NAME

cdeftutorial − Alex van den Bogaerdt's CDEF tutorial

DESCRIPTION

Intention of this document: to provide some examples of the commonly used parts of RRDtool's CDEF language.

If you think some important feature is not explained properly, and if adding it to this document would benefit most users, please do ask me to add it. I will then try to provide an answer in the next release of this tutorial. No feedback equals no changes! Additions to this document are also welcome. — Alex van den Bogaerdt <alex@vandenbogaerdt.nl>

Why this tutorial?

One of the powerful parts of RRDtool is its ability to do all sorts of calculations on the data retrieved from its databases. However, RRDtool's many options and syntax make it difficult for the average user to understand. The manuals are good at explaining what these options do; however they do not (and should not) explain in detail why they are useful. As with my RRDtool tutorial: if you want a simple document in simple language you should read this tutorial. If you are happy with the official documentation, you may find this document too simple or even boring. If you do choose to read this tutorial, I also expect you to have read and fully understand my other tutorial.

More reading

If you have difficulties with the way I try to explain it please read Steve Rader's rpntutorial. It may help you understand how this all works.

What are CDEFs?

When retrieving data from an RRD, you are using a "DEF" to work with that data. Think of it as a variable that changes over time (where time is the x−axis). The value of this variable is what is found in the database at that particular time and you can't do any modifications on it. This is what CDEFs are for: they takes values from DEFs and perform calculations on them.

Syntax

```
DEF:var_name_1=some.rrd:ds_name:CF
CDEF:var_name_2=RPN_expression
```

You first define "var_name_1" to be data collected from data source "ds_name" found in RRD "some.rrd" with consolidation function "CF".

Assume the ifInOctets SNMP counter is saved in mrtg.rrd as the DS "in". Then the following DEF defines a variable for the average of that data source:

```
DEF:inbytes=mrtg.rrd:in:AVERAGE
```

Say you want to display bits per second (instead of bytes per second as stored in the database.) You have to define a calculation (hence "CDEF") on variable "inbytes" and use that variable (inbits) instead of the original:

```
CDEF:inbits=inbytes,8,*
```

This tells RRDtool to multiply inbytes by eight to get inbits. I'll explain later how this works. In the graphing or printing functions, you can now use inbits where you would use inbytes otherwise.

Note that the variable name used in the CDEF (inbits) must not be the same as the variable named in the DEF (inbytes)!

RPN-expressions

RPN is short-hand for Reverse Polish Notation. It works as follows. You put the variables or numbers on a stack. You also put operations (things-to-do) on the stack and this stack is then processed. The result will be placed on the stack. At the end, there should be exactly one number left: the outcome of the series of operations. If there is not exactly one number left, RRDtool will complain loudly.

Above multiplication by eight will look like:

1. Start with an empty stack
2. Put the content of variable inbytes on the stack
3. Put the number eight on the stack
4. Put the operation multiply on the stack
5. Process the stack
6. Retrieve the value from the stack and put it in variable inbits

We will now do an example with real numbers. Suppose the variable inbytes would have value 10, the stack would be:

1. ||
2. |10|
3. |10|8|
4. |10|8|*|
5. |80|
6. ||

Processing the stack (step 5) will retrieve one value from the stack (from the right at step 4). This is the operation multiply and this takes two values off the stack as input. The result is put back on the stack (the value 80 in this case). For multiplication the order doesn't matter, but for other operations like subtraction and division it does. Generally speaking you have the following order:

```
y = A - B  -->  y=minus(A,B)  -->  CDEF:y=A,B,-
```

This is not very intuitive (at least most people don't think so). For the function f(A,B) you reverse the position of "f", but you do not reverse the order of the variables.

Converting your wishes to RPN

First, get a clear picture of what you want to do. Break down the problem in smaller portions until they cannot be split anymore. Then it is rather simple to convert your ideas into RPN.

Suppose you have several RRDs and would like to add up some counters in them. These could be, for instance, the counters for every WAN link you are monitoring.

You have:

```
router1.rrd with link1in link2in
router2.rrd with link1in link2in
router3.rrd with link1in link2in
```

Suppose you would like to add up all these counters, except for link2in inside router2.rrd. You need to do:

(in this example, "router1.rrd:link1in" means the DS link1in inside the RRD router1.rrd)

```
router1.rrd:link1in
router1.rrd:link2in
router2.rrd:link1in
router3.rrd:link1in
router3.rrd:link2in
--------------------   +
(outcome of the sum)
```

As a mathematical function, this could be written:

`add(router1.rrd:link1in , router1.rrd:link2in , router2.rrd:link1in , router3.rrd:link1in , router3.rrd:link2.in)`

With RRDtool and RPN, first, define the inputs:

```
DEF:a=router1.rrd:link1in:AVERAGE
DEF:b=router1.rrd:link2in:AVERAGE
DEF:c=router2.rrd:link1in:AVERAGE
DEF:d=router3.rrd:link1in:AVERAGE
DEF:e=router3.rrd:link2in:AVERAGE
```

Now, the mathematical function becomes: `add(a,b,c,d,e)`

In RPN, there's no operator that sums more than two values so you need to do several additions. You add a and b, add c to the result, add d to the result and add e to the result.

```
push a:         a     stack contains the value of a
push b and add: b,+   stack contains the result of a+b
push c and add: c,+   stack contains the result of a+b+c
push d and add: d,+   stack contains the result of a+b+c+d
push e and add: e,+   stack contains the result of a+b+c+d+e
```

What was calculated here would be written down as:

```
( ( ( (a+b) + c) + d) + e) >
```

This is in RPN: `CDEF:result=a,b,+,c,+,d,+,e,+`

This is correct but it can be made more clear to humans. It does not matter if you add a to b and then add c to the result or first add b to c and then add a to the result. This makes it possible to rewrite the RPN into `CDEF:result=a,b,c,d,e,+,+,+,+` which is evaluated differently:

```
push value of variable a on the stack: a
push value of variable b on the stack: a b
push value of variable c on the stack: a b c
push value of variable d on the stack: a b c d
push value of variable e on the stack: a b c d e
push operator + on the stack:          a b c d e +
and process it:                        a b c P    (where P == d+e)
push operator + on the stack:          a b c P +
and process it:                        a b Q      (where Q == c+P)
push operator + on the stack:          a b Q +
and process it:                        a R        (where R == b+Q)
push operator + on the stack:          a R +
and process it:                        S          (where S == a+R)
```

As you can see the RPN expression `a,b,c,d,e,+,+,+,+,+` will evaluate in `((((d+e)+c)+b)+a)` and it has the same outcome as `a,b,+,c,+,d,+,e,+`. This is called the commutative law of addition, but you may forget this right away, as long as you remember what it means.

Now look at an expression that contains a multiplication:

First in normal math: `let result = a+b*c`. In this case you can't choose the order yourself, you have to start with the multiplication and then add a to it. You may alter the position of b and c, you must not alter the position of a and b.

You have to take this in consideration when converting this expression into RPN. Read it as: "Add the outcome of b*c to a" and then it is easy to write the RPN expression: `result=a,b,c,*,+` Another expression that would return the same: `result=b,c,*,a,+`

In normal math, you may encounter something like "a*(b+c)" and this can also be converted into RPN. The parenthesis just tell you to first add b and c, and then multiply a with the result. Again, now it is easy to write it in RPN: `result=a,b,c,+,*`. Note that this is very similar to one of the expressions in the previous paragraph, only the multiplication and the addition changed places.

When you have problems with RPN or when RRDtool is complaining, it's usually a good thing to write down the stack on a piece of paper and see what happens. Have the manual ready and pretend to be RRDtool. Just do all the math by hand to see what happens, I'm sure this will solve most, if not all, problems you encounter.

Some special numbers

The unknown value

Sometimes collecting your data will fail. This can be very common, especially when querying over busy links. RRDtool can be configured to allow for one (or even more) unknown value(s) and calculate the missing update. You can, for instance, query your device every minute. This is creating one so called PDP or primary data point per minute. If you defined your RRD to contain an RRA that stores 5−minute values, you need five of those PDPs to create one CDP (consolidated data point). These PDPs can become unknown in two cases:

1. The updates are too far apart. This is tuned using the ‘‘heartbeat’’ setting.
2. The update was set to unknown on purpose by inserting no value (using the template option) or by using ‘‘U’’ as the value to insert.

When a CDP is calculated, another mechanism determines if this CDP is valid or not. If there are too many PDPs unknown, the CDP is unknown as well. This is determined by the xff factor. Please note that one unknown counter update can result in two unknown PDPs! If you only allow for one unknown PDP per CDP, this makes the CDP go unknown!

Suppose the counter increments with one per second and you retrieve it every minute:

```
counter value    resulting rate
10'000
10'060            1; (10'060-10'000)/60 == 1
10'120            1; (10'120-10'060)/60 == 1
unknown           unknown; you don't know the last value
10'240            unknown; you don't know the previous value
10'300            1; (10'300-10'240)/60 == 1
```

If the CDP was to be calculated from the last five updates, it would get two unknown PDPs and three known PDPs. If xff would have been set to 0.5 which by the way is a commonly used factor, the CDP would have a known value of 1. If xff would have been set to 0.2 then the resulting CDP would be unknown.

You have to decide the proper values for heartbeat, number of PDPs per CDP and the xff factor. As you can see from the previous text they define the behavior of your RRA.

Working with unknown data in your database

As you have read in the previous chapter, entries in an RRA can be set to the unknown value. If you do calculations with this type of value, the result has to be unknown too. This means that an expression such as `result=a,b,+` will be unknown if either a or b is unknown. It would be wrong to just ignore the unknown value and return the value of the other parameter. By doing so, you would assume ‘‘unknown’’ means ‘‘zero’’ and this is not true.

There has been a case where somebody was collecting data for over a year. A new piece of equipment was installed, a new RRD was created and the scripts were changed to add a counter from the old database and a counter from the new database. The result was disappointing, a large part of the statistics seemed to have vanished mysteriously ... They of course didn’t, values from the old database (known values) were added to values from the new database (unknown values) and the result was unknown.

In this case, it is fairly reasonable to use a CDEF that alters unknown data into zero. The counters of the device were unknown (after all, it wasn’t installed yet!) but you know that the data rate through the device had to be zero (because of the same reason: it was not installed).

There are some examples below that make this change.

Infinity

Infinite data is another form of a special number. It cannot be graphed because by definition you would never reach the infinite value. You can think of positive and negative infinity depending on the position relative to zero.

RRDtool is capable of representing (−not− graphing!) infinity by stopping at its current maximum (for positive infinity) or minimum (for negative infinity) without knowing this maximum (minimum).

Infinity in RRDtool is mostly used to draw an AREA without knowing its vertical dimensions. You can think of it as drawing an AREA with an infinite height and displaying only the part that is visible in the current graph. This is probably a good way to approximate infinity and it sure allows for some neat tricks. See below for examples.

Working with unknown data and infinity

Sometimes you would like to discard unknown data and pretend it is zero (or any other value for that matter) and sometimes you would like to pretend that known data is unknown (to discard known-to-be-wrong data). This is why CDEFs have support for unknown data. There are also examples available that show unknown data by using infinity.

Some examples

Example: using a recently created RRD

You are keeping statistics on your router for over a year now. Recently you installed an extra router and you would like to show the combined throughput for these two devices.

If you just add up the counters from router.rrd and router2.rrd, you will add known data (from router.rrd) to unknown data (from router2.rrd) for the bigger part of your stats. You could solve this in a few ways:

- While creating the new database, fill it with zeros from the start to now. You have to make the database start at or before the least recent time in the other database.
- Alternatively, you could use CDEF and alter unknown data to zero.

Both methods have their pros and cons. The first method is troublesome and if you want to do that you have to figure it out yourself. It is not possible to create a database filled with zeros, you have to put them in manually. Implementing the second method is described next:

What we want is: ‘‘if the value is unknown, replace it with zero’’. This could be written in pseudo-code as: if (value is unknown) then (zero) else (value). When reading the rrdgraph manual you notice the ‘‘UN’’ function that returns zero or one. You also notice the ‘‘IF’’ function that takes zero or one as input.

First look at the ‘‘IF’’ function. It takes three values from the stack, the first value is the decision point, the second value is returned to the stack if the evaluation is ‘‘true’’ and if not, the third value is returned to the stack. We want the ‘‘UN’’ function to decide what happens so we combine those two functions in one CDEF.

Lets write down the two possible paths for the ‘‘IF’’ function:

```
if true  return a
if false return b
```

In RPN: `result=x,a,b,IF` where ‘‘x’’ is either true or false.

Now we have to fill in ‘‘x’’, this should be the ‘‘(value is unknown)’’ part and this is in RPN: `result=value,UN`

We now combine them: `result=value,UN,a,b,IF` and when we fill in the appropriate things for ‘‘a’’ and ‘‘b’’ we’re finished:

`CDEF:result=value,UN,0,value,IF`

You may want to read Steve Rader’s RPN guide if you have difficulties with the way I explained this last example.

If you want to check this RPN expression, just mimic RRDtool behavior:

```
For any known value, the expression evaluates as follows:
CDEF:result=value,UN,0,value,IF  (value,UN) is not true so it becomes 0
CDEF:result=0,0,value,IF         "IF" will return the 3rd value
CDEF:result=value                The known value is returned

For the unknown value, this happens:
CDEF:result=value,UN,0,value,IF  (value,UN) is true so it becomes 1
```

```
CDEF:result=1,0,value,IF          "IF" sees 1 and returns the 2nd value
CDEF:result=0                     Zero is returned
```

Of course, if you would like to see another value instead of zero, you can use that other value.

Eventually, when all unknown data is removed from the RRD, you may want to remove this rule so that unknown data is properly displayed.

Example: better handling of unknown data, by using time

The above example has one drawback. If you do log unknown data in your database after installing your new equipment, it will also be translated into zero and therefore you won't see that there was a problem. This is not good and what you really want to do is:

- If there is unknown data, look at the time that this sample was taken.
- If the unknown value is before time xxx, make it zero.
- If it is after time xxx, leave it as unknown data.

This is doable: you can compare the time that the sample was taken to some known time. Assuming you started to monitor your device on Friday September 17, 1999, 00:35:57 MET DST. Translate this time in seconds since 1970–01–01 and it becomes 937'521'357. If you process unknown values that were received after this time, you want to leave them unknown and if they were "received" before this time, you want to translate them into zero (so you can effectively ignore them while adding them to your other routers counters).

Translating Friday September 17, 1999, 00:35:57 MET DST into 937'521'357 can be done by, for instance, using gnu date:

```
date -d "19990917 00:35:57" +%s
```

You could also dump the database and see where the data starts to be known. There are several other ways of doing this, just pick one.

Now we have to create the magic that allows us to process unknown values different depending on the time that the sample was taken. This is a three step process:

1. If the timestamp of the value is after 937'521'357, leave it as is.
2. If the value is a known value, leave it as is.
3. Change the unknown value into zero.

Lets look at part one:

```
 if (true) return the original value
```

We rewrite this:

```
 if (true) return "a"
 if (false) return "b"
```

We need to calculate true or false from step 1. There is a function available that returns the timestamp for the current sample. It is called, how surprisingly, "TIME". This time has to be compared to a constant number, we need "GT". The output of "GT" is true or false and this is good input to "IF". We want "if (time > 937521357) then (return a) else (return b)".

This process was already described thoroughly in the previous chapter so lets do it quick:

```
if (x) then a else b
   where x represents "time>937521357"
   where a represents the original value
   where b represents the outcome of the previous example

time>937521357       --> TIME,937521357,GT

if (x) then a else b --> x,a,b,IF
```

```
substitute x         --> TIME,937521357,GT,a,b,IF
substitute a         --> TIME,937521357,GT,value,b,IF
substitute b         --> TIME,937521357,GT,value,value,UN,0,value,IF,IF
```

We end up with: `CDEF:result=TIME,937521357,GT,value,value,UN,0,value,IF,IF`

This looks very complex, however, as you can see, it was not too hard to come up with.

Example: Pretending weird data isn't there

Suppose you have a problem that shows up as huge spikes in your graph. You know this happens and why, so you decide to work around the problem. Perhaps you're using your network to do a backup at night and by doing so you get almost 10mb/s while the rest of your network activity does not produce numbers higher than 100kb/s.

There are two options:

1. If the number exceeds 100kb/s it is wrong and you want it masked out by changing it into unknown.
2. You don't want the graph to show more than 100kb/s.

Pseudo code: if (number > 100) then unknown else number or Pseudo code: if (number > 100) then 100 else number.

The second "problem" may also be solved by using the rigid option of RRDtool graph, however this has not the same result. In this example you can end up with a graph that does autoscaling. Also, if you use the numbers to display maxima they will be set to 100kb/s.

We use "IF" and "GT" again. "if (x) then (y) else (z)" is written down as "CDEF:result=x,y,z,IF"; now fill in x, y and z. For x you fill in "number greater than 100kb/s" becoming "number,100000,GT" (kilo is 1'000 and b/s is what we measure!). The "z" part is "number" in both cases and the "y" part is either "UNKN" for unknown or "100000" for 100kb/s.

The two CDEF expressions would be:

```
CDEF:result=number,100000,GT,UNKN,number,IF
CDEF:result=number,100000,GT,100000,number,IF
```

Example: working on a certain time span

If you want a graph that spans a few weeks, but would only want to see some routers' data for one week, you need to "hide" the rest of the time frame. Don't ask me when this would be useful, it's just here for the example :)

We need to compare the time stamp to a begin date and an end date. Comparing isn't difficult:

```
TIME,begintime,GE
TIME,endtime,LE
```

These two parts of the CDEF produce either 0 for false or 1 for true. We can now check if they are both 0 (or 1) using a few IF statements but, as Wataru Satoh pointed out, we can use the "*" or "+" functions as logical AND and logical OR.

For "*", the result will be zero (false) if either one of the two operators is zero. For "+", the result will only be false (0) when two false (0) operators will be added. Warning: *any* number not equal to 0 will be considered "true". This means that, for instance, "−1,1,+" (which should be "true or true") will become FALSE ... In other words, use "+" only if you know for sure that you have positive numbers (or zero) only.

Let's compile the complete CDEF:

```
DEF:ds0=router1.rrd:AVERAGE
CDEF:ds0modified=TIME,begintime,GT,TIME,endtime,LE,*,ds0,UNKN,IF
```

This will return the value of ds0 if both comparisons return true. You could also do it the other way around:

```
DEF:ds0=router1.rrd:AVERAGE
CDEF:ds0modified=TIME,begintime,LT,TIME,endtime,GT,+,UNKN,ds0,IF
```

This will return an UNKNOWN if either comparison returns true.

Example: You suspect to have problems and want to see unknown data.

Suppose you add up the number of active users on several terminal servers. If one of them doesn't give an answer (or an incorrect one) you get "NaN" in the database ("Not a Number") and NaN is evaluated as Unknown.

In this case, you would like to be alerted to it and the sum of the remaining values is of no value to you.

It would be something like:

```
    DEF:users1=location1.rrd:onlineTS1:LAST
    DEF:users2=location1.rrd:onlineTS2:LAST
    DEF:users3=location2.rrd:onlineTS1:LAST
    DEF:users4=location2.rrd:onlineTS2:LAST
    CDEF:allusers=users1,users2,users3,users4,+,+,+
```

If you now plot allusers, unknown data in one of users1..users4 will show up as a gap in your graph. You want to modify this to show a bright red line, not a gap.

Define an extra CDEF that is unknown if all is okay and is infinite if there is an unknown value:

```
    CDEF:wrongdata=allusers,UN,INF,UNKN,IF
```

"allusers,UN" will evaluate to either true or false, it is the (x) part of the "IF" function and it checks if allusers is unknown. The (y) part of the "IF" function is set to "INF" (which means infinity) and the (z) part of the function returns "UNKN".

The logic is: if (allusers == unknown) then return INF else return UNKN.

You can now use AREA to display this "wrongdata" in bright red. If it is unknown (because allusers is known) then the red AREA won't show up. If the value is INF (because allusers is unknown) then the red AREA will be filled in on the graph at that particular time.

```
  AREA:allusers#0000FF:combined user count
  AREA:wrongdata#FF0000:unknown data
```

Same example useful with STACKed data:

If you use stack in the previous example (as I would do) then you don't add up the values. Therefore, there is no relationship between the four values and you don't get a single value to test. Suppose users3 would be unknown at one point in time: users1 is plotted, users2 is stacked on top of users1, users3 is unknown and therefore nothing happens, users4 is stacked on top of users2. Add the extra CDEFs anyway and use them to overlay the "normal" graph:

```
  DEF:users1=location1.rrd:onlineTS1:LAST
  DEF:users2=location1.rrd:onlineTS2:LAST
  DEF:users3=location2.rrd:onlineTS1:LAST
  DEF:users4=location2.rrd:onlineTS2:LAST
  CDEF:allusers=users1,users2,users3,users4,+,+,+
  CDEF:wrongdata=allusers,UN,INF,UNKN,IF
  AREA:users1#0000FF:users at ts1
  STACK:users2#00FF00:users at ts2
  STACK:users3#00FFFF:users at ts3
  STACK:users4#FFFF00:users at ts4
  AREA:wrongdata#FF0000:unknown data
```

If there is unknown data in one of users1..users4, the "wrongdata" AREA will be drawn and because it starts at the X–axis and has infinite height it will effectively overwrite the STACKed parts.

You could combine the two CDEF lines into one (we don't use "allusers") if you like. But there are good reasons for writing two CDEFS:

- It improves the readability of the script.
- It can be used inside GPRINT to display the total number of users.

If you choose to combine them, you can substitute the "allusers" in the second CDEF with the part after the equal sign from the first line:

```
CDEF:wrongdata=users1,users2,users3,users4,+,+,+,UN,INF,UNKN,IF
```

If you do so, you won't be able to use these next GPRINTs:

```
COMMENT:"Total number of users seen"
GPRINT:allusers:MAX:"Maximum: %6.0lf"
GPRINT:allusers:MIN:"Minimum: %6.0lf"
GPRINT:allusers:AVERAGE:"Average: %6.0lf"
GPRINT:allusers:LAST:"Current: %6.0lf\n"
```

The examples from the RRD graph manual page

Degrees Celsius vs. Degrees Fahrenheit

To convert Celsius into Fahrenheit use the formula F=9/5*C+32

```
rrdtool graph demo.png --title="Demo Graph" \
   DEF:cel=demo.rrd:exhaust:AVERAGE \
   CDEF:far=9,5,/,cel,*,32,+ \
   LINE2:cel#00a000:"D. Celsius" \
   LINE2:far#ff0000:"D. Fahrenheit\c"
```

This example gets the DS called "exhaust" from database "demo.rrd" and puts the values in variable "cel". The CDEF used is evaluated as follows:

```
CDEF:far=9,5,/,cel,*,32,+
1. push 9, push 5
2. push function "divide" and process it
   the stack now contains 9/5
3. push variable "cel"
4. push function "multiply" and process it
   the stack now contains 9/5*cel
5. push 32
6. push function "plus" and process it
   the stack contains now the temperature in Fahrenheit
```

Changing unknown into zero

```
rrdtool graph demo.png --title="Demo Graph" \
   DEF:idat1=interface1.rrd:ds0:AVERAGE \
   DEF:idat2=interface2.rrd:ds0:AVERAGE \
   DEF:odat1=interface1.rrd:ds1:AVERAGE \
   DEF:odat2=interface2.rrd:ds1:AVERAGE \
   CDEF:agginput=idat1,UN,0,idat1,IF,idat2,UN,0,idat2,IF,+,8,* \
   CDEF:aggoutput=odat1,UN,0,odat1,IF,odat2,UN,0,odat2,IF,+,8,* \
   AREA:agginput#00cc00:Input Aggregate \
   LINE1:aggoutput#0000FF:Output Aggregate
```

These two CDEFs are built from several functions. It helps to split them when viewing what they do. Starting with the first CDEF we would get:

```
idat1,UN --> a
0        --> b
idat1    --> c
if (a) then (b) else (c)
```

The result is therefore "0" if it is true that "idat1" equals "UN". If not, the original value of "idat1" is put back on the stack. Lets call this answer "d". The process is repeated for the next five items on the stack, it is done the same and will return answer "h". The resulting stack is therefore "d,h". The expression has been simplified to "d,h,+,8,*" and it will now be easy to see that we add "d" and "h", and multiply the result with eight.

The end result is that we have added "idat1" and "idat2" and in the process we effectively ignored unknown values. The result is multiplied by eight, most likely to convert bytes/s to bits/s.

Infinity demo

```
rrdtool graph example.png --title="INF demo" \
   DEF:val1=some.rrd:ds0:AVERAGE \
   DEF:val2=some.rrd:ds1:AVERAGE \
   DEF:val3=some.rrd:ds2:AVERAGE \
   DEF:val4=other.rrd:ds0:AVERAGE \
   CDEF:background=val4,POP,TIME,7200,%,3600,LE,INF,UNKN,IF \
   CDEF:wipeout=val1,val2,val3,val4,+,+,+,UN,INF,UNKN,IF \
   AREA:background#F0F0F0 \
   AREA:val1#0000FF:Value1 \
   STACK:val2#00C000:Value2 \
   STACK:val3#FFFF00:Value3 \
   STACK:val4#FFC000:Value4 \
   AREA:whipeout#FF0000:Unknown
```

This demo demonstrates two ways to use infinity. It is a bit tricky to see what happens in the ‘‘background’’ CDEF.

```
"val4,POP,TIME,7200,%,3600,LE,INF,UNKN,IF"
```

This RPN takes the value of ‘‘val4’’ as input and then immediately removes it from the stack using ‘‘POP’’. The stack is now empty but as a side effect we now know the time that this sample was taken. This time is put on the stack by the ‘‘TIME’’ function.

‘‘TIME,7200,%’’ takes the modulo of time and 7’200 (which is two hours). The resulting value on the stack will be a number in the range from 0 to 7199.

For people who don’t know the modulo function: it is the remainder after an integer division. If you divide 16 by 3, the answer would be 5 and the remainder would be 1. So, ‘‘16,3,%’’ returns 1.

We have the result of ‘‘TIME,7200,%’’ on the stack, lets call this ‘‘a’’. The start of the RPN has become ‘‘a,3600,LE’’ and this checks if ‘‘a’’ is less or equal than ‘‘3600’’. It is true half of the time. We now have to process the rest of the RPN and this is only a simple ‘‘IF’’ function that returns either ‘‘INF’’ or ‘‘UNKN’’ depending on the time. This is returned to variable ‘‘background’’.

The second CDEF has been discussed earlier in this document so we won’t do that here.

Now you can draw the different layers. Start with the background that is either unknown (nothing to see) or infinite (the whole positive part of the graph gets filled).

Next you draw the data on top of this background, it will overlay the background. Suppose one of val1..val4 would be unknown, in that case you end up with only three bars stacked on top of each other. You don’t want to see this because the data is only valid when all four variables are valid. This is why you use the second CDEF, it will overlay the data with an AREA so the data cannot be seen anymore.

If your data can also have negative values you also need to overwrite the other half of your graph. This can be done in a relatively simple way: what you need is the ‘‘wipeout’’ variable and place a negative sign before it: ‘‘CDEF:wipeout2=wipeout,−1,*’’

Filtering data

You may do some complex data filtering:

```
MEDIAN FILTER: filters shot noise

  DEF:var=database.rrd:traffic:AVERAGE
  CDEF:prev1=PREV(var)
  CDEF:prev2=PREV(prev1)
  CDEF:median=var,prev1,prev2,3,SORT,POP,EXC,POP
  LINE3:median#000077:filtered
  LINE1:prev2#007700:'raw data'
```

```
    DERIVATE:

      DEF:var=database.rrd:traffic:AVERAGE
      CDEF:prev1=PREV(var)
      CDEF:time=var,POP,TIME
      CDEF:prevtime=PREV(time)
      CDEF:derivate=var,prev1,-,time,prevtime,-,/
      LINE3:derivate#000077:derivate
      LINE1:var#007700:'raw data'
```

Out of ideas for now

This document was created from questions asked by either myself or by other people on the RRDtool mailing list. Please let me know if you find errors in it or if you have trouble understanding it. If you think there should be an addition, mail me: <alex@vandenbogaerdt.nl>

Remember: **No feedback equals no changes!**

SEE ALSO

The RRDtool manpages

AUTHOR

Alex van den Bogaerdt <alex@vandenbogaerdt.nl>