Subject: Diving Deep into the CTS256A-AL2 Firmware
Posted by jayindallas on Thu, 10 Oct 2024 04:07:24 GMT
View Forum Message <> Reply to Message

DISCUSSION: Diving Deep into the CTS256A-AL2 Firmware

CONTENTS:

CTS Code to Say "OK"
When you power up a CTS256A-AL2 CODE-TO_SPEECH processing chip with the circuitry including the SP0256 Narrator(tm) Speech Processor, it silently initializes and if everything is working, the 'CTS' will make the 'SP0' voice-synthesize, "O-K". As that's the first thing we hear, its an interesting and easy place to start looking at the internal CTS original ROM code.

ADDR MCODE Time ifJP LINE# LABEL OPCODE OPERANDS BYTES REALTIME Line by line description
===== ======== ==== ==== ===== ====== ====== ========== ===== ========== ========================

```
                                                    ___ ____________
F1A8: 4F2D4B   -   -   242: STROK  TEXT   'O-K'      | 3B|       0Tc| The 'O-K' data, takes 3 bytes, but zero time
F1AB: 0D       -   -   243:        BYTE   >0D        | 1B|       0Tc| The 'carriage return' taked 2 byte, zero time
F1AC: 73F90A   9Tc -   245: SAYOK  AND    %>F9,R10   | 3B| 1x09=  9Tc| This sets a buffer necessary flag
F1AF: C5       5Tc -   246:        CLR    B          | 1B| 1x05=  5Tc| Clear Register B, initialize the loop counter (0:4)
F1B0: AA1000   13Tc -   248: LF1B0  LDA    @STROK(B)  | 3B| 4x13= 52Tc| 4x: get next B indexed character, put in Register A
F1B3: 8EF1E2   14Tc -   249:        CALL   @STINPB    | 3B| 4x14= 56Tc| 4x: call subroutine to store A into the input buffer
F1B6: C3       5Tc -   250:        INC    B          | 1B| 4x05= 20Tc| 4x: bump B to point the the next char
F1B7: 5D04     7Tc -   251:        CMP    %>04,B     | 2B| 4x07= 28Tc| 4x: test loop counter, if 4: DONE (zero flag updated)
F1B9: E6F5     5Tc 07Tc  252:      JNZ    LF1B0      | 2B| 3x07= 21Tc| 3x: JNZ_/ jump back to loop on counter = {1,2,3}
                                  (+jnz drop through) | - | 1x05=  5Tc| 1x: JNZ \ drop through on counter = {4}
                                                |===|     =====|
```

```
                                |19B|      196Tc| TOTAL RUN-TIME
                                |___|___________| (not including subroutine duration)
```

The code above is easy to understand because its similar to other CPU instructions and the descriptions in the right margin can also help. The 'TOTAL RUN-TIME' of the code is summed in the box. Example: '4x13= 52Tc' means this instruction is done 4 times in the loop construct. Each time this instruction is executed, it takes 13Tc units of time and the total that that instructiong is 52Tc.

Tc is a shortened version of Tc(C) which is the internal state cycle period, or how long one cycle takes based upon (1) the frequency of the crystal or external clock source and (2) the CTS chip's internal divide-by-2 or divide-by-4 masked-circuit. The Data Manual offers this example: With a 5Mhz crystal and a divide-by-2 internal circuit, it has an internal frequency of 2.5Mhz. The period of the internal frequency is its reciprical, 400ns per Tc(C). In the code above, 196Tc * 400ns = 78.4 us (micro-seconds).

Line# 248 is the most unusual instruction in the code. Its going to load Register A with a value from an addressed table (STROK) using Register B as an index or offset relative to the table address, to get the correct data/character. While we don't know the specifics of why Line# 245 used a particular value, nor what specifically subroutine STINPB* does, other than stick a single character into the input buffer, we can understand this simple piece of code. *STINPB will be covered in a later posting about CTS Buffers.

An Introduction to "Register Files"
Line# 245, does a logical AND operation on the contents of R10. That is how most of the CPU internal RAM, called "Register File", is referenced. The original CTS chip has 128 bytes of RAM, known as R0 through R127 using decimal numbering. Register A and B are R0 and R1, respectively. Registers A and B are used by instructions intended to execute faster; these often use a set of opcode byte that are micro-coded to use register A or B, instead of using a full byte for another operand. Reducing operands is one way to speed up instructions; one less fetch.

An interesting feature of the TMS7000 CPU is that any two consecutive registers, (Rp-1,Rp) such that Rp does not equal zero, can also be used as a 16-bit value or pointer at any time. The 16-bit register pair such as R5:6 is designated by the least significant byte (LSB) R6 (MSB:LSB). Example, the third line of code executed upon a hardware RESET is:

```
F003: 8820002D   14:  MOVD  %>2000,R45    ; loads the external address of the 'SP0' into register pair R44:45
```

The opcode is a MOVD (move double, i.e. 16-bit value) using an immediate value of hex 2000 and store it into register pair R44:45 (MSB:LSB). There is no rule about the LSB being ODD or EVEN. The only case they warn about is using something like "MOVD %>2000,R0" as the MSB would be Rp-1 or negative 1. That would conceptionally wrap around to R127:0 and the CPU may or may not support the wrap-around situation.

The CTS code has ODD and EVEN register pairs:

```
F0F8: 9C31      152:  BR    *R49        ; ODD, Branch (jump) to a 16-bit address stored in R48:49
F170: DB34      212:  DECD  R52          ; EVEN, 16-bit decrement instruction on the value in
R51:52 (no 16b INCD)
F1D4: 9A2F      269:  LDA   *R47        ; ODD, Reads the parallel port input using the address in
R46:47
```

The last thing to say about CPU internal RAM is that >00xx is its 16-bit address range. The CPU can access internal RAM and control ports using a 16b pointer: LDA *R7 or STA *R7. As its a block of 128 or 256 RAM bytes in various TMS7000 CPUs, Note that the upper address is always >00 when addressing internal RAM as by a 16-bit pointer address. This means you could use 8b buffer constructs and just keep the MSB of the pointer cleared to zero.

For Efficiency and Speed, Use 8-bit Buffer Constructs Where Possible
The algorithm to convert Text-to-Voice is more than a FIFO buffer. When focus starts of one letter, the buffer will be scanned for letter groupings that might be a pattern for one group allophone. This means that more time is spent in the buffer that is used for Text-to-Voice and some scans will be wasted efforts looking until it can abandon looking further. Due to this nature of multiple scanning for letter pattern scenarios, this buffer should be carefully constructed for efficiency and speed. Whether the special buffer is run in internal RAM or external RAM, it can be constructed to use less than 256 locations. Separating the Algorithm buffer to 16 or 32 bytes would allow it to run quickly. All the math is 8-bit, and the buffer boundaries can be forced by incrementing or decrementing the 8-bit pointers and using a logical AND to remove any out of range values; an automatic wrap-around in the buffer; as one example.

The CTS implementation uses 16-bit buffers and has a flag table to configure its acrobatic algorithm scans described above. The buffer code has to read those flags again when asked to do something. Its big overhead and slows things down. When there is no designated external RAM, the two buffers are created in internal RAM. It uses a 16-bit construct (because it might run a big buffer in external RAM with the same code, if its configured?) The method is to bump a 16-bit pointer then compare it to the over-range value in a 16-bit register pair. If they equal, the buffer needs to wrap-around and another stored 16-bit value is then written over the pointer to make it wrap around. Too much activity when simplicity would be easier to code.

To make matters worse, the CTS code in one of the pattern testing routines, uses the multiply instruction, which run in the range of 44Tc to 49Tc (MPY %>02,B runs in 46Tc) whereas the longest non-Multiply instructions is 17Tc. A x2 multiplication can be done in binary math as a 'RLC B' at 5Tc if the Carry flag is already cleared. Worst case is 'CLRC @ 6Tc' + 'RLC B @ 5Tc = 11Tc.'

In fairness to the original CTS coders, they had a schedule to get the code working so they could sell the CTS. They accomplished that requirement. In addition, poor CPU documentation of an 'interesting' instruction, BR LABEL(B) is poorly documented in all three versions of the CPU manual. CTS coders hacked a routine that worked and moved on using a x3 to jump to a table of 3b BR (Branch) instructions. It works.

In 2024, we have different project constraints and anyone can look at the code and modify it.

This would be a good modification of codespace for 2024+. :)

An Introduction to Assembling TMS7000 Machine Code:
Take another look at the F003: machine code. It illustrates the general way an instruction is assembled into machine code.

```
F003: 8820002D   14:  MOVD  %>2000,R45
      ^^^^^^^^        ^^^^    ^^^^ ^^^
```

The first byte of the machine code of an instruction is the opcode. The TMS 7000 CPU has several instruction opcodes that work for one particular operand syntax. MOVD has three unique opcodes, one for each operand syntax. For example:

Opcode	Mnemonic Syntax	Bytes	Operands	Machine Code	Comments
88	MOVD %>iop,Rd	4	16b_iop,8b_dst	88.mm.ll.dd	--.mm.ll.-- are the 16b immediate operand in MSB.LSB order
A8	MOVD %>iop(B),Rd	4	16b_iop,8b_dst	A8.mm.ll.dd	--.--.--.dd is the 8b hex value of the Rd decimal value.
98	MOVD Rs,Rd	3	8b_src,8b_dst	98.ss.dd	--.ss.dd are 2 8b hex values of the Rs,Rd decimal values.

From an assembler perspective, you identify the operand field syntax to identify the which instruction Opcode applies. Then you place the operands in the next bytes of machine code in the order the are listed in the operand field, left to right. Be aware that references to Register A or B are usually designated in the opcode byte, so they are generally *NOT* in an additional operand byte of machine code. Keeps it simple and faster. One exception is an assembler instruction that moves values between A and B, but its technically not an instruction, its of the syntax A,Rx or B,Rx and the assembler is expected to convert the destination by Register Name (A or B) to its associated Rx name (R0 or R1 respectively). If I write an assembler I'll dig and tabulate those exceptions.

However, note that the "%>iop(B)" operand does NOT need to specify the Reg B (R1) because, (1) there are not enough bytes in the machine code to explicitly reference B, and the A8 opcode syntax obviously makes it an implied operand. Opcode A8 uses B, there is nothing variable about that. So the machine code A8.mm.ll.dd has no room for A or B explicitly; that would require A8.mm.ll.ss,dd where ss is 01 for R01 aka Register B.

There is a file in the ZIP-File_01.zip that has these machine code constructs:
The_Method_of_Assembling_Machine_Code.lst

When this makes sense to you, you'll have a natural method to assemble machine code without looking through the tedious explanations, mostly in the 1983 TMS7000 manual. If you want to write an assembler, then I suggest you read the manual too.

Yes, I used "Rs,Rd" instead of the manual's "Rp,Rp" for register pair. This adds clarity that the source is always/usually listed first and the latter operands tend to be destinations. Using "Rp,Rp" doesn't help the novice know which register is which.

Say "OK" a Faster Way:
Being a fan of the Z80, I started to write a TMS7000 "DJNZ version" using Reg B as the counter and the index too. I was surprised when I used Linux GREP to see what instructions the CTS coders used: they didn't use any DJNZ!?!? However, looking at the loop "4x...=" over an over, I thought I should try a straight drop-through LOAD&CALL routine for speed and length comparison. I wrote the following code, with a TRAP n instruction which I'll explain later; for now... think of it as a short-cut, 1-byte CALL that isn't any faster that a 3-byte call but a feature that allows you to recover a lot of codespace when you have subroutines called from many places in the code.

```
ADDR  MCODE    Time ifJP LINE# LABEL  OPCODE OPERANDS   BYTES REALTIME    Line by line description
===== ======== ==== ==== ===== ====== ====== ========== ===== ========== ======================
                                          ___ ___________
F1A8: 73F90A    9Tc  -    245: SAYOK  AND    %>F9,R10   | 3B| 1x09=  9Tc| set some flags for the buffer...
F1AB: 224F      7Tc  -    246:       MOV    %>4F,A     | 2B| 1x07=  7Tc| put an "O" into Register A
F1AD: EE       14Tc  -    247:       TRAP   6          | 1B| 1x14= 14Tc| TRAP 6 is a 1-byte CALL to STINPB
F1AE: 222D      7Tc  -    248:       MOV    %>2D,A     | 2B| 1x07=  7Tc| put an "-" into Register A
F1AF: EE       14Tc  -    249:       TRAP   6          | 1B| 1x14= 14Tc| TRAP 6 is a 1-byte CALL to STINPB
F1B0: 224B      7Tc  -    250:       MOV    %>4B,A     | 2B| 1x07=  7Tc| put an "K" into Register A
F1B2: EE       14Tc  -    251:       TRAP   6          | 1B| 1x14= 14Tc| TRAP 6 is a 1-byte CALL to STINPB
F1B3: 220D      7Tc  -    252:       MOV    %>0D,A     | 2B| 1x07=  7Tc| put an ASCII 'CR' character into Reg A
F1B5: EE       14Tc  -    253:       TRAP   6          | 1B| 1x14= 14Tc| TRAP 6 is a 1-byte CALL to STINPB
...                                      |   |         |
FFF2: DATA STINPB                                |+2B|          | CALL address in table, +2B
                                         |===|     =====|
                                         |17B|      93Tc| TOTAL RUN-TIME (Twice as fast)
                                         |___|__________| (not including subroutine duration)
```

Note: During initialization, execution speed is less important. When initialization is completed and the CPU has some real-time constraints to perform its function, then speed coupled with efficient algorithms and processing is important.

LESSON??? Could it be... Sometimes the best Real-Time code is writing like it was straight-line BASIC!?!? In a way... YES. I'd re-phrase that to say that the better lesson is to be careful of your assumptions. Spend a little time to verify assumptions. When the unexpected is the best solution, its good to be the one that figured that out.

Next Time...
An introduction to the ZIP_File_01.zip, A useful collection of Tables, Charts and Notes about CTS and TMS7000 CPU.

Coming Soon...
How switching the most used subroutines CALLs to TRAP n instructions saves 122 bytes of CTS codespace.

Subject: Table of Contents: ZIP_File_01.zip
Posted by jayindallas on Sat, 19 Oct 2024 05:53:53 GMT
View Forum Message <> Reply to Message

DISCUSSION: Diving Deep into the CTS256A-AL2 Firmware

CONTENTS:

Introduction to the ZIP-Task and its Contents
This posting describes the contents of ZIP_File_01.zip, which can be downloaded at the bottom of this message. It has a collection of TMS7000 Series documents I've been creating as I was 'Diving Deep into the CTS256A-AL2 Firmware.' I've included RELEASED and PRE-RELEASE files at this time. Later I'll complete the PRE-RELEASE. Even though I have more information to add to those files, its currently very useful to anyone that needs to code in TMS7000 or PIC7000.

Back in my early day of real-time coding, I realized that having a condensed header file of the most useful stuff from the data manuals and chip specification in a commented text format, inserted into my code file, made it a lot easier and quicker to look up needed information as I was coding. If blocks of this information were in my file, I could reference it while coding without taking my hands off the keyboard or display. I'd keep the IO port and bit definitions near the code that used it. The rest of the info-block was stuffed at the beginning or end of the code file. Press HOME or END and I was there without juggling data books and chip specifications on my desk. I'd use the assembler directive to turn off printing of those blocks to keep my printed listings about code only, unless I needed a certain block as for reference during prototype testing. It saved me a lot of time working even though it required a lot of typing. Any time I worked on the same micro, I'd just pull the blocks out of an older project and was ready to code.

On this CTS Project 2024, the starting postion on this code was rough: (1) no original coders a

phone call away, (2) no project notes, (3) no code listings, (4) no code comments, (5) no code flowchart, and (6) no code meaningful-labels. Just a ROM image.

The CTS Disassmbly Project of 2022 did a great job analyzing the code image and give a lot of key labels and comments to the code. That will help anyone learning about the CTS code. To get into the head of the original coder/coders, I did an analysis of code image many ways, and you'll see some of the products of that in these files. Adding that to the 2024 project made a big difference.

As its a more interesting topic of learning code from just a ROM image, I decided it was worth sharing some of this strategy and discovery. This dump of documents in the ZIP_File_01 I've created, will help anyone else that might want to do some coding on the Daisy256 that Andrew Lynch is developing. Even if not, it might give you some insights on how to approach this sort of challenge if it something like this ever drops on your work desk. :)

I'm posting this as the second message because it defines the path of some posting topics, in a way. And rather than attaching files after every posting, it seemed easier to post it here with everything the is RELEASED and PRE-RELEASE level. Later there may be 2 more ZIP files attached to this message. When I'm posting messages, I'll just reference the ZIP_File number and the files that are relevent to the posting. Its easier that way.

I also have a file included that will be posted on my next posting about "How switching the most used subroutines CALLs to TRAP n instructions saves 122 bytes of CTS codespace. Its an image of my spreadsheet in that upcoming posting. I like the first message's use of inserting examples from the actual CTS code and so I'm editing my postings to provide that more often. In digging those files up, I was doing most of the ZIP-file collection, so it got jumped to posting #2. I think it makes sense.

So... for now, the rest of this posting lists and describes the ZIP_File_01...

ZIP File #1
ZIP: The_Directory_of_ZIP_File_Contents.lst
This is the table of contents for the ZIP file. It consists of this description in a text file format.

About the original CTS firmware
CTS: The_Reference_CTS_ASM_Listing.lst
The original listing was provided by the 2022 CTS disassembly project. That coder did a great job of learning the CTS code and adding the labels and comments to the disassembly listing. I'll update this with the 2022 author's name when I find it in my CTS directories.
|
This is my reference listing, for the 2024 CTS+SP0 Project initiated by Andrew J. Lynch. This listing is a derivative of the original and the top of the file describes the modifications.

CTS: The_Sorted_CTS_Addr-to-Label_and_Label-to-Addr.lst
A simple lookup table to find a label address in the original CTS firmware, or to find what label is at a particular address. I used this mostly to calculate the relative address when hand assembling

code and to find out if a label or data block is within range of 256 bytes of relative addressing.

CTS: The_Survey_of_CTS_Instructions_used.lst
My first look at the code was to use the Linux GREP utility to find out the set of TMS7000 instructions and the amount of their use, to get an idea about how well the original coders utilized the unique power of the TMS7000 and to get an impression on prior experience with that CPU.

CTS: The_Survey_of_CTS_Labels.lst
This is a simple Linux GREP of all labels to reveal some of the flowchart structure. Each label set has the line of code using the label first then all lines that invoke or reference that label.

CTS: The_Survey_of_CTS_Rx_Register_Files.lst
Another Linux GREP to reveal how the 'Register File' area (CPU RAM) was utilized. It identifies register pairs being used as 16-bit values such as pointers.

Data Manuals for the CPU
CPU: The_1983_TMS7000_Data_Manual_SPND001A.pdf
Click to Download: https://bitsavers.org/components/ti/TMS7000/SPND001A_TMS7000_Family_Data_Manual_1983.pdf
The original CPU manual, 1 of 3 editions. It covers the 70x1, used in the CTS256A-AL2, and the 70x0. Its a very techie presentation, seemingly written by two of the CPU designers, with slightly contrary views on terms, operand syntax, descriptions and document composition. That often leads to ambiguity.

It has a lot of fascinating detail that the subsequent manuals removed. Some of its 'jewels' are:
(1) descriptions of how masked-ROM users could create new instructions at the microcode level,
(2) a software serial port for the 70x0 CPU that has no serial control port, and
(3) a raw description on instruction assembly for writing a TMS7000 assembler.
Beware of early edition mistakes in its documentation.

CPU: The_1986_TMS7000_Data_Manual_SPND001B.pdf
Click to Download: https://bitsavers.org/components/ti/TMS7000/SPND001B_TMS7000_Family_Data_Manual_1986.pdf

The second edition focused on commercial CPU market; it keeps it simple and more concise. Its a good manual that introduces the TMS70Cx2. I use my paperback as my first GOTO source and I keep all three editions in PDFs when researching TMS7000 information.

CPU: The_1989_TMS7000_Data_Manual_SPND001C.pdf
Click to Download: https://bitsavers.org/components/ti/TMS7000/SPND001C_TMS7000_Family_Data_Manual_1989.pdf
The final edition, best to use among the PDF files for accurate information. My only complaint is that the description of the instruction, 'BR LABEL(B)' is the same in all editions and is insufficient. I suspect the CPU designers thought it was enough, because they already understood how it worked. In subsequent editions the tech writers couldn't find anyone to explain it so they could improve the explanation. That happens all the time. :)

Useful CPU Information for Coding
CPU: The_Sorted_List_of_CPU_Instructions.lst
A table of opcodes and their operands, byte count and instruction execution time with the affect on the status flags.

*** THE FOLLOWING APPLIES TO ALL FIVE PERIPHERAL MAPS FILE BELOW ***
The various TMS7000 CPUs have different maps. The following files are useful to insert into code when converting the CTS code to another TMS7000 CPU. It includes both ways to address the control registers, 'Px' and 16b addresses. It also describes the change to the ports under the 'full expansion' memory mode used by the CTS firmware. The 2024 pcb design forces the CPU into microprocessor memory mode to access CTS code in external memory. The full expansion descriptions apply to both.

CPU: The_Peripheral_File_Memory_Map_of_the_70x0.txt
A TMS70x0 map of its Peripheral File, a block of control registers for ports and internal modules.

CPU: The_Peripheral_File_Memory_Map_of_the_70x1.txt
A TMS70x1 map of its Peripheral File, a block of control registers for ports and internal modules.

CPU: The_Peripheral_File_Memory_Map_of_the_70x2.txt
A TMS70x2 map of its Peripheral File, a block of control registers for ports and internal modules.

CPU: The_Peripheral_File_Memory_Map_of_all_70Cx2.txt
A TMS70Cx2 map of its Peripheral File, a block of control registers for ports and internal modules.

CPU: The_Peripheral_File_Memory_Map_of_all_70xx.txt
All four TMS7000 maps above in one file.

CPU: The_Peripheral_File_Register_Maps_For_All_TMS7000_PRE-RELEAS E.lst
CPU: The_Peripheral_File_Control_Registers_Ver3_PRE-RELEASE.lst

Files used in the Discussion Messages
MSG: Recovering_Bytes_by_using_TRAP_instead_of_CALL.png
A spreadsheet image used in the posting about replacing some CALL instructions with the TRAP. Applied to the original CTS code, it immediately recovers 122 bytes of codespace for other use.

Useful Information for Writing a TMS7000 Assembler
CPU: The_Method_of_Assembling_Machine_Code.lst
This chart has most of the information needed to make a data structure for an efficient TMS 7000 Series CPU assembler/disassembler and even to make a less efficient pretzel-logic version.
|
This chart contains:
(1) the CPU instructions, alphabetically sorted by opcode, each with its subset of operand syntax.
(2) the instruction's number of bytes of machine code, the first byte opcode in hex, and the

subsequent order of operand bytes identified by symbols.
(3) the normal instruction execution time in 'Tc' units and the +2Tc time when a conditional jump is made.
(4) the affected status flags caused by the instruction.
(5) the legend, or description of what the operand symbols reference, in the right margin.

CPU: The_Method_of_Assembling_Machine_Code_with_CTS.lst
This file has all the information of the file above, but it adds all the CTS code lines using each instruction, under each instruction block. This gives you examples of real code assembly. NOTE: The machine code in the reference listing yet been validated against CTS rom image or disassmbler listing. This file will be updated when when the machine code is validated.

Next Time...
How switching the most used subroutines CALLs to TRAP n instructions saves 122 bytes of CTS codespace.

Coming Soon...
CTS Buffers: STINPB & A look inside a Mattel Electronics, "IntelliVoice, Voice Synthesis Module."

The ZIPFile for download:

File Attachments

1) ZIP_File_01.zip, downloaded 55 times

Subject: Saving 122 Bytes of CTS codespace using TRAP Instructions
Posted by jayindallas on Mon, 21 Oct 2024 01:11:16 GMT
View Forum Message <> Reply to Message

DISCUSSION: Diving Deep into the CTS256A-AL2 Firmware

CONTENTS:

How I Met 'TRAP n'
In 1986, I chose the TMS70C42 for a consumer cordless phone design; the first to use CPU-based packet over voice radio communications. The decision was simple; 7000 Series had the lowest power consumption and consumers wanted cordless phone handsets that didn't have

to be recharged all the time. All other aspects of the 7042 didn't matter, as the digital-design/firmware team member, I'd have to adapt.

My boss and I were at odds over how much masked-ROM space would be needed, he budgeted for 1K, based on nothing, and I told him 2K mask would be needed at 80% probability based on my experience doing this. He said TI says they have a special trick that can reduce the codespace. The TRAP instruction was his unknown basis of defense.

So What is the 'TRAP n' instruction?
No matter what 8-bit CPU you favor, one machine-code pattern exists regarding a JUMP or CALL to a 16-bit address: it takes 3-Bytes, 1B for an opcode and 2B for the address. Often added to that are a few spare opcodes for 1B implied address, JUMP or CALL. The TRAP n instruction is an extension of the 7000 Series interrupt service routine vector lookup table.

When a hardware RESET is actuated, the 70xx CPU will look for a interrupt vector at FFFEh. It reads that address and jumps to that destination to execute the RESET initialization code. For the CTS code, the RESET vector looks like this:

```
LOOKUP  VECTOR
ADDR    ADDRESS
------------------
FFFE    DATA START
```

The interrupts continue this pattern, and RESET is actually INT0, so this is the interrupt vector table; look for the surprise at the end of the comment lines. :

```
LOOKUP  VECTOR
ADDR    ADDRESS
-------------------------------------------------------------------------TRAP 0 through TRAP 5
FFFE    DATA INT0   ; triggered by (1) power-up, (2) hardware RESET, or (3) 'Trap 0' instruction*
FFFC    DATA INT1   ; triggered by (1) interrupt INT1 or (2) 'Trap 1' instruction
FFFA    DATA INT2   ; triggered by (1) interrupt INT2 or (2) 'Trap 2' instruction
FFF8    DATA INT3   ; triggered by (1) interrupt INT3 or (2) 'Trap 3' instruction
FFF6    DATA INT4   ; triggered by (1) interrupt INT4 or (2) 'Trap 4' instruction
FFF4    DATA INT5   ; triggered by (1) interrupt INT5 or (2) 'Trap 5' instruction
```

*Note: A hardware RESET/Power-Up includes some hardware initialization. Therefore Trap 0, a software RESET, is not identical unless you add code to make it do the hardware RESET actions, also. This will be covered later in the topic of INTERRUPTS.

The 70xx had a structure ready for some 1B CALL instructions. They continue at 'Trap 6' through 'Trap 23' decimal numbering:

```
LOOKUP  VECTOR                  MACHINE
ADDR    ADDRESS      INSTRUCTION   CODE     BYTES   TIME
-------------------------------------------------------------------------TRAP 0 through TRAP 5
FFFE    DATA RESET    TRAP 0       E8        1B     14Tc   ; use RETI These interrupts include
FFFC    DATA INT1     TRAP 1       E9        1B     14Tc   ; use RETI   a PUSH ST operation, as
```

```
FFFA    DATA INT2     TRAP 2       EA        1B     14Tc   ; use RETI   a RETI includes a POP ST.
FFF8    DATA INT3     TRAP 3       EB        ...
FFF6    DATA INT4     TRAP 4       EC
FFF4    DATA INT5     TRAP 5       ED
-------------------------------------------------------------------------TRAP 6 through TRAP 23
FFF2    DATA GFLAGS   TRAP 6       EE        1B     14Tc   ; use RETS These TRAP instructions
use
FFF0    DATA FETCH    TRAP 7       EF        1B     14Tc   ; use RETS   a RETS to return.
FFEE    DATA GNEXT    TRAP 8       F0        1B     14Tc   ; use RETS They take 14Tc just like
FFEC    DATA DECR3    TRAP 9       F1        1B     14Tc   ; use RETS   a CALL instruction.
FFEA    DATA RWBUFR   TRAP 10      F2        ...
FFE8    DATA DECR17   TRAP 11      F3
FFE6    DATA ENINT    TRAP 12      F4
FFE4    DATA INCR3    TRAP 13      F5
FFE2    DATA RSEEKB   TRAP 14      F6
FFE0    DATA CHKLTR   TRAP 15      F7
FFDE    DATA CHKPAT   TRAP 16      F8
FFDC    DATA INIPTR   TRAP 17      F9
FFDA    DATA ROLINP   TRAP 18      FA
FFD8    DATA ROLR13   TRAP 19      FB
FFD6    DATA SELRUL   TRAP 20      FC
FFD4    DATA STINPB   TRAP 21      FD
FFD2    DATA UNGET    TRAP 22      FE
;                  TRAP 23 NOT USED
```

Now you know almost everything about a TRAP instruction the easy way; a picture is worth a thousand words. It follows that a progression of pictures are worth a million words.

Saving 122 Bytes of CTS codespace using TRAP Instructions
If you read and understood everything above, you already know the secret of this magic trick. All you're missing is the CTS data.

In the Directory of ZIP-File_01.zip, you'll see the file 'The_Survey_of_CTS_Instructions_used.lst'. Inside is a block listing all the lines of code that have a CALL instruction. CALLs were sub-sorted by subroutine names, each sub-group members counted and the the sub-groups were arranged by highest count to lowest count. That is the best order for maximizing TRAP 6:23 as replacement for CALL instructions.

Below I post that CALL group in a condensed version:
NOTE: There are two ways to calculate the savings; Long version is better for seeing the difference and short version is best for quickly getting the savings:
long version: Compute for CALL and for TRAP then subtract for the SAVINGS: 17x3=51 vs 17x1+2=19 Saves 51-19=32 Bytes
short version: Compute for the SAVINGS: 17x2-2=32 Bytes (this wil be used below)

```
------ CALL ------------------------------------------------------------------------------------------------
```

```
 858:  PATNLT  CALL    @GFLAGS
 864:  PAT1MC  CALL    @GFLAGS
 870:         CALL    @GFLAGS
 875:  PAT2MV  CALL    @GFLAGS
 881:         CALL    @GFLAGS
 887:         CALL    @GFLAGS
 892:  PAT0MC  CALL    @GFLAGS
 898:         CALL    @GFLAGS
 906:  PATSUF  CALL    @GFLAGS
 998:  PATVOW  CALL    @GFLAGS
1004:         CALL    @GFLAGS
1010:  PATVOC  CALL    @GFLAGS
1018:  PAT1CO  CALL    @GFLAGS
1023:  PATFVO  CALL    @GFLAGS
1028:  PATBVO  CALL    @GFLAGS
1033:  PATSIB  CALL    @GFLAGS
1039:  PATPLU  CALL    @GFLAGS TRAP  6, (x17) Saves 17x2-2=32Bytes
 585:  LF3F4   CALL    @FETCH  TRAP  7, (x14) Saves 14x2-2=26B
 835:  LF58C   CALL    @GNEXT  TRAP  8, (x 8) Saves  8x2-2=14B
 321:         CALL    @DECR3  TRAP  9, (x 6) Saves  6x2-2=10B
 352:  STOCHR  CALL    @RWBUFR TRAP 10, (x 5) Saves  5x2-2= 8B
 584:         CALL    @DECR17 TRAP 11, (x 3) Saves  3x2-2= 4B
 274:         CALL    @ENINT  TRAP 12, (x 3) Saves  3x2-2= 4B
 203:         CALL    @INCR3  TRAP 13, (x 3) Saves  3x2-2= 4B
 605:         CALL    @RSEEKB TRAP 14, (x 3) Saves  3x2-2= 4B
 953:  LF64E   CALL    @CHKLTR TRAP 15, (x 2) Saves  2x2-2= 2B
 612:         CALL    @CHKPAT TRAP 16, (x 2) Saves  2x2-2= 2B
 155:         CALL    @INIPTR TRAP 17, (x 2) Saves  2x2-2= 2B
1072:         CALL    @ROLINP TRAP 18, (x 2) Saves  2x2-2= 2B
 392:         CALL    @ROLR13 TRAP 19, (x 2) Saves  2x2-2= 2B
 593:  LF403   CALL    @SELRUL TRAP 20, (x 2) Saves  2x2-2= 2B
 249:         CALL    @STINPB TRAP 21, (x 2) Saves  2x2-2= 2B
 901:  LF5FB   CALL    @UNGET  TRAP 22, (x 2) Saves  2x2-2= 2B
                     TRAP 23 is not used.       ---
                                                 122B total Saved bytes of codespace
 156:         CALL    x1@SAYOK   <-- 'x1' = no advantage 3B either way
            CALL    x1 ...     <-- 'x1' = no advantage 3B either way
```

Questions about the 'TRAP n' instruction
(1) Does the FFD0-FFFF block need to be reserved for that use? NO. In fact the CTS code only uses FFF6-FFFF. The rest is filled with data tables.

(2) What happens if the wrong RETURN instruction is used? The answer is, the stack will become unbalanced; i.e. the number of PUSH and POP instructions are out of balance. Using the wrong

return instruction means the execution put a kink in the PUSH-POP stack balance, by either one extra, or one less POP operation. When you start to recover data of the stack at the kink position, you'll get the wrong data for every register and chaos follows. The Data Manuals don't differentiate between TRAP 0:5 and TRAP 6:23. Its an unfortunate mistake.

Scenario: A tech writer sees that his engineering notes says "TRAP 6:23" in regard to TRAP instruction description. Wondering why 6 to 23, so the tech writer asks the engineer the wrong question, "How many TRAP instructions are there?" The engineer would answer "0-23" and the right answer to the wrong question makes a Data Manual error.

I think I read a warning somewhere among the three versions of Data Manuals. If I find it, I'll update this reply.

(3) Are the TRAP instructions faster can the CALL instructions? No, they both execute in 14Tc time. TRAP instructions save codespace only. If you substitute a 3 byte 'CALL Subroutine' with a 1 byte TRAP n with its 2 byte vector entry, both require 3 bytes. Like the CTS example above, you get a codespace savings when the subroutine is CALLed from many locations in codespace. The TRAP 6 replaced 17 3-byte CALL instructions and saved 32 bytes of codespace.

(4) Did your boss get masked-ROM 1K as he budgeted? NO. It was a 2K Masked ROM fee. The code was about 1.6K+ and TI found 4 bytes that could be recovered from codespace.

Next Time...
CTS Buffers: STINPB | STore char in INP Buffer

Coming Soon...
A look inside a Mattel Electronics, "IntelliVoice, Voice Synthesis Module."

File Attachments

1)
03--RPI--RB--Recovering_Bytes_by_using_TRAP_instead_of_CALL.png, downloaded 611 times

---

Subject: CTS Buffers: STINPB (STore in INPut Buffer) PART 1
Posted by jayindallas on Wed, 06 Nov 2024 14:11:13 GMT
View Forum Message <> Reply to Message

DISCUSSION: Diving Deep into the CTS256A-AL2 Firmware

UPDATE: FRI Nov 8 11AM: I think I figured out the paradox of the Question-Mark, among 7 symbol & punctuation characters, that seemed NOT to be buffered. I think they do, by passing through all of the 8 Switch:Cases unaffected and then simply drop through into DELIMT without any hint. If that confirms to be true, I'll incorporate that into my replacement STINPB code and

update this message next week.

CTS Buffers: PART 1: STINPB CODE
CONTENTS:
1). Why STINPB?
2). The Default Size of the CTS Input and Output Buffer.
3). Introduction to Subroutine, STINPB.
4). Walk-Through the STINPB Subroutine.
5), How ASCII Characters are Designated by STINPB.
6). Documentation Error on JPZ, "Jump on Positive or Zero."

1). Why STINPB?
I didn't have a plan to follow-up from the SAYOK routine to the unknown STINPB subroutine mentioned in that prior code-block; It just happened because I was looking for more information about the buffers and the way they were parsed when converting from text to allophones. I searched for "input buffer" references in the 2022-Project's added code comments... That led to STINPB. The comments into its Switch:Case logic tapered into uncertainty, and I can understand why... the conditional jumps on the TMS7000 are strange. I decided to update the comments by doing the work to map it out, as tedious as it was.

My purpose for looking was because I had seen that the Text-to-Allophone algorithm seemed to maintain ASCII characters as ASCII characters in the buffer. Considering that the data tables were all index-based for instruction operands in the @Label(B) syntax, it seemed a natural advantage to encode the ASCII characters into a SET.INDEX byte format, instead. That would allow taking the buffered encoded-character and use the high 3-bits as a set designator and the lower 5-bits as a INDEX from 0:31 for 32 possible characters in each SET. I assumed that a lot of ASCII characters would be useless, especially the control characters and some odd symbols that never needed to be voiced.

When looking through the 8 stages of CMP %>xx,A|JNZ LABEL, it seemed like the SET bits would allow buffered letters, unused letters, and editing-commands to be quickly acted upon in large groups, as opposed to the top of the 8 stages being searches about a single character. The CTS code put the biggest block of letters about the bottom of the SWITCH:CASE structure, meaning the most numerous SET of characters that needed to be buffered would have to be delayed the maximum amount of time before they were recognized and buffered. Very bad efficiency.

Thinking of SET bits being (b7,b6,b5) I figured the text-to-voice characters like A-Z, a-z, 0-9, and common symbols and punctuation, would be in four SET using binary 0xx, and special characters would be mapped into 1xx SETs. The code didn't look like the SWITCH:CASE was using the ASCII character set the way I assumed; so I had to give-up or commit to a tedious analysis of its comparison+conditional jumps definition of a set of ASCII characters. I worked on some TMS7000 CPU in a few designs back around 86-89, but that didn't help me fly through the conditional jumps with ease. I did it by making a ASCII table in a spreadsheet, and reading the description of the conditional jumps and the flags and flag pattern oddities.

I worked through it and created a map of the ASCII characters and how they were used, but it

didn't make a lot of sense. The control characters were all buffered as delimiters (that would be good if they all were encoded into 1 or so generic delimiters) but the delimited characters were buffered with simply adding B7=1 instead of 0. I can't say yet if the CTS code did that for a good reason, so I have to wait until get through more code, to answer that question.

Besides most or all the control-characters NOT being stripped-out of the buffering, I also found it odd that the Question Mark was not buffered. Voice for a question is often sounded differently. The oddity seemed to be a mistake on my Switch:Case code interpretation. Next I tried doing it a more methodical way, relying on a spreadsheet to simulate the flags and conditional jumps so I could test and be sure how the conditional jumps made groups of ASCII characters to be buffered. I got the same answer using that and it took awhile to get all the mistakes out of the flag rule behavior and then the conditional jump behavior and copy those formulas over to a block were I could insert a value for each of the 8 CASES with their assortment of various conditional jumps, to experimentally find out which pattern created a TRUE or FALSE. I've got the same answer doing it that way, but I'm still not confident for now that I've got it right. I'll post what I mapped out and I'll come back and look at it later and see if I can find a mistake.

Part of MURPHY'S LAW residing in the spreadsheet was caused by an error I found in the 1986 & 1989 version of the TMS7000 DATA MANUAL. The 1983 version had it correct. I describe the error and correction in PART 1, Section 7 at the bottom of this page. The mistake turned a JPZ (jump on positive or zero) into a JZ (Jump on Z, only).

As my purpose was to find out which ASCII characters are used and not used, buffered and not buffered and which invoke special editing on the buffer, I'll describe how the ASCII character set is used, after the STINPB subroutine walk-through.

2). The Default Size of the CTS Input and Output Buffer:
The input and output buffer defaults to internal RAM. The stack pointer is loaded as 03Ah, leaving 21 bytes of stack space before it would collides with the input buffer. According to the survey of internal RAM use, Address 3A is an unused byte of RAM; CTS coders probably thought the stack address loaded the first byte in it... but the PUSH instruction shows that a Stack pointer is incremented before loading the first value. So setting the Stack pointer to 39h would be fine because it would actually use the byte at 03A first, giving 22 bytes of Stack space. The first byte of the Input Buffer is then the 23rd byte from the Stack initialization improvement, i.e. using 39 instead of 3A so that 3A holds the first stack value.

The Input and Output Buffer that follows the allocated Stack space uses every RAM byte until its ending address at 07Fh. To increase buffer size, the required stack needs to be reduced, or/and any internal RAM use needs to be reduced. The latter would be relatively easy with all the CTS code's use of 16-bit pointers where 8-bit would suffice. A TMS70C42 has 256 bytes of internal RAM which could be used to expand the buffer sizes.

CTS INPUT BUFFER:

```
        |<---- Internal RAM ---------- INPUT BUFFER ---------- Internal RAM ---->|
Address:
|051|052|053|054|055|056|057|058|059|05A|05B|05C|05D|05E|05F|060|061|062|063|064|065|
Count:   | 01  02  03  04  05  06  07  08  09  10  11  12  13  14  15  16  17  18  19  20  21|
```

|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|

CTS OUTPUT BUFFER:
|<---- Internal RAM -------------- OUTPUT BUFFER ------------------------------------ Internal RAM ---->|
|066|067|068|069|06A|06B|06C|06D|06E|06F|070|071|072|073|074|075|076|077|078|079|08A|07B|07C|07D|07E|07F|
| 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26|
|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|

3). Introduction to Subroutine STINPB:
In the first article about the CTS code, we saw some easy code to make the SP0256 say "O-K". One of the unknowns in that code was CALL STINPB, which is the subject of this article.

Subroutine STINPB's task is to take received ASCII characters, from the upstream serial or parallel port, and to put most of the characters into the input buffer. Some ASCII codes are not placed into the buffer because they are used to issue buffer-editing commands, like (1) Backspace, (2) Clear Buffer, (3) Clear back to last delimiter. They only activate a buffer modifications and are not part of the text that is converted to voice. STINPB does these tasks and executes a CALL RWBUFR to put an immediate character into the buffer.

STINPB is called with newly received ASCII character in Register A. Most of the STINPB code-block is used to recognize single ASCII characters or groups of characters to handle them appropriately. This structure is similar to a C/C++ Switch:Case structure, where a lot of code uses CMP & Conditional-Jumps to route the characters eventually to their correct code-block for action.

The CTS code, in general, maintains all buffered ASCII characters in ASCII form; seems reasonable but not necessarily efficient. When the CTS code (not in STINPB) needs to access information quickly and efficiently, it converts a letter into an 'index' in Register B so it can use "index addressed mode" instructions with their operand syntax: @LABEL(B). For example, to get a single byte of bit-mapped phonetic attribue flags of any letter, A to Z, it encodes the ASCII character into a zero-relative index where "A"=00h and "Z"=19h and places that in Register B. A "STA #LABEL(B)" quickly fetches the letter's 1 byte phonetic attributes without pretzel logic. Index form is also be used to jump to code for processing any letter rule sets, with a BR @LABEL(B). This will be a subject in "PART 2: IMPROVING STINPB."

The STINPB subroutine code-block includes LINE#279->365. The following breaks that down to small code-blocks with a text header describing the activity in the code-block. The listings following that are minimized for web viewing. You can view the full listing with all fields by extracting "The_Reference_CTS_ASM_Listing.lst" from the ZIP-File-01.zip in the second message of this discussion.

4). Walk-Through of the STINPB Code-Block:
Code-Block A: LINES#279->283: Four PUSH instructions save internal RAM registers that need to be restored before returning from this subroutine. LINE#283 clears 2-bits of a buffer control

flags in R10, to select input mode, when/if subroutine RWBUFR is called.
STINPB Subroutine Initializes.
; 279: STINPB PUSH B
; 280: PUSH R10
; 281: PUSH R12
; 282: PUSH R13
; 283: AND %>F9,R10

Code-Block B: LINES#284->294: This code-block looks for an 'ESC' (escape) character to make the CTS clear the buffer. All other 127 ASCII characters executing only this CMP & JNZ, jump to the next code-block, CASE#2. LINE#287->294 re-initialize the buffer and stack and then jumps to a polling routine to await the next upstream character received by the serial or parallel port.
Switch:Case #1: Does Reg A contain an 'ESC' Character?
; 284: CMP %>1B,A ;## CASE #1: Test for an ASCII 'ESC' character
; 285: JNZ NOTESC ;## If not, jump to CASE#2. CASE-finding delay through Case 1: 14Tc (CMP@7Tc+JNZ@7Tc)
; 287: ANDP %>FE,P0
; 288: CALL @INIPTR
; 289: MOV %>3A,B
; 290: LDSP
; 291: MOVD R45,R27
; 292: STA *R27
; 293: CALL @ENINT
; 294: BR @POLL
Status: 1 new character done, 1 total processed, 127 still unprocessed and 14Tc total delay through CASE#1 to CASE#2.

Code-Block C: LINES#297->308: This code-block looks for an ASCII Control-R (^R) character to make the CTS clear the buffer back to the last delimiter position saved in a 16-bit pointer. All other 126 ASCII characters executing only this CMP & JNZ, jump to the next code-block, CASE#3. LINE#308 is the final branch to XSTINP, restoring the four registers that were PUSHed upon entry to the STINPB subroutine and then returns to the code that called STINPB.
Switch:Case #2: Does Reg A contain a Control-R Character?
; 297: NOTESC CMP %>12,A ;## CASE #2: Test for an ASCII Control-R character
; 298: JNZ NOTCTR ;## If not, jump to CASE#3. CASE-finding delay through Case 2: 14Tc (CMP@7Tc+JNZ@7Tc)
; 300: BTJO %>01,R11,LF21E
; 301: SUB R25,R3
; 302: SBB R24,R2
; 303: SUB R3,R52
; 304: SBB R2,R51
; 305: MOVD R25,R3
; 306: MOV %>01,R57
; 308: LF21E BR @XSTINP
Status: 1 new character done, 2 total processed, 126 still unprocessed and 28Tc (14Tc+14Tc) total delay through CASE#1 to CASE#3.

Code-Block D: LINES#310->328: This code-block looks for an ASCII Backspace character to

make the CTS remove the last character in the buffer and adjust the pointers. 125 characters are still being delayed until they are recognized at the CASE# that includes them.
Switch:Case #3: Does Reg A contain a Backspace Character?
; 310: NOTCTR CMP %>08,A ;## CASE #3: Test for an ASCII Backspace, BS
; 311: JNZ NOTBKS ;## If not, jump to CASE#4. CASE-finding delay through Case 3: 14Tc (CMP@7Tc+JNZ@7Tc)
; 313: CMP R2,R4
; 314: JNZ LF22F
; 315: CMP R3,R5
; 316: JZ XSTINP
; 318: LF22F PUSH R3
; 319: PUSH R2
; 320: MOVD R5,R3
; 321: CALL @DECR3
; 322: MOVD R3,R5
; 323: POP R2
; 324: POP R3
; 325: INC R52
; 326: JNC XSTINP
; 327: INC R51
; 328: JMP XSTINP
Status: 1 new character done, 3 total processed, 125 still unprocessed and 42Tc (28Tc+14Tc) total through CASE#1 to CASE#4.

Code-Block E: LINES#330->331: This code-block tests for a ASCII apostrophe and jump immediately to STOCHR to store that character into the buffer if detected, else drops through to CASE#5. 124 characters are still being delayed until they are recognized at the CASE# that includes them. It would be wiser to prioritize big sets of letter or text sets of characters rather than these specific characters. More will be stated about that in that summary section.
Switch:Case #4: Does Reg A contain an apostrophe Character?
; 330: NOTBKS CMP %>27,A ;## CASE #4: Test for an ASCII apostrphe '
; 331: JZ STOCHR ;If not, jump to CASE#5. CASE-finding delay through Case 4: 14Tc (CMP@7Tc+JNZ@7Tc)
Status: 1 new character done, 4 total processed, 124 still unprocessed and 54Tc (42Tc+12Tc) total through CASE#1 to CASE#5.

Code-Block F: LINES#332->333: This code-block tests for the 4 ASCII characters | } ~ or 'DEL' and those will immediately be processed by the DELIMT routine. 120 characters are still being delayed until they are recognized at the CASE# that includes them.
Switch:Case #5: Does Reg A contain a ASCII character higher than "{" : | } ~ or 'DEL' ?
; 332: CMP %>7B,A ;## CASE #5: Test for ASCII " | } ~ del " aka ASCII higher than "{"
; 333: JP DELIMT ;## If not, jump to CASE#6. CASE-finding delay through Case 5: 12Tc (CMP@7Tc+DropThru@5Tc)
Status: 4 new character done, 8 total processed, 120 still unprocessed and 66Tc (54Tc+12Tc} total through CASE#1 to CASE#6.

Code-Block G: LINES#336->337: This code-block tests for ASCII characters lower than "0." That is the set of characters from 00h to 2Fh'. Thats a block of 48 ASCII characters, the 32

control-characters and 16 characters from 'space' to "/" but four characters in this block have already been processed as buffer-editing commands and buffered character. So the new characters processed by this CASE#6, is 44 (48-4) characters. This block of new ASCII characters are stored as delimiters. 75 characters are still being delayed until they are recognized at the CASE# that includes them. For now it seems wasteful to allow all these control characters into the buffer as the CTS purpose is the convert text of spoken English into synthesized voice. What purpose do that serve?
Switch:Case #6: Does Reg A contain an ASCII character lower than "0"?
; 334: CMP %>30,A ;## CASE#6: Test for ASCII lower than "0"
; 335: JN DELIMT ;## If not, drop through to CASE#7. CASE-finding delay through Case 6: 12Tc (CMP@7Tc+DropThru@5Tc)
Status: 44 new character done, 52 total processed, 76 still unprocessed and 78Tc (66Tc+12Tc) total through CASE#1 to CASE#7.

Code-Block H: LINES#336->337: This code-block tests for ASCII characters less than ":" which only adds 10 new characters:{0123456789}. These numbers will be stored in the buffer. 65 characters are still being delayed until they are recognized at the CASE# that includes them.
Switch:Casr #7: Does Reg A contain an ASCII character less than "0"?
; 336: CMP %>3A,A ;## CASE#7: Test for ASCII {0,1...9}
; 337: JN STOCHR ;## If not, drop through to CASE#8, else jump to STOCHR. CASE-finding delay through Case 7: 12Tc (CMP@7Tc+DropThru@5Tc)
Status: 10 new character done, 62 total processed, 66 still unprocessed and 90Tc (78Tc+12Tc) total through CASE#1 to CASE#8.

Code-Block I: LINES#338->339: This test for the rest of the ASCII characters, "ABC...XYZ[...`abc...xyz}" and simply stores them into the buffer. The new characters processed by this CASE are 59 in number.
Switch:Case #8: Does Reg A contain an ASCII character "A" to "{"
; 338: CMP %>41,A ;## CASE#8: Test for ASCII higher or equal to "A"
; 339: JPZ STOCHR ;## if not, drop through to DELIMT, else jump STOCHR. CASE-finding delay through Case 8: 12Tc (CMP@7Tc+DropThru@5Tc = 12Tc) to CASE#9
Status: 59 new character done, 121 total processed, 7 still unprocessed and 78Tc+12Tc = 90Tc total to CASE#8.

Code-Block J: LINES#341->350: Delimiter characters are routed to this code-block. First it checks a mode flag to see any delimiter will update the last delimiter position, 16-bit pointer that can be used with a buffer-edit command ^R to clear the buffer back to the last delimiter. If the flag is not set, then the only delimiter that updates the last delimiter position is a carriage return.

Line#351 is entered by a recognized delimiter ASCII character. If the mode flag is set, every delimiter passing through this code will update the 'last delimiter position pointer. If that mode flag is not set it drops down into a test for a carriage return in register A. If its not a <CR> it goes into the buffer with its b7 bit set high. If it is a <CR> then LINE#344 drops into LINT#344 and it sets a flag that releases the idle loop so the CTS can start processing line of text into allophone. LINES#348-357 increment the 16-bit counter of the buffer's character count. It then drops into the next code-block to execute LINE#352 to actually load the delimiter tagged character into the input buffer.
DELIMT

```
; 341: DELIMT BTJO %>01,R11,LF267
; 342:  CMP %>0D,A
; 343:  JNZ LF26A
; 344:  OR %>10,R11
; 345: LF267 MOVD R3,R25
; 347: LF26A OR %>80,A
; 348:  INC R57
; 349:  JNC STOCHR
; 350:  INC R56
```

Code-Block K: LINES#352->359: any character being sent to the buffer, be it a delimiter or a buffer character, flows into LINE#352 to write it into the buffer. LINE#353 checks the buffer full flag. If it is active the code jumps to return from the call to STINPB as the registers are restored to the values off the stack, saved upon entry to STINPB. If the buffer is not full, a carriage return delimiter is put in register A, the 16-bit buffer size is incremented for the <CR> and the carriage return is appended to the buffer.

```
STOCHR
; 352: STOCHR CALL @RWBUFR
; 353:  BTJZ %>20,R11,XSTINP
; 354:  MOV %>8D,A
; 355:  INC R57
; 356:  JNC LF281
; 357:  INC R56
; 359: LF281 CALL @RWBUFR
```

Code-Block L: LINES#361->365: This routine simple restore the registers that were saved upon entry to STINPB and then it returns to the code that called STINPB.

```
XSTINP
; 361: XSTINP POP R13
; 362:  POP R12
; 363:  POP R10
; 364:  POP B
; 365:  RETS
```

5): How ASCII Characters are Designated by STINPB:

STINPB Subroutine has 8 Switch/Case structures to deal with single ASCII characters and groups of ASCII characters. There is also a group of characters that are not used, and never get buffered (at least as it appears now; might be corrected as the resulting ASCII map of characters doesn't seem correct to me.

I'll create a CASE #0 for the unused and not buffered ASCII characters. Maybe the coders initially planned for some expansion features before finding out that was unlikely to be done for market reasons.

```
CASE #       COUNT DESCRIPTION
--------     -----    ----------------------------------------------------------------------------------------------------
CASE #0:  7 Not used and not buffered, characters : ; < = > ? @
CASE #1:  1 Buffer-Edit command, 'ESC'
```

CASE #2: 1 Buffer-Edit command, ^R (control-R, DC2)
CASE #3: 1 Buffer-Edit command, ^H (BS, Backspace)
CASE #4: 1 Buffered Character, Apostrophe
CASE #5: 4 Buffered Delimiters, Highest 4 ASCII Characters: | } ~ 'DEL' encoded as Delimiters by b7=1
CASE #6: 44 Buffered Delimiters, the lowest 48 ASCII Characters minus the 4 characters handled by CASE #1 through #4
CASE #7: 10 Buffered Characters, those not previously acted upon leave only the set "9" through "0".
CASE #8: 59 Buffered Characters, The ASCII character set from A:Z [\ ] ^ _ ` a:z {
---
128 Total

Buffered Characters: 70 |
Buffered Delimiters: 48 |___118 sub total: characters that can be in the input buffer.
Buffer-Editing Commands: 3 | 'esc' ^R BS
Not used, not Buffered: 7 | : ; < = > ? @ These are the characters between 9 and A
--- |
128 | number of ASCII characters

00h-07h Case #6 8 of 44 Buffered Delimiters
08h Case #3 1 Buffer-Edit Command, not buffered
09h-11h Case #6 9 of 44 Buffered Delimiters
12h Case #2 1 Buffer-Edit Command, not buffered
13h-1Ah Case #6 8 of 44 Buffered Delimiters
1Bh Case #1 1 Buffer-Edit Command, not buffered
1Ch-26h Case #6 11 of 44 Buffered Delimiters
27h Case #4 1 Buffered Character
28h-2Fh Case #6 8 of 44 Buffered Delimiters
30h-39h Case #7 10 Buffered Characters
3Ah-40h Case #0 7 Not used, not buffered
41h-7Bh Case #8 59 Buffered Character
7Ch-7Fh Case #5 4 Buffered Delimiters
---
128 Total

6). Documentation Error on JPZ:
The table for the conditional jump instructions in the 1986 and 1989 data manual is wrong. It was correct in the 1983 manaul. The correct defintion is on page 3-34, in TABLE 3.21. The second to the last line of the table states:

INSTRUCTION	MNEMONIC	STATUS FLAG CONDITIONS FOR JUMP		
		CARRY	NEGATIVE	ZERO
Jump if Positive	JP	x	0	0	Positive excludes negative and zero
1983: Jump if Positive or Zero	JPZ	x	0	x	Positive or Zero == not Negative
1986: Jump if Positive or Zero	JPZ	x	0	1	The '1' under Zero rules out

Positive.
1989: | Jump if Positive or Zero | JPZ | x | 0 | 1 | Error carried forware to 1989.
|_____|
I found this while making a spreadsheet mimic the TRUE/FALSE result for each of the 8 'CASE#x' conditional jumps in the SWITCH/CASE structure of the CTS subroutine STINPB, shown above. I don't see why STINPB would allow the buffering of all control characters. It allows 4 buffer-editing commands to be recognized for operations on the buffer, before throwing the control character away; aka not buffered. Maybe that some control characters are buffered to alter some of the aspects of the text-to-voice routines. Considering the size of the default input buffer, it seems wise to filter out as much unused characters as possible.

Subject: PART 2: Improving STINPB Buffer Decision Code
Posted by jayindallas on Mon, 02 Dec 2024 03:27:09 GMT
View Forum Message <> Reply to Message

Title: PART 2: Improving STINPB Buffer Decision Code

DISCUSSION: Diving Deep into the CTS256A-AL2 Firmware

PART 2: Improving STINPB Buffer Decision Code

SECTION CONTENTS:

1). The Buffer Decision Code V1.4
The CTS buffer decision code begins with the label 'BUFDEC' (BUFfer DECision) in the subroutine 'STINPB' (STore INPut Buffer). When any 7-bit ASCII character is received from the upstream text source, it is placed into register A, and the subroutine STINPB is called. BUFDEC's first task is to categorize the received character as one of three types: (1) CTS Alphanumeric characters, (2) CTS Delimiter characters and (3) CTS Command characters.

Once the character is categorized, the code directs the character to one of five service routines within STINPB: STOCHR, DELIMT, EDIT1, EDIT2 or EDIT3. Alphanumeric characters (letters, numbers and the apostrophe) are sent to service routine STOCHR, to store the character into the text input buffer for subsequent text-to-voice processing. Delimiter characters (symbols,

punctuation, special ASCII codes and control characters) are sent to service routine DELIMT, which sets bit 7 high for quick recognition, then updates a single buffer pointer to the last delimiter in the buffer and finally drops into STOCHR to add the modified delimiter character into buffer for subsequent text-to-voice processing. The three Command characters are keyboard buffer editing commands: Backspace, Escape and control-R.), are never put in the input buffer; they each have their own service routines, EDIT1, EDIT2 or EDIT3, that clears all or a portion of the buffer and exits the STINPB routine.

In the ASCII table of 128 characters, like-characters of the same CTS category make the following 15 neighboring blocks:

```
; _____ _____ _____ _____ _____ _____ _____ _____ _____ _____ _____ _____ _____ _____
_____
;|BLK01|BLK02|BLK03|BLK04|BLK05|BLK06|BLK07|BLK08|BLK09|BLK10|BLK11|BLK12|BLK13|
BLK14|BLK15| <-- Block number 01:15
;| Del | Cmd | Del | Cmd | Del | Cmd | Del | Alp | Del | Alp | Del | Alp | Del | Alp | Del | <-- CTS
category
;|  8  |  1  |  9  |  1  |  8  |  1  | 11  |  1  |  8  | 10  |  7  | 26  |  6  | 26  |  5  | <-- Block member count
;|00 to| 08h |09 to| 12h |13 to| 1Bh |1C to| 27h |28 to|30 to|3A t0|41 t0|5B t0|61 t0|7B to| <-- First
ASCII character of the block
;|  07h|     |  11h|     |  1Ah|     |  26h|     |  2Fh|  39h|  40h|  5Ah|  60h|  7Ah|  7Fh| <-- Last ASCII
character of the block
;|_____|_____|_____|_____|_____|_____|_____|_____|_____|_____|_____|_____|_____|_____|
_____|
```

However, the original CTS version used only 13 blocks, by combining uppercase and lowercase letters with the 6 delimiter-like characters between them as one block. The result of this "13 blocks" is simply that the BLK12, BLK13 and BLK14 are directed to their service routine, STOCHR, from the same branch-decision and thus all at the same arrival time. The combined BLK would contain the first character 41h to last character 7Ah. When blocks are combined in one branch-decision, they'll be listed as like "BLK{12,13,14}" and the arrival time assigned to each block combined. This allows the spreadsheet calculates the performance for "13 Block" or "15 Block" definitions. New versions of BUFDEC code will use 15 Blocks, but new 'replacement' code will use the same 13 blocks that the original CTS code used. The performance metrics table list whether it uses 13 Blocks or 15 Blocks.

Section 2 of this article defines the performance-metrics that allow performance comparisons between different versions of BUFDEC. A key component in those calculations are the CPU time to process any of the 128 ASCII character from BUFDEC to the CPU's arrival at any of the five above-mentioned service routines. The BUFDEC code listing are a derivative "decision-format" to make the calculations and decision flow easier to measure and understand. This requires adding a comment line that tag the arrival time when a branch-decision directs to a service routine; something that would not appear in a typical code listing.

When a decision drops into its service routine, that service routine is omitted from the code and replaced with a comment line specifying the arrival time at the service routine. The comment line describes that a drop-through to the service routine was used. Example branch-decision 01-03: 'br'= branch#, 'de'=decision#, 'tn'= timer before line executes, 'dt'= timer Tc when drop-into, 'jp'= timer Tc when jump-to

```
 _____ _______ ________ ____________________________________
```

```
_____________________________________
|br-de|-tnnn-|b-dt-jp|
|TREE |TIMER
|EXETIME|_____________________________________________________________________
|_____|______|_______|; Original CTS buffer decision code
|01-03|->t28 |2-07   | NOTCTR CMP %>08,A  ; BLK02: if 08h command key, drop to EDIT1
|   03|t35 ->|2-05-07|  JNZ NOTBKS  ; else jump to branch-decision 01-04
|___03|t40___|drop_in|;_EDIT1___________DECISION 01-03 drops into service routine EDIT1 to
process BLK02 after 40Tc
```

In the code lines above there are only two lines of real code. The third line captures the data for the performance metric, defined in the next section. When a drop-through into the service routine is used, the conditional jump is used to skip to the next decision.

This construct is used three times in the original CTS BUFDEC code, but it is less efficient than a construct that jumps to the service routines, instead. The reason is that the drop through takes 5Tc to execute and a jump takes 7Tc; dropping through into the next decision accumulates 5Tc per decision whereas jumping to the next decision requires 7Tc each. A new version of the original CTS is the first variant version shown and it re-prioritizes the decision order so that the most common characters in English text is processed first, so that the ASCII letters used in speech generation is buffered quickly, and it incorporates the "Only Jump To Service Routines" construct to minimize the accumulating decision sequence delays.

When a decision jumps to its service routine, that service routine is also omitted from the code and replaced with a comment line specifying the arrival time at the 'jumped-to" service routine. The comment line describes that a jump to the service routine was used. Example branch-decision 01-04:
'br'= branch#, 'de'=decision#, 'tn'= timer before line executes, 'dt'= timer Tc when drop-into, 'jp'= timer Tc when jump-to

```
 _____ ______ _______ ____________________________________
_____________________________________
|br-de|-tnnn-|b-dt-jp|
|TREE |TIMER
|EXETIME|_____________________________________________________________________
|_____|______|_______|; Original CTS buffer decision code
|01-04|->t42 |2-07   | NOTBKS  CMP %>27,A  ; BLK08: if 27h apostrophe, jump to STOCHR
|   04|t49 ->|2-05-07|  JZ STOCHR  ; else drop to branch-decision 01-05
|___04|->t56_|jump_to|;_STOCHR__________DECISION 01-04 drops into service routine
STOCHR to process BLK08 after 56Tc
```

The second line above shows "|2-05-07|" in the worksheet block, meaning the line takes 2bytes, 5Tc to drop to the next line of code, or 7Tc to make a conditional jump. The third line differs slightly than the previous example, in that has a "|->t56_|" in the second column of the worksheet block. It includes an arrow pointing to the arrival time to suggest that it got there by a jump. In contrast the previous example had a "|t40___|" suggested that its executing the next line down (drop-through) and it gets there at timer value t40, meaning 40Tc.

The reason these constructs are used is that it helps to calculate the timing of arrivals at various service routines through the segments of code. Without this construct its less easy to calculate and its more likely to result in human errors.

The original CTS buffer decision code is listed next in decision-format. After that a list of what the next three sections will cover.
'br'= branch#, 'de'=decision#, 'tn'= timer before line executes, 'dt'= timer Tc when drop-into, 'jp'= timer Tc when jump-to

_____ _____ _____ _____
_____
|br-de|-tnnn-|b-dt-jp|
|TREE |TIMER
|EXETIME|_____
|_____|_____|_____|; Original CTS buffer decision code with BUG-FIX on DECISION 01-05: "JP DELIMT" edited to "JPZ DELIMT"
|01-01|t0 |2-07 | BUFDEC CMP %>1B,A ; BLK06: if 1Bh command key, drop to EDIT3
| 01|t7 ->|2-05-07| JNZ NOTESC ; else jump to branch-decision 01-02
|___01|t12___|drop_in|;_EDIT3_____DECISION 01-01 drops into service routine EDIT3 to process BLK06 after 12Tc
|01-02|->t14 |2-07 | NOTESC CMP %>12,A ; BLK04: if 12h command key, drop to EDIT2
| 02|t21 ->|2-05-07| JNZ NOTCTR ; else jump to branch-decision 01-03
|___02|t026__|drop_in|;_EDIT2_____DECISION 01-02 drops into service routine EDIT2 to process BLK04 after 26Tc
|01-03|->t28 |2-07 | NOTCTR CMP %>08,A ; BLK02: if 08h command key, drop to EDIT1
| 03|t35 ->|2-05-07| JNZ NOTBKS ; else jump to branch-decision 01-04
|___03|t40___|drop_in|;_EDIT1_____DECISION 01-03 drops into service routine EDIT1 to process BLK02 after 40Tc
|01-04|->t42 |2-07 | NOTBKS CMP %>27,A ; BLK08: if 27h apostrophe, jump to STOCHR
| 04|t49 ->|2-05-07| JZ STOCHR ; else drop to branch-decision 01-05
|___04|->t56_|jump_to|;_STOCHR_____DECISION 01-04 drops into service routine STOCHR to process BLK08 after 56Tc
|01-05|t54 |2-07 | CMP %>7B,A ; BLK15: if 7B:7Fh delimiters, jump to DELIMT *** CTS-BUG: does only 4 not 5
| 05|t61 ->|2-05-07| JPZ DELIMT ; else drop to branch-decision 01-06 *** CTS-BUG: FIXED: JPZ DELIMT, not JP
|___05|->t68_|jump_to|;_DELIMT_____DECISION 01-05 jumps into service routine DELIMT to process BLK15 after 68Tc
|01-06|t66 |2-07 | CMP %>30,A ; BLK{01,03,05,07,09}: if 00:2Fh delimiters, jump to DELIMT
| 06|t73 ->|2-05-07| JN DELIMT ; else drop to branch-decision 01-07
|___06|->t80_|jump_to|;_DELIMT_____DECISION 01-06 jumps into service routine DELIMT to process BLK{01,03,05,07,09} after 80Tc
|01-07|t78 |2-07 | CMP %>3A,A ; BLK10: if 30:39h alphanumeric, jump to STOCHR
| 07|t85 ->|2-05-07| JN STOCHR ; else drop to branch-decision 01-08
|___07|->t92_|jump_to|;_STOCHR_____DECISION 01-07 jumps into service routine STOCHR to process BLK10 after 92Tc
|01-08|t90 |2-07 | CMP %>41,A ; BLK{12,13,14}, if 41:7Ah mixed, jump to STOCHR
| 08|t97 ->|2-05-07| JPZ STOCHR ; else drop to branch-decision 01-09 with 3A:40h delimiters
|___08|->t104|jump_to|;_STOCHR_____DECISION 01-08 jumps into service routine STOCHR to process BLK{12,13,14} after 104Tc

|01-09|t102__|drop_to|;_DELIMT_____DECISION 01-09 drops into service routine DELIMT to process BLK11 after 102Tc
|_____|_____|_____|_____

The inefficiency of this original CTS is apparent in the first four branch-decisions 01-01 though 01-04. By the time the fifth branch-decision begins, the timer has measured 56T(c) and only tested for four characters of the 128 ASCII character set. The remaining five branch-decisions have already accumulated that 56T(tc) delay in their own path to their service routines. The original CTS algorithm-method used is indeterminant, in that it doesn't use the given character to jump to a shorter branch that would more quickly find its service routine. Code that uses the given character to more closely jump to its category recognition branch-decision would be more determinant and efficient, but may consume more code space; there are signs in the original CTS code that suggest that codespace was very tight.

This is why an effective performance-metrics table is needed to compare buffer decision code variants, to identify algorithms that may be worth exploring or rejecting. A previous article about the 'TRAP n' instruction recovered about 128 bytes of code-space, so it is feasible to write new code that consumes some of that recovered code-space. It would be wise preserve most of those bytes for other improvements. This situation is why I decided to dig deeper into the STINPB subroutine and making it more efficient... it can be a learning exercise on coding the TMS7000 series CPU and see the tradeoffs in its instruction set; what it does well and what it does poorly.

The application-note regarding the CTS and SP0, offers a work-around by slowing the baud rate to avoid losing text characters. This suggests that improvements to the buffering of upstream input text characters, and improvements to speed up the text-to-voice processing, would likely reduce or fix the AP-Note problems. A Version 2 CTS code, if ever written, should plan to fix that problem.

COMING UP IN THE NEXT SECTIONS BELOW:
Section 2, "Performance-Metrics for STINPB Buffer Decision Code" will define the performance metrics used to compare various BUFDEC versions.
Section 3, "Algorithm Method 0: The CTS Original: Parsing ASCII into CTS Categories," will define the method used to categorize any of the 128 ASCII characters that may be received from the upstream text portal. Note that the original code is not very efficient in regard to quickly buffering the most used or common ASCII text characters.
Section 4, "Algorithm Method 1: Re-prioritizing Original Buffer Decisions" improves the CTS original BUFDEC significantly and uses the performance metrics tables to confirm it and its a new version that fits in the same codespace. That confirms that significant improvements can be made.

2). Performance Metrics for STINPB Buffer Decision Code
STINPB defines all ASCII codes as one of three categories: (1) Alphanumeric which are buffered immediately, (2) Delimiters which are buffered after being tagged with bit7 and updating a last-delimiter pointer, then (3) Commands that are never buffered and instead invoke a specific service routine to edit the buffer and return from STINPB.

In the ASCII table of 128 characters, like-characters of the same category make 15 neighboring subgroups, though the CTS versions used only 13 subgroups by adding uppercase and lowercase

letters with the 6 delimiter-like characters between them. For a yet unknown reason, CTS combined subgroups #12, #13 and #14. Deeper code will reveal its purpose, or its error; until then, most of the new versions of code will deal with all 15 subgroups, except for the re-prioritization of the original CTS buffer decision code. As that's modifying code in-place, it uses the same 13 subgroups. A version doing all 15 subgroups re-prioritization might be coded, given enough time.

The rest of the example versions of decision code will have to perform with the 15 subgroups by separating the 6 delimiters between upper and lowercase letters, as a necessary threshold for 'improvement'. The image below defines the 15 subgroups:

A performance metric is needed to compare versions of decision code to the original CTS buffer decision code in STINPB. As there is no data on the percentage of use among the three CTS ASCII categories, the performance metric will be neutral on that, but it will provide metrics on each category, should anyone want to weight the values of the categories. However, it is obvious that the Alphanumeric (text letters, numbers and the apostrophe, minus other punctuation marks) are the majority of text; open any book and you'll see mostly letters and spaces (space would be a valid priority character to factor). It follows that quickly processing letters at a higher priority will yield faster performance with real world CTS text-to-voice applications. A common sense priority among the three categories as numbered above: (1) Alphanumeric are the highest priority ASCII characters to process, (2) Delimiters are the mid-range priority and lastly (3) Commands are the most infrequent and thus the least priority characters and they are few.

The performance metric will include a value assigned to each category for total Tc time units it take to process every character in that category. That will be the sum of the categories ASCII subgroup's Tc time multiplied by the number of members in the subgroup. An overall average of subgroup Tc time is a useful metric for example comparison. The numbers allow weighting the categories by personal opinion by multiplying it by a personal chosen weighting values.

The unit of time for the performance metric will be Tc, as used in the TMS 7000 Series manual and listed in all the instruction tables. Anyone can convert that to other units of time if they know the crystal frequency and the internal divide-by-2 or divide-by-4 mask-option of their CPU. Other parameters of the performance metric report will include the number of code byte, data bytes, and maybe stack and internal RAM use too (but not for PART 2).

3). Method 1: Parsing ASCII into CTS Categories
The Butter-Stick Analogy: Below is a diagram of the ASCII code table starting from 00h on the left side, to 7Fh on the right side. When there are no critical time constraints, this a good way to start. Butter as a metaphor works particular well in this basic example, because butter is usually cut from one side and the portion cut off is then extracted from that side. When one piece is removed, and another is about to be removed, the prior extracted piece has no bearing on the secondary cut; it has been processed and its gone from the problem. That applies to the coding aspect too.
ASCII 00h ____ ____ ____ ____ ____ ____ ____ ____ ____ ____ ____ ____ ____ ____ ____ ____
ASCII 7Fh
Category: |Del |Com |Del |Com |Del |Com |Del |Alp |Del |Alp |Del |Alp |Del |Alp |Del |
By Letter:| D | C | D | C | D | C | D | A | D | A | D | A | D | A | D |

```
 How many:|  8 |  1 |  9 |  1 |  8 |  1 | 11 |  1 |  8 | 10 |  7 | 26 |  6 | 26 |  5 |
 Start at:|00h | 08h|09h | 12h|13h | 1Bh|1Ch | 27h|28h |30h |3Ah |41h |5Bh |61h |7Bh |
 Ends  at:| 07h|    | 11h|    | 1Ah|    | 26h|    | 2Fh| 39h| 40h| 5Ah| 60h| 7Ah| 7Fh|
Subgroup#:| #1 | #2 | #3 | #4 | #5 | #6 | #7 | #8 | #9 |#10 |#11 |#12 |#13 |#14 |#15 |
          |____|____|____|____|____|____|____|____|____|____|____|____|____|____|____|
```

Cutting Butter with Code: If block #14 was a priority because it contains the 26 lowercase letters, the most common character set used in text, this simple method would first make a cut at the boundary line between blocks #14 and #15. The "cut" in code is a 'compare and conditional jump' ("CMP+JMP") to test if a member of Block #15 is in Reg A; if not then the CPU drops through to the next test. Knowing that Reg A is not in block #15, that block is known to be empty in regard to the contents of Reg A.; Thus Block #15 is 'conceptually' removed from the diagram like a piece of butter cut and removed from a 'stick of butter'. A block next to the open end can be tested with a single 'CMP+JMP' using a greater-than or less-than conditional conditional jump; that means a single 'cut' is efficiently all that is needed, no left and right bound testing is required and this approach is inherently efficient.

Notching to Make Groups of Subgroups: The CTS code starts off by 'removing' the four single character 'notches' blocks #2, #4, #6, and #8. If none of those 4 ASCII characters are detected, then their subgroup is conceptually empty leaving a large group of delimiter subgroups #1, #3, #5, #7, and #9. That Group only require one compare and conditional jump if the character in Reg A is in that group, it jump to DELIMT for processing the character in Reg A. This is one of the 'tricks' to reduce the number of category decision. CTS reduced the 15-set categories down to 9 decisions, 8 'CMP+JMP' + the 1 final drop-through.

To 'cut' that group of 44 delimiters requires only one code 'decision-pair': "CMP %>30,A" & "JN DELIMT." That decision can identify 44 delimiters and route them to DELIMT; much better than using 5 decision pairs by processing them individually. Note that its efficiency is based upon a strategic order of parsing the categories of the CTS use of ASCII characters.

Unfortunately the four 'notch-removals' accumulated a lot of delay. That's the nature of coding these situations. If there is no critical timing, the delays don't matter. If it needs to be faster, different solutions need to be devised.

Cut from Either End: That description above gets to all the core factors of the "Parsing ASCII" method. Subsequent concepts can be explained with short text now, using the foundation above. The next enhancement to the basic method is taking the choice to cut off either end of the abstraction stick of butter. In code its no problem at all as long as you get your compare and conditional jump correct. The TMS 7000 Series conditional jump flags scenarios can be confusing and some of their documentation is WRONG.

I added a note to PART 1, that 1986 and 1989 manual is in ERROR on the JPZ flag states that make it act. In the "Conditional Jump Instructions" chart on page 6-40 in the 1986 Data Manual listing the required states for status flags "|C|N|Z|", it wrongly states that "JPZ |x|0|1|," when the correct status flag pattern is "JPZ |x|0|x|," which also describes more obvious the status flag state for NOT NEGATIVE which JPZ (JUMP ON POSITIVE OR ZERO) is just another way to say the same thing." The 1983 manual has it correct. Documentation errors can cause a lot of coding frustration, so be aware particularly with TMS Conditional Jumps. BTJZ and BTJO instructions are also different that first impressions with using more that one bit.

CTS OOOOPS!: The CTS code has one of those mistakes. When it tries to remove block #15, the five delimiter category characters { | } ~ del it got the boundary wrong. LINE# 332 and 333 it codes: "CMP %>7B,A & JP DELIMT" which unfortunately only separates four delimiters and leaves the { character in the alphanumeric category. Two correct ways to code that are: "CMP %>7B,A" & "JPZ DELIMT" or "CMP %>7A,A" & "JP DELIMT." The conditional JPZ includes the compare character and all characters greater than. JP just includes those that are greater than. So be cautious.

Branching to Reduce Accumulated Delays: The original CTS code parses the ASCII table in one decision column of nine decisions; eight decision pairs of compare and conditional jumps and one final drop-through. That is a single decision branch. Efficiencies are gained by cleverly shortening the accumulating delays by jumping to several shorter branches. As each branch has less decisions to make, less delays are accumulated, until a given character flows into the decision code that will direct it to its service routine. BTJO and BTJZ can be a little quicker to make a conditional jump to another branch but those two instructions use a OR condition between specified bits tested, not an AND; meaning it doesn't work the way a quick read and experience with other CPU's would suggest. A single bit in the operand of those instructions cause less problems. BTJO are not a replacement for the 'CMP+JMP' decision pair. Its is used in examples when it can make a decision on a single bit pattern, like the code that link the three command ASCII characters to their service routine. Another BTJZ is used to jump to split the ASCII table into two branches; 00:3Fh and 40:7Fh; Section 9, "Methods with Tree Structure to Reduce Stacked-Delays."

There are tradeoffs and sometimes you just have to code a few and see which creates a better balanced solution. The code examples herein will likely help you do that. One extra decision to divide the progression into roughly a third, then divide the two thirds segment into two halves works well. Any coder has done this many times.

4). Method 2: Re-prioritizing the Original Decision Code
The STINPB code uses 'ASCII Parsing' to identify the category, of any character that is loaded in Reg A before the subroutine is called. For now all that needs to known is that the original code is one branch of nine decisions including the final drop-through.

The priority of categories was not considered in the original code, so all this section applies to is rearranging the nine decisions to different positions in the branch. This code has been written and its performance metric calculated. It is a 32 byte direct replacement that runs faster for the most-used ASCII letters in English text.

Another improvement in this section that 'could' be used, is the 'Shadow Pairing' that categorizes the top half of the ASCII table, using a 'coding trick,' to do that in three decisions. That is covered in the next section; that's all that can be said about it now, that it could be applied to shorten the code parsing ASCII. Note that Shadow Pairing works on the 15 subgroup set whereas CTS original only works on 13 subgroups, as previously mentioned and explained, above.

The reason this Method 2 is viable, is because the original code did not consider the time delays for this code space. The first three category decisions are used to recognize three ASCII

character, the complete command category. Most of the text-to-voice characters are going to letters and numbers, CTS's last category. By changing the order of this table, it should rank higher in the performance metric than the original code.

That has now been confirmed by the Performance Metrics for the two versions listed in this section below.

Note that the 'CMP+JMP' decision-pairs cannot be switched around in any pattern, there is a processing pattern described back in Section 3, "Method 1: Parsing ASCII into CTS Categories," that will explain, using the Stick of Butter analogy. The original code used the first four, single byte categories, trying to process them first and by doing so, creating a large combined delimiter group, an example described far above. That unfortunately meant that the most common characters in text was delayed excessively; something this section tries to improve. Section 9, "Methods with Tree Structure to Reduce Stacked Delays," also shows improvements by creating decision branches so that each branch as less delays, because each branch has fewer decisions.

This is the original CTS buffer decision Code:
Note: "jif" is my code term for "Jump If"

```
;00Tc  284:  CMP %>1B,A ;2b 7t ; DECISION #1: chr=1Bh? clear buffer command
;07Tc  285:  JNZ NOTESC ;2b 5t 7; DECISION #1: jif not, else drop-into EDIT3 to clear the buffer
;  EDIT3   (12Tc at drop-thru into EDIT3)
;14Tc  297: NOTESC CMP %>12,A ;2b 7t ; DECISION #2: Chr=12h? command Control-R clear buffer to the last delimiter
;21Tc  298:  JNZ NOTCTR ;2b 5t 7; DECISION #2: jif not, else drop-into EDIT2 to backspace to last delimiter
;  EDIT2   (26Tc at drop-thru into EDIT2)
;28Tc  310: NOTCTR CMP %>08,A ;2b 7t ; DECISION #3: chr=08h? command backspace last character
;35Tc  311:  JNZ NOTBKS ;2b 5t 7; DECISION #3: jif not, else drop-into EDIT1 to backspace last character
;  EDIT1   (40Tc at drop-thru into EDIT1)
;42Tc  330: NOTBKS  CMP %>27,A ;2b 7t ; DECISION #4: chr=27h? 'apostrophe' is an alphanumeric
;49Tc  331:  JZ STOCHR ;2b 5t 7; DECISION #4: jif to buffer it
;     (56Tc at jump arrival to STOCHR)
;54Tc  332:  CMP     %>7B,A ;2b 7t ; DECISION #5: *** BUG does only 4 *** chr=top 5 ASCII?
;61Tc  333:  JP      DELIMT ;2b 5t 7; DECISION #5: FIX: CMP %>7A|JP DELIMT or CMP %>7B|JPZ DELIMT
;     (68Tc at jump arrival to DELIMT)
;66Tc  334:  CMP     %>30,A ;2b 7t ; DECISION #6: remaining ASCII less than "0" are delimiters
;73Tc  335:  JN      DELIMT ;2b 5t 7; DECISION #6: jif to buffer them
;     (80Tc at jump arrival to DELIMT)
;78Tc  336:  CMP     %>3A,A ;2b 7t ; DECISION #7: remaining less than "9"+1 are alphanumeric "0:9"
;85Tc  337:  JN      STOCHR ;2b 5t 7; DECISION #7: jif to buffer them
;     (92Tc at jump arrival to STOCHR)
;90Tc  338:  CMP     %>41,A ;2b 7t ; DECISION #8: "A:Z" + "[\]^_`" + "a:z" (delimiters in middle?)
;97Tc  339:  JPZ     STOCHR ;2b 5t 7; DECISION #8: jif >="A" to buffer chr set as alphanumeric
```

;   (104Tc at jump arrival to STOCHR)
;102Tc 340: DELIMT   ; DECISION #9: DROP-THRU into DELIMT with :;<=>?@ characters

The new version starts processing category subgroups on the high end of the ANSII code values instead of the bottom end as the OLD code above. The OLD DECISION #5 must precede OLD DECISION #8 so that pairing can go first as NEW DECISION #1 & #2. That would push the most of the Alphanumeric to the top of the decision code so that they are buffered much more quickly.

NEW DECISION #1 takes the top 5 delimiters first, then NEW DECISION #2 takes the next top 58 ASCII characters subset=" A...Z [\ ] ^ _ ' a...z ". Note the six delimiters in between the upper and lower letters. It is assumed that those delimiters are being categorized as alphanumeric for a reason, unknown for now. New code versions, using other methods than this section, will be written to separate those delimiters, but this is just a reordering of the original code so it will use its same 13 subgroups.

NEW DECISION #3 takes the next top delimiters, 7 of them, so NEW DECISION #4 can take the next 10 Alphanumeric, the ten digits, 0...9. What remains of the ASCII table now is four single character subgroups separated by delimiter subgroups which is the way the original CTS code started. NEW DECISION #5 takes the apostrophe as an alphanumeric; all alphanumeric are processed by this point. NEW DECISIONS #6,#7,#8 take the 3 command single character subgroups, leaving only 5 delimiters groups of 44 characters that will be processed by the final drop-through into its service routine, DELIMT.

Here is the NEW version "Re-Prioritized CTS Buffer Decision Code":
```
;00Tc  xxx:  CMP %>7B,A ;2b 7t ; NEW DECISION #1: subgroup 7B:7Fh=" { | } ` del "
;07Tc  xxx:  JPZ DELIMT ;2b 5t 7; NEW DECISION #1: jif >=7Bh to DELIMT as Delimiters
;     (14Tc at jump arrival to DELIMT)
;12Tc  xxx:  CMP %>41,A ;2b 7t ; NEW DECISION #2: subgroup 41:7Ah="A:Z" + "[\]^_`" + "a:z"
;19Tc  xxx:  JPZ STOCHR ;2b 5t 7; NEW DECISION #2: jif >=41h to STOCHR as alphanumeric
;     (26Tc at jump arrival to STOCHR)
;24Tc  xxx:  CMP %>3A,A ;2b 7t ; NEW DECISION #3: subgroup 3A:40h=" : ; < = > ? @ "
;31Tc  xxx:  JPZ DELIMT ;2b 5t 7; NEW DECISION #3: jif >=3Ah to DELIMT as Delimiters
;     (38Tc at jump arrival to DELIMT)
;36Tc  xxx:  CMP %>30,A ;2b 7t ; NEW DECISION #4: subgroup 30:39h=" 0123456789 " as
alphanumeric
;43Tc  xxx:  JPZ STOCHR ;2b 5t 7; NEW DECISION #4: jif >=30h to STOCHR as Alphanumeric
;     (50Tc at jump arrival to STOCHR)
;48Tc  xxx:  CMP %>27,A ;2b 7t ; NEW DECISION #5: chr=27h? 'apostrophe' is an alphanumeric
;55Tc  xxx:  JZ STOCHR ;2b 5t 7; NEW DECISION #5: jif to buffer it
;     (62Tc at jump arrival to STOCHR)
;60Tc  xxx:  CMP %>1B,A ;2b 7t ; NEW DECISION #6: chr=1Bh? clear buffer command
;67Tc  xxx:  JZ EDIT3 ;2b 5t 7; NEW DECISION #6: jif EDIT3 to clear buffers
;     (74Tc at jump arrival to EDIT3)
;72Tc  xxx:  CMP %>12,A ;2b 7t ; NEW DECISION #7: Chr=12h? command backspace to last
delimiter
;79Tc  xxx:  JZ EDIT2 ;2b 5t 7; NEW DECISION #7: jif EDIT2 to backspace to last delimiter
;     (86Tc at jump arrival to EDIT2)
;84Tc  xxx:  CMP %>08,A ;2b 7t ; NEW DECISION #8: chr=08h? command backspace last
```

character
;91Tc xxx: JZ EDIT1 ;2b 5t 7; NEW DECISION #8: jif EDIT1 to backspace last character
;   (98Tc at jump arrival to EDIT1)
;96Tc DELIMT (96Tc at the DECISION #9 final drop-thru into DELIMT service routine for delimiters.

The re-prioritized original CTS decision code is much better than the original. It moves more priority characters first. It might be worth seeing if Shadow Pairing would be faster for the upper half of the ASCII table. As stated far above, when 44 Delimiters drop into DELIMT is saves 2Tc of time for each of those 44 characters. Drop-Throughs that 5Tc, Jumps take 7Tc.

5). Method 3: Exploiting ASCII Patterns in CTS Categories
PART 2 was initially just for some suggestions on improving the code, but when I saw this simple pattern in the CTS Categories of ASCII characters, I had to write some real code to explore it.

#1: Separate all Three Categories with Two Decisions: If a lookup table was indexed by the ASCII code of the 'given' character, and it returned a value that meant the given character was a delimiter or not, than all three categories could be separated by just two decisions!

A simple construct of that would be that the lookup table returned a value that set the status flags to zero if it was not a delimiter and those flags were non-zero if the given character was a delimiter. That would process any of all the the delimiters in the first decision.

If it was not a delimiter the remaining characters were already separated into two ends of the ASCII table, with the alphanumeric at 27:7Fh and the remaining command characters at 00:26h. The second decision pair would be "CMP %>27,A" + "JPZ STOCHR." That would process any of the alphanumeric characters and all the three command characters would drop through the JPZ conditional jump.

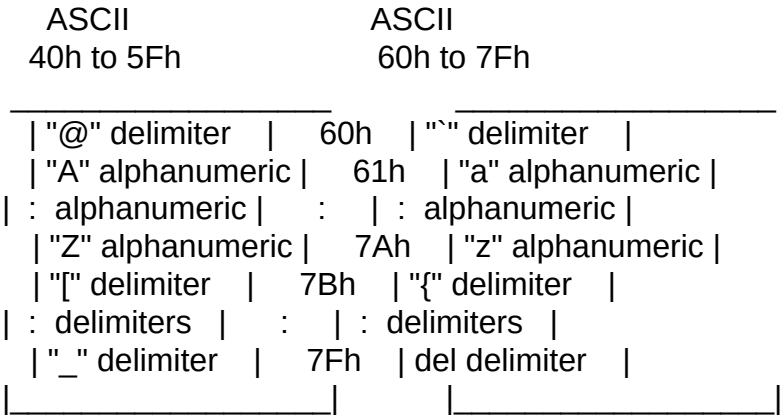
All category characters are separated. However, the command characters are different as they're intended to invoke one of three buffer editing service routines and then to exit STINPB witout being buffered.

A BTJO instructions jump to the first command's service routine by using a high bit that doesn't appear in the two commands ASCII code. That process is repeated in the next BTJO instruction; they're faster than the decision pairs. The last command character drop-through into its service routine. It cannot be much more efficient than that.

This section introduces the first of two significant ASCII table 'tricks.' The lookup table is described in more detail with various code versions in Section 6, "Method 4: ASCII Index to Lookup Category." The other ASCII pattern trick is described in detail below.

Double the ASCII tested with Shadow Pairing: One of my STINPB buffer decision code examples takes advantage of how CTS categorizes the ASCII code table. I spotted a nice pattern in the upper half. 40h to 5Fh and 60h to 7Fh both have the same CTS category pattern of 1 delimiter, 26 alphanumeric and 5 more delimiters. Note that the one delimiter at 40h is part of block #11. The diagram below illustrates that if you use three decisions to process the left diagram below, those

decisions apply to the right diagram too. Below this diagram, the code is described and the 'trick' only requires two additional lines of code. The pattern is illustrated below:



The CTS processes the given character from Register A. The 'trick' is to copy Reg A into Reg B, then mask-out bit5 of Reg B and process it logically from Reg B as the 40h to 5Fh left diagram above, and when you send the character to DELIMT or STOCHR, those routines grab the real character from Reg A... not the trick construct in Reg B.

It is true that you could use the right table (60h to 7Fh) instead, if you set bit5 of Reg B instead of zeroing it as the provided code.

The functionality result is that Reg B logic processes the correct characters for all delimiters and alphanumeric in the range from 40h to 7Fh, the top 64 half of the ASCII table. And 26+26=52 of high priority alphanumeric characters get quickly buffered.

This efficient 'trick' can be used with some other methods like re-prioritization and using by branches. The trick would apply to "Parsing ASCII" code.

The code listing is below. The Shadow Pairing code segment begins with a branch. The code looks like this:

```
      BTJZ  %>40,A,LT40   ;3b 9t11;priority branch drops through, processing the top 64 ASCII codes in 3 decisions: 2 'CMP+JMP' and 1 DROP-THRU:
GT3F  MOV   A,B           ;1b 6t  ;***SETUP CODE: B will identify the delimiter, alphanumeric, delimiter pattern on both 32B blocks in the upper half of ASCII
      AND   %>DF,B        ;2b 7t  ;***SETUP CODE: B is mapped into B=40:5F, if A is in 40:4F or 60:7F.
      CMP   %>5B,B        ;2b 7t  ;***USE B NOT A: if Reg A is in 5B:5F or 7B:7F it will be sent to DELIMT for processing as a Delimiter character
      JPZ   DELIMT        ;2b 5t 7;if TRUE 1 of 2*5=10 delimiters takes the jump for processing it
      CMP   %>41,B        ;2b 7t  ;***USE B NOT A: if Reg A is in 41:5A or 71:7A it will be sent to STOCHR for processing as an Alphanumeric character
      JPZ   STOCHR        ;2b 5t 7;if TRUE 1 of 2*26=52 alphanumeric takes the jump for processing it
                          ;when using this segment in code, replace the following line with the label the processes block #11.
      JMP   DELIMT        ;2b 7t  ;if drop-thru in must be 40h or 60h, it will be sent to DELIMT for processing as a Delimiter character
```

At the last line of this code, only ASCII character 40h and 60h have not been ruled out. Instead of

processing them immediately as coded in this example, jumping to the code extracting block #11 would be more efficient as the greater-than of that test would include the remaining 40h and 60h delimiters. One less decision. Some less delay.

When subgroup #11 is processed with a greater-than type of conditional jump, those two bytes will be sent to DELIMT.

That code processes any of the ASCII uppercase or lowercase letters in 48Tc as opposed to the original CTS doing it in 104Tc and it supports the 15 subgroups whereas the CTS original only supports the 13 subgroups.

The GT3F (greater than 3Fh ASCII) label code ends with a JMP DELIMT. A drop-through could have been used but its more efficient if a big block of delimiters drops into DELIMT because a drop through takes 5Tc whereas a conditional jump takes 7Tc, so big blocks of delimiters taking a drop through into DELIMT, each saves 2Tc. Letting 40h and 60h drop through would only save 2x2=4Tc. The bigger the group, the more Tc saved. The biggest delimiter block is usually the group formed after the commands and the alphanumeric apostrophe has been 'notched' out below ASCII code 30h. That saves 48-4(notches)=44 delimiters x2 each=88Tc. That stacked-delay might be shortened a little if the set of four 'notches' were done by two different branches.

I may code that and post it, if a PART 3 seems necessary as a non-sequential, later article. I write all the code versions and calculate the performance metrics to see how they compare. Battle of the STINPB replacements?

6). Method 4: ASCII Index to Lookup Category
As described above in Section 5, "Method 3: Exploiting ASCII Patterns in CTS Categories," the first ASCII pattern that I sought to exploit required a lookup table to process any of the delimiters in the first decision. Section 5 described that method in detail, except for the data table construct which will be described below.

This section will describe the design of data tables and looks at the spectrum, from the very fast data table access, to others that pack more answers into each data byte. The latter saves code space by reducing the data bytes needed. Note that byte-packing in general, adds code used to unpack the data byte and adds code to modify the index to the shorter data byte count. Seeing this process should build understanding of trade-offs among different data table constructs. Sometimes coding a few quick versions will reveal the most efficient trade-offs for the needs of data table code.

The first decision in this example, described above in Section 5, needs a data table to recognize all the delimiters ASCII codes. The ASCII code itself, or some manipulation of it for one or more indexes into one or more data tables to get an answer to the question, "Is the ASCII character given in Register A, a member of the set of all delimiters? Yes or No."

Two data table constructs will be coded below: (1) will be extremely inefficient using 128 data bytes, each encoding the 1-bit answer, and (2) a byte-packing version that puts eight 1-bit answers in each of sixteen data bytes.

Construct (1) will be the minimal code because no additional code is required to modify the ASCII character in Register A, into an index to a byte-packed table, and no unpacking code is added too. Data byte-packing can sometimes be a severe trade-off. Sometimes it depends upon what instructions your CPU has to solve the trade-offs efficiently.

This is the extreme version wastefully using 128 data bytes with 1 bit of each data byte used:

```
FLGDLM DATA >8080,>8080,>8080,>8080,>0080,>8080,>8080,>8080 ;big waste of code space
 DATA >8080,>0080,>8080,>8080,>8080,>8000,>8080,>8080
 DATA >8080,>8080,>8080,>8000,>8080,>8080,>8080,>8080
 DATA >0000,>0000,>0000,>0000,>0000,>8080,>8080,>8080
...
 MOV A,B   ;1b 5t  ;use the 'given' character in register A, as index (B)
 LDA @FLGDLM(B)  ;3b13t  ;immediately index to the characters data byte. As it loads int Reg A, the status flag update
 JNZ DELIMT ;Decision #1 ;2b 7T ;immediately conditional-jump on NZ because that is a Delimiter, otherwise drop-through
 CMP %>27,A   ;2b 7t ;separate the alphanumeric from the commands at the placement of the first alphanumeric
 JPZ STOCHR ;Decision #2 ;2b 7t 5;jump to buffer alphanumeric if the 'given' character is equal or greater than 27h (apostrophe)
      ;else its not an alphanumeric and known now to be one of three commands
 BTJO %>02,A,EDIT2 ;D. #3 ;3b 9t11;test unique bit high for this character, if true go to "CLEAR BUFFER TO LAST DELIMITER"
 BTJO %>01,A,EDIT3 ;D. #4 ;3b 9t11;test unique bit high for this character, if true go to "CLEAR BUFFER"
    ;D. #5 ; ;else it must be EDIT1, drop-into EDIT1 service routine
300: EDIT1     ;legacy code
```

For all the wasted code space consumed by data bytes, this code is very fast and very efficient. It processes any delimiter in 5+13+7=25 Tc.
It processes all of the priority alphanumeric characters in 25+14=39 Tc. Edit2 in 29-2+11=42 Tc. Edit3 in 42-2+11=51 Tc. And finally Edit1 in 51-2=49 Tc. In the original CTS code, high priority alphanumeric characters are processed last at 104 Tc.

The next data table construct packs eight answer bits in sixteen data bytes. This version turns the ASCII character in two smaller indexes into two data tables. I wrote my first version in a manner that would make the creation of the data tables easy in a spreadsheet. That version was slower because it used b2:0 and the index to which bit should be tested, and b6:3 to index one of the 16 data bytes of eight answers. The bit test index could be done two ways, (1) it could rotate a single high bit in a loop counting down the 3 bit index. Loops add delays so I avoided that and tried a second data table of 8 data bytes, each holding a cleared byte with one bit set for testing. If ASCII character b2:0 were 011 binary than the index would be 3 and a data byte with bit pattern "00001000" binary was easy received from the bit test data table. That value was held in a register to be AND(ed) with the 1 of 16 data byte that held the answer bit for the ASCII character. Instead of a second data table, some calculation could be used to convert b2:0 into that bit pattern. A shift or rotate loop would add too much delay.

One of the obstacles this choice created is that the b2:0 and the b6:3 were not already within nibble (4-bit) frames so it required addition RL or RR instructions to frame both indexes into

nibbles and the mask out one and use the swap nibbles instruction. The code seemed too big and the only ways to improve it two change the two indexes to be already nibble-framed: bit test index was changed to ASCII character b6:4 and data byte with the answer, used index b3:0. It saves two instructions; not looking good.

Below is the all the code necessary to do the latter set of indexes without the RL instruction:

My first code solution took two tables with 16+8 data bytes and two ASCII character modifications to create two indexes. The code was still faster than the original CTS, and with the previous article about saving 122 code space bytes using the TRAP N instruction instead of CALL instructions, it would fit in the CTS code. But it still seemed too inefficient to be wasting that code space in the STINPB subroutine, when other unknown prices of code might need faster code more than STINPB.

The first version was easier to make the data table, using ASCII character b2:0 as an index into eight data byte holding the bit-test pattern the sixteen data byte lookup that would hold the character's answer bit among eight answers per data byte. ASCII b6:3 provided the index to retrieve the correct data byte. With the data byte and the test bit, the two and AND(ed) to get the answer.

While a valid construct, the choice of partial ASCII into two indexes where not byte-nibble positioned and it required extra instruction to shift the index into a nibble frame before AND(ing) the byte to mask-out all but the zero-relative index.

The second version (below) is a slight improvement of that. It uses the ASCII b3:0 as the data-byte index and b6:4 as the test-bit index. The rotates were no longer needed as the two indexes are already within byte nibble frames. There is a interesting lesson in that; I certainly learned something new. I think there is another 'trick' to avoid the bit-test table's and its 8 data-bytes.

This second version code is just a little faster than the first version so I coded it completely and the results were not good.

```
 PUSH A  ;1b 6t ;copy given ASCII character once
 PUSH A  ;1b 6t ;copy given ASCII character twice
 MOV A,B  ;1b 5t ;make index to test bit# B=0bbbiiii where bbb index (iiii is data byte index)
 SWAP B  ;1b 8t ;swap nibbles, Reg B=iiii0bbb
 AND %>0F,B  ;2b 7t ;char b6:4 now is test bit index Reg B=00000bbb
 LDA @BITDLM(B) ;3b13t  ;get test bit image
 MOV A,R80  ;2b 8t ;store test bit image
 POP B  ;1b 6t ;get char image
 AND %>0F,B  ;2b 7t ;char b3:0 now is data byte index Reg B=0000iiii
 LDA @FLGDLM(B) ;3b13t  ;get data byte
 MOV A,B  ;1b 5t ;store date byte image in Reg B
 POP A  ;1b 6t ;restore given ASCII char image back to Reg A
 AND R80,B  ;2b 7t ;data byte bit tested and status flags set
 JNZ DELIMT  ;2b 5t 7;given char is a delimiter, jump to service routine
    ;else its not an delimiter only alphanumeric and commands remain
 CMP %>27,A  ;2b 7t ;divide alphanumeric from the commands at the placement of the first
```

alphanumeric
 JPZ STOCHR ;2b 5t 7;jump to buffer alphanumeric if the CMP yields a positive or zero status flag; i.e. if it is 27h or higher.
    ;else its not an alphanumeric and know now to be one of three commands
 BTJO %>02,A,EDIT2 ;3b 9t11;test unique bit for this character, if true go to "CLEAR BUFFER TO LAST DELIMITER"
 BTJO %>01,A,EDIT3 ;3b 9t11;test unique bit for this character, if true go to "CLEAR BUFFER"
    ;else it must be EDIT1, drop-into EDIT1 service routine
300: EDIT1   ;legacy code
For 16+8 data-bytes for two data tables; a test-bit pattern in a single byte indexed by the given ASCII b6:4, for a byte with with the test-bit to AND with the 1 in 16 data bytes that have the eight 1-bit answer for eight ASCII characters. When both data bytes are AND(ed) together the status flags are updated and the next instruction can do a conditional jump via those updated status flags.

The speeds are: A delimiter processed in 104 Tc. An alphanumeric in 109 Tc.

It looks like data tables with unpacking won't speed up STINPB. And the added code in the version above seems independent to how big the data tables are; if true than making the data tables smaller will not help.

7). Method 5: ASCII Index to Lookup Service Routine
The inefficiency seen in the previous data table method, is in the problem of mapping 128 ASCII characters into only 5 different answers. That requires a big data table or more code to unpack a larger data table. Reducing the data table will not lead to a better result. Only reducing the number of characters, could assuredly do that. Not that that helps with CTS Version 1. This crosses the bounds between CTS Version #1, into CTS Version #2.

A real CTS Version #2 solution, which will be an article topic, after the text-to-voice algorithm is a topic, because that text-to-voice code would govern the most advantageous, new encoding for alphanumeric, and delimiters beyond STINPB; command characters are not buffered so they do not exist beyond STINPB.

CTS Version #2, at the STINPB stage, would simply convert all given input ASCII character as an index into a new encoding from a 128 byte lookup table, that would provide a significant advantage for processing alphanumeric and delimiters through the whole of the rest of the CTS code, most particularly the text-to-voice algorithm. The problem with CTS Version #1 is that anytime it needs to make decisions based upon the ASCII character code input to STINPB, all the ASCII characters are scrambled around in 15 subgroup blocks, that makes all character based decisions harder to identify. CTS Version #2 would need to remove that inefficient obstacle to make the code more lean by reducing the existing difficult identification of characters.

The obvious example would take all the alphanumeric and sort them into a single Block #1 at the low end of the new encoded character range. The delimiters would likewise sorted to their unique advantageously new sorting in Block #2. Groups of delimiters applicable to certain routines or text-to-voice sections, can be subgroup to together and sorted to allow quick identification.

For the rest of the code, alphanumeric and delimiters can be separated by one comparison and conditional jump. Block #1 alphanumeric could simply be of the range 0:7F sorted to the low end and Block #2 delimiters could be in the range 80:FF (remember, its not ASCII anymore). That would already make the identification of alphanumeric and delimiters automatic when they are moved as it sets the status flags as positive for alphanumeric and negative for delimiters; probably an advantage that CTS Version #1 used by setting b7 on delimiters in routine DELIMT.

Alphanumeric and delimiters could also be directed by a single BTJZ/BTJO testing bit-7.

It may be that upper and lower case characters in their new encoding, are a single value, if the text-to-voice algorithm doesn't require both. Or they might be sorted as "A,a,B,b..." so a quick single RR instruction could convert any letter into a single 26 index for a text-to-voice data table when upper/lower case doesn't matter.

Code that used a ASCII character comparison, would just use a EQUATE like "LETR_A" instead. Simplifies the conversion unless efficiencies make a Version #2 code-section rewrite, a better choice.

All this is only feasible in a new version where the text-to-voice code and data tables would have to change. This has some very good potential, but I'll only write deeper on this subject when the text-to-voice algorithm is discussed.

8). Method 6: The Minefield Index
While looking for a new 'trick' to make the connection between 128 ASCII characters and five service routines, with very little setup code; this popped into my head. Its structure is a grid of "TRAP" instructions and when one is triggered, it goes "POP POP." A name for it was obviously "minefield."

But its actually a feasible Lookup Service Routine construct without the data tables. It still requires 128 Bytes to vector to all the service routines.

The quick description is:
1). Place 128 single byte, of prepared TRAP N instructions, before the START label. START becomes address F080h.
2). Each TRAP N instruction uses a value N of the vectored address to the appropriate service routine: STOCHR, DELIMT, EDIT1, EDIT2, or EDIT3.
3). STINP does its normal, first 5 instruction initialization. (never included in the new code versions nor in the time metrics).
4). The next instructions are:
```
 MOV %>F0,R79 ;3b 9t ;initialize the pointer MSB to F000h ;R79:80 is the bottom of the stack, known not to be reached by this code.
 MOV A,R80  ;2b 8t ;ASCII character in Reg A becomes the LSB.
 BR *R79  ;2b 9t ;jump to the single byte TRAP N instruction (prepared for ASCII characters) in the block F000:F07F
;------------------------------;
F0nn: TRAP N  ;1b 14 ;1 of 128, vectored call to service routine
;------------------------------; ;all characters are sent to their service routine in 40 Tc or 52 Tc with the two POPs.
```

```
STOCHR POP B  ;1b 6t ;each service routine does this first
 POP B  ;1b 6t ;
 ...
```
5). The CPU jumps to F000 plus the ASCII value in Register A where
6). The TRAP N pushes a return address on the stack (not needed).
7). The five service routines are modified to begin with two POP instructions to remove the unneeded return address.

As the only priority is to get alphanumeric and delimiters buffered, and because EDIT1, EDIT2 and EDIT3 are single use fake-subroutines, it would likely be a waste of TRAP N instructions to include all 3 of them. One Trap N could send all three to this code and let the BTJO direct them.

```
EDITS POP B  ;1b 6t ;each service routine does this first
 POP B  ;1b 6t ;
 BTJO %>02,A,EDIT2 ;3b 9t11;test unique bit for this character, if true go to "CLEAR BUFFER TO LAST DELIMITER"
 BTJO %>01,A,EDIT3 ;3b 9t11;test unique bit for this character, if true go to "CLEAR BUFFER"
     ;else it must be EDIT1, drop-into EDIT1 service routine
300: EDIT1    ;legacy code
```

It takes 18B of code in STINPB compared to 32B for the original, 128 pseudo-data byte that are table of code bytes, 40 Tc or 52 Tc if you add the two POPs which seems reasonable.

NOT UPLOADED: 9). Methods with Tree Structure to Reduce Stacked-Delays

File Attachments

1) Fig_1.png, downloaded 269 times
2) Fig_2.png, downloaded 267 times
3) Fig_4.png, downloaded 266 times
4) Fig_5.png, downloaded 266 times
5) Fig_3.png, downloaded 250 times

Subject: A look inside a Mattel Electronics, "IntelliVoice, Voice Synthesis Module.
Posted by jayindallas on Mon, 03 Feb 2025 04:37:04 GMT
View Forum Message <> Reply to Message

DISCUSSION: Diving Deep into the CTS256A-AL2 Firmware

A look inside a Mattel Electronics, "IntelliVoice, Voice Synthesis Module." Updated 20250223
Awhile back, on one of the CTS topic threads, I posted something I read on a website while doing CTS research... that named a few vintage electronic devices that had General Instrument CTS and Voice Synthesis chips or GI's AY-3-8910 (-891x etc) programmable sound generator in their circuit boards. One that caught my attention was the Mattel IntelliVision, and its IntelliVoice module had both sets of chips.

I found the family IntelliVision system and opened up the case of the IntelliVoice and sure enough

it had the two General Instrument chips (socketed) under a different part number below. There were no SPR-xx serial ROMs in the IntelliVoice PCB nor in the game cartridge.

These are the ICs on the Voice module circuit board:

```
 _______________
|   SPB-640   |  40p DIP Socketed
|) GI 8247 CFA|  (CTS-256A-AL2 equivalent?)
|_____________|

 _______________
| SP-0256-012 |  28p DIP Socketed
|) GI 8246 CKA|  (SP-0256-AL2 equivalent?)
|_____________|

 _______________
| SN72LS12N   |  14p DIP (T.I. logo) Triple 3-input Positive NAND gates with open-collector
outputs.
|_____________|

 _______________
| LM324N      |  14p DIP (T.I. logo)
|_____________|

 _______________
| LM358N      |  14p DIP (T.I. logo)
|_____________|
```

It appears to have the analog output signal onboard, with a volume control dial at the front of the IntelliVoice module. I didn't open up the IntelliVision module to confirm the AY-3-8910 because I have one on a S-100 wire-wrapped board. I'll do that later.

Photo #1 (below)shows the IntelliVision module on the left, in the middle is the IntelliVoice module that plugs into the game cartridge socket of the IntelliVision, and on the far right is a game cartridge that came with the purchase of the IntelliVoice in 1987.

Photo #2 (below) shows the IntelliVoice opened up with the circuit board. The -Voice module was well designed for assembly, the -Vision module was less well designed. If you take the latter apart, you might use a video camera to help you reassemble it. Their cable runs and connectors are loosely run together through the assembly.

Between the 40 pin DIP CTS and the 28 pin DIP SP0, halfway and slightly to the left, is a 15x2 header which might be used to control the IntelliVoice module. The top of the photo layout PCB has a volume dial and if I recall, a small speaker on the circuit-side, that would be at the front of the module.

Photo #3 (below) shows the IntelliVision opened up with the circuit board or the right, still inside its EMI shielding.

I've also seen a few 'Talking' electronic products for sale on Goodwill's online auction site. You could likely get a better price for something there, but they don't list the chips inside, so you would

be taking a chance on some of the items they've sold. I have not bid on any of them.

Later I'll probably try to extract the code from the 'SPB-640' and see it its the same code we found from the 2022 project. I noticed that the circuit board had an unused, wide DIP 14 pin where a DIP-switch would fit. Another unused header was on the board; that might be interesting to look into. Might be able to run/control it with an Arduino hookup?

If I decide to play with the two GI Voice chips plus the AY-3-8910 PSG, I'll likely be inclined to do a Raspberry Pi 'hat' board or plug them into my old Heathkit ET-3300 Laboratory Breadboard and experiment a bit. Ultimately I'd like to emulate the CTS+SP0 entirely in code running on a RPi Zero 2W. It would be nice to have sensor or schedule events text-for-voice transmitted over RPi wifi to a central RPi that would emulate the text-to-voice and send it to a bluetooth sound bar.

I'll use this message for posting additional vintage-electronics that have voice synthesis or programmable sound effect chips inside and/or updates to the above. I'll probably reduce the image sizes later this week.

File Attachments

1) IntelliVoice01of03.jpg, downloaded 149 times
2) IntelliVoice02of03.jpg, downloaded 144 times
3) IntelliVoice03of03.jpg, downloaded 147 times