

# The Chirp Language Specification

Li Xu and Fred Martin  
{xu, fredm}@cs.uml.edu

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

July 2005

## 1 Introduction

Chirp is a simple C-like language designed for programming educational robots, in particular, the Handy Cricket [1] based embedded robots. The Chirp language exposes the capabilities of the Handy Cricket virtual machine, as discussed by Martin et al [3]. This document describes the language features and specification of Chirp.

Chirp has a syntax similar to C, which allows programmers who are familiar with C/C++ or Java languages to quickly pick it up and start programming the cricket robots. However, as Chirp is designed to support embedded robot control rather than general-purpose computing, the language features of Chirp have been greatly simplified and only include data types and language constructs essential for writing cricket control programs. As a result, Chirp has a small language structure and provides a quick and yet extensible mechanism to develop cricket based robotic applications.

Chirp supports only two basic data types: integer and byte. Integer types are 16-bit numbers and byte types are 8-bit. Boolean values are treated similar to C: 0 represents logical false, and any non-zero values represent logical true. Chirp also supports single dimension arrays of the basic data types. There are no reference types such as pointers in Chirp.

Chirp only supports global variables and formal variables of functions. Functions do not have local variables. This restriction is consistent with the cricket run-time system which has small stacks for function invocation.

Computation in Chirp is realized through expression evaluation. Chirp supports C-like general expressions, in which variable and array references, function calls and constants are used as building blocks to construct more complex expressions, including arithmetic, relational and boolean expressions. Expression values are used in Chirp control statements.

Chirp supports a set of simplified control constructs. Allowable statements are assignments to global variables, if-else conditional statements, unconditional loops, loops with a calculated loop count and return statement to return from functions.

Chirp supports control abstraction through functions. Chirp functions are similar to C functions, which can take formal arguments, return a result or have void return type. The function body is a sequence of Chirp statements. Like C, the function with name “main” and void return type is designated as the entry point of Chirp programs. Each Chirp program must have a “main” program with the specified type signature.

To support robotic control interface, Chirp provides a special construct “interface”. An interface aggregates related I/O functions within a single namespace. The language run-time provides built-in interfaces, including System, System.Motor, System.Sensor, System.Bus, etc, which user can use directly in Chirp programs. User can also define new interfaces using the `interface` construct.

```

//Comment: example chirp program for motor and sensor control

trigger SensorChecker {
  (System.Sensor.getA() > 10) : {
    System.Sound.beep();
  }
}

void main() {
  loop {
    System.Motor.selectA();
    System.Motor.run();
    System.wait(10);
    System.Motor.stop();
    System.Motor.selectB();
    System.Motor.run();
    System.wait(10);
    System.Motor.stop();
  } with SensorChecker;
}

```

Figure 1: Example Chirp program

Another special construct of Chirp is “trigger”, which allows user to define a second thread of control in addition to the main control flow. Cricket robots allow user to define two threads of control flow: one foreground thread is the main control flow running on the processor, and the other is a background thread (“trigger” thread) which specifies a condition and an execution block. The processor will check the condition and if it is satisfied, the execution block will run in the trigger thread. The `trigger` construct allows user to specify the condition and execution block in Chirp. User can use the `with` statement to attach a trigger to a loop construct in the Chirp program.

Figure 1 shows an example Chirp program, which will repeatedly turn on and off the cricket A and B motor and check the A sensor reading in the background. If the sensor reading is above the threshold 10, the cricket will make a beep.

## 2 Chirp Syntax

Chirp is case sensitive, that is, `Foo` and `foo` are distinct names. Chirp keywords are reserved – the programmer cannot use a Chirp keyword as the name of a variable. The valid keywords are: `int`, `byte`, `if`, `else`, `with`, `loop`, `return`, `interface`, `trigger`.

The following characters have special meaning in a Chirp program:

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

The Chirp grammar in Figure ?? provides the usage of the special characters.

Chirp accepts C/C++ style comments: comments are delimited by matching `/*` and `*/`, or follow `//` until the end of the line.

<i>program</i>	→	( <i>declaration</i> )+
<i>declaration</i>	→	<i>variable_declaration</i>   <i>function_declaration</i>   <i>interface_declaration</i>   <i>trigger_declaration</i>
<i>variable_declaration</i>	→	<i>data_type</i> ID ( = INT )? ;   <i>data_type</i> ID [ INT ] ;
<i>data_type</i>	→	int   byte
<i>function_declaration</i>	→	<i>function_type</i> ID ( ( <i>formal</i> ( , <i>formal</i> )* )? ) <i>block</i>
<i>function_type</i>	→	<i>data_type</i>   void
<i>formal</i>	→	<i>data_type</i> ID
<i>block</i>	→	{ ( <i>statement</i> )* }
<i>statement</i>	→	<i>assignment_stmt</i> ;   <i>if_else_stmt</i>   <i>loop_stmt</i>   <i>return_stmt</i> ;   <i>function_call</i> ;   <i>block</i>
<i>assignment_stmt</i>	→	ID ( [ <i>expression</i> ] )? = <i>expression</i>
<i>if_else_stmt</i>	→	if ( <i>expression</i> ) <i>statement</i> ( <b>else</b> <i>statement</i> )?
<i>loop_stmt</i>	→	loop ( ( <i>expression</i> ) )? <i>block</i> ( <b>with</b> ID ; )?
<i>return_stmt</i>	→	<b>return</b> ( <i>expression</i> )?
<i>function_call</i>	→	ID ( . ID )* ( ( <i>expression</i> ( , <i>expression</i> )* )? )
<i>expression</i>	→	! <i>relational_expr</i>   <i>relational_expr</i> ( ( &&      ) <i>relational_expr</i> )*
<i>relational_expression</i>	→	<i>arithmetic_expression</i> ( ( ==   <   > ) <i>arithmetic_expression</i> )*
<i>arithmetic_expression</i>	→	<i>term</i> ( ( +   - ) <i>term</i> )*
<i>term</i>	→	<i>factor</i> ( ( *   /   % ) <i>factor</i> )*
<i>factor</i>	→	ID ( [ <i>expression</i> ] )?   INT   ( <i>expression</i> )   <i>function_call</i>
<i>interface_declaration</i>	→	<b>interface</b> ID ( . ID )* { ( <i>function_declaration</i> )+ }
<i>trigger_declaration</i>	→	<b>trigger</b> ID { ( <i>expression</i> ) : <i>block</i> }

Figure 2: Chirp grammar

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, function, interface and trigger names. Chirp identifiers are defined as follows:

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

Chirp constants are integers, which can be in either decimal or hexadecimal form, for example, 999 and 0xff.

Chirp supports the following boolean, relational and arithmetic operators:

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

The syntax of Chirp is described using the EBNF grammar shown in Figure ???. The terminal tokens are in the **typewriter** font.

## 3 Chirp Specification

### 3.1 Data Types

Chirp supports two basic data types: `int`, `byte`. The `int` is 16-bit signed integer value, and `byte` is 8-bit unsigned integer value. 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. Type conversions between the `int` and `byte` types follow the following rules:

- when converting `byte` to `int`, the high byte of the result will be extended to be 0;
- when converting `int` to `byte`, the high byte of the `int` value will be truncated and the low byte will be the `byte` result.

Similar to C, Chirp uses integer types to represent boolean values: 0 represents boolean false and any non-zero value represents boolean true. Chirp allows function return type to be `void`, indicating there is no return value for the function.

Chirp also supports one-dimension array type with array elements of the basic types.

### 3.2 Functions and Variables

User can define functions in Chirp similar to C. Functions can have multiple formal arguments of the basic data types and take no or single return value. Chirp supports only global variables and functions don't have on-stack local variables. This is due to the limited stack size of the cricket embedded processor.

### 3.3 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.

### 3.4 If-Else Statement

The if-else statement evaluates the 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.

### 3.5 Loop Statement

Chirp does not provide the general-purpose C-style *while* or *for* loop. Instead, Chirp provides the simple unconditional and the fixed-iteration count loop constructs. The `loop` statement will either execute the loop body in repetition, or evaluate the loop count expression and iterate the loop body by the specified loop count.

### 3.6 Return Statement

The return statement terminates the function and returns the control flow to the caller. For function with non-void return type, function result will be returned to its caller.

### 3.7 Function Invocation

Chirp functions are invoked by calling the function with the required arguments. Function return values can be used in expression evaluation. For functions defined in Chirp interfaces, function names should be qualified with appropriate interface names.

## 3.8 Block Statements

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

## 3.9 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 `==` for equality, `<` for less-than, and `>` for greater-than comparison. Chirp also supports common arithmetic operations. Result of non-void function call can also be used in expression evaluation.

## 3.10 Interfaces

User can use the interface construct to aggregate related I/O interface functions. Chirp interface consists of a list of interface functions. Interface functions can be called through the qualified interface and function names. Chirp interfaces do not have state data.

Chirp provides the following built-in interfaces which users can use directly in their programs.

- **System**

The System interface has the following functions:

`void wait(int time)` wait for the duration of `time`. Time is specified in tenth of second, eg., `wait(10)` will cause the cricket to wait and idle for one second.

`int random()` return a random integer value.

- **System.Motor**

System.Motor controls the A and B motors of the cricket:

`void selectA()` select the A motor as the current motor.

`void selectB()` select the B motor as the current motor.

`void selectBoth()` select both A and B motors.

`void setPower(int level)` set the power level of the selected motor. Valid argument range is 0 to 8, in which 8 is the highest.

`void setThisWay()` set the selected motor direction to the green LED on the cricket.

`void setThatWay()` set the selected motor direction to the red LED on the cricket.

`void runFor(int time)` run the selected motor for the duration of `time`.

`void runForever()` run the selected motor contiguously.

`void reverse()` reverse the direction of selected motor.

`void stop()` stop the selected motor.

- **System.Sensor**

System.Sensor controls the A and B sensors of the cricket:

`int getA()` get reading of A sensor.

`int getB()` get reading of B sensor.

- **System.Switch**

System.Switch controls the A and B switches of the cricket:

```
int getA() get on/off switch status of A switch.
```

```
int getB() get on/off switch status of B switch.
```

- **System.Sound**

System.Sound controls cricket sound interface:

```
void beep() make the cricket beep once.
```

```
void note(int pitch, int time) make the cricket to sound the pitch note for duration of time.
```

- **System.Timer**

System.Timer controls cricket timer interface:

```
int read() read the timer value.
```

```
void reset() reset the timer.
```

- **System.IR**

System.IR controls cricket IR communication:

```
void send(byte data) send data through the IR interface.
```

```
int hasNew() check whether IR has received new data.
```

```
int read() read the received data.
```

- **System.Bus**

System.Bus controls cricket bus:

```
void send(byte data) send data through the cricket bus.
```

```
int sendWithReply(byte data) send data onto bus and return the reply value.
```

### 3.11 Triggers

The cricket robot supports two threads of control: foreground main thread and background trigger thread. Users can specify a boolean condition and the execution code. The cricket will check against the condition when running the main thread. If the condition is true, the processor will switch to run the specified execution code in the second trigger thread. Chirp trigger construct allows user to define this second thread of control flow as a `trigger` attached to a `loop` statement. A trigger construct has a condition clause and statement block. If the trigger condition is true, the statement block will be executed in the trigger thread on the cricket. An example use of the trigger construct is shown in Figure 1.

## 4 Acknowledgments

We would like to thank Kareem Abu Zahra for his help with this work.

## References

- [1] Handy Cricket Documentation. <http://handyboard.com/cricket/>.
- [2] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice Hall, 1988.
- [3] F. Martin, B. Mihkak, and B. Silverman. Metacricket: A designer's kit for making computational devices. *IBM Systems Journal*, 39(3), 2000.