COSC 460

Honours Project Report

Department of Computer Science

University of Canterbury

MacCOSC1

An Assembly Language Trainer on the Macintosh

Supervisor : Dr. M. A. Maclean

K. F. Chong

October 1987

Christchurch

--- Contents ---

Acknowledgements

I would like to thank the following people for their help in my project and the preparation of this report :

To **Dr. M. A. Maclean**, my project supervisor, for his valuable advice about the project.

To **James Collier**, whose experience and knowledge in Macintosh programming has been a great help to me.

To **Chee Ping** and **Ju Li**, for typing the greatest part of this report.

And to **Sara** for her continuous encouragement and support especially during times of frustation.

Section 1
Introduction

The COSC-1 Assembly Language Simulator running on the Micro Network was developed as a tool for teaching the basics of computer structure and assembly language programming. COSC-1 is a simple computer of complexity about equivalent to a small personal computer. There is no actual hardware that behaves like a COSC-1 computer, instead its behavior is simulated on another computer. While this is somewhat unreal, it has the advantage that the Micro Network provides quite a sophisticated programming environment to make life easier for the users. Another advantage is that some of the awkward features of real machines have been left out so that simple programs are more straightforward to write and understand. The network also provides file storage.

The COSC-1 Simulator allows assembly-language programs to be entered, assembled and debugged interactively on the IBM Personal Computer workstations used with the Micro Network. There is an editor for preparing COSC-1 programs and facilities for translating these into machine language and running them. In addition, it is possible to start and stop a COSC-1 program and peep into a COSC-1 machine to study what happens at each step while a program is running.

This project was proposed by Dr. M. A. Maclean, the original writer of the COSC-1 Assembly Language Simulator; the idea is to implement a similar trainer on the Macintosh. The reason being that the Macintosh user interface is seen to have great potential in making the COSC-1 Assembly Language Trainer a better teaching tool, as well as making it easier for the student to use and understand assembly language programming better.

Very few people after experiencing the Macintosh are not impressed by its revolutionary user interface. Instead of memorising a command language and typing it on the keyboard, the Macintosh uses a mouse as a pointing device with "icons" and pull-down menus. Pointing and choosing directly on the screen is a far more natural way of giving commands. The program also communicates with the user via dialog boxes or alert boxes, avoiding inter-mixing with the information displayed in different windows, and thus minimising confusion and chaos on the screen. Multiple overlapping windows provide isolation of different information into separate windows and yet allow each to be accessed on the same screen. This is an excellent environment for a teaching tool like the COSC-1 trainer. It is believed that the Macintosh will certainly make using the COSC-1 trainer easier than the present system.

The C programming language was chosen to develop the trainer because, firstly, it has all the data structures that are likely to be used by the COSC-1 trainer, and secondly the existing COSC-1 Simulator was written in C as well, which means that part of the program for the assembler and the interpreter can be used with only minor modifications. The version of the C language used is Lightspeed C, a special version that provides an interface to the Macintosh Toolbox routines, and provides an integrated program development environment with facilities like editing, compiling, linking and program execution. It also provides a special facility that allows the program to be a stand alone Macintosh application. And lastly it is chosen because of its speed : it is a fast compiler.

The COSC-1 Assembly Language Trainer on the Macintosh is named MacCOSC1 to reflect its root, the COSC-1 machine, and the host that it resides on, the Apple Macintosh. It is designed to resemble closely the existing Simulator so that the user will feel familiar enough, at the same time it is "Macintosh-ish" enough to make regular users of the Macintosh feel at home.

Because of the vast difference between the programming environment of the IBM PC and the Macintosh, the only part of the existing Simulator that can be transported to MacCOSC1 is the assembler and part of the debugger. The essential functioning of the assembler will remain even though the data structures used will be different. Communication with the user and displaying of information will be vastly different. A lot of work was expected in putting up the user interface, the editor and the debugger.

Section 2
Aims and Objectives

2.0

The aim of the project is : " to implement the COSC-1 Assembly Language Trainer on the Macintosh ". Like the existing Simulator, the COSC-1 Assembly Language Trainer, MacCOSC1, will be an integrated system comprising a screen editor, an assembler and an interactive debugger. In developing MacCOSC1 the aims were to make it easier to use and to help the user to understand assembly-language programming better.

2.1 Initial Goals

At the beginning of the project, the following development plan was proposed. It consists of five stages, together they contribute to the complete aim of the project.

Stage 1: developing the user interface
The first step in tackling the problem is to design the user interface and to implement it. This requires knowledge of how MacCOSC1 will be used, and how should it function, the relation between the different components of MacCOSC1, the editor, the assembler and the debugger, and how they interact. Design of the menus and screen layout must conform to the Macintosh User Interface Standard : point-and-select on the pull-down menus, windows for displaying information at different times as well as dialog and alert boxes for communication with the user. Once the user interface is implemented, the remaining components can be "added" to it.

Stage 2: developing the editor

A screen editor that suits the Macintosh environment is to be developed. It must have the "normal" features of a Macintosh text editor like cut-and-paste. Advanced features like pattern searching and replacement, and setting tab positions, are also desirable. The editor should not only allow programs to be entered and modified, but must also be able to keep the program for use by the assembler. Storing a copy of the program on disk should be catered for. The editor window must also have scroll bars and support all the normal window operations.

Stage 3: installing the assembler and the interpreter

The assembler must be able to take the program prepared using the editor, assemble it into object codes and store them into the simulated memory. It must have an error recovery mechanism during assembly, the ability to locate the source of the error and pass it on to the editor. A successfully assembled program will be able to be executed by the COSC-1 computer to produce the desired result. The execution of COSC-1 programs is simulated by the application program, this will need to be installed in MacCOSC1 as well. There should be facilities for displaying the object codes of the program for the purpose of studying or debugging.

Stage4: developing the debugger

When errors occur during execution of the program, the user would like to fix up the program. The debugger should provide facilities to pin-point the problem and test any changes made. Helpful information like the states of the machine registers, the program counter and the flag word should be easily accessible to the user. There must be means by which the user can pause the execution at different locations to study the program behaviour. These are the general requirements of an interactive debugger which it is hoped to include in MacCOSC1.

Stage 5: finishing touches
If all goes well, by this stage all the loose ends should be tied up and thorough testing will be performed to verify the performance and usability of MacCOSC1.

2.2 This project's goals

Some time after the development work had started, a review of the aims and objectives led to a more realistic expectation of the project given the time limit. Instead of aiming for a fully operational MacCOSC1 satisfying all the initial planning, work was narrowed down to Stages 1 to 3. This decision was made based on the time constraints and the demands of acquiring technical knowledge of the Toolbox routines and the special behaviour of the Macintosh. The interactive debugger has been left as a future extension. This narrowing of the aims and objectives does not interrupt with the aims of producing a high quality application program. However, the design does not deny room for improvement and extensions to MacCOSC1.

Section 3
Design and Implementation

3.0

With the aims and objectives clearly defined, the next step in the development is design and implementaion. Through a period of less than six months, time was spent in designing, programming, testing and debugging to bring the MacCOSC1 into shape. However, with other course work going on in parallel during this period of time; work was carried out in fragments of time found in between other work. Due to the limitation and constraints on time, not all of the desired features and components of MacCOSC1 were implemented, they are left as extensions in the future.

3.1 Overall Design

The first step in the development of MacCOSC1 is an overall design that gives a conceptual model of the desired product. The environment that the Assembly Language Trainer is going to live in is the Macintosh. This has a rather special kind of user interface that consists of a pointing device, called a mouse; pull-down menus which contain various commands and pictorial "icons" that represent different kinds of objects that are recognisable to the computer. Its screen layout is typified by the presence of multiple overlapping windows, which can be moved around and whose sizes can be changed at the wish of the user. Clearly MacCOSC1 must support all these features so that it is acceptable to the users.

MacCOSC1 is designed to be an integrated system having different components or utilities which include a screen editor, an assembler and interactive debugger. There must be a means whereby the user can switch from one utility to another conveniently and with the least possible risk of confusing the user as to which utility he or she is using at any particular time. The pull-down menus offers an answer to the first issue, with them the user can instruct MacCOSC1 to switch from one utility to another. For each utility, there exists a set of commands to be used, these can be put in a group within the menu so that the user can locate them easily. Multiple overlapping windows, seemingly confusing, can in fact minimise the effect of confusion to the user. Each window has a title bar to differentiate it from others, and relevant information is confined to the same window instead of inter-mixing with other unrelated data. This also allows specific functions to be provided to a particular window and not to another to avoid possible illegal operations on data or information in other windows.

Three utilities are needed for this integrated system. One of them is a screen editor that allows the user to enter and modify assembly-language programs. This editor must support the standard Macintosh "cut-and-paste" features, pattern searching facilities should also be included as advanced features of the editor. The editor will have its own kind of window that behaves like a Macintosh word processor but is much less powerful. A second utility is the assembler, this is the heart of MacCOSC1. It must be able to take from the editor the program prepared by the user, assemble it into object code that can be executed by the COSC-1 machine. The assembled object codes must be able to be displayed in a separate window for the purpose of debugging. And finally, to help the programmer in debugging his or her program, an interactive debugger is needed to allow the user to observe the status of the machine during execution, make changes to the object codes directly to detect the different behaviour in execution and to stop execution at various place to inspect the state of the machine.

Coordination must be maintained at all time so that the right kind of window shows up for the corresponding utility in use, and menu items are enabled only if they are applicable to the utility in use. The Macintosh is an event-driven system, i.e. every single action causes an event to be delivered to the application program. By processing these events carefully and correctly, the result is a user-friendly and intelligent application program.

3.2 User Interface Design

A successful application program must be able to communicate with the user in a simple and effective way. The Macintosh user interface provides just that : its pull-down menus free the user from memorising a set of commands, the pictorial "icons" reflect visually what objects are there to be accessed and the windows provide isolation of relevant information to individual windows and yet allow them to appear on the screen together so that the user can access their content easily.

The menus

A crucial part of the user interface design for MacCOSC1 is the set of menus because they are one of the main ways for the user to communicate with the application program. After much careful thought, a "standard" menu bar was chosen, and items in each menu were finalised. Figure 3.2.1 shows the "standard" menu bar used by MacCOSC1.

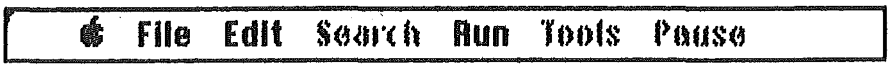


Figure 3.2.1 MacCOSC1's menu bar

This menu bar is "standard" because it is the only one that is used by MacCOSC1, it contains all the necessary menu items to allow the user to communicate with MacCOSC1. The decision to have only one standard menu bar was based on consistency and the "least surprise" requirement : the user can be sure from the beginning to the end of the session with MacCOSC1 that at any time he or she selects a menu item from the menu, the same effect will result as it has before; and the same item will always be at the same place where the user found it before.

Under each menu title is a collection of menu item, each representing a command. These collections of menu items are grouped on two criteria : (i) to put all the relevant and closely related items together, and (ii) to keep the menu items applicable to each utility under separate collections. However, the command sets for the utilities actually overlap to some extent and preference is given to criterion (i) when conflicts arise during the final settlement of which item is to go into which collection. All these menu items will be present at all times, but not all of them are applicable to the utility in use at a particular time. To avoid those menu items that aren't applicable from being selected, they are "disabled" so that attempts to select them will have no effect.

The first menu title (starting from the left) on the menu bar is the **Apple** menu, symbolised by an "apple" mark. Under this menu are desk accessories supported by the Macintosh operating system. These items are not part of MacCOSC1, they are included from the operating system. The only item under this menu that is directly supported by MacCOSC1 is the **About MacCOSC1...** , which is a display of a very brief description of MacCOSC1. The second one is the **File** menu. In this menu are the filing operations, viz : **New**, **Open**, **Close**, **Save**, **Save as ...** , **Page Setup**, **Print** and **Quit.** The third is the **Edit** menu with the standard editing operations : **Undo**, **Cut**, **Copy**, **Paste** and **Clear.** At present, the **Undo** command has not been implemented and it is disabled permanantly to reflect this. These three menus are organised according in the standard Macintosh way and they fit well into the two criteria set above, in the **File** menu is a collection of all relevant commands operating on files except **Quit** which finds it place here purely because of convention : many Macintosh applications place it in the **File** menu. Under the **Edit** menu is a set of all the editing functions.

The **Search** menu contains the advanced features for editing. Items include **Find again**, **Replace**, **Everywhere** and **What to find ....** This is a group of relevant commands that operate on the editor only. Unfortunately, because of time constraints, these features are not implemented at present but their intended use is described in Section 6.3.

The **Run** menu has three groups of items which are nevertheless related commands. **Reset** will clear up the COSC-1 machine memory so that the next execution will start at a standard state : all registers' contents cleared, program counter set to memory address 1, flag word cleared and memory contents cleared. **Assemble**, **Go**, **Go-Go**, **Step** and **Step-Step** are commands for assembling and execution of a COSC-1 assembly-language program. **Go** is a fast run mode, **Go-Go** will stop execution at break points and continue only if instructed. **Step** causes execution of one instruction at a time, it will continue only if instructed whereas **Step-Step** performs executions with a short pause between instructions and it will stop at break points. This variety of execution modes are designed to be part of the debugging facilities. The last item in this collection is **Preferences**. At present the only item that has been implemented is **Assemble**. Further description on these commands is found in Section 6.3.

The **Tools** menu houses more debugging tools : **Observe** gives a display of the machine register's contents during execution, **Trace** displays the value of an expression (specified by user) during execution and **Breakpt** allows programmer to set break points. **Binary**, **Decimal**, **Hexadec** and **Character** are memory content display modes. The **Pause** menu has only one item : **Halt** which is an interrupt button to stop execution of the program. Both the **Tools** and **Pause** menus are not implemented at this stage. Their intended use is described in Section 6.3.

The Macintosh Toolbox routines provide two alternatives to build the pull-down menus, windows, and dialog boxes. One way is to do it explicitly in the program code. This will involve first calling a Toolbox routine to create the object giving details of the different characteristcs of the object. The other alternative is to use "resources". A program's resources can include all the odds and ends it needs to do its job : they can be usedto specify menus, windows, icons, cursors and character fonts. Each of these has its own resource type which can be stored on a resource file to be used by the application program. They are like templates, when needed the application program just summons them from the resource file, and "tailors" them to suit the different situations. They can be used again and again. One of the advantages of using resources is that they can be packaged outside the program where they can be accessed and modified. To create the resources, the software tool **ResEdit** can be used. This is a very powerful (but somewhat flaky) software utility. A lot of time was spent in learning how to use it because of the lack of documentation and a reference manual. In fact , for the most part of it was learnt by trial-and-error ! **ResEdit** is actually a collection of resource editors, one for each type of resource.

For MacCOSC1, seven different menus were created as resources, they are appended to the menu bar in the application program to build the complete menu bar displayed on the screen.

The windows

The other part of the user interface design is the displaying of information or data on the screen. A Macintosh application program typically does this through the windows. There are different kinds of windows ; some allow the user to edit the content, other are there to give advice or warning (the Alert box) and expect response from the user, yet others are there to collect more information from the user before certain further action can be carried out. MacCOSC1 makes full use of these features provided by the Toolbox.

The design of the windows is very much dependent on their usage. Alert boxes are used for warning and error messages, whereas document windows are for displaying text, and usually the content is user-modifiable. The initial positions of the windows need to be considered beforehand so that they look appropriate on the screen. The final solution thought of is to place all alert boxes and dialog boxes in the "middle" of the screen to capture the user's attention since they normally require immediate attention. Document windows are movable according to the taste of the user, their initial positions are arranged in such a way that they occupy different portions of the screen, with possible partial overlapping so that the user can switch from window to window more easily. Figure 3.2.2 and Figure 3.2.3 show the layout of the screen with the different windows in their initial positions.

The editor window behaves very much like a Macintosh word processor's window. Assembly-language programs can be entered and modified inside the window, they are treated purely as text. All the editing operations, namely **Cut**, **Copy**, **Paste** and **Clear** are applicable to its content. A vertical scroll bar is provided so that a "long" program can be scrolled in and out of the window to read the different parts of it. The "go away" is used to dismiss its appearance on the screen (same as the **Close** command, see Section 6.3) and the "size box" is for altering the window size. More design issues about the editor window are defered to Section 3.3.

The object codes window behaves much the same but the intention is to make it more restricted so that only the area of the window where instruction codes or data are displayed is user modifiable. The other contents of the window are static. This is to avoid confusion between the editor window and the object codes window and also to reflect the function of the window. The object codes window is to be used in conjunction with the debugger, where the only thing that can be changed on the object codes window is the content of the memory word and

What other contents?

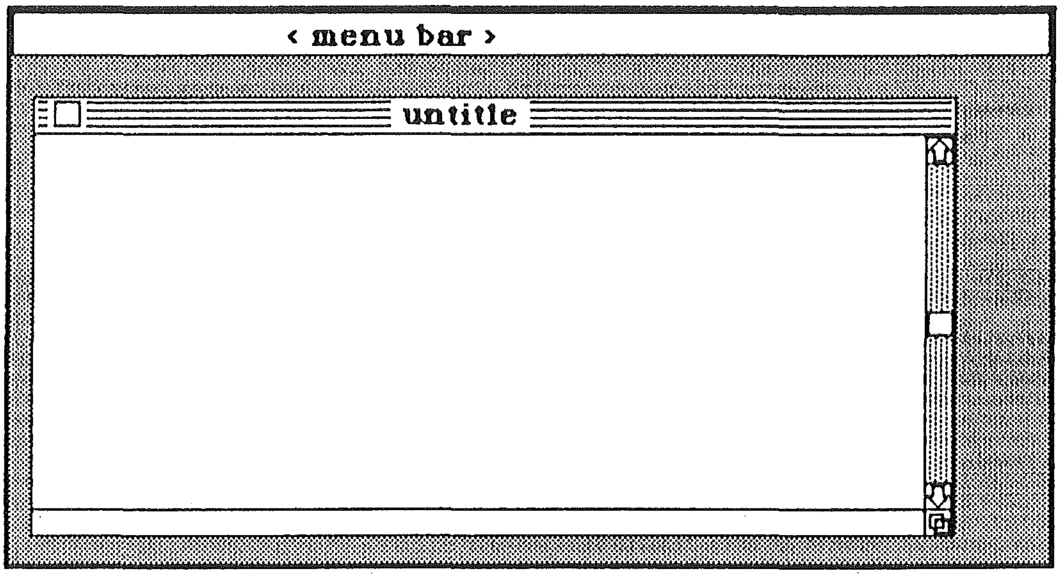


Figure 3.2.2 Proposed editor window

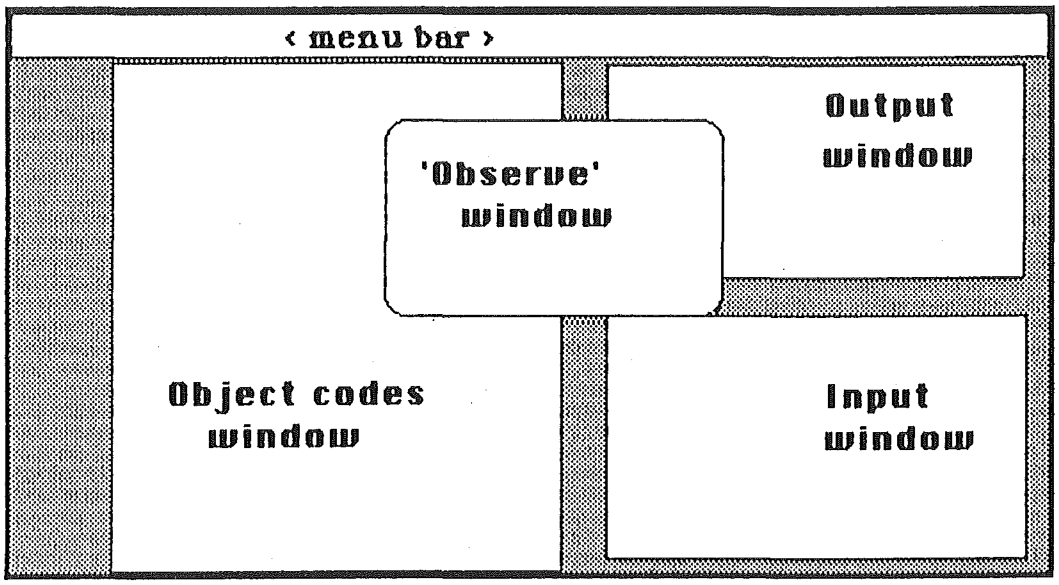


Figure 3.2.3 Proposed screen layout Object codes window, Input & Output window & 'Observe' window

nothing else. It would not be meaningful for the user to change the memory address. However, at present this restriction has not been imposed, instead the whole content of the object codes window is made static.

The different utilities provided by MacCOSC1 are used at different states of the session, for example using the editor will mean that it is in the "editing state" whereas at execution and debugging time MacCOSC1 is in "execution-debugging state". There need to be ways to show the distinction between the two states to help the user in using MacCOSC1. One of the ways MacCOSC1 shows the different states is by its choice of the active window. The editor window is associated with editing and the object codes window is associated with execution and debugging. The window that is active at any moment indicates which state MacCOSC1 is in at that moment.

Another indicator for the different states is the menus. Items in the menus are enabled only if they are applicable to the utility that is in use at that moment and this shows which state MacCOSC1 is in. For example, the items in the **Tools** menu will not be enabled unless MacCOSC1 is in "execution-debugging state". Likewise, if the only items that are enabled in the **Run** menu are **Assemble** and **Preferences**, it is indicating that the program is still in "editing state" since it has not been assembled yet and thus can not be executed, which explains why the executing commands are not enabled. The **Pause** menu is only enabled when the program is being executed; it is not meaningful to allow the user to stop execution when nothing is being executed. The menus are updated at each change of state.

The **Input** and **Output** windows will appear during execution of a program when input data is required or there is output from the program. The contents of these windows remain static at most time except when input is required during which time data can be entered in the **Input** window. This will be signaled by a blinking cursor appearing in the **Input** window, which is now the active window with its title bar highlighted. Output will be displayed on the **Ouput** window, its contents is not editable. The decision to have two separated windows for input and output was based on one factor : very frequently the user will want the output of the program to be un-interrupted by other data even though the program reads in input at intervals in between output. A separate window for input and ouput answers this need. In cases where the user does want the input data to be interleaved with the output, the same effect can be achieved by making the COSC-1 program echo the input data as it reads them in. It is much harder (if possible) for a single input/output window to handle the first case. The **Input** window is for input from the keyboard, should an input data file is used, MacCOSC1 will read input from the file directly. Both the **Input** and **Output** window have vertical scroll bars so that in case the input or output data is more than what can fit into the window, the scroll bar will allow the user to view the data in parts. Standard window operations on window apply to them : dragging, re-sizing and closing the window.

The **Observe** window is for displaying information about the state of the COSC-1 machine during execution : the registers' contents, the program counter and the flag. These are useful information to the user in debugging. They allow the user to inspect the behaviour of the machine during execution and often give important clues to errors. To update this window continually is likely to slow execution considerably. Another way of doing this is to update the window only when there is a pause, but this will mislead the user as to what is happening inside the machine during execution. In line with the goal of helping the user to understand the COSC-1 machine, the first alternative should be chosen. Another thing that will be displayed on this window is the contents of any memory word specified by the user, in any of the modes he or she chooses from the **Tools** menu. This is just to help the user in converting the information from one notation to another since the object codes are displayed in hexadecimal form by default.

The **Input**, **Output** and **Observe** windows are associated with execution of the program and are not implemented at this stage.

Windows used in MacCOSC1 are made from templates set up in the resource file. Specifications are made for each type of window using **ResEdit** and stored in the resource file and later summoned by the program.

The program is actually sitting in an infinite loop waiting for events to occur. When events occurred, they are processed one after another in sequence. This means that only if the user gives an instruction, i.e. causing an event to be reported, will the program react. In order that the user can give instructions with precise meaning, the user interface must be explicit and clear enough and should be designed to be fool-proof. For these reasons the user interface of MacCOSC1 is treated with such importance.

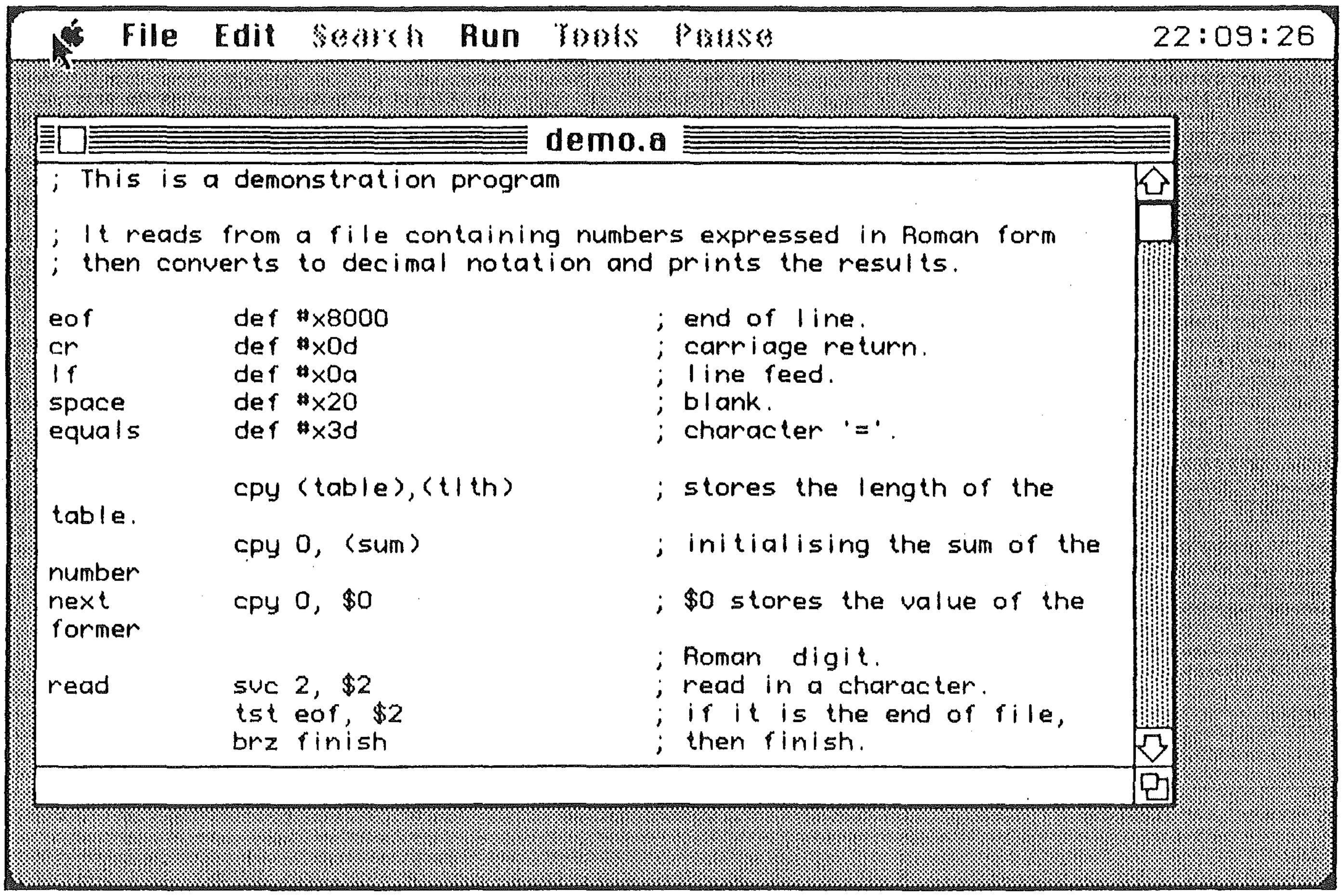
3.3 The editor

From the beginning MacCOSC1 is designed to be an integrated system, and therefore having a program editor of its own is crucial. The user can then prepare an assembly-language program, assemble it and execute it all within one environment and there is freedom of switching from one utility to another. Further, it makes MacCOSC1 more powerful in terms of aiding the user to locate errors during assembling of the program. If the assembler discovers any syntax error, the editor is activated and the faulty source code is pin-pointed as closely as possible in the source program.

The editor is to behave like a text editor or a word processor but minus the fancy features that beautify the contents. This is done for the sake of consistency in that users who are familiar with other Macintosh applications which have text editors will feel at home with the editor. It produces a pure text document which can be saved as a text file on the storage medium (usually disk). It will also allow the user to make modification to the program, like inserting some more text, removing erroneous source code or moving block of statements to a different position. These editing functions will be carried out in a "Macintosh-ish"way. It must also allows the user to scroll portion of the text into and out of the window so that a "long" program can be accessed in parts.

To implement the editor, two structures are needed, one for displaying on the screen and the other to store the text internally in the memory. Putting text on the screen is best done with a document window whereas the internal structure for storage of the text is provided by the Macintosh Toolbox structure type called **TERecord**, which stands for Text Edit Record, a structure that not only stores the text content of a window, but also all the necessary information needed to format the content onto the window. Figure 3.3.1 shows what the MacCOSC1's editor window looks like on the screen.

A few points concerning the characteristics of the editor window need special consideration. One of them is whether to have one scroll bar (the vertical one) or two (one for scrolling vertically and one horizontally). It is obvious that for any meaningful program written in COSC-1 (or any) assembly language, it will be of a length that can't possibly fit into one window even if it is extended to the full size of the Macintosh screen. Hence having a vertical scroll bar is compulsory. As far as the horizontal scroll bar is concerned, it leads to close investigation of another feature of the window, that is whether a source code line extending beyond the width of the window should be wrapped around or not. Allowing the text to wrap around would mean that a horizontal scroll bar is not necessary : there isn't any text beyond the left and right sides of the window. The disadvantage of this is that the wrapped around text causes confusion to the user about the definition of a source line; which is recognised by the assembler as a sequence of characters terminated by a end-of-line character (carriage return). For example in Figure 3.3.1, there are two source lines with comments wrapped around to the next line, this can easily lead the user to the (wrong) conclusion that typing a line starting at the first position is a continuation to the previous line, or that a comment line (on its own) does not need a ";" at



```
; This is a demonstration program

; It reads from a file containing numbers expressed in Roman form
; then converts to decimal notation and prints the results.

eof         def #x8000                  ; end of line.
cr          def #x0d                    ; carriage return.
lf          def #x0a                    ; line feed.
space       def #x20                    ; blank.
equals      def #x3d                    ; character '='.

            cpy (table),(tlth)          ; stores the length of the
table.
            cpy 0, (sum)                ; initialising the sum of the
number
next        cpy 0, $0                   ; $0 stores the value of the
former
                                        ; Roman  digit.
read        svc 2, $2                   ; read in a character.
            tst eof, $2                 ; if it is the end of file,
            brz finish                  ; then finish.
```

Figure 3.3.1 How the editor window looks on the screen

the beginning of the line. This is, of course, contrary to the aim of helping the user to understand assembly-language programming. A better solution is not to wrap around the text, but to provide instead a horizontal scroll bar so that the user can still look at the part of the line beyond the right side of the window. This will also simplify the process of getting source code for assembling because then the array **lineStarts** in **TERecord** can be used to locate the start of each line easily. At this moment, MacCOSC1 has not implemented the horizontal scroll bar and the text has to be wrapped around to allow the user to read the whole line.

One other thing that will make using MacCOSC1 easier is to allow the user to set tab positions. A program will look tidy and neat if the **labels**, **operators** and starts of comments can be lined up in vertical columns. It will make the program easier to read too. For this reason the provision of tabs is highly desirable. One way of doing this is to provide fixed tab positions across the window width so that every time the *tab* key is pressed, the cursor will advance to the next nearest tab position. Another way is to allow for variable tabs, with the user being allowed to set tab at positions of his or her choice. Given the fact that every user has his or her own programming style, variable tabs sound most desirable. This would mean that the program must be able to detect the cursor positions when tabs are set. Since the Toolbox manintains the cursor position at all times, this will not be a big fuss to implement. Unfortunately, due to limitation of time, this feature has had to be left out from this version of MacCOSC1. It is hoped that in a future implementation this will be included. Instead, the only tab facility provided at present is a simple, fixed size tab which moves the cursor across the window by a fixed (5 characters) width.

Standard Macintosh operations on the window like dragging, resizing and closing the window and standard editing functions like **cut**, **copy** and **paste** are all supported by MacCOSC1. The user will be asked whether he would want the contents to be saved (if edited since last saved) when closing a window.

Another feature that MacCOSC1 can have is to allow multiple source files to exist at the same time. This of course would mean that the editor would have to support multiple windows. Since the editor allows cut-and-paste between windows, this will mean that the user can merge several small programs into a large one. For example, the user can write different parts of a program separately, each forming a small program and testing can be performed independently, then this parts can be merged together using cut-and-paste into one program. This will help the user in writing a "big" program.

Text or program that has been prepared in the editor window can be stored onto disk, when the user requests it. The actual file operation is done via the Toolbox routines.

It was intended to implement advanced features for the editor like pattern searching and replacing. Unfortunately, due to the limitation on time, they have had to be left as future extensions. In Section 6.3 a brief description is made to indicate what was intended for pattern searching and associated functions.

3.4 The assembler

The assembler is based on the original version currently used in the First year undergraduate course in our Department. The program code was written by Dr. M. A. Maclean and was written in the programming language C. A large part of the new assembler follows very closely to this original version, except where necessary the code is modified to suit the new environment and / or the Lightspeed C language requirement. The major changes made are

(i) the assembler now actually performs "2-pass" assembly as opposed to the original "1-pass-and-backtrack-filling-in" method, and
(ii) listing options for assembled object codes are provided

A COSC-1 assembly-language program is seen as a sequence of lines by the assembler. Each of these lines is one instruction and it contains combination of predefined symbols, user-defined symbols, operators and literals known to the assembler. The assembling process is simply a loop that takes one source line at a time, checks its syntax, builds up a symbol table for the user-defined symbols and encodes the instruction into object codes. In the original version, the editor stores each line as an array of characters, and the lines are linked together as a linked list to represent the whole program. This has the advantage of easy access to each line by the assembler. For MacCOSC1, the source program is stored within the **TERecord** of the editor window as a long sequence of characters. To enable the assembler to process the source code line by line, the program reads from the **TERecord** one line at a time (each line is terminated by an end-of line-character) into an array and then passes it on to the assembler.

The general format of a COSC-1 instruction is as follows :

label operator operand1, operand2 ; comment

Certain components of the instruction may be optional depending on the nature of the instruction. The assembler takes each source line and checks it against the general format. Each token of the source line is checked to see if it is a legal symbol. User-defined symbols are entered into the symbol table so that during encoding their actual values may be found. If a line (instruction) is found to be legal, then it will be encoded into object codes and stored in the simulated memory ; which is represented internally by an array of **MWord**, a record structure that represents one memory word.

Not all of the user-defined symbols will have their values defined when they are used in the instructions. These undefined symbols, assuming the program is syntactically correct, will eventually become defined as the process goes on. When this happened, the object codes of those instructions which use the symbols need to be updated. This can be done in a number of different ways. The original version does this by keeping lists of undefined symbols, each occurrence of the symbol is linked together by pointers ; when a symbol is defined, all the instructions using that symbol can be resolved by following that particular linked list. At the end of the assembly process, all symbols will be defined and hence all the instructions coded correctly. This method results in rather complicated data structures to represent the COSC-1 machine memory words.

MacCOSC1 takes a different approach : it employs a full "2-pass" assembly. The assembly process is essentially the same as in the previous method, but requires that the source program be scanned twice. The first time it checks the syntax and builds up the symbol table. At the end of the first pass, all user defined symbols must be defined, otherwise an error has occurred. With the complete symbol table, the program does a second passg, encoding each instruction and storing it into the simulated memory straight away since all symbols are now defined. This method does not have the complication of the data structures but its speed was a concern at first. However, experiments and tests show that even with 2 passes, the speed is well within tolerance level. (See Appendix A)

When the **assemble** command is issued, MacCOSC1 will carry out the assembling immediately and the editor window will stay on the screen. If the the assembly is successful, the **Object codes** window will appear on top of the editor window and become the active window. The reason for retaining the editor window is to remind the user to save any changes made : before closing the window MacCOSC1 will ask if the user would like to save the contents if it has been edited since last saved.

Following the standard Macintosh way, error messages during assembly come out in alert boxes. When an error is detected, the assembling process will terminate immediately and control will be passed to the editor. To locate the problem , a counter is used during assembling to track the source line that is being assembled currently. Upon detection of error, the counter is

This isn't very standard.

used together with the information about starting positions of each line from the **TERecord** to position the cursor right on the erroneous line. There is one exception to this, any symbol remaining undefined at the end of the first pass causes an error but the assembler cannot tell where the defining should occur because the user can define symbols anywhere within the program.

It is most useful for debugging purpose to have the assembled object codes displayed on the screen. For this reason, a specialised window is used to differentiate the object codes display from the editor window. At present, the window is set up so that its content is not modifiable as mentioned in 3.2. There are (at least) two possible ways of listing the assembled object codes, either to display them in sequence according to the memory address or according to the sequence of the source lines. In most cases these two options are the same but there are occasions where the two sequences differ. The COSC-1 assembly language provides instruction that allows the user to alter the memory address sequence so that the order of the source lines does not correspond directly to the memory address sequence. There is another complication, some instructions involve "pseudo operators" that will not cause any object code to be generated and therefore cannot tell their presence by just looking at the memory words. Displaying the object codes according to the memory address sequence would not show these source lines n.

To avoid complication, it was decided that a listing according to memory address sequence will not have the corresponding source lines displayed. This can be done at the end of the assembly when all the memory words hold their appropriate contents, the program needs only to go through the memory once, accessing one word at a time and putting its contents in the **Object codes** window. To give a listing that has the source codes displayed along side the object codes, it will be displayed according to the source lines' sequence and this will be done during the second pass of the assembly. As each source line is being assembled and encoded, the object codes is displayed immediately together with the source line. With this method the lines with "pseudo operators" can be displayed in the right sequence too. There is a trade off in speed : displaying object codes only is much faster than listing with source codes alongside the object codes. It is suspected that the extra time taken is a memory management problem : each time as a line is to be added to the window, memory spaces need to be allocated for the **TERecord** to store the line of text and this takes up time. One possible way of improving the performance would be to allocate a block of memory space enough to hold the entire object code and source lines before assembling so that it can then be used without affecting the memory management much.

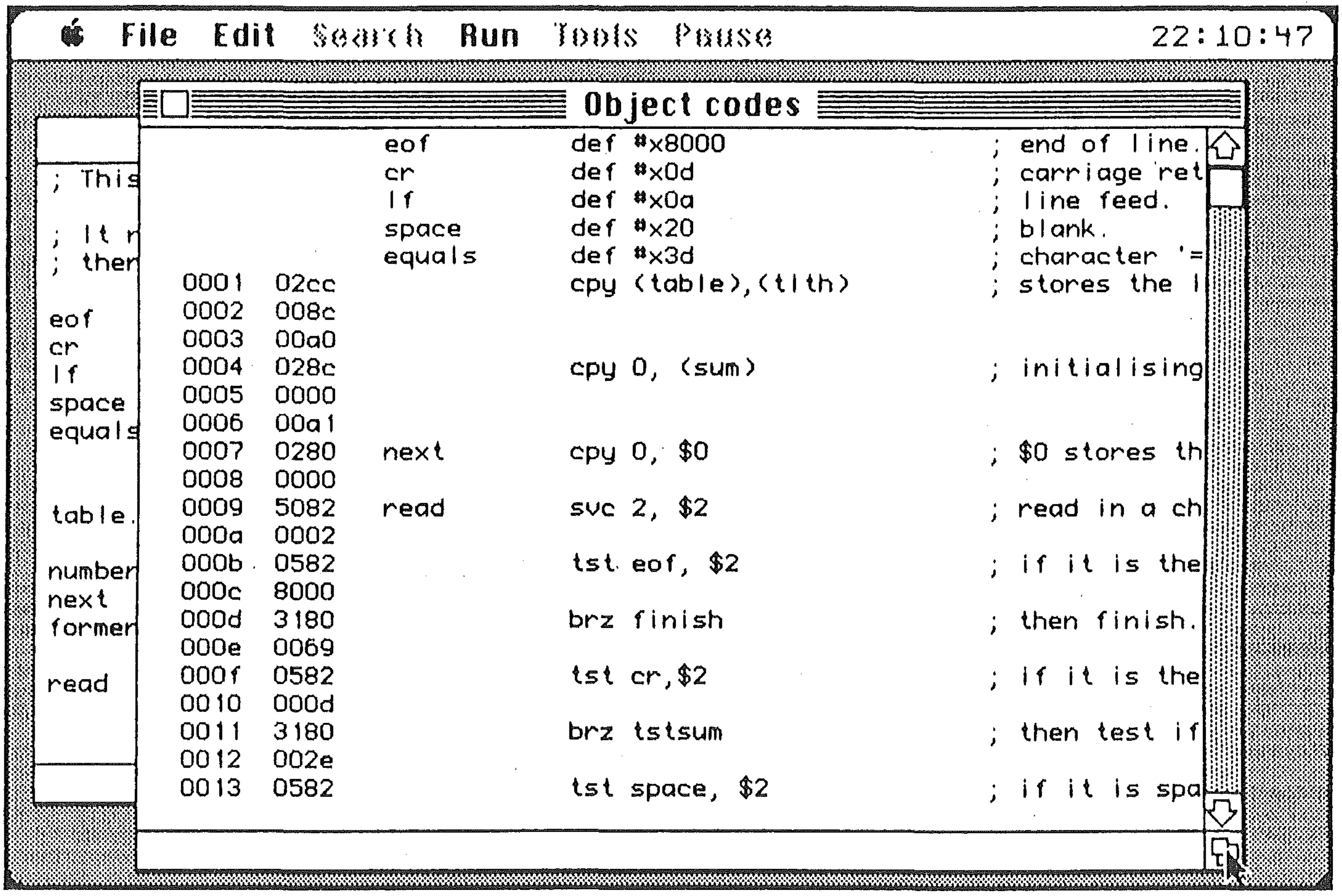
The two options mentioned above will be made available to the user before the object code is displayed through a dialog box. The suggested option is to display the object code in the order of the source lines because this will reflect the actual working of the COSC-1 instructions : if an instruction alter the sequence of the address, it is a good idea to see it vividly on the screen, it helps the user to understand COSC-1 programming better.

If it is desired to have the source code displayed alongside the object code even though the code is displayed in memory-address order, a way can be devised, perhaps with the overhead of storing more things in the memory words. Each memory word will need to have two extra fields, one as a flag to indicate whether it is the <u>first word of an instruction</u> and the second to store <u>an index to the source line</u> in the **TERecord** if it is the first word. When displaying, the program would first display the address and the contents, then check if it is a first word. If so, the index would be used to access the source line and put it into the window. Those source lines that did not generate any code could perhaps have their indices stored in a separate list and displayed at the end.

The object codes window has only a vertical scroll bar and the text is wrapped around. For the same reasons discussed in Section 3.3 concerning the editor window, it would be best for the object codes window to have both vertical and horizontal scroll bars, and not wrap around the text, from the point of view of helping the user to understand COSC-1 programming. However, because of limitation of time, the horizontal scroll bar was not implemented. This does not defeat the purpose of the object codes window. For the most part, the text that disappear out of the right side of the window is made up of comments ; further the size of the window can be extended to show a larger area. This should be a temporary measure only, in a future implementation the horizontal scroll bar is strongly recommended.

Figure 3.4.1 shows an example listing of assembled object codes of a COSC-1 assembly-language program.

The assembled object codes are stored in the simulated memory ready to be executed by the COSC-1 machine. Since COSC-1 is not a "real" machine, execution of COSC-1 programs is simulated by the program too. Unfortunately this part was not implemented because of shortage of time.



```
; This
; It r
; then

eof
cr
lf
space
equals

table.

number
next
former

read
```

```
                    eof         def #x8000                  ; end of line.
                    cr          def #x0d                    ; carriage ret
                    lf          def #x0a                    ; line feed.
                    space       def #x20                    ; blank.
                    equals      def #x3d                    ; character '=
0001    02cc                    cpy (table),(tlth)          ; stores the l
0002    008c
0003    00a0
0004    028c                    cpy 0, (sum)                ; initialising
0005    0000
0006    00a1
0007    0280        next        cpy 0, $0                   ; $0 stores th
0008    0000
0009    5082        read        svc 2, $2                   ; read in a ch
000a    0002
000b    0582                    tst eof, $2                 ; if it is the
000c    8000
000d    3180                    brz finish                  ; then finish.
000e    0069
000f    0582                    tst cr,$2                   ; if it is the
0010    000d
0011    3180                    brz tstsum                  ; then test if
0012    002e
0013    0582                    tst space, $2               ; if it is spa
```

Figure 3.4.1 Assembled object codes displayed on "Object codes" window

3.5 Problems and future extensions

Throughout the process of developing MacCOSC1, problems arose in different places. Among these difficulties, learning to use the Toolbox routines was the most tedious and time consuming. There are some hundreds of special routines built into the ROM (Read Only Memory) of the Macintosh for the purpose of supporting its user interface. A Macintosh application program usually relies heavily on these Toolbox routines to support the same kind of environment, besides performing its intended functions. Macintosh Revealed [3] offers great help in identifying these routines and their special functions.

Memory management is one of the most tricky things in Macintosh programming. Address errors usually result in a fatal error whereby rebooting is the only way to recover from the error. The Macintosh allocates memory spaces to the application mainly from the heap and these blocks of memory space, as far as possible, are made relocatable. It is in using these relocatable blocks that problems usually arise. The reponsibility of "locking" these relocatable blocks when in use has to be borne by the application program. Fortunately as development progresses these occurrences are less frequently.

There is one subtle bug in MacCOSC1 that remains to be identified. It is related to multiple window handling. If both the editor window and object codes window are opened, in the midst of switching from one window to the other the contents of the windows sometimes seem to disappear mysteriously. The contents are "lost but not lost", lost in that they disappear from the window and can't be brought back into view, but not lost because they remain in the storage in **TERecord**. The scroll bar continues to function but the content will not appear in the window. There is no regular pattern to these occurrences, nor any symptoms before it happens. Work is still going into finding the cause of this problem.

Lightspeed C consumes a huge amount of memory space in execution ; even with a Macintosh Plus, a ramdisk of larger than 450K in size will most likely prohibit a successful compilation and execution of the application program, which itself takes up 240K storage place. However, if the application program is built into a Macintosh application, it only occupies approximately 40K of memory. This certainly takes away the fear of the application being too large to be used on a smaller Macintosh. The shortage of memory space during development is a real nuisance as it requires (very) frequent disk swapping. Furthermore, Lightspeed C itself has no interactive debugging facilities at all ; many times to restart the Macintosh is the only error "recovery" facility !

There remain two more components to be developed to make MacCOSC1 a complete system. So far, with the assembler, a COSC-1 program can be assembled but no execution is performed by the COSC-1 machine. Execution of the object codes has to be simulated by the program and it is hoped that it will be implemented in a future version. Also, as mentioned earlier on, the interactive debugger has not been developed at this stage. The reason behind this is again due to limitation of time.

Extensions to MacCOSC1 in the future hopefully will complete all the features suggested for the editor ; multiple source files existing at the same time. And of course full features of the debugger will be expected from it.

Section 4
Conclusion

This project has presented a great challenge from the start. It brought me into a new area of programming contrasted to what I have done in the past years. Unfortunately, due to shortage of time, the aim of implementing a fully operational MacCOSC1 was not accomplished. However, for the part that has been achieved, it confirms the feasibility of implementing MacCOSC1 given sufficient time. Besides being a programming-intensive project, it certainly has been a training ground for learning to be patient too !

Section 5
References

The following references are used in preparation of this report and in the development of MacCOSC1.

[1] B. W. Kernighan & D. M. Ritchie;
The C Programming Language ; Prentice-Hall.

[2] *Lightspeed C™ Users' Manual* ; THINK TECHNOLOGIES INC.

[3] S. Chernicoff ; *Macintosh Revealed* (vol. 1 & 2) ;
Hayden Book Company.

[4] "The COSC-1 Assembly Language Simulator" ;
COSC112 Handout, Dept. of Computer Science.

[5] *Inside Macintosh* (vol. 1, 2 and 3); Apple Computer, Inc.

[6] "The COSC-1 Computer - Structure and programming" ;
COSC112 Handout, Dept. of Computer Science.

Section 6

MacCOSC1 Users' Manual

6.0

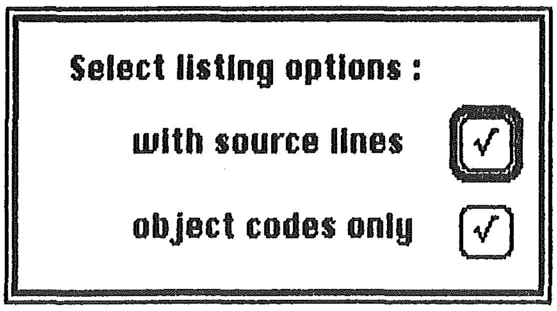
This section provides a walkthrough of the basic features of the MacCOSC1 environment. It describes how to start up a new program, modifying existing programs and assembling a program. The description below assumes the user has experience in using Macintosh. If you have not used a Macintosh before, you should first consult *Macintosh*, the Macintosh owner's manual, before starting on MacCOSC1. You are also assumed to be familiar with the COSC-1 assembly language and its instruction format; if in doubt you should refer to the reference materials [6].

6.1 Getting Started

Double click on the MacCOSC1 icon to start up the MacCOSC1 programming environment. The editor will be started off with an empty window ready for writing a new program. If you are starting up a new program, just go ahead and type in your COSC-1 program and it will show up in the window. If you want to look at an existing program, select the **Open** option in the **File** menu and you will see a file selection dialog box appearing on the screen. Select the file that you want to read and click on the **Open** button, the editor will bring the file to the window.

After you have completed writing a program, or modified an existing program, you can save it by choosing the **Save** command in the **File** menu. You can also print a copy of your program by selecting the **Print** command from the **File** menu. Detail descriptions of the menus are found in sub-section **6.3 Menu Reference.**

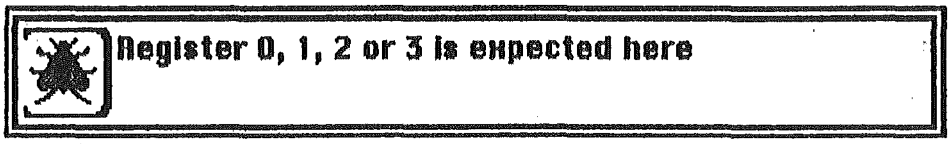
To assemble a program, you must first ensure that the program is present in the editor window, then choose from the **Run** menu the option **Assemble** and the assembly process will start. If there is no syntax error in your program, a dialog box will appear to ask you to select a listing option for the assembled object codes. Click on the appropriate button to confirm your choice.


Select listing options :

with source lines ✓

object codes only ✓

If the assembler discovered any error in your program, an error message will appear on the top portion of the screen. For example,


Register 0, 1, 2 or 3 is expected here

You can dismiss the message by clicking anywhere inside the message box. MacCOSC1 will then put you back to the editor and the cursor will be positioned right on the line where the error was found. An exception to this, if the error is "undefined symbol", is that the cursor position has no meaning since the assembler can not predict where the user might want to define the symbol.

When you have finished with your work, you can leave MacCOSC1 by selecting the **Quit** command from the **File** menu and you will end up in the Finder, where you started before.

6.2 Editing

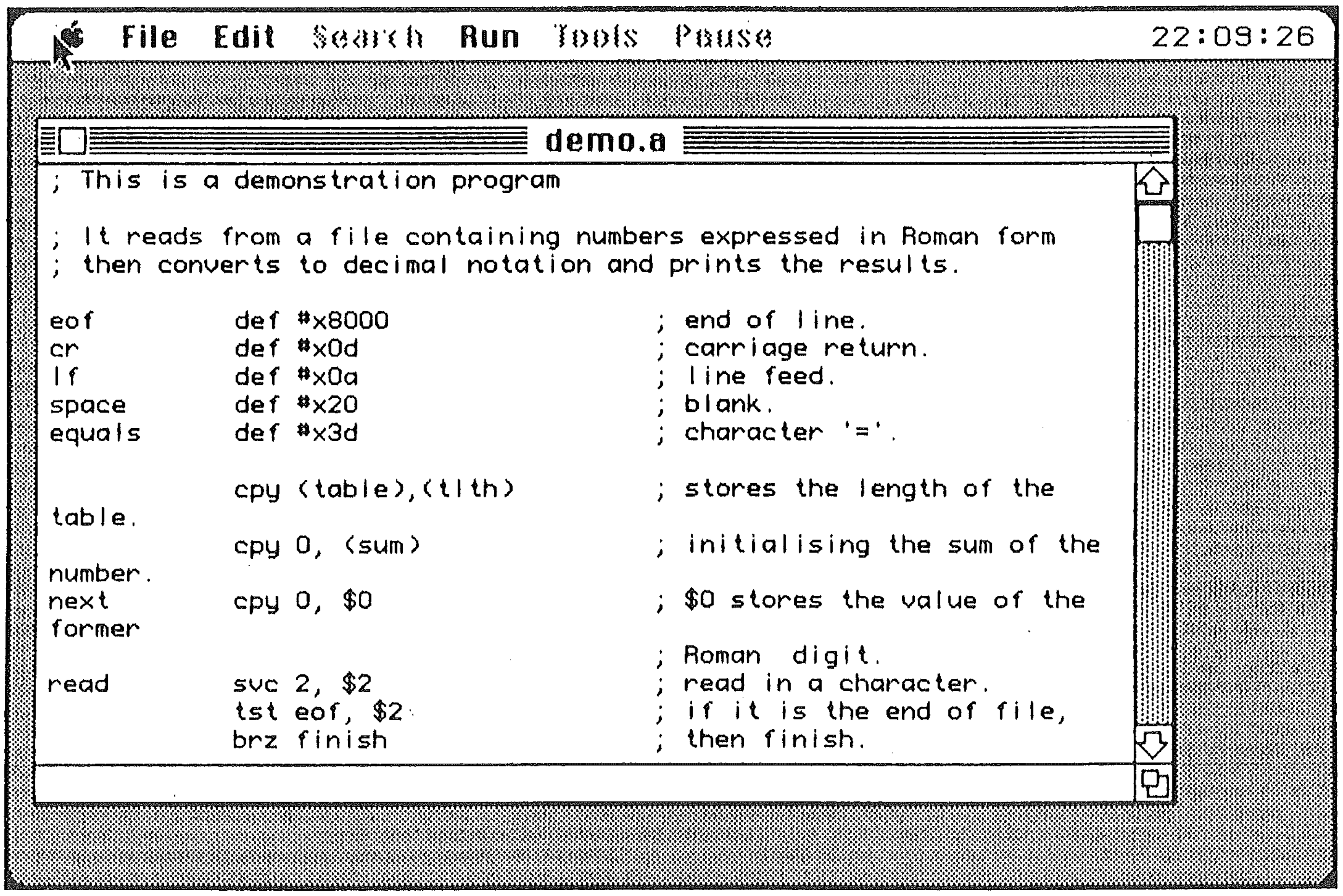
MacCOSC1 has its own text editor for entering and modifying COSC-1 programs. Figure 6.2.1 shows how the editor window looks on the screen.

Moving around in a File

In the usual Macintosh fashion, you can move around in the edit window by moving the mouse cursor to the point where you want to insert text. Then click the mouse button once to move the text cursor to that point. We'll call that the *insertion point* from now on.

You can control what part of the file is displayed in the window using the scroll bar. If the file is larger than will fit in the current window, you will notice that there is a little white box (called the thumb) in the gray bar on the right side of the screen. The position of the thumb in that bar is proportional to the position of the current screenful of text in the body of the file. To shift the window to view another part of the file, use the mouse to move the thumb up or down in the scroll bar. The window will shift accordingly. If the entire contents of the file will fit in the current window, the thumb will not be visible in the scroll bar.

You can also move up or down a line at a time by clicking on the arrow at the top or bottom of the scroll bar. Clicking in the gray area above or below the thumb moves a screenful at a time. Holding the mouse down in either location causes repeated scrolling of the text.



```
; This is a demonstration program

; It reads from a file containing numbers expressed in Roman form
; then converts to decimal notation and prints the results.

eof         def #x8000                    ; end of line.
cr          def #x0d                      ; carriage return.
lf          def #x0a                      ; line feed.
space       def #x20                      ; blank.
equals      def #x3d                      ; character '='.

            cpy (table),(tlth)            ; stores the length of the
table.
            cpy 0, (sum)                  ; initialising the sum of the
number.
next        cpy 0, $0                     ; $0 stores the value of the
former
                                          ; Roman  digit.
read        svc 2, $2                     ; read in a character.
            tst eof, $2                   ; if it is the end of file,
            brz finish                    ; then finish.
```

Figure 6.2.1 MacCOSC1 editor window

Changing the Size and Position of the Window

The editing window is growable in the standard Macintosh fashion by grabbing the bottom right corner with the mouse cursor and dragging it to shrink or expand the window. You can move it by grabbing it anywhere in the title bar and dragging it to the new location.

The ability to shrink or grow windows and move them around on the screen is most useful when you want to switch back and forth between multiple windows and want to make sure that a portion of each window is visible so that you can easily switch back and forth between them.

Editing Text

The **Cut**, **Copy**, **Paste** and **Clear** commands on the **Edit** menu are supported in the standard Macintosh fashion.

To select text for these *commands, move the mouse cursor to* the beginning (or end) of the text to be selected. Then, while holding down the mouse button, move the mouse cursor to the opposite end of the text to be selected. The text that is inverted from black-on-white to white-on-black has been selected. You can select a single word by double-clicking on it.

The **Cut** command removes selected text and places it in the Clipboard. It replaces the current contents of the Clipboard (if there are any). Material that has been cut can be pasted back into the file, as long as nothing else has been cut or copied to the Clipboard, by moving the insertion point to the desired location and issuing a **Paste** command.

Text can also be deleted (without the option of pasting it back somewhere else) by selecting it, and then choosing **Clear** from the **Edit** menu or by pressing the backspace key.

The **Copy** command copies the selected text and places it in the Clipboard. The copy can now be pasted somewhere else using the **Paste** command.

Selecting any region of text and pressing any key which would insert a character will replace the entire selection with the character generated by that keystroke. This is a standard, but dangerous, Macintosh feature.

Notice that if a line is longer than can fit into the width of the window, it is wrapped around to the next line by the editor. This is not a new line as far as MacCOSC1 is concerned, a new line in MacCOSC1 is started when the ‹return› key is pressed.

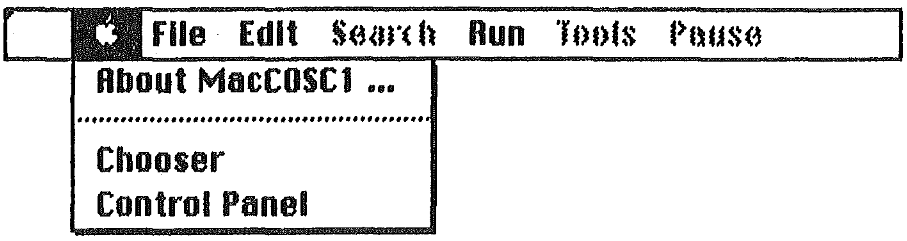
The **Undo** command on the **Edit** menu has no function in MacCOSC1 and is not implemented.

6.3 Menu Reference

This subsection gives a brief summary of the function of each of the MacCOSC1 menu commands. It is organised by menu, from left to right along the menu bar. Within each menu, commands are described in the order in which they appear on the menu. This allows you to quickly look up the function of each of the commands.

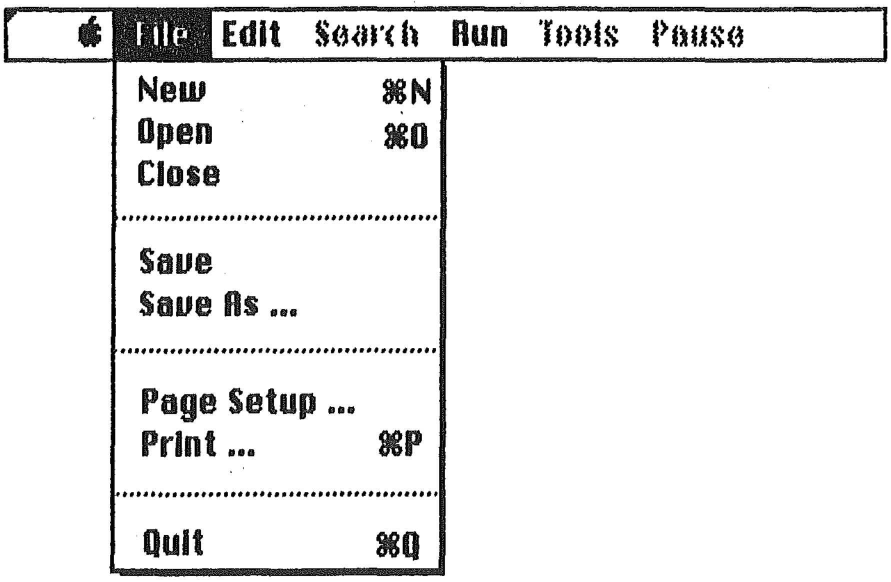
Commands that are not implemented yet will be mentioned by name and their intended use is described. They appear on the screen "dimmed".

The Apple Menu



About MacCOSC1 ...
This command gives version information about MacCOSC1. Click the **OK** button to exit.

The File Menu



New
This command opens a new file, with a window name *Untitled.*

Open
This command displays a dialog box that allows you to select from existing files on the disk and open them for editing.

Close
This command allows you to close any window with a *go away* box in its upper left corner, including the active edit window or desk accessory. If you try to close an edit window, and the file has been modified since it was last saved, a dialog box will ask you if you want to save the changes, discard them or cancel the close command. Issuing this command is the same as clicking on the go away box.

Save
This command saves the file in the active edit window to disk. If the file is currently untitled, a dialog file will ask you to name the file.

Save As....
This command allows you to save the current file under another name. If you have made edits in the current session, they will be saved under the new name. The original file will remain unchanged, and as you continue editing, you will be editing the new file.

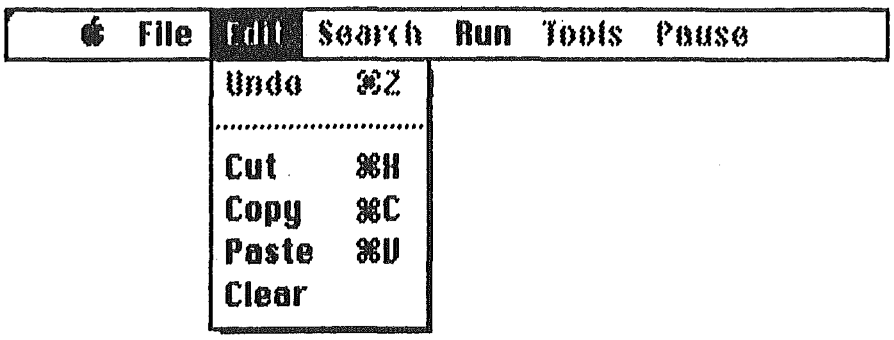
Page Setup ...

This command allows you to specify the size of the paper you're printing on, and whether the file should be printed upright on the page (tall orientation) or sideways (wide orientation).

Print....
This command allows you to print the current file. A dialog box will prompt you for various settings. Note that the document of the *active* window is the one to be printed and at present only source programs and object code listings can be printed.

Quit
This command exits MacCOSC1 and returns to the desktop or Mini Finder.

The Edit Menu



The **Edit** menu contains the standard Macintosh editing functions **Undo, Cut, Copy, Paste** and **Clear.**

Undo
This command does nothimg in MacCOSC1 and is therefore dimmed.

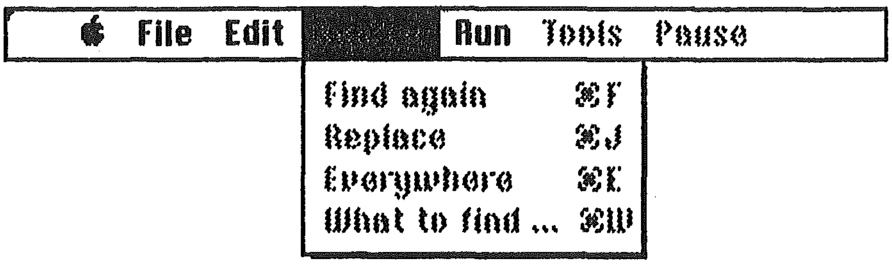
Cut
This command removes selected text and places it in the Clipboard. It replaces the current contents of the Clipboard (if there are any). Material that has been cut can be pasted back into the file, as long as nothing else has been cut or copied to the Clipboard, by moving the cursor to the desired location and issuing a **Paste** command. Text can also be deleted (without the option of pasting it back somewhere else) by selecting it, and then choosing **Clear** from the **Edit** menu or pressing the backspace key.

Copy
This command copies the selected text and places it in the Clipboard. The copy can now be pasted somewhere else using the **Paste** command.

Paste
This command copies the contents of the Clipboard into the file being edited at the insertion point. If the text is currently selected, it is replaced.

Clear
This command clears the selected text. The selection is not placed on the Clipboard, and cannot be recovered.

The Search Menu



This menu has not been implemented yet. The intended use of each command is described below.

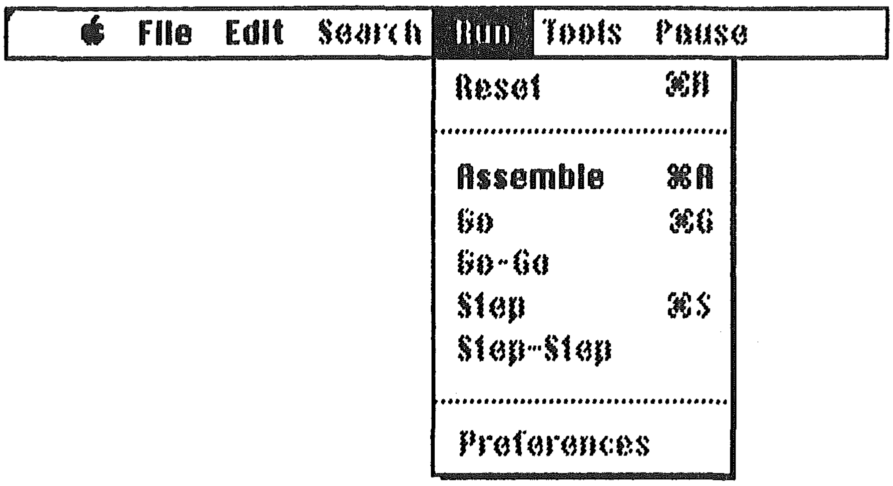
Find again
Finds the next occurrence of the pattern specified.

Replace
Replaces the pattern found with the replacement specified.

Everywhere
Replaces all the occurrences of the specified pattern with the replacement given.

What to find ...
Brings up a dialog box to enter the pattern to be searched for and the replacement, if desired.

The Run Menu



Reset
Resets the COSC-1 machine so that the next execution of that program is just like starting from new. Not implemented yet.

Assemble
This commands causes assembling of the COSC-1 progrram currently worked on. Successful assembly will produce the object codes which will be displayed in the **Object codes** window.

Go
Performs a "fast run" : execution only stops if it is completed or encounter an error. Not implemented yet.

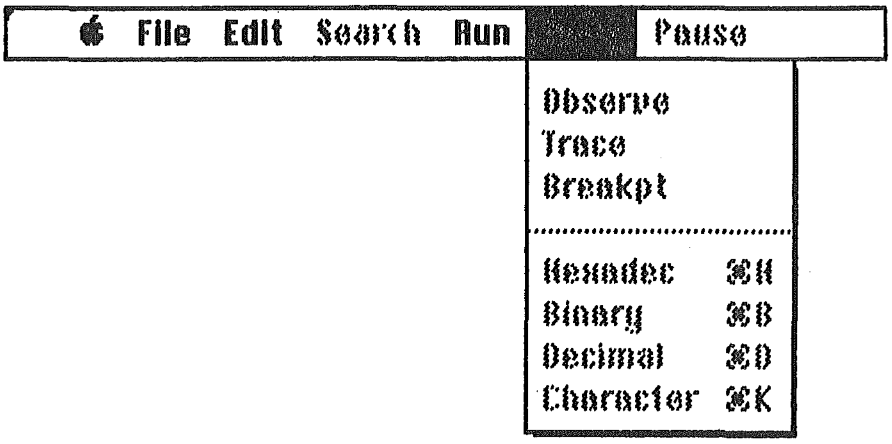
Go-Go
Runs the program but pauses at breakpoints, execution will resume if you click on the **continue** button which appears when there is a pause. Not implemented yet.

Step
Performs execution of one instruction and waits for a **Step** command from the user to continue to the next. Click on the **continue** button (which will appear in the right bottom corner) or press <RETURN> key to execute the next instruction. Not implemented yet.

Step-Step
Performs execution of the program instruction by instruction with a short pause in between. It will stop at breakpoints and resume execution only if instructed to do so (click on the **continue** button). Not implemented yet.

Preferences
Brings up a dialog box that allows the user to set various parameters like : auto indent mode, set font, output to printer as well as on the **Output** window, input source - file or keyboard and so on. It is intended that listing options for object codes would be moved into this dialog box as well. Settings are for the current session only. Not implemented yet.

The Tools Menu



This menu has not been implemented yet. The intended use of each command is described below.

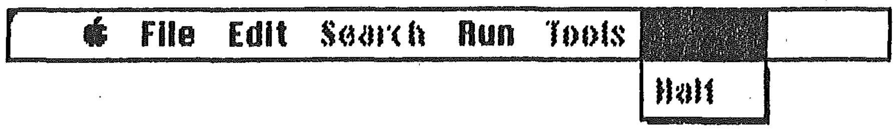
Observe
Opens up the **Observe** window so that at execution time, the contents of the registers, the program counter and the flag word will be shown.

Trace
Opens up the **Trace** window that allows you to enter expressions whose values you want to examine during execution.

Breakpt
This command will highlight the "breakpoint column" on the object codes window with a "breakpoint mark" at the left bottom corner. Click on the mark and drag a copy of it along the "breakpoint column" to the line where you want to set the breakpoint, release the mouse button and a breakpoint will be set there as indicated by the "breakpoint mark".

Hexadec
Binary
Decimal
Character
To use any of these commands, first select the contents of a memory word in the object codes window, then select any of the above commands to convert the information into the desired notation : **Hexadec** for hexadecimal notation, **Binary** and **Decimal** are self-explaining, **Character** converts the information into the corresponding ascii character.

The Pause Menu



This menu has not been implemented yet. It is intended that during execution, clicking the mouse on **Pause** will temporarily stop the execution, if the user then selected the command **Halt** the execution will be terminated.

Appendix A

Sample COSC-1 Program

The following pages are the source listing, the assembled object codes listing with and without source lines of the program written by a COSC112 for his assigment. Tests performed on this program shows that the time taken to assemble the program is well below 1 second.

```
; This is a demonstration program

; It reads from a file containing numbers expressed in Roman form
; then converts to decimal notation and prints the results.

eof         def #x8000                    ; end of line.
cr          def #x0d                      ; carriage return.
lf          def #x0a                      ; line feed.
space       def #x20                      ; blank.
equals      def #x3d                      ; character '='.

            cpy (table),(tlth)            ; stores the length of the table.
            cpy 0, (sum)                  ; initialising the sum of the number.
next        cpy 0, $0                     ; $0 stores the value of the former
                                          ; Roman  digit.
read        svc 2, $2                     ; read in a character.
            tst eof, $2                   ; if it is the end of file,
            brz finish                    ; then finish.
            tst cr,$2                     ; if it is the carriage return,
            brz tstsum                    ; then test if the sum >0 ?
            tst space, $2                 ; if it is space,
            brz tstsum                    ; then test if the sum > 0 ?
            cpy $2, (ch)                  ; store the character at location ch.
            brs value, $3                 ; branch to subroutine value.
            tst -2,$1                     ; is it an invalid character ?
            brz prerrmsg                  ; if it is, print an error message.
            cpy $1,$2                     ; copy the value of the digit in $2.
            tst $2,$0                     ; if former digit > present digit,
            brp addition                  ; then add former digit to the sum.
            tst $2,$0                     ; if former digit < present digit,
            brm subtract                  ; then subtract former digit from sum
subtract    sub $0, (sum)                 ; former digit is subtracted from sum
            bru swap
addition    add $0, (sum)                 ; former digit is added to the sum.
            bru swap
tstsum      cpy 0, $2                     ; initialise the last digit to zero.
            add $0, (sum)                 ; add the last digit to the sum.
            tst 0, (sum)                  ; if the sum is non-zero,
            brn prsum                     ; then print the sum.
swap        cpy $2, $0                    ; store the present digit in the
                                          ; former digit location before
            bru read                      ; reading the next digit.
prerrmsg    svc 3, (ch)                   ; print the invalid character.
            svc 2, (ch)                   ; read the next digit in the invalid
                                          ; character.
            tst cr, (ch)                  ; if it is the end of the invalid
            brz printch                   ; number, print the error message.
            tst space, (ch)               ; if it is the end of the invalid
            brz printch                   ; number, print the error message.
            bru prerrmsg                  ; go and read the next digit in
                                          ; the invalid number.
printch     svc 3, equals                 ; print character '='.
            cpy errmsg, $3                ; $3 gets address of the first
                                          ; character in the error message.
again       svc 3, ($3)                   ; print the character in the message.
            add 1,$3                      ; go to the next address.
            tst 0, ($3)                   ; is it the end of the message ?
            brn again                     ; if it is not, print the next
                                          ; character in the message.
            bru loop                      ; if it is, read the next number
prsum       svc 3,equals                  ; print character '='.
            svc 5, (sum)                  ; print the sum.
loop        svc 3, cr                     ; advance to the
            svc 3, lf                     ; next line.
            bru next                      ; go and read the next digit.
finish      hlt                           ; stop.
;
; subroutine to convert the Roman number system to the ordinary
; decimal  positional notation.
```

```
value       cpy $3, (save3)           ; save the return address.
            cpy table, $3             ; $3 gets the address of table.
            cpy 0, (count)            ; initialise the counter for
                                      ; counting the no. of addresses
                                      ; already checked.
test        add 1, (count)            ; increment the counter by 1.
            add 1, $3                 ; go to next address.
            tst $2, ($3)              ; is the character in this address ?
            brz copy                  ; if yes, go and print the character.
            tst (tlth), (count)       ; have all the locations in the
                                      ; table been checked ? if yes, and the
                                      ; character read is not in it,
            brz assign
            bru test                  ; if all the locations has not been
                                      ; checked, then go to the next
                                      ; address.
assign      cpy -2, $1                ; $1 gets value -2 if the character
                                      ; is invalid.
            bru return                ; digit to zero.
copy        svc 3, (ch)               ; print the valid character.
            cpy ($3+1), $1            ; store its value in $1.
return      cpy (save3), $3           ; restore the return address in $3.
            bru $3                    ; and return to the calling program.
table       wrd 18                    ; length of the table.
            wrd #x0a                  ; space.
            wrd 0                     ; zero.
            wrd #x0d                  ; carriage return.
            wrd 0                     ; zero.
            wrd #x49                  ; letter 'I'.
            wrd 1                     ; one.
            wrd #x56                  ; letter 'v'.
            wrd 5                     ; five.
            wrd #x58                  ; letter 'x'.
            wrd 10                    ; ten.
            wrd #x4c                  ; letter 'l'.
            wrd 50                    ; fifty.
            wrd #x43                  ; letter 'c'.
            wrd 100                   ; one hundred.
            wrd #x44                  ; letter 'd'.
            wrd 500                   ; five hundred.
            wrd #x4d                  ; letter 'm'.
            wrd 1000                  ; one thousand.
count       wrd 0                     ; for counting the no. of addresses
                                      ; already checked in the table.
tlth        wrd 0                     ; the no. of addresses in the table.
sum         wrd 0                     ; reserve a space for storing the sum
save3       wrd 0                     ; reserve a space for storing the
                                      ; return address of the subrountine.
ch          wrd 0                     ; reserve a space for storing the
                                      ; character read.
errmsg      wrd #x45                  ; letter 'e'.
            wrd #x52                  ; letter 'r'.
            wrd #x52                  ; letter 'r'.
            wrd #x4f                  ; letter 'o'.
            wrd #x52                  ; letter 'r'.
            wrd 0                     ; sero for indicating end of message.
            end
```

```
                eof         def #x8000              ; end of line.
                cr          def #x0d                ; carriage return.
                lf          def #x0a                ; line feed.
                space       def #x20                ; blank.
                equals      def #x3d                ; character '='.
0001    02cc                cpy (table),(tlth)      ; stores the length of the table.
0002    008c
0003    00a0
0004    028c                cpy 0, (sum)            ; initialising the sum of the number.
0005    0000
0006    00a1
0007    0280    next        cpy 0, $0               ; $0 stores the value of the former
0008    0000
0009    5082    read        svc 2, $2               ; read in a character.
000a    0002
000b    0582                tst eof, $2             ; if it is the end of file,
000c    8000
000d    3180                brz finish              ; then finish.
000e    0069
000f    0582                tst cr,$2               ; if it is the carriage return,
0010    000d
0011    3180                brz tstsum              ; then test if the sum >0 ?
0012    002e
0013    0582                tst space, $2           ; if it is space,
0014    0020
0015    3180                brz tstsum              ; then test if the sum > 0 ?
0016    002e
0017    022c                cpy $2, (ch)            ; store the character at location ch.
0018    00a3
0019    4083                brs value, $3           ; branch to subroutine value.
001a    006a
001b    0581                tst -2,$1               ; is it an invalid character ?
001c    fffe
001d    3180                brz prerrmsg            ; if it is, print an error message.
001e    003a
001f    0212                cpy $1,$2               ; copy the value of the digit in $2.
0020    0520                tst $2,$0               ; if former digit > present digit,
0021    3380                brp addition            ; then add former digit to the sum.
0022    002a
0023    0520                tst $2,$0               ; if former digit < present digit,
0024    3480                brm subtract            ; then subtract former digit from sum
0025    0026
0026    040c    subtract    sub $0, (sum)           ; former digit is subtracted from sum
0027    00a1
0028    3080                bru swap
0029    0037
002a    030c    addition    add $0, (sum)           ; former digit is added to the sum.
002b    00a1
002c    3080                bru swap
002d    0037
002e    0282    tstsum      cpy 0, $2               ; initialise the last digit to zero.
002f    0000
0030    030c                add $0, (sum)           ; add the last digit to the sum.
0031    00a1
0032    058c                tst 0, (sum)            ; if the sum is non-zero,
0033    0000
0034    00a1
0035    3280                brn prsum               ; then print the sum.
0036    005b
0037    0220    swap        cpy $2, $0              ; store the present digit in the
0038    3080                bru read                ; reading the next digit.
0039    0009
003a    508c    prerrmsg    svc 3, (ch)             ; print the invalid character.
003b    0003
003c    00a3
003d    508c                svc 2, (ch)             ; read the next digit in the invalid
003e    0002
003f    00a3
0040    058c                tst cr, (ch)            ; if it is the end of the invalid
```

```
0041  000d
0042  00a3
0043  3180               brz printch           ; number, print the error message.
0044  004c
0045  058c               tst space, (ch)       ; if it is the end of the invalid
0046  0020
0047  00a3
0048  3180               brz printch           ; number, print the error message.
0049  004c
004a  3080               bru prerrmsg          ; go and read the next digit in
004b  003a
004c  5088    printch    svc 3, equals         ; print character '='.
004d  0003
004e  003d
004f  0283               cpy errmsg, $3        ; $3 gets address of the first
0050  00a4
0051  5087    again      svc 3, ($3)           ; print the character in the message.
0052  0003
0053  0383               add 1,$3              ; go to the next address.
0054  0001
0055  0587               tst 0, ($3)           ; is it the end of the message ?
0056  0000
0057  3280               brn again             ; if it is not, print the next
0058  0051
0059  3080               bru loop              ; if it is, read the next number
005a  0061
005b  5088    prsum      svc 3,equals          ; print character '='.
005c  0003
005d  003d
005e  508c               svc 5, (sum)          ; print the sum.
005f  0005
0060  00a1
0061  5088    loop       svc 3, cr             ; advance to the
0062  0003
0063  000d
0064  5088               svc 3, lf             ; next line.
0065  0003
0066  000a
0067  3080               bru next              ; go and read the next digit.
0068  0007
0069  0100    finish     hlt                   ; stop.
006a  023c    value      cpy $3, (save3)       ; save the return address.
006b  00a2
006c  0283               cpy table, $3         ; $3 gets the address of table.
006d  008c
006e  028c               cpy 0, (count)        ; initialise the counter for
006f  0000
0070  009f
0071  038c    test       add 1, (count)        ; increment the counter by 1.
0072  0001
0073  009f
0074  0383               add 1, $3             ; go to next address.
0075  0001
0076  0527               tst $2, ($3)          ; is the character in this address ?
0077  3180               brz copy              ; if yes, go and print the character.
0078  0084
0079  05cc               tst (tlth), (count)   ; have all the locations in the
007a  00a0
007b  009f
007c  3180               brz assign
007d  0080
007e  3080               bru test              ; if all the locations has not been
007f  0071
0080  0281    assign     cpy -2, $1            ; $1 gets value -2 if the character
0081  fffe
0082  3080               bru return            ; digit to zero.
0083  0089
0084  508c    copy       svc 3, (ch)           ; print the valid character.
0085  0003
```

```
0086  00a3
0087  02f1               cpy ($3+1), $1            ; store its value in $1.
0088  0001
0089  02c3   return      cpy (save3), $3           ; restore the return address in $3.
008a  00a2
008b  3030               bru $3                    ; and return to the calling program.
008c  0012   table       wrd 18                    ; length of the table.
008d  000a               wrd #x0a                  ; space.
008e  0000               wrd 0                     ; zero.
008f  000d               wrd #x0d                  ; carriage return.
0090  0000               wrd 0                     ; zero.
0091  0049               wrd #x49                  ; letter 'I'.
0092  0001               wrd 1                     ; one.
0093  0056               wrd #x56                  ; letter 'v'.
0094  0005               wrd 5                     ; five.
0095  0058               wrd #x58                  ; letter 'x'.
0096  000a               wrd 10                    ; ten.
0097  004c               wrd #x4c                  ; letter 'l'.
0098  0032               wrd 50                    ; fifty.
0099  0043               wrd #x43                  ; letter 'c'.
009a  0064               wrd 100                   ; one hundred.
009b  0044               wrd #x44                  ; letter 'd'.
009c  01f4               wrd 500                   ; five hundred.
009d  004d               wrd #x4d                  ; letter 'm'.
009e  03e8               wrd 1000                  ; one thousand.
009f  0000   count       wrd 0                     ; for counting the no. of addresses
00a0  0000   tlth        wrd 0                     ; the no. of addresses in the table.
00a1  0000   sum         wrd 0                     ; reserve a space for storing the sum
00a2  0000   save3       wrd 0                     ; reserve a space for storing the
00a3  0000   ch          wrd 0                     ; reserve a space for storing the
00a4  0045   errmsg      wrd #x45                  ; letter 'e'.
00a5  0052               wrd #x52                  ; letter 'r'.
00a6  0052               wrd #x52                  ; letter 'r'.
00a7  004f               wrd #x4f                  ; letter 'o'.
00a8  0052               wrd #x52                  ; letter 'r'.
00a9  0000               wrd 0                     ; sero for indicating end of message.
                         end
```

```
0001  02cc
0002  008c
0003  00a0
0004  028c
0005  0000
0006  00a1
0007  0280
0008  0000
0009  5082
000a  0002
000b  0582
000c  8000
000d  3180
000e  0069
000f  0582
0010  000d
0011  3180
0012  002e
0013  0582
0014  0020
0015  3180
0016  002e
0017  022c
0018  00a3
0019  4083
001a  006a
001b  0581
001c  fffe
001d  3180
001e  003a
001f  0212
0020  0520
0021  3380
0022  002a
0023  0520
0024  3480
0025  0026
0026  040c
0027  00a1
0028  3080
0029  0037
002a  030c
002b  00a1
002c  3080
002d  0037
002e  0282
002f  0000
0030  030c
0031  00a1
0032  058c
0033  0000
0034  00a1
0035  3280
0036  005b
0037  0220
0038  3080
0039  0009
003a  508c
003b  0003
003c  00a3
003d  508c
003e  0002
003f  00a3
0040  058c
0041  000d
0042  00a3
0043  3180
0044  004c
0045  058c
```

```
0046  0020
0047  00a3
0048  3180
0049  004c
004a  3080
004b  003a
004c  5088
004d  0003
004e  003d
004f  0283
0050  00a4
0051  5087
0052  0003
0053  0383
0054  0001
0055  0587
0056  0000
0057  3280
0058  0051
0059  3080
005a  0061
005b  5088
005c  0003
005d  003d
005e  508c
005f  0005
0060  00a1
0061  5088
0062  0003
0063  000d
0064  5088
0065  0003
0066  000a
0067  3080
0068  0007
0069  0100
006a  023c
006b  00a2
006c  0283
006d  008c
006e  028c
006f  0000
0070  009f
0071  038c
0072  0001
0073  009f
0074  0383
0075  0001
0076  0527
0077  3180
0078  0084
0079  05cc
007a  00a0
007b  009f
007c  3180
007d  0080
007e  3080
007f  0071
0080  0281
0081  fffe
0082  3080
0083  0089
0084  508c
0085  0003
0086  00a3
0087  02f1
0088  0001
0089  02c3
008a  00a2
```

```
008b   3030
008c   0012
008d   000a
008e   0000
008f   000d
0090   0000
0091   0049
0092   0001
0093   0056
0094   0005
0095   0058
0096   000a
0097   004c
0098   0032
0099   0043
009a   0064
009b   0044
009c   01f4
009d   004d
009e   03e8
009f   0000
00a0   0000
00a1   0000
00a2   0000
00a3   0000
00a4   0045
00a5   0052
00a6   0052
00a7   004f
00a8   0052
00a9   0000
00aa   0000
```