

1) Webpage Requirements for 11/14/06  
2) Notes on Assembly Language Programming  
3) `mtx.asm` – a 68HC11 Multitasking Kernel that fits under 900 bytes  
CPSC / ECE 3710, Fall 2006

Section 1 tells you how to have your webpages prepared for the review of 11/14/06 to be conducted by the TAs (I'll be on the road, but will be viewing your webpages remotely.) This section also tells you about your final take-home exam. Section 2 gives you tips on assembly programming. Section 3 gives you an example assembly program that illustrates much of the advice given in Section 2.

## 1 Webpage Requirements – and final take-home exam

You must be updating your webpages regularly. Your success in creating a *very informative* final project report depends on how much information you have captured in a *timely* manner.

Remember that an engineer reports not just the fact that “everything is working.” The engineer reports

- the process of developing their ideas
- their **failed experiments**
- how the failure(s) were overcome
- what “success” really means (to what extent has the final system been tested), and
- what more could be done.

Your final project report must contain the above details in order to earn a decent number of points.

Well you may say, “but I never failed!” So why are you forcing me to report on failures? All I can say is that if you never failed, you probably did not push yourselves enough (you did not incorporate enough new features in your project).

Some failures are known to “mysteriously go away” without one’s being able to find true explanations. This is a non-ideal situation, being permitted only because we don’t want to become hung-up at every stage (we don’t have unlimited play-time). So, in case of failures that disappeared as if through magic, you must at least make a good educated guess of the likely reasons.

*You must be as professional and objective as you can, in your writeup.* Thus, if the likely causes of the download suddenly beginning to work includes Mercury occluding our sun’s disk, well, . . ., I may get a chuckle out of it, but my chuckles won’t turn into points, as I’ll worry about your doing the same when working for, say Intel or IBM (in which case your manager will throw you out). Therefore, I am going to look for a high quality and objective writeup. Unwarranted forays into humor, terribly poorly organized reports, and reports not containing enough critical analysis of your successes and failures – all will cause points to be docked off.

The golden rule is to represent each point/thought somewhere in your writeup; the idea is *not* to rub it into the minds of the reader through needlessly long prose. At the same time, omitting crucial facts will cause points to be taken off.

## 1.1 There is always a better rewrite!

Given that we cannot attain perfection in writing, and you are all learning how to write well, I am willing to provide you critical feedback on your project reports *one time*. So, if you submit your *preliminary* project reports on November 28th (hardcopies please!) to me, I can go over them and give you feedback on Nov 30th, as to whether you are heading in the right direction.

## 1.2 Your take-home final exam

You will be given a take-home final exam. You have to solve the exam individually. You'll be given this exam on December 7th (last day of classes) and the exams will be due the next day.

*The exams will have to be solved individually.* No two exams will be identical - especially across groups. Here is what will be in the exam (all the questions are solved on paper):

- Simple properties of 2's complement numbers
- Simple questions on CR-16 assembly programs (what does this program do?)
- Simple combinational design
- Simple sequential design
- I will go through your designs individually (all design files are expected to be on your webpages by December 4th evening 5pm). I'll then ask a very specific question about your own project. I may ask "what is this unit doing" or "describe the flow of information through hardware blocks when you do a conditional branch."

All this means that each group member must become familiar with all aspects of their project. *One good way* to become familiar with one's project is to document it well on the webpage! In doing this documentation, you'll essentially have reviewed your design. So if each project member does some web updates, as well as reviews info on the web, they will benefit with respect to the take-home exam.

## 1.3 When is a project "done"?

There is no hard and fast rule to tell when each project has covered sufficient ground. Clearly, the baseline instructions, working with the SDRAM, and working with the VGA are expected of everyone. Beyond that, I had indicated various extensions that amount to reasonable projects. So if a group achieves keyboard interfacing, the creation of interrupt hardware, and writing some simple routines, that is a situation where a project is considered "safely locked in." The groups must then try more things in a "risk free" mode (if those experiments failed, you should be able to restore your working final designs). So you should all aim to lock in a decent project, and then leave some time for demonstrating extra creativity.

## 2 Notes on Assembly Programming

Assembly language programming is, actually, a very intellectually satisfying activity:

- It forces one to discipline oneself utmost

- It forces one to learn the CPU they are working with really well
- It has no “artificial barriers” such as type-checking rules or restrictions due to object oriented programming peculiarities. It allows one unlimited freedom for being expressive!

One will also easily learn that one allows for “unlimited rope” to hang oneself. However there are many simple but powerful rules one can use to avoid bugs from being created.

Speaking from my own personal experience, I have written two large assembly programs. One was the runtime for a Prolog language compiler (MC 68000 assembly language). The other is a tiny multitasking kernel called MTX that has all the functionality of a standard operating system, but the entire code fits under 900 bytes! In both these projects, I learned very quickly that unless I am extremely defensive and program with extreme care, nothing at all will work. But with such care exercised, the code looks beautiful! I hope the body of code I am presenting in Section 3 makes this clear.

Here are the rules of thumb I’ve evolved for readable and reliable assembly programming. *I require that* the assembly code you write is also similarly beautifully documented!

### 2.0.1 Document (virtually) every line of assembly code

There is no escaping this! Assembly instructions are not very intuitive, and so before reading an assembly instruction, one must read some helpful English text.

There are two ways of documenting, and both must be used judiciously:

**Documentation paragraphs:** These are preambles to major sections, constant initialization blocks, etc. Add ASCII pictures also!

Examples from MTX:

```

; IX
; |
; v
; -----
; | KQ |
; -----
;
;
; Now get the next free location beyond the keyboard queue - it's easy in our
; design: the pointer for that free location is the first word of the keyboard
; queue; The following instruction accomplishes the task!
      LDX      0,X

;
; IX
; |
; v
; -----
; | KQ |
; -----
;
;
; Create the display buffer at this address of size DQSize
      LDAB     #DQSize
      JSR      createq

```

**One line of documentation preceding an assembly instruction:** This helps remind the programmer/reader what is about to happen. It's good to keep these sentences succinct. Sometimes a one-line sentence documents a group of instructions; in that case, the instructions in that group must be logically related.

Examples from MTX (of one documentation line per line of code, and one documentation line for many lines of code):

```

; Copy current SP into IX - SP pointing to stack frame of SWI interrupt
    move_s_to_x

; Save this IX
    PSHX

; Bump saved PC in stack - also returns IY pointing at the SWI opcode
    JSR bump_pc_in_stk

; Test what SWI it is
    LDAB 0,Y
    CMPB #GETCH_SWI
    BNE  must_be_putch

```

**Documentation coming on the same line as the instruction, but following it:** This is highly recommended for constant settings. Otherwise, these are more like caveats, and side thoughts. In other words, it's good never to force a programmer to read the assembly line and then wonder what is being said. Examples from MTX:

```

SCDR  EQU          $102F  ; SCI data register
SCSR  EQU          $102E  ; SCI status reg
RDRF  EQU          $20    ; Bit RDRF,

```

Here is another example where both styles of comments - main comments, and side notes - are used

```

; Set bit1 of DDRC, preparing to o/p 1 on Tx (Tx is the same as PD1)
    LDAA  #$02
    STAA  JPORTD,X ; bit1 of DDRD is set - so drive PD1 high
    STAA  JDDRD,X  ; Configure data direction of PD1 as output

```

## 2.0.2 Use Macros

Macros are like functions, except there is no function call / return overhead. The use of short and meaningful macros can indeed make the code highly readable, understandable, and the number of lines of code (in the listing) to go down. Here are examples of macros used in MTX.

```

; MACRO DEFINITIONS
;
$MACRO move_x_to_y
    PSHX
    PULY
$MACROEND
$MACRO move_s_to_y
    TSY
    DEY ; TSY copies SP+1 to IY - so correct it!

```

```
$MACROEND
```

```
; MACRO CALL USAGES
;
; ...one usage...
; To do the above, first transfer IX to IY
;   move_x_to_y
;
; ...another usage...
; Copy current SP into IX - SP pointing to stack frame of SWI interrupt
;   move_s_to_x
```

### 2.0.3 Define Constants and Calculate Other Constants

If you define a few key constants and define others in terms of the original ones, the code becomes very readable and portable. Here is an example where the last line containing LDAB is capable of describing this picture, thanks to the nicely set-up constants. (Of course your assembler can easily be extended to do these calculations for you.)

```
;           IX
;           |           PTESiz
;           v           <---->
; -----
; |   |   |   |   |   |   |   |   |   |   |   |
; | KQ | DQ | PT header stuff | PTE1 | PTE2 | ... | PTEn | Null-task | Task1 |
; |   |   |   |   |   |   |   |   |   |   |   |
; -----
;           |--- PTE10ff ----->           |<----->|
;
;
;           InitSPOff
;
;           |-- (PTE10ff + BPTSize * PTESiz) + InitSPOff ----->
;
; In the above, "n" is the same as BPTSize
;
; As per this picture, we need to deposit the SP for the null task at
;
; IX + (PTE10ff + BPTSize * PTESiz) + InitSPOff
;
; Let's compute this value in IY and pass IY as a parameter to creatept
;
; To do the above, first transfer IX to IY
;   move_x_to_y
;
; Then load ((PTE10ff + BPTSize * PTESiz) + InitSPOff) - 1 into ACB
; The "-1" is to make the initial SP point one above the place where
; CCR is stored so that "RTI" works OK...
;
; LDAB    #(((PTE10ff+(BPTSize*PTESiz))+InitSPOff)-1)
```

## 2.0.4 Within functions, try to save registers (so that we don't clobber them)

Below, I present part of a subroutine that shows how registers are saved on the stack and restored before return.

```
-----  
; Subroutine createq : create a queue of given specifications  
;  
; Input:  IX   = Queue object base  
;        ACCB = size of the queue  
;  
; Memory side-effects: Creates a queue object beginning at Qobject  
;                      and capable of holding size number of bytes  
;  
; No registers affected  
  
createq EQU      *  
  
; Save registers A and IY  
    PSHA  
    PSHY  
  
; Get contents of Reg X into Reg Y (a trick method - better ones?)  
    move_x_to_y  
  
; For various reasons (lack of certain opcodes 6811) free up ACCD (hence ACCB)  
    PSHB  
  
    ..more code not shown...  
  
; Restore ACCA and IY and return  
    PULY  
    PULA  
    RTS
```

## 2.0.5 Other Ideas

A few other ideas may also be learned from the code for `mtx.asm`. You are invited to read through and understand the code, as this includes all the elements of a simple OS kernel.

# 3 MTX: a 68HC11 Multitasking Kernel that fits under 900 bytes

We provide a brief overview of `mtx.asm`. This multitasking kernel allows for three processes, including the null process. Each process has 32 bytes of data space to work in. So for instance, one can have a 16-byte stack and a 16-byte “heap.”

The code first creates a keyboard buffer and a display buffer. Then the process tables for the three processes are created. The process-table entry for each process stores the saved PC values, registers, and flags (including interrupt mask).

We basically manufacture the saved stack for each process and do an “RTI” – as if we are returning from an interrupt from the past. This makes the execution go to the null task, which simply prints “.” on the screen. Whenever a key is pressed, the SCI interrupt happens. The SCI interrupt handler runs and sees various status flags. If everything looks fine, the keyboard driver

`do_keyboard` is called. This driver checks if, as a result of getting this interrupt, a process needs to be made runnable (done through `JSR unblock_and_ready`).

`unblock_and_ready` is where context-switching is happening. Basically, when this subroutine is called, the stack pointer is initialized to the stack top of the runnable process. Then when an RTI is executed, we return from interrupt, but begin executing another process!!

The remaining details will be clear once you study this code. Again, the 68HC11 assembly language is not at all hard. Please study the instruction set and other details by consulting

<http://www.hc11.demon.nl/thrsim11/68hc11/tech.htm>

The entire listing of `mtx.asm` can be found online.