Mattlang Language Definition

Matt Diesel (md639)
Matt Judge (mcj33)
Matt March (mdm64)

2nd June 2016

Abstract

This document provides a comprehensive technical specification of the Mattlang language, for use with the Mattlab circuit simulators. The language is currently in initial development, so is not stable and is likely to change in ways that break existing Mattlang files.

Although fully specified by this document, implementations are also encouraged to refer to the reference implementation contained within Mattlab and, where possible, mimic its behaviour in order to be compatible with existing tools.

Mattlang is a markup language used to describe digital logic circuits, which can then be simulated using Mattlab. The language is designed to be syntactically simple, whilst still being intuitive for developers and leaving options open for expansion of the language in future versions.

Version 1.0 25/05/16

- Initial Version

Version 1.1 02/06/16

- Added SIGGEN device type
- DTYPE behaviour changed to have non-zero hold time.

Contents

1 Lexical Elements

1.1 Source Code Representation

Mattlang files must be ISO 8859 encoded. UTF-8 files will be accepted provided they don't include a BOM or any unicode code points.

1.2 Case Insensitivity

All elements of the language are case insensitive. As a result the grammar will only define terminals in a single case.

1.3 Characters

The following terms are used to describe the fundamental sets of characters used in files:

```
letter         = "a" | "b" | "c" | "d" | "e" | "f" | "g"
                     | "h" | "i" | "j" | "k" | "l" | "m" | "n"
                     | "o" | "p" | "q" | "r" | "s" | "t" | "u"
                     | "v" | "w" | "x" | "y" | "z" ;
digit          = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" ;
bit            = "0" | "1" ;
whitespacechar = " " | "\t" | "\r" | "\n" | "\v" | "\f"  ;
```

In addition, the following extra sets are defined for use by the grammar:

```
alphanum       = letter | digit | "_" ;
```

1.4 Comments

Comments serve as markup documentation, and are treated as whitespace by the scanner. There are two forms:

1. Line comments, which start with the character sequence "//" and continue until a newline character.

2. General comments, which start with the character sequence "/*" and continue until the ending sequence "*/".

Comments cannot be started from inside a string literal or comment.

1.5 Identifiers

"Identifier" is the general term used to describe alphanumeric sequences that can represent keywords, device types or device names in the language. There is no limit on identifier length, other than memory constraints.

```
identifier     = letter , { alphanum } ;
```

1.6 Keywords

The following set of identifiers are keywords for the language, and cannot be used as device names, monitor aliases or device types:

```
dev monitor as import
```

1.7 Operators and Delimiters

The following character sequences represent special token types in the language.

```
= :   ; { } .
```

1.8 Number Literals

Mattlang uses numbers for clock periods as well as high/low logic states.

```
number         = digit , { digit } ;
```

For a number to represent a logic state it can only take the value of zero or one. Numbers can be in the range 0 to 2147483647. Attempting to use numbers higher than the maximum will result in a syntax error.

1.9 String Literals

A string literal represents a sequence of characters that should be used verbose throughout the program. There is no limit on string length, other than memory constraints.

A string starts with a double quote (quote character), and may contain any character, including new lines, non-printing characters and other characters not normally allowed in the source file, except for an unescaped quote character. The quote character is escaped by doubling it up, so to use the string `Hello, "World"!` The source code would show `"Hello, ""World""!"` .

```
string         = substr , { substr } ;
substr         = '"' , { (all characters - '"') } , '"' ;
```

1.10 Bitstream Literals

Bit stream literals represent arbitrary digital signals in time. There is no limit on bit stream length, other than memory constraints.

```
bitstream      = "$" , bit , { bit } ;
```

1.11 Device Names

Device names can be any identifier that is not a keyword or device type. The device name is set when it is first declared/defined, and printing of the device name follows the capitalization of the first use (including internal uses).

```
devicename     = identifier ;
```

1.12 Signal Names

Signal names comprise of a device name and optional pin.

```
signalname     = devicename , [ "." , pin ] ;
```

The pin not being set is only valid for devices with a single output, such as gates or switches. For other types, such as DTYPE latches, the output pin is not optional and must always be given.

1.13 Device Types

Device types are identifiers, used to represent classes of device.

```
typename       = letter , { letter } ;
```

The following identifiers are used for device types:

```
CLOCK SWITCH AND NAND OR NOR DTYPE XOR SIGGEN SELECT
```

Custom device types can be imported, by using a string literal as the device type. The Custom Devices section explains this functionality in more detail.

2 Statements

A circuit is made up of a set of statements which define the elements of the network.

```
file           = { statement } ;
statement      = definedevice | definemonitor | import ;
```

2.1 Device Definitions

A device definition statement gives the device name, its type if its being declared for the first time, and optionally the input pins and properties of the device. It takes the form:

```
definedevice   = "dev" , devicename , [ "=" , type ] , data ;
type           = typename | ( "import" , string ) ;
```

If a type is specified, then a warning is given if the device has been declared before with the same type, and an error if it has been previously declared with a different type. If the type is omitted then the device must have been declared previously.

2.1.1 Data Block

The data block follows a device definition, and can be empty (in which case it may be replaced with a semi-colon). It comprises of a set of zero or more key-value pairs specifying device inputs and properties.

```
data           = optionset | ";" ;
optionset      = "{" , { option } , "}" ;
option         = key , ":" , value , ";" ;
key            = identifier ;
value          = signalname | number | bitstream;
```

In an option pair, the key must reference a valid input pin or property of the device being defined. Valid input pins and properties are given in the Devices section. The value may be an existing signal in the network or a number, provided setting the key to that value is valid for that device.

2.2 Define Monitor Statement

Monitors allow signals within the network to be traced. Multiple monitor points can be defined in a single monitor statement, but this has no difference in behaviour to using separate statements.

```
definemonitor  = "monitor" , monitorset , ";" ;
monitorset     = monitor , { "," , monitor } ;
monitor        = signalname , [ "as" , signalname ] ;
```

Monitors can be shown to the user under an alias signal name, allowing signals in the network to be re-interpreted if they have other meanings. For example if designing a JK bistable circuit, the gates driving the output could be aliased to `Q` and `QBar`, to make the signal meaning clearer.

2.3 Import Statement

Import statements allow other Mattlang files to be included in the network as though they were in the same file. It takes a string filename, given relative to the file importing.

```
import         = "import" , string , ";" ;
```

The imported files use the same namespace, so care must be taken to avoid name conflicts. A file can never be imported twice.

3 Devices

A number of standard devices are predefined in Mattlang.

3.1 `CLOCK`

A clock device is a simple oscillator, given a period in terms of simulation cycles. It has no inputs, and one output switches at the set frequency. At the start of a simulation cycle, the clock internal counter is updated and checked against the period. An example of creating a clock with period 2 is:

```
dev CK1 = CLOCK {
        Period : 2;
}
```

3.1.1 `Period` [Property]

The clock period is a positive integer between 1 and 32767, giving the number of simulation cycles before the output switches. It is worth noting that this definition means period is half the time of the full clock cycle, so a period of 2 results in a trace of:

```
Simulation Cycle : 1  2  3  4  5  6  7  8  9 ...
CK1 Output       : 0  0  1  1  0  0  1  1  0 ...
```

3.2 SWITCH

A switch device is a user settable toggle between high and low, that has an initial value that is changed by the user through the Mattlab interface. It is included to allow user testing of circuits by modifying inputs. An example of a `SWITCH` definition, initially set low is:

```
dev SW1 = SWITCH {
        InitialValue : 0;
}
```

3.2.1 `InitialValue` [**Property**]

The `SWITCH.InitialValue` property is the state the switch begins in, and can be either of the logical states.

3.3 Standard Gates - `AND`, `NAND`, `OR`, `NOR`

The standard gates are all defined and used in the same way, but differ in the logical operation they perform. The logical operations are defined in terms of a required input level that all inputs must be to give an output level, with the rule being that:

$$Output = \begin{cases} Y, & \text{IFF } \mathrm{I}_N = X \quad \forall N \\ \neg Y, & \text{otherwise} \end{cases} \tag{1}$$

The device logical function is then defined by the following X and Y values.

Table 1: Standard Gates Operation Table

Device Type	X	Y
`AND`	1	1
`NAND`	1	0
`OR`	0	0
`NOR`	0	1

An example of creating a gate, in this case AND, with 3 inputs set to the example clock definition, the example switch definition and the high logical state is:

```
dev G1 = AND {
        I1 : CK1;
        I2 : SW1;
        I3 : 1;
}
```

3.3.1 `I1 - I16` [**Input Pin**]

The gate input pins are given by an I followed by a number from 1 to 16. These input pins can be assigned any valid signal in the network, or to a logical state to fix their level. The pins can be connected in any order, and do not need to use contiguous integers. At least one input pin must be assigned for each gate.

3.4 DTYPE

A DTYPE latch is a basic memory block that copies its input to its output on the rising edge of the clock. In its simplest form it is defined with a DATA and CLK input, in this case set to SW1 and CK1 respectively:

```
dev D1 = DTYPE {
        DATA : SW1;
        CLK : CK1;
}
```

3.4.1 DATA [Input Pin]

The input pin to the DTYPE device. Its value is used at the rising edge of the clock as the new output value. DTYPE devices have a non-zero hold time, and in the case that the DATA and CLK pins change in the same simulation cycle the output from both Q and QBAR pins is indeterminate.

3.4.2 CLK [Input Pin]

The clock/enable input pin, the rising edge of which updates the output value of the device.

3.4.3 SET, CLEAR [Input Pin]

Most DTYPE latches in electronics also include SET and CLEAR input pins, that force the output to a state regardless of CLK edge or DATA value. If either SET or CLEAR is set to high, the output follows the following truth table:

Table 2: DTYPE SET and CLEAR Truth Table

Inputs		Outputs	
SET	CLEAR	Q	QBAR
1	0	1	0
0	1	0	1
1	1	0	1

In Mattlang, the SET and CLEAR pins are optional, by default they will be both set low.

3.4.4 Q, QBAR [Output Pins]

Q is the output of the device, QBar its inverse except where both SET and CLEAR are set to high.

3.5 XOR

The eXclusive OR (XOR) gate takes exactly two inputs, and gives an output according to the following truth table:

Table 3: XOR Truth Table

I1	I2	Output
0	0	0
0	1	1
1	0	1
1	1	0

3.5.1 `I1`, `I2` [**Input Pin**]

The gate input pins are `I1` and `I2`. Unlike the other gate inputs, exactly two must be assigned input signals or logic states. The pins may be assigned in either order.

3.6 Imported

Imported devices are taken from other Mattlang file, using switches as inputs and monitors as outputs. They are defined using a string literal instead of a device type identifier in the definition statement. For example, if another Mattlang file called "JK.matt" defined a JK bistable using the following switches and monitors:

```
dev J = SWITCH { InitialValue: 0; }
dev K = SWITCH { InitialValue: 0; }
dev CLK = SWITCH { InitialValue: 0; }

// JK Bistable Logic
// G7 and G8 are the final gates setting the output

monitor G7 as Q, G8 as QBar;
```

This could then be used as a device, in this case toggling on the falling edge of `SW1`, would be defined as:

```
dev myJK = "JK.matt" {
        J : 1;
        K : 1;
        CLK : SW1;
}
```

The same imported device can be used multiple times in the same file. The imported file is itself a complete Mattlang file, and so can use all elements described in this specification.

3.6.1 Input Pins

The inputs pins of an imported device is the set of switches in the imported network. Input pins cannot be optional.

3.6.2 Output Pins

The output pins of the imported device that can be referenced by the importing file are any signals with a monitor, whose alias is just a device name. Importing a file that includes monitors with pin names will trigger a warning, and those monitor points will be ignored. Only signals being monitored can be used as outputs from the device.

3.7 `SELECT`

The `SELECT` device allows an input signal to switch the output between two other input signals. For example, an inverter for the signal `CK1` could be constructed by setting `HIGH` to low and `LOW` to high:

```
dev SL1 = SELECT {
        SW : CK1;
        HIGH : 0;
        LOW : 1;
}
```

3.7.1 `SW` [Input Pin]

The `SW` (SWitch) pin determines the input selected. If `SW` is high, then the HIGH input is selected, otherwise the `LOW` input is selected.

3.7.2 `HIGH`, `LOW` [Input Pin]

`HIGH` and `LOW` are the inputs that are selected between, depending on the value of `SW`. When selected their signal state is copied to the output pin.

3.8 `SIGGEN`

The SIGnal GENerator device maps an arbitrary input wave form to its output. At the same time as clocks are updated in the simulation cycle, the signal generator checks its internal counter against the period and updates the output to the next bit in the stream. If the end of the stream is reached then the stream resets to the start.

3.8.1 `SIG` [Property]

The waveform should be set to a bitstream literal, or a logical state for constant output.

3.8.2 `Period` [Property]

The period of the signal generator is the number of simulation cycles each digit of the signal is held for. This must be a non-zero, positive integer in the range 1 to 32767.

4 Simulation

The atomic unit of time for simulation is called a “simulation cycle”. Within each simulation cycle, the devices in the network evaluate their inputs to update their outputs in “machine cycles”.

4.1 Simulation Cycle

A simulation cycle begins by updating all clocks and siggen devices in the network. If the clock/siggen's counter has reached its period then the output toggles. Updating clock/siggen cycles includes updating all clock and signal generators in imported device networks.

The simulation cycle then executes machine cycles until the output of all devices in the network stabilises, or the number of machine cycles exceeds the limit. If it reaches the limit then it is assumed the network is oscillating and an error is reported.

The last stage of a simulation cycle is that all monitor points record the stabilised signal value.

4.2 Machine Cycle

Within a machine cycle, the devices in the network are evaluated according their inputs. The execution order of devices is implementation defined, with the exception that clocks and signal generators are always evaluated last.

When importing devices, a single machine cycle requires the imported network to stabilise fully before returning. As a result, a single machine cycle in the parent file can be thought to correspond to a simulation cycle in the imported network.

5 Errors

Mattlang errors are grouped into three types; syntactic, semantic and runtime. A file failing to match the grammar should produce a syntax error, if it passes the grammar but does not represent a valid network then semantic errors should be produced. Finally if it produces a valid network but cannot execute due to execution errors or unstable output it should report a runtime error. Implementations are required at the minimum to report errors mentioned in the specification. A network with one or more errors, as defined by the standard, is invalid and cannot be simulated.