document version 1.8

30 August, 1997

The CORTEX (version 5) Timing File Reference Manual

by:

Steve Macknik, Andy Mitz, Tom White, and Robert Desimone

Lab Neuropsychology, NIMH

1. Introduction

At CORTEX's heart is a C compiler. The CORTEX C compiler has many ANSI C features but is not strictly ANSI C compliant (this is described more in the CORTEX User's Manual). This C compiler has a name: the CORTEX State System, or CSS for short. Hence, in many places in the various help files and source code files that make up CORTEX, you'll find this mnemonic. You might also find references to "State" files or "CSS" files, both refer to -- what is called in this manual -- timing files.

The menu system, item files, condition files, etc. are niceties for allowing one to easily define the variables and items that the CSS is to manipulate. Once these variables and items have been defined, by loading item and condition files (or by setting these values manually in the menu system), the timing file is loaded (that is to say COMPILED) and a stack of functions is created in object code. This object code is run just as one would run any other executable program.....with one major exception. CORTEX object code is run only from the RUN:START menu, not from DOS. CORTEX also does a good deed for the user by including many hidden aspects in the program that allow CORTEX to not only run the graphical operations requested by the user, but to also collect spikes, monitor and analyze eye-movements, operate and record from a plethora of behavioral apparatus, and control general analog-to-digital operations, all on-line at up to 10 kHz precision. Most of these operations are done in the background to the graphical operations, but can be accessed and controlled from the timing file.

Just as the CSS is CORTEX's heart, the timing file is its brain. From the timing file, the user defines the timing of the graphical operations and can perform virtually unlimited analytic operations on the incoming data on-line, by accessing the various autonomic systems buried within CORTEX that are not directly controlled by the user.

This manual is dedicated to organizing and describing the sundry routines available from timing files in a way that the beginning programmer will be able to understand the many operations and capabilities of CORTEX.

2. Timing File Routines by Category

The routines that are called in a timing file are listed here by category. Later in this manual is an alphabetical listing of the functions and more detail individual descriptions of each routine.

1. Behavioral Data Collection and Reward Routines

The routines represent the routines for on-line sampling of behavior from within a timing file. Most of these functions are a superset of the functions found in the section on Input/Output Routines, so the user can use those if these functions do not serve a specific need.

1. Eye-movement monitoring and analysis

These routines are dedicated to supplying the user with information concerning the position of the eye and its relationship to various items in the visual field, such as the fixation window or a visual stimulus.

Routine	Use
EYEget_dva()	Gets the eye position in dva.
get_fixation_state()	Returns whether the monkey's eye is within the fixation window.
get_saccade_state()	Returns whether the monkey's eye is within the fixation window and if it is stable or moving within the fixation window.
ITEM_POSeye_delta()	Gets distance of the eye_spot from an item's center as well as other handy statistics.
ITEM_POSeye_ishere()	Checks if the eye position is within the bounds of an item.
ITEM_POSeye_iswithin()	This is a more conservative version of ITEM_POSeye_ishere().
set_fixwin_params()	Specifies dynamic vs. static nature of the fixation window and sets the horizontal and vertical extent of the fixation window if static.

2. Hand-bar sampling

This routine supplies the user with information concerning the status of the hand bar.

Routine	Use
get_bar_state()	Returns the current state of the monkey's response bar.

3. Reward

This routine supplies a pulse to the monkey's reward system.

Routine	Use
reward()	Gives the monkey a reward.

4. Behavioral Results Analysis/Manipulation Routines

This section categorizes the functions that the user can access to analyze and manipulate the behavioral data (not the spike or eye-movement data) from within the timing file. CORTEX defines an experimental condition as a single type of trial, typically with either a specific pattern of items in each test_screen (see the condition file section of the CORTEX USER's Manual) or a specific pattern of behavioral contingencies (or whatever). Since all trials with the same condition number are labelled as such in the data file, the condition # is one of the ways that the data for each trial can be sorted for off-line statistical analysis. For example, if you wanted to compare the reaction time (or firing rate) for a red stimulus presented on one trial compared to a blue stimulus presented on a different trial, you could put the red and blue stimuli in separate conditions. CORTEX defines a BLOCK to be a group of conditions. The BLOCK number is also used to label the trials in the data file and is, therefore, also a way to sort the data afterwards. In addition, CORTEX assumes that all of the conditions in a block are run together, usually in an interleaved fashion, separately from the conditions in a different block. For example, to run conditions 1-10 randomly interleaved followed by conditions 11-20 randomly interleaved, you would use two blocks.

Behavioral data are summarized on-line for all of the conditions within each block. A circular buffer is established in CORTEX, which holds the behavioral results from the previous trials sampled within each block of conditions.

Routine	Use
BLOCKclear_stats()	Clears the percent correct and circular buffer tables for a given block or condition.
BLOCKget_pct_correct()	Gets percent correct information for a given block or condition.
BLOCKget_stats()	Gets the statistics for various variables concerning a specific block or condition.
get_block_pct_correct()	Returns the average percentage of correct trials in a block.
get_cond_pct_correct()	Returns the average percentage of correct trials in a condition.
recent_block_status()	Returns whether at least a minimum number of the last several trials were correct.
block_clear()	clear the percent correct and/or the circular buffer tables for the current block.

5. Behavioral Results Storage Routines

This section lists the functions that the user calls in order to record the behavioral performance of the subject in the permanent data file on disk and in the circular buffers (see Behavioral Data Analysis/Manipulation Routines for a discussion of the circular buffers). When the user has decided, from within the timing file, that the trial is either correct or incorrect, these functions can be called to record the result.

Routine	Use
break_fixation_error()	Records that the monkey has broken fixation.
no_fixation()	Records that the monkey never achieved fixation.
response_before_test()	Records that a behavioral response was received before the presentation of the test stimulus occurred.
response_correct()	Records that there was a correct behavioral response.
response_early()	Records that there was a behavioral response earlier than expected.
response_late()	Records that there was a behavioral response later than expected.
response_missing()	Records that there was an absence of any behavioral response in the trial.
response_no_bar_down()	Records that there was no bar down at the start of the trial.
response_wrong()	Records that there was an unexpected or incorrect response.

6. Data Collection Control Routines

In general, CORTEX does not entrust the user with the direct handling of spike collection/storage or eye-movement monitoring/storage. This is just as well since the collection of these two types of data are done via CPU interrupts. For those of you uninitiated in computer lingo, interrupts are simply functions that are called directly by computer hardware initiated events, such as a signal from the real-time clock on the motherboard. CORTEX programs the real time clock to provide an interrupt at a user-defined rate, typically every millisecond. The interrupt routine samples the spike inputs and the eog inputs and clears the external flip flop circuits that take in the spikes. The user does have control of when data collection (meaning spike and eye-movement collection) begins and ends by virtue of functions that can be called from the timing files (CORTEX state system), described below. Other types of digital and (in a newer versions of CORTEX) analog data that are not sampled in the interrupt routines are accessible from functions in the timing files as well.

Routine	Use
collect_data()	Instructs CORTEX to either begin or stop collecting spike data for storage in the data file.
end_trial()	Cleans up after the trial and turns off the mapping stimulus in play mode.
put_eye_data_in_buf()	Start/stop storing the eye movement data in a buffer (channels 3&4 of DASH16)
put_epp_data_in_buf()	Start/stop storing analog data from channels 1&2 of DASH16.
start_trial()	Initializes the spike input flip flop hardware and sets the bin_width for the on-line cumulative histogram display.

7. Directory Control

The data conversion functions in CORTEX allow the user to convert numbers to strings.

Routine	Use
chdir()	Changes the current working directory.
getcwd()	Gets the current working directory for the specified drive.
mkdir()	Makes a new directory.
rmdir()	Removes an existing directory.

8. Encode Events in the Data File

Each trial's data in the permanent disk data file begins with a header followed by an event array containing the codes for all types of events that occured during the trial (spikes, behavioral inputs, etc) as well as the time of occurrence for each event. Following the event array is an array holding the (optional) eog values. The condition number and block number are stored automatically in the header. User-called functions for storing the behavioral outcome of the trial in the header (ie. classifying a trial as behaviorally correct or incorrect) are described in section 2.3 **Behavioral Results Storage Routines**. The only events stored automatically in the event array are the spike events. All other events must be explicity entered into the event array by the encode() function called in the timing file (state system). The encode function stores the code for the event, supplied by the user when he or she calls the funciton in the timing file. The time of the event is automatically places in the file along with the code for the event. The default **EVENT CODES** are listed in section 4 of this manual. The encode function is passed either the name of the event (listed in css_inc.h; remember to #include "css_inc.h" in every timing file that calls the encode() routine) or a user-defined number that stands for a specific event. The user can change css_inc.h as seen fit to customize EVENT CODES to a specific type of experiment. The only macros that should not be changed are the SPIKE1-16 and the BAR_* event codes. These have special meaning to the PROCORTX utility during data analysis, but the user is encouraged to develop new event codes as seen fit.

Routine	Use
encode()	Records an event code in the cortex data file

9. Experimental Design Routines

This section categorizes the functions that the user can access to create specialized experimental designs, such as stair casing within CORTEX. CORTEX internally controls randomization or sequential ordering of the user defined blocks of trials and specific trials within those blocks. For standard designs (such as randomized presentation of trials), the user can set up the blocking from the RUN:PARAMETERS:BLOCK/REPEAT menu. For more complex designs, the user can over-ride this standardized from within the timing file by using these functions. The staircasing system has been completely redesigned such that many of the older functions are obsolete. The new functions all begin with "BLOCK", and the values returned are from 1 to max_blocks (the original functions ran from 0 to (max_blocks-1), which was more confusing).

Routine	**Use**
BLOCKget_block_num()	Returns the current block being tested.
BLOCKget_cond_num()	Returns the current condition being tested.
BLOCKget_control_info()	Returns information about the current control parameters for staircase design.
BLOCKget_max_vals()	Returns the currently set maximum block and maximum condition values.
BLOCKset_control_info()	Sets control parameters for staircase design.
BLOCKset_next()	Sets the block and condition to be run in the next trial.
get_block_num()	Returns the current block being tested.
get_cond_num()	Returns the current condition being tested.
repeat_cond_if_pct_correct()	Repeats the current condition if the percent correct for that condition is less than a critical value.
repeat_block_if_pct_correct()	Repeats the current block if the percent correct for that block is less than a critical value.
block_choose()	Specify the next block relative to the current block.

10. EYE_WINS

The EYE_WIN array was set up in cortex to store information about rectangular spatial windows across trials. Typically, the user put the size and location of windows into the array, and then in the timing file he or she could check to see if the eye position was within one of the stored windows, in an eye movement task, for example. This array is also useful for storing any arbrary data across trials. An EYE_WIN is NOT a fixation window, even though it sounds like it should be. An EYE_WIN is simply a structure that is static (it does not change from trial to trial) that can be characterized from the RUN:PARAMETERS:EYE_WINS menu or from within the timing file. Imagine that you have a stimulus of some sort that the subject manipulates freely with a joystick or by eye-movement during the trial. If you wanted to compare the final position of your stimulus ACROSS trials, there would be only two relatively easy ways to do it: 1) you could access a global variable from within the timing file (this might require some programming of CORTEX itself with the Microsoft C version 7 compiler, depending on what you wanted to do), or 2) you could record the positional data by binding an EYE_WIN to the final position of your stimulus. The total number of EYE_WINs needed is set in the cortex.cfg file. The routines below allow the user to modify the EYE_WINs from within the timing file.

Routine	Use
EYE_WINcopy()	Copies the physical attributes of an item or an EYE_WIN and copies it to a specified EYE_WIN for storage.
EYE_WINreset()	Clears all of the saved EYE_WINs from the EYE_WIN scratch buffer.
EYE_WINset()	Stores a position (center and size) in a scratch buffer for future reference.

11. File Handling

The current file-handling routines allow the user to remove and rename files.

Routine	Use
remove()	Deletes files.
rename()	Renames files.

12. File Stream Control

File stream control functions handle data as a continuous stream of characters. The routines outlined here operate with handles to files, instead of the FILE structure you might find in some file stream control functions in ANSI C. It is generally not necessary for the typical user to use these functions, as CORTEX handles the creation and storage of the data file for the user. Do not use any of the following functions while the clock is running during a trial..

CORTEX now supports the use of buffered I/O (using the FILE structure). These functions begin with the ‘f’ prefix. One should not use buffered I/O routines on files opened with "open()", or low-level I/O routines on files opened with "fopen()." In addition, the three standard FILE streams, stdin, stdout, and stderr, are available as external variables within CSS.

Routine	Use
close()	Closes a file.
dup()	Creates a second handle for a file.
dup2()	Reassigns a handle for a file.
eof()	Tests for end-of-file.
lseek()	Repositions the file pointer to a given location.
open()	Opens a file.
read()	Reads data from a file.

tell()	Gets current file-pointer position.
write()	Writes data to a file.
fprintf()	Like printf(), but writes to an open file.
fscanf()	Read data from a file, parsing it into variables.
fclose()	Closes a file.
feof()	Detects whether the end of the file has been reached.
ferror()	Prints the appropriate error string if an I/O error occurs.
fflush()	Flushes the I/O buffers for a given file (i.e. writes the contents to disk).
fgetc()	Get the next character from a open file.
fgets()	Get the line-feed terminated string from an open file.
fopen()	Open a file for reading, writing, or appending - as binary or text.
fputc()	Write a character to a file.
fread()	Read a specified amount of data from a file.
freopen()	Assign a new FILE handle to an already opened file.
fseek()	Move to a new position in an open file.
ftell()	Return current position in an open file.
fwrite()	Write a specified amount of data to a file

13. Graphics Routines

This section describes the internal timing file calls that allow the user to perform general graphical operations. See example at the end.

1. New Graphics Routines

These routines are the new improved versions of the routines to be found in the Old Graphics Routines section below. The primary difference between the new and the old routines is that only one old routine could run at any given time. The new graphics kernel allows any number of graphical operations to be run simultaneously. The reason that the Old Graphics Routines (and thus earlier versions of CORTEX) were more limited is that they calculated the number of milliseconds it would take to complete the requested graphical operation, then they usurped control of the graphical environment for that period of time. With the New Graphical Routines the user is responsible for updating the queue of graphical commands every so often with a call to Gflush(). Thus, the New Graphics Routines give the user the power to add new things simultaneously. Gflush() will update everything at once. Thus, if you want to make many changes simultaneously, request the changes (thereby adding them to the queue), then call Gflush() when the requested routines are to be processed (thereby processing the commands in the queue) and CSS will keep track of the various operations at any given time, so the user need only be concerned with operations they want to add, not operations they have ordered in the past (if a test_screen is sweeping, you can add a request it to start scrolling it and the CSS will correctly handle the simultaneously sweeping/scrolling test_screen).

Here are some other important differences between the New Graphics Routines and the Old Graphics Routines:

(1) The Old Graphics Routines were written before CSS, and the old state system only handled integers. Therefore, there was the extra annoyance and time burden of having to multiply every degree of visual angle (dva) by 100 (in order to be able to calculate things in hundredths of a degree), instead of just passing floating-point value directly.

(2) The New Graphics Routines are also faster, since they take into account the fact that the monitor is refreshed at a certain rate (such as 60 Hz) and so they check to see whether previously made calls access the same test_screen. If so, they make all of the modifications to that test_screen (newly requested as well as previously requested) simultaneously, whereas with the Old Graphics Routines, every single little change to a test_screen took an entire screen refresh to enact.

(3) The New Graphics Routines allow many more stimulus parameters than previously available. For example, any graphics call can have a duration (-1 means run it forever, or at least until deleted or purged) and when that amount of time has passed, the test_screen will stop preforming the requested graphical operation. Next, changes in the location of a test_screen (the Gscroll or Gmove routines) can be requested to be relative to current position of the test_screen (ie. move the test_screen 5 degrees to the right of where it currently is), or to an absolute position on the screen (ie. move the test_screen 5 degrees to the right of the center of the screen). Additionally, positions can be specified either in degrees of visual angle or pixels (sometimes pixel based operations might give you better smoothness of movement). Finally, coordinates can be specified in x,y values, or in r,t (radius and theta) so that polar notation can be used with circular type movements! If the routines you are interested in do not have these options, then use Gadd() (which is the master routine....all routines are made from Gadd()).

Routine	**Use**
Gadd()	General interface to graphics kernel. The mother of all New Graphical Routines.
Gadd_with_wait()	General interface to graphics kernel that has a delayed onset.
Gcheck()	Returns the time remaining to complete a graphical operation.
GcolorABS()	Resets the color lookup table one color at a time by changing a single value within a color lookup table relative to its current value.
GcolorLUT()	Changes the entire color palette mid-trial using preloaded LUTs (from the LUT menu).
GcolorREL()	Resets the color lookup table one color at a time by changing a single value within a color lookup table relative to its current value.

Gdel()	Removes an graphical operation that has been added to the stack by Gadd() or one of Gadd()'s daughter functions.
Gflush()	After queuing a number of graphics commands, they need to be flushed to ensure that they are drawn to the screen.
GmoveABS()	Moves the center of a test_screen or EYE_WIN to an absolute position.
GmoveREL()	Moves the center of a test_screen or EYE_WIN to a relative position.
Gmovie()	Runs a movie.
Gmovie_step()	Changes the current frame of a movie.
Gon_off()	Turns a test_screen on or off.
Gpan()	Pan a test_screen within its window.
GpanABS()	Pan a test_screen across absolute coordinates while its window stays still.
GpanREL()	Pan a test_screen across a position relative to its current position while its window stays still.
Gpriority()	Changes the priority of the specified test_screen.
Gpurge()	Deletes ALL pending graphics operations at once.
Gscroll()	Scrolls a test_screen.
Gsweep()	Sweeps a test_screen.
GwinSizeABS()	Changes the size of the window surrounding test_screen to an absolute size.
GwinSizeREL()	Changes the size of the window surrounding test_screen relative to its current size.

2. Old Graphics Routines

These routines are the general graphics routines for displaying test_screens left over from older versions of CORTEX available for backward compatibility reasons.

Routine	**Use**
display_fixspot()	Turns on and off the fixation spot.
display_play()	Turns on and off the mapping stimulus in play mode.
display_sample()	Turns on and off the sample stimulus (TEST0).
display_test()	Turns on and off the specified test_screen.
foreback_wins()	Executes the simultaneous motion of two test_screens.
init_foreback()	Initializes the variables for moving two test_screens.
init_movie()	Initializes the variables required to run a movie.
init_pan()	Initialize a test_screen for panning within a stationary window.

init_scroll()	Initializes variables for moving a test_screen around the graphics display.
init_sweep()	Initializes the variables necessary to moves a test_screen across the screen so that it passes through the center of the test_screen's current location.
init_sweep_with_fix()	Initializes the variables necessary to moves a test_screen across the screen so that it passes through the center of the test_screen's current location and the FIXSPOT tracks it.
init_toggle()	Initializes the variables for flipping between two test_screens.
load_CLT()	Changes the entire color palette mid-trial using prel oaded CLTs (from the CLT menu).
load_CLT_subset()	Changes a subset of the color palette mid-trial using preloaded CLTs (from the CLT menu).
move_eye_window()	Move the fixation window without moving the fixspot.
move_fixspot()	Moves and turns on or off the fixation spot.
move_sample()	Moves and turns on or off the sample stimulus (TEST0).
move_test()	Moves and turns on or off a specified test_screen.
pan_win()	Pan a test_screen inside a window.
pan_wkstABS()	Manual panning of a test_screen within a window at an absolute position.
pan_wkstREL()	Manual panning of a test_screen within a window at an position relative to the current position of the test_screen.
run_movie()	Starts a film rolling.
scroll_win()	Scroll a test_screen across the screen.
scroll_win_with_fix()	Scroll a test_screen across the screen with a tracking FIXSPOT.
set_CLT_load_index()	Sets the color lookup table index.
set_colorABS()	Resets the color lookup table one color at a time by changing a single value within a color lookup table.
set_colorREL()	Resets the color lookup table one color at a time by changing a single value within a color lookup table relative to its current value.
sweep_win()	Sweeps a test_screen.
sweep_win_with_fix()	Sweeps a test_screen with a tracking fixation window.
toggle_wins()	Toggles between two test_screens.

-

Example of CSS code using New and Old Graphics Routines

Excerpt from gtest.0 in the \demos directory (only parts of the gtest.0 file are shown below)

```
#include "css_inc.h"      //This comes in very
handy.include it always (it's in the \demos directory)
void circle(float deg_per_sec, float radius);    //function
prototype...just like ANSI C
```

```
main() {
#define CIRCLE_RAD 5.0
#define SPEED 60.0
   float  rad,speed;
   Gon_off(FIXSPOT, ON);          //Turn on fixation spot
request
   Gon_off(TEST1, ON);            //Turn on test_screen #1
request
   Gflush(1);                     //Process all pending
requests simultaneously and return when they are done
   circle(SPEED,CIRCLE_RAD);      //Run the circle function
(coded below)
   Gon_off(FIXSPOT, OFF); //Turn off fixation spot request
   Gon_off(TEST1, OFF);           //Turn off test_screen #1
request
   Gflush(1);                     //Process all pending
requests simultaneously and return when they are done
}
/** Making handy macros here so that our Gadd() calls are
easier to understand when read later **/
#define    GmoveRELrt(who,rad,theta)
Gadd(who,G_rt_MOVE_REL,1,(rad),(theta))  //uses radial
coords
#define    GpanRELrt(who,rad,theta)
Gadd(who,G_rt_PAN_REL,1,(rad),(theta))   //uses radial
coords
/** Specify degrees per sec (so 360 == 1 loop per sec) **/
#define    FRAMES_PER_SEC         60.
#define    PI                            3.1415926535
#define    DURATION       500                    // in
milliseconds
/** This subroutine runs and sweeps movies as well as moves
a bitmap around in a circle (while it is being **/
/** simultaneously bitpanned). A square is at the control of
the USER (using the arrow keys). **/
void circle(float deg_per_sec, float radius)
{
   float  rad_unit,theta_unit;
   float  angle = 0;
   float  fix_unit,fix_angle;
   long           count,start,stop;
   int            init = 0;
   int            pending=0;
   theta_unit = deg_per_sec / FRAMES_PER_SEC;
   rad_unit = (PI * 2 * radius) * (theta_unit / 360.);
   fix_unit=.05;
   fix_angle=0;
/* Request the movies be run and swept simultaneously...this
won't actually happen until Gflush() is called*/
   Gmovie(TEST0,DURATION,2,-100,1,1);    // set auto_on_off
to 1, so movie will shut itself off
   Gsweep(TEST0, 2, 45, DURATION,0);     // sweep the movie
while it runs
   Gmovie(TEST2,DURATION,1,100,0,1);     // set auto_on_off
to 1, so movie will shut itself off
   Gsweep(TEST2, 2, 135, DURATION,0);    // sweep the movie
while it runs
/* this is the loop that samples the keyboard for user input
and continually flushes the queue so that */
/* everything runs smoothly */
   while(1) {
           /* sample the keyboard, the keyboard keys are
defined in css_inc.h */
           if (KeyPressed()) {
                   switch(GetAKey()) {
                          case K_LEFTARROW: fix_angle=180;
break;
                          case K_RIGHTARROW: fix_angle=0;
break;
                          case K_UPARROW: fix_angle=90;
break;
                          case K_DOWNARROW: fix_angle=270;
break;
                          case K_PAGEUP: fix_angle=45;
break;
                          case K_PAGEDN: fix_angle=315;
break;
                          case K_HOME: fix_angle=135;
break;
                          case K_END: fix_angle=225; break;
                          case K_GREY_PLUS: fix_unit *= 2;
break;
                          case K_GREY_MINUS: fix_unit /= 2;
break;
                          default: Gpurge(); return;
                   }
           }
           /* Move and pan the bitmap */
           GmoveRELrt(TEST1,rad_unit,angle);
           GpanRELrt(TEST1,rad_unit,angle);
           /* Move the square in the way designated by the
keyboard input */
           GmoveRELrt(FIXSPOT,fix_unit,fix_angle);
           angle += theta_unit;
           /* nothing gets done until this is called...and
nothing is updated or changed either*/
           pending = Gflush(1);
   }
}
```

14. Input/Output Routines

This section includes routines both common to ANSI C and exclusive to CORTEX.
Almost all I/O calls are supersets of the base functions inp() and outp(), which can do just about any I/O operation in the hands of a trained professional.

Routine	**Use**
DEVinp()	Reads one byte from a specified port on a specific I/O device.

DEVinpw()	Reads a two-byte word from the specified I/O port on a specific I/O device.
DEVoutp()	Writes one byte to the specified I/O port on a specific I/O device.
DEVoutpw()	Writes a two-byte word to the specified I/O port on a specific I/O device.
inp()	Reads one byte from the specified I/O port.
inpw()	Reads a two-byte word from the specified I/O port.
outp()	Writes one byte to the specified I/O port.
outpw()	Writes a two-byte word to the specified I/O port.
byte_out()	Writes a byte to the A port of PIO24, if it is active. Superceded by outp().
byte_c_out()	Writes a bytes to the C port of PIO24, if it is active. Superceded by outp().
get_digital_input()	Reads a byte from the C port of PIO24, if it is active. Superceded by inpt().

15. Item Position Comparison Routines

This section describes the internal timing file calls that allow the user to easily compare the position of items on the graphics screen.

Routine	**Use**
contact()	Identifies when a circle and a rectangular object overlap each other.
distance_to_line()	Returns the minimum distance between a line of a specified slope passing through the origin and a point in visual space.
find_DC()	Computes the denominator constant (DC) of a line that stretches between the origin and a point in visual space.
find_slope()	Computes the slope of a line stretched between the origin and point in visual space.
in_corridor()	Computes whether a point in space is near (within a specified error) to a line that transects the origin and has a specified slope.
in_window()	Tests if a point is within a square target area.
ITEM_POSbind_fixspot()	Moves the fixation window to a specified item's position, and has the fixation window track to location of that item should it be moved.
ITEM_POSget()	Returns the center and size of any item, EYE_WIN, or fixation window.
ITEM_POSlut_index()	Returns the current LUT index for the requested item's color.

16. Keyboard-Console Input/Output Routines

During run-time in CORTEX there are two possible modes for the VGA USER screen: TEXT mode and GRAPHICS mode. This is true for all programs in DOS. CORTEX's default run-time screen mode is GRAPHICS mode (where you'll see the spike histograms and the EOG dis play, for instance). There may be times when the user wants to leave GRAPHICS mode and do keyboard-console I/O with standard text on the screen. An example of this would be if between trials, the user needed to choose the next trial type from a menu of choices. With these functions, one can change the mode to TEXT mode (using calls supplied in the USER SCREEN Manipulation section), display text in many formats, and then return to GRAPHICS mode and redraw the entire screen (again using calls supplied in the USER SCREEN Manipulation section).

17. Input routines

Routine	Use
GetAKey()	Waits for a key press and returns the key and its attributes.
getch()	Gets a character from the keyboard input stream.
gets()	Gets an entire string from the keyboard input stream.
KeyGet()	Waits for a key press and returns the key and attributes. Exactly the same as GetAKey().
KeyHit()	Checks if a key has been pressed. Exactly the same as KeyPressed().
KeyPressed()	Checks if a key has been pressed.
sprintf()	Formats and stores a series of characters.
scanf()	Get a string of input from the user, and parse into variables.
sscanf()	Parse a string into variables.

18. TEXT mode routines

Routine	Use
printf()	Prints a string at the current cursor location.
printxy()	Prints a string at the supplied coordinates with the supplied attributes (ie. color).
putchar()	Writes a single character to the current position of the cursor on the screen.
puts()	Writes a string to the current position of the cursor on the screen.

19. GRAPHICS mode routines

Routine	Use

MessageFloat()	Prints a float to the USER status screen.
MessageInt()	Prints an integer to the USER status screen.
MessageLong()	Prints a long integer to the USER status screen.
MessageString()	Prints a string to the USER status screen.
Mprintf()	GRAPHICS mode version of printf() to the USER status screen. Supercedes Message routines.

20. Math Routines

These functions are common math routines.

Routine	**Use**
abs()	Computes the absolute value of an integer.
acos()	Calculates the arc-cosine of a floating-point value.
asin()	Calculates the arc-sine of a floating-point value.
atan()	Calculates the arc-tangent of a floating-point value.
atan2()	Calculates the arc-tangent of two floating-point values.
ceil()	Calculates the smallest integer that is greater than or equal to a floating-point value.
clip()	Clips the range of an integer.
cos()	Calculates the cosine of a floating-point value.
cosh()	Calculates the hyperbolic cosine of a floating-point value.
exp()	Calculates the value of the constant *e* to the power of a floating point number.
fabs()	Computes the absolute value of a floating-point value.
fclip()	Clips the range of a floating-point value.
floor()	Calculates the smallest integer that is less than or equal to a floating-point value.
fmax()	Finds the maximum of two floating-point numbers.
fmin()	Finds the minimum of two floating-point numbers.
log()	Computes floating point natural log of a floating-point value.
log10()	Calculates the logarithm (base 10) of a floating-point value.
max()	Finds the maximum of two integers.
min()	Finds the minimum of two integers.

pow()	Calculates the value of one floating point number to the power of another.
random()	Calculates a random integer within a specified range.
sin()	Calculates the sine of a floating-point value.
sinh()	Calculates the hyperbolic sine of a floating-point value.
sqrt()	Calculates the square root of a floating-point value.
tan()	Calculates the tangent of a floating-point value.
tanh()	Calculates the hyperbolic tangent of a floating-point value.

21. Memory Management

These functions are common memory management routines and can be found in any C compiler.

Routine	**Use**
calloc()	Allocates an array in memory with elements initialized to 0.
free()	Frees (unallocates) a specific block of currently allocated memory.
malloc()	Allocates an array in memory.
realloc()	Reallocates a block of memory (change the size of a currently allocated block of memory).

22. Miscellaneous Routines

These functions are not easily categorized, but useful.

Routine	**Use**
dont_unload_conds()	Prevents CORTEX from unloading the current condition's worth of graphics information.
getenv()	Searches the list of environment variables for an entry corresponding to the specified variable name.
system()	Executes a DOS operating system command.

23. Mouse Monitoring

These functions allow the user to keep track of the mouse or track-ball device.

Routine	Use
MouseMoved()	Checks to see if the mouse has moved since last call and measures the distance in two axes.
MousePressed()	Checks to see if any of the mouse buttons are currently being pressed.

24. Permanent Variables in CORTEX

One potential weakness of programming in CSS has been that variables are erased between trials. So a list of redeclared permanent variables (permanent in the sense that they are not erased between trials) have been introduced into CORTEX version 5 to aid the programmer. The variable names are automatically recognized from anywhere within CSS (be it in the middle of the function, or external to all of them).

Other than using the EYE_WINS (see the next section) and these permanent variables, there is no way for the user to store information across trials....all variables cast from within the timing file are otherwise local (and thus their values last for only one trial). A list of the new permanent variables are shown below. What they will allow, for instance, is effective use of routines such as malloc() or calloc(). Say a novel staircasing design is required, and information about previous trials is needs to be retained across trials, the user could calloc() a chunk of memory and save the address of that memo ry into _pchar0 (one of the variable types listed below) and set _int0 = 1 (so that subsequent trials know that the memory has already been calloced and does not need to be calloced again). Then, the user can access the information within the _pchar0 array across trials. Static variables in CSS, as mentioned previously, only last the life of the trial: the user would use them, for example, when calling a function repeatedly or recursively from CSS, or if using the functions as part of a CSS thread.

The available external variables by type (initialized to 0 upon CORTEX start-up)	
Type	**Variables Available**
characters	_char0, _char1, _char2, ..., _char29
integers	_int0, _int1, _int2, ..., _int29
long integers	_long0, _long1, _long2, ..., _long29
floating-point	_float0, _float1, _float2, ..., _float29
Pointers	**Variables Available**
pointers to characters	pchar0, _pchar1, _pchar2, ..., _pchar29
pointers to integers	_pint0, _pint1, _pint2, ..., _pint29
pointers to long integers	_plong0, _plong1, _plong2, ..., _plong2
pointers to floating-point	_pfloat0, _pfloat1, _pfloat2, ..., _pfloat29

Handles (pointers to pointers)	**Variables Available**
handles to characters	_ppchar0, _ppchar1, _ppchar2, ..., _ppchar29
handles to integers	_ppint0, _ppint1, _ppint2, ..., _ppint9
handles to long integers	_pplong0, _pplong1, _pplong2, ..., _pplong9
handles to floating-point	_ppfloat0, _ppfloat1, _ppfloat2, ..., _ppfloat9

The user can add new permanent variables (the procedure for which is described in the CORTEX USER's manual) only by changing CORTEX's master code and recompiling the entire program.....so we suggest that you simply redefine (rename) the existing permanent variables to better fit the code you are writing. Say you need a few ints to store some parameters across trials, this is an example of how to do this:

```
#define initialized _int0         /* once they are renamed,
they work just like normal variables except*/
#define last_val _int1            /* that their value lasts
across trials */

void show_last_value()
{
   if (initialized)
           Mprintf(1,"last_val was %i", last_val);
}
```

25. String Handling

These functions are common string handling routines and can be found in any C compiler.

Routine	**Use**
atof()	Converts a string to a floating point value.
atoi()	Converts a string to an integer value.
atol()	Converts a string to a long value.
strcat()	Appends one string with another.
strchr()	Searches a string for the first occurrence of a specified character.
strcmp()	Compares two strings lexicographically.
strcpy()	Copies one string to another.
strdup()	Allocates storage space for and duplicates a string.
strlen()	Counts the number of characters in a string.
strstr()	Searches a string for the first occurrence of another smaller string.

26. Timer Routines

These functions comprise the new and old functions internal to CORTEX that provide the user with count-down timers. The MS_TIMER commands allow the user to access up to 32 separate timers. The set_timer() family of routines is older and manipulates a single timer. So calling, for instance, set_timer(), then set_random_interval() immediately afterwards, will negate the first command. Keep in mind that another way to measure a length of time during a trial is to read the trial counter (a global variable called TIMERms_counter) directly, as in this example:

```
/* This file shows the user how to read the trial clock
directly */
#include "css_inc.h"
main()
{
           long start = TIMERms_counter;

           <do something...>

           long duration = (TIMERms_counter - start);
}
```

The value of duration will be the amount of time needed to <do something...>. Below is a listing of the TIMER Routines:

Routine	**Use**
MS_TIMERcheck()	Checks the number of milliseconds left in the count-down of current timer set by MS_TIMERset().
MS_TIMERset()	Starts and sets the total length (in milliseconds) of a count-down timer.
set_random_interval()	Sets the timer to count-down from a number which is randomly chosen to be within a specified range of milliseconds.
set_random_timer()	Sets the timer to count-down from a number which is randomly chosen.
set_timer()	Sets a timer.
timer_expired()	Returns whether or not the timer has expired. This routine only works for the set_timer() family of routines.

27. Data Type Definitions and Retrieving Modified Information from Routines with CORTEX

When programming with the CORTEX STATE SYSTEM (CSS), as in programming in C, the user needs to define the data type of each variable so that the compiler knows how much memory to acquire for the program. Every variable has a type, including subroutines. This is actually very clever since the compiler knows what type of information the subroutine will return to the calling routine (if any at all). The CSS has most of the types that regular C compilers have, such as:

Type	Abbreviation	Amount of Memory
characters	char	8-bits
integers	int	16-bits
long integers	long	32-bits
floating-point	float	32-bits

The CSS does not, however, have unsigned data types, or data types such as short, or double long integer (commonly called a "double"). So there are some potentially important differences/limitations of CSS compared to other C compilers. When passing/retrieving information to/from a subroutine, things are fairly simple so long as you only want a single variable returned from the subroutine. For example, let's say you had an integer and you wanted to write a subroutine to do a mathematical operation on it:

```
/* This file is a demo file showing the use of a subroutine
and how it can pass information back to the */
/* calling routine */
#include "css_inc.h" /* it's best to include this in CSS
files....it often comes in handy */
int coolness_multipilier (int someones_coolness); /* This is
the function prototype */
main()
{
   int     steves_cool = 10, steves_final_coolness;
   int     earls_cool     = 10, earls_final_coolness;
   steves_final_coolness =
coolness_multiplier(steves_cool);
   earls_final_coolness = earls_cool;
   Mprintf(1, "steves_final_coolness = %d",
steves_final_coolness);
   Mprintf(2, "earls_final_coolness = %d",
earls_final_coolness);
}
/* Here's the subroutine...notice that it is defined as an
int data type....*/
/*  that means it is returning a value that is an integer
(16-bits in size)*/
int coolness_multiplier (int someones_coolness)
{
   return(someones_coolness*10);
}
```

The final result, of course, is that Steve is 10 times cooler than Earl. But it also shows a subroutine can only return a SINGLE variable ...and even then the subroutine must be defined as that variable type to do so. This is rather limiting; how could we have a subroutine modify many numbers without calling the subroutine many times? The answer is by using *pointers* to variables. Pointers are special types of variables which hold the memory address to the data instead of the data itself. This means that a pointer can hold the address of a list of numbers (a list of any size and type)...thus when manipulating the list one only need pass a single variable (the pointer--the address of the entire list). A regular variable cannot do this. To delineate pointers from the other data types, CSS requires that a 'p' be placed in front of the *type* name. This is different than in the C programming language in which an asterisk is place before the *variable*'s name:

Type	**Abbreviation in CSS**	**Correlate in C**
pointers to characters	pchar	char **variable*
pointers to integers	pint	int **variable*
pointers to long integers	plong	long **variable*
pointers to floating-point	pfloat	float **variable*

To demonstrate how pointers can be used to pass/return information to/from subroutines, the earlier example is rewritten below with pointers for your convenience. As you will see, it seems to be more complicated than the earlier example, but keep in mind that the list being manipulated is only 2 members long....if there were 5000 members in the list it would be MUCH shorter than an equivalent piece of code written without pointers. Thus, it is best to use pointers for medium or large lists (or lists that vary in size). An intermediate or advanced programmer will also note that there are redundancies in the following code (ie. there is no reason to return the pointer since the memory has been changed), but this is left in to keep a sense of continuity with the earlier example.

```
/* This file is a demo file showing the use of a subroutine
and how it can pass information back to the */
/* calling routine using pointers */
#include "css_inc.h" /* it's best to include this in CSS
files....it often comes in handy */
pint coolness_multiplier(pint coolness_list);
main()
{
   int     steves_cool = 10, earls_cool = 10;
   pint    coolness_list;
   /* get memory for the data pointed to by coolness_list,
notice that I cast the output of */
   /* of calloc() as a pint so as to match the data type of
coolness_list */
   coolness_list = (pchar) calloc(2, sizeof(pint));
   coolness_list[0] = steves_cool;
   coolness_list[1] = earls_cool;
   coolness_multiplier(coolness_list);
   Mprintf(1, "steves_cool = %d", coolness_list[0]);
   Mprintf(2, "earls_cool = %d", coolness_list[1]);
}
/* Here's the subroutine...notice that it is defined as a
pint data type....        */
/* that means it is returning a value that is a pointer to
an integer */
pint coolness_multiplier (pint coolness_list)
{
   coolness_list[0] = coolness_list[0]*10;
   return coolness_list;
}
```

In summary, the user must define each variable as a specific data type so that the compiler knows to acquire enough memory for the program. When manipulating lists or arrays, using a pointer to the variable type that makes up the list (or array) will allow easy processing of those lists (instead of passing each member of a list one-at-a-time). Of course, the usefulness of pointers goes much deeper than what has been discussed here, but the basic concept of a pointer versus a normal variable type is all that is necessary for simple programs.

28. USER SCREEN Manipulation

During run-time in CORTEX there are two possible modes for the VGA USER screen: TEXT mode and GRAPHICS mode. This is true for all programs in DOS. CORTEX's default run-time screen mode is GRAPHICS mode (where you'll see the spike histograms and the EOG display, for instance). There may be times when the user wants to switch between TEXT mode and GRAPHICS mode. An example of this would be, if between trials, the user needed to choose the next trial type from a menu of choices. With the following functions, one can change the mode to TEXT mode, display text in many formats (using the calls listed in the Keyboard-Console Input/Output Routines section), and then return to GRAPHICS mode and redress the entire screen.

1. Generic routines

Routine	Use
SCREENmode()	Changes the mode between TEXT mode and GRAPHICS mode.

2. TEXT mode routines

Routine	Use
Cls()	Clears the TEXT screen.
CurMov()	Moves the position of the TEXT mode cursor.

3. GRAPHICS mode routines

Routine	Use
display_eye_path()	Draws the eye-movement data collected since put_eye_data_in_buf() was last called.
display_histogram()	Causes the histogram for the current condition to be displayed.

display_trial_progress()	Turns on or off the current trial's cumulative on-line data display (raster and histograms).
DrawBox()	Draws a box on the EOG display.
histogram_Ctik()	Draws a colored tik mark on the current bin of the histogram(s).
histogram_tik()	Draws a tik mark on the current bin of the histogram(s).
ITEM_POSmark_pos()	Shows the size and location of a specified item on the EOG display.
ITEM_POSmark_all()	Shows the size, location, and visibility (colored vs. invisible) of all items on the EOG display.
SCREENdraw_entire_screen()	Redraws the entire CORTEX USER screen.
update_histogram()	Update the histogram with the current trial's data.

-

3. EVENT CODES

0	NOCODE
1	SPIKE1
2	SPIKE2
3	REWARD
4	BAR_UP
5	BAR_LEFT
6	BAR_RIGHT
7	BAR_DOWN
8	FIXATION_OCCURS
9	START_INTER_TRIAL
10	END_INTER_TRIAL
11	START_WAIT_FIXATION
12	END_WAIT_FIXATION
13	START_WAIT_LEVER
14	END_WAIT_LEVER
15	START_PRE_TRIAL
16	END_PRE_TRIAL
17	START_POST_TRIAL

18	END_POST_TRIAL
19	START_PAUSE
20	END_PAUSE
21	START_RANDOM_PAUSE
22	END_RANDOM_PAUSE
23	TURN_TEST0_ON
24	TURN_TEST0_OFF
25	TURN_TEST1_ON
26	TURN_TEST1_OFF
27	TURN_TEST2_ON
28	TURN_TEST2_OFF
29	TURN_TEST3_ON
30	TURN_TEST3_OFF
31	TURN_TEST4_ON
32	TURN_TEST4_OFF
33	TURN_TEST5_ON
34	TURN_TEST5_OFF
35	TURN_FIXSPOT_ON
36	TURN_FIXSPOT_OFF
37	START_FIXSPOT_DIM
38	START_UP_LEVER
39	END_UP_LEVER
40	START_LEFT_LEVER
41	END_LEFT_LEVER
42	START_RIGHT_LEVER
43	END_RIGHT_LEVER
44	START_EYE1
45	END_EYE1
46	START_EYE2
47	END_EYE2

48	TURN_TEST6_ON
49	TURN_TEST6_OFF
50	TURN_TEST7_ON
51	TURN_TEST7_OFF
52	TURN_TEST8_ON
53	TURN_TEST8_OFF
54	TURN_TEST9_ON
55	TURN_TEST9_OFF
56	START_SCROLL_BITMAP_TEST0
57	END_SCROLL_BITMAP_TEST0
58	START_SCROLL_BITMAP_TEST1
59	END_SCROLL_BITMAP_TEST1
60	START_SCROLL_BITMAP_TEST2
61	END_SCROLL_BITMAP_TEST2
62	START_SCROLL_BITMAP_TEST3
63	END_SCROLL_BITMAP_TEST3
64	START_SCROLL_BITMAP_TEST4
65	END_SCROLL_BITMAP_TEST4
66	START_SCROLL_BITMAP_TEST5
67	END_SCROLL_BITMAP_TEST5
68	START_SCROLL_BITMAP_TEST6
69	END_SCROLL_BITMAP_TEST6
70	START_SCROLL_BITMAP_TEST7
71	END_SCROLL_BITMAP_TEST7
72	START_SCROLL_BITMAP_TEST8
73	END_SCROLL_BITMAP_TEST8
74	START_SCROLL_BITMAP_TEST9
75	END_SCROLL_BITMAP_TEST9
76	START_SCROLL_SCREEN_TEST0
77	END_SCROLL_SCREEN_TEST0

78	START_SCROLL_SCREEN_TEST1
79	END_SCROLL_SCREEN_TEST1
80	START_SCROLL_SCREEN_TEST2
81	END_SCROLL_SCREEN_TEST2
82	START_SCROLL_SCREEN_TEST3
83	END_SCROLL_SCREEN_TEST3
84	START_SCROLL_SCREEN_TEST4
85	END_SCROLL_SCREEN_TEST4
86	START_SCROLL_SCREEN_TEST5
87	END_SCROLL_SCREEN_TEST5
88	START_SCROLL_SCREEN_TEST6
89	END_SCROLL_SCREEN_TEST6
90	START_SCROLL_SCREEN_TEST7
91	END_SCROLL_SCREEN_TEST7
92	START_SCROLL_SCREEN_TEST8
93	END_SCROLL_SCREEN_TEST8
94	START_SCROLL_SCREEN_TEST9
95	END_SCROLL_SCREEN_TEST9
96	REWARD_GIVEN
97	START_EXTRA_LEVER
98	END_EXTRA_LEVER
99	BAR_EXTRA
100	START_EYE_DATA
101	END_EYE_DATA