IPL-TR-087

Introduction
to
The TI Explorer

Douglas Lyon

October 22, 1986

IMAGE PROCESSING LABORATORY
Electrical, Computer, and Systems Engineering Department
Rensselaer Polytechnic Institute
Troy, New York 12180-3590

Table of Contents

PREFACE

The following manual is intended for the first time user of the Explorer. This is not a reference manual but an introduction. The Explorer is a Texas Instruments version of the MIT CADR machine and as such is properly classed as a Lisp machine. The native machine code is a dialect of lisp called Zetalisp. To provide compatibility a less powerful dialect of Lisp called Common Lisp is also available. The intention of the Explorer is to provide an environment for performing AI research. The machine gives the user access to the entire operating system (written in Zetalisp). The Explorer is set up as a personal computer with multiprocessing capabilities. There is no protection against accidental deletion of files. The naive user may well delete files by accident. Since there is no password protection and no file protection the new user is advised to be very careful about deleting files. All users are advised to back up files.

ACKNOWLEDGEMENT

The author is happy to acknowledge the following assistance received during the course of the development of this manual:

To Professor Jim Modestino who first suggested that this manual would be of use to new users of the Explorer.

To Steve Rezsutek for initial outline suggestions and for testing the manual on the Explorer as it was being written.

To Junichi Kanai for reading an initial draft of this document and for suggesting the section on programming.

To the Symbolics Education Center whose section on Flavors was borrowed to help make the section on programming complete.

To all those whom I have forgotten to mention.

Thanks!

D.L.

IPL PERSONNEL

Laboratory Director : Jim Modestino

Laboratory Manager : Ken Walter JEC 2304 x6800

Computer Operator : Doug Lyon JEC 6049 x8229

IPL Secretary : Linda Fischer JEC 6001 x6330

REBOOTING SYSTEM

Many time the user of the system will need to reboot. A reboot can take the form of a reinitialization of the lisp world. This will happen more frequently as more people begin to use larger programs. The reason is that every function uses space in the lisp world. Not only does it make the world smaller but it also uses up function names. Since all functions are global (except lambda expressions) most users of functions will want to start with a clean slate and a smaller world. Packages (which are not described here) and Flavor (which come later) are techniques which address this problem.

STOPPING THE PROGRAM
You can usually stop a running program which has gone awry by typing ABORT. If ABORT does not work, you can try META-ABORT or META-CTRL-ABORT.

WARM BOOT
If the system is totally locked up, you can warm boot by pressing META-CTRL-CTRL-RETURN. This will restart all processes without destroying the contents of virtual memory of edited buffers.

COLD BOOT
This destroys all data in core. Type META-CTRL-META-CTRL-ABORT.

CYCLING POWER
This is performed by pressing the button below the left most disk drive so that it is released. Then press it in again so that it is latched.

GENERAL USER I/O

MOUSE NOTATION

The optical mouse which accompanies the Explorer has 3 buttons. They are mouse-left (ML), mouse-middle (MM), and mouse-right (MR) respectively. Standard user I/O convention is to use the notation MXN to denote a mouse action. Here X may be L,M, or R for Left Middle or Right and N is an integer which indicates the number of times the button is pressed (generally once or twice). Thus to obtain the system menu type: MR2 (Mouse-Right-Twice). During your use of the Explorer you will discover that the mouse cursor will cause items on the display to become high-lighted. These items are called mouse sensitive items. The system convention for obtaining a default menu on a mouse sensitive item is to MOUSE-RIGHT.

KEYBOARD USE

Most of the control keys are not latched, that is you must hold them in order to have them modify the character code of another. For example HYPER-space requires you to hold the HYPER key down while pressing the space bar. HYPER, SUPER, META, SYMBOL, and CTRL are all unlatched keys. The exceptions are the keys at the top of the keyboard. HELP, SYSTEM, NETWORK, STATUS, and TERM are all latched keys. To select the LISP LISTENER you type SYSTEM-L. That is you push the SYSTEM key release it and then type an L.

SYSTEM MENU

The system menu contains many system programs for the user to try. They may be selected with a MOUSE-LEFT. You may use the system menu to enter the LISP LISTENER or ZMACS or PEEK (these will be described later).

SELECTING USING THE SYSTEM KEY

Another way to select system programs is to use the SYSTEM key. To see the available programs type: SYSTEM-HELP. This may be used to enter most of the programs listed in the system menu. The user may wish to try typing SYSTEM-G to obtain a glossary.

MAKING A DEMO SYSTEM

The user may wish to try making a demo system by evaluating an S-EXPRESSION in the LISP LISTENER. Type:

```
     SYSTEM-L
     (make-system 'demo :noconfirm :silent)
```

to load the demo system. This may not be necessary of the DEMO system has already been loaded. To tell if it is loaded, enter the system menu and look for DEMO under programs. If demo is listed, then it is loaded and may be entered using a mouse selection.

CREATING A LOGIN

STARTING A NEW USER

To start a new use on the system, enter the system menu and select the NEW USER selection under the USER AIDS section. When you are done following the directions given there you will be logged into the Explorer.

LOGGING OUT

To logout from the Explorer you select the LISP LISTENER and evaluate

```
(logout)
```

This will return 't' and you will be logged out of the machine.

LOGGING IN

Once your login ID has been established you may log back in by entering the LISP LISTENER and typing:

```
     (login 'ID)
```

Where ID is your user ID.

WINDOW MENUS

To customize the window configurations for your comfort, you must adjust the window attributes. You do this by selecting the window, entering the system menu, and selecting the WINDOW ATTRIBUTES section under the WINDOWS column. Some commonly adjusted attributes are: REVERSE VIDEO and FONT SELECTION.

SUBSYSTEMS

The following is a list of commonly used software subsystems and a brief description of each. Use the programming primer to get practice using these subsystems. The big 3 covered here are: ZMACS, LISP LISTENER and the PEEKER.

ZMACS COMMAND SUMMARY

CTRL-V View the next page
META-V View the previous page
CTRL-L Refresh this page
CTRL-F Move cursor Forward one character
CTRL-B Move cursor Backward one character
CTRL-N Move cursor down to Next line
CTRL-Z Move cursor up to previous line
META-< Move cursor to beginning of file
META-> Move cursor to end of file
META-S Move cursor to search forward
META-R Move cursor to search backward
CTRL-D Delete character at the cursor
CTRL-H Delete character before cursor
CTRL-K Delete text from cursor to end of line
CTRL-X CTRL-S Save buffer in editing file
CTRL-X CTRL-F Find file and read into buffer
CTRL-X CTRL-W Write buffer to file
CTRL-SH-A retrieves an argument list
META-. searches for a function
CTRL-META-A beginning of lisp form
CTRL-META-Q reformat lisp form
The following Meta-x commands require that the user type in the ZMACS subcommand.
META-X compile changed definitions
META-X save all files
META-X find unbalabced parenthesis
SYSTEM <CHAR> To select another system and exit ZMACS

You are now ready to type in your first program. Into the ZMACS buffer you must configure your file header. To do this type META-X set fonts RETURN The mini-buffer will prompt you for some font names. Try typing

```
     cptfont RETURN
```

Now it says: Change the (Blah) ? (Y or N)
Type a Y here and all subsequent times. This will change the file header in a human and machine readable form. You could just type a file header into the file but you would have to reevaluate the header. Now try to set the BASE. Type
META-X set base RETURN 10.
For this example, use base 10.
Now we will define a flavor for making a fractal texture.

```
;;;
;;; define a flavor for the fractal texture
;;;  generator
(defflavor fractal-texture-maker ((a 125)
                                  (b 125)
                                  (window terminal-io)
                                  (number-of-colors 1))
           ()
   :settable-instance-variables
;;; This flavor is a template for objects in the lisp
;;; environment. When we make an object from a flavor
;;; we have an instance of the object. The object
;;; has instance variables in it. In our case
;;; They are a, b, window and number-of-colors.
;;; Their defaults are 125, 125, terminal-io and 1
;;; respectivly.
;;;
;;; The following method will allow the message
;;; :doug-set to be sent to an instace of the
;;; fractal-texture-maker. The &key argument
;;; in the parameter list allows local variables
;;; to the method to be bound by default to the
;;; values accompaning them in the parenthesis.
;;; The user may change these upon invocation.
;;; More on this later.
(defmethod (fractal-texture-maker :doug-set) (&key (xoffset 0)
                                                   (yoffset 0)
                                                   (step .00001)
                                                   (x -.749)
                                                   (y  .1)
                                                   (xmax 1000)
                                                   (ymax 512))

;;; LET* allows  variables to be used
;;; as they are being calculated. Use of LET implies
;;; a parallel binding.
    (let* ((aspect 1)
           (temp 0)
           (k 0)
           (l 0)
           (xmax-on-2 (// xmax 2))
           (ymax-on-2 (// ymax 2)))
;;;
;;; Loop and while are macros which work like progn.
;;;
       (loop with j
             initially
              (setq j (- y (* step aspect ymax-on-2 (// ymax 2)))
           while (< i (+ x (* xmax-on-2 1) step))) do
              (setq i (+ i step))
              (setq temp (send self :iterate i j))
              (if (eq nil temp) (setq temp 255))
              (if (> temp 127)
                  (send window
                        :draw-point
                             (+ xoffset x) (+ yoffset y)))
              (setq k (+ k 1)))
        (setq l (+ l 1)))))

(defmethod (fractal-texture-maker :iterate) (c1 c2)
   (let* ((iter 255) (t1 0) (z1 0) (z2 0))
     (loop for i from 1 to iter do
       (if (or (> z1 2.0) (< z1 -2.0) (> z2 2.0) (< z2 -2.0))
           (return i)
           (setq t1 (+ (* (+ z1 z2) (- z1 z2)) c1)
                 z2 (* z1 z2)
                 z2 (+ z1 z1 c2)
                 i (+ i 1)
                 z1 (+ (* (+ t1 z2) (- t1 z2)) c1)
                 z2 (* z1 t1)
                 z2 (+ z2 z2 c2))))))
```

To save/compile the program type

```
CTRL-W pl : <login id>;  fractal.lisp
META-X COMPILE BUFFER
```

Now you may select the LISP LISTENER and make an instance of the new flavor. Type:

```
(setq foo (make-instance 'fractal-texture-mak
er))
```

A print name should be returned. Now you can send it a message. Before you start to output onto the screen, lets consider what will happen. If the program draws on the screen upon which you perform your typing then your typing will destroy what the program draws. If you were to describe foo by typing:

```
(decribe foo)
```

you would discover that the window instance variable in the foo instance of the fractal-texture-maker flavor is bound to an instance of the LISP LISTENER. You can verify this to be true by typing:

```
(print terminal-io)
```

Now an interesting way to go about drawing the DOUG-SET is to select zmacs and typing:

```
BREAK
```

This will suspend the zmacs edit session and give you a tempory typeout window with a LISP LISTENER. From here you may run your DOUG-SET. Type:

```
     (send foo :doug-set)
```

and then select your initial LISP LISTNER by typing

```
     SYSTEM-L
```

This will take about 20 minutes and should produce a pretty picture on the explorer.

LISP LISTENER

The LISP LISTENER is a read eval loop which expects to see lisp s-expressions. You select the LISTENER by typing SYSTEM-L. You may use this to evaluate programs.

PEEKER

The PEEKER is a subsystem for displaying the processes running on your Explorer. You may find that in the course of normal operations the window system locks and is unable to accept input. You may also find that a process needs to be reset or arrested. Type:

```
     SELECT-P
```

to peek at the processes on the Explorer. This will give you a menu which includes the option PROCESSES. Type a P or MOUSE on PROCESSES in order to display the processes. Some processes are quite important and should not be deleted, arrested or reset. You may have to cold boot if you do this. All the processes are mouse sensitive and a default menu may be obtained by using a MOUSE-RIGHT.

PROGRAMMING TUTORIAL

This section is intended for the programmer new to the Zetalisp environment. The naive programmer may be tempted to stick with the already acquired knowledge of Lisp and thus use Common Lisp, or not take advantage of the flavor system in Zetalisp. Be warned, the flavor system is the key to increased productivity on the Lisp machine and should be embraced by any programmer who wants to be a full fledged Lisp machine hacker. Your productivity will increase, your code will look cleaner, run faster and writing in Zetalisp will make your soul fly! Remember, you heard it here first.

ENTERING LISP MODES

To enter Zetalisp you type:

```
     (turn_zeta_lisp_on)
```

To enter Common Lisp you type:

```
     (turn_common_lisp_on)
```

To read more about language modes read section F of the Programming Primer.

DIFFERENCES BETWEEN ZETALISP AND COMMON LISP

Later in this section of the manual I will introduce Zetalisp for beginning Lisp programmers. I will not treat Common Lisp here because Lisp Experience is assumed on behalf of the reader and the standard reference for the Explorer (Common Lisp by Steele) is an excellent book for beginners. The main difference between Zetalisp and Common Lisp is the object orientation and its' implementation called the Flavor system.

WHAT ARE FLAVORS?

Flavors are programmer-defined data types, much like the kind that the type statement of Pascal lets you define (or the defstruct of Zetalisp). Many of the more complicated system data types are implemented as flavors. Understanding flavors is your key to the system.

Objects that have a flavor as their data type are called instances of that flavor. You can't create instances of a flavor until that flavor is defined. The system defines some flavors for you, so you can make instances of them right away.

Each flavor has a name, which is a symbol used to identify that flavor. The function that creates new instances (it's called make-instance) takes a flavor name as one of its arguments.

Each flavor instance has a table of instance variables. Each instance variable has a name, which is a symbol, and a value, which is any Lisp object. All the instances of one flavor have instance variables with the same names, but the values can differ from instance to instance. Two instances of different flavors can have completely different instance variables.

You can create new flavors with the special operator DEFFLAVOR, like this:

```
     (defflavor "flavor-name"
                 ("instance-variable-1"
                 "instance-variable-2"
                         .
                         .
                 "instance-variable-n"
                 ()
       :settable-instance-variables)
```

Note: Defflavor is like defun in that is is always at the top level in your file.

For example, let's create a flavor called ship.

```
     (defflavor ship
          (x y captain)
          ()
       :settable-instance-variables)
```

This form only defines the flavor ship; we have not yet made any instances of the ship flavor. All instances of the ship flavor will have three instance variables, x, y, and captain. The values of these instance variables will vary from instance to instance: that's why they are called instances variables.

CREATING INSTANCES

You may create an instance of the ship flavor by typing:

```
     (Setq bigship  (make-instance 'ship :x 300 :y
     400 :captain 'hook))
```

The printed representation will look something like:

```
#<ship 604824>
```

You do not have to initialize the instance variables, but if you don't they will be left unbound. Defaults may be added by placing an extra set of parenthesis around the instance variables. For example:

```
     (defflavor ship
          ((x 200)
           (captain 'hook))
          ()
       :settable-instance-variables)
```

This defines the default initialization for which is evaluated in the global environment at make-instance time.

METHODS

When we compile the DEFFLAVOR for ship, we not only get a template for creating instances of a ship, but we also get several methods. Methods may take arguments, perform some section and return a result. When ship is compiled, six new methods appear, :x, :y, :captain, :set-x, :set-y and :set-captain. The set methods allow the user to change the values of the instance variables and are created because of the :settable-instance-variables option used in the DEFFLAVOR form.

A method is run when a user sends a message to an instance. Send is a function. It takes one of these forms:

```
     (send instance :message)

     (send instance :message arg1 ...  argn)
```

WRITING YOUR OWN METHODS

The following examples show you how to use the DEFMETHOD special operator to write your own methods.

```
     (defmethod (ship :draw) ()
       (send terminals-io :draw-rectangle 40 20 (-
      x 20) (- y 10) tv:alu-xor)
```

The first operand to DEFMETHOD is a list containing the flavor name and the message name.

The second operand is a parameter list. This methods has no parameters but, as with defun, the () are still required.

We reference X and Y, the instance variables, without sending for them. This is because a method runs inside an instance, and all the instance variables are available as local variables of the method.

To run this method type:

```
     (send bigship :draw)
```

For more information about Zetalisp see the Programming Primer section 5.

GUIDE TO TI DOCUMENTATION

The first time user should get familiar with the Operations Guide and Programming Primer. Try some of the examples in the Primer.

Here is a list of all the TI documentation.

Lisp Reference
Window System Reference
System Software Release Information
System Software Installation Guide
Technical Summary
Glossary
System Software Design Notes
Programming Primer
Programming Concepts And Tools
Zmacs Editor Reference
Zmacs Editor Tutorial
Comm User'S Guide
Tcp/Ip User'S Guide
Other Comm Guides
Operations Guide
Cit User's Guide
Master Index

Happy computing!!

DL.