will retransmit it automatically to all of the engines in the

LashUp (up to 10 Engines). The TIU automatically handles the head/middle/tail engines. For example, if you blow the whistle, only the whistle of the head engine will actually sound.

For another example, here is a Direction command:

```
55556699659965666959555A5A55AAAA9A955555666A965A4D
  00  66  64  31  2C  05  0A  FF  CA  00  37  87
         'd' '1' ','                             direction LashUp 4 and 9
```

6. To break up a LashUp, send the "m4" command to each engine in the normal way, that is, one engine at a time. You may need to send a Feature Reset "F0" command to each engine as well but I've not determined if that is needed.

- LashUp Commands DCS version5 and later

The LashUp commands also use an ASCII format. I will explain a little more about the LashUp commands. DCS Version 5.0 effected a major change in the format for LashUp Commands. The engine numbers, previously transmitted as binary are now converted to ASCII for transmission. The RTC program uses the version number returned by the TIU to use either the old or new format.

1. In LashUp command packets, the DCS engine# is **always** 0x66 (DCS #102).

2. A LashUp string is first created which lists all of the engines in the LashUp. This is a ASCII string in hexidecimal. The string contains up to 10 two digit hexidecimal numbers which are the DCS engine numbers. The first two digit number is the head engine, the last two digit number is the tail engine, any in between two digit numbers are middle engines. Engine numbers are always represented in base 16 (hexidecimal) converted to two ASCII characters.

Engine #16 or DCS #17 would be represented by 0x11 which is 17 in hexidecimal.

For example, for a two engine LashUp consisting of engine 4 and engine 10, the string would be:

0511 (0x30 0x35 0x31 0x31) Using DCS #5 and #17 (Engines #4 and #16).

If the engine is run in reverse in the LashUp, turn on the high order bit in the engine number. That is, for example, engine #16 would be DCS 17, in hex that is 0x11 which would become 0x91 which is then represented in ASCII as 0x39 0x31:

0591 (0x30 0x35 0x39 0x31) Using DCS #5 and #17 with engine #17 running in reverse.

There can be up to 8 middle engines. For example if we add engine #7 as a middle engine, running in reverse, the string would be:

058891 (0x30 0x35 0x38 0x38 0x39 0x31) Using DCS #5, #8 and #17 with engines #7 and #17 running in reverse.

Then add a 0xFF byte to the end of the string to indicate the end of the LashUp:

0x30 0x35 0x38 0x38 0x39 0x31 0xFF

3. Once you create this string, you need to create the LashUp. Do this with the “U” command so that the command part of the packet is:

“U” 0x30 0x35 0x38 0x38 0x91 0x31 0xFF

To this, prepend the sync byte and the LashUp DCS #102 (0x66) and append the sequence number, the TIU/Remote byte, the CRC, and the EOT byte:

0x00 0x66 “U” 0x30 0x35 0x38 0x38 0x91 0x31 0xFF “sequence#” “TIU#/Remote#” “CRC Hi” “CRC Lo” 0x4D

Here is an actual example of a “U” command as created by the Remote. Then this can be sent to the TIU to create the LashUp:

```
5555669955AA555A5A55AAAA9599555566999AAA4D
  00  66  55  05  0A  FF  C4  00  66  DF          Lashup of Engines 4 and 9
          'U'
```

This example shows:

1. Sync byte of 00
2. DCS Engine # of 0x66 or 102
3. the ‘U’ command
4. the two engines in the LashUp, DCS #5 and #10 (Engine numbers #4 and #9).
5. the end of LashUp indicator 0xFF
6. 0xC4 was the sequence number of this packet
7. 0x00 which is the TIU#/Remote# (TIU #1 and Remote #0)
8. 0x66 and 0xDF – the calculated CRC

4. CRC is calculated the same way as any other command. The CRC covers the bytes of the command packet from the DCS Engine #102 (0x66) through the TIU#/Remote# byte. Don't forget – feed the command to the CRC algorithm in reverse byte order.

5. Once the LashUp is created, you can use it with any valid command. There is a slight difference here from the ‘U’ command in that a “,” (a comma) or 0x2C needs to be inserted between the command letters and the LashUp string. So for example, a play sound command to a LashUp of engines #1 and #15 would be:

```
555566996A99656966666959656566656566656AAAA95A55A6AAAAA69594D
  00  66  6E  34  33  2C  30  32  31  30  FF  D0  1F  FF  2C
         'n' '4' '3' ',' '0' '2' '1' '0'
                                                 startup LashUp 1 and 15
```

This example shows:

1. Sync byte of 00
2. DCS Engine # of 0x66 or 102
3. the ‘u4’ command

4. 0x2C – a comma ‘,’
5.the two engines in the LashUp, DCS #2 and #16 (Engine numbers #1 and #15).
6. the end of LashUp indicator 0xFF
7. 0xD0 was the sequence number of this packet
8. 0x1F which is the TIU#/Remote# (TIU #2 and Remote #15)
9. 0xFF and 0x2C – the calculated CRC

All of the commands to LashUps follow this format, including the ‘u5’ shutdown. You only have to send one command to the TIU. The TIU will retransmit it automatically to all of the engines in the LashUp (up to 10 Engines). The TIU automatically handles the head/middle/tail engines. For example, if you blow the whistle, only the whistle of the head engine will actually sound.

6. To break up a LashUp, send the “m4” command to each engine in the normal way, that is, one engine at a time. You may need to send a Feature Reset “F0” command to each engine as well but I've not determined if that is needed.

- ## Super TIU Mode

It appears that Super TIU Mode is signaled by turning on the high order bit of the TIU nybble in the command. The actual TIU number remains in the 3 low order bits of the nybble. This is my best guess right now. I don't have a layout that can make use of this so I have not been able to seriously test it. My implementation includes all TIU into Super Mode. Feedback appreciated.

- ## **Proof of Concept Program**

I took all of what I learned and I wrote a program to control my trains from a PC. I've named the program RTC for Remote Train Control. This program is written using a very old version of Borland C++ Builder. I used this because I had experience with it back when I was working as a programmer. C++ Builder is not really good at handling an asynchronous system as we see in RS-232 packets coming back from TIU. The program occasionally crashes. But it does show what can be done.

You can see a video of my program and download the latest version on my web page at:

`http://www.silogic.com/trains/RTC_Running.html`

If you want to try your hand at enhancing my RTC program, I will send you the source code (or maybe you are just interested in looking at it). I’ve ported the program to used the latest version of what is now called the Embarcadero C++ v10.1. This version runs only on Windows 7,8 and 10. Other component libraries that cost money are also required.

The RTC program source code is released under the terms of the GNU General Public License version 3 or newer.

RTC requires either a wired or radio interface to the TIU. My original work was done with a wired interface based on a design from Mike Hewett. After that, I developed a radio interface. You need to build one or the other – I suggest the radio.

PC Control of MTH Engines by Serial Connection to the TIU

14. Sep. 2020

- ## The Wired Interface

The wired interface uses a simple interface circuit and telephone cables to connect the TIU to the PC. Don't use this, the radio works much better. Several versions of this interface are shown on my RTC program web page:

`http://www.silogic.com/trains/RTC_Running.html`

- ## The Radio Interface

The wired interface was a less than optimal solution. I knew that developing a radio interface would be much more difficult and it was. I finally succeeded.

Look on my web page for a write up of what I did for a radio interface.

`http://www.silogic.com/trains/OOK_Radio_Support.html`

● Translation Table – Databyte to Codeword

Databyte Decimal	Databyte Hex	Encoded Binary (Bitswap)	Encoded Hex	Codeword
0	0x00	00000000	0x00	5555
1	0x01	00000001	0x01	5556
2	0x02	00010000	0x10	5655
3	0x03	00010001	0x11	5656
4	0x04	00000010	0x02	5559
5	0x05	00000011	0x03	555A
6	0x06	00010010	0x12	5659
7	0x07	00010011	0x13	565A
8	0x08	00100000	0x20	5955
9	0x09	00100001	0x21	5956
10	0x0A	00110000	0x30	5A55
11	0x0B	00110001	0x31	5A56
12	0x0C	00100010	0x22	5959
13	0x0D	00100011	0x23	595A
14	0x0E	00110010	0x32	5A59
15	0x0F	00110011	0x33	5A5A
16	0x10	00000100	0x04	5565
17	0x11	00000101	0x05	5566
18	0x12	00010100	0x14	5665
19	0x13	00010101	0x15	5666
20	0x14	00000110	0x06	5569
21	0x15	00000111	0x07	556A
22	0x16	00010110	0x16	5669
23	0x17	00010111	0x17	566A
24	0x18	00100100	0x24	5965
25	0x19	00100101	0x25	5966
26	0x1A	00110100	0x34	5A65
27	0x1B	00110101	0x35	5A66
28	0x1C	00100110	0x26	5969
29	0x1D	00100111	0x27	596A
30	0x1E	00110110	0x36	5A69
31	0x1F	00110111	0x37	5A6A
32	0x20	01000000	0x40	6555
33	0x21	01000001	0x41	6556
34	0x22	01010000	0x50	6655
35	0x23	01010001	0x51	6656
36	0x24	01000010	0x42	6559
37	0x25	01000011	0x43	655A
38	0x26	01010010	0x52	6659
39	0x27	01010011	0x53	665A
40	0x28	01100000	0x60	6955

41	0x29	01100001	0x61	6956
42	0x2A	01110000	0x70	6A55
43	0x2B	01110001	0x71	6A56
44	0x2C	01100010	0x62	6959
45	0x2D	01100011	0x63	695A
46	0x2E	01110010	0x72	6A59
47	0x2F	01110011	0x73	6A5A
48	0x30	01000100	0x44	6565
49	0x31	01000101	0x45	6566
50	0x32	01010100	0x54	6665
51	0x33	01010101	0x55	6666
52	0x34	01000110	0x46	6569
53	0x35	01000111	0x47	656A
54	0x36	01010110	0x56	6669
55	0x37	01010111	0x57	666A
56	0x38	01100100	0x64	6965
57	0x39	01100101	0x65	6966
58	0x3A	01110100	0x74	6A65
59	0x3B	01110101	0x75	6A66
60	0x3C	01100110	0x66	6969
61	0x3D	01100111	0x67	696A
62	0x3E	01110110	0x76	6A69
63	0x3F	01110111	0x77	6A6A
64	0x40	00001000	0x08	5595
65	0x41	00001001	0x09	5596
66	0x42	00011000	0x18	5695
67	0x43	00011001	0x19	5696
68	0x44	00001010	0x0A	5599
69	0x45	00001011	0x0B	559A
70	0x46	00011010	0x1A	5699
71	0x47	00011011	0x1B	569A
72	0x48	00101000	0x28	5995
73	0x49	00101001	0x29	5996
74	0x4A	00111000	0x38	5A95
75	0x4B	00111001	0x39	5A96
76	0x4C	00101010	0x2A	5999
77	0x4D	00101011	0x2B	599A
78	0x4E	00111010	0x3A	5A99
79	0x4F	00111011	0x3B	5A9A
80	0x50	00001100	0x0C	55A5
81	0x51	00001101	0x0D	55A6
82	0x52	00011100	0x1C	56A5
83	0x53	00011101	0x1D	56A6
84	0x54	00001110	0x0E	55A9
85	0x55	00001111	0x0F	55AA
86	0x56	00011110	0x1E	56A9
87	0x57	00011111	0x1F	56AA
88	0x58	00101100	0x2C	59A5

89	0x59	00101101	0x2D	59A6
90	0x5A	00111100	0x3C	5AA5
91	0x5B	00111101	0x3D	5AA6
92	0x5C	00101110	0x2E	59A9
93	0x5D	00101111	0x2F	59AA
94	0x5E	00111110	0x3E	5AA9
95	0x5F	00111111	0x3F	5AAA
96	0x60	01001000	0x48	6595
97	0x61	01001001	0x49	6596
98	0x62	01011000	0x58	6695
99	0x63	01011001	0x59	6696
100	0x64	01001010	0x4A	6599
101	0x65	01001011	0x4B	659A
102	0x66	01011010	0x5A	6699
103	0x67	01011011	0x5B	669A
104	0x68	01101000	0x68	6995
105	0x69	01101001	0x69	6996
106	0x6A	01111000	0x78	6A95
107	0x6B	01111001	0x79	6A96
108	0x6C	01101010	0x6A	6999
109	0x6D	01101011	0x6B	699A
110	0x6E	01111010	0x7A	6A99
111	0x6F	01111011	0x7B	6A9A
112	0x70	01001100	0x4C	65A5
113	0x71	01001101	0x4D	65A6
114	0x72	01011100	0x5C	66A5
115	0x73	01011101	0x5D	66A6
116	0x74	01001110	0x4E	65A9
117	0x75	01001111	0x4F	65AA
118	0x76	01011110	0x5E	66A9
119	0x77	01011111	0x5F	66AA
120	0x78	01101100	0x6C	69A5
121	0x79	01101101	0x6D	69A6
122	0x7A	01111100	0x7C	6AA5
123	0x7B	01111101	0x7D	6AA6
124	0x7C	01101110	0x6E	69A9
125	0x7D	01101111	0x6F	69AA
126	0x7E	01111110	0x7E	6AA9
127	0x7F	01111111	0x7F	6AAA
128	0x80	10000000	0x80	9555
129	0x81	10000001	0x81	9556
130	0x82	10010000	0x90	9655
131	0x83	10010001	0x91	9656
132	0x84	10000010	0x82	9559
133	0x85	10000011	0x83	955A
134	0x86	10010010	0x92	9659
135	0x87	10010011	0x93	965A
136	0x88	10100000	0xA0	9955

137	0x89	10100001	0xA1	9956
138	0x8A	10110000	0xB0	9A55
139	0x8B	10110001	0xB1	9A56
140	0x8C	10100010	0xA2	9959
141	0x8D	10100011	0xA3	995A
142	0x8E	10110010	0xB2	9A59
143	0x8F	10110011	0xB3	9A5A
144	0x90	10000100	0x84	9565
145	0x91	10000101	0x85	9566
146	0x92	10010100	0x94	9665
147	0x93	10010101	0x95	9666
148	0x94	10000110	0x86	9569
149	0x95	10000111	0x87	956A
150	0x96	10010110	0x96	9669
151	0x97	10010111	0x97	966A
152	0x98	10100100	0xA4	9965
153	0x99	10100101	0xA5	9966
154	0x9A	10110100	0xB4	9A65
155	0x9B	10110101	0xB5	9A66
156	0x9C	10100110	0xA6	9969
157	0x9D	10100111	0xA7	996A
158	0x9E	10110110	0xB6	9A69
159	0x9F	10110111	0xB7	9A6A
160	0xA0	11000000	0xC0	A555
161	0xA1	11000001	0xC1	A556
162	0xA2	11010000	0xD0	A655
163	0xA3	11010001	0xD1	A656
164	0xA4	11000010	0xC2	A559
165	0xA5	11000011	0xC3	A55A
166	0xA6	11010010	0xD2	A659
167	0xA7	11010011	0xD3	A65A
168	0xA8	11100000	0xE0	A955
169	0xA9	11100001	0xE1	A956
170	0xAA	11110000	0xF0	AA55
171	0xAB	11110001	0xF1	AA56
172	0xAC	11100010	0xE2	A959
173	0xAD	11100011	0xE3	A95A
174	0xAE	11110010	0xF2	AA59
175	0xAF	11110011	0xF3	AA5A
176	0xB0	11000100	0xC4	A565
177	0xB1	11000101	0xC5	A566
178	0xB2	11010100	0xD4	A665
179	0xB3	11010101	0xD5	A666
180	0xB4	11000110	0xC6	A569
181	0xB5	11000111	0xC7	A56A
182	0xB6	11010110	0xD6	A669
183	0xB7	11010111	0xD7	A66A
184	0xB8	11100100	0xE4	A965

185	0xB9	11100101	0xE5	A966
186	0xBA	11110100	0xF4	AA65
187	0xBB	11110101	0xF5	AA66
188	0xBC	11100110	0xE6	A969
189	0xBD	11100111	0xE7	A96A
190	0xBE	11110110	0xF6	AA69
191	0xBF	11110111	0xF7	AA6A
192	0xC0	10001000	0x88	9595
193	0xC1	10001001	0x89	9596
194	0xC2	10011000	0x98	9695
195	0xC3	10011001	0x99	9696
196	0xC4	10001010	0x8A	9599
197	0xC5	10001011	0x8B	959A
198	0xC6	10011010	0x9A	9699
199	0xC7	10011011	0x9B	969A
200	0xC8	10101000	0xA8	9995
201	0xC9	10101001	0xA9	9996
202	0xCA	10111000	0xB8	9A95
203	0xCB	10111001	0xB9	9A96
204	0xCC	10101010	0xAA	9999
205	0xCD	10101011	0xAB	999A
206	0xCE	10111010	0xBA	9A99
207	0xCF	10111011	0xBB	9A9A
208	0xD0	10001100	0x8C	95A5
209	0xD1	10001101	0x8D	95A6
210	0xD2	10011100	0x9C	96A5
211	0xD3	10011101	0x9D	96A6
212	0xD4	10001110	0x8E	95A9
213	0xD5	10001111	0x8F	95AA
214	0xD6	10011110	0x9E	96A9
215	0xD7	10011111	0x9F	96AA
216	0xD8	10101100	0xAC	99A5
217	0xD9	10101101	0xAD	99A6
218	0xDA	10111100	0xBC	9AA5
219	0xDB	10111101	0xBD	9AA6
220	0xDC	10101110	0xAE	99A9
221	0xDD	10101111	0xAF	99AA
222	0xDE	10111110	0xBE	9AA9
223	0xDF	10111111	0xBF	9AAA
224	0xE0	11001000	0xC8	A595
225	0xE1	11001001	0xC9	A596
226	0xE2	11011000	0xD8	A695
227	0xE3	11011001	0xD9	A696
228	0xE4	11001010	0xCA	A599
229	0xE5	11001011	0xCB	A59A
230	0xE6	11011010	0xDA	A699
231	0xE7	11011011	0xDB	A69A
232	0xE8	11101000	0xE8	A995

233	0xE9	11101001	0xE9	A996
234	0xEA	11111000	0xF8	AA95
235	0xEB	11111001	0xF9	AA96
236	0xEC	11101010	0xEA	A999
237	0xED	11101011	0xEB	A99A
238	0xEE	11111010	0xFA	AA99
239	0xEF	11111011	0xFB	AA9A
240	0xF0	11001100	0xCC	A5A5
241	0xF1	11001101	0xCD	A5A6
242	0xF2	11011100	0xDC	A6A5
243	0xF3	11011101	0xDD	A6A6
244	0xF4	11001110	0xCE	A5A9
245	0xF5	11001111	0xCF	A5AA
246	0xF6	11011110	0xDE	A6A9
247	0xF7	11011111	0xDF	A6AA
248	0xF8	11101100	0xEC	A9A5
249	0xF9	11101101	0xED	A9A6
250	0xFA	11111100	0xFC	AAA5
251	0xFB	11111101	0xFD	AAA6
252	0xFC	11101110	0xEE	A9A9
253	0xFD	11101111	0xEF	A9AA
254	0xFE	11111110	0xFE	AAA9
255	0xFF	11111111	0xFF	AAAA

PC Control of MTH Engines by Serial Connection to the TIU
14. Sep. 2020

- **Translation Table – Codeword to Databyte**

Codeword	Encoded Hex	Encoded Binary (Bitswap)	Databyte Hex	Databyte Decimal
5555	0x00	00000000	0x00	0
5556	0x01	00000001	0x01	1
5559	0x02	00000010	0x04	4
555A	0x03	00000011	0x05	5
5565	0x04	00000100	0x10	16
5566	0x05	00000101	0x11	17
5569	0x06	00000110	0x14	20
556A	0x07	00000111	0x15	21
5595	0x08	00001000	0x40	64
5596	0x09	00001001	0x41	65
5599	0x0A	00001010	0x44	68
559A	0x0B	00001011	0x45	69
55A5	0x0C	00001100	0x50	80
55A6	0x0D	00001101	0x51	81
55A9	0x0E	00001110	0x54	84
55AA	0x0F	00001111	0x55	85
5655	0x10	00010000	0x02	2
5656	0x11	00010001	0x03	3
5659	0x12	00010010	0x06	6
565A	0x13	00010011	0x07	7
5665	0x14	00010100	0x12	18
5666	0x15	00010101	0x13	19
5669	0x16	00010110	0x16	22
566A	0x17	00010111	0x17	23
5695	0x18	00011000	0x42	66
5696	0x19	00011001	0x43	67
5699	0x1A	00011010	0x46	70
569A	0x1B	00011011	0x47	71
56A5	0x1C	00011100	0x52	82
56A6	0x1D	00011101	0x53	83
56A9	0x1E	00011110	0x56	86
56AA	0x1F	00011111	0x57	87
5955	0x20	00100000	0x08	8
5956	0x21	00100001	0x09	9
5959	0x22	00100010	0x0C	12
595A	0x23	00100011	0x0D	13
5965	0x24	00100100	0x18	24
5966	0x25	00100101	0x19	25
5969	0x26	00100110	0x1C	28
596A	0x27	00100111	0x1D	29
5995	0x28	00101000	0x48	72

5996	0x29	00101001	0x49	73
5999	0x2A	00101010	0x4C	76
599A	0x2B	00101011	0x4D	77
59A5	0x2C	00101100	0x58	88
59A6	0x2D	00101101	0x59	89
59A9	0x2E	00101110	0x5C	92
59AA	0x2F	00101111	0x5D	93
5A55	0x30	00110000	0x0A	10
5A56	0x31	00110001	0x0B	11
5A59	0x32	00110010	0x0E	14
5A5A	0x33	00110011	0x0F	15
5A65	0x34	00110100	0x1A	26
5A66	0x35	00110101	0x1B	27
5A69	0x36	00110110	0x1E	30
5A6A	0x37	00110111	0x1F	31
5A95	0x38	00111000	0x4A	74
5A96	0x39	00111001	0x4B	75
5A99	0x3A	00111010	0x4E	78
5A9A	0x3B	00111011	0x4F	79
5AA5	0x3C	00111100	0x5A	90
5AA6	0x3D	00111101	0x5B	91
5AA9	0x3E	00111110	0x5E	94
5AAA	0x3F	00111111	0x5F	95
6555	0x40	01000000	0x20	32
6556	0x41	01000001	0x21	33
6559	0x42	01000010	0x24	36
655A	0x43	01000011	0x25	37
6565	0x44	01000100	0x30	48
6566	0x45	01000101	0x31	49
6569	0x46	01000110	0x34	52
656A	0x47	01000111	0x35	53
6595	0x48	01001000	0x60	96
6596	0x49	01001001	0x61	97
6599	0x4A	01001010	0x64	100
659A	0x4B	01001011	0x65	101
65A5	0x4C	01001100	0x70	112
65A6	0x4D	01001101	0x71	113
65A9	0x4E	01001110	0x74	116
65AA	0x4F	01001111	0x75	117
6655	0x50	01010000	0x22	34
6656	0x51	01010001	0x23	35
6659	0x52	01010010	0x26	38
665A	0x53	01010011	0x27	39
6665	0x54	01010100	0x32	50
6666	0x55	01010101	0x33	51
6669	0x56	01010110	0x36	54
666A	0x57	01010111	0x37	55
6695	0x58	01011000	0x62	98

6696	0x59	01011001	0x63	99
6699	0x5A	01011010	0x66	102
669A	0x5B	01011011	0x67	103
66A5	0x5C	01011100	0x72	114
66A6	0x5D	01011101	0x73	115
66A9	0x5E	01011110	0x76	118
66AA	0x5F	01011111	0x77	119
6955	0x60	01100000	0x28	40
6956	0x61	01100001	0x29	41
6959	0x62	01100010	0x2C	44
695A	0x63	01100011	0x2D	45
6965	0x64	01100100	0x38	56
6966	0x65	01100101	0x39	57
6969	0x66	01100110	0x3C	60
696A	0x67	01100111	0x3D	61
6995	0x68	01101000	0x68	104
6996	0x69	01101001	0x69	105
6999	0x6A	01101010	0x6C	108
699A	0x6B	01101011	0x6D	109
69A5	0x6C	01101100	0x78	120
69A6	0x6D	01101101	0x79	121
69A9	0x6E	01101110	0x7C	124
69AA	0x6F	01101111	0x7D	125
6A55	0x70	01110000	0x2A	42
6A56	0x71	01110001	0x2B	43
6A59	0x72	01110010	0x2E	46
6A5A	0x73	01110011	0x2F	47
6A65	0x74	01110100	0x3A	58
6A66	0x75	01110101	0x3B	59
6A69	0x76	01110110	0x3E	62
6A6A	0x77	01110111	0x3F	63
6A95	0x78	01111000	0x6A	106
6A96	0x79	01111001	0x6B	107
6A99	0x7A	01111010	0x6E	110
6A9A	0x7B	01111011	0x6F	111
6AA5	0x7C	01111100	0x7A	122
6AA6	0x7D	01111101	0x7B	123
6AA9	0x7E	01111110	0x7E	126
6AAA	0x7F	01111111	0x7F	127
9555	0x80	10000000	0x80	128
9556	0x81	10000001	0x81	129
9559	0x82	10000010	0x84	132
955A	0x83	10000011	0x85	133
9565	0x84	10000100	0x90	144
9566	0x85	10000101	0x91	145
9569	0x86	10000110	0x94	148
956A	0x87	10000111	0x95	149
9595	0x88	10001000	0xC0	192

9596	0x89	10001001	0xC1	193
9599	0x8A	10001010	0xC4	196
959A	0x8B	10001011	0xC5	197
95A5	0x8C	10001100	0xD0	208
95A6	0x8D	10001101	0xD1	209
95A9	0x8E	10001110	0xD4	212
95AA	0x8F	10001111	0xD5	213
9655	0x90	10010000	0x82	130
9656	0x91	10010001	0x83	131
9659	0x92	10010010	0x86	134
965A	0x93	10010011	0x87	135
9665	0x94	10010100	0x92	146
9666	0x95	10010101	0x93	147
9669	0x96	10010110	0x96	150
966A	0x97	10010111	0x97	151
9695	0x98	10011000	0xC2	194
9696	0x99	10011001	0xC3	195
9699	0x9A	10011010	0xC6	198
969A	0x9B	10011011	0xC7	199
96A5	0x9C	10011100	0xD2	210
96A6	0x9D	10011101	0xD3	211
96A9	0x9E	10011110	0xD6	214
96AA	0x9F	10011111	0xD7	215
9955	0xA0	10100000	0x88	136
9956	0xA1	10100001	0x89	137
9959	0xA2	10100010	0x8C	140
995A	0xA3	10100011	0x8D	141
9965	0xA4	10100100	0x98	152
9966	0xA5	10100101	0x99	153
9969	0xA6	10100110	0x9C	156
996A	0xA7	10100111	0x9D	157
9995	0xA8	10101000	0xC8	200
9996	0xA9	10101001	0xC9	201
9999	0xAA	10101010	0xCC	204
999A	0xAB	10101011	0xCD	205
99A5	0xAC	10101100	0xD8	216
99A6	0xAD	10101101	0xD9	217
99A9	0xAE	10101110	0xDC	220
99AA	0xAF	10101111	0xDD	221
9A55	0xB0	10110000	0x8A	138
9A56	0xB1	10110001	0x8B	139
9A59	0xB2	10110010	0x8E	142
9A5A	0xB3	10110011	0x8F	143
9A65	0xB4	10110100	0x9A	154
9A66	0xB5	10110101	0x9B	155
9A69	0xB6	10110110	0x9E	158
9A6A	0xB7	10110111	0x9F	159
9A95	0xB8	10111000	0xCA	202

9A96	0xB9	10111001	0xCB	203
9A99	0xBA	10111010	0xCE	206
9A9A	0xBB	10111011	0xCF	207
9AA5	0xBC	10111100	0xDA	218
9AA6	0xBD	10111101	0xDB	219
9AA9	0xBE	10111110	0xDE	222
9AAA	0xBF	10111111	0xDF	223
A555	0xC0	11000000	0xA0	160
A556	0xC1	11000001	0xA1	161
A559	0xC2	11000010	0xA4	164
A55A	0xC3	11000011	0xA5	165
A565	0xC4	11000100	0xB0	176
A566	0xC5	11000101	0xB1	177
A569	0xC6	11000110	0xB4	180
A56A	0xC7	11000111	0xB5	181
A595	0xC8	11001000	0xE0	224
A596	0xC9	11001001	0xE1	225
A599	0xCA	11001010	0xE4	228
A59A	0xCB	11001011	0xE5	229
A5A5	0xCC	11001100	0xF0	240
A5A6	0xCD	11001101	0xF1	241
A5A9	0xCE	11001110	0xF4	244
A5AA	0xCF	11001111	0xF5	245
A655	0xD0	11010000	0xA2	162
A656	0xD1	11010001	0xA3	163
A659	0xD2	11010010	0xA6	166
A65A	0xD3	11010011	0xA7	167
A665	0xD4	11010100	0xB2	178
A666	0xD5	11010101	0xB3	179
A669	0xD6	11010110	0xB6	182
A66A	0xD7	11010111	0xB7	183
A695	0xD8	11011000	0xE2	226
A696	0xD9	11011001	0xE3	227
A699	0xDA	11011010	0xE6	230
A69A	0xDB	11011011	0xE7	231
A6A5	0xDC	11011100	0xF2	242
A6A6	0xDD	11011101	0xF3	243
A6A9	0xDE	11011110	0xF6	246
A6AA	0xDF	11011111	0xF7	247
A955	0xE0	11100000	0xA8	168
A956	0xE1	11100001	0xA9	169
A959	0xE2	11100010	0xAC	172
A95A	0xE3	11100011	0xAD	173
A965	0xE4	11100100	0xB8	184
A966	0xE5	11100101	0xB9	185
A969	0xE6	11100110	0xBC	188
A96A	0xE7	11100111	0xBD	189
A995	0xE8	11101000	0xE8	232

A996	0xE9	11101001	0xE9	233
A999	0xEA	11101010	0xEC	236
A99A	0xEB	11101011	0xED	237
A9A5	0xEC	11101100	0xF8	248
A9A6	0xED	11101101	0xF9	249
A9A9	0xEE	11101110	0xFC	252
A9AA	0xEF	11101111	0xFD	253
AA55	0xF0	11110000	0xAA	170
AA56	0xF1	11110001	0xAB	171
AA59	0xF2	11110010	0xAE	174
AA5A	0xF3	11110011	0xAF	175
AA65	0xF4	11110100	0xBA	186
AA66	0xF5	11110101	0xBB	187
AA69	0xF6	11110110	0xBE	190
AA6A	0xF7	11110111	0xBF	191
AA95	0xF8	11111000	0xEA	234
AA96	0xF9	11111001	0xEB	235
AA99	0xFA	11111010	0xEE	238
AA9A	0xFB	11111011	0xEF	239
AAA5	0xFC	11111100	0xFA	250
AAA6	0xFD	11111101	0xFB	251
AAA9	0xFE	11111110	0xFE	254
AAAA	0xFF	11111111	0xFF	255