Conceptual Views of EMACS's Architecture

Chris Mennie - 97024327
camennie@uwaterloo.ca

Troy Gonsalves - 97083748
tgonsalv@uwaterloo.ca

January 30th 2003

1.0 Abstract **2**
2.0 Introduction **2**
2.1 Document Conventions 2
2.2 Overview 3
3.0 Conceptual Architecture **3**
3.1 High-level View 3
3.2 Layered View 4
3.3 Component View 5
3.4 Edge Descriptions 5
4.0 Subsystems **6**
4.1 Command Dispatcher 6
4.1.1 Command Interpreter 6
4.1.2 Dispatch Table 6
4.2 LISP 7
4.2.1 LISP Interpreter 7
4.2.2 Loaded Libraries 7
4.3 Display Processor 7
4.4 Primitives 8
4.4.1 Text Manipulation Primitives 8
4.4.2 I/O Primitives 8
4.5 Buffer 8
4.5.1 Display Buffer 8
4.5.2 Internal Sub-Editor 8
4.5.3 Internal Buffer 9
4.6 Operating System (OS) 9
4.6.1 Graphical User Interface (GUI) 9
4.6.2 File I/O 9
4.6.3 Socket I/O 9
5.0 Validation Scenarios **9**
5.1 Loading a LISP Script 9
5.2 Invoking a LISP Method 11
6.0 Extensibility **13**
7.0 Conclusion **13**
8.0 Data Dictionary **13**
9.0 References **13**

1.0 Abstract

EMACS is a powerful text editor with a LISP interpreter at its core. Often imitated since its creation in 1976, EMACS has proven itself to be one of the most useful programming editors available.

Little in the way of developer documentation exists for EMACS, with most of the code being written in an ad-hoc fashion. We propose a layered architecture for EMACS and provide some scenarios to validate it.

2.0 Introduction

The Hacker's Dictionary defines EMACS as:

> *EMACS /ee'maks/ /n./*
> *[from Editing MACroS] The ne plus ultra of hacker editors, a programmable text editor with an entire LISP system inside it. It was originally written by Richard Stallman in TECO under ITS at the MIT AI lab; AI Memo 554 described it as "an advanced, self-documenting, customizable, extensible real-time display editor". It has since been reimplemented any number of times, by various hackers, and versions exist that run under most major operating systems. Perhaps the most widely used version, also written by Stallman and now called " GNU EMACS" or GNUMACS, runs principally under Unix. It includes facilities to run compilation subprocesses and send and receive mail; many hackers spend up to 80% of their tube time inside it. Other variants include GOSMACS, CCA EMACS, UniPress EMACS, Montgomery EMACS, jove, epsilon, and MicroEMACS.*
>
> *Some EMACS versions running under window managers iconify as an overflowing kitchen sink, perhaps to suggest the one feature the editor does not (yet) include. Indeed, some hackers find EMACS too heavyweight and baroque for their taste, and expand the name as `Escape Meta Alt Control Shift' to spoof its heavy reliance on keystrokes decorated with bucky bits. Other spoof expansions include `Eight Megabytes And Constantly Swapping', `Eventually malloc()s All Computer Storage', and `EMACS Makes A Computer Slow'.*

Started in 1976 as an extension to TECO, it has grown and become one of the most popular programming editors in use. Using LISP at its core EMACS has enjoyed great customizability, with hundreds of extensions readily available.

The design philosophy behind EMACS has been that of hackers. There is no intentional design, and if anything, there is an intentional reluctance to make or follow any. With the lack of documentation and convoluted code it makes it difficult for new developers to contribute. However, from what little documentation there is it is possible to discern the overall architecture for EMACS.

This document proposes a layered architecture and give a layered and component based view of the system. All the subcomponents are detailed, with some scenarios given to validate the proposed architecture.

2.1 Document Conventions

To avoid crisscrossing lines in our diagrams, the "multiplexer" notation was introduced. The meaning of the multiplexer symbol is that every input edge is connected to every output edge. For example, in the Overview diagram the LISP component places calls to the Display Processor and Primitives

components, as does the Command Dispatcher. Without the multiplexer the edges from the LISP and Command Dispatcher components would cross over each other.

2.2 Overview

The following section presents the proposed architectural diagrams of EMACS and briefly describes the meaning/reasoning behind each edge in the Layered and Component Views. Section 4.0 describes the subsystems depicted in these views. Section 5.0 presents two scenarios to demonstrate the operation our proposed architecture. Section 6.0 briefly describes the extensibility of EMACS. Section 7.0 provides our conclusions. Section 8.0 contains a Data Dictionary (Glossary) and Section 9.0 contains our references.

3.0 Conceptual Architecture

The architecture of the EMACS system can viewed in several ways. The following subsection presents the high-level architecture, which serves as an overview of the system. In subsections 3.2 and 3.3 we present two complementary views of the system that expand on the clustering suggested by the high-level architecture. Subsection 3.4 contains a table describing each of the edges in Figure 3.2 and Figure 3.3

3.1 High-level View

A high-level view of the system implies a layered architecture with functional clusters at each layer (see Figure 3.1). At the top layer we have the LISP interpreter and libraries (labelled *LISP* in Figure 3.1) and the *Command Dispatcher*, which work together to handle user input. These components may use the *Display Processor* and a set of *Primitive* functions to carry out commands like refreshing the screen, inserting a character, and loading a file. These commands amount to operations on the *Buffer* subsystem and/or calls to the operating system (*OS*). Hence, the bottom layer consists of the *Buffer* and *OS* components.

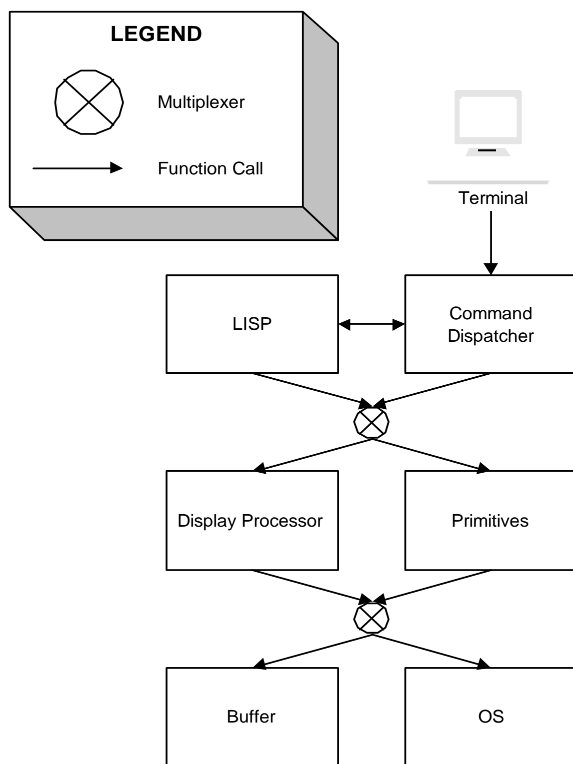

Figure 3.1 EMACS Conceptual Architecture - High-level View

3.2 Layered View

Expanding on the concept of a layered architecture we obtain the structure depicted in Figure 3.2. We use this view of the system when examining component interaction and flow of control. For example, in Section 4.6.1 we use this view of the architecture to trace the flow of control for two typical scenarios.

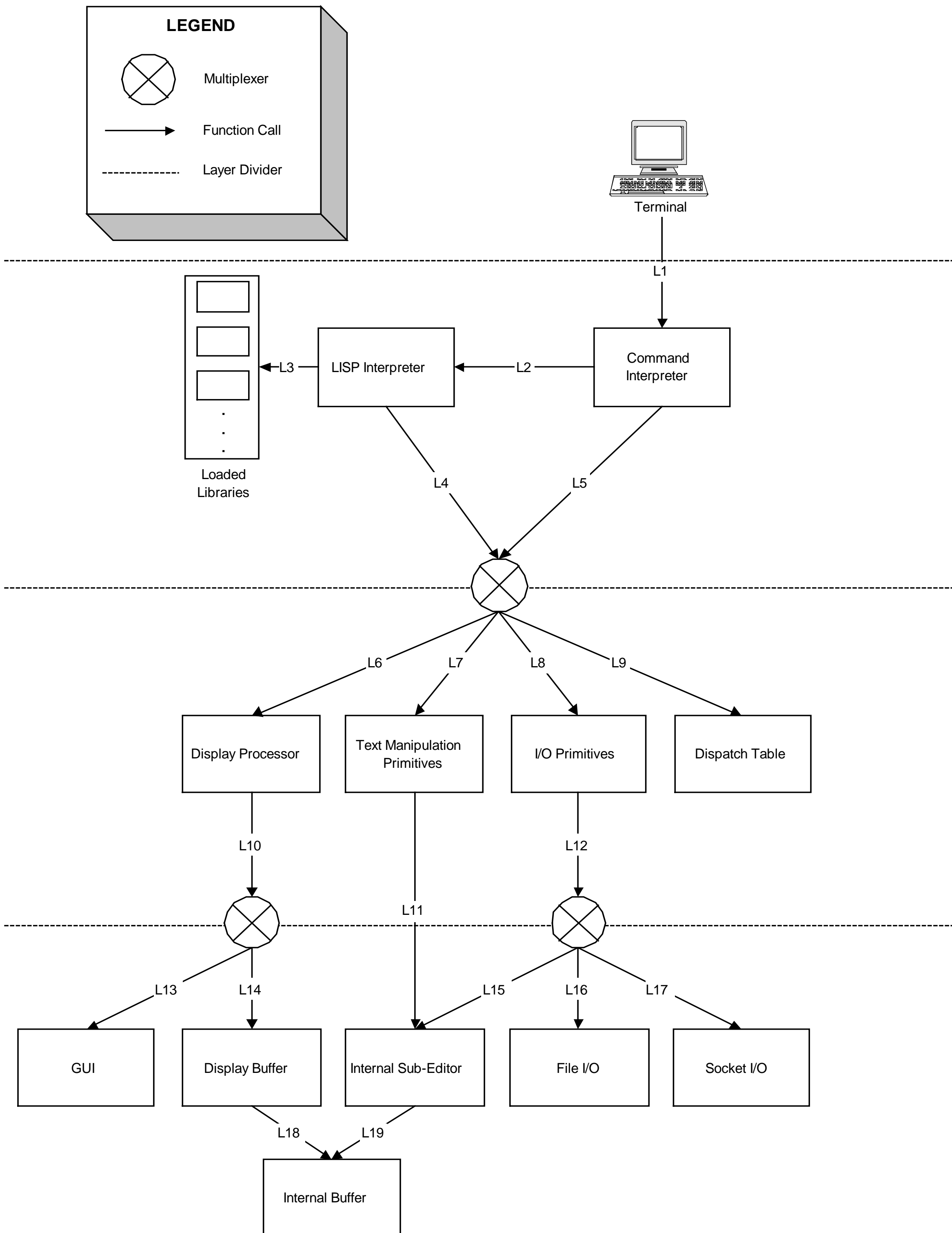

Figure 3.2 EMACS Conceptual Architecture - Layered View

3.3 Component View

Expanding on the idea of functional clustering we obtain the structure depicted in Figure 3.3. We use this view in the next section, where we focus on the function of each subsystem.

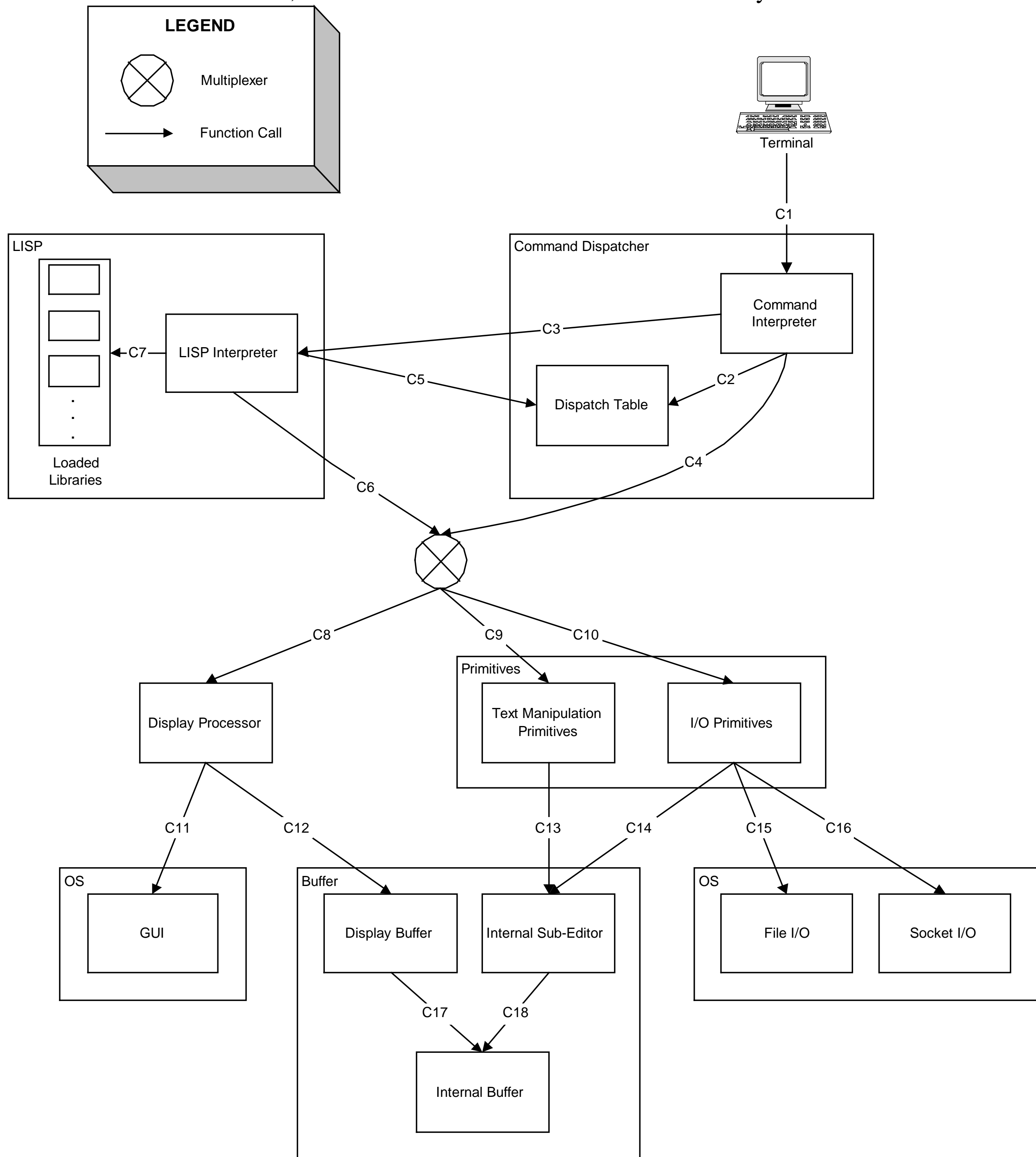

Figure 3.3 EMACS Conceptual Architecture - Component View

3.4 Edge Descriptions

Source	Destination	Component View Label	Layered Label	Purpose
Terminal	Command Interpreter	C1	L1	Terminal (User) invokes the Command Interpreter to handle input.
Command Interpreter	Dispatch Table	C2	L5 – L9	Command Interpreter uses key sequences (taken from user input) to look up bound functions in the Dispatch Table.
Command Interpreter	LISP Interpreter	C3	L2	Command Interpreter invokes LISP Interpreter to execute LISP scripts.
Command Interpreter	Display Processor	C4 – C8	L5 – L6	Command Interpreter suspends the Display Processor while it processes input events. Command Interpreter also sends buffer update information to the Display Processor.
Command Interpreter	I/O Primitives	C4 – C10	L5 – L8	Command Interpreter may invoke an I/O Primitive to carry out a command (e.g. *Save*).
Command Interpreter	Text Manipulation Primitives	C4 – C9	L5-L7	Command Interpreter may invoke a Text Manipulation Primitive to carry out a command (e.g *Insert Character*).
LISP Interpreter	Dispatch Table	C5	L4 – L9	The LISP Interpreter may access and/or modify the command bindings stored in the Dispatch Table as the result of executing a LISP script.
LISP Interpreter	Loaded Libraries	C7	L3	In response to a *load library* command the LISP Interpreter loads the library and stores it in Loaded Libraries. Also, when a LISP function is called the LISP Interpreter searches Loaded Libraries to find the definition of the function.
LISP Interpreter	Display Processor	C6 – C8	L4 – L6	The LISP Interpreter may request an explicit refresh from the Display Processor.
LISP Interpreter	I/O Primitives	C6 – C10	L4 – L8	As a result of executing a LISP script, the LISP Interpreter may call I/O Primitives to access OS level I/O functions.
LISP Interpreter	Text Manipulation Primitives	C6 – C9	L4 – L7	As a result of executing a LISP script, the LISP Interpreter may call Text Manipulation Primitives in order to modify text in the Internal Buffer (though the Internal Sub-Editor).
Display Processor	GUI	C11	L10 – L13	The Display Processor uses the operating system's GUI manipulation functions for refresh the screen.
Display Processor	Display Buffer	C12	L10 – L14	The Display Processor uses the Display Buffer to verify that the text displayed on the screen matches the text in the Internal Buffer.
Text Manipulation Primitives	Internal Sub-Editor	C13	L11	Text Manipulation Primitives uses the Internal Sub-Editor to access and modify the text in the Internal Buffer.
I/O Primitives	Internal Sub-Editor	C14	L12 – L15	I/O Primitives may use the Internal Sub-Editor to read or write to the Internal Buffer as a whole. For example to load a file into the buffer or to save the contents of the buffer to a file.
I/O Primitives	File I/O	C15	L12 – L16	I/O Primitives uses File I/O to interface with the file system.
I/O Primitives	Socket I/O	C16	L12 – L17	I/O Primitives uses Socket I/O to access the networking facilities of the OS.
Display Buffer	Internal Buffer	C17	L18	The Display Buffer must be able to read from the Internal Buffer in order to compare its contents to the text displayed on the screen.
Internal Sub-Editor	Internal Buffer	C18	L19	The Internal Sub-Editor must be able to read from and write to the Internal Buffer.

4.0 Subsystems

The following subsections describe the subsystems depicted in the architecture diagrams shown in the previous section.

4.1 Command Dispatcher

The Command Dispatcher consists of two subcomponents: the Command Interpreter and the Dispatch Table. The Command Interpreter is responsible for reading input from the terminal and deciding how handle that input. Part of this decision involves examining the Dispatch Table, which maps key events to LISP functions.

4.1.1 Command Interpreter

The first thing the Command Interpreter does after receiving input is suspend the Display Processor. This is done because the current input may invalidate the screen update being preformed by the Display Processor. Once the Display Processor has been suspended the Command Interpreter examines the input and decides how to handle it with the actual handling being implemented in other components.

The first step in deciding how the handle a specific input is to determine the type of event. If the event is a mouse event then a LISP script is called to handle the event. If the event is not a mouse event then it must be a key-press event, and the Command Interpreter must determine which command to invoke as a result. This is done by performing a lookup in the Dispatch Table to determine if the key sequence implied by the key event is bound to a command. If no binding exists then the input is a normal key press and is handled by calling one of the text manipulation primitives (e.g. insert character). If a key binding does exist then the event is treated as a command and the bound method is executed. If the bound method is a LISP script then it is executed via the LISP Interpreter. Otherwise, the function is primitive (i.e. a member of the Primitives cluster), and is handled using a standard C call.

Functions that alter the contents of the Internal Buffer will return information that specifies what part(s) of the buffer were modified. This information is passed along to the Display Processor. After passing this information the Command Interpreter looks for another input event. If an event is found immediately then it is handled as described above, with one exception: the Display Processor does not need to be suspended again. Once all input events are processed the Command Interpreter becomes dormant and the Display Processor proceeds.

4.1.2 Dispatch Table

The Dispatch Table stores a mapping from key sequences to command. For example, "<Ctrl>+_" is mapped to the Undo command. The command may be LISP scripts or primitive functions (i.e. members of the Primitives cluster).

The Command Interpreter uses the Dispatch Table to determine if a given key sequence is bound to a command. If the sequence is bound then the corresponding command is returned.

The Dispatch Table can be accessed and modified using LISP scripts, which are processed by the LISP Interpreter.

4.2 LISP

At the heart of EMACS is a LISP interpreter through which most of the interesting editor functionality is performed. At runtime, calls to LISP functions/scripts are interpreted and invoked by the LISP interpreter. EMACS's highly customizable nature is due, almost exclusively, to its reliance on the LISP subsystem.

4.2.1 LISP Interpreter

The LISP Interpreter facilitates the overwhelming extensibility of the EMACS system. Most features in EMACS are implemented using LISP scripts. These scripts can be interpreted at runtime, as opposed to being compiled into the system. This allows for a greatly extensible system as new functionality can be added dynamically (i.e. at runtime). Even the default behaviour of EMACS can be overridden and customized. As an example of the extensibility of EMACS and the power of the LISP interface consider the fact that EMACS can be turned into a Tetris game.

The LISP Interpreter stores key bindings in the Dispatch Table. The Command Interpreter then uses the Dispatch Table to retrieve this binding information and invoke LISP methods. This, in turn, allows libraries to define key bindings, which can be used by the user to invoke LISP scripts

Special LISP methods are needed to allow LISP to access the low-level editor functionality, which is implemented in the compiled C portion of EMACS. These special LISP methods allow the LISP Interpreter to access the Display Processor and Primitives subsystem.

4.2.2 Loaded Libraries

Loaded Libraries is technically a part of the LISP Interpreter's global environment. This is where all the LISP function definitions are stored. Since EMACS's extensibility is built around the "plug-in" nature of LISP scripts, Loaded Libraries can be seen as a collection of feature packages implemented in LISP. Some examples of the features implemented by these packages are:

- Language-specific environments: syntax highlighting, automatic indentation
- The help system
- The shell interface
- Web browsing
- Email
- The EMACS tags system
- File differencing
- A variety of games

4.3 Display Processor

The Display Processor is responsible for updating the screen to reflect the contents of the Internal Buffer. To do this the Display Processor uses the Display Buffer to validate the text being displayed. This validation is performing on a line-by-line basis. If a line is valid then the Display Processor moves on to the next line, looping to the start of the Buffer after the last line. When a line is invalid the Display Buffer is updated accordingly. The Display Processor then uses the updated line, and one or more calls to the GUI component of the operating system, to update the screen. If the Display Processor validates all of the lines being displayed then it waits for the Command Interpreter or the LISP Interpreter to send notice of an update or to request an explicit refresh.

The Display Processor runs only when there is nothing else to do. That is, the Command Interpreter suspends the Display Processor when there are input events to be handled. The Display Processor resumes only when there are no input events waiting to be processed (i.e. when the Command Interpreter becomes dormant). This approach is atypical and has been adopted for the sake of performance. The performance gain comes from the fact that many unnecessary redraw commands are eliminated. For example, if a user is entering a line of text, then a typical editor will refresh the screens after each character is released. EMACS on the other hand will only refresh the screen when the user stops typing for long enough to allow the Display Processor to refresh the line.

4.4 Primitives

The Primitives component contains the low-level functions used to manipulate text and access the basic I/O functions of the operating system.

4.4.1 Text Manipulation Primitives

This subsystem allows the user to manipulate text in the buffer. For example, text searching or replacement would be dealt with here. Updating the buffer is done indirectly through the Internal Sub-Editor.

4.4.2 I/O Primitives

This subsystem allows the user to perform I/O operations (e.g. load a file, save a file) and access network resources. I/O Primitives is the only subsystem that has direct access to the operating systems File I/O and Socket I/O systems. Since operations such as loading and saving a file requires read and write access to the Internal Buffer, the I/O Primitives also make use of the Internal Sub-Editor.

4.5 Buffer

Each file or unsaved document has a buffer associated with it. Rather than expose other subsystems to the raw buffer, different interfaces to the buffer are used.

4.5.1 Display Buffer

The Display Buffer is responsible for keeping track of what part of the Internal Buffer is actually displayed on the screen and where. A list of pointers into the Internal Buffer is kept and commands to query for changes are also made to the Internal Buffer.

One Display Buffer exists for each Internal Buffer.

4.5.2 Internal Sub-Editor

The purpose of the Internal Sub-Editor is to abstract the interface to the Internal Buffer. Simple operations such as reading and writing text from and to the buffer are made available through this subsystem. The position of the editing cursor is also stored in this subsystem.

One Internal Sub-Editor exists for each Internal Buffer.

4.5.3 Internal Buffer

The Internal Buffer stores all textual information that pertains to a given document. This including internal representations of the text such as control codes. Operations to modify the document are performed through this subsystem. Each open file and unsaved document has its own Internal Buffer.

4.6 Operating System (OS)

The OS component represents the low level operating system interface that is external to EMACS. While most of EMACS's functionality is handled in LISP and the internal primitives, at some point the operating system must be used to perform fundamental operations such as updating the screen, etc.

4.6.1 Graphical User Interface (GUI)

The GUI subcomponent represents interface to the display being used. This may be a terminal interface, or a more modern windowed interface.

4.6.2 File I/O

The File I/O subcomponent represents the interface to the actual low-level file system. Operations such as actually reading and writing a file are performed through here.

4.6.3 Socket I/O

Access to the network interface by means of sockets is done using the Socket I/O.

5.0 Validation Scenarios

To demonstrate the operating of our proposed conceptual architecture the section examines a few scenarios. Each scenario is presented using a collaboration diagram and a table describing the control flow through the system. The following two scenarios are presented:

- Loading a LISP script
- Invoking a LISP method

5.1 Loading a LISP Script

Load a LISP script from a file and add new bindings as needed. The script is assumed to not assign any key bindings.

Step	Description
1	User invokes the script loading starting with the Command Interpreter.
2	Command Interpreter tells the LISP Interpreter to read a script.
3	LISP Interpreter uses the I/O Primitives to load file.
4	I/O Primitives repeatedly uses whatever OS methods necessary to read in file.
5	OS finishes reading in file.
6	LISP Interpreter is told file is completely read in, then LISP Interpreter executes script.
7	Any function or variable declarations (i.e. LISP bindings) made by the script are added to the Loaded Libraries.
*	If key bindings had been set, then the LISP Interpreter would have updated the Dispatcher Table during step 7.

Table 5.1 Description of Steps Depicted in Figure 5.1

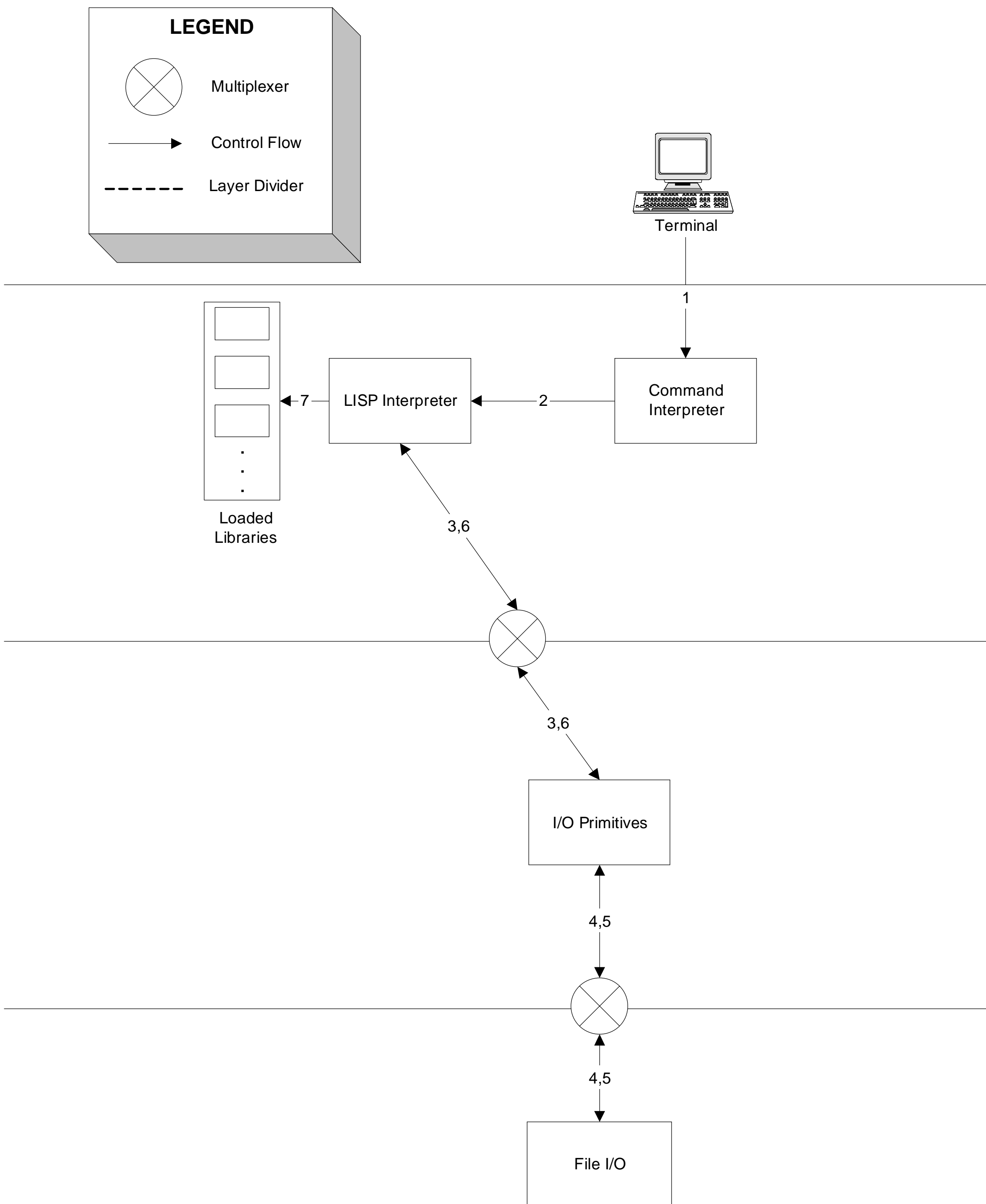

Figure 5.1 Loading a LISP Script

5.2 Invoking a LISP Method

Assuming that the *Paste* command is bound to "<Ctrl>-v". Also assume that the clipboard is non-empty at the start of this scenario.

Step	Description
1	Terminal (User) invokes the Command Interpreter to handle the input "<ctrl>+v"
2	Suspend Display Refreshing
3	Use "<ctrl>+p" to look up the paste function
4	Invoke LISP Interpreter to execute the paste function.
5	Search Loaded Libraries to find the definition of the paste function.
6	The paste script calls a Text Manipulation Primitive to modify text in the Internal Buffer
7	Use the Internal Sub-Editor to insert text in the Internal Buffer.
8	Insert text into the Internal Buffer.
9	Send the Display Processor the information returned by the function call (i.e. the range of the buffer that was updated).
*	Command Interpreter becomes dormant and Display Processor Activates
10	Use the Display Buffer to verify that the text displayed on the screen does not match the text in the Internal Buffer.
11	Retrieve the new text from the buffer
12	Uses the operating system's GUI manipulation functions to refresh the screen.
*	Steps 10-12 are repeated for each line of text that was modified

Table 5.2 Description of steps depicted in Figure 5.2

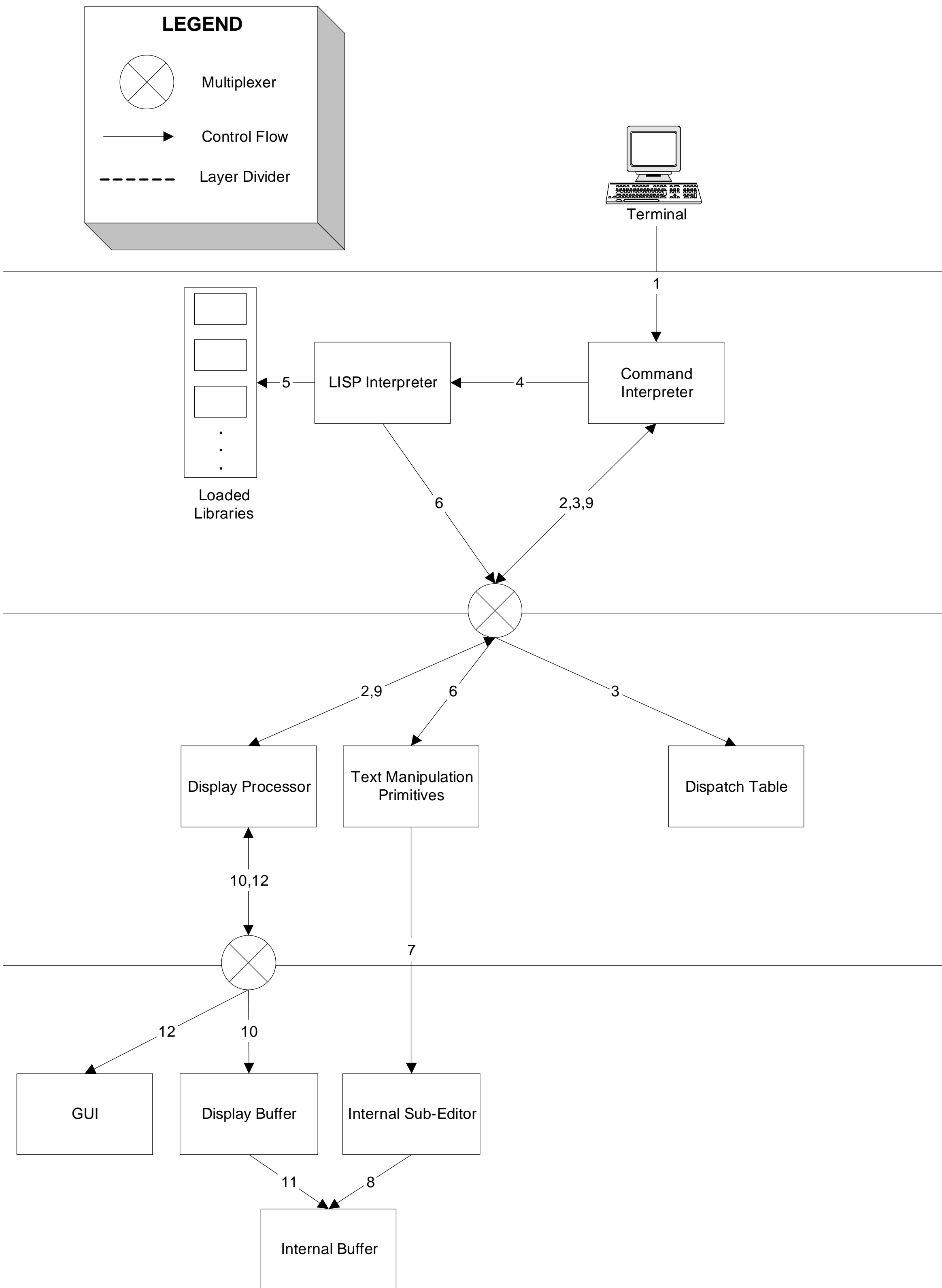

Figure 5.2 Invoking a LISP Method

6.0 Extensibility

EMACS's greatest strength is the ease in which is may be customized. Through the LISP interface and its integration with a text editor, virtually limitless customization is possible. Complex text manipulation, such as encrypting/decrypting text, can be performed. The degree of customization possible allows for a user to completely alter the behaviour of the editor to do things such as play Tetris or a text adventure.

The only limiting factor is LISP itself and the interfaces provided to it. Only so much performance can be expected from interpreted LISP. Likewise only so much can be expected from the display in which EMACS is running. Modern GUI environments such as gtk are available, but the interfaces to them are still rather primitive.

7.0 Conclusion

EMACS is a greatly powerful and highly configurable editor. Overall it has a layered architecture with a plugin-style library providing most of EMACS's extensibility. Fundamentally, EMACS is made up of two major components: a LISP interpreter and a simple text editor. By hooking these two components together EMACS is able to facilitate a large variety of functions. From text editing to web browsing to games the possibilities are limitless.

The proposed component and layered views give a clear picture of the detailed architecture in EMACS. Together with the validation scenarios we show that the proposed architecture is reasonable.

8.0 Data Dictionary

Term	*Definition*
LISP	[from `LISt Processing language', but mythically from `Lots of Irritating Superfluous Parentheses'] AI's mother tongue, a language based on the ideas of (a) variable-length lists and trees as fundamental data types, and (b) the interpretation of code as data and vice-versa. Invented by John McCarthy at MIT in the late 1950s, it is actually older than any other HLL still in use except FORTRAN. Accordingly, it has undergone considerable adaptive radiation over the years; modern variants are quite different in detail from the original LISP 1.5. The dominant HLL among hackers until the early 1980s, LISP now shares the throne with C.
GUI	Graphical User Interface
Tetris	An annoyingly addictive game wherein the user tries to find a solution to an NP-C problem involving falling blocks.
Text Adventure I/O	Computer games popular in the 1980s which display all output as text. Shorthand for Input and Output

9.0 References

[EMACS] EMACS: The Extensible, Customizable Display Editor, **http://www.gnu.org/software/emacs/emacs-paper.html**

[CRAFT] The Craft of Text Editing, **http://www.finseth.com/~fin/craft/**

[HACK] The Hacker's Dictionary, **http://www.jargon.8hz.com**