anewin.com

DCD Script Reference

Table of Contents

Introduction

Five Wire provides the ability to acquire protocol data from your embedded microprocessor or FPGA. To make the contents of the protocol human readable, Five Wire provides decoders which separate the individual transactions and convert the bit stream into human readable numbers and labels.

To apply a decoder to a particular data stream the user must select the decoder to be applied such as RS232, I2C, SPI, LIN, VALUE, etc. Next the decoder must be configured to provide needed parameters such as which channels to use, serial baud rate, etc. Once the configuration is complete the decoder will analyze the selected channels and add transaction marks with human readable decode strings.

The decode process takes place in two steps. First the decoder scans the acquisition and identifies the transaction units. If the protocol is complex it will further subdivide the transaction into its constituent elements. In the second step the DCD script is applied to each transaction or transaction element to generate the text label that will describe the transaction. Each transaction is then marked with a box and label in the display.

The DCD script is a simple text file that can be edited with notepad or other simple text editors. The DCD script provides an opportunity for the user to customize the decode output text to be more descriptive of their usage of the protocol. For example, the DCD script can be modified to interpret numeric values that represent register addresses and display the register name instead of its address. This document describes the operation and syntax of DCD script files to support user customization.

DCD Script Working environment

As described in the introduction the DCD script receives transaction or transaction elements as its input. The script operates on these elements to generate a text string that describes the transaction. Each time the script is called, it is provided transaction data and the context of that data represents, for example address or data. Script commands within a context block operate on the data to generate the human readable output text which describes that part of the transaction.

The DCD script includes a number of commands dedicated to generating output text. These commands include `dec` and `hex` which display numeric values and `str` and `var` which display literal string values. Parameters to `dec` and `hex` allow bit field extraction and the number of digits to be displayed. For example, `hex 1 7,6,5,4` will display one hexadecimal digit extracted from of second nibble of the input data.

Execution control is provided by `if`, `else`, `endIf` and `using` blocks which contain one or more `case` statements. This allows different execution paths depending on bit fields within the input data. For example, if a transaction data bit defines the transaction to be a read or a write, a `using` block operating on the bit can be used to provide different coding for read and write transactions. Additional hierarchical `using` blocks can be included within the `case` block to further refine execution.

Two types of variable storage are provided. The `define` statement provides program constants and are initialized when the script is loaded. The `define` statements have precedence over simple variables but are limited to simple assignment. Variables are created when they are assigned a value during script execution. Variables provide the ability to capture the input value, output string, another variable, and if they contain a numeric value can be incremented and decremented providing a counting function. The decoder may also pass additional parameters to or from the script in a variable.

To provide additional flexibility, a script can override the context provided with the data using `setContext` and `deleteContext`. This supports different decode actions depending on prior received data.

Example DCD scripts

A number of standard scripts are provided in the Anewin/Five Wire/decode directory. One of the simplest scripts, rs2322.dcd, is the RS232 script shown here.

```
name Anewin_RS232_Decoder
context character
        var ascChar
        str :0x
        hex 2 7,6,5,4,3,2,1,0
endContext
```

The first line contains the name command which simply assigns a name to the script. The first stage RS232 decoder simply identifies individual characters and calls the DCD script with the context set to character and a variable *ascChar* set to a string containing the ASCII representation of the character.

The `var` command adds the character in the *ascChar* variable to the output text. The *str :0x* command adds the literal string ":0x" to the output text. The *hex 2 7,6,5,4,3,2,1,0* command adds a two-digit hexadecimal number built from bits 7..0 to the output text. The output text is then added to the mark for that character in the display.

A second version of this script, rs232_b.dcd, is provided which eliminates the display of the character's hexadecimal value.

```
name Anewin_RS232_b_Decoder
context character
        var ascChar
endContext
```

A more complex DCD is the I2C decoder. It contains several contexts which handle sub-elements of the transaction. The initialize context is called before any context to initialize define constants and variables.

```
name Anewin_I2C_Decoder

define sdiSdoSeparator ,_
define legend I2C_Decode

define idleClocks 2
define digits 2
define dataBits 7,6,5,4,3,2,1,0
define addrBits 6,5,4,3,2,1,0
```

```
context address
        str Addr:_
        hex digits addrBits
        str ,_
endContext

context readWrite
        if 0 0
                str Write_
        else
                str Read_
        endIf
endContext

context addressAck
        if 0 1
                str <NACK>
        endIf
endContext

context data
  hex digits dataBits
endContext
```

The *address* context is called with the I2C address field bits from the start byte. It generates the text “Addr: xx,” with xx replaced by 2 hexadecimal address digits. The *readWrite* context adds either “Read “ or “Write “ to the string based on the transaction read/write bit value.

Note that spaces are not allowed in literal strings. To include a space character replace it with underscore character ‘_’. All underscore characters are replaced with space characters when added to the output text.

The *addressAck* context will add “<NACK>” to the text if the transaction is not acknowledged but nothing if it is acknowledged. Finally, the *data* context is called for each data byte in the transaction. The decoder automatically inserts the contents of the *sdiSdoSeparator* variable between each byte value.

Here is an example output: “Addr: 68, Read 47, 31, 20, 02, 31, 07, 18”. This was captured from an RTC chip as it returned its current date time values July 31, 2018, on Tuesday, at the time 20:31:47. You can imagine that knowing the transaction contains a date/time, you could enhance the DCD script to convert the data bytes to a human readable date by including byte counting and using variables to capture each data field and then processing the variables on the last data byte to produce the date time text “RTC read: July 31, 2018 at 20:31:47”.

The LIN decoder DCD is more complex and includes nested using statements to separate the components of the protocol. It uses comment lines which start with the ‘//’. Otherwise it uses the same elements already described in these examples.

Syntax descriptions

Basic rules

DCD scripts are case sensitive

Empty lines or lines starting with the comment command '//' are ignored

Literal strings *literal* may contain any printable character but not a space character.

Underscore characters are converted to space characters during text generation

Binary numbers *binaryVal* are a sequence of the characters 0, 1, x or X (don't care)

Bit selectors *bitSelect* are a sequence of comma separated numbers

Numeric values can be decimal or hexadecimal. Hexadecimal values must be prefixed with 0x or 0X for example 0xA5 or 0XA5

Variables and defines are all of type string. If the use of the variable expects a numeric value, the string is converted to a number before it is used

Variable names <varName> must start with an alpha character and may not include spaces.

The define names have the same rules as variable names

Commands and parameters are separated by any combinations of space and tab characters

Commands and their associated parameters must exist on a single line

Commands that generate text output

Note: If the syntax allows multiple types for a parameter the possible types are enclosed in curly brackets separated by a bar { a | b } where either a or b can be used in that position. Parameters that are optional are surrounded by square brackets [a].

dec

```
dec {digits | <varName>} {bitSelect | <varName>} [<value varName>]
```

Command dec: Add a decimal numeric representation to the output

Parameter digits: The number of character digits to output. May be supplied by a define/variable.

Parameter bitSelect: The set of bits used in the numeric value. May be supplied by a define/variable.

Optional Parameter value: If supplied, the value comes from the define/variable. Otherwise it is the input value.

hex

`hex {digits | <varName>} {bitSelect | <varName>} [<value varName>]`

Command `hex`: Same as `dec` except the number is formatted as a hexadecimal value

str

`str litString`

Command `str`: Add a literal string to the output

Parameter litString: The string of characters that are added to the output

var

`var <varName>`

Command `var`: Add the string contents of a variable to the output

Parameter varName: The name of an existing variable

Commands that control execution

Context

`context contextName`

Command `context`: Starts a context block

Parameter contextName: A literal string that identifies the context

endContext

`endContext`

Command `endContext`: Terminates a context block

setContext

`setContext contextName`

Command `setContext`: Overrides the context provided by the decoder

Parameter contextName: A literal string that identifies the context

Usage: If a local context is set it will be used on each following execution of the script until it is deleted

deleteContext

`deleteContext`

Command `deleteContext`: Removes the context override

if, ifNot

`if {bitSelect | <varName>} {testBits | <varName>} [<value varName>]`

Command `if`: Starts a conditional execution block which is executed if the test result returns true

Parameter bitSelect: Determines which input bits will be tested

Parameter testBits: The binaryVal that will be tested against the input

Optional parameter value: If provided, input data will come from the variable

else

`else`

Command `else`: Terminates an if block and starts the else block. The else block is executed when the test result is false

endIf

`endIf`

Command `endIf`: Terminates an if or else block

using

`using {bitSelect | <varName>} [<value varName>]`

Command `using`: Starts a using block which can contain one or more case commands

Parameter bitSelect: Selects which input bits are used in case evaluation. May be supplied by a variable

Optional parameter value: If supplied, the contents of the variable will be used as the input

Usage: Only one case block (if any) will be executed within the using block

endUsing

`endUsing`

Command `endUsing`: Terminates a using block

case

`case {value | <varName>}`

Command `case`: Starts a case block which will be executed if its parameter matches the input

Parameter value: A binaryVal that is compared with the input value

endCase

`endCase`

Command `endCase`: Terminates the case block

Commands that change the script state

name

`name litString`

Command `name`: Sets decoder name

Parameter litString: The name of this DCD script

define

`define <varName> litString`

Command `define`: Creates or updates a define variable

Parameter varName: The identifier of this define

Parameter litString: The value to assign to this variable

Usage: All defines are evaluated when the script is loaded. The name space for defines and variables are separate – a define and a variable can have the same name. However, the define name space is evaluated before the variable name space so the define value will be returned when the identifiers are the same.

save

`save <varName> litString`

Command `save`: Creates or updates a variable

Parameter varName: The identifier of this variable

Parameter litString: The value to assign to this variable

Usage: Variables must be created before they are referenced in the script.

saveInput

`saveInput <varName> [bitSelect | <varName>]`

Command `saveInput`: Creates or updates a variable and sets its value to the transaction input data value

Parameter varName: The identifier of this variable

Parameter bitSelect: Determines which bits are included from the data input. May be provided by a define or variable.

saveOutput

```
saveOutput <varName>
```

Command `saveOutput`: Creates or updates a variable and sets its value to the currently generated output text

Parameter varName: The identifier of this variable

copy

```
copy <destination varName> <source varName>
```

Command `copy`: Copies the value of the source variable to the destination variable

Parameter destination varName: The variable that will be overwritten

Parameter source varName: The variable that will supply the data to be copied

decrement

```
decrement <varName>
```

Command `decrement`: Reduces the numeric value of the variable by 1 but not lower than zero

Parameter varName: The identifier of the variable to decrement

Usage: The variable must contain a positive decimal literal that can be converted to a string

increment

```
increment <varName>
```

Command `increment`: Increases the numeric value of the variable by 1

Parameter varName: The identifier of the variable to decrement

add

`add <varName> {value | <varName>}`

Command `add`: Adds value to the variable (var += value)

Parameter varName: The identifier of the variable to operate on

Parameter value: The value added to the variable

Usage: The variable and value must be numeric unsigned values

sub

`sub <varName> {value | <varName>}`

Command `sub`: Subtracts value to the variable (var -= value)

Parameter varName: The identifier of the variable to operate on

Parameter value: The value subtracted to the variable

Usage: The variable and value must be numeric unsigned values

rshift

`rshift <varName> {value | <varName>}`

Command `rshift`: Right shift the variable (var >>= value)

Parameter varName: The identifier of the variable to operate on

Parameter value: The number of bit positions to shift 0-64

Usage: The variable and value must be numeric unsigned values

lshift

`lshift <varName> {value | <varName>}`

Command `lshift`: Left shift the variable (var <<= value)

Parameter varName: The identifier of the variable to operate on

Parameter value: The number of bit positions to shift 0-64

Usage: The variable and value must be numeric unsigned values

and

```
and <varName> {value | <varName>}
```

Command `and`: Bitwise AND the variable (var &= value)

Parameter varName: The identifier of the variable to operate on

Parameter value: The value that is AND’ed with the variable

Usage: The variable and value must be numeric unsigned values

or

```
or <varName> {value | <varName>}
```

Command `or`: Bitwise OR the variable (var |= value)

Parameter varName: The identifier of the variable to operate on

Parameter value: The value that is OR’ed with the variable

Usage: The variable and value must be numeric unsigned values

Common Resources

Initialize context

Before the decoder sends the first transaction data it will call the DCD script context *initialize* if it exists. This allows initialization of defines and variables.

Reserved variables

`showMark` :: {true | false}

This variable is initialized to true. The DCD script can update its value to true or false. When the DCD script completes the decoder inspects this variable to determine if the transaction mark should be shown. If false, the mark will not be shown.

`input` :: unsigned long integer

This variable is set to the transaction data value each time the DCD script is called. It may be referenced to access the data value in any context except the initialize context.

`localContext` :: unsigned integer

This variable is reserved and should not be directly accessed by a script. It is created by the `setContext` command and destroyed by the `deleteContext` command.

`context none`

This context is reserved and should not be used in a script.

`legend`

If the define legend exists, its value will be displayed below the first mark in the acquistion

Decoder Specific Interface

RS232

variables

ascChar: ASCII string representation of the current character

context

character: A transaction with contents of single character

I2C

define

idleClocks: The number of clock periods expected between transactions

sdiSdoSeparator: A string that will be inserted between data bytes

context

address: A transaction element containing the I2C device address – 6 bits

readWrite: A transaction element containing the I2C read/write bit

addressAck: A transaction element containing the I2C address acknowledge bit

data: A transaction element containing an I2C data byte

SPI

variable

bits: Bit selector for transaction

digits: Number of characters needed to display a transaction value

sdiSdoSeparator: String inserted between data bytes

context

sdi: Value of the serial data in

sdo: Value of the serial data out

LIN

variables

pidValue

pidPrityErrorFlags

numberOfDataBytes

hasDataFlag

enhancedChecksum

expectedChecksum

checksumFailFlag

context

pidValue

data

checksumValue

VALUE

variables

channels: List of channels used

extendMark: {true | false} Indicates that multiple values should be combined to a single mark textOffset: Text label offset in pixels

context

value