Reviving Lisp for smaller programmable machines

Raman Gopalan
Volunteer free software programmer, SimpleMachines, Italy
April 2015

Accepted for publication in "Electronics for you", India (Article number: 9960)

Abstract — "if some languages claim to be the Swiss army knife of programming, then Pico Lisp may well be called the scalpel of programming: sharp, accurate, small, lightweight but also dangerous in the hands of the inexperienced". Lisp offers a practical mathematical notation to write computer programs, mostly being influenced by lambda calculus. It is still the most favored programming language for artificial intelligence research.

Programming microcontrollers is a real challenge, given the complexity of today's microcontroller architecture. The C language is widely used to program them. The philosophy of cross-compiling a program for the target seems to be the most popular choice among microcontroller aficionados. An ARM Cortex M4 clone (such as the Infineon XMC4500) typically has about a megabyte of flash and a few hundred kilobytes of internal RAM. With resources of that order on the chip, a much more interesting approach is to program the device natively with a powerful, dynamic language like Lisp.

This project aims at reviving Lisp for native, interactive and incremental microcontroller program development by running a dialect of Lisp as virtual machine on the target. The project also demonstrates the power of Lisp through the execution of various expressive Lisp programs on a microcontroller.

Keywords — Lisp; AI (Artificial Intelligence); *Virtual machines; Compilers; HAL; Interpreters; Lambda calculus; Microcontrollers; ARM-Cortex M4; REPL (Read-evaluate-print-loop);*

I. Introduction

Dynamic languages like Lisp [1] have been in existence as a versatile tool for rapid application development. It has heavily influenced and furthered computation in various fields. Myriad system programs use high level languages to natively extend their functionality. The recent (but nonchalant) trend of the application of dynamic languages for programming embedded devices has seen a ramp. A lot of interesting, practical embedded solutions have been developed so far with such languages, supported as a part of a virtual machine. A port of Python-2.5 to the Nintendo DS console [2] is one such example. With an increasing magnitude of applications being written in high-level languages on embedded devices, there is a very high possibility of this trend occupying a hot spot in the embedded market for a selective set of system applications in the near future. Figure 1 presents the system architecture of a natively programmable, digitally controlled system.

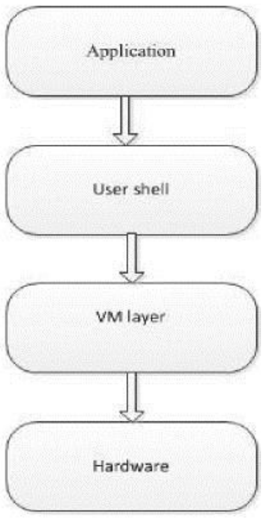

Figure 1: General MCU software system with a VM layer

With the above architecture, it is possible to write abstract, self-adapting middle-level drivers for hardware modules on the microcontroller. This enables the possibility of platform independent, native embedded software development.

Lisp is the second-oldest high-level programming language (the first one being FORTRAN). It is known for its association with AI. Linked lists are one of its major data structures. One of the most interesting properties of Lisp is its homoiconic nature. A program written in Lisp is itself constructed with lists. The equivalence of code and data is a major advantage. The powerful Lisp macro system heavily depends on the fact that Lisp programs can manipulate Lisp code as if it were data. This permits the creation of new syntax within the context of Lisp.

Although projects like PICOBIT [3] and ARMPIT Scheme [4] already implement a compact Lisp programming language for a microcontroller, there are many reasons to consider yet another dialect of Lisp for programming microcontrollers.

PICOBIT implements a Scheme system in less than 7 KB of main memory and provides an efficient virtual machine but fails to incorporate a powerful hardware abstraction layer for writing efficient, portable Lisp across various hardware architectures. ARMPIT Scheme is an implementation of the

Scheme programming language for RISC machines with an ARM core. Its implementation is based on the description in

the R5RS standard. It is implemented in the ARM assembly language, making the system difficult to port across various RISC machines without an ARM core. Since the core of the implementation is written in assembly language, the extension of the core to support various hardware specific features (marshalling) in Scheme becomes a herculean task.

Tools like SWIG can generate language binding code between C and a high-level language like Scheme but given the nature of PICOBIT's implementation, generating ARM assembly code for interfacing user code (mostly written in C) with PICOBIT becomes a practical impossibility. This project aims at providing a solution for the aforementioned concerns by using a well-accepted dialect of Lisp called Pico Lisp [5] and constructing hardware abstraction layers for various hardware peripherals on the MCU to ensure efficient, portable application development in Lisp. In addition, the project shows how to extend Pico Lisp to include support for hardware extensions.

II. WHY PICO LISP?

Pico Lisp is constructed as a virtual machine. It is also a dialect of the Lisp programming language. It is written in portable C and is easily extendable. After much research and programming to narrow down on a Lisp implementation, Pico Lisp was chosen as a virtual machine for the following reasons:

- Dynamic data types and structures
- Formally homoiconic
- Functional programming paradigm
- An interactive REPL
- Pilog – a declarative language with semantics of Prolog in Pico Lisp
- Small memory footprint
- Permissive, non-copyleft free software license

At the lowest level, Pico Lisp programs are constructed from a single data structure called "cell". A cell is a pair of machine words, which traditionally are called CAR and CDR in the Lisp terminology. These words can represent either a numeric value (scalar) or the address of another cell (pointer). All higher level data structures are built out of these cells. Pico Lisp supports the following basic data types: numbers, symbols and lists. As a result, Pico Lisp is one of the fastest Lisp dialects available since only fewer options are checked at runtime to parse a value.

Pico Lisp promotes the following key characteristics:

- Programs are written by gluing existing components
- The language is highly scalable and extensible
- Automatic memory management
- Dynamically bound and typed
- Interactive programming with symbolic interpretation

Pico Lisp in addition supports an integrated database system. This is a huge advantage for embedded system applications requiring a convenient facility to perform data transactions.

III. PICO LISP ON RISC MACHINES

Pico Lisp cannot be directly compiled for a 32 bit RISC machine. There are many issues to address before one can use the Pico Lisp REPL over the UART or TCP/IP interface on the microcontroller. For instance, support programs such as memory allocators are required for Pico Lisp to function correctly. Since we intend for Pico Lisp to run on bare metal, we use the Newlib C library [6] and implement stubs of code for the memory allocator. We also have to concern ourselves with issues like routing plain I/O over the UART or TCP/IP interface of the microcontroller. Listing 1 shows an example of a stub file written for Newlib. The file needs to be compiled along with the Pico Lisp code base.

We also require support for an MMC interface to store Pico Lisp programs. We can then load the Lisp programs at runtime. This implies a requisite for a file system. We also need to implement stubs of code for file I/O support over the SPI protocol. For the file system, we use the FatFs FAT file system module [7].

Once all the support programs are in place, getting Pico Lisp to run on the microcontroller is then fairly straightforward process. On account of its small size, it can be easily embedded on a microcontroller (bare metal or from within the context of an operating system) in less than 256KB of flash. It can be easily compiled for a given architecture with a tool-chain like GNU gcc. Currently, a Python based build system called SCons [8] is being used to compile the code base.

```
#define USE_MULTIPLE_ALLOCATOR
# define CNAME(func) dl##func
#else
# define CNAME(func) s##func
#endif

void *_malloc_r(struct reent *r, size_t size) {
  return CNAME(malloc)(size);
}

void _free_r(struct reent *r, void *ptr) {
  CNAME(free)(ptr);
}

void *sbrk_r(struct reent *r, ptrdiff_t incr) {
  // sbrk implementation here.
}

// _open_r looks for a device first and opens it.
int open(const char *name, int flags, mode_t mode) {
  return _open_r(_REENT, name, flags, 0);
}
```

Listing 1: Sample C stub file for Newlib

IV. SOFTWARE ARCHITECTURE

The overall logical software structure for running full-fledged Pico Lisp on the microcontroller is indicated in Figure 2. It shows the communication between the Pico Lisp virtual machine and various other modules in the code base.

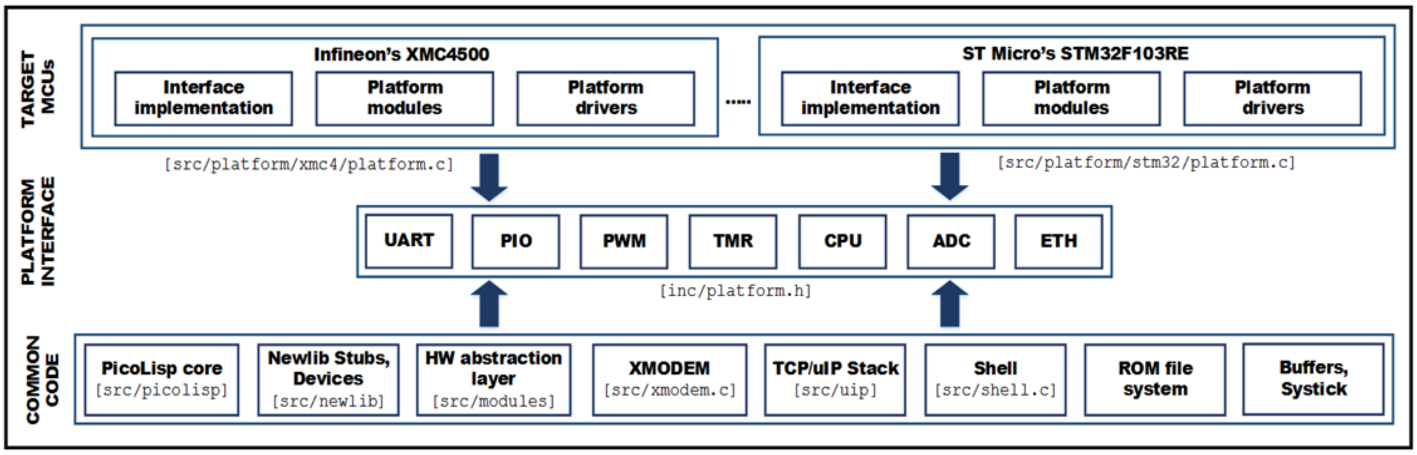

Figure 2: Software system architecture for Pico Lisp

The code uses the notion of "platform" to denote a group of CPUs that share the same core structure, although their specific silicon implementation might differ in terms of integrated peripherals, internal memory and other such attributes. A port of Pico Lisp implements support code for running Lisp on one or more CPUs from a given platform. For example, the Infineon XMC4000 port of Pico Lisp can run on XMC4500, XMC4400 and XMC4200 CPUs, all of them forming a part of the XMC4000 platform. The code base remains highly portable across various platforms and architectures simply by using the following key principles:

- Code that is platform-independent is "common code" and should be written in portable ANSI C as much as possible. Pico Lisp itself is a part of the common code section and is written this way.
- Code that is not generic (mostly peripheral and CPU specific code) must still be made as portable as possible by using a common interface that must be implemented by all platforms on which Pico Lisp runs. This interface is called "platform interface".
- Platforms vary greatly in capabilities. The platform interface tries to group only common attributes of different platforms.

  Access to specific functionality on a given platform (like the High Resolution PWM module on the XMC4400 which is not available on the XMC4500) should be done by using a "platform module".

A. Common code

The following provides a list of items that can be classified as common code:

- The Pico Lisp code base with modifications to make it work in limited memory conditions.
- Components like the ROM file system, an XMODEM protocol implementation to receive data from another device such as a PC, the shell, the onboard clone of the vi text editor [9] for editing files, the TCP/IP stack.
- C library specific code (allocators and Newlib stubs).
- Generic peripheral support code, like the ADC support code that is independent of the actual ADC hardware.

B. Platform interface

The platform interface allows writing extremely portable code over a large variety of platforms, from C and Lisp. An important property of the platform interface is that it tries to group only common attributes of different platforms. For example, if a platform has a UART module that can work in loopback mode, but the others platforms lack this support, the loopback feature will not be included in the platform interface. The platform interface is mainly used by the generic modules to allow Lisp code to access platform peripherals. It can also be used by C code that wants to implement a generic module that needs access to peripherals. For example, the drivers for a graphical LCD device are implemented this way by using the generic platform interface.

The platform interface is declared in the *inc/platform.h* header file from source distribution. It is a collection of various components like UART, SPI and timers. Each component has an identifier which is a number that identifies that component in Pico Lisp. Generally, numbers are assigned to components in their "natural" order: for example, PORTA will have the identifier value as 0. PORTB will have 1 and so on. Similarly, the second SPI interface (SPI1) of the MCU will probably have an identifier value of 1. Pin 0 in PORT1 on the Infineon XMC4500 will be called ‘P1_0 in Pico Lisp. Similarly, pin 27 in PORTB on the Atmel at32uc3a0256 will be called ‘PB_27 (notice the quote).

C. Platform specific modules

A platform implementation might also contain one or more platform specific modules. Their purpose is to allow Lisp to use the full potential of the platform peripherals -- Not just the functionality covered by the platform interface, but also functionality specific to the platform. For example, the Lisp extensions for the OLED module (over SPI) on the Infineon XMC4000 Hexagonal kits are implemented this way.

D. Booting Pico Lisp on the microcontroller

Given below is the sequence of events that occur after the microcontroller is powered up:

The platform initialization code is executed. This program does very low level platform setup, copies ROM contents to the internal RAM, zeroes out the BSS section, sets up the stack pointer and long jumps to the C main function.

1. The main function calls the platform specific initialization function and returns a result which can, be either a value indicating success or failure. If it fails, main instantly blocks. A debugger can then inspect the internals of the state machine.
2. The main function then initializes the rest of the system: the ROM file system, XMODEM, and terminal support.
3. If the files “/rom/autorun.l” or “/mmc/autorun.l” exist, they are executed. If one file is found before the other, it terminates further execution of the other file and the context jumps to the next step. If it returns after execution, or if the files are not found, the boot process continues with the next step.
4. If the boot parameter is set to 'standard' and the shell was compiled in the image, it is started. In the absence of the shell, the standard Pico Lisp server is started.

V. USING PICO LISP -- MCU SOFTWARE DEVELOPMENT

Pico Lisp can be compiled to either support a user console over UART (the default and by far the most popular) or a console over TCP/IP.

Pico Lisp can run on a wide variety of microcontrollers. Some of the practical aspects of using Pico Lisp are listed below:

1. The code base is hardware independent. It proves to be extremely portable across different architectures.

2. Programs in Pico Lisp are highly adaptable, field-programmable and re-configurable for a variety of practical applications.

3. Programming the MCU follows a very natural iterative process because Pico Lisp permits the user to develop programs in an interactive and incremental way. The code supplies various tools to aid in native Lisp programming (like an onboard vi-clone text editor and an XMODEM implementation to share files).

4. Pico Lisp is a very extensible piece of software. Adding support for newer peripherals or modules is a naturally smooth process.

5. The code bears a royalty-free, permissive, non-copyleft free software license. This permits code reuse within a proprietary digital base.

VI. EXAMPLES

Once we have Pico Lisp running on the board, one can access all MCU peripherals from Lisp. Symbols can be passed around, manipulated and inspected at runtime. Let us now see a few examples on how this can be done.

A. Hello world - blinking the LED

Listing 2 shows a simple Pico Lisp program which toggles an LED on the board every second. The quoted symbols PB_29 and PX_16 are Pico Lisp symbols which correspond to the pin number 29 on PORTB and pin number 16 on PORTX respectively. The parsing of these values is done in the generic PIO Lisp module (via the hardware abstraction layer). Transient symbols `*pio-output*` and `*pio-input*` are used to set directions to the port pins. To see how these values are in Pico Lisp, see [10].
A simple infinite loop reads the button on the input pin and toggles the LED if pressed.

```
# A sample for user-buttons.

# Declare pins
(setq led 'PB_29 button 'PX_16)

# A simple delay function
(de delay (t)
   (tmr-delay 0 t) )

# Make sure the LED starts in
# the "off" position and enable
# input/output pins
(de init-pins ()
   (pio-pin-sethigh led)
   (pio-pin-setdir *pio-output* led)
   (pio-pin-setdir *pio-input* button) )

# And now, the main loop
(de prog-loop ()
   (init-pins)
   (loop
      (if (= 0 (pio-pin-getval button))
         (pio-pin-setlow led)
         (delay 100000)
         (pio-pin-sethigh led)
         (delay 100000) ) ) )

(prog-loop)
```

Listing 2: Programming usr buttons in PicoLisp

B. Send a byte to the I2C multiplexer

Listing 3 is a Pico Lisp program that can be used to send a byte of data to a slave device over the I2C multiplexer. The I2C module for Pico Lisp is platform agnostic. Most platforms (including the XMC4000) include an I2C interface. Please note: The program in Listing 3 works on an Atmel AT32UC3A0512 microcontroller. The print operation (prinl) prints the output on the Pico Lisp console (UART or TCP/IP).

```
 # Send byte to i2c mux to enable left i2c bus

 (setq

   id 0             # Which i2c bus to use?
   mux-addr 112     # The slave address of the i2c mux
   mux-disable 0    # Control word to disable the mux
   mux-left 4       # Control word to enable left bus
   mux-right 5 )    # Control word to enable right bus

 (i2c-start 0)

 (if (not (= T (i2c-address id
                            mux-addr
                            *i2c-transmitter*)))
   (prinl "The multiplexer did not reply")
      (if (not (= (i2c-write id mux-left) mux-
left))
    (prinl "The mux did not ack the write") ) )

 (i2c-stop id)
```

Listing 3: Send a byte to the I2C multiplexer

VII. CONCLUSION

The problems of the non-portability of source code and application code in various Scheme implementations mentioned in the introduction section are solved by using Pico Lisp. Using Pico Lisp as a virtual machine on the microcontroller proves to be a very handy tool for doing rapid application development. It permits easy incremental programming on the REPL. Complex programs involving a database or decision trees can be written in just a handful of lines of code. For more information, see tic-tac-toe [11] running on a microcontroller. The code base is developed as free and open source software. It is hosted on Github [12].

VIII. REFERENCES

[1] John McCarthy. “Recursive functions of symbolic expressions and their computation by machine, part-1”, MIT Cambridge, 1960.

[2] Lorenzo Mancini. “Nintendo DS/Stackless Python 2.5” http://disinterest.org/NDS/Python25.html

[3] Vincent St-Amour, Marc Feeley. “PICOBIT: A compact Scheme system for microcontrollers”. ni ersit e de Montreal

[4] Fischell Department of Bioengineering. “ARMPIT Scheme – A Scheme interpreter on a microcontroller” http://armpit.sourceforge.net/

[5] Alexander Burger. “Pico Lisp: A radical approach to application development”, June 2006.

[6] Corinna Vinschen, Jeff Johnston. “Newlib C library” http://sourceware.org/newlib/

[7] Elm Chan. “FatFs FAT file system module” http://elm-chan.org/fsw/ff/00index_e.html

[8] The SCons foundation. http://www.scons.org/

[9] Christopher Cole, (modified by Raman Gopalan). “iv - a vi clone for microcontrollers” https://github.com/simplemachines-italy/Alcor6L/blob/master/src/iv/iv.c

[10] Raman Gopalan. “Internal transient symbols for PIO”. https://github.com/simplemachines-italy/Alcor6L/blob/master/src/picolisp/src/tab.c

[11] Alexander Burger. “tic-tac-toe” https://github.com/simplemachines-italy/examples/blob/master/tic-tac-toe/ttt.l

[12] SimpleMachines. “Alcor6L” https://github.com/simplemachines-italy/Alcor6L