Employee Record Retrieval Program

Development Report

Wade Tregaskis
02557793

Subject: CSE22MAL
Lecturer: Richard Hall

26th September 2003

Contents

1.0 How the Program Works

1.1 Overview

The program is based loosely around how you might construct such a thing in C. As such, there is a sequence of code known as “main”, and several dozen procedures for performing specific tasks, such as reading from and writing to employee database on disk. Execution ultimately loops within the main method until the user chooses to quit.

1.2 Memory

To avoid having to recompile the program in order to adjust the maximum number of records, a pseudo-dynamic memory system was implemented. This consists of a malloc and free function pair which manage memory within a specific space designated for that express purpose. The intention was to allow the runtime memory usage of the program to vary.

However, due to time limitations this ideal was not met. As it turns out, it is rather difficult in the x86 segment model to implement even a simple dynamic memory system. As such, the system as it stands is fairly limited - memory allocation is sequential, with free only being capable of releasing memory from the end of the linear allocation. In other words, the dynamic memory acts like a stack in which you cannot remove items within it without first removing those above.

Nonetheless, it was an interesting exercise, and lent itself naturally to the use of pointers extensively, which resulted in much greater efficiency and performance throughout the program.

1.3 Record format

The record structure was designed for simplicity, whereby all fields but the salary are kept in their string form. This neatly dealt with the large number issues, since for our purposes only the salary need be manipulated numerically. For that purpose, the salary is stored as a 32-bit floating point number. It could have been stored as a string, and converted to and from forms as needed, but given no real disadvantage either way, storing the salary as a 32-bit number saved an extra 4 bytes per record.

As mentioned, the other numerical values could also have been stored in their machine native forms, but this would add extra overhead without any feature benefit - searching and sorting would need to make use of the FPU, rather than simple byte comparisons. If memory usage were more of a consideration, the native data forms would perhaps be preferable.

1.4 Procedures vs Macros

It has been my observation from the start that macro’s are not sensible assembly constructs, due to the potential issues with more than one parameter. Thus, I declined to use them for all but the simplest tasks, choosing instead to write full procedures. This also helped significantly reduce the compiled code size.

In hindsight it would have been better to use the stack for parameter passing, from a design point of view, since it avoids a lot of headaches with regards to swapping registers around and the like. However, the performance advantage of passing parameters in registers is clear, and thankfully for our purposes there was not a single occasion in which this method was entirely impracticle.

1.5 Strings

Due to my existing C/C++/ObjC experience I found it natural to consider strings as null terminated. Clearly this was not an entirely sensible idea, given that the majority of string operations where with regards to displaying to screen. This required some overhead in copying strings and replacing null terminators with ‘$’. This overhead could be reduced significantly, but not entirely, given more time for development.

But the conscious choice to standardise the terminating character made it much easier for me to manage data in the program, of all sorts, and ensured that generic functions like stringCopy and displayString always worked as expected.

2.0 Pseudo-code

2.1 Data structures

```
STRUCTURE Employee
        [string] full name (30 + 1)
        [string] age (3 + 1)
        [string] employee number (10 + 1)
        [string] telephone (17 + 1)
        [string] state (20 + 1)
        [number] salary
END
```

2.2 Main

```
PROCEDURE Main
        INPUT maximum number of records to allow for
        CHECK desired number is practical
        ALLOCATE employeeList
        ZERO employeeList
        INPUT database file name
        OPEN database
        READ records into employeeList
        CLOSE database

        DO:
                DISPLAY main menu
                INPUT selection

                IF selection IS NOT 7 THEN
                        IF selection IS 1 THEN
                                DISPLAY records
                        ELSE IF selection IS 2 THEN
                                SORT records BY name
                        ELSE IF selection IS 3 THEN
                                SORT records BY age
                        ELSE IF selection IS 4 THEN
                                SORT records BY employee number
                        ELSE IF selection IS 5 THEN
                                FIND employee BY employee number
                        ELSE IF selection IS 6 THEN
                                incrementSalary
                        ELSE
                                DISPLAY invalid selection error
                        END
                END
        WHILE selection IS NOT quit

        INPUT database file name
        OPEN database
        WRITE records from employeeList
        CLOSE database

        EXIT
END
```

2.3 Control procedures

```
FUNCTION increaseSalary
        DISPLAY request for employee number
        INPUT employee number
        FIND record FOR employee number

        IF found THEN
                DISPLAY request for increase percentage
```

```
            INPUT percentage
            INCREASE salary BY percentage
        ELSE
            DISPLAY record not found
        END
END

FUNCTION findEmployee
        DISPLAY request for employee number
        INPUT employee number
        FIND record FOR employee number

        IF found THEN
            DISPLAY record
        ELSE
            DISPLAY record not found
        END
END

FUNCTION displayList
        FOR EACH record IN list
            DISPLAY record
        END
END

FUNCTION displayMenu
        DISPLAY menu
        INPUT choice
        RETURN choice
END
```

2.4 Utility procedures

```
FUNCTION saveString
        FOR EACH record IN list
            WRITE record TO file

            IF file error THEN
                RETURN fail
            END
        END
END

FUNCTION loadString
        WHILE NOT list is full
            READ record FROM file

            IF error THEN
                RETURN fail
            ELSE
                IF record IS NOT null THEN
                    ADD record TO list
                ELSE
                    RETURN success
                END
            END
        END

        IF list is full AND file is not finished THEN
            RETURN fail
        ELSE
            RETURN success
        END
END

FUNCTION displayRecord
        DISPLAY name
        DISPLAY age
```

```
        DISPLAY employee number
        DISPLAY telephone
        DISPLAY state
        CONVERT salary TO string
        DISPLAY salary as string
END

FUNCTION reverseString
        SET a TO start of string
        SET b TO end of string

        WHILE a IS BEFORE b
                SWAP a AND b
        END
END

FUNCTION doubleWordToString
        LOAD value INTO fpu

        WHILE value in fpu IS NOT null
                COMPUTE remainder OF value in fpu DIVIDED BY 10
                CONVERT remainder TO ascii number
                APPEND ascii number TO result
                DIVIDE value in fpu BY 10
        END

        reverseString result
        RETURN result
END

FUNCTION stringToDoubleWord
        LOAD null INTO fpu

        WHILE NOT end of string
                MULTIPLY value in fpu BY 10
                ADD value OF current digit
        END

        RETURN value in fpu
END

FUNCTION stringToWord
        SET value TO null

        WHILE NOT end of string
                MULTIPLY value BY 10
                ADD value OF current digit
        END

        RETURN value
END

FUNCTION getLine
        SET buffer size character TO size of buffer
        call dos input

        IF error THEN
                RETURN fail
        ELSE
                RETURN result
        END
END

FUNCTION stringReplace
        WHILE NOT end of string
                IF current character EQUALS search character THEN
                        REPLACE current character WITH replacement character
                END
        END
```

```
END

FUNCTION findMinimum
        SET offset TO null

        WHILE NOT end of string
                IF current character EQUALS search character OR termination character THEN
                        RETURN  offset
                ELSE
                        INCREMENT offset
                END
        END

        RETURN fail
END

FUNCTION stringLength
        SET offset TO null

        WHILE NOT end of string
                INCREMENT offset
        END

        RETURN offset
END

FUNCTION stringSwap
        WHILE count IS NOT null
                READ byte FROM string one
                READ byte FROM string two
                WRITE byte from string one TO string two
                WRITE byte from string two TO string one
                DECREMENT count
        END
END

FUNCTION displayMessage
        COPY message INTO temporary buffer
        REPLACE null IN message in temporary buffer WITH dollar sign
        call dos display
END

FUNCTION memSet
        WHILE count IS NOT null
                WRITE value TO current position in string
                INCREMENT current position in string
                DECREMENT count
        END
END

FUNCTION searchForNumber
        searchForRecord WITH offset of number field within record structure
END

FUNCTION searchForRecord
        WHILE NOT end of list
                IF search value EQUALS search field of current record THEN
                        RETURN current record
                ELSE
                        NEXT record
                END
        END

        RETURN not found
END

FUNCTION sortList
        REPEAT record count - 1 TIMES:
                SET current record TO start of list
```

```
            REPEAT current above number of times - 1 TIMES
                IF sort field of current record IS GREATER THAN sort field of next record THEN
                    SWAP current record AND next record
                ELSE
                    NEXT record
                END
            END
        END
END

FUNCTION stringCompare
        REPEAT:
            IF NOT current character of string one  EQUALS current character of string two THEN
                IF current character of string one IS LESS THAN current character of string two THEN
                    RETURN less than
                ELSE
                    RETURN more than
                END
            ELSE
                IF current character of either string IS null THEN
                    RETURN same
                ELSE
                    NEXT character of string one
                    NEXT character of string two
                END
            END
        END
END

FUNCTION stringCopy
        call stringCopyWithLength IGNORING returned length
END

FUNCTION stringCopyWithLength
        SET length TO null

        WHILE NOT end of string AND NOT current character of source EQUALS null
            COPY current character of source TO current character of destination
            INCREMENT length
        END

        RETURN length
END

FUNCTION malloc
        IF requested memory size IS GREATER THAN available memory size THEN
            RETURN fail
        ELSE
            STORE current memory position
            INCREASE  memory position BY requested memory size
            RETURN stored memory position
        END
END

FUNCTION free
        IF address IS valid THEN
            SET memory position TO address
        END
END

FUNCTION saveRecord
        WRITE name TO file
        WRITE age TO file
        WRITE employee number TO file
        WRITE telephone TO file
        WRITE state TO file
        CONVERT salary TO string
        WRITE salary as string TO file
```

```
END

FUNCTION readRecord
        READ INTO buffer FROM file

        IF amount read IS null THEN
                RETURN null AND no error
        ELSE
                ALLOCATE new record
                COPY name FROM buffer INTO new record

                IF buffer is empty THEN
                        RETURN error
                ELSE
                        COPY age FROM buffer INTO new record

                        IF buffer is empty THEN
                                RETURN error
                        ELSE
                                COPY employee number FROM buffer INTO new record

                                IF buffer is empty THEN
                                        RETURN error
                                ELSE
                                        COPY telephone FROM buffer INTO new record

                                        IF buffer is empty THEN
                                                RETURN error
                                        ELSE
                                                COPY state FROM buffer INTO new record

                                                IF buffer is empty THEN
                                                        RETURN error
                                                ELSE
                                                        CONVERT salary as string TO salary as number
                                                        COPY salary as number INTO new record
                                                END
                                        END
                                END
                        END
                END
        END

        MOVE position in file BACK BY number of bytes left in buffer

        RETURN new record
END

FUNCTION min
        IF value one IS GREATER THAN value two THEN
                RETURN value two
        ELSE
                RETURN value one
        END
END
```

3.0 Conclusion

3.1 What Went Right

My decision to make everything as general as possible worked wonderfully. There is only one sort function, which can sort any string field of the record. Likewise for the search function, although it is only utilised for one field in particular. Choosing null as the terminating character for strings was perhaps a mistake, given that the majority of uses required a dollar sign, but nonetheless the unity in termination type allowed for reliable, general-purpose string and memory functions.

The choice of using an array of record pointers, rather than the records themselves, resulted in sort times that were much less than 55ms (the smallest measurable unit of time), even given the use of bubble sort, and even with as many as 300 records.

Surprisingly, writing most of the code in a “sterile” environment (i.e. not on a PC) enforced good practices and reliable coding, and nearly all the problems I encountered were design issues. Certainly, I had not a single significant error due to a mistyped operand or improper opcode, or whatever else. Clearly when you’re writing with only your head to check your code, you do so much more thoroughly.

Use of the FPU was strangely easy. Although I dislike its limited stack-based design, for our very simple usage it was sufficient. Perhaps I was just lucky in getting all my FPU code to work first time, but given that little else I wrote worked first time, it seems a clear abnormality.

3.2 What Went Wrong

There are a lot of things I’m not fully happy about, although nothing really significant, which I suppose is something of a blessing. I would rather have stored the numerical values as actual numerical values, rather than as strings, because it saves a significant amount of space. It would also make searching and sorting those fields much faster. However, as mentioned in earlier sections, to do so would have required more time than was available, and would also reduce the reusability of a lot of the record-handling functions, like those for search and sort.

File handling [now] seems to be quite strong, but isn’t tested as much as I’d like, and was up until the very last minute prone to breaking every time I changed the sample file around a bit. There was some quite exotic errors, such as when two records happened to be exactly 88 bytes together, and there was no return on the last record... this had the result of ignoring the last record. It, like many other such bugs, seem to be fixed now, but as I’ve said, I wouldn’t trust it. Buffer management is a lot harder in assembly than higher languages.

The other significant problem was simply time. Due to other assignments - most notably AI - being due in at around the same time, I wasn’t able to even start the MAL assignment until just a week before it was due. This resulted in a couple of late nights in the labs, which were hardly entertaining.

Even in hindsight, there was little that could be done about this. Assignment load for the last two weeks has been excessively high, and given that we were only given the assignment shortly before then, there seems to be no good solution. My hope for future years is that the assignment be handed out earlier. Most people will still leave it too late, but at least those like me who do try to get in early will have a much easier time, and be able to produce a much better program as a result.

There were a few other minor issues, most of which have been mentioned in passing in other sections. Most of these were simply due to inexperience, and thus, as always, next time will be much easier.

3.3 Comments

Having programmed to a degree in PPC assembly before, in addition to taking concurrently the subject ELE22MIC, I can say on reasonable grounds that x86 assembly is a real pain in the proverbial. I hate to consider what it would be like programming a modern x86, like the Pentium 3 or 4, given the hundred-fold increase in idiosyncrasies over the already challenging 8086.

While it was a novel experience, I think it has served mainly to enforce a respect for high level languages, and to prove true the common notion that x86 assembly is diabolical.

My only others thought is that the marking scheme seems pretty scary if you can’t get things to work properly. I found that the vast majority of the effort went in to just getting most things working a bit, let alone properly reading in the

file and so forth. It seems that the marking scheme should allow for people who simply can't get things to work right - like reading the records in properly - but whom have still written all the other code, i.e. for searching, sorting, etc.