

The Chirp Language Specification

Version 2

Li Xu

Department of Computer Science
University of Massachusetts Lowell
Lowell, MA 01854

1 Introduction

Chirp is a simple C-like imperative language targeting embedded microcontrollers used in personal robot devices, eg. Parallax Scribbler robots. Processors used in this domain of embedded devices are highly cost-sensitive and generally have only limited set of resources and operation commands. Software running on such devices typically has no operating system support; instead, user programs take direct control of hardware through dedicated hardware control operations. Chirp is designed to support programming in this domain with simplified data types and control constructs reflecting common hardware features. To allow high-level access of hardware control operations, Chirp provides the special abstraction of System interface functions which programmers can use to control various I/O devices, eg. sensors and motors, of the system. Chirp compiler will translate System interface function calls to hardware control commands as part of the compiled program. The compiled code can then be uploaded to the device and run directly on hardware.

Chirp (version 2) is evolved from the original Chirp (v1) specification which targets the Cricket robot controller [4]. The first implementation target of Chirp v2 is the Parallax Scribbler robot, the hardware control is done through the System.Scribbler interface functions. In the future, support for other targets will be added with similar interface functions. Another goal of Chirp is to serve as a target compiler language for introductory and advanced courses on compiler construction. This document describes the language features and specification of Chirp.

2 Chirp Syntax

Chirp has a syntax similar to C. Chirp programs consist of variable and function definitions. Chirp variables are of several predefined types; user functions consist of statements including assignments, control flow statements, function calls, etc. Chirp is case sensitive, eg. `Foo` and `foo` are distinct names. Chirp keywords are reserved — the programmer cannot use a Chirp keyword as the name of a variable or function. The set of keywords include data type and statement keywords: `int`, `byte`, `nib`, `bit`, `pin`, `rom_int`, `rom_byte`, `if`, `else`, `loop`, `while`, `until`, `for`, `return`.

The following characters have special meaning in a Chirp program:

`{ } < > = + - * / % ! [] () . , ; : "`

Chirp accepts C/C++ style comments: multi-line comments are delimited by matching `/*` at the beginning and `*/` at the end; single-line comments follow `//` until the end of the line.

Chirp identifiers start with a letter or the underscore character which is then followed by letter, underscore or digit characters. Chirp identifiers are used for variable, array and function names. The lexical tokens for Chirp identifiers are defined as follows:

<i>program</i>	→	(<i>var_def</i> <i>func_def</i>)*
<i>var_def</i>	→	(const)? <i>data_type</i> ID (= INT)? ; <i>data_type</i> ID ([INT])+ ;
<i>data_type</i>	→	int byte nib bit pin (INT) rom_int rom_byte
<i>func_def</i>	→	(<i>data_type</i> void) ID (<i>arg_list</i>) <i>func_body</i>
<i>arg_list</i>	→	(<i>data_type</i> ID (, <i>data_type</i> ID)*)?
<i>func_body</i>	→	{ (<i>var_def</i>)* (<i>stmt</i>)* }
<i>stmt</i>	→	<i>assign_stmt</i> ; <i>if_else_stmt</i> <i>loop_stmt</i> <i>return_stmt</i> ; <i>break_stmt</i> ; <i>func_call</i> ; <i>block_stmt</i>
<i>assign_stmt</i>	→	ID ([<i>expr</i>])* = <i>expr</i>
<i>if_else_stmt</i>	→	if (<i>expr</i>) <i>stmt</i> (else <i>stmt</i>)?
<i>loop_stmt</i>	→	loop (while (<i>expr</i>))? <i>block_stmt</i> (until (<i>expr</i>))? for ID (<i>expr</i> : <i>expr</i> (: <i>expr</i>)?) <i>block_stmt</i>
<i>return_stmt</i>	→	return (<i>expr</i>)?
<i>break_stmt</i>	→	break
<i>func_call</i>	→	ID (. ID)* ((<i>expr</i> (, <i>expr</i>)*)?)
<i>block_stmt</i>	→	{ (<i>stmt</i>)* }
<i>expr</i>	→	! <i>relational_expr</i> <i>relational_expr</i> ((&&) <i>relational_expr</i>)*
<i>relational_expr</i>	→	<i>arithmetic_expr</i> ((== != <= < >= >) <i>arithmetic_expr</i>)*
<i>arithmetic_expr</i>	→	<i>term</i> ((+ -) <i>term</i>)*
<i>term</i>	→	<i>factor</i> ((* / %) <i>factor</i>)*
<i>factor</i>	→	ID ([<i>expr</i>])* INT STRING <i>func_call</i> (<i>expr</i>)

Figure 1: Chirp grammar

<i>Letter</i>	→	a b c ... z A B ... Z _
<i>Digit</i>	→	0 1 2 ... 9
<i>ID</i>	→	<i>Letter</i> (<i>Letter</i> <i>Digit</i>)*

Chirp constants include integer and string constants. String constants are quoted by double quotes and consist of character sequence of non-quote characters. Chirp does not have a string type, and string constants are used directly in Chirp programs.

Chirp supports the following boolean, relational and arithmetic operators:

- && for boolean and, || for or, ! for not;
- relational comparison ==, !=, <=, <, >=, >;
- arithmetic operation +, -, *, /, %. The % is the integer mod operator.

Chirp boolean values follow the C convention: 0 represents boolean false and non-0 value represents true.

The EBNF grammar of Chirp is shown in Figure 1. The lexical tokens are in the `typewriter` font.

3 Overview of Chirp Language Constructs

As shown in Figure 1, Chirp syntax is close to C. This is intentional, users who are familiar with C/C++ or Java languages can quickly pick it up and start programming robots using Chirp. However, as Chirp is designed to support embedded robot control rather than general-purpose computing, the

feature set of Chirp is greatly simplified and only include data types and control constructs essential for writing embedded robot control programs.

Chirp supports numeric data types, such as int and byte types. Reflecting the capability of the underlying hardware, Chirp also has bit, nib (4-bit), pin (1-bit with pin number) for I/O pin data, and rom_int and rom_byte for ROM based data. Users can define variables and arrays of the base types. Unlike general-purpose languages such as C, Chirp does not have abstract data type constructs for building user-defined types, reflecting again the characters of its target domain.

User computation in Chirp is done through function definitions. User functions can take arguments, define local variables, execute Chirp statements, and return a data value as result or null. Like C, the main function in Chirp program serves as the entry point of user code.

3.1 Scribbler Target

Chirp can be used to program the Scribbler robots. Scribblers are manufactured by Parallax Inc [3]. Scribblers run on embedded BASIC Stamp 2 (BS2) microcontrollers. As a highly constrained embedded system, a Scribbler has only 32 bytes RAM (for program variables) and 2K bytes ROM (code and data). The BS2 microcontroller has 16 programmable I/O pins which connect to Scribbler's 9 sensors, 2 DC motors, 3 LEDs, an on-board speaker and serial port for PC communication. Parallax provides a BASIC-style low-level command set (42 PBASIC commands) to program the robot [1, 2].

The pin assignments of Scribbler sensors and motors are listed in Figure 2.

Components	Pin #
Light Sensor: left	2
Light Sensor: center	1
Light Sensor: right	0
Line Sensor: enable	3
Line Sensor: left	5
Line Sensor: right	4
Stall Sensor	7
LED: left	10
LED: center	9
LED: right	8
Speaker	11
DC Motor: left	13
DC Motor: right	12
Obstacle Sensor: detector	6
Obstacle Sensor: left	15
Obstacle Sensor: right	14

Figure 2: Scribbler pin specification

Chirp data types match directly to the target data types on BS2 microcontroller:

- int: 16-bit integer data;
- byte: 8-bit integer data;
- nib: 4-bit integer data;
- bit: 1-bit data;
- pin: 1-bit data to access I/O pins; in addition to its value, the pin type specifies the pin number (int constant) of the pin;

- `rom_int`: 16-bit integer data in EEPROM;
- `rom_byte`: 8-bit integer data in EEPROM.

User can define variables with `int`, `byte`, `nib`, `bit` and `pin` types. The variables will be allocated in RAM memory. User can also use `rom_int` and `rom_byte` types to specify ROM-based data. Computation in Chirp is realized through expressions and statements. Chirp supports C-like general expressions, in which variable and array references and constants are used as building blocks to construct more complex expressions, including arithmetic, relational and boolean expressions. Expression values are used in Chirp statements. Chirp supports a set of simplified control constructs. Allowable statements are assignments, if-else conditional statements, unconditional and conditional loops, for-loops with a calculated loop count, function call and return. Like C, Chirp supports control abstraction through functions. The function body is a sequence of Chirp statements.

To support device control, Chirp provides special System “interface” functions. The System interface aggregates hardware I/O functions within the System namespace. Users can call System interface functions to control hardware devices, such as robot motors and sensors.

3.2 Example Chirp Program

An example Chirp program for Scribbler robot is shown in Figure 3.

It consists of variable definitions and three functions (including the `main` function): the robot starts by playing music notes on the speaker pin (Pin 11), then starts the motors and moves forward; it checks the stall sensor to see if it hits obstacles, and if so, it turns on all three LEDs, backs away and turns left. The code calls the `System.Scribbler` interface functions to control the robot.

4 Chirp Specification

4.1 Data Types and Conversion

Chirp data types are: `int`, `byte`, `nib`, `bit`, `pin`, `rom_int`, `rom_byte`. Type conversions in Chirp are implicit, that is, the compiler will determine the need of type conversions in expression evaluation if mixed types are used.

Similar to C, Chirp uses integer types to represent boolean values: 0 represents boolean false and any non-zero value represents boolean true.

Chirp also supports array types with array elements of the basic types. Array dimensions are fixed integer constants.

4.2 Functions and Variables

User can define functions in Chirp similar to C. Functions can take arguments, define local variables, and specify return value. Users can define global, function formal and local variables of basic and array types.

4.3 Assignment Statement

The assignment statement requires that its left-hand side to be a variable or array element and its right-hand side an expression.

4.4 If-Else Statement

The if-else statement evaluates the condition expression to a boolean value (using the zero/non-zero numeric value as boolean result) and executes the following “then” or “else” branch respectively. The `else` statement is always bound to the nearest unbound `if` statement to resolve ambiguity.

```

const int do = 391; // note Do's frequency
const int re = 494; // note Re
const int mi = 523; // note Mi
int tune_time;

bit stalled;
pin(11) speaker;

void startTune() {
  tune_time = 200; // 200ms
  System.Scribbler.sound(speaker, tune_time, do);
  System.Scribbler.sound(speaker, tune_time, re);
  System.Scribbler.sound(speaker, tune_time, mi);
}

void checkState() {
  Scribbler.senseStall(stalled);
  if (stalled) {
    System.Scribbler.setLED(1, 1, 1);
    System.Scribbler.moveBackward(5, 5, 1000);
    System.Scribbler.moveLeft(5, 5, 1000);
  } else {
    System.Scribbler.setLED(0, 0, 0);
  }
}

void main() {
  startTune();
  loop {
    System.Scribbler.moveForward(5, 5, 0);
    checkState();
  }
}

```

Figure 3: Example Chirp program

4.5 Loop and Break Statements

Chirp provides unconditional and conditional loops with `while` and `until` condition constructs. The `loop` statements will either execute the loop body in repetition (infinite loop), or evaluate the `while` (loop-while-body) or `until` (loop-body-until) expression and iterate the loop body depending on the condition: for `while` loop, the loop body will be executed if the condition is true; for `until` loop, the loop body will terminate if the condition is true. Chirp also has `for`-loop statements, in which the loop counter variable iterates from the loop-start value to the loop-end value with an optional step increment (`for-id-start-end-step`). The `break` statement can be used in the loop body to terminate loop iteration.

4.6 Return Statement

The `return` statement terminates the function and returns the control flow to the caller.

4.7 Function Call

Chirp functions are invoked by calling the named functions. Function calls need to supply the required arguments and function names should be qualified with appropriate interface names.

4.8 Block Statements

Similar to C, Chirp allows user to use curly braces to aggregate statements into code blocks.

4.9 Expressions

Chirp expressions compute values of the basic types. As boolean values are represented by integer values, Chirp defines boolean *and*, *or* and *not* operator to evaluate boolean values. The relational operators are `==`, `!=`, `<=`, `<`, `>=`, `>`. Chirp also supports common arithmetic operations.

4.10 Interfaces

The interface construct aggregates related I/O interface functions. The `System` interface consists of a list of interface functions. Interface functions can be called through the qualified interface and function names.

- `System.Scribbler`

The `System.Scribbler` interface functions provide low-level primitives to control the Scribbler robot:

```
void wait(time) wait for time millisecond,
```

```
void sound(pin_var, time, freq) play the specified frequency note for the duration of time  
millisecond on the speaker pin_var.
```

```
void input(var1, var2, ...) read variable values from the serial port.
```

```
void print(exp1, exp2, ...) print out expression values (including string constants) through  
the serial port.
```

```
void senseStall(stall_var) read the stall sensor state to stall_var (0 or 1).
```

```
void setLED(left, center, right) set Scribbler LEDs to specified state left, center,  
right, in which 0 for off and 1 for on.
```

```
void senseLight(left_var, center_var, right_var) read the light sensor states to inte-  
ger variables left_var, center_var and right_var.
```

`void senseObjLeft(obj_var)` detect whether object is at the left and store result in `obj_var`.
`void senseObjRight(obj_var)` detect whether object is at the right and store result in `obj_var`.
`void senseLine(left_var, right_var)` read line sensor states to `left_var` and `right_var`,
0 for no-line detected, 1 for line detected.
`void moveForward(left_speed, right_speed, time)` run Scribbler motors forward at `left_speed`, `right_speed` for the `time` duration. Motor speed is between 0 — 10, time is in millisecond. If time is 0, the motors will run continuously.
`void moveBackward(left_speed, right_speed, time)` run Scribbler motors backward at `left_speed`, `right_speed` for the `time` duration. Motor speed is between 0 — 10, time is in millisecond. If time is 0, the motors will run continuously.
`void moveLeft(left_speed, right_speed, time)` run Scribbler motors to left at `left_speed`, `right_speed` for the `time` duration. Motor speed is between 0 — 10, time is in millisecond.
`void moveRight(left_speed, right_speed, time)` run Scribbler motors to right at `left_speed`, `right_speed` for the `time` duration. Motor speed is between 0 — 10, time is in millisecond.

5 Acknowledgments

This work has been supported by NSF grant DUE-0737054.

References

- [1] Andy Lindsay. *What's a Microcontroller*. Parallax, Inc.
- [2] Parallax, Inc. *BASIC Stamp Reference Manual, version 2.1*.
- [3] Parallax, Inc. Scribbler Robot web page. <http://www.scribblerrobot.com/>.
- [4] Li Xu and Fred Martin. The chirp language specification. Technical Report TR-2005-003, Dept. of Computer Science, UMass Lowell.