CSA COURSEWORK DECEMBER 2011

# Report



To insert assembler code into a high level language, it must be clear where the assembly code begins and ends. In the case of C, the term \_\_asm { is used to begin a section of assembly code, and a curly bracket is used to end it. This can be used several times within a single program so as to create subroutines.

The above diagram shows a breakdown of all the required tasks for the project. This not only makes writing and editing the code easier, but also helps to break down the assembler code into easy to manage subroutines.

Handling Function Parameters

In the project two parameters were used and defined in C:

char wtmode[] = "wb";

char rdmode[] = "rb";

The first is the parameter used to write. This allows the characters required to be transmitted from the original text file to be written to the COM port and transfer through the R232 cable to the attached computer. The second is the parameter used to read. This allows the characters in the transmitting data file to be read in preparation to be moved onto the stack ready for transmission.

Setting up the Stack Frame

Parameters must be loaded onto the stack so that they can be accessed by a function.

lea eax,rdmode

push eax

These lines ‘load the effective address’ and ‘push’ the parameter onto the stack.

The stack frame holds the return address, local variables and procedure parameters. A marker is needed for when more than one procedure is on the stack frame at once to show where the top of the stack frame is and where the bottom of the pervious frame is. This is where the stack frame base pointer helps to see the partitions. The stack frame base pointer is stored within a CPU register. It acts as a partition within the stack. Any items from previous functions are higher up on the stack, and will not be modified.

Stack frames work so that each subroutine can act independently of its location on the stack, and each subroutine acts as if it’s the top of the stack.

“*The current function always has access to the "top" of the stack, and so functions do not need to take account of the memory usage of other functions or programs.”* - Wikibooks⁵

Accessing the Parameters

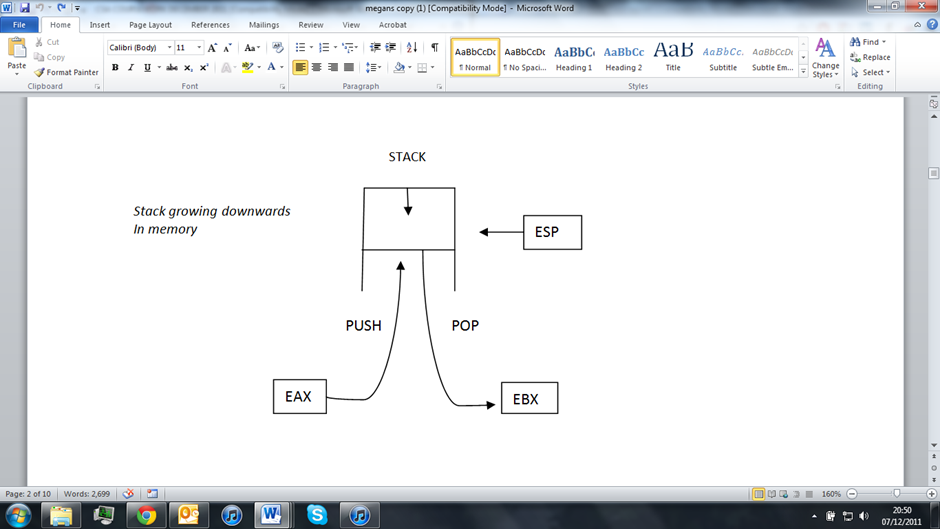
The parameters must be accessed from the stack using the latest added item. The following lines access the parameter:

lea eax,fname

push eax

call DWORD PTR (fopen)

Firstly ‘fname’ is loaded into memory using 4 bytes of space. Then the function fopen is called and is allocated another 4 bytes of memory. This executes the specified file opening ready to be read from. This is stored in eax.



Clearing the Stack Frame

The below code can be used to scrub parameters off the stack.

add esp,8

Although similar, the below code has a slight different function to the line above. The above code clears 8 bytes of data from the stack, whereas the below code only clears 4 bytes of data. This is achieved in both cases by moving the stack pointer up, so all data below it is ignored.

add esp,4

Passing Back the Return Value

or eax,eax

jnz loadComPort

\_\_asm { loadComPort:

mov DWORD PTR (fileptr),eax

The first line checks that the file has been loaded correctly by checking it against itself and making sure the value stored for the file does not equal 0. If eax is empty the program will run and output an error and close down.

If it does load correctly and the value stored is not zero it will jump to the subroutine ‘loadComPort’, jumping over the error message and program abortion. The value of the file is held in the last line and is copied to the file pointer, fileptr, so that eax can be cleared without losing its value.

# Outline Description of Encryption Technique

Transmitter

rol al, 2 // Move ASCII value two to the left

ror al, 4 // Move ASCII value four to the right

xor al, 'b' // Xor CHAR against binary value of b

Receiver

xor al, 'b' // Xor CHAR against binary value of b

rol al, 4 // Move ASCII value four to the left

ror al, 2 // Move ASCII value two to the right

Both the code for the transmitter and for the receiver are found within the ‘Transmit Data’ / ‘Receive Data’ sub-routines respectively. Once a character has been found it can be encrypted before being written to the COM port when transmitting, or decrypted after reading the character but before writing it to the new data file. AL is used because the data is located on the A register.

In the above encryption the ROL and ROR commands are used. These shift the ASCII value of the letter along the designated amount. Although this is a very basic encryption technique, it is simple to implement and effective for the purpose. It is rolled back and forth in each for added complexity, although it makes little difference and only the one line is really needed.

Another technique used is the use of XOR-ing the character by the ASCII value of the character ‘b’. The following diagram shows how each binary value of a character is XOR’ed against the binary value of the selected letter to create an encrypted character, and how it is done again to decrypt it.

The Decimal value for ‘H’ = 72 The Binary value for 72 = 0100 1000

The Decimal value for ‘b’ = 98 The Binary value for 98 = 0110 0010

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| ‘H’ | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 |
| ‘b’ | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 |
| XOR | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 |

The XOR value gives 0010 1010 = 42 The ASCII character for this decimal value is ‘\*’.

So when an ‘H’ is read from the document a ‘\*’ will be sent along the COM port. The same is done to decrypt the character the other end:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **‘\*’** | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 |
| **‘b’** | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 |
| **XOR** | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 |

0100 1000 is produced, so the correct character ‘H’ is written into the text file on the receivers end.

The phrase ‘Hello’ should encrypt to ‘\*BELSOSOcr’ using this technique.

When decrypting, it is important that each step is reversed so that the original character is reverted back to its former state before the next encryption method is reversed otherwise only a non-decipherable message would be received by the transmitter.

These methods are still weak compared to the use of password encryption. This is essentially a complex XOR system but rather than using a single character a password is created. The below sudo-code displays how this could be implemented:

SET myPasswordIndex

READ data InputCharacter FROM data file

LOOP WHILE dataInputCharacter NOT EQUAL -1

INCREMENT myPasswordIndex

IF myPasswordIndex GREATER THAN LENGTH of myPassword THEN

SET myPasswordIndex = 0

ENDIF

SET dataInputCharacter = dataInputCharacter XOR myPassword[myPasswordIndex]

PRINT dataInputCharacter to COMport

READ dataInputCharacter FROM dataFile

END LOOP

**References**

*1. Williams, R. (1998) 'The Ix86 Assembler' , Vol 3,*

*2. Walters, J. () 'Encryption, a Brief Introduction' UfcEHV-20-1 CSA: Course Work, December 2011, pp. 2-5*

*3. Williams, R. (2006) Computer Systems Architecture. 2nd ed. Essex, England: Pearson Education Limited*

*4. Wikipedia, . () 'X86 Assembly/Print Version' [online] Available at: <http://en.wikibooks.org/wiki/X86\_Assembly/Print\_Version> [Accessed 30-11-2011]*

*5. Wikibooks, . ()x86 Disassembly/Functions and Stack Frames' [online] Available at: <http://en.wikibooks.org/wiki/X86\_Disassembly/Functions\_and\_Stack\_Frames> [Accessed 30-11-2011]*

//############ TRANSMITTER #############

//\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

// C Program

//\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

//============= Declarations ===========

#undef UNICODE // Turns off Unicode

#include <stdio.h> // Library for basic commands

#include <conio.h> // Library to create text user interface #define EOT 0x40 // Defines End of Transmission character (@)

//======================================

int main(void) {

//============= Declarations ===========

char wtmode[] = "wb"; // Parameter for write

char rdmode[] = "rb"; // Parameter for read

char fname[] = "W:\\TFile.txt"; // File location

char pname[] = "COM1"; // Name sending computer

char error1[] = "data file failed to open\n"; // Error message

char error2[] = "com port failed to open\n"; // Error message

FILE \* fileptr; // Handles COM port

FILE \* portptr; // Handles COM port

//======================================

//\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

// ASM Program

//\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

//=========== Data File ================

//-------Load Data File to Read---------

\_\_asm { loadDatFile:

lea eax,rdmode // Parameters onto stack

push eax // Move to stack

lea eax,fname // Load data file onto eax

push eax // Move to stack

call DWORD PTR (fopen) // Open data file for reading

add esp,8 // Scrub parameters off stack

or eax,eax // Check file opened OK

jnz loadComPort // Skip to loadComPort

}

//--------------------------------------

//--------Error with Data File----------

\_\_asm { dataFileError:

lea eax,error1 // Move error message to eax

push eax // Move to stack

call DWORD PTR (printf) // Fail to open file

add esp, 4 // Scrub stack pointer off stack

jmp endit // Abort program

}

//--------------------------------------

//======================================

//============ COM Port ================

//------------Load COM Port-------------

\_\_asm { loadComPort:

mov DWORD PTR (fileptr),eax // Save file handle

lea eax,wtmode // Parameters onto stack

push eax // Move to stack

lea eax,pname // COM name onto stack

push eax // Move to stack

call DWORD PTR (fopen) // Open COM port for writing

add esp,8 // Scrub parameters off stack

or eax,eax // Set Z flag if eax is 0

jnz portOK // Skip to portOK

}

//--------------------------------------

//--------Error with Com Port----------

\_\_asm { comError:

lea eax,error2 // Move error message to eax

push eax // Move to stack

call DWORD PTR (printf) // Fail to open port

add esp, 4 // Scrub stack pointer off stack

mov eax, DWORD PTR (fileptr) // Clear file handle

push eax // Move to stack

call DWORD PTR (fclose) // Close data file from reading

add esp,4 // Scrub stack pointer off stack

jmp endit // Abort program

}

//--------------------------------------

//======================================

\_\_asm { portOK:

mov DWORD PTR (portptr),eax // Save port handle

}

//========== Transmit Data =============

//-------------Get Letter---------------

\_\_asm { getLetter:

mov eax, DWORD PTR(fileptr) // Copy char on file to eax

push eax // Move to stack

call DWORD PTR (fgetc) // Get next char from data file

}

//--------------------------------------

//------------Encrypt Letter------------

\_\_asm { encryptLetter:

rol al, 2 // Move ASCII value two to the left

ror al, 4 // Move ASCII value four to the right

xor al, 'b' // Xor CHAR against binary value of b

}

//--------------------------------------

//---------Check For End Of File--------

\_\_asm { endOfFile:

add esp,4 // Scrub stack pointer off stack

cmp eax, EOF // Check for end of data

jz lastChar // Skip to lastChar if no more data

}

//--------------------------------------

//--------Send Character & Loop---------

\_\_asm { sendChar:

mov ebx,eax // Load ebx with the value in eax

mov eax, DWORD PTR (portptr) // Clear port handle

push eax // Move to stack

push ebx // Move to stack

call DWORD PTR(fputc) // Send char to COM port

add esp,8 // Scrub stack pointer off stack

jmp getLetter // Loop back to getLetter

}

//--------------------------------------

//-------------End Of File--------------

//--------Transmit Last Character-------

\_\_asm { lastChar:

mov eax, DWORD PTR (fileptr) // Clear file handle

push eax // End of data so

mov eax, EOT // Send terminating sentinal

push eax // Move to stack

call DWORD PTR (fputc) // Get last char from data file

add esp, 8 // Scrub stack pointer off stack

}

//--------------------------------------

//-----------Close Transmitter----------

\_\_asm { closeTransmitter:

mov eax, DWORD PTR(fileptr) // Clear file handle

push eax // Move to stack

call DWORD PTR (fclose) // Close data file

add esp,4 // Scrub stack pointer off stack

mov eax, DWORD PTR(portptr) // Clear port handle

push eax // Move to stack

call DWORD PTR (fclose) // Flush and close COM port

add esp,4 // Scrub stack pointer off stack

endit:

}

//--------------------------------------

return 0; // Close program

} // End of C

//############# RECEIVER ###############

//\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

// C Program

//\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

//============= Declarations ===========

#include <stdio.h> // Library for basic commands

#include <conio.h> // Library to create a text user interface

#define EOT 0x40 // Defines End of Transmission character (@)

//======================================

int main(void) {

//============= Declarations ===========

char wtmode[] = "wb"; // Parameter for write

char rdmode[] = "rb"; // Parameter for read

char fname[] = "W:\\RFile.txt"; // File locatiion

char pname[] = "COM2"; // Name sending computer

char error1[] = "data file failed to open\n"; // Error message

char error2[] = "com port failed to open\n"; // Error message

FILE \* fileptr; // Handles COM port

FILE \* portptr; // Handles COM port

//======================================

//\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

// ASM Program

//\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

//=========== Data File ================

//-------Load Data File to Write--------

\_\_asm { loadDatFile:

lea eax,wtmode // Parameters onto stack

push eax // Move to stack

lea eax,fname // Load data file onto eax

push eax // Move to stack

call DWORD PTR (fopen) // Open data file for reading

add esp,8 // Scrub parameters off stack

or eax,eax // Check file opened OK

jnz loadComPort // Skip to loadComPort

}

//--------------------------------------

//--------Error with Data File----------

\_\_asm { dataFileError:

lea eax,error1 // Move error message to eax

push eax // Move to stack

call DWORD PTR printf // Fail to open file

add esp, 4 // Scrub stack pointer off stack

jmp endit // Abort program

}

//--------------------------------------

//======================================

//============ COM Port ================

//------------Load COM Port-------------

\_\_asm { loadComPort:

mov DWORD PTR (fileptr),eax // Save file handle

lea eax,rdmode // Parameters onto stack

push eax // Move to stack

lea eax,pname // COM name onto stack

push eax // Move to stack

call DWORD PTR (fopen) // Open COM port for writing

add esp,8 // Scrub parameters off stack

or eax,eax // Set Z flag if eax is 0

jnz portOK // Skip to portOK

}

//--------------------------------------

//--------Error with Data File----------

\_\_asm { comError:

lea eax, error2 // Move error message to eax

push eax // Move to stack

call DWORD PTR (printf) // Fail to open port

add esp, 4 // Scrub stack pointer off stack

mov eax, DWORD PTR (fileptr) // Clear file handle

push eax // Move to stack

call DWORD PTR (fclose) // Close data file from reading

add esp,4 // Scrub stack pointer off stack

jmp endit // Abort program

}

//--------------------------------------

//======================================

\_\_asm { portOK:

mov DWORD PTR (portptr),eax // Save port handle

}

//========== Transmit Data =============

//-------------Get Letter---------------

\_\_asm { getLetter:

mov eax, DWORD PTR (portptr) // Copy char on port to eax

push eax // Move to stack

call DWORD PTR (fgetc) // Get next char from data file

}

//--------------------------------------

//------------Decrypt Letter------------

\_\_asm { decryptLetter:

xor al, 'b' // Xor CHAR against binary value of b

rol al, 4 // Move ASCII value two to the right

ror al, 2 // Move ASCII value four to the left

}

//--------------------------------------

//-----Check For End Of Transmission----

\_\_asm { endOfTransmission:

add esp,4 // Scrub stack pointer off stack

cmp eax, EOT // Check for end of data

jz lastChar // Skip to lastChar if no more data

}

//--------------------------------------

//--------Write Character & Loop--------

\_\_asm { writeChar:

mov ebx,eax // Load ebx with the value in eax

mov eax, DWORD PTR (fileptr) // Clear file handle

push eax // Move to stack

push ebx // Move to stack

call DWORD PTR (fputc) // Write char to file

add esp,8 // Scrub stack pointer off stack

jmp getLetter // Loop back to getLetter

}

//--------------------------------------

//-------------End Of File--------------

//--------Receive Last Character--------

\_\_asm { lastChar:

mov eax, DWORD PTR (fileptr) // Clear file handle

push eax // End of data so

mov eax,EOT // Send terminating sentinal

push eax // Move to stack

call DWORD PTR (fputc) // Get last char from data file

add esp, 8 // Scrub stack pointer off stack

}

//--------------------------------------

//------------Close Receiver------------

\_\_asm { closeReceiver:

mov eax, DWORD PTR (fileptr) // Clear file handle

push eax // Move to stack

call DWORD PTR (fclose) // Close data file

add esp,4 // Scrub stack pointer off stack

mov eax, DWORD PTR (portptr) // Clear port handle

push eax // Move to stack

call DWORD PTR (fclose) // Flush and close COM port

add esp,4 // Scrub stack pointer off stack

endit:

}

//--------------------------------------

return 0; // Close program

} // End of C