

# Circ Program

## Overview

Circ was designed to allow disparate stimulus presentation and eeg digitizing hardware to coexist and to allow the user to view subject response data in real time. Circ is primarily concerned with the concept of an event code, a 16 bit unsigned number, and a device associated with the event code. Circ distinguishes between input and output devices: input devices are sources of event codes (e.g. serial port, game port) and are most often associated with stimulus presentation marks and subject input, output devices are sinks of event codes and are most often associated with digitizing marks and stimulus presentation cues. Circ also has the provision to do calculations based upon functions that are built in to the program using variables that are implemented as 8 byte double precision floating point values. Unlike event code input and output, these calculations do not happen real-time, but rather before any given display refresh. Circ also allows a direct connection to the internals of the program using system messages. Circ processes these objects with the help of a user-defined command file which specifies how input events are to be formed in order that appropriate actions may be taken. These conditions are specified using an erpss bdf style convention (see eman bdf(E5)) with the extension that all cells first contain the device name followed by a colon and without the flag field. Once running, circ uses a state machine derived from the command file to decide if any condition has been met with each new event code arriving on an input device. If a condition has been met actions specified by the user will be taken in the following order: all output to devices occur instantly, all variable calculations are queued until the next display update and all system events are queued until all output devices have been serviced. These priorities ensure that circ acts as a transparent router for all codes that may be destined for other devices (especially stimulus presentation to digitization hardware).

## Usage

circ [options] [commandfile](#)

options:

- ?     Display a simple description of program invocation and option syntax.
- h     Use old horizontal display. There was a time that the primary display of circ showed codes scrolling from left to right across the display. This was found to be confusing in conjunction with other programs which had codes scrolling top to bottom and successive codes from the same device, if spaced closely enough in time, would step on each other, losing information as to the number of codes arriving. To solve these problems the current default top to bottom display was implemented. Note that messages are not implemented in the old display and that much more effort has gone into the new vertical display making it much more robust. Use the horizontal display only in emergencies or for testing.
- p     Keep preprocessed file `__circ__.tmp`. When processing the circ command file, circ first preprocesses the file to expand macros and include other command files, normally this preprocessed file is then deleted. This option keeps the file so that it may be viewed by the user. This is most often used to view macro expansions.
- t     Test mode. This option prints out a summary of the

circ command file. It is for testing only and most likely will be removed soon.

**-r delay**

Refresh delay in milliseconds. Circ does not update the screen every time a code comes in but rather only after an appropriate delay (default 250ms). This option sets the number of milliseconds between display refreshes. If this value is set too low circ may not have an even refresh as input, output and variable calculations have higher priority. Also setting this value below the refresh rate of the monitor (typically about 17ms) has no effect on the output. This option is most often used in conjunction with the -h option (horizontal display).

**-D name value**

Define name to value. This option sets a defined macro to value, see circ.4 for a description of the circ command file format.

**-s**

Use system for timing. Normally circ uses a timer card with 1ms accuracy for timing. This option tells circ to use the system clock for timing thereby obviating the need for special hardware to run circ. This is not without it's drawbacks as in order for the system clock to be used without destroying the machines current time of day, a 1ms pulse is not efficient and the pulse is set to a slightly higher frequency. This fact throws off timing calculations throughout circ (which assumes a 1ms clock tick) and therefore this option should only be used for testing.

## Diagnostics

Circ tries to report parsing errors by printing out the offending file and line number. this should only be used as a guide, though, as not much effort has gone into error reporting and as often as not the line numbers are off by a bit. If an error occurs during run time circ most often exists with output something like: Running: State list overflow. Currently, there is no precise definition of these run time errors and hopefully the messages will be self explanatory.

## Bugs

Parsing errors are not caught very well. If a device is not present or if pending input to a device exists circ will often hang. Dos critical errors are not caught (disk reads or writes with no floppy present, divide by zero, ...).

# Circ Command Files

## Overview

Circ command files are 7-bit ASCII text files that contain commands that are interpreted by the circ program. By convention they are suffixed with a .cir extension. Circ command files consist of preprocessor commands, definitions, conditions and actions. Preprocessor commands are expanded by the preprocessor before circ actually interprets the command file. All preprocessor commands are prefixed with the character

followed by a name (with no intervening spaces) and include utilities to expand macros show messages on the circ display and include other circ command files into the current command file. Definitions allow the user to assign names, add additional parameters and instantiate variables and devices for use later in the conditions and actions. Conditions allow the user to parse the input streams of codes into functional blocks are then used to initiate actions that include updating variables, sending codes to output devices and performing systems requests.

## Preprocessor Syntax

Before the circ program interprets the command file it preprocesses it, creating a file called `__circ__.tmp` in the current working directory (don't name any user files `__circ__.tmp` as they will be overwritten). The `!include` directive is used to include another circ file at the current location in the current file, the `!message` directive prints a message on the circ display at startup time and the `!define` directive allows the user to assign text to a name. Here is an example:

```
!include "inputs.cir" # read in file inputs.cir
!include "outputs.cir" # read in file outputs.cir
!include "vars.cir" # read in file vars.cir

!define ZERO "0" # only the 0 is substituted (not the ''s)

var n=ZERO # translates to var n=0

!message "Hi Mom!" # sent to circ display at startup
```

## Definition Syntax

In order for a device or variable to be used it must be defined in the circ command file. A variable can be defined and given a name, an initial value and whether or not it should be displayed. A device can be defined by specifying the device name, specifying an optional name and additional device specific parameters in a string. Here is an example:

```
# create a variable named "n" set it to 1
# initially and don't show it on the circ
# display
var n=1 "display=0"

# create a variable named "another" and don't
# initialize it but do display it.
var another

# initialize a device to talk to the serial
# port named "sp" and set appropriate parameters
device serial sp "baud=9600"

# initialize a device to talk to the digitize
# program, give it no name and no additional parameters.
# The name used will be the device name.
device dig
```

A variable may only be defined once but a device can often be defined multiple times with different parameters, e.g. the serial device can be called with different parameters that specify which serial port to look for (com1 or com2). Condition and Action List Syntax In order for the circ program to do anything with the devices and variables defined it must be able to separate the various input codes from the different input devices into functional units. The syntax chosen for this separation is similar to that used by the erpss package bdf (bin descriptor file) format (see eman bdf on systems with erpss). The circ condition syntax

varies from the bdf syntax in that there are no flags and, at the beginning of a cell, there is an input device name followed by a colon. When a condition matches it then invokes the action list following the condition. The action list can specify that circ output the codes received to output devices, update variables with the code values, code times and functions of these values and also perform system functions such as sending messages to the circ display, exiting the program and forcing states to reset themselves. Here is an example:

```
# on any keyboard input but q, copy to outputs, kbcnt++
{kb:~'q'}          # any keyboard event but 'q'
  file=$1          # send the code to the output device file
                  # $1 refers to the value of the code in
                  # the first (only) cell
  kbcnt=plus(kbcnt,1) # kbcnt = kbcnt + 1
  message "Got a key" # send a message to the circ display
  end              # done with this action list

# compute a response time
{gp:1}{gp:t<10..1000>2}# gameport 1 followed by a 2
                        # in the time window 10-1000ms
  rt = minus(%2, %1)  # get the response time
                      # %2 refers to the time that
                      # code # 2 (gp:2) event occurred
                      # %1 refers to the time that
                      # code # 1 (gp:1) event occurred
                      # minus is a function that subtracts
                      # the second argument from the 1st
                      # and assigns the result to the
                      # variable rt
  end                 # done with this action list

# user control: 'q' to quit
{kb:'q'}            # keyboard 'q' event
  abort             # quit circ program
  end               # done with this action list
```

## Implemented Devices

Name	Input	Output	Description
keyboard	y	n	Keyboard input device
gameport	y	n	Joystick button interface
bbdt2821	y	n	Input from vvsp (interrupt)
bbpolled	y	n	Input from vvsp (polled)
serial	y	y	Com1 and Com2 interface
infile	y	n	Input from a file of codes
outfile	n	y	Output to a file
erpss	n	y	Output to an erpss log file
dig	n	y	Output to digitize program
midiout	n	y	Output to a midi device

## Implemented Functions

Name	Description
divide	Divide two numbers
minus	Subtract two numbers
plus	Add two numbers
times	Multiply two numbers
beta	Compute Beta statistic
dprime	Compute dprime statistic

## System access

Name	Description
abort	Exit circ
message	Show a message on circ display

## Bugs

There is no way to specify strings in macro definitions in the circ file. Old horizontal display does not show messages.

## Bin Descriptor Files:

(Thanks to Todd Handy for this)

NOTE: The circ condition syntax varies from the bdf syntax in that there are no flags and, at the beginning of a cell, there is an input device name followed by a colon. Use the following for information purposes only – see the circ example section for the exact syntax for your input files.

Bin descriptor files (bdf's) are files used by the ERPSS program ECDBL to sort data based upon the event codes used in a particular experiment. Bdfs enable the user to average eeg epochs as well as tally event and response numbers. Events in a bdf are defined by the user based upon stimulus and response code sequences (usually understood as the event/response codes that make up a trial) from a particular experiment. Like sequences, or trials, are placed into common "averaging" bins. That is, trials are separated between both different trial types and like trial types with different responses. ECDBL then produces a tally of how many events were placed into each bin and allows for subsequent eeg averaging (using other ERPSS programs) within each bin. Bin definitions, or *descriptions*, can be simple, such as putting each occurrence of a particular stimuli into one bin, or very complex, such as specifying an exact order and timing for a **series** of stimulus/response events to qualify for a particular bin. The task at hand then is to learn the syntax that is used to make bin definitions that actually enable sorting of trials and responses to occur.

Bdf's themselves are simply a list of bins. Each bin is composed of three parts, each on a separate line: a bin number, a bin label, and a bin description. The format looks like this:

```
Bin 1
Chunks of Cheddar
.{99}
```

where "Bin 1" is the bin number, "Chunks of Cheddar" is the bin label, and ".{99}" is the bin description. Each bin is required to have a different bin number. Understanding bin numbers is simple: The first bin in your bdf must be "Bin 1"; the nth bin in your bdf file must be "Bin n". How a bin is labeled is up to the creator, but it is important to ensure that labels are clear and sagacious. Bin descriptions require the use of specific syntax, the explanation of which makes up the remainder of the boring documentation in front of you.

As hinted at above, bin descriptions are used to sort through your EEG data, placing like events into common bins. These events, if you recall, are defined based upon the stimulus and response code values used in one's experiment. This is best understood by example. Suppose you have one bin to describe, wherein you want to place every "9" code recorded on the EEG. The 9 may correspond to either a stimulus

or a response; what matters here, however, is that you know what the 9 represents. To have any and all 9's in the EEG placed into a bin, the description (line 3 in the above bin example) would simply be:

```
.{9}
```

It's that simple, but let's explain it a little. Each bin must include, or reference, at least one event code, be it a stimulus code or a response code. That event is called **the home item**. **Thus, every bin, at the minimum, has a home item**. In the above example, the home item is a 9. **This syntax of a code value enclosed in curly brackets, following a period, is the root of all bin descriptions**. A home item then is **always** in every bin definition and **always** is enclosed in curly brackets and **always** follows a period (.).

Now that home items are understood (if not, keep re-reading the above paragraph) we have a foundation upon which to build a deeper understanding. All bin descriptions are made up of what are called item specifiers. Therefore, all home items are also item specifiers. However, not all item specifiers are home items. Additional item specifiers can be placed into a bin description, along with the omnipresent home item, that serve to further define the conditions that must be met for a code to be placed into a bin. These item specifiers can be understood as conditional modifiers of the home item. That is, the home item is always the item specifier that determines the code to be placed into a bin (if it matches); additional item specifiers can be used to make the matching process more selective. It is to what these additional item specifiers are that we now turn our glacial head.

Let's start with an easy example. Suppose the home item corresponds to a stimulus code with a value of 9. Now, let's pretend that each time the subject saw this stimulus they were supposed to make a response via a button press which was recorded on the EEG as a "2". So now, instead of picking up all of the 9's as in our original example, we want to be more selective, picking up only 9's on the EEG that were followed **in time** by the correct response, a 2. The bin description would look like this:

```
.{9}{2}
```

ECDBL will now look for any 9's on the EEG followed in time by a 2 and put any such occurrences into the bin. If a 9 is followed in time by any other event code it will not be placed into that bin. Using a similar syntax, one can also look for specific event codes preceding the designated home item in time. For instance, suppose now that we have to separate stimuli, coded with separate values (say 9 and 8) that the subject sees before making a response (coded 2). Assuming that the first stimulus was coded 8 and the second stimulus (below shown again as the home item) was coded 9, Bin 1 would become:

```
{8}.{9}{2}
```

One other simple thing can be introduced here. Suppose now instead of one subject response (2) there are two (2 or 3) the subject may have made. If we want to modify our most recent Bin 1 to pick up 9's preceded by 8's and followed by either 2's **or** 3's, we can use a semi-colon in the bin specifier to separate the 2 and 3, which ecdbl sees as an "or" operator:

```
{8}.{9}{2;3}
```

Similarly, if we want to pick up 9's preceded by 8's **or** 7's and followed by 2's **or** 3's, we'd write:

```
{7;8}.{9}{2;3}
```

Any number of codes can be linked in this fashion. This is nice, but let's say you want to pick up all occurrences of 9's followed by **anything but** a 6. If there were only two other possible codes (other than a 6) that might follow a 9, then the above "or" syntax is okay. However, if **20** other possible codes could follow the 9 things become a little more time consuming to type out. Fortunately there is a way to say "anything but": **the tilde sign ~**. So, if we wanted to place all 9's followed by **anything but** a 6 into some random bin, we would type:

.{9}{~6}

This works just the same for events placed **before** the home item. However, because we are all functionally intelligent, I will parsimoniously dispense with an example.

At this point, all the basics to describing bins have been shown.

First, there is always a home item, **distinguished by immediately following the period** in the bin description (9 in all the above examples). Second, codes following the home item can be included in the bin definition (2 and 3 in the above examples). And third, codes preceding the home item can be included in the bin definition (7 and 8 in the above examples).

Complexities now start to arise by introducing syntax that allows for temporal specifications. That is, one can select for codes, either before or after the home item, based upon **when in time** the event code occurred relative to the home item. This is often useful, for example, when one wants to pick up a response that occurred within a certain time window following a target event. It's actually not all that bad, so let's peruse an example:

First, suppose I'm interested in 9's followed by 2's (as seen in an earlier example). However, for whatever reason (usually setting time limits for picking up responses to stimuli), it is imperative that the 2 occur in a time window from 200 - 800 msec **after** the 9. If there is no 2 in that time window following the 9, I don't want that 9 in the bin at hand. To do this, the bin description would look like this:

.{9}{t<200-800>2}

In this description, there always must be a "t" followed by a "<", then the time window (in msec) followed by a ">" and then the desired code value.

Now, using syntax described above, it is simple to alternatively have a bin that looks for the **absence** of a particular code within a given time window after (or before, as we shall see) the home item:

.{9}{~t<200-800>2}

This means: if there is a 2 between 200 and 800 msec following the 9, then **don't** include it in the bin.

One can also use the semi-colon "or" operator with time specifiers. The following bin description is similar to the first time example, except now we will accept 2's **or** 3's within the time window:

.{9}{t<200-800>2;3}

Alternatively, we could say we **don't** want any 2's or 3's within a time window following the home item:

.{9}{~t<200-800>2;3}

Using similar syntax, you can look for events in time windows that occur **before** the home item. Suppose I want to pick up any 9's **preceded** by either a 7 or an 8. However, it's important that the 7 or 8 occurred 100 to 300 milliseconds **before** the 9. This would be written as:

{t<100-300>7;8}.{9}

Syntax remains the same whether the time window is placed before or after the home item. However, **what is important to note here is that all defined time windows are time locked to the home item.**

Consequently, any time windows placed **before** the home item (that is, **preceding** the period in the description) are interpreted by ecdbl as time **before** the home item. Likewise, any time windows placed **after** the home item imply time **following** the home item. This should make intuitive sense; if not, take a rest and get some coffee.

Okay, there are only a few complicated things left. First, the order of item specifiers in the bin description is important. To help understand why we care about this, it is good to know how ecdbl works. Ecdbl starts at the beginning of a log file and reads the first code in the file. Having identified the code value it then looks into the bdf and compares the code value with each and every bin description; any time it finds a match it will place that code into that bin (which implies an important point: **each event code can, and often does, go into more than 1 bin**). Now is the really important stuff, so pay attention. To determine if a given code matches a bin description, ecdbl starts at the home item **and then reads right**, comparing the specified sequence (i.e. the actual bin description written by you) with what is on the log file. As soon as it finds something that **doesn't** match, it terminates the search and moves on to the next bin description. However, if everything **proceeding** the home item matches with what's on the log file, ecdbl will then and only then compare the event code with the item specifiers **preceeding** the home item in the bin description. Why is this important? Let's use another example:

Suppose I have an experiment with a two-choice, forced-choice task. That is, to each target/no target stimulus the subject must respond with a yes or a no. Consequently, to pick up hits, false alarms, misses, and correct rejects, my bin descriptions would have the target/no target codes as the home items and then search for corresponding response codes. To really make this clear, let's draw it out. Say the stimulus requiring a yes response is coded with an 8 and the stimulus requiring a no response is coded with a 9. A yes response will be a 2 and a no response will be a 3. In addition, let's put in a time window from 200 to 600 msec in which a response must occur to be included in the bin. The bin descriptions would be as follows:

```
.{8}{t<200-600>2} "hit" bin
.{8}{t<200-600>3} "miss" bin
.{9}{t<200-600>3} "correct reject" bin
.{9}{t<200-600>2} "false alarm" bin
```

So far, great. But suppose for a moment that the task is such that sometimes subjects will accidentally press the wrong response button first and then press the correct one (it happens). This can cause trouble because that event would get placed into two bins, screwing up your d prime calculations. Consequently, one must actually modify the above bins to not only search for the presence of one response code (as they already do), but also search for (and confirm) the **absence** of the other response code. Well, we can do that using the "~":

```
.{9}{t<200-600>3}{~t<200-600>2} (this is wrong; see below)
```

However, **the way in which ecdbl reads bin descriptions now becomes very important**. Why? Well ecdbl has a bit of a glitch. In the above example, ecdbl, upon first matching the home item (a 9), will then look for a 3 within the designated window following the 9. If it doesn't find one, then there is a failure to match and it will move on to the next bin in the bdf. However, if it does find a 3 in the designated time window things get weird. Ecdbl will then check for the absence of a 2, **but only from the time the 3 occurred to the end of the window; if a 3 occurred BEFORE the 2 then ecdbl won't detect it!** Fortunately, one can get around this with relative, but important, ease: Place the "~" time specifier first:

```
.{9}{~t<200-600>2}{t<200-600>3}
```

There is now only one last thing of importance to dispense with. That is the use of what are called "flags". Flags are nifty little things that allow one to mark desired event or response codes within a log file. What does this mean? Well, in use with response codes one often wants to ensure that a given response code is paired to one and only one preceeding event code; flags can be used to do this. With event codes, flags can be used to mark trials that have been rejected due to things like eye movement artifacts. How is it done? Commands can be written into your bin descriptions that can both place flags onto specific codes and check

codes for the presence of flags. The details of flag use can be a bit complicated, so it is best perhaps to defer to the official bdf manual (starting on page 8).

## CIRC EXAMPLES

Begin actual circ file here:

```
# Basic circ file for a go/no go task
```

```
#Define some variables of interest
```

```
var  stims=0          # count the number of stimuli from VAPP
var  response=0       # count number of responses codes from fiber-optic LumiTouch system
var  targets=0        # count the number of targets (go stimuli)
var  nogo=0           # count the number of nogo stimuli
var  hit=0            # hit counts
var  rt=0  "display=0" # reaction time variable – defined but not on the circ display
var  rt_sum=0 "display=0" # rt total value – to be divided by number of hits to give rt's for hits
var  hit_rt=0         # define hit reaction time variable
var  misses=0         # number of missed targets
var  fa_nogo=0        # false alarms to nogo stimuli
var  scan=0           # this variable will count the number of scanner codes from GE system
```

```
#Define the devices we are going to look for:
```

```
# the first one below (gameport) is not used anymore but is left for historical and illustrative purposes
```

```
#device gameport  gp  "sab1=201 sab2=200 sbb1=201 sbb2=200"
```

```
device keyboard  kb          #define the keyboard
```

```
device serial    scanner "port=2 baud=9600 "      # scan codes arrive via serial port 2
```

```
device serial    stim  "port=1 baud=9600"        # VAPP codes arrive via serial port 1
```

```
#now we can define our output file – all codes stored in ERPSS format or can be stored on circ in
```

```
# ascii format – at UBC we use erpss format and then use custom matlab code to chop
```

```
# out events and put them into timing files for analyses in SPM99 (code available upon request)
```

```
# for groups without access to ERPSS, you may want to use the ascii option for output
```

```
device erpss      of  "name=go1sxx.log display=0"  #create a erpss binary log file of all codes
```

```
#now let's define all our bin descriptors and how we are going to chop up data stream
```

```
# on keyboard input 'q' (escape), abort circ program
```

```
.{kb:'q'}
  abort
end
```

```
# clear stims variable upon hitting the 'c' on the keyboard (we could add all other variables from
```

```
#above if we wanted to
```

```
.{kb:'c'}
  stims=0
end
```

```
# on any stim input, send code to log files, increment count of stims variable
```

```

.{stim:*}
  stims=plus (stims,1)
  of=$1
end

#on any scanner input via parallel port 2 – send code to log file and increment scan variable
.{scanner:*}
  of=$1
  scan=plus (scan,1)
end

# on any response code (predefined at event number 53) input increment count
.{stim:53}
  response=plus (response,1)
end

# on any go stimulus (ie target) increment count
.{stim:8}
  targets=plus (targets,1)
end

# on any no go stimulus increment count
.{stim:7}
  nogo=plus (nogo,1)
end

# hits – define a go code followed in time window 100 ms to 1500 ms by a response code 53 as a hit
.{stim:8}{stim:t<100..1500>53}
  hit=plus (hit,1)
end

# misses – define a miss as any go code not followed in time window by a response code 53
.{stim:8}{stim:t<100..1500>~53}
  misses=plus (misses,1)
end

# false alarms to no go stimuli – any no go code of 7 followed by a 53 response is an error trial
.{stim:7}{stim:t<100..1200>53}
  fa_nogo=plus (fa_nogo,1)
end

#compute reaction time measure for correct hits
.{stim:8}{stim:t<100..1200>53}
  rt= minus (%2,%1)
  rt_sum = plus (rt_sum, rt)
end

# at the end of every vapp run we send the event code 1 out with the last stimulus to compute rts
.{stim:1}
  hit_rt = divide (rt_sum, hit)
end

```

