Nifty50 Language Reference Manual

Concise Programming Language

Aamir Sarwar Jahan (aj2599@columbia.edu)

Table of Contents

1. Nifty50 – The Core Concept:

Nifty50 is a concise programming language that is designed to present a computer program in blocks of 50 lines or less. The concept behind the apparent restriction is to divide the program into multiple modules with a unique top level module that presents an overview of the program. Nifty50 is targeted mainly for embedded systems development and has specific tools to be used for firmware. The programming tenets of the language can be summarized below:

1.1. Object-oriented: Nifty50 is a purely object oriented programming language where types are modeled as objects that include primitives. Basic types such as Integers, Characters, Strings, and Arrays can also be treated as objects.

1.2. Strong I/O support: Since, the language is designed to be used in embedded systems, Nifty50 supports convenient input scanning and output methods.

1.3. Versatile: The language allows a programmer to place custom algorithms in the program to perform computations in a powerful way.

1.4. Compact: Each module is restricted to 50 instructions but the program is not divided into modules because of increased length but for different functionality. Each module performs specific tasks or algorithms and is connected to the top level module.

The motivation of the new language is to reduce the time a programmer takes to write the top module that will form the skeleton of the main program and then work on the little details in the individual modules. It is imperative that the new language is easy to read and debug. The compiler for it will evaluate the syntax and semantics of each module that will aid in debugging the software as each module can be debugged independently.

2. Lexical Elements:

2.1. Whitespace: The language will convert all new line (\n), form feed, carriage return, and vertical tab characters as vertical whitespace. Tokens are separated by horizontal or vertical space of any length.

2.2. Identifiers: Identifiers are used for the identification of variables, methods and types. An identifier is a sequence of alphanumeric characters, uppercase and lowercase, and underscores. For Nifty50, the identifiers must start with a letter and cannot start with a numeric or a special character.

2.3. Keywords: Keywords cannot be used as identifiers and are reserved for use. Keywords are special set of strings that perform a specific function in the language. The language has the following keywords:

```
  module     perform      if        case        do        main
initialize     new        and        or        nand       nor
   until       stop      break      while      true      false
    port       cycle     return     create    methods
```

2.4. Operators: The operators in this language are listed below. It has a mix of unary and binary operators.

```
    +          -          *          /          ‘          =
    =/         <          >          <=         >=         %
    (          )         and        or         nor        nand
   xor        not
```

2.5. Integer Literals: An integer is a sequence of digits only and can have an optional unary operator at the beginning. Examples of integers are:-

```
   42         -42         00         999        -999
```

2.6. Float Literals: A float literal contains an integer or a fraction part or both seperated by a decimal ‘.’ point. It can also be expressed as an exponent ‘e’ and can be signed or unsigned.

```
  42.2       -42.3       32.0       32e4      -99.4449
```

2.7. Boolean Literals: A boolean literal can be expressed as true or false and also as its binary form.

```
     true       false        0          1
```

2.8. String Literals: A string literal consists of a single character or a sequence of characters in double quotes. It can have escape sequences such as linefeed, carriage return, backslash etc.

```
   “Hello World”          “ “          ‘f’      “COMS W4115”
“This\n next line”        ‘c’
```

2.9. Port Literals: Ports are an exclusively Nifty50 language specific literal and consists of an eight bit binary, decimal, hexadecimal, or an alphanumeric literal. Ports can be instantiated to read inputs on the pins of a microcontroller or output to the pins. Examples are given below:

```
  port C =00110010    port C = 0xAE        port C = ON
```

2.10 Comments: The language describes the comments within the /* and */ characters. Anything in between these set of characters is ignored by the compiler. Please note that comments do not count as part of the 50 lines for a module.

```
                /* This is a comment */
```

3. Semantics

3.1. Types and Variables

The variables in Nifty50 can be of any of the literal types discussed in the lexical elements section with variable names as *identifiers.* The types are static and will be known at compile time.

3.2. Modules

Modules are essential objects in the language. For every program, there is only one unique toplevel module object. Additional modules can be created and used with the toplevel module. Modules do not have to be of any type and can take single or multiple arguments and return a single value of any type. The modules are declared in the toplevel and the type compatibility is checked during compile time.

3.3. Methods

Methods are snippets of code that can be recycled to be used in different modules. A module can be a method in itself or have multiple methods in it and thus methods have the same property as a module except that a method can be nested inside a module. The scope of mehtods is global.

3.4 Targets

Targets are objects declared in a program that can have properties with global scope. They are modelled after hardware that the program can interact with in embedded systems applications. Each target can be configured to have ports. The ports are considered as either input-only, output-only, or both input-output capable.

3.5 Expressions and Statements

The language has both expressions and statements. Expressions generate values where statements can call an expression, a module, a method, or a target.

4. Syntax

The syntax for this language can be defined in a mix of essential and optional elements. Since, the number of instruction lines in a module is restricted to 50, each constructor syntax has linear constrains.

4.1 Declarations

4.1.1 Variable

A variable is assigned a type and a value in the same line

```
Identifier <type> = expression
```

4.1.2. Array

An array is assigned type, identifier and expression within the same line

```
Identifier <array of <type>> = expression
```

4.1.3. Method

All method objects are globally defined and are declared as

```
Method Identifier<arg_type 1, arg_type 2…> :
```

A method can return values but the return type is implicit and is not declared at the time of instrantiation.

4.1.4. Module

A module can be declared in the top level as

```
Create Module Identifier [return_type]
```

Modules must declare what they are returning at the time of instantiation.

4.1.5. Target

A target is an object with defined parameters and can be declared as:

```
Create Target Identifier <in: p1, p2, …> <out: p3, p4>
```

The I/O pins are defined within the < > and any pins that appear in both input and output list will be instantiated as I/O pin. A target pin is referenced as:

```
Identifier.p1 = expression
```

4.2 Conditional Structures

4.2.1. If Statements

The If statements are linearly arranged as:

```
If <statement;true=expression; false = expression>;
```

The first argument is the statement to be evaluated and the second expression is executed if statement is true. The third argument sets the expression performed if the statement is false.

4.2.2. For Statements

The For statements can be declared as:

```
For <condition_statement> do <execution statement>;
```

4.3 Others

4.3.1 Print and Scan

The print function can be invoked as:

```
Print “example string” ;
```

The Print function will print anything within the “ ” to the terminal window. C-style variables can be inserted into strings by using the %i, %d, %f, etc. symbol.

The scan function can be invoked as:

```
Scan <argument1; argument2> as [variable1; variable2] ;
```

4.3.2 The Semi-colon ‘;’

A semi-colon is used as termination to a statement or as a seperator for argument or return types.

THE END