Engineering a Robot using CLISP Environment

Tzewen Wang
Department of Electrical and Computer Engineering
Portland State University
tzewen@cecs.pdx.edu

Abstract

In this paper, a LISP programming environment for robots is presented. LISP is a high-level programming language that has been widely used in many artificial intelligence communities. The problems in artificial intelligence are mostly searching problems. The nature of LISP's data structure allows programmers to concentrate on development of search algorithms. The GNU LISP environment – CLISP – has provided many packages extensively. FFI package – Foreign Function Interface package – is possibly the most powerful package among those. It provides a set of primitives to pass program control flow between the functions inside and outside of the LISP environment back and forth. A program written in LISP is then able to invoke system calls and communicate with hardware abstraction layer through FFI package. A humanoid robot successfully controlled by a desktop computer that uses the environment is also demonstrated.

1 Introduction

Most problems in the area of artificial intelligence are likely to be searching problems. For instance, machine learning is a method to change a function or a knowledge base internally. An evaluation function evaluates correctness under certain context from the output of the function and assigns a Boolean value. This value is used to determine the action of updating the function or the knowledge base. As time goes, the correctness of output should be improved. The curve is known as the learning curve. The function itself may be doing a pattern-matching on the input with the knowledge base, voice recognition, for example. Pattern-matching is no more than a search problem. Decision diagrams are widely used in dealing with these problems and they are often presented in form of tree.

```
(* 1 2 (+ 3 4) 5)
```

(a) An algebraic expression in prefix form. This is equivalent of 1 * 2 * (3 + 4) * 5)

```
(first `(a b c))
```

(b) Calling function `first` with argument `` `(a b c) ``.

```
(first second third)
```

(c) A list of data (atoms).

LISP is functional language invented by John McCarthy in 1958 [1]. LISP is capable to do any computation like C, Java, and many other languages. The heart of LISP is driven by its data structure – List. A list contains a number of elements. An element can be either an Atom or another List. The List is especially useful for expressing a tree data structure in a plain, human readable expression. A List is not only a piece of data but also a function call. The very first element in a List is the name of a function and rest of the elements are simply the arguments for the function. The result of the function can be an Atom or a List. This allows LISP to have uniform syntax – operations are described within a pair of parenthesis. Figure 1 shows some examples of computation expressions in LISP. Another implication of List data structure is easy to do recursive computation. For instance, an element in a List may be another List. A LISP program with a few lines of code

may be sufficient to traverse the List entirely by using recursion. Figure 2 shows a code snippet that walks through and clones a tree-like list. As a consequence, LISP is also a heavy recursive language. In short, program written in LISP are concise and easy to understand.

```
(defun clone (x)
  (cond ((atom x) x)
        (T (cons (clone (first x)) (clone (rest x))))))
```

Figure 2: A LISP code snippet that clones a tree recursively.

Since LISP is a high-level language, an environment is needed to interpret LISP code. In some sense, there must be some kind of just-in time compiler and virtual machine that compiles and executes the code. This places LISP in the category of platform independent language. Theoretically, a piece of LISP code should be able to execute and produce same results in the same LISP environment under any platform because LISP functions are not related to any machine architecture whatsoever. This is another good reason to implement programs in LISP.

There are several LISP environments available in both commercial and public domain. In commercial domain, there are Allegro CL, LispWorks, and Digitool. In public domain, there are CMUCL, CLISP, ILISP and OpenMCL. Commercial LISP environment often comes with powerful package such as XML parser and some hardware control routines but they are expensive and not affordable for academic research. On the other hand, publicly available LISP environment often lacks maintenance and key routines for crafting a real robot control program.

GNU CLISP implementation [2] by Bruno Haiblo is a public-domain ANSI Common LISP environment. The environment is bound to GNU GPL license. It requires only 2MB ram to run and provides many useful packages. More importantly, this environment has been already ported to UNIX, Linux, and Windows platforms. Precompiled binary executable can also be found at [2]. This results in a very inexpensive way to get a LISP programming environment to programming a robot using LISP. In this paper, we will focus on the most important package among all – Foreign Function Interface.

Rest of the paper is structured as follows: Section 2 briefly discusses FFI package. Section 3 presents programming model of a general robot program using CLISP and examples. Section 4 concludes the paper and section 5 shows the source code of a simple humanoid robot control program.

2 Foreign Function Interface – FFI package

2-1 What is FFI package?

FFI stands for Foreign Function Interface. A foreign function refers to a function that is not physically in CLISP environment. In some sense, this interface acts as a gateway between functions inside and outside CLISP environment. FFI consists of a set of LISP primitives that are necessary to convert LISP data types to foreign data types such as C data structures. The primitives are also used to define presences of foreign functions in CLISP environment as symbols. All FFI functions are stored in a LISP package called FFI.

2-2 Why FFI package is important?

As computer industry grows, more and more computer languages are being created to solve problems. The problems themselves may be all the same but the formulations can be quite different. A language then is created to solve problems in particular models. For instance, Prolog is a predicate language designed

specifically for theorem proving. C/C++ is a general purpose language designed to interfacing general calculations. Matlab is a language designed specifically doing matrix computation. As a consequence, each computer language has its own strength and weakness. Choosing an appropriate language for a particular problem is critical. Choosing an inappropriate language for a computation model may result in longer development cycle. Most programs today are written using several languages. For example, at operating system level, C is often used. Creating a graphic user interface, Visual Basic is certain a top choice. Having an internally collaborated program composed by different languages becomes extremely important. In fact, the mechanisms to link functions together have been out there for a long time. A commonly used mechanism is called dynamic linked library. Functions can be compiled and stored in a file that will be linked and called at the runtime of any program. Certainly, the caller also has to have the capability to access and interfacing with the library. As a consequence, programs implemented using different favorable languages are able to work collaboratively.

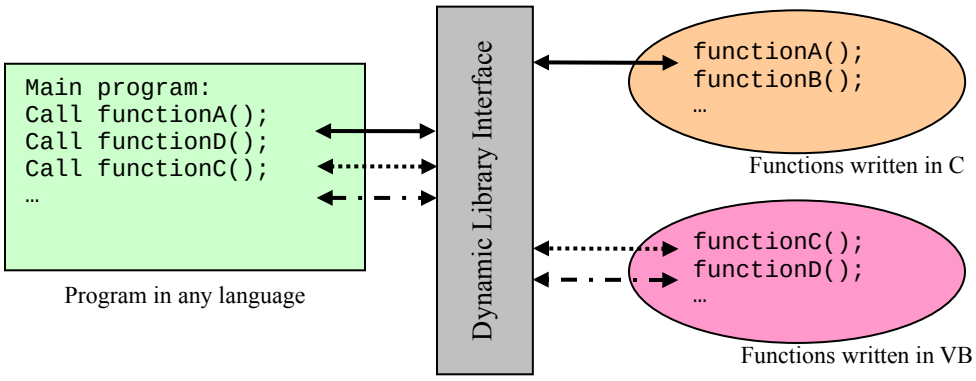


Figure 3: An illustration of a complex program.

Figure 3 shows a general picture of a program that uses a variety of function written in different languages linked by dynamic library interface at runtime. The left box is a conceptual main program which can be a control loop. The two ellipses on the right represent two sets of assistive functions written in different languages. The middle box is a binder that binds the functions between left and right side. This is usually handled at operating system level. The libraries must be stored in a certain format of files specified by OS platform. For examples, Windows recognize files with extension `.dll` as library files. Linux and UNIX recognize files with extension `.so` as library files.

If we apply this model to CLISP environment, a LISP program would be on the left hand side and library packages such as OpenCV [3] and Winsocket 2 [4] would be on the right hand side. CLISP FFI package would be in between the left box and the middle box.

2-3 FFI Primitives

As mentioned earlier, the package comes with symbols, macros, and functions to interfacing with foreign functions. Current package implementation only supports C type libraries. So, we will only focused on issues between C and LISP types. The package can be classified in four categories: symbol declaration, type declarator, variable instantiation and operations.

2-3-1 Symbol Declaration

In a LISP environment, symbols are in contrast to symbols written in C. All LISP symbols are case insensitive. This is an immediate problem for calling the function outside LISP environment. Thus, we need to use some FFI primitives to define symbols representing foreign functions. A foreign function symbols has to consist of the following information: foreign function type, function name in string type, language type, argument types, and function return type. Recall that control flow can be passed back and forth between native and foreign functions. As a consequence, we need to first specify what type of foreign function is. The type is either a call-out function or call-in function. A call-out function passes control flow from LISP to foreign functions. A call-in function passes control flow from foreign to LISP functions. We use `def-call-out` and `def-call-in` to declare call-out and call-in functions respectively. Options are NAME, ARGUMENTS, RETURN-TYPE, LANGUAGE, and LIBRARY. Figure 4 shows synopsis of `def-call-out` and `def-call-in` declarations.

```
(def-call-out  LISP-NAME
  :LANGUAGE    {:C | :STDC | :STDC-CALL}
  :NAME        C-FUNC-NAME
  :LIBRARY     LIB-PATH
  :ARGUMENTS   ((ARG-1 C-TYPE-1 [PARAM MODE] [ALLOC])
                (ARG-2 C-TYPE-2 [PARAM MODE] [ALLOC])
                ...)
  :RETURN-TYPE C-TYPE [ALLOC])
```

(a) Synopsis for def-call-out

```
(def-call-in   LISP-NAME
  :LANGUAGE    {:C | :STDC | :STDC-CALL}
  :NAME        C-FUNC-NAME
  :ARGUMENTS   ((ARG-1 C-TYPE-1 [PARAM MODE] [ALLOC])
                (ARG-2 C-TYPE-2 [PARAM MODE] [ALLOC])
                ...)
  :RETURN-TYPE C-TYPE [ALLOC])
```

(b) Synopsis for def-call-in

Figure 4: Syntax of foreign function declaration. Literals with italic need to replaced by appropriate strings or symbols. For example, *LISP-NAME* should be replaced by a syntactic LISP symbol. *C-FUNC-NAME* should be replaced by a string.

Now, we give a brief explanation for the each option. Option LANGUAGE is used to describe what calling convention is used for that particular foreign function. Calling methods slightly differ from language compilers. ":C" denotes K&R C, ":STDC" denotes ANSI C, and "STDC-STDCALL" denotes ANSI C with stdcall calling convention. Option NAME simply describes the name of the foreign function with a double quoted string. For instance, "SHBrowseForFolder", a shell function in Win32 API. Option LIBRARY is used to specify the location of the foreign function. It must be a valid path to a DLL file containing the function, "C:\Windows\System32\Shell32.dll" or "/lib/libmp.so", for example. The path type depends on platform on which Clisp runs. Option ARGUMENT specifies a list of arguments for the function. Each argument is list with a LISP symbol as the first element and c-type symbol as second element. A c-type can be either one of C basic types which are predefined in FFI package or a user-defined LISP c-type symbol. Please refer to section 2-3-2 for details of predefined c-type symbols. Note that each argument has an optional specifier to indicate storage allocation type and an optional specifier to indicate access mode. Access mode is either ":IN" or ":OUT" or ":INOUT" for read-only, write-only, and read-write, respectively. Storage allocation type is extremely important when passing a structure from LISP to a foreign function. It is possible that the storage is allocated in the LISP and freed by the foreign function or vice versa, especially for the case of a pointer to a structure. Failed to maintain storage integrity could result in unexpected errors such as application crash. The options are ":MALLOC-FREE", ":ALLOCA", and ":NONE". The first option is that storage is allocated in the

LISP and freed by foreign function. Second option is that storage will be allocated in the LISP prior to the call and be freed as soon as the call returns. Option ":NONE" does not allocate any storage. Lastly, option RETURN-TYPE is used to describe returning type of the foreign function. This must be either a predefined or user-defined c-type symbol.

2-3-2 Type Declerator

A type declarator refers to a LISP c-type symbol. The C language is a strong typed language which requires exact type information on variables at compile time and disregard type checking during runtime. In CLISP, programmers can also define their own c-type symbol in additional to predefined c-type such as structure, union, and enumeration. Table 1 is a summary of c-type symbols and their corresponding C equivalent data types.

LISP Symbol	C Type Equivalent	LISP Type Eqivalent	Comment
`NIL`	`void`	`NIL`	As a result type only
`BOOLEAN`	`int`	`BOOLEAN`	
`CHARACTER`	`char`	`CHARACTER`	
`char`	`signed char`	`INTEGER`	
`uchar`	`unsigned char`	`INTEGER`	
`short`	`short`	`INTEGER`	
`ushort`	`unsigned short`	`INTEGER`	
`int`	`int`	`INTEGER`	
`uint`	`unsigned int`	`INTEGER`	
`long`	`long`	`INTEGER`	
`ulong`	`unsigned long`	`INTEGER`	
`uint8`	`uint8`	`(UNSIGNED-BYTE 8)`	
`sint8`	`sint8`	`(SIGNED-BYTE 8)`	
`uint16`	`uint16`	`(UNSIGNED-BYTE 16)`	
`sint16`	`sint16`	`(SIGNED-BYTE 16)`	
`uint32`	`uint32`	`(UNSIGNED-BYTE 32)`	
`sint32`	`sint32`	`(SIGNED-BYTE 32)`	
`uint64`	`uint64`	`(UNSIGNED-BYTE 64)`	Does not work on all platforms
`sint64`	`sint64`	`(SIGNED-BYTE 64)`	Does not work on all platforms
`SINGLE-FLOAT`	`float`	`SINGLE-FLOAT`	
`DOUBLE-FLOAT`	`double`	`DOUBLE-FLOAT`	

Table 1: C-type symbols and their C data types (Source: clisp.cons.org).

Another distinct data type is *pointer*. Pointers are widely used in C programs. FFI package also provides several c-pointer primitives. `c-ptr` is pointer wrapper used in conjunction with c-type symbols. For example, an argument or c variable declared as `(c-ptr int)` refers to *a pointer to integer*. This is equivalent to a C declaration `int*`. In many cases, variables are declared as generic pointers and casted to other types according to program behavior at runtime. A generic pointer type in C is known as `void*`. FFI package also provides such special c-type symbol known as `c-pointer`. Moreover, complex C structures such as structures and union are also supported known as `c-struct` and `c-union`, respectively. Table 2 describes special data types in summary.

C Types	LISP Primitives	Synopsis
null-terminated string	`c-string`	`c-string`
pointer (to any type)	`c-pointer`	`c-pointer`
pointer to TYPE	`c-ptr`	`(c-ptr C-TYPE)`
	`c-ptr-null`	`(c-ptr-null C-TYPE)`

array of pointer to TYPE	`c-array-ptr`	`(c-array-ptr C-TYPE)`

array of TYPE	`c-array`	`(c-array C-TYPE DIM)`
		or
		`(c-array C-TYPE (DIM1 DIM2 ...))`

structure	`c-struct`	`(c-struct LISP-NAME`
		`(ID-1 C-TYPE-1)`
		`(ID-2 C-TYPE-2)`
		`...)`

union	`c-union`	`(c-union (ID-1 C-TYPE-1)`
		`(ID-2 C-TYPE-2)`
		`...)`

function	`c-function`	`(c-function`
		`(:arguments`
		`(ARG-1 C-TYPE-1 [PARAM MODE] [ALLOC])`
		`(ARG-2 C-TYPE-2 [PARAM MODE] [ALLOC])`
		`...)`
		`(:return-type C-TYPE [ALLOC])`

Table 2: Special C-type symbols and synopsis.

All italic literals should be replaced by appropriate strings described in the previous section. Please refer section 2-3-1 for detail description about PARAM MODE and ALLOC options. In addition to specify a structure while creating a foreign variable, FFI provides `def-c-struct` allow programmers to define a new c-type symbol which is structure. The syntax for `def-c-struct` is identical to `c-struct`.

2-3-3 Variable Instantiation and Operation

All arguments or parameters to be passed to a foreign function must be allocated a special stack in CLISP environment. Thus, variables have to be created using special LISP functions prior to pass to foreign functions. FFI package provides `with-c-var` macro to create a foreign variable. The synopsis is `(with-c-var (`*`C-TYPE`* `[`*`INIT`*`]) `*`BODY`*`)` where C-TYPE is an either backquoted predefined or user-defined c-type symbol and BODY is code accessing the variables. Optional INIT is initial value for the variable.

In regular C language, there are many special operators such as & and * used to obtain a variable address and dereference a variable. FFI package provides to `c-var-address` and `deref` to incorporate with the operations. In addition, `slot` is used to access members in a structure and `cast` is used to perform type casting with a foreign variable.

2-4 Code Example

In this section, we give an example of LISP code using FFI package to make foreign function calls. We start with a simple example to open a browse window for a path using Win32 GUI API. The name of function call is called `SHBrowseForFolder` with one argument.

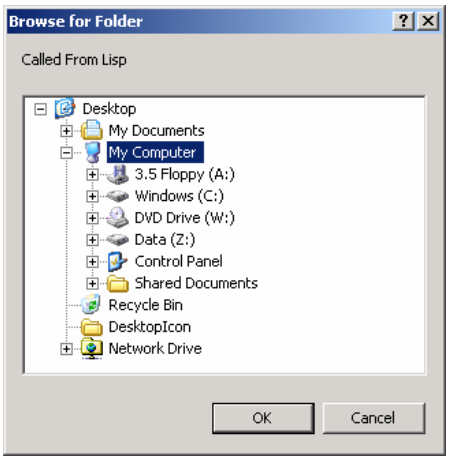
```
LPITEMIDLIST SHBrowseForFolder(LPBROWSEINFOA lpbi);

typedef struct _browseinfo {
  HWND          hwndOwner;
  LPCITEMIDLIST pidlRoot;
  LPSTR         pszDisplayName; // Return display name of item selected.
  LPCSTR        lpszTitle;      // text to go in the banner over the tree.
  UINT          ulFlags;        // Flags that control the return stuff
  BFFCALLBACK   lpfn;
  LPARAM        lParam;         // extra info that's passed back in callbacks
  int           iImage;         // output var: where to return the Image index.
} BROWSEINFO, *PBROWSEINFO, *LPBROWSEINFO;
```

```
typedef struct _ITEMIDLIST {
     SHITEMID    mkid;
 } ITEMIDLIST;

typedef struct _SHITEMID {
    USHORT      cb;             // Size of the ID (including cb itself)
    BYTE        abID[1];        // The item ID (variable length)
} SHITEMID;
```

(a) Function prototype and structures in C.



(b) A sample of the function call.

Figure 5: A Win32 GUI Function – SHBrowseForFolder.

Figure 5 shows a running snapshot and function prototype of `SHBrowseForFolder`. The function takes a structure as an argument. The structure has eight members. Return value of the function is also a structure with a substructure of two members. Thus, we need to define at least for symbols before we make the call. The four symbols are `SHBrowseForFolder`, `_browseinfo`, `_ITEMIDLIST`, and `_SHITEMID`. The first is a function symbol and the rest are c-type symbol. In this case, SHBrowseForFolder is a call-out function because it is invoked from CLISP which control flow is transferred from CLISP to the function. The three c-type symbols are created using `def-c-struct` as it is shown as following:

```
(def-c-struct LBPI_T
  (hwndOwner       INT)
  (pidlRoot        INT)
  (pszDisplayName  C-STRING)
  (lpszTitle       C-STRING)
  (ulFlags         UINT)
  (lpfn            UINT)
  (lParam          UINT)
  (iImage          INT))

(FFI:def-c-struct (ITEMIDLIST_T :typedef)
  (CB    USHORT)
  (abID  UINT8))
```

For simplification, `_ITEMDLIST` is replaced by `_SHITEMID` since `_ITEMDLIST` contains only one member. The corresponding foreign function symbol is then defined as following:

```
(FFI:def-call-out SHBROWSEFORFOLDER
             (:language    :stdc)
             (:name         "SHBrowseForFolder")
```

```
              (:library      "SHELL32.DLL")
              (:arguments   (LBPI (FFI:c-ptr LBPI_T)))
```

Note that this function is located in file SHELL32.DLL [6]. Lastly, the following LISP code is an example to invoke this call:

```
(SHBrowseForFolder (make-LBPI_T  :hwndOwner        0
                                 :pidlRoot         0
                                 :pszDisplayName   "Called From Lisp"
                                 :lpszTitle        "Called From Lisp"
                                 :ulFlags          0
                                 :lpfn             0
                                 :lParam           0
                                 :iImage           0))
```

In this example, we directly instantiate the structure right before the function call with using `make-LBPI_T` primitive. The intermediate pointer return by `make-LBPI_T` is automatically passed to the function.

2-5 More and More

Materials of FFI package covered in this section are just part of the package. This should be sufficient for most of foreign functions. Current FFI implementation only supports library functions written in C and limited storage allocation capability for different parameter passing modes. Storage allocation for ":IN" arguments are fully implemented. As for ":OUT" arguments, only ":ALLOCA" storage allocation method is implemented. For more details about the package, please refer to [6].

3 Program a Robot Using CLISP

3-1 Robot System Architecture

With today's technology, a humanoid robot is affordable. A popular, infrared remote controllable humanoid robot, Robosapien [7], is on the market for less than one hundred dollar. This robot comes with several servos driven by a microcontroller that takes a command from the IR controller. IR has physical limitations on transmitting and receiving communication signals that makes the robot's communicating to a desktop computer harder. To overcome this, radio equipment is added on top of the robot logically in parallel with the IR receiver*. The robot then can take commands either from the IR remote controller or a radio transmitter.

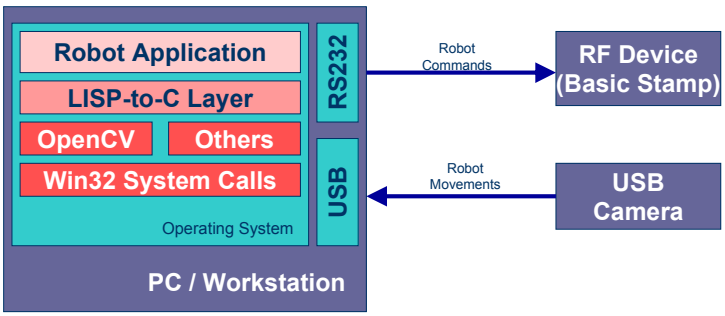


Figure 6: A robot control system implemented by a PC.

* Modification of this robot is done by Kristine Smith.

Some radio transmitters on the market are designed specifically for small systems as a peripheral in a microcontroller system [8][9]. The system is capable to communicate with other computer system through standard I/O ports such as RS232, USB, and IEEE-1394. As a consequence, desktop computers are now capable to send command signals as well as receive sensor feedback through these devices and control any robot. Furthermore, sensor feedback can also be a source off a robot. For instance, the robot can placed underneath an USB camera which sends images back to the desktop for robot command generation. This robot system involves many disciplines including AI programming, system programming, communication, micro programming, digital hardware systems, and control systems. In this paper, we will focus only on system programming because this is the key to enable robot's remote control capability.

Figure 6 shows a general system block diagram of a robot control system control on a desktop computer system. The computer sends robot commands through a radio transmitter and collects robot movement through a camera. The interface between computer and devices are RS232 and USB, respectively. The bottom layer interfacing between software applications and hardware devices is operating system. In this case, the operating system is Microsoft windows. Operating systems often come with sets of functions called Application Programming Interface (API) or system calls. As the name suggests, these are routines used for applications to access software and hardware resources. Hardware resources are hardware devices on a computer such as I/O ports, sounds cards, graphic cards, and timers. Software resources are mostly sets of routines doing particular computation. Namely, Intel OpenCV [3] is a set of image process functions written in C and stored in a form of dynamic linked library (DLL). An application can easily invoke or call the functions under platform specific mechanism and achieve computational goals without starting from scratch. The application can be written in any language as long as the language provides a way to link to the functions. FFI package introduced in section 2 is essentially the LISP-to-C layer in figure 6. This layer interfaces between LISP functions and other functions existing in any libraries including both software and hardware library. Next two sections briefly discuss how to access to a software library and a hardware device using system calls.

3-2 Access to Serial Port

As pervious section suggested, the robot is virtually connected to a robot through an I/O port. There are several I/O ports available on desktop computers today. For simplicity, we will use standard serial port for the discussion.

Each serial port under Windows system is treated as a special file with a specific name [10]. In Windows, file system has a different property from UNIX system – filename is case insensitive. Every port is assigned a number starting from zero. The number is determined by Windows COM port database [11]. All serial ports have the same prefix `COM` followed by the assigned number. For example, `COM1` is the first serial port and `COM3` is third serial port. These ports may be fragmented due to settings of hardware I/O address on ports. Namely, `COM2` may be missing while `COM1` and `COM3` are present. The latest Windows supports up to 255 serial ports. The ways accessing to ports as special files are also true for Linux and UNIX systems but different naming scheme under different file systems. We will only discuss the usage on Windows platform for rest of the paper.

Now, we know how a port can be accessed. A port can be accessed using file routines. In C, the language provides a set of file routines in `stdio` library to operate on files. This routines includes `fopen()`, `fclose()`, `fread()`, `fwrite()`, `fgetc()`, `fputc()`, and many others. On the other hand, in Common LISP, the routines are `open`, `close`, `with-open-file`, `read-byte`, `write-byte`, `read-char`, `write-char`, and others. Like C, LISP programs have to maintain file descriptors once files are opened. A file descriptor is a pointer to the physical file at operating system level and can be treated as an object. In LISP, a file descriptor is called a *stream*. Reads and writes operate on a stream object. Furthermore, `with-open-file` macro provides a convenient way to access to a file. This macro has the following form:

```
(with-open-file (LISP-NAME FILEPATH OPTIONS) BODY)
```

LISP-NAME is a local variable symbol that retains the opened file descriptor. FILEPATH is a fullpath of the file to be opened in double-quoted string. OPTIONS specifies file access mode. BODY is an implicit `progn` form. `with-open-file` closes file stream as soon as control flow leaves the body either normally or abnormally. Available options are the same as `open` function: "`:direction`", "`:element-type`", "`:if-exists`", "`:if-does-not-exists`" and "`:external-format`". Only "`:direction`" and "`:element-type`" are relevant to serial port access. "`:direction`" is either "`:input`" or "`:output`" or "`:io`". "`:element-type`" specifies the unit of transaction. "`character`" is default and "`unsigned-char`" is usually used for low-level IO operations. Ports opened with different element type have to use different read and write functions. For instance, a port's element type with character uses `read-char` and `write-char`. A port's element type with unsigned-char uses `read-byte` and `write-byte`.

```
fd = fopen("com1", "w+");  // Open as write-only.
fwrite("hello", 5, 1, fd); // Write.
fclose(fd);                // Close.
```

(a) Send string "hello" to COM1 (C code).

```
with-open-file (com1 "COM1" :direction output)
  (write-string "hello" com1))
```

(b) Send string "hello" to COM1 (LISP code).

```
(with-open-file (com1 "COM1"
                 :direction output
                 :element-type `unsigned-char)
  (write-byte 65 com1))
```

(c) Send byte value 65 to COM1 (LISP code).

Figure 7: Examples of sending data to serial port COM1.

Figure 7 shows two examples of sending data to serial port COM1. Figure 7a is a simple C code that sends a string "hello" to COM1. Line1 opens the port as a file. Line 2 writes the string and line 3 closes the port. Figure 7b is equivalent LISP code of the C code. Figure 7c is a LISP code snippet to send a byte value to serial port. Note that element type is specified in 7c and write-byte is used.

This approach is the simplest and most convenient to write data to serial ports. However, there is no way to set up hardware-specific settings such as baud rate, stop bit, and data length when open ports with built-in `open` routine. Therefore, we need to open and set up port parameters through system calls. `CreateFileW()` is a Win32 system call that opens a port and return a file descriptor. `GetCommState()` and `SetCommState()` are the system calls for obtaining and setting serial port respectively. Both system calls take two parameters: a file handle (file descriptor) and a device control block (DCB). A DCB is an input type parameter for `SetCommState()` while DCB is an output type for `GetCommState()`. There is a slight difference on invocations of these two calls within CLISP environment as it is shown in figure 8. Operating system creates a session when CLISP environment runs. Initial hardware setting will inherit from somewhere and stay at the lastest setting by SetCommState(). Thus, a LISP program may setup using these functions and then operate the port using standard LISP file routines such as `open`, `close`, `with-open-file`, `read-byte`, `write-byte`, `read-char`, `write-char`, and others.

```
(defmacro with-open-com1 ((stream) &body body)
  (multiple-value-bind (body-rest declarations) (system::parse-body BODY)
   `(let ((,stream))
    (with-c-var (lpDCB `DCB)
    (with-c-var (hfile `int)
    (declare (read-only ,stream) ,@declarations)
    (setf hFile (CreateFileW "COM1" -2147483648 0 NIL 3 0 0))
    (GetCommState hFile (c-var-address lpDCB))
    (SetCommState hFile (make-DCB :DCBlength          28
                                  :BaudRate           9600
                                  :fFlag              0
                                  :wReserved          0
                                  :XonLim             2048
                                  :XoffLim            512
                                  :ByteSize           8
                                  :Parity             0
                                  :StopBits           0
                                  :XonChar            17
                                  :XoffChar           19
                                  :ErrorChar          0
                                  :EofChar            26
                                  :EvtChar            0
                                  :wReserved1         0))
    (CloseHandle hFile)
    (setf ,stream (open "COM1" :direction :io :element-type `unsigned-byte))
    (unwind-protect
      (multiple-value-prog1 (progn ,@body-rest)
        (when ,stream
          (close ,stream)
          (setf hFile (CreateFileW "COM1" -2147483648 0 NIL 3 0 0))
          (SetCommState hFile lpDCB)
          (CloseHandle hFile)))
      (when ,stream
        (close ,stream :abort t)
        (setf hFile (CreateFileW "COM1" -2147483648 0 NIL 3 0 0))
        (SetCommState hFile lpDCB)
        (CloseHandle hFile)))))))
```

Figure 8: A macro definition whose structure is similar to `with-open-file`.

Figure 8 shows a marco that does following: 1) open a port, 2) obtain and save current settings, 3) apply new settuns, 4) close the port, 5) open the port with regular LISP file routine, 6) executes an implicit progn forn, 7) close the port and restore port settings as soon as control flow leaves the body either normally or abnormally. Note that line 8 and line 9 show different parameter passing for `GetCommState()` and `SetCommState()`. Despite both pass pointers to a DCB structure as a parameter, one pass the address of a variable created by `with-c-var` while the other pass a pointer to a DCB structure created dynamically.

3-3 Access to an USB Camera

In previous section, we have shown how to access to operating system calls from CLISP environment using FFI package. In order to demonstrate power of FFI package, we show an example to open an USB camera, fetch motion images, and calculate frame rate using Intel OpenCV library. Function of calculating frame rate is a local function (i.e. LISP function), and functions of opening and fetching images are foreign functions (i.e. C functions). This example demonstrates program control flow is passing between inside and outside CLISP environment. Before we discuss the program, we would like to briefly discuss functions that will be used in this example. Intel OpenCV library provides a set of GUI functions called `highgui` which allow programmer

to load, create, show, and release images. The library also comes with several camera routines called `cvcam`. These routines are used to access the cameras.

Library (.h) and Location* (.dll)	Function Name and Arguments	Description
highgui.h highgui096.dll	`int cvNamedWindow(const char* name, int flags);`	Create a GUI window by name.
	`void* cvGetWindowHandle(const char* name);`	Get a GUI window handle by name
	`int cvWaitKey(int delay);`	Wait for a key and allow user drags GUI windows.
	`void cvDestroyAllWindows(void);`	Destroy all GUI windows.
cvcam.h cvcam096.dll	`int cvcamGetCamerasCount();`	Obtain number of available cameras. This routine will "trigger" all cameras.
	`int cvcamSetProperty(int camera, const char* property, void* value);`	Setup a particular camera's property.
	`int cvcamInit(void);`	Initialize all cameras.
	`int cvcamStart(void);`	Start receiving images.
	`int cvcamStop(void);`	Stop receiving images.
	`int cvcamExit(void);`	Finalize all cameras.

*Depending on version of the library. 096 is the version number for beta4.

Figure 9: Summary of OpenCV functions used in the example.

Camera library support multiple cameras at once. To enable cameras, programmer has to switch on individual camera's property in settings. Furthermore, `cvcamInit()` is not the very first step to initialize cameras. Instead, `cvcamGetCamerasCount()` is used. `cvcamInit()` is used for initialize cameras after properties are set. Each camera can individually call back to a function that processes individual frame when property RENDER is enabled. The callback function is invoked upon a frame is received.

Figure 10 shows an example to call up a camera and calculate frame rate. Recall that each camera can carry one callback function for processing individual frame. A camera's property takes only a pointer to the callback function. Thus, we need to somehow store a pointer to a LISP function in a camera's property. This example consists of two paragraphs. The first paragraph defines body of the function (LISP code), and the second paragraph is a series of OpenCV function call that opens the camera and does the computation. Note that there are four variables created: `cvTrue`, `cvFalse`, `hWnd`, and `callback`. All variables are type of c-pointer. This makes easier to make type casting or conversion for store a camera's property. The value of a property is a `void*` type in C. The actual data type of the value is determined by property item. This is the nature of OpenCV's internal data structure. As a LISP programmer, we have no power to change it. In C program, an integer may be casted implicitly to an address. That is, programmer can assign integer value to a variable whose type is pointer. However, this is not doable. FFI package checks all data type on c-type variables assignments. Thus, an explicit casting has to be done on these variables before the call. `cvTrue` and `cvFalse` are basically used for storing constants 1 and 0 as Boolean value. `hWnd` is a handle to a window. This is also a `void*` type defined in OpenCV. `callback` is a pointer to the callback function. Line 3 to 7 of second paragraph, these `setf` uses cast to explicitly convert data type.

3-4 A Simple Textual Terminal for Robosapien

This is a LISP program that allows user to enter commands in text to communicate with Robosapien. This program uses the marco described in section 3-2 to access a serial port. Each text command is assigned an

```
;;; Callback function (body)
(defun _callback (image)
  (if (> (- (get-internal-real-time) i) 0)
    (write (float (/ 10000000 (- (get-internal-real-time) i)))))
  (write-string " fps  ")
  (write-char #\Return)
  (setf i (get-internal-real-time)))

;;; Open camera.
(with-c-var (cvTrue   'ffi:c-pointer)
(with-c-var (cvFalse  'ffi:c-pointer)
(with-c-var (hWnd     'ffi:c-pointer)
(with-c-var (callback 'ffi:c-pointer)
(setf (cast callback  '(ffi:c-function (:language    :stdc)
                                       (:arguments   (image ffi:c-pointer))
                                       (:return-type nil))) #'_callback)
(setf (cast cvTrue 'int) 1)
(setf (cast cvFalse 'int) 0)
(cvNamedWindow "Camera1" 0)
(setf hWnd (cvGetWindowHandle "Camera1"))
(cvcamGetCamerasCount)
(cvcamSetProperty 0 "enable" cvTrue)
(cvcamSetProperty 0 "render" cvTrue)
(cvcamSetProperty 0 "window" (ffi:c-var-address hWnd))
(cvcamSetProperty 0 "callback" callback)
(cvcamInit)
(cvcamStart)
(cvWaitKey 0)
(cvcamStop)
(cvcamExit)
(cvDestroyAllWindows)))))))))
```

Figure 10: An example of open a camera using OpenCV.

8-byte value which is the internal command code of Robosapien*. This program can only take single command in a word. The algorithm is simple. A text command is searched in the command table recursively and returns the corresponding command code. Then, the code is sent to the robot through a serial port. The port is opened with the following parameter: 9600 bps, no parity, 8 bit length, 1 stop bit. If a command is not found in the table, an "error command" is sent. An error command is essentially a specific robot movement that adds more flavors between user and robot. Furthermore, specific commands are sent when the program begins and terminates. Please refer to section 5 for the entire source code.

There are a few problems that I discovered along the way program the robot. First, the robot does not take command upon start of the terminal program. This is caused by BasicStamp's characteristic. BasicStamp resets its microprocessor each time when port is opened and closed, which takes about one second before running the program. Thus, a time delay has to be inserted as shown in the source code. Second problem is that the robot occasionally does not take commands. From control program's prospective, this is caused by unreliable hardware. The command has to be injected into robot's internal microprocessor through a number of wired and unwired serial transmissions. A command could be failed if anything goes wrong along the way. The system should have some indicators on both computer and robot sides. This can save amount of time during the development.

* The command code is provided by robot's modifier Kristine Smith.

4 Conclusion

I have demonstrated how to use a publicly available Common LISP environment to cooperate with libraries – sets of functions written in the languages other than LISP. With FFI package, LISP is even capable to do low-level programming. Doing low-level programming such as sending robot control code and accessing a camera and other devices is an essential. Furthermore, a complete robot program now can be devised into several parts using different language including LISP. FFI package allows programmer to call LISP functions from other languages. This suggests that it is possible to have LISP program be either major control program or assistive program.

5 Source Code

```
(defun run ()
  (with-open-com1 (com1)
    ; Local varaible declaration.
    (let ((input) (parsedInput))
    ; Robot initilization
    (pause 1000) ; *** BasicStamp takes 1000ms to intialize.
    (write-byte 177 com1)
    ; Loop body.
    (loop (write-string "ROBOSPAIEN> " *terminal-io*)
          (setf input (read-line-no-punct))
          (cond ((equal input `quit)
            (write-byte 163 com1) ; Exit program.
            (return `bye)))
          (setf parsedInput (lookup input commandSet))
          (cond ((null parsedInput) (write-byte 194 com1)) ; Send unknown comamnd.
                (T (write-byte parsedInput com1)))))))     ; Send command.


(setf commandSet `(
  (reset              174)
  (stop               142)
  (turn-right         128)
  (turn-right-step    160)
  (tilt-body-right    131)
  (right-arm-up       129)
  (right-arm-down     132)
  (right-arm-in       133)
  (right-arm-out      130)
  (right-hand-thump   161)
  (right-hand-pickup  164)
  (right-hand-throw   162)
  (right-hand-sweep   193)
  (right-hand-strike  197)
  (right-hand-strike2 195)
  (right-hand-strike3 192)
  (turn-left          136)
  (turn-step-step     168)
  (tilt-body-left     139)
  (left-arm-up        137)
  (left-arm-down      140)
  (left-arm-in        141)
  (left-arm-out       138)
  (left-hand-sweep    201)
  (left-hand-strike   205)
  (left-hand-strike2  203)
```

```
  (left-hand-strike3  200)
  (left-hand-thump    169)
  (left-hand-pickup   172)
  (left-hand-throw    170)
  (walk-forward-step  166)
  (walk-forward       134)
  (go-fwd-step        166)
  (go-fwd             134)
  (lean-forward       173)
  (walk-backward-step 167)
  (walk-backward      135)
  (go-back-step       167)
  (go-back            135)
  (lean-backward      165)
  (high5              196)
  (talk-back          204)
  (whistle            202)
  (dance              212)
  (bulldozer          198)
  (opps               199)
  (roar               206)
  (demo1              210)
  (demo2              211)
  (demoall            208)))



(defun lookup (key database)
  (cond ((null database) NIL)
        ((equal (first (first database)) key) (second (first database)))
        (T (lookup key (rest database)))))



(defun read-line-no-punct ()
  "Read an input line, ignoring punctuation."
  (read-from-string
    (concatenate 'string (substitute-if #\space #'punctuation-p (read-line)))))



(defun punctuation-p (char)
  (find char ".,;:`!?#()\\"))



(defun print-with-spaces (list)
  (mapc #'(lambda (x) (prin1 x) (princ " ")) list))



(defun print-with-spaces (list)
  (format t "~{~a ~}" list))



(defun pause (MILLISECOND)
  (do ((N (get-internal-real-time)))
       ((> (- (get-internal-real-time) N) (* MILLISECOND 10000)))) T)



(defmacro with-open-com1 ((stream) &body body)
```

```
  (multiple-value-bind (body-rest declarations) (system::parse-body BODY)
   `(let ((,stream))
    (ffi:with-c-var (lpDCB `DCB)
    (ffi:with-c-var (hfile `ffi:int)
    (declare (read-only ,stream) ,@declarations)
    (setf hFile (CreateFileW "COM1" -2147483648 0 NIL 3 0 0))
    (GetCommState hFile (ffi:c-var-address lpDCB))
    (SetCommState hFile (make-DCB :DCBlength          28
                                  :BaudRate           9600
                                  :fFlag              0
                                  :wReserved          0
                                  :XonLim             2048
                                  :XoffLim            512
                                  :ByteSize           8
                                  :Parity             0
                                  :StopBits           0
                                  :XonChar            17
                                  :XoffChar           19
                                  :ErrorChar          0
                                  :EofChar            26
                                  :EvtChar            0
                                  :wReserved1         0))
    (CloseHandle hFile)
    (setf ,stream (open "COM1" :direction :io :element-type `unsigned-byte))
    (unwind-protect
      (multiple-value-prog1 (progn ,@body-rest)
        (when ,stream
          (close ,stream)
          (setf hFile (CreateFileW "COM1" -2147483648 0 NIL 3 0 0))
          (SetCommState hFile lpDCB)
          (CloseHandle hFile)))
      (when ,stream
        (close ,stream :abort t)
        (setf hFile (CreateFileW "COM1" -2147483648 0 NIL 3 0 0))
        (SetCommState hFile lpDCB)
        (CloseHandle hFile))))))))



;/**
; * typedef struct _DCB {
; *   DWORD DCBlength;
; *   DWORD BaudRate;
; *   DWORD fBinary           :1;
; *   DWORD fParity           :1;
; *   DWORD fOutxCtsFlow      :1;
; *   DWORD fOutxDsrFlow      :1;
; *   DWORD fDtrControl       :2;
; *   DWORD fDsrSensitivity   :1;
; *   DWORD fTXContinueOnXoff :1;
; *   DWORD fOutX             :1;
; *   DWORD fInX              :1;
; *   DWORD fErrorChar        :1;
; *   DWORD fNull             :1;
; *   DWORD fRtsControl       :2;
; *   DWORD fAbortOnError     :1;
; *   DWORD fDummy2           :17;
; *   WORD  wReserved;
; *   WORD  XonLim;
; *   WORD  XoffLim;
; *   BYTE  ByteSize;
; *   BYTE  Parity;
; *   BYTE  StopBits;
```

```
; *   char  XonChar;
; *   char  XoffChar;
; *   char  ErrorChar;
; *   char  EofChar;
; *   char  EvtChar;
; *   WORD  wReserved1;
;/* } DCB;
(ffi:def-c-struct DCB
  (DCBlength  ffi:int)    ; 28 bytes.
  (BaudRate   ffi:int)
  (fFlag      ffi:uint32) ; bitwise see original struct in C.
  (wReserved  ffi:short)
  (XonLim     ffi:short)
  (XoffLim    ffi:short)
  (ByteSize   ffi:char)
  (Parity     ffi:char)
  (StopBits   ffi:char)
  (XonChar    ffi:char)
  (XoffChar   ffi:char)
  (ErrorChar  ffi:char)
  (EofChar    ffi:char)
  (EvtChar    ffi:char)
  (wReserved1 ffi:short))



;/**
; * BOOL GetCommState(
; *   HANDLE hFile
; *   LPDCB lpDCB
; * );
;*/
(ffi:def-call-out GetCommState
  (:language :stdc)
  (:name "GetCommState")
  (:library "KERNEL32.DLL")
  (:arguments (hfile ffi:int)
              (lpDCB ffi:c-pointer))
  (:return-type BOOLEAN))


;/**
; * BOOL SetCommState(
; *   HANDLE hFile
; *   LPDCB lpDCB
; * );
;*/
(ffi:def-call-out SetCommState
  (:language :stdc)
  (:name      "SetCommState")
  (:library   "KERNEL32.DLL")
  (:arguments (hfile ffi:int)
              (lpDCB (ffi:c-ptr DCB)))
  (:return-type BOOLEAN))



;/**
; * HANDLE CreateFileW(
; *   LPCTSTR               lpFileName
; *   DWORD                 dwDesiredAccess
; *   DWORD                 dwShareMode
; *   LPSECURITY_ATTRIBUTES lpSecurityAttributes
```

```
; *   DWORD                 dwCreationDisposition
; *   DWORD                 dwFlagsAndAttributes
; *   HANDLE                hTemplateFile
; * );
;*/
(ffi:def-call-out CreateFileW
  (:language :stdc)
  (:name      "CreateFileA")
  (:library   "KERNEL32.DLL")
  (:arguments (lpFileName            ffi:c-string)
              (dwDesiredAccess       ffi:int)
              (dwShareMode           ffi:int)
              (lpSecurityAttributes  ffi:c-pointer)
              (dwCreationDisposition ffi:int)
              (dwFlagsAndAttributes  ffi:int)
              (hTemplateFile         ffi:int))
  (:return-type ffi:int))



;/**
; * BOOL CloseHandle(
; *   HANDLE hObject
; * );
;/*
(ffi:def-call-out CloseHandle
  (:language :stdc)
  (:name        "CloseHandle")
  (:library     "KERNEL32.DLL")
  (:arguments   (hObject ffi:int))
  (:return-type ffi:boolean))



;/**
; * BOOL CloseHandle(
; *   HANDLE hObject
; * );
;/*
(ffi:def-call-out GetLastError
  (:language :stdc)
  (:name "GetLastError")
  (:library "KERNEL32.DLL")
  (:return-type ffi:int))


; Run the program.
(write-line "")
(run)
```

Reference

[1] History of Lisp Programming Language. *http://en.wikipedia.org/wiki/Lisp_programming_language#History*.

[2] CLISP – an ANSI Common Lisp Implementation. *http://clisp.cons.org/*.

[3] Open Source Computer Vision Library. *http://www.intel.com/research/mrl/research/opencv/*.

[4] Introduction to Windows Socket 2. *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/winsock/winsock/windows_sockets_start_page_2.asp*.

[5] Description of SHBrowseForFolder. *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/shellcc/platform/shell/reference/functions/shbrowseforfolder.asp.*

[6] The Foreign Function Call Facility. http://clisp.cons.org/impnotes/dffi.html.

[7] Robosapien. http://www.wowwee.com/robosapien.

[8] BASIC Stamp Module. http://www.parallax.com/html_pages/products/basicstamps/basic_stamps.asp.

[9] Radio Frequency Module. http://www.parallax.com/html_pages/products/communication/rf_modules.asp.

[10] Understanding Device Names and Symbolic Links. http://support.microsoft.com/default.aspx?scid=kb;en-us;235128.