# Table of Contents

Introduction ........................................................................................................................................... 3

Technical Background .......................................................................................................................... 3

Laboratory Exercises ............................................................................................................................. 4

1. C Programming Language .................................................................................................................. 4
   1.1. §4.2: Printing arguments ............................................................................................................... 4
   1.2. §4.3: Simple Calculator ............................................................................................................... 4
   1.3. §4.4: Buffering data ...................................................................................................................... 6
   1.4. §4.5: Linking libraries .................................................................................................................. 7

2. Assembly Language Programming ..................................................................................................... 10
   Exercise #4 Program structure and basic I/O ....................................................................................... 10
   Exercise #5 ........................................................................................................................................ 12
   Exercise #6 ........................................................................................................................................ 14

Conclusions ........................................................................................................................................... 16

Appendix A ............................................................................................................................................ 17
   A.1 Flowchart of function calculate of exercise §4.3 ........................................................................... 17
   A.2 Flowchart of assembly exercise #5 ............................................................................................... 18

Appendix B ............................................................................................................................................ 19
   B.1 §4.2 printing arguments ............................................................................................................... 19
   B.2 §4.3: Simple Calculator ............................................................................................................... 19
   B.3. §4.4 : Buffering data ................................................................................................................... 22
   B.4. §4.5: Linking libraries ................................................................................................................ 23

Appendix C ............................................................................................................................................ 25
   C.1. Program structure and basic I/O ................................................................................................. 25
   C.2. Four function calculator ............................................................................................................. 27
   C.3. Sorting numbers .......................................................................................................................... 29

References ............................................................................................................................................ Error! Bookmark not defined.
Introduction

C and assembly programming languages provide access to very low levels of control for a device. They are very important when basic operations need to be performed in a device with limited resources, like an embedded system. The creation of programs with very small memory image, the possibility of highly optimized code and avoiding unnecessary overhead are some of the advantages provided by them and make them popular for embedded system design.

In this report, basic concepts of C and assembly are described. This is achieved through presenting code written for the lab exercises. In the Section 1, operations with pointers, memory allocation and library linking are presented while in Section 2, subroutines, conditional statements and stack operations are shown.

Technical Background

Two tools are essential when programming high level languages like C. The first is an editor that facilitates code development and the second is the compiler. The term ‘compiling’ also refers to the process of linking. Specifically, compiling translates the C program in binary code, while linking provides a connection between the program and external code found in other libraries. These libraries might reside somewhere on the system as executable code, or might be statically embedded in the program.

All C exercises were written in Bloodshed Dev-C++ version 4.9.9.2. Dev-C++ is an editor that allows the creation of projects or compilation of individual files. In both cases command line arguments can be passed to the program through the GUI. Linking external libraries are also possible through it. Dev-C++ uses compiler gcc (minGW) version 3.4.2 as its default compiler. This compiler was used for all the exercises. The working machine is an Intel Core 2 Duo processor running a 32-bit version of Windows 7.

When developing assembly code an assembler is needed. The assembler provides functionality equivalent to that of the compiler. Of its main operations are the translation of commands to binary code and resolving symbolic names (data labels, subroutine names etc) to addresses.

In this report, assembly code for the 68K family of microprocessors was developed. Easy68K v5.6.1 was used on the aforementioned machine. Easy68K is an editor and assembler. It also includes a simulation environment with a GUI (Sim68K v5.8.2) for running and debugging programs. This allows us running programs without loading them in a real 68K microprocessor.

Ollydbg v.2.00.01 was used for debugging of the library modules as is discussed in section 1.4 of this report.
Laboratory Exercises

Section 1.  C Programming Language

1.1. §4.2: Printing arguments

In this exercise, argc and argv main’s parameters are used to pass data to the program that are later printed on the command line. The code is given in Appendix B.1. The arguments typed in when calling the program are passed to it by the command line interpreter. They are separated using the space delimiter. In memory, they are converted to an array of null terminated strings. In the program, argv denotes the start of the array and argc the number of the elements (number of arguments). In detail: argv[0] contains the name of the calling program, argv[1] the first variable, argv[2] the second etc. Argv[argc] is a null pointer.

Function main() can be declared in two ways:

1) int main(int argc, char* argv[])
2) int main(int argc, char** argv)

In the first case, argv is an array of pointers that point at the beginning of each argument. In the second, argv is a pointer to a pointer to the first argument. In both cases we can use printf() function for printing a string by passing to it the pointer of the first character of each argument i.e.:

printf("%s\n", argv[i]);

where i = 1,2,..., argc-1 and character “\n” includes a new line after each character. The output of this program when given the parameters “printing command line arguments” is shown in Figure 1.1.

![Figure 1.1. Output of program for exercise §4.2 (printing arguments)](image)

1.2. §4.3:C Calculator

For this exercise, arguments are not passed from the command line because of a significant obstacle. The command line interpreter substitutes the asterisk (*) with the list of the contents of the current directory, therefore multiplication is impossible to achieve. Input is given after the initialization and the supplied data are stored in a memory location allocated with the use of malloc(). A termination character is appended at the end of the data. This procedure is performed in a separate routine (get_input()) which is also used in later exercises.

All the calculations take place in the recursive routine calculate(). The latter accepts as an argument a pointer to a pointer to a char. This specific type of parameter was used to accommodate the implementation of a parser. This parser traverses the given string and performs the equivalent calculations. It also jumps to different routines. If a simple pointer to a char is used then each time the execution jumps to a different routine a new memory location will be created to hold the address of the current character. Returning from this routine would mean that all the information about the current position in memory is lost. On the other hand, using the same memory
location to hold the current position for all the routines, calculation is effectively progressing in the string, one character at a time. The difference between a pointer to a char and a pointer to a pointer to a char is depicted in Figure 1.2.

![Diagram](image)

**Figure 1.2. Difference between a pointer to a char and a pointer to a pointer to a char**

The code for this exercise is shown in Appendix B.2. First, a memory space that will hold the user data is allocated and cleared,

```c
buffer = (char*) malloc(BUF_MAX);
if (!buffer) //check if buffer allocated
    exit(printf("buffer not allocated"));
memset (buffer,'\0',BUF_MAX); //clear the memory location
```

Having defined,

```c
#define BUF_MAX 256
```

Next, function `get_input()` is used for filling the memory location with the data supplied by the user. Inside this function, we use `getchar()` for placing the character given by the user in the current position and then advance the position by one. After enter is pressed, “new line” character is replaced by null pointer,

```c
while(--nchars > 0 && *(buffer-1) != '\n' ){
    *(buffer++) = getchar(); //get one character given by the user
}
*(buffer-1) = '\0'; //null terminate string
```

Once the string is ready we use it to call function `calculate()`, whose flowchart is given in Appendix A.1.

As already mentioned, this is a recursive function. It goes through the string and checks each character before doing the equivalent calculations. Function `myatol()` is not explicitly mentioned in the flowchart. This function converts a part of the string where a number lies to its numeric value so that this number can then be added/subtracted etc. from the previous result. It does this by filling up a different string with up to 5 ASCII digits, and then calling STL function `atol()` passing it this string. It returns the result of the latter, which is the equivalent numeric value of the given string.

The calculator code that was developed has some limitations, i.e., division by zero is not handled. Also, it does not support operator precedence, so operations are performed from left to right. A more flexible way to implement it would be with the use of a stack where numbers are pushed before or after their operation to be performed between them (postfix/prefix).
Operation of the calculator is verified in figure 1.3.

1.3. §4.4: Buffering data

This simple exercise mostly contains code that has already been written for the previous programs. The complete code is given in Appendix B.3. First, a memory space of BUF_MAX bytes is allocated. Function `get_input()` is used to place the characters supplied by the user inside this memory location and null terminate the string. Finally, `strtok()` is used to split the string in separate words using the space delimiter and count the number of words in this sentence. This last action is performed in the following section,

```c
slide = strtok (buffer," ");  //split the string at first instance of <space>
while (slide != NULL)
{
    slide = strtok (NULL, " ");  //split the string at next instance of <space>
    wordn++;
}
```

Variable `slide` (pointer to a char) is used for pointing at the current position in the string (the first character of each word). Each time `strtok()` is called, one word is split from the sentence and null terminated to create a new string. Declaration of the string is performed only at the first call. Consecutive calls use the same string. If `strtok()` returns a null character, there are no further words in the sentence. The output is shown in Figure 1.4.

![Figure 1.3. Output for exercise §4.3 (calculator)](image1)

![Figure 1.4. Output of program for exercise §4.4 (buffering data)](image2)
1.4. §4.5: Linking libraries

In exercise §4.5, an executable library was used for encrypting and decrypting a sentence given by the user with a key, again given by the user. The code is shown in Appendix B.4.

For the encryption and decryption operations, library libcrypto was used which was statically linked in the program. This means that after compilation, the file libcrypto.a is not needed for running the program because its code is embedded in the final executable. To use this library the header file that contains the declarations of the variables and functions must be included (#include "crypto.h" statement). Also, for the compilation process, the executable version of the library must reside in the same directory as the source code so that the linker can embed it in the executable code.

Referring to the code in Appendix B.4, and similar to the previous exercises, the same operations of allocating space for user data is shown in Table 1.

```c
/*allocate space for input, output, and key strings*/
if(!(input = (char*) malloc(BUF_MAX)))
  exit(printf("error in malloc"));
memset (input,"\0",BUF_MAX);
if(!(output = (char*) malloc(BUF_MAX)))
  exit(printf("error in malloc"));
memset (output,"\0",BUF_MAX);
if(!(key = (char*) malloc(BUF_MAX)))
  exit(printf("error in malloc"));
memset (key,"\0",BUF_MAX);
```

Table 1. Space allocation for I/O and key strings

Function get_input() from §4.3 was again used for filling the memory allocated with user supplied data. Encryption function is called in the following way,

```c
/*encrypt the input using the key*/
ret = perform_function(input, output, key, (strlen(input)+1 << 8)|ENABLED|ENCRYPT);
printf("Encrypted: %s\n", output);
```

where input is encrypted using key and stored in output. The OR operation is used for creating the opcode. ENABLED and ENCRYPT are defined in crypto.h. The opcode needs the length of the input therefore strlen() is used to count the characters in it. This function does not count the null character at the end which is needed, so 1 is added. At last, the result is shifted 8 bytes to the left, because it must fill the second and third bytes of the 32-bit opcode.

The perform_function(), when given the ENCRYPT option, takes one character of key and one of input and adds their ASCII values. We can check this in the disassembly output of our executable, as shown in figure 1.6 (1). The local variables and arguments are as listed in table 1.1. The disassembly output also reveals an omission from the code that can be best described after running the program (figure 1.5).
As is observed in the output, only the first few characters of the text are encrypted. This happens because the library does not check whether the end of the key parameter is reached. It only checks whether there is an inconsistency between the input data and the size parameter given when calling the function (figure 1.6 (2)) and whether the end of the input string has been reached (Figure 1.6 (3)).
Because the memory location that holds the key is cleared before filled, after the parsing of \textit{key} finishes, encryption performs addition of one character of the input with zero. This produces a text where only the first $n$ characters are encrypted, with $n$ being the number of characters of key (excluding the null terminating character). If the same procedure is reproduced using the \textit{argc} and \textit{argv} arguments, it completes successfully because after the end of key, non-zero characters follow, which take part in the addition. This is essentially a type of buffer overrun in the sense that characters outside the ‘buffer’ are used.
Section 2. Assembly Language Programming

Exercise #4 Program structure and basic I/O

As an introduction to 68K assembly language, a program that performs basic I/O operations was devised. The purpose is to acquire familiarity with a) subroutines, b) data types and basic storage operations and c) software traps. The code is shown in Appendix C.1. The execution flow is very simple: A character, a string and a number are read from the keyboard in sequence and stored in predefined memory locations. They are then printed on the command line with the same order. The output is depicted in Figure 2.1.

![Figure 2.1. Output of program for exercise 4: Program structure and basic I/O](image)

This code is used as an example for describing the basic structure of a 68K assembly program. The program starts with the origin directive (ORG) which denotes the starting address of the code in memory. The previous EQU expressions have no effect on the final executable. The assembler replaces the labels defined by them with the equivalent values. START is a label to the beginning of the program and can be replaced by the ORG address. The following commands halt the processor,

```
MOVE.B #9,D0
TRAP #15
```

While the command,

```
STOP #$2000
```

loads 2000₁₆ in the status register and stops the 68000. The status register contains flags from which the microprocessor derives information about the state of the program when the latter stopped its execution.

The subroutines follow the core section. These are placed in a non-consecutive memory location. The memory segments can be observed using the Sim68K debugger (Figure 2.2). As can be seen, the code for the main routine is loaded at address $1000, while the code of the subroutines starts at address $600.
By changing the starting address of the subroutines an interesting observation is made (Figure 2.3). The assembler does not check if there are duplicate addresses to be loaded. It seems that the code of get_num subroutine overwrites the main routine’s code. If the code in Figure 2.3 is run, the simulator stops the execution at the starting address of get_num, blocking the program.

The subroutines are immediately followed by the data in memory. It would, however, be more useful to place them in a different non-consecutive memory space. Read only/write operations could later be applied to these locations.

**Subroutines**

Subroutines are very useful for creating more structured and readable code. They begin with a keyword that identifies them (i.e. put_char). They are called using the branch to subroutine command (BSR <name of subroutine>) which transfers execution to the address where the routine resides. The BSR command also stores the address of the next instruction to be executed on the stack so that the microprocessor knows where to return at the event of an RTS command (return from subroutine).
Data types

As already mentioned, this program records the input of three data types: a character, a string and a number. These are stored in three memory locations of 1, 80 and 4 bytes respectively. The string is not necessarily taking up all of the space allocated to it, but the 80 bytes represent an upper limit to the number of characters that will be read. Also, since 4 bytes are allocated for a number, to guarantee correct operation of the program, the number supplied by the user should not be greater than that fitting in 32 bits.

To allocate space for storing data, the assembler directive ‘define storage’ (DS) is used in combination with a number that defines the length of the memory location. For example, ‘DS.B 80’ allocates space of 80 bytes, while ‘DS.L 80’ allocates 80x4 bytes. Each memory location has a label associated with it that can be used instead of it in the code. The memory allocation can be combined with data assignment using the ‘define constant’ directive (DC). In this case, the data will be placed in their position during loading of the program.

Software traps

The I/O operations in our code (read a character, write a character, read a string etc) are performed with the help of traps. A trap caused by a program is called a software trap to distinguish it from traps caused by hardware. When we call them execution is switched to privileged mode. An advanced set of actions can be performed there.

To inform the processor of which trap is required, the TRAP command is used followed by the number of the trap. For example, command ‘TRAP #15’ calls the trap for I/O operations. The exact I/O operation is defined by placing the number of the specific task in a register that depends on the number of trap.

Each task has a service routine associated with it. The parameters needed by the service routine are placed in registers defined by the processor specifications. For example, in order to display a single character on the command line, the task number #6 is placed on D0 register and the character to be printed is placed in the lower byte of D1. TRAP #15 is then issued. The processor sees the I/O request, switches to privileged mode and searches a table of service routines to find the one for task #6. It then uses the character placed in D1.B as the parameter for the service routine which prints it on the command line. Similar operations are performed to read a character supplied by the user. This time the trap does not require any parameters, instead 68000 returns the character in a predefined register.

Exercise #5

In this exercise, a simple calculator was created to examine conditional statements and branching in 68000. Its function is very basic and can be observed in the flowchart of Appendix A.2. The code is listed in Appendix C.2. The program first requests for one input at a time, namely: first number, operator and second number. If the first number is zero then the program exits after printing a goodbye message. After the calculation is performed, the program loops requesting a second operation, until zero is pressed. If the operator is not recognized, the function is not performed and looping is repeated. The program does not handle division by zero and non-integer division results.

The BSR command for branching to subroutines has already been examined. A similar command, ‘branch always’ (BRA), is used for jumping to commands inside the main program. The difference here is that there is no RTS command because execution never returns to the initial point; instead it carries on from there. This also means that no storing of the instruction pointer is required.

Branching commands in this program are used in conjunction with conditional statements in order to check the input supplied and take appropriate actions. For example, in the following code excerpt:
the value of register D1 is compared with the ‘+’ sign using ‘branch if equal’ (BEQ) command. If the condition is true then execution continues at the address where addition is performed. Similar consecutive statements can be combined to form ‘if-else’ cases.

Conditional operations in assembly are performed with the help of the condition codes that can be found at the lower bits of the status register. For example, as can be seen in Figure 2.4, the Z flag of the status register is raised after the ‘CMP.B ‘+’ , D1’ command.

After the conditional statements execution is passed to the correct code segment that performs the desired operation. The result is printed using the same put_num subroutine and a software trap as was described in the previous exercise. The complete output of the program is shown in Figure 2.5.

---

**Figure 2.4.** Z flag is raised after the execution of ‘CMP.B ‘+’ , D1’ command
Exercise #6

In this exercise, the stack is used for storing and retrieving data. The stack is nothing more than consecutive memory locations used by the program for temporary storage. When compiling higher level language programs stack frames are created whenever a new function is called. The starting address of the stack frame is also stored on the stack. This allows the previous stack frame to be restored after the function returns. This operation is not necessary for this exercise, however, a basic knowledge of the stack structure is needed.
Being a LIFO structure (figure 2.6), data on the stack are always pushed to and popped from the ‘top’. New data are stored to lower addresses. In other words, the address of the top of the stack is decreased as new data are pushed to it. Register A7 is used for pointing the top of the stack. In this exercise, a number is pushed on the stack using the following command,

\[
\text{MOVE.L D1, -(A7)} \quad \text{*push the number in the stack and auto-decrement}
\]

Where D1 contains the number returned from calling the “read number from keyboard” subroutine.

Register A7 is contained in brackets. This means that the contents of D1 will be moved in the address that A7 contains and not in A7 itself. The minus sign auto-decrements the value of A7 by the size that the extension of the MOVE instruction dictates (4 bytes for long-word, 2 for word etc).

The aforementioned instruction is contained within a loop that counts six repetitions before exiting (get_all). A number is read from the keyboard and pushed on the stack on each iteration. The loop is constructed with the help of the conditional instructions described in the previous exercise. A second loop pops the numbers that were pushed on the stack one by one and prints them. Doing this, the numbers are printed in reverse order since, as already mentioned, under regular operation a stack is only read from the ‘top’. The pop operation is performed with the following instruction,

\[
\text{MOVE.L (A7)+, D1} \quad \text{*pop from the stack}
\]

This is the reverse operation, namely, pop the data and increment the stack pointer. It is not mandatory that data are read from the top of the stack. Indirect addressing with an offset can also be used for accessing the stack, which leaves its contents as they are. For example, to read the third number from the stack the following command could be used,

\[
\text{MOVE.L 2(A7), D1} \quad \text{*read third number on the stack}
\]

Execution of the program finishes after printing all the numbers on the command line. The complete output can be seen in Figure 2.7.

![Sim86K I/O](image)

Write six numbers separated by [enter]:
1 2 3 4 5 6

The numbers you entered are (in reverse order):
6 5 4 3 2 1

Figure 2.7. Output of assembly exercise 3: Sorting numbers
Conclusions

Some basic features of C and assembly languages were presented through the development of the laboratory exercises. It became apparent that both languages provide a low level of control which makes them suitable to embedded systems programming. Assembly is more flexible in controlling system resources, but C requires less programming effort and produces less amount of code for the same functionality. On the other hand, C is more dependent on the use of libraries, which produces either less portable code, or, when choosing static linking, executables of higher size. The decision of one language over the other depends on the system and requirements at hand.
Appendix A

Flowchart schematics

A.1 Flowchart of function *calculate* of exercise §4.3

```
START
res = 0
Current position = first position

res
Calculate numeric value and advance

Current position = '0'?
YES
NO

Current position = '+'?
YES
NO

Current position = '-'?
YES
NO

Current position = '*'?
YES
NO

Current position has digit?
YES
NO

Next position = '('?
YES
NO

Next position has digit or space?
YES
NO

res = 0

Current position = first position

YES
NO

END

RES = numeric value of next

YES
NO

END

ERROR
```
A.2 Flowchart of assembly exercise #5

START

print initial messages

Input First number

<enter> pressed?

YES

NO

Input operator

Input second number

Addition? NO subtraction? NO multiplication? NO division? NO

YES YES YES YES

Add numbers Subtract numbers Multiply numbers Divide numbers

Print result

END
Appendix B

C Programming exercises

B.1 §4.2 printing arguments

```c
#include <stdio.h>

int main(int argc, char** argv) {
    int i = 0;
    //initialize counter
    while( ++i < argc )
        //skip argv[0] (name of the program)
    printf("%s\n", argv[i]);
    //print each argument in separate line
    return 0;
}
```

B.2 §4.3: Simple Calculator

```c
#include <stdio.h>
#include <stdlib.h>
#define BUF_MAX 256
/*=======================================================================
 4 function calculator supporting brackets
Limitations:
- There is no precedence, calculations are done from left to right
- Will get wrong result if somewhere in between operations result becomes greater than MAX_SIGNED_LONG
- Does not handle division by 0 or by an empty parenthesis
- Will get wrong result if not putting a sign before parenthesis
=======================================================================*/

/*
This function takes as argument a memory location of BUF_MAX size and fills it with the data supplied by the user
*/
void get_input(char* buffer) {
    int nchars = BUF_MAX - 1;

    /*get one character at a time until enter is hit, or BUF_MAX-2 is reached*/
    *(buffer++) = getchar();
    while(--nchars > 0 && *(buffer-1) != \n) {
        *(buffer++) = getchar();
    }

    *(buffer-1) = \0;
    /*null terminate string*/

    return;
}

/*=======================================================================
Converts a string that starts at the position shown by exp to its equivalent numeric value. Stops when there are no further numbers, or when five digits have been parsed.
=======================================================================*/
long myatol(char** expp){
```
char restr[6];  /*store the string that will be converted to number*/
char* slide;  /*pointer to current position*/
slide = restr;
int i = 0;

*slide = '\0';
while(**expp == ' '){
    *(expp)++;
}  //skip spaces

/*get only 5 numbers (positions 0-4 in restr) position 5 is \0*/
while(**expp >= '0' && **expp <= '9' && ++i <= 5)
    *slide++ = *(*expp)++;

*slide = '\0';  //null terminate string

return atol(restr);  //convert it to long and return it
}

long calculate(char** expp){
    long res;  //stores the result
    char expch;  //expression character: temporarily store current character
    res = 0;

    if((**expp >= '0' && **expp <= '9') || **expp == ' ')
        res = myatol(expp);  //find number and store it in res

    while(**expp != '\0'){
        expch = **expp;
        *(expp)++;
        switch (expch) {
            /*if true then parenthesis is either at the start of the expression or following a number*/
            case '(':
                res = calculate(expp);
                break;
            /*return after recursion. this also means that if ')' is given without '(', function returns abruptly*/
            case ')':
                return res;
            /*if end of string do nothing, case is not executed again*/
            case '\0':
                break;
            /*------ADDITION------*/
            case '+':
                if(**expp == '(')
                    res += calculate(expp);
                else if((**expp >= '0' && **expp <= '9') || **expp == ' ')
                    break;
                else if((**expp == '1')
                    res += calculate(expp);
                else
                    break;
            /*------MULTIPLICATION------*/
            case '*':
                if(**expp == '(')
                    res *= calculate(expp);
                else if((**expp >= '0' && **expp <= '9') || **expp == ' ')
                    break;
                else
                    break;
            /*------SUBTRACTION------*/
            case '-':
                if(**expp == '(')
                    res -= calculate(expp);
                else if((**expp >= '0' && **expp <= '9') || **expp == ' ')
                    break;
                else
                    break;
            /*------DIVISION------*/
            case '/':
                if(**expp == '(')
                    res /= calculate(expp);
                else if((**expp >= '0' && **expp <= '9') || **expp == ' ')
                    break;
                else
                    break;
            /*------EXPONENT------*/
            case '^':
                if(**expp == '(')
                    res **= calculate(expp);
                else if((**expp >= '0' && **expp <= '9') || **expp == ' ')
                    break;
                else
                    break;
            /*------MODULUS------*/
            case '%':
                if(**expp == '(')
                    res %= calculate(expp);
                else if((**expp >= '0' && **expp <= '9') || **expp == ' ')
                    break;
                else
                    break;
            /*------LOGARITHMS------*/
            case 'log':
                if(**expp == '(')
                    res = log(calculate(expp));
                else if((**expp >= '0' && **expp <= '9') || **expp == ' ')
                    break;
                else
                    break;
            /*------TRIGONOMETRIC------*/
            case 'sin':
                if(**expp == '(')
                    res = sin(calculate(expp));
                else if((**expp >= '0' && **expp <= '9') || **expp == ' ')
                    break;
                else
                    break;
            case 'cos':
                if(**expp == '(')
                    res = cos(calculate(expp));
                else if((**expp >= '0' && **expp <= '9') || **expp == ' ')
                    break;
                else
                    break;
            case 'tan':
                if(**expp == '(')
                    res = tan(calculate(expp));
                else if((**expp >= '0' && **expp <= '9') || **expp == ' ')
                    break;
                else
                    break;
        }
    }
    return res;
}

long myatol(char** expp){  //convert string of numeric characters to a long
    long res;  //stores the result
    while(**expp >= '0' && **expp <= '9')
        res = res * 10 + **expp - '0';
    *expp = '\0';  //null terminate string
    return res;
}
res += myatol(expp); // if number go on to find its numeric value and add it to
res
else
exit(printf("error in addition\n");
break;

/******SUBTRACTION------*/
case '-':
if(**expp == '(')
res = calculate(expp); // if parenthesis, recall the function and break
else if((**expp >= '0' && **expp <= '9') || **expp == ' ')
res = myatol(expp); // if number go on to find its numeric value and subtract from res
else if(**expp == ' ');
else
exit(printf("error in subtraction\n");
break;

/******MULTIPLICATION------*/
case '*':
if(res) res = 1; // if first character was * ignore it (multiply by 1)
if(**expp == '(')
res *= calculate(expp); // if parenthesis, recall the function and break
else if((**expp >= '0' && **expp <= '9') || **expp == ' ')
res *= myatol(expp); // if number go on to find its numeric value and multiply by res
else if(**expp == ' ');
else
exit(printf("error in multiplication\n");
break;

/******DIVISION------*/
case '/':
if(**expp == '(')
res /= calculate(expp); // if parenthesis, recall the function and break
else if((**expp >= '0' && **expp <= '9') || **expp == ' ')
res /= myatol(expp); // if number go on to find its numeric value and multiply by res
else if(**expp == ' ');
else
exit(printf("error in division\n");
break;
default:
printf("something is wrong here:%c\n", expch);
}

return res;

int main(){
    long res;
    char* buffer;
    char* exp;
    int i = 2;

    buffer = (char*) malloc(BUF_MAX);
    if (!buffer) // check if buffer allocated
        exit(printf("buffer not allocated"));
memset(buffer, '\0', BUF_MAX); //clear the memory location

while(1){
    printf("Input calculation and press 'enter': ");
    get_input(buffer);
    if(!strlen(buffer)) { //check if no arguments were given
        printf("Bye!\n");
        return 0;
    }

    exp = buffer;
    res = calculate(&exp);
    printf("%s = %d\n", buffer, res);
}

return 0;

B.3. §4.4 : Buffering data

#include <stdio.h>
#include <stdlib.h>
#define BUF_MAX 256

void get_input(char* buffer) {
    int nchars = BUF_MAX - 1;

    /* get one character at a time until enter is hit, or BUF_MAX - 2 is reached*/
    *(buffer++) = getchar();
    while(--nchars > 0 && *(buffer - 1) != '\n'){
        *(buffer++) = getchar();
    }

    *(buffer - 1) = '\0'; //null terminate string
    return;
}

int main() {
    char *slide, *buffer; //buffer holds user input, slider points at current position
    int nchars = BUF_MAX - 1; //helps in not exceeding BUF_MAX
    int wordn = 0; //holds number of words in sentence
    buffer = (char*) malloc(BUF_MAX); //allocate space for user input
    if (!buffer)
        exit(1);

    printf("Tell me something: ");
    get_input(buffer); //get user input
    slide = strtok(buffer, " "); //split string at the instance of <space>
    while (slide != NULL)
    {
        slide = strtok(NULL, " "); //split string at next instance of <space>
        wordn++;
    }
}
```c
void get_input(char* buffer) { 
    int nchars = BUF_MAX - 1; 
    *(buffer++) = getchar(); 
    while(!--nchars > 0 && *(buffer-1) != '\n'){
        *(buffer++) = getchar();  /*get user input*/
    }
    *(buffer-1) = '\0'; /*null terminate the string*/
    return;
}

int main() {
    char *input, *output, *key;
    int ret;
    int nchars = BUF_MAX - 1;

    /*allocate space for input, output, and key strings*/
    if(!(input = (char*) malloc(BUF_MAX)))
        exit(printf("error in malloc");
    memset (input, '\0',BUF_MAX);
    if(!(output = (char*) malloc(BUF_MAX)))
        exit(printf("error in malloc");
    memset (output, '\0',BUF_MAX);
    if(!(key = (char*) malloc(BUF_MAX)))
        exit(printf("error in malloc");
    memset (key, '\0',BUF_MAX);

    /*get user supplied string to be encrypted*/
    printf("Tell me something: ");
    get_input(input);

    /*get key*/
    printf("Give me key: ");
    get_input(key);

    /*encrypt the input using the key*/
    ret = perform_function(input, output, key, (strlen(input)+1 << 8)|ENABLED|ENCRYPT);
    printf("Encrypted: %s\n", output);

    return 0;
}
```

B.4. §4.5: Linking libraries

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "crypto.h"

#define BUF_MAX 0xff

void get_input(char* buffer) {
    int nchars = BUF_MAX - 1;
    *(buffer++) = getchar();
    while(!--nchars > 0 && *(buffer-1) != '\n') {
        *(buffer++) = getchar();  //get user input
    }
    *(buffer-1) = '\0';  //null terminate the string
    return;
}

int main() {
    char *input, *output, *key;
    int ret;
    int nchars = BUF_MAX - 1;

    /*allocate space for input, output, and key strings*/
    if(!(input = (char*) malloc(BUF_MAX)))
        exit(printf("error in malloc");
    memset (input, '\0',BUF_MAX);
    if(!(output = (char*) malloc(BUF_MAX)))
        exit(printf("error in malloc");
    memset (output, '\0',BUF_MAX);
    if(!(key = (char*) malloc(BUF_MAX)))
        exit(printf("error in malloc");
    memset (key, '\0',BUF_MAX);

    /*get user supplied string to be encrypted*/
    printf("Tell me something: ");
    get_input(input);

    /*get key*/
    printf("Give me key: ");
    get_input(key);

    /*encrypt the input using the key*/
    ret = perform_function(input, output, key, (strlen(input)+1 << 8)|ENABLED|ENCRYPT);
    printf("Encrypted: %s\n", output);
    return 0;
}
```
memset(input, '\0', BUF_MAX);

/* decrypt the input using the key */
ret = perform_function(output, input, key, (strlen(output) + 1 << 8) | ENABLED | DECRYPT);
printf("Clear text: %s\n", input);

return 0;
}
Appendix C

Assembly programming exercises

C.1. Program structure and basic I/O

BS   EQU $08   Backspace
HT   EQU $09   Horiz Tab
LF   EQU $0A   Line Feed
VT   EQU $0B   Vert Tab
FF   EQU $0C   Form Feed
CR   EQU $0D   Car Ret

* Read input from keyboard:
* a char <enter>, a number <enter>, a string <enter>
* print the given supplied data in the same order

START    ORG    $1000
LEA prompt1,A1
BSR put_string *print 'Write a character:'
BSR get_char
MOVE.B D1, char *store the input character
BSR put_crlf *line feed
LEA prompt2,A1
BSR put_string *print 'Write a string:'
LEA in_str,A1 *address that the string will be stored
BSR get_string
LEA prompt3,A1
BSR put_string *print 'Write a number:'
BSR get_num
MOVE.L D1, number *store the number
LEA finalmsg,A1
BSR put_string *print 'Your input was: '
BSR put_crlf
MOVE.B char,D1
BSR put_char *print the char
BSR put_crlf
LEA in_str,A1
BSR put_string *print the string
BSR put_crlf
MOVE.L number,D1
BSR put_num *print the number
BSR put_crlf
BRA DONE

END    STOP    #$2000
ORG    $600
<table>
<thead>
<tr>
<th>Routine</th>
<th>Code</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>get_char</td>
<td>MOVE.B #5,D0</td>
<td>*trap for getting a char from the cmd #15</td>
</tr>
<tr>
<td></td>
<td>TRAP</td>
<td></td>
</tr>
<tr>
<td></td>
<td>RTS</td>
<td></td>
</tr>
<tr>
<td>put_char</td>
<td>MOVE.B #6,D0</td>
<td>*subroutine for printing a character #15</td>
</tr>
<tr>
<td></td>
<td>TRAP</td>
<td></td>
</tr>
<tr>
<td></td>
<td>RTS</td>
<td></td>
</tr>
<tr>
<td>get_num</td>
<td>MOVE.B #4,D0</td>
<td>*subroutine for reading a number #15</td>
</tr>
<tr>
<td></td>
<td>TRAP</td>
<td></td>
</tr>
<tr>
<td></td>
<td>RTS</td>
<td></td>
</tr>
<tr>
<td>put_num</td>
<td>MOVE.L #3,D0</td>
<td>*subroutine for printing a number #15</td>
</tr>
<tr>
<td></td>
<td>TRAP</td>
<td></td>
</tr>
<tr>
<td></td>
<td>RTS</td>
<td></td>
</tr>
<tr>
<td>get_string</td>
<td>MOVE.B #2,D0</td>
<td>*subroutine for reading a string #15</td>
</tr>
<tr>
<td></td>
<td>TRAP</td>
<td></td>
</tr>
<tr>
<td></td>
<td>RTS</td>
<td></td>
</tr>
<tr>
<td>put_string</td>
<td>MOVE.B #14,D0</td>
<td>*subroutine for printing a string #15</td>
</tr>
<tr>
<td></td>
<td>TRAP</td>
<td></td>
</tr>
<tr>
<td></td>
<td>RTS</td>
<td></td>
</tr>
<tr>
<td>put_crlf</td>
<td>LEA newline,A1</td>
<td>*subroutine for printing a new line char</td>
</tr>
<tr>
<td></td>
<td>MOVE.L #14,D0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>TRAP</td>
<td></td>
</tr>
<tr>
<td></td>
<td>RTS</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>DC.B</td>
<td>0</td>
</tr>
<tr>
<td>in_str</td>
<td>DS.B</td>
<td>80</td>
</tr>
<tr>
<td>number</td>
<td>DC.L</td>
<td>0</td>
</tr>
<tr>
<td>prompt1</td>
<td>DC.B</td>
<td>'Write a character: ',0</td>
</tr>
<tr>
<td>prompt2</td>
<td>DC.B</td>
<td>'Write a string: ',0</td>
</tr>
<tr>
<td>prompt3</td>
<td>DC.B</td>
<td>'Write a number: ',0</td>
</tr>
<tr>
<td>finalmsg</td>
<td>DC.B</td>
<td>'Your input was: ',0</td>
</tr>
<tr>
<td>newline</td>
<td>DC.B</td>
<td>' ',CR,LF,0</td>
</tr>
</tbody>
</table>

DONE MOVE.B #9,D0
TRAP #15
END START
C.2. Four function calculator

```assembly
BS EQU $08   Backspace
HT EQU $09   Horiz Tab
LF EQU $0A   Line Feed
VT EQU $0B   Vert Tab
FF EQU $0C   Form Feed
CR EQU $0D   Car Ret

START   ORG    $1000

*------------------------START OF INPUT------------------------*

LEA    txt1,A1
BSR    put_string   *print '4-function calculator'
BSR    put_crlf

LEA    txt2,A1
BSR    put_string   *print '    <enter> for exit    '
BSR    put_crlf

INPUT

LEA    txt3,A1
BSR    put_string   *print '---------------------'
BSR    put_crlf

LEA    txt4,A1
BSR    put_string   *print 'Type first number: '

BSR    get_num
MOVE.L D1, number1 *store the number

CMP     #0, D1     *check if enter pressed
BEQ     EXIT       *exit the program

LEA    txt5,A1
BSR    put_string   *print 'Type operator: '

BSR    get_char
MOVE.B D1, opr     *store the operator
BSR    put_crlf

LEA    txt6,A1
BSR    put_string   *print 'Type second number: '

BSR    get_num
MOVE.L D1, number2 *store the second number
BSR    put_crlf

*--------------------------START OF CALCULATION----------------------*

MOVE.B opr,D1
CMP.B #'+', D1    *case addition
BEQ     addition

CMP.B '#-', D1    *case subtraction
BEQ     subtraction

CMP.B '#*', D1    *case multiplication
BEQ     multiplication
```
CMP #',', D1  "case division"
BEQ division

LEA txt8,A1
BSR put_string  "print 'Invalid operator, try again'"
BSR put_crlf
BRA INPUT

addition
MOVE.L number1,D1
ADD.L number2,D1  "perform addition"
BRA print

subtraction
MOVE.L number1,D1
SUB.L number2,D1  "perform subtraction"
BRA print

multiplication
MOVE.L number1,D1
MOVE.L number2,D2
MULU D2,D1  "perform multiplication"
BRA print

division
MOVE.L number1,D1
MOVE.L number2,D2
DIVU D2,D1
BRA print

print
BSR put_num
BSR put_crlf
BRA INPUT

EXIT
LEA txt7,A1
BSR put_string  "print 'Goodbye!'"
BSR put_crlf
BRA DONE

END
STOP #$2000
ORG $600

----------START OF SUBROUTINES----------

get_char
MOVE.B #5,D0
TRAP #15
RTS

get_num
MOVE.B #4,D0
TRAP #15
RTS

put_num
MOVE.L #3,D0
TRAP #15
RTS

put_string
MOVE.B #14,D0
TRAP #15
RTS

put_crlf
LEA newline,A1
MOVE.L #14,D0
TRAP #15
RTS
C.3. Sorting numbers

*--------------------------------------------------
*Exercise 6
*Print six numbers supplied from the keyboard
*in reverse order
*--------------------------------------------------
BS EQU $08   Backspace
HT EQU $09   Horiz Tab
LF EQU $0A   Line Feed
VT EQU $0B   Vert Tab
FF EQU $0C   Form Feed
CR EQU $0D   Car Ret

START    ORG     $1000
LEA    prompt,A1
BSR    put_string  *print 'Write six numbers separated by [enter]:'
BSR    put_crlf

get_all   BSR    get_num  *get a number from keyboard
MOVE.L   D1, -(A7)  *push the number in the stack and autodecrement
SUB.B    #$01,count *decrement loop counter
BNE    get_all    *jump to get_all if counter is zero

LEA    output_msg,A1
BSR    put_string  *print 'The numbers you entered are (in reverse order):'
BSR    put_crlf

put_all   MOVE.B    #$06,count
MOVE.L   (A7)+,D1  *pop from the stack
BSR    put_num    *print the number
BSR    put_crlf   *line feed
SUB.B    #$01,count *loop if necessary
BNE    put_all
BRA    DONE

END    STOP    #$2000
ORG    $600
get_num
    MOVE.B  #4,D0
    TRAP  #15
    RTS

put_num
    MOVE.L  #3,D0
    TRAP  #15
    RTS

put_string
    MOVE.B  #14,D0
    TRAP  #15
    RTS

put_crlf
    LEA    newline,A1
    MOVE.L  #14,D0
    TRAP  #15
    RTS

count
    DC.B   6

prompt
    DC.B 'Write six numbers separated by [enter]: ','0
output_msg
    DC.B 'The numbers you entered are (in reverse order):','0
finalmsg
    DC.B 'Your input reversed: ','0
newline
    DC.B  ','CR,LF,0

DONE
    MOVE.B #9,D0
    TRAP  #15
    END START

Bibliography


