

Psychological Toolbox:
C/C++ functions for Psychology Experiments

Version 0.7

Matt Peterson

The University of Kansas

OPENING REMARKS **4**

INTRO **6**

GETTING STARTED	6
PROGRAM LAYOUT	7
HEADER FILES	10
GLOBAL VARIABLES	10
THE MAIN() FUNCTION	11
EXPERIMENT DESIGN	13
EXPERIMENT IO	16

BASIC TOPICS **21**

PRACTICE TRIALS	21
RANDOMIZATION	21
GRAPHICS	23
DRAWTOSCREEN	23
COLORDEPTH	23
COLORS	23
USING AND FILLING RECTANGLES	24
TEXT	25
TCARDS	26
TIMERS	30
GETTING RESPONSES	30
KEYBOARDPOL()	30
GETAKEY()	31
VOICEKEY	31

ADVANCED TOPICS **33**

INSERTING BREAKS	33
DESIGNS WITH UNEQUAL N'S	33
BLOCKED EXPERIMENTAL DESIGNS	34
DISK AND CONSOLE IO	37
FORMATTED INPUT AND OUTPUT	37
A QUICK OVERVIEW OF COMMANDS	38
WORDS AND STRINGS	39
WHAT IS A STRING?	39
IMPORTING STRINGS FROM A TEXT FILE	39
STRING MANIPULATION FUNCTIONS	40
GRAPHICS	42
COLORS	42
TRANSFER MODES	43
ADVANCED TCARD FUNCTIONS	45
MASKS	45
MASKTOCARD()	45


COPYTOCARD()	45
RESOURCES	45
REEDIT	45
CREATING NEW RESOURCES	46
USING RESOURCES IN YOUR PROGRAM	46
PICTURE (PICT) RESOURCES	46
SOUND (SND) RESOURCES	47
KEYBOARD POLLING ACCURACY ISSUES	49
METHOD	49
RESULTS	50
CONCLUSIONS	52
TURNING THE MOUSEOFF()	53
KEYBOARDPOL() OR GETAKEY()?	53
SYNCANDPOLL()	53
DEFINITIONS	54
<hr/>	
COMMON VARIABLE TYPES	54
NOTES ON POINTERS – WHEN TO USE * OR &	54
POINTERS, ARRAYS, AND STRINGS	55
HANDLES	56
COMMON STRUCTURES AND VARIABLES	57
COMMON MACINTOSH FUNCTIONS	57
COMMON C/C++ FUNCTIONS	59
INPUT/OUTPUT FUNCTIONS	59
STRING MANIPULATION FUNCTIONS	60
PSYCHOLOGICAL TOOLBOX GLOBAL VARIABLES	62
PSYCHOLOGICAL TOOLBOX FUNCTIONS	62
INITIALIZATION AND ENVIRONMENT FUNCTIONS	62
TIMING AND RESPONSE FUNCTIONS	63
GRAPHICS FUNCTIONS	65
COLOR FUNCTIONS	66
RANDOMIZATION AND ARRAY FUNCTIONS	68
STRING FUNCTIONS	69
MATH FUNCTIONS	69
PSYCHOLOGICAL TOOLBOX OBJECTS	70
PSYCHOLOGICAL TOOLBOX STRUCTURES	70
REFERENCES	71
<hr/>	
PRODUCTS	71
PUBLICATIONS	71

Opening Remarks

The routines contained in the *Psychological Toolbox* and the programming philosophy expressed throughout this guide represent the culmination of 3 frustrating years of programming (the last two were a breeze, since I finally knew what I was doing). When I came to the University of Kansas, I had some knowledge of FORTRAN, a deep knowledge of BASIC, and an intermediate knowledge of Pascal. All proved useless. FORTRAN wasn't used much, all my BASIC knowledge centered around the Apple II and some DOS-PC, and I hated Pascal.

So I had to start from scratch. Everyone seemed to be using C, and although it is very similar to Pascal, it had two advantages for me: it wasn't nearly as wordy, and it was less structured. I spent 3 years learning how to program in C, program the Macintosh Toolbox, and design the layout and logic of my programs. I hope that no one ever has to go through that again!

I've tried to design the *Psychological Toolbox* so that a lot of the Macintosh Toolbox and C++ code is transparent from the user. Most people should find this pretty easy to pick up, and it should be no more difficult (if not easier) than programming SPSS or SAS.

Roughly half of these routines are platform independent. That is, they can be used on any machine that has a C++ compiler. The rest of the routines are Macintosh specific, and these are marked by a .

My overriding goal when designing this guide and the functions and objects available in the *Psychological Toolbox* was to allow researchers to quickly write error free experiments. I have tried to accomplish this by keeping everything as simple as possible. This has meant using arrays and functions when possible (the graphics objects are the exception), and staying away from objects, structures, linked-lists, binary trees, and the lot. Although objects and lists do have advantages in some situations, such as writing a database program to manage words or sentences, I think that they can add unnecessary complications in some situations.

Future Directions:

Ultimately, I would like for these functions to be cross platform compatible between MacOS and DOS machines. However, MacOS 7.x will not be around for too much longer and is due to be replaced by a UNIX/NeXTStep based MacOS 9 (Rhapsody) in January 1998. OpenStep is reported to be a very easy environment to program for, and I suspect the functions and classes presented in this manual will be easy to translate to MacOS 8. The good news is that MacOS 8 will be available for Intel machines (as a replacement for the current version of OpenStep for Intel), which means that any programs written for MacOS 8 for the PowerPC will work on Intel machines with a simple recompile.

Acknowledgments:

Not all of the code in these routines is original. I would like to thank Nick Prins for the ideas behind the randomization routines (don't ask what I was trying to do before he came along!) and the method for inserting breaks into the experiment. I would also like to thank the guys who wrote Think Reference, the guys at Metrowerks (the makers of CodeWarrior), and the MacPsych discussion archives for giving me hints on how to write the ADB keyboard polling routines.

Matt Peterson
June 1997

Styles used throughout this guide:

Commentary

Input/Output results

C/C++ Code

`function()`

`function(..., ...)`

`function(short, double)`

`double function(short)`

🍏

∞

`void function(short varname)`

`void function(char * varname)`

12 point Times

10 point Courier

9 point Monaco

a function that takes no parameters

a function that expects 2 parameters, but are not stated in order to save space

a function that takes 2 parameters, a short and a double

a function that takes a short and returns a double

Macintosh specific variables or functions

Function will work on PC and Mac

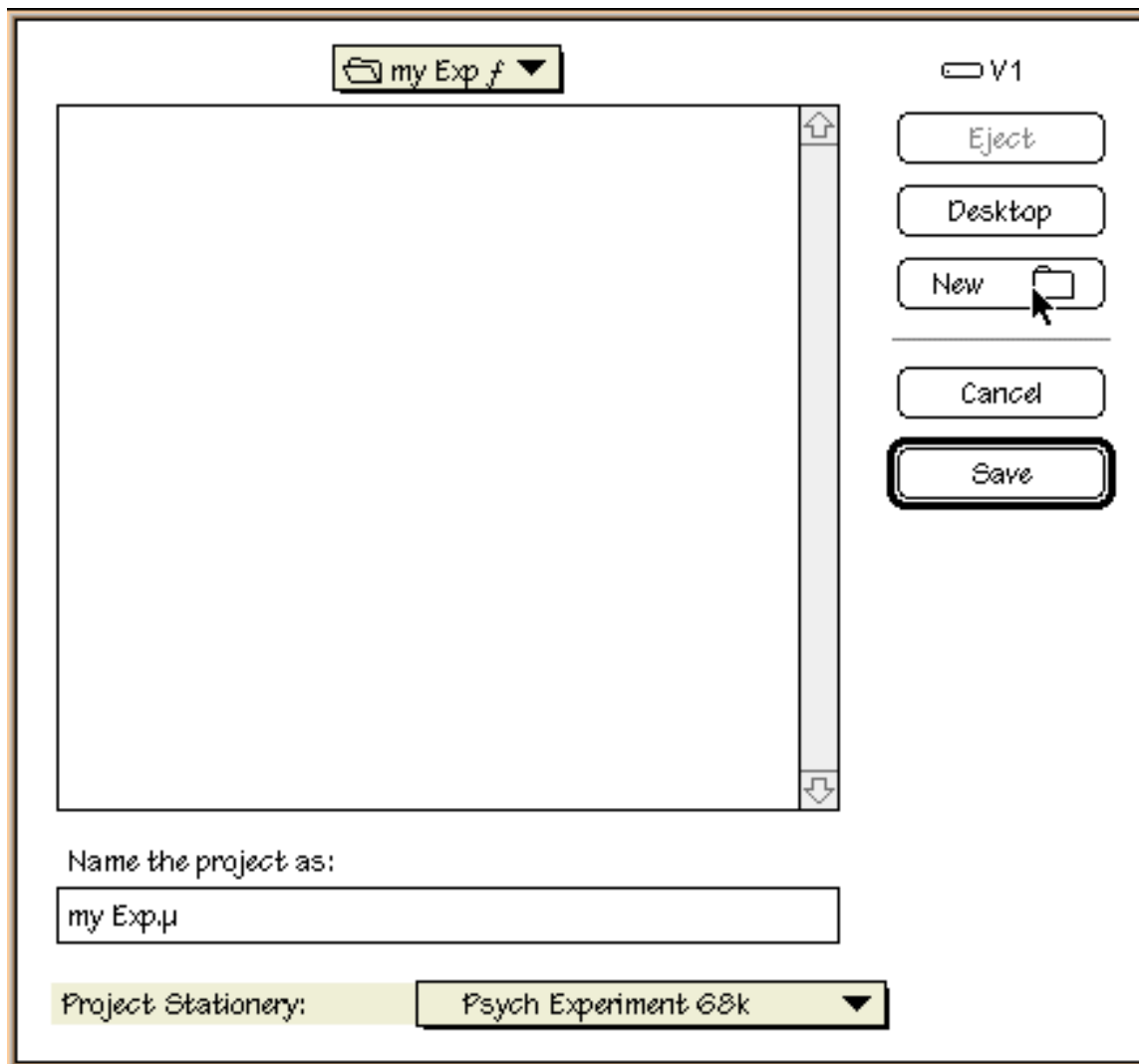
the italicized parameter name is a short

the italicized parameter name is a pointer to a char

Intro

Getting Started

The first thing that you need to do is start a new project. Select “New Project ” from the “File” menu. You will be presented with a dialog box that looks like this:



Name your project and select a folder to put it in. Remember to add μ (*option + m*) to the end of your project name. You also need to select the project stationery to use, either *Psych Experiment 68k* or *Psych Experiment PPC*. The project stationery will start you out with skeleton of a program that you can use to base your experiment on. It also ensures that all of the proper *Psych Experiment Library* files are included.

Stationary Files – stationary files, denoted by the < and > around their name, are automatically added to a new project. As a general rule, you should never modify these files. The one exception is if you want to customize the *Psych Experiment* stationary files to suit your own needs.

You are provided with several stationary files containing the beginning pieces of a psychology experiment program. These files are indicated by the < and > surrounding their names. You will need to rename these files so that you can use them in your experiment.

File	Code	Data	
▶ Segment 1	0	0	☑
▶ Segment 2	0	0	• ☑
▶ Segment 3	0	0	☑
▼ Segment 4	0	0	• ☑
<replace me Psych>.cpp	0	0	• ▶
<psych design>.cpp	0	0	• ▶
<psych io>.cpp	0	0	• ▶
18 file(s)	0	0	

Program Layout

Psychology Experiment programs tend to follow this general layout:

- 1) Initialization
 - a) get the information about the subject
 - b) initialize the environment
 - c) define and randomize the trials
- 2) Present the practice trials
- 3) Loop through the trials
 - a) make the stimuli
 - b) start the timer (if applicable)
 - c) show the stimuli
 - d) get the subjects' response and stop the timer
- 4) Housecleaning.
 - a) Save and analyze the data
 - b) reset the environment
- 5) Quit

The <replace me Psych>.cpp provides a general framework for writing a program. After you have renamed and saved this file (e.g. "myExp.cpp"), you can modify this file to suit your needs.

Text Colors – CodeWarrior allows predefined text to automatically be colored. The default settings for the *Psych Experiment Library* are:

Blue – standard ANSI C and C++ keywords
Orange – comments or commented out lines
Green – predefined *Psychological Toolbox* functions.

```

<replace me Psych>.cpp

#include "PsychRoutines.h"
#include "myExp.h"

char      Response[NumTrials];
double    RT[NumTrials];
extern short OrderArray[NumTrials];
extern short Order;
extern Rect screenRect; // externally defined as 640x480

void      Quit(void);
void      DoPractice(void);

void main (void)
{ short i;

  GetSubjectInfo();
  //
  InitPsychRoutines(); // Required
  WindowInit(0); // 0 = 640 x 480, 1=832x624
  //
  MakeTrials();
  RandTrials();
  // other initialization stuff, e.g. "InitTcards();"
  HideCursor();
  HideMenuBar();
  BackToBlack(screenRect);
  //
  // "DoPractice()", "ShowFixation("
  for( i = 0; i < NumTrials; i++)
  { Order = OrderArray[i]; // the trial to present nex
    // MakeCues(Order); // offscreen drawing
    // MakeStims(Order); // offscreen drawing
    // ShowCues(); // display the cue scre
    WaitSyncs(10); // as
    StartTimer(); // isecnd t
    // ShowStims(); // display the stimuli
    Response[Order] = KeyBoardPol(); // waits for a keypress, the
    RT[Order]=StopTimer(); // tops the timer and recor
    WaitSyncs(75); // wait 75 refreshes
  }
  Quit(); // Save and Analyze Data ets eve
}

//=====
void      Quit(void)
{      Cleanup();
      SaveData();
      exit(0);
}

//=====
void      DoPractice(void) // Same as main loop, but 1) does no
{ short i; // and 2) randomly samples trials
  short PractOrder[15]; // The randomly sampled practice tri

```

Header files

PsychRoutines.h – Any file that uses *Psych Experiment Library* functions must #include the PsychRoutines.h header file.

myExp.h – Like the template files, this is a generic main header file that you can modify for your own uses. It contains some very useful definitions:

#define NumTrials 100	A global constant that defines how many trials are in the experiment. With careful program design, you should be able to change this one constant to change the number of trials.
#define ColorDepth 8	The color depth to be used in this experiment. Color depth not only effects how many colors can be displayed, but also effects how quickly images can be drawn to the screen and how much memory the images occupy.
#define centerX 319	The horizontal center of the screen. This number can be changed to suit the size of your monitor.
#define centerY 239	The vertical center of the screen. In this example, the point (319,239) is the center of a 640x480 pixel monitor (the standard size for 13 and 14 inch monitors).

Global variables

For the most part, the global variables you use are up to you. The global variables included in this example are simply suggestions to help you get started. Most of the global variables listed above are arrays, the exception being `Order` and `screenRect`.

<code>Response[...]</code>	The subject's response. Usually the key pressed. (A character)
<code>RT[...]</code>	The subject's response time for the keypress. (double precision floating point)
<code>OrderArray[...]</code>	The random order in which the trials will be given.

*Note that the size of the arrays are defined as `[NumTrials]` in the example code. Defining the size of your arrays in this manner allows you to easily change their size by editing the defined value of `NumTrials` in the header file `myExp.h`, e.g

```
#define NumTrials 100
```

<code>Order</code>	The value of <code>OrderArray[i]</code> for trial <code>i</code> . It is for aesthetics only – it makes the Main Loop easier to read.
<code>screenRect</code>	A global variable defined elsewhere in the <i>Psych Experiment Library</i> . <code>screenRect</code> is a <code>Rectangle</code> the size of the screen. It can be used for such tasks as clearing the screen – e.g. <code>BackToBlack(screenRect)</code> draws a black rectangle the size of the screen at the screen's coordinates.

extern - the keyword `extern` is simply a way of telling the compiler that the variable was defined in another file. A variable must always be defined somewhere at least once. Each `.cpp` file that uses your global variables must know the definitions of those variables. Using `extern` allows those `.cpp` files to know what the definitions of these global variables. If you omit `extern`, the variable will be defined twice and will cause an error message to occur.

The main() function

The `main()` function is the master function that controls your program. When you run your program, `main()` is the function that is automatically called first. The `main()` function contains the general flow and layout of your program and calls other functions to carry out their specialized tasks. The layout of `main()` generally consists of:

Program Initialization

- a) get the information about the subject
- b) initialize the environment
- c) define and randomize the trials

(Practice Trials)

The Main Loop

- a) make the stimuli
- b) start the timer (if applicable)
- c) show the stimuli
- d) get the subjects' response and stop the timer


Housecleaning.

- a) Save and analyze the data
- b) reset the environment

Program initialization

Program initialization includes all of the tasks that you need to do before the experiment (and practice trials) begins. While some of the following examples are specific to the *Psychological Toolbox*, most routines are not. The general layout is a good guide as to how you might want to write your own programs.

`GetSubjectInfo()` – (included in `<psych_io>.cpp`) The purpose of this function is to get all of the information that might be relevant about the subject you will be running. It uses a simple console window to ask for such things as the subject's ID#, which experimental group the subject is in (if applicable), etc. I usually like to define the name of the subject's data file in this routine.

`InitPsychRoutines()` –  function included with the *Psychological Toolbox*. It takes care of and hides a lot of the dirty work, such as seeding the random number generator, defining some functions, and initializing the Macintosh toolbox. It must be called early in the program to ensure that the functions contained in the *Psychological Toolbox* work correctly.

`WindowInit(0)` – 🍏s another *Psychological Toolbox* specific function. It initializes the graphics window that will be used during the experiment. 0 = 640 x 480 (~13-15 inch monitor), while 1 = 832x764 (generally 15-17 inch monitors).

`MakeTrials()` – is contained in the file <psych design>.cpp. `MakeTrials()` defines the variables that will be used to describe your trials. For example, for a cueing study 80% of your trials might be valid cues, while 20% might be invalid cues. You could use `MakeTrials()` to fill an array, `Validity[NumTrials]`, with 80 1's (valid) and 20 0's (invalid).

`RandTrials()` – A psychology experiment (usually) would not be worth much if the trials were predictable, right? For example, you usually would not want the first 80 trials in a cueing experiment to be valid and the last 20 trials invalid. To randomize the order of the trials, a good practice is to fill an array (such as `OrderArray[NumTrials]`) full of the ordered trial numbers and then shuffle the order of the trials in `OrderArray[...]`.

`HideCursor()` – 🍏s another Macintosh specific command. It makes the cursor or arrow invisible.

`HideMenubar()` – 🍏removes the menubar from the graphics window.

`BackToBlack(screenRect)` – 🍏`BackToBlack()` fills a rectangle (in this case, `screenRect`) with black. Other examples of this include `BackToWhite()` and `BackToGray()`. You can easily hack your own variations.

The Main Loop

The job of the main loop is to loop through your trials until the experiment is completed. An easy way to do this is use a `for(..., ..., ...)` statement:

```
for( i = 0; i < NumTrials; i++)
{
  ...
}
```

The above example will take the variable `i`, set it to 0, increment variable `i` (`i++`) each time the loop is executed, and exit the loop when `i` is no longer less than the number of trials (`i < NumTrials`).

Order: To make the program easier to read, I like to use the variable `Order` instead of `OrderArray[i]`. I do this by setting `Order = OrderArray[i]` in first line of the Main Loop. Remember that `OrderArray[...]` contains the randomly shuffled trial numbers. Thus, `OrderArray[0]` might be equal to 57. This is equivalent to saying that this subject's first trial will be trial #57, as defined in `MakeTrials()`.

🍏When I draw the stimuli I will be presenting to the subject, instead of drawing the stimuli in real time, I draw the stimuli offscreen, usually during the intertrial interval (see `MakeCues(Order)` and `MakeStims(Order)`). To ensure that I can count on the stimuli being drawn to the screen in one refresh (which means that I can rely on the presentation duration), I first draw the stimuli into offscreen `Tcards` (an object class used to emulate tachistoscope cards). I then blast the images to the screen (located in the function `ShowStims()`). Most images can be copied to the screen in one refresh, but see the `Tcard` section for limitations.

So, let's say that we have now called our functions to draw our stimuli into the `Tcards`. You typically would want to wait x amount of time, start the timer, show the stimuli, wait for a response, and then stop the timer (assuming you are doing response time experiments). This sequence might look like this:

```
for( i = 0; i < NumTrials; i++)
{
    ...
    WaitSyncs(10);
        Wait 10 screen refreshes. For a 66.7 Hz monitor, 1 refresh = 15 ms.
    StartTimer();
        Starts the millisecond timer
    ShowStims(); // myCard.Show()
        Blasts the images from your Tcards onto the screen.
    Response[Order] = KeyBoardPol();
        Continuously polls the keyboard until a key is pressed. Returns the value
        of the key.
    RT[Order] = StopTimer();
        KeyBoardPol() automatically returns once a key is pressed. Therefore, the
        next function to execute is StopTimer(), which stops the timer and puts the
        elapsed time (in milliseconds) into the array RT[...].
    WaitSyncs(75);
        Waits 1125 ms (75x15ms refreshes). A fairly quick intertrial interval.
}

```

Housecleaning

The last series of steps your program will need to carry out after the Main Loop has presented all of the trials to your subject is what I categorize as *Housecleaning*. This includes such things as making the cursor and the menubar visible again, changing the color depth of the monitor back to its original settings (say, from black and white back to 16-bit color), turning the mouse back on, and so forth. Luckily most of these functions are contained in the function `CleanUp()`.

Lastly, when all is said and done, you don't want to forget to save your data! An example of a `SaveData()` function (using standard ANSI C++) is contained in the file `<psych io>.cpp`.

Although it might not always be necessary, good etiquette says that the last line of your program should be `exit(0)`.

Experiment Design

Getting the design of your experiment correct is one of the most crucial aspects of writing your program. It is probably the most difficult part of writing your program, but if you follow these examples you should be able to stay out of trouble. First, let's take a look at the header file `PsychDesign.h`:

```
// PsychDesign.h

void  MakeTrials(void);           // function prototypes
void  RandTrials(void);
```

There really is very little to this header file. One thing it contains are the prototypes for the functions contained in `PsychRoutines.cpp`. Any files that call these routines must `#include "PsychDesign.h"` at the beginning of the file.

MakeTrials()

```

#include "PsychRoutines.h"
#include "myExp.h"

short  TargColor[NumTrials];    // 0 = r, 1 = g, 2 = b, 3= y, 4= o
short  Direction[NumTrials];    // 0 = left, 1 = right.
short  OrderArray[NumTrials]

//-----
void MakeTrials(void)
{
    short  i, j, k, m;          // local variables
    const  short  Cells=10;     // 2 x 5
    const  short  Repetitions = // # of Repetitions is dependent
                NumTrials/Cells; // on Cells and NumTrials
    short  which=0;            // Used to calculate which
                                // trial we are on.

    for(i = 0; i< Repetitions; i++) // repetitions
    {
        for( j = 0; j< 2; j++)      // left or right
        {
            for( k = 0; k< 5; k++)  // color
            {
                which++;           // which trial number
                //---
                Direction[which] = j;
                TargColor[which] = k;
            }
        }
    }
}
//=====
...

```

First, let's start out with an easy experimental design. In this experiment, subjects are cued (100% validity) to the location where a target will occur. The target can occur at the left or right of fixation. The target is drawn in one of five colors: red, green, blue, yellow, or orange. The design is a rather simple 2 x 5 (direction x color) factorial design.

```

short  TargColor[NumTrials];    // 0 = r, 1 = g, 2 = b, 3= y, 4= o
short  Direction[NumTrials];    // 0 = left, 1 = right.

```

TargColor and Direction are our global variables. We use global variables because a) they are easy to use, b) they are easy to use, c) why not? Other files (probably the graphics and io files) will have to define these variables using extern in order to be able to use them.

```

const  short  Cells=10;
const  short  Repetitions = NumTrials/Cells;

```

This defines the number of repetitions (or samples) for each cell of our design. const short is very similar to using #define. const says that you can never change the value of that variable, because it is a constant.

We know that we have 10 cells in our design (with equal n's), we also know that the number of repetitions per cell is dependent on the number of trials. Therefore, if we change the number of trials from 100 to 200, the number of repetitions per cell would automatically change from 10 to 20.

The rest of the MakeTrials() function is fairly straightforward. We loop through each level of each variable in our design. which is used to keep track of which trial we are currently defining. Each time we loop through, the counter which is incremented by one.

RandTrials()

`RandTrials()` is a rather simple function. All it does is generate a list of numbers from `0...NumTrials`. After that, it shuffles the order of the numbers and puts that value into the `OrderArray`.

The example below might not look like it is doing all of that, but it is. Actually, since generating a random list of numbers is such a common task, I made that task into the function `RandomArray(..., ...)`. Why call the function `RandTrials()` in the first place? Later, we will be building on this function to allow us to randomize blocked designs.

```
//=====
void  RandTrials(void)
{
    RandomArray(OrderArray, NumTrials);
}
```

First, let's take a look at `RandomArray(..., ...)`

```
void RandomArray(short * Array, short Length)
```

What is a `short *`? It is a pointer to a list of `shorts`. Am I losing you? Well don't worry – the name of an array sans brackets `[]` is a pointer to the beginning of an array. Note that this function is taking a pointer to a list of `shorts`. That is, don't expect it to generate a list of `longs`, `floats`, `doubles`, or `Rects`. Since `shorts` range from `-32768...32767`, that gives us plenty of trials to work with (I can't imagine more than 1500 trials in a session) without wasting memory.

We also need to pass `RandTrials (... , ...)` the length of the array. Well, we know how long the array containing the order of the trial presentations is, it must be as long as the number of trials, or `NumTrials` long. Wasn't that simple?

Experiment io

Computers are a great way to collect data, but the data is of no use to us unless it is saved to disk! Your program will usually have at least two *io* (input/output) functions: a function to get information about the subject and a function to save the data. First, let's start with the function `GetSubjectInfo()`, which is located in the file `<psych io>.cpp`.

GetSubjectInfo()

```
#include "PsychRoutines.h"
#include "myExp.h"

extern double RT[NumTrials];           // Response Times
extern short OrderArray[NumTrials];    // The random order of the trials
extern short Correct[NumTrials];       // Correct Response?
extern short TargColor[NumTrials];     // Target Color
extern short Direction[NumTrials];     // Cue direction
char FileName[30];                     // a string

void SaveData(void)
{
...
}

//=====

void GetSubjectInfo(void)
{ char SessionString[2];

  printf("ID Number:\t");               // Ask for the subject's ID #.
  scanf("%s", FileName);                // Get the Subject's ID number.
  strcat(FileName, " MyExp.data-");     // Append it to the front of the data
                                          // FileName.
  printf("Session #:\t");               // Ask for the session number.
  scanf("%s", SessionString);          // Get the session number
  strcat(FileName, SessionString);     // Append it to the end of the FileName.
}
...
```

Notice the global variables at the top of the file. The variables marked with `extern` in front of their type are already defined in other files. We'll be introducing a new type of variable in this section: the string. A string is simply an array of characters. For example, the variable `FileName` is a string (or array) 30 characters in length. To be truthful, the string `FileName` is 29 characters long – the last character in a C string is always a null (“\0”) character and signifies that that is the end of the string. More information about strings is available in *Advanced Topic: Words and Strings*.

The function `GetSubjectInfo()` is used to get the subject's ID number so that we can use it to name the data file. We also ask for the current session number. The subject's ID number will form the prefix of the file name and the session number will form the suffix. For example, if we were running subject #10 and it was their second session, `GetSubjectInfo()` would generate the `FileName` “10 MyExp.data-2”.

To do this, we first need to ask for the subject's ID number. To simplify things, we use CodeWarrior's built in console `io`. The console resembles the type of displays used in text

terminals or in MS-DOS. Although not pretty, it provides us with an easy way to get information about the subject before the experiment begins.

```
printf("ID Number:\t");           // Ask for the subject's ID #.
```

The `printf()` function is a standard C routine for printing text to the screen. That is, it will work in UNIX, DOS, or MacOS (when using the console window). The “\t” tells `printf()` to print a tab.

```
scanf("%s", FileName);           // Get the Subject's ID number.
```

Now that we have prompted the experimenter for the subject’s ID #, we need a way to read the ID number in from the keyboard. That is what `scanf()` does. It scans the keyboard for input. When we use `scanf()`, we need to tell `scanf()` which variable to assign the keystrokes to and what type of variable that variable is. In this case, we will be reading the subject’s ID # into the string `FileName`. For `scanf()` to do this correctly, we also need to tell it the `FileName` is a string. We do this by using the formatted input code “%s”, which tells `scanf()` that we will be reading in a string. For more information on formatted input and output, see the section *Disk and Console io: Formatted Input and Output*.

```
strcat(FileName, " MyExp.data-"); // Append it to the front of the data
//   FileName.
```

Now that we have the subject’s ID# number in the string `FileName`, our next step in this example is add the name of the experiment to the file name. We do this using the function `strcat(to, from)`. The function `strcat()` is used to concatenate one string onto the end of another string. After completing this step, `FileName` should consist of the characters “10 MyExp.data-\0”. Remember that all C strings must end with a zero. Since there is no printable character corresponding to the ASCII value zero (not to be confused with the symbol “0”, which has an ASCII value of 48), the symbol “\0” is often used in debuggers to symbolize the character with the ASCII value zero.

Our final series of steps consists of asking for the session number, getting the response, and concatenating it onto the end of the string `FileName`. Just as we had done when we had asked the experimenter for the subject’s ID number, we use `printf()` to ask for the session number. We then use `scanf()` to scan the session number into the string `SessionString`. In this example, our experiment will only have two sessions. Since the number of sessions will always be in the single digits, the string `SessionString` is only two characters long (remember that there must be a trailing null, or zero, value). After scanning in the session number, we use `strcat()` to append the string `SessionString` to the end of the string `FileName`. After all of these steps have been completed, `FileName` should consist of the characters “10 MyExp.data-2\0”.

SaveData()

```

#include "PsychRoutines.h"
#include "myExp.h"

extern double RT[NumTrials];           // Response Times
extern short OrderArray[NumTrials];
extern short Correct[NumTrials];      // Correct Response?
extern short TargColor[NumTrials];    // Target Color
extern short Direction[NumTrials];    // Cue direction
char FileName[30];                    // a string

void SaveData(void)
{
    FILE *fp;                          // a file pointer
    short i;                            // our friend the local 'i'
    short Order;

    if ((fp=fopen (FileName, "w"))==NULL) // Try to open the file for writing.
    {
        printf("Cannot open file\n");    // If can't, inform user & exit.
        exit (0);
    }

    fprintf (fp, "number\torder\tTargetColor\tDirection\tCorrect?\tRT\n");
    for(i=0; i< NumTrials; i++)
    {
        Order = OrderArray[i];          // for simplicity's sake
        fprintf (fp, "%hd\t%hd\t%hd\t%hd\t%hd\t%lf\n",
            i, Order, TargColor[Order], Direction[Order],
            Correct[Order], RT[Order]);
    }
    fclose(fp);
}
...

```

The function SaveData performs four steps: it 1) creates a file called FileName and opens it for writing, 2) saves the data column names to disk, 3) writes the data to disk, and 4) closes the file.

```
FILE *fp;                          // a file pointer
```

The variable `fp` is a pointer to the file we are creating. That is `fp`, is the variable that represents the file. The asterisk (*) before the variable name tells the compiler that the variable will be a pointer, while `FILE` is the type of variable it will be pointing to. Don't worry about it too much – if you use the above example of `SaveData()` to base your own version of `SaveData()` on, you will probably never have to change that variable definition.

```

if ((fp=fopen (FileName, "w"))==NULL) // Try to open the file for writing.
{
    printf("Cannot open file\n");    // If can't, inform user & exit.
    exit (0);
}

```

Our next step is to try to open the file for writing. If the file can not be created for some reason (perhaps we do not have enough disk space left), we return an error message to the user and quit the program. Let's break these lines of code down into smaller steps and examine what each one of them does.

```
fp=fopen (FileName, "w")
```

The function `fopen` attempts to open a new file with the name `FileName` for writing (`w`). It then returns the results into the `FILE` pointer `fp`. If `fopen` succeeds, `fp` will now point to a new file, otherwise, `fp` will be equal to `NULL`.

```
if ((...)==NULL)
```

The `if()` function checks to see if the pointer `fp` is equal to (`==`) `NULL`. If `fp` is equal to `NULL`, then opening a new file for writing has failed, and we execute the lines of code between the brackets. If it succeeded, then `if()` skips those lines of code and allows the rest of the `SaveData()` function to execute. If this confuses you, don't worry: you will probably never have to change this part of the `SaveData()` function.

```
fprintf (fp, "number\torder\tTargetColor\tDirection\tCorrect?\tRT\n");
```

The `fprintf()` function is the file version of the `printf` function. When you use the `fprintf()` function, you must specify which `FILE` you will be printing to (`fp`), then any text or variable placeholders, and finally any variables that will be inserted into the placeholders. In the above example, there are no variables, and we are simply printing a tab-delimited row of column headings. The `fprintf` function should print the following text to disk:

```
number    order    TargetColor    Direction    Correct?    RT
```

You probably noticed the characters `\t` and `\n` in the `fprintf` statement. These are used to print characters that are often difficult to type into the keyboard or are invisible. For example, `\t` will print tabs, and `\n` causes `fprintf` to print a new line. You can use whatever characters you want to delimit your data columns, but I prefer tabs. Other options include using spaces and commas. Most programs can read tab, space, and comma delimited text files equally well (e.g. Excel, SuperAnova), but some programs are finicky (e.g. SAS and SPSS).

```
for(i=0; i< NumTrials; i++)
{
    Order = OrderArray[i];          // for simplicity's sake
    fprintf (fp, "%hd\t%hd\t%hd\t%hd\t%hd\t%lf\n",
            i, Order, TargColor[Order], Direction[Order],
            Correct[Order], RT[Order]);
}
```

After we have printed the column headings, we then loop through each trial and print the data from each trial into a new row in the text file. As we did before when using `fprintf()`, we first specify which file we are printing to (`fp`), then text or variable placeholders, and lastly the variables to be inserted into the placeholders.

```
fprintf (FILE filePointer, "text and/or variable placeholders" , variables);
```

The strange series of characters between the quotes consists of tabs (\t), line returns (\n), and the variable placeholders (%hd and %lf). When fprintf writes to disk, each variable placeholder will be replaced by the appropriate variables listed after the quote marks. For example, if the following variables had these values:

i	0
Order	57
TargColor	1
Direction	2
Correct	0
RT	475.6

then the first two lines of our data file would look like this:

number	order	TargetColor	Direction	Correct?	RT
0	57	1	2	0	475.6

For more information on variable placeholders, see *Advanced Topics: Disk and Console io: Formatted Input and Output*.

Basic Topics

Practice Trials

```

void DoPractice(void) // Same as main loop, but 1) does not save data
{
  short i; // and 2) randomly samples trials.
  char dummyResponse; // A variable to take the response

  for( i = 0; i < 15; i++) // present 15 practice trials
  {
    Order = OrderArray[RandomRange(NumTrials)];
    // MakeCues(Order); // offscreen drawing
    // MakeStims(Order); // offscreen drawing
    // ShowCues(); // display the cue screen
    WaitSyncs(10); // wait 10 refreshes
    // ShowStims(); // display the stimuli screen(s)
    dummyResponse = KeyBoardPol(); // get the response
    WaitSyncs(75); // wait 75 refreshes
  }
}

```

Presenting subjects with practice trials is usually a fairly strait-forward task. I make a function called `DoPractice()`. The function `DoPractice()` is identical to the main loop except: a) it presents only practice trials, and b) does not record any responses.

```

for( i = 0; i < 15; i++)
{
  Order = OrderArray[RandomRange(NumTrials)];
  ...
}

```

To make the practice trials, each time the `DoPractice()` loops, the program randomly grabs one of the real trials and uses it as a practice trial. To do this, use `RandomRange(NumTrials)` to randomly generate a number between 0 and `NumTrials-1`. Of course, if you are using word stimuli, this method might not be appropriate. In that case, you might want to have a separate list of words to us only for the practice trials. This is covered in the section *Advanced Topics: Using External Stimuli Files*.

Since we are presenting practice trials, we usually do not want to record participants' responses. However, the routines that get responses from the keyboard or microphone will be returning values. To get around this, simply make a variable, such as `dummyResponse`, to record the response into.

Randomization

So far, we've already come across two randomization functions: `RandomArray()` and `RandomRange()`. `RandomArray()` was used by `RandTrials()` to generate a random list of number sampled without replacement (or more accurately, it creates an ordered list of numbers and then randomly shuffles the list). `RandomRange` was used by `DoPractice()` to generate a random number between 0 and x . Let's take a closer look at these functions and some other useful randomization functions. For a more comprehensive list, see *Definitions: Psychological Toolbox Functions: Randomization and Array Functions*.

```
short RandomRange(short top)
```

`RandomRange()` returns a random value from 0 to *top* - 1. Thus, if you pass `RandomRange()` the value 100, it will return a random number from 0 to 99. Although this might seem a little odd, it works in perfect harmony with the C/C++ method of array indexing. Remember that C/C++ starts indexing arrays at 0. Thus, if you have an array that is 100 elements long, the first element is #0, and the last element is #99.

```
short RandomBetween(short bottom, short top)
```

`RandomBetween()` works much like `RandomRange()`, except that it returns a value from *bottom* to *top* - 1.

```
double FloatRandom(double top)
```

Much like `RandomRange()`, `FloatRandom()` returns a random value that ranges from 0 to < *top*. However, `FloatRandom()` returns an array of double precision floating point numbers.

```
void RandomArray(short * arrayName, short length)
```

`RandomArray()` fills an array with a random (without replacement) series of numbers. Simply pass the name of the array, sans brackets [], and the *length* of the array to the function.

```
void OrderedArray(short * arrayName, short length)
```

`OrderedArray()` acts exactly like `RandomArray`, except that no randomization occurs. That is, `OrderedArray()` fills an array with shorts that starts at 0 and ends at *length* - 1.

```
void ScrambleArray(short * arrayName, short top)
```

`ScrambleArray()` is the complement to `OrderedArray()`. `ScrambleArray()` takes an array of shorts and randomly shuffles their order around. In fact, `RandomArray()` consists of a call to `OrderedArray()` to generate an ordered array, and then a call to `ScrambleArray()`.

Graphics

DrawToScreen

Whenever you want the screen to be the active drawing device, call `DrawToScreen()`.

ColorDepth

Color depth refers to the number of colors that a Macintosh can display. The color depth can be changed by using the *Monitors* control panel, the Control Strip, shareware utilities such as *Depth Charge*, or by using the function `SetColorDepth()`.

bits	number of colors		notes
1	2	(black & white)	
2	4	(B&W, highlight, & gray)	
4	16		
8	256		“indexed” color
16	32,768	“thousands”	5 bits per red, green, & blue
32	16,777,216	“millions”	only 24 bits used, i.e. 8 bits per r, g, & b

Color depth affects not only the number of colors that can be displayed, but also the graphics speed and the size of your program. A crucial method of ensuring that your graphics run at optimal speeds is to make sure that the color depth of the screen is the same color depth as your Tcards.

```
void SetColorDepth(GDHandle theScreenGD, short Cdepth)
```

or usually:

```
SetColorDepth(onScreenGD, ColorDepth);
```

To set the color depth of the screen, you need to pass it two variables: the GDHandle of the screen and the color depth of the screen in bits. Most of the time, you will pass it the predefined global variable `onScreenGD`, and the constant `ColorDepth`. Remember that we had previously #defined `ColorDepth` in the header file `myExp.h`.

Setting the color depth of the Tcards is discussed in the Tcard section below.

Colors

Apple’s Color QuickDraw makes working with colors a breeze. To make things even easier, the *Psychological Toolbox* contains several predefined colors. This include black, white, gray, red, blue, green, and yellow. To learn how to define your own colors, see the section *Advanced Topics: Graphics: Colors*.

Foreground Colors

Colors can be used for two different purposes: as a foreground or background color. The foreground color defines the current *pen* color. Anything that is drawn, such as lines or letters, is controlled by the current foreground color.

Psychological Toolbox includes several functions to set the current foreground color to a predefined color. These include `MakeBlack()`, `MakeWhite()`, `MakeGrey()`, and `MakeRed()`. A more comprehensive list is included in the section *Definitions: Psychological Toolbox Functions: Color Functions*. To learn how to make your own custom foreground colors, see the section *Advanced Topics: Graphics: Colors*.

Pen? What is a pen? Never forget: your best friend is *Think Reference*. *Think Reference* can be accessed (assuming you have it installed) by typing + '. Typing this key combination will launch *Think Reference* (if it isn't already running) and attempt to look up whatever text has been highlighted (assuming you have highlighted some text). There are two exceptions to this rule: 1) *Toolbox Assistant* might be checked in *Preferences: Extras*, or 2) you might have highlighted a variable or function defined somewhere in your program.

Background Colors

Background colors act as *fill* colors. That is, if you are drawing a rectangle, circle, or oval, the color that fills those objects is the background color. Like the foreground colors, *Psychological Toolbox* includes several predefined functions to set the background colors. These include `MakeWhiteBack()`, `MakeBlackBack()`, and `MakeGreyBack()`. The section *Advanced Topics: Graphics: Color* contains information about using your own custom colors as the background color.

Using and Filling Rectangles

Rectangles have numerous uses. They can be used to fill the background, erase an image, or to draw a visible rectangle. Luckily for us, rectangles are a common type of data structure used by the MacOS.

Rect

A `Rect` is a structure that defines the coordinates of a rectangle. A `rect` is defined by the four values *top*, *bottom*, *left*, and *right*. For more information, see the section *Definitions: Common Structure: Rect*.

SetRect()

```
void SetRect(Rect * theRect, short left, short top, short right, short bottom);
```

To define a `Rect`, call the function `SetRect`. `SetRect` takes five values: a pointer to your `rect`, the top, the bottom, the left side, and the right side. So, if we wanted to define a `Rect` called `theRect` to be the size of a 640x480 screen:

```
SetRect(&theRect, 0, 0, 639, 479);
```

BackToBlack

Psychological Toolbox contains three functions that can be used to draw a rectangle filled with a background color. These functions are `BackToBlack()`, `BackToGray()`, and `BackToWhite()`. These three functions are most commonly used to erase the screen or a Tcard.

```
Void BackToBlack(Rect theRect);
```

When you call these functions, simply pass them the name of your rectangle.

PaintRect()

```
void PaintRect(Rect * theRect)
```

`PaintRect` fills a rectangle with the current background color.

```
PaintRect( &theRect);
```

Text

The MacOS provides a powerful ways to display text. The MacOS treats displaying text like drawing any other graphic object – text can be colored, stretched, filtered, or what ever you can imagine.

TextFont()

```
void TextFont(short fontNumber)
```

To set the current font, call `TextFont()` and pass it the number of your font. Where can you find the font number? Use *Think Reference!* For example, try looking up the function `TextFont`.

TextSize()

```
void TextSize(short size);
```

To set the size of your font, call `TextSize`.

drawstring()

```
void drawstring(const char * string);
```

`drawstring()` is used to draw C-style strings. This is not to be confused with the older function `DrawString()`, which was used to draw Pascal style strings.

Pascal style strings have a long history with the Macintosh. Before there was C++, Apple collaborated with Nickolas Wirth (the inventor of Pascal) to come up with Object Pascal. Only recently, with the release of the Universal Headers, has Apple made using C style strings easier.

How do the two types differ?

- C strings end with a null value (zero)
 - Pascal strings are length prefixed. That is, the first value in a string defines the length of the string.
-

To draw the string “Hello World\0” to the screen, you would type:

```
drawstring("Hello World");
```

Or, to draw a string variable to the screen, you could do the following:

```
char  myString[12];    // a string 12 characters long

strcat( myString, "Hello World");
drawstring(myString);
```

`drawstring()` should not be confused with the ANSI C functions such as `printf()`. `printf()` prints only to the console window and can print nearly any variable type as long as the proper format specifier is present. `drawstring()` draws only strings and does not draw to the console.

Tcards

In many ways, Tcards are the heart of the *Psychological Toolbox*. They make using graphics extremely easy. Tcards, as the name implies, are virtual tachistoscope cards. Like tachistoscope cards, they can be written on, erased, and shown.

Tcards are objects. Objects are essentially structures that have their own predefined functions. Since Tcards are objects, they must be declared like any other variable:

```
Tcard myCard;
```

You can also make arrays of Tcards. This is very useful when making experiments using RSVP (Rapid Serial Visual Presentation).

```
Tcard RSVPCards[20];
```

Examples are included in the file `<psych_graphics>.cpp`.

Defining

Before using a Tcard, you must define its size (a `Rect`) and its color depth. An example of this is provided in the file `<psych_graphics>.cpp`.

```

Tcard stimCard;
Rect  StimCardRect;

void  InitTcards(void)
{ SetRect(&StimCardRect, 280, 200, 359, 279);    // define our Rect
  stimCard.RectDef(StimCardRect);              // set the Tcard Rect
  stimCard.MakeWorld(ColorDepth);              // set the Tcard color depth
  stimCard.BeginDraw();                        // prepare for drawing
      BackToWhite(StimCardRect);              // erase it to white
  stimCard.EndDraw();
}

```

The two functions you need to call to define a Tcard are `Tcard.RectDef()` and `Tcard.MakeWorld()`. The function `Tcard.RectDef()` must be called before you call `Tcard.MakeWorld()`. If you attempt to call `Tcard.MakeWorld()` first, your program will automatically exit.

Tcard.RectDef()

```
void  Tcard.RectDef(Rect  myRect);
```

First, you must define the physical size of the Tcard. This is done by calling the function `RectDef()` and passing it the name of your rectangle. Remember that when using objects, you use the object's name, followed by a period, and then the function name:

```
stimCard.RectDef(StimCardRect);
```

Tcard.MakeWorld()

```
void  Tcard.MakeWorld(short  colorDepth);
```

The second thing that must be done is to set the color depth of the Tcard. Remember that graphics are drawn most quickly when the color depth of the Tcard matches the color depth of the screen. This is easily accomplished by passing the function `MakeWorld()` the global variable `ColorDepth`.

```
stimCard.MakeWorld(ColorDepth);
```

Drawing in

When ever you draw into a Tcard, your drawing routines must be bracketed by the object functions `BeginDraw()` and `EndDraw()`. Thus, if you had a Tcard called *StimCard* in which you were going to draw a white rectangle the size of *StimCardRect*, your code would look like this:

```

stimCard.BeginDraw();          // make the card the current graphic device
    BackToWhite(StimCardRect); // draw a white rectangle
stimCard.EndDraw();

```

Although you do not need to indent your drawing functions, I do this to make the code easier to read.

Displaying

Two steps are required to display a Tcard. First, you must set the active graphic device. That is, you must tell the Macintosh where all drawing functions will occur. Secondly, you must copy the image in the Tcard to the new graphic device.

To make the screen the new graphic device, call:

```
DrawToScreen( );
```

To copy the image contained in the Tcard to the screen, use the object function Show(). Thus, if you wanted to copy the contents of a Tcard called *StimCard* to the screen, your code would read:

```
DrawToScreen( );
StimCard.Show( );
```

If you wish to display another Tcard, DrawToScreen() does not need to be called again – the display is still the current graphic device. If you draw into a Tcard using BeginDraw and EndDraw, then you must call DrawToScreen() again to make the screen the active graphic device.

Notes on Tcards size, color depth, and copying speed

The size and the color depth of your Tcard determine how much memory is used and the graphics speed.

To determine how much memory a Tcard uses, multiply the bit depth by the width and height of the card. Dividing this by 8 will give you the number of bytes the card will use in memory (actually, it will use a little more).

$$\text{Number of bytes used} = \frac{\text{TcardColorDepth} \times \text{width} \times \text{height}}{8}$$

So, a Tcard that has an 8-bit color depth (256 possible colors) and is 480 pixels high by 640 pixels wide would use 307200 bytes of memory ($307200/1024 = 300\text{k}$):

$$\frac{8 \times 640 \times 480}{8} = 307,200 \text{ bytes}$$

If you allocate enough memory to your program, by using the Finder's Get Info command (+ I) or by using CodeWarrior's *Preferences: 68k (or PPC) Project*, you shouldn't run out of memory.

The speed at which you can copy images to the screen is affected by the size of the image, the color depth of the image, and the machine you are using. To determine the video speed of your Macintosh, use the program *TimeVideo* (Pelli, 1996). *TimeVideo* will check the refresh rate of

you machine at all of it's resolutions (that is, screen size = 640x480, or 832x624, etc.) and all of it's color depths. The program returns the results to a file called *TimeVideo results*.

PowerMac 6100/66 "Built-In DRAM Video" (.Display_Video_Apple_Sonora version 7)						
8-bit dacs. 640x480 pixels. (colordepth)						
pixel size (colordepth)	1	2	4	8	16	bits
pages	1	1	1	1	1	
mode	0x80	0x81	0x82	0x83	0x84	
frame rate (refreshes in Hz)	66.5	66.5	66.5	66.5	66.5	Hz
interrupts per frame	1.0	1.0	1.0	1.0	1.0	
CopyBits movie size	6.49	3.74	1.71	0.78	0.28	screen
CopyBitsQuickly movie size	5.80	3.30	1.74	0.79	0.28	screen
CopyBits data rate	15.82	18.24	16.67	15.29	10.84	MB/s
CopyBitsQuickly data rate	14.12	16.10	16.97	15.41	10.87	MB/s
cscSetEntries duration	1.00	1.00	1.00	1.00	1.00	frames
cscSetEntries suppresses ints.for	-0.0	-0.0	0.0	-0.0	-0.0	frames
GDSsetEntriesHighPriority duration	1.00	1.00	1.00	1.00	1.00	frames
color: cscSetEntries test	ok	ok	ok	ok	ok	
gray: cscSetEntries test		ok	ok	ok	ok	ok
(ROut± 0.3%) (0.30 0.59 0.11) (RIn)						
(GOut± 0.3%)=(0.30 0.59 0.11)x(GIn)						
(BOut± 0.3%) (0.30 0.59 0.11) (BIn)						

An example of what might be found in *TimeVideo results* is shown above. The three most important pieces of information you will see in this file are:

- the screen size in pixels
- the refresh rate for that screen size
- the copying speeds available for each color depth at that screen size (CopyBits movie size)

The example above is from a PowerMac 6100/66. The size of the screen in the above example is 640x480 pixels (more are possible) and the refresh rate is 66.5 Hz (15 msec per refresh).

If we take a look at the row labeled CopyBits movie size, we will see several values displayed for each possible pixel depth:

pixel size (ColorDepth)	1	2	4	8	16	bits
...						
CopyBits movie size	6.49	3.74	1.71	0.78	0.28	screen

What this says, is if we are in 1 bit color mode (black and white) we can draw an image 6.49 x the size of the screen (area = 640 x 480 = 307,200 pixels) within one refresh (15 msec). In other words, we can safely copy an image that is the size of the screen within one screen refresh.

However, that is not the case when we are in 8-bit color mode. When running in 8-bit color mode, the maximum image size we can safely copy to the screen within one refresh is 0.78 screens, or 239616 pixels. This is usually not a problem unless your image is supposed to be the size of the entire screen. For example, if you are presenting words, you can probably get by with a 200x100 Tcard (20,000 pixels).

Timers

As of now, because the timer is implemented as a function, you can only time one thing at a time – in the future, they will be implemented as objects. To start the timer, call:

```
void StartTimer(void);
```

To stop the timer, use the function `StopTimer()`:

```
double StopTimer(void)
```

```
double myTime;
```

```
StartTimer();
```

```
myTime = StopTimer();
```

Getting Responses

There are two different functions you can use to get keyboard responses. `KeyBoardPol()` should be used when you are using a keypress to stop a timer (response time experiments). `GetaKey` should be used in all other circumstances.

The three methods (including the voice key method) for recording responses share one thing in common: they all hog the computer when waiting for a response. That is, the computer is incapable of performing other functions (such as drawing) while these functions are waiting for a response. There are ways to get around this: a) use the `SyncAndPoll()` as discussed in the section *Advanced Topics: Keyboard Polling Accuracy Issues: SyncAndPoll*, or b) use an external button box and timer (such as the Carnegie-Mellon [CMU] button box).

KeyBoardPol()

`KeyBoardPol()` bypasses a lot of system functions to ensure accurate timing. Unfortunately, this also has a few side effects. `KeyBoardPol()` can not be called again until 150-200 ms seconds have elapsed, or else you will get spurious results. You can enforce this delay by inserting a `WaitSyncs(17)` between your `KeyBoardPol()`s. This make `KeyBoardPol()` unsuitable for Psychological Refractory Period (PRP) type experiments that time more than one keypress.

```
Byte KeyBoardPol(void)
```

`KeyBoardPol()` returns the keyboard character code that was pressed. These keycodes can change depending on what model keyboard you are using. To find out the keyboard code for your keyboard, try *Keyboard Character Codes* in *Think Reference*.

Sorry, but as of now, *Psychological Toolbox* does not include a function to translate between the keyboard character codes and the ASCII value.

```

char    theResponse;    // yes, it should be a byte, but use a char
double  myTime;
Boolean Correct;
...
theResponse = KeyBoardPol();    // continuously poll the keyboard for a
response
myTime = StopTimer();           // stop the timer and record the time
switch (theResponse):
{
    case 0x06:                 // 0x06 (hex) is the keycode for 'z'
        Correct = FALSE;     //if keyDown== 'z', then absent
        break;
    case 0x2c:                 //if keyDown== '/', then present
        Correct = TRUE;
        break;
    default:
        Correct = -1;        //else, the subject pressed the wrong key
}

```

GetAKey()

The other keyboard function is `GetAKey()`. Unlike `KeyBoardPol()`, `GetAKey()` can safely read in several keypresses in a row. `GetAKey()` also returns the character value, not the keyboard value of the key pressed. That is, if the subject hits the 'c' key, `GetAKey()` returns the character 'c'.

So, if you were writing an Attentional Blink experiment that required the subject to type in the identify (in order) of the two uniquely colored letters, your code might look like this:

```

char    Answer1[NumTrials];    // response for first target
char    Answer2[NumTrials];    // response for second target

void    main(void)
{
    ...
    for(i = 0; i<NumTrials; i++)    // main loop loops through trials
    {
        ...
        ShowDisplay();             // display stimuli
        Answer1[Order] = GetAKey(); // get the first response
        Answer2[Order] = GetAKey(); // get the second response
        ...
    }
}

```

VoiceKey

The keyboard is by not the only method for timing responses. You can also use the Macintosh's built in sound input (available on most Macs and clones since 1988?). Like the previous two functions, the `VoiceKey` functions take over the computer when waiting for a response.

Early in your program, before the main loop, you should call the `InitMicrophone()` initialization function.

To use the voicekey, simple call the function:

```

void    PollMicrophone(short threshold);

```

threshold can vary between 0 and 255 and is the intensity threshold at which the voicekey is triggered. You can use the program *Mic Meter Level app* to determine what a good threshold value for you. I have found that 60 is a good level.

```
double    myTime;

void main(void)
{
    ...
    for(i = 0; i<NumTrials; i++)    // main loop loops through trials
    {
        ...
        ShowDisplay();              // display stimuli
        StartTimer();
        PollMicrophone(60);         // poll the microphone until the level is exceeded
        myTime = StopTimer();
    }
}
```

Advanced Topics

Inserting Breaks

```

for(i=0; i< NumTrials; i++)    // main loop
{
  if(i % 50 == 0)              // blocks are 50 trials, and then a break
  {
    if(i != 0)                 // if not the first trial, then...
      TakeABreak();           // give the poor souls a break!
  }
  ...
}

```

To insert breaks between blocks of trials, use the ‘%’ (“modula”) C operator. The modula operator returns the remainder of a division problem. Thus, if the remainder is zero, you know that the current trial count is equal to your divisor.

```

numerator % denominator = remainder
OR
trial count(usually i) % block size = remainder

```

Designs with unequal n’s

```

#include "PsychRoutines.h"
#include "myExp.h"

short  CueValidity[NumTrials]; // 1 = valid, 0 = invalid
short  Direction[NumTrials];   // 0 = left, 1 = right.
short  OrderArray[NumTrials]

void MakeTrials(void)
{
    short  i, j, k;             // local variables
    const short  Cells = 10;
    const short  Repetitions = // # of Repetitions is dependent
        NumTrials/Cells;      // on Cells and NumTrials.
    short  NumBuf;              // Used to calculate which
                                // trial we are on.

    for(i = 0; i< Repetitions; i++) // repetitions
    {
        for( j = 0; j< 2; j++)      // left or right
        {
            NumBuf = (i*Cells)+ (j*5);
                // = (i*10) + (j*5)
            CueValidity[NumBuf+0] = 0; // invalid validity ratio
            Direction[NumBuf +0] = j;  // invalid is 80:20 = 4:1
            //---
            CueValidity[NumBuf+1] = 1; // valid
            Direction[NumBuf +1] = j;  // valid
            //---
            CueValidity[NumBuf+2] = 1; // valid
            Direction[NumBuf +2] = j;  // valid
            //---
            CueValidity[NumBuf+3] = 1; // valid
        }
    }
}

```

```

        Direction[NumBuf +3] = j;    // valid
        //---
        CueValidity[NumBuf+4] = 1;    // valid
        Direction[NumBuf +4] = j;    // valid
    }
}
...

```

Before we jump into `MakeTrials()`, let me explain the design of the experiment. In this experiment, subjects are given a central arrow cue to the probable location of a target. On half of the trials, the target occurs to the left of fixation, and on the other half it occurs to the right of fixation. The cue is 80% valid. That is, the cue is incorrect 20% of the time. This sounds like a simple 2 x 2 design (direction x validity), right?

Well it is, at least until we sit down to program the experiment. The direction variable is easy to code: each direction occurs equally often. However, each type of validity does not. Since the cue is 80% valid, valid cue trials will occur four times as often as invalid cues.

In our earlier `MakeTrials()` example, we used `Cells` to define the number of cells in our design. How many cells do we have in this design? The correct answer is 4. However, from a programming perspective, our design has 10 cells (direction x validity = 2 x [4 valid + 1 invalid] = 2 x 5 = 10).

We have four times as many valid trials as invalid trials. An easy way to define the trials when you have an asymmetric number of repetitions is to list out all of the possible trials. That is, if we have 4 valid trials for every 1 invalid trial, we should code 4 valid and 1 invalid trial definitions. For example:

```

CueValidity[NumBuf+0] = 0;    // invalid validity ratio
Direction[NumBuf +0] = j;    // invalid is 80:20 = 4:1

```

The `MakeTrials()` example at the beginning of this section has five groups of these definitions. The first definition is for the single invalid trial, and the next four definitions are for the four valid trials. Each definition is referenced by `NumBuf + x`, where $x = 0, 1, 2, 3, \text{ or } 4$.

The variable `NumBuf` is used to help us calculate which trial we are currently defining. As an example, let `NumTrials = 100`. Since `Cells = 10`, `Repetitions` must equal 10.

i (0<=i<Repetitions)	j (0<=j<2)	x (0<=x<5)	NumBuf (0<=NumBuf<NumTrials)
0	0	0	0
0	0	1	1
0	1	0	5
0	1	1	6
1	1	0	10
10	1	4	99

Blocked Experimental Designs

Let's say we have an experimental design in which subjects are to judge the duration of a stimulus. Subjects are presented with two stimuli: first, the reference stimulus (the duration never changes during the experiment), and then the judged stimulus. Since our monitor has a refresh rate of 15 ms, let's make our reference stimulus always 495 milliseconds (33 refreshes) in length. The judged stimuli range from 420 ms in length (five refreshes below) to 570 (five above) ms in length in steps of 15 ms. Thus, subjects will be judging the length of 11 different durations.

We also decide to vary the modality of the stimuli to see if modality has an effect. So, on half of the trials, the reference and judged stimuli are tones, and on the other half they are flashes of light. This gives us a 2 x 11 (modality x duration) design.

We test our experiment and quickly determine that randomly switching the modality of the stimuli between trials is rather aggravating – we are never sure which modality to expect. Since our study isn't concerned with how well people can switch modalities, we decide to get rid of this extraneous noise by presenting the trials block by modality. That is, the first half of the experiment will consist of one modality, and the second half consists of another.

Since we have a 2 x 11 design, that gives us 22 cells. To be kind to our subjects, we decide to have only 10 repetitions per cell. This gives us 220 trials in total (NumTrials).

```

void    MakeTrials(void)
{
    short    i, j, k;                // local variables
    const    short    Cells=22;        // 2 x 11
    const    short    Repetitions =    // # of Repetitions is dependent
                NumTrials/Cells;      // on Cells and NumTrials
    short    which=0;                // Used to calculate which
                                    // trial we are on.

    for(i = 0; i < Repetitions; i++)  // repetitions
    {
        for( j = 0; j < 2; j++)        // auditory or visual
        {
            for( k = 0; k < 11; k++)    // duration
            {
                which++;
                //---
                Modality[which] = j;    // 0 = audio, 1 = visual
                Duration[which] = 420+(k*15); // 420 - 570 ms, steps of
15
            }
        }
    }
}

```

So, as it now stands, the first 110 trials are auditory trials, and the second 110 trials are visual trials. We need to counterbalance the presentation order of the blocks across subjects. We'll use the function GetSubjectInfo() to record which group (A then V, or V then A) each subject is in:

```

short    GroupNum;                // the global variable GroupNum

void    GetSubjectInfo(void)
{
    char    Group[2];                // which group the subject is in

    printf("ID Number:\t");          // Ask for the subject's ID #.
    scanf("%s", FileName);           // Get the Subject's ID number.
    strcat(FileName, " MyExp.data-"); // Append it to the front of the data
                                    // FileName.
    printf("Group # (0=AV, 1=VA):\t"); // Ask for the Group number.
    GroupNum = -1;
    do
    {
        scanf("%hd", GroupNum);      // Get the Group number
    }
    while( (GroupNum < 0) || (GroupNum >1); // make sure it is within range
    Group[0] = GroupNum + 48;         // '0' ASCII value = 48
    Group[1] = 0;                     // Null terminate the C string
    strcat(FileName, Group);          // Append it to the end of the FileName.
}

```

```
}

```

Since the group that the subject belongs to determines the presentation order of the trials, the function `RandTrials()` is where this takes place:

```
extern short  GroupNum;                // the global variable GroupNum
...
void  RandTrials(void)
{
    short  i;
    short  TrialBuf;
    const  short  HalfTrials = NumTrials/2;    // for easier readability
    short  start, end;

    OrderedArray( OrderArray, NumTrials);    // fill OrderArray with 0... NumTrials-1
    for(i=0; i<2; i++)                      // two blocks
    {
        start = i * HalfTrials;              // calculate the beginning of the block
        end = start + HalfTrials;            // calculate the end of the block
        ScrambleBetween(OrderArray, start, end);    // Scramble each 1/2 the
                                                    // trials separately.
    }
    if(GroupNum == 1)                        // if video then audio
    {
        for(i=0; i< HalfTrials; i++)        // ...exchange trials.
        {
            TrialBuf = OrderArray[i];
            OrderArray[i] = OrderArray[i+HalfTrials];
            OrderArray[i+HalfTrials] = TrialBuf;
        }
    }
}

```

Notice the constant `HalfTrials`. Using this constant not only ensures greater readability, but also ensures that if you change the number of trials, `HalfTrials` will also change.

The first thing we need to do is to fill the array `OrderArray` with an ordered set of numbers (from 0 to `NumTrials-1`). This is accomplished by calling the function `OrderedArray()`.

The second step requires us to randomize each half of the trials independently. That is, we shuffle the order of the first half of the trials, then we shuffle the second half. This is accomplished by using the function `ScrambleBetween()`.

If the current subject is to receive the auditory trials first, then we are done. However, if the subject is to receive the visual trials first, then we need to change the sequence in which the trials are presented. As it stands now, the first half of the trials are randomly shuffled auditory trials, and the second half are visual trials. To change the order in which the blocks are presented, all we need to do is to exchange trials between the blocks.

To accomplish this, we use the `for` loop:

```
for(i=0; i< HalfTrials; i++)    // ...exchange trials.
{
    TrialBuf = OrderArray[i];
    OrderArray[i] = OrderArray[i+HalfTrials];
    OrderArray[i+HalfTrials] = TrialBuf;
}

```

This `for` loop starts with trial 0 and places it in a buffer. Trial 0 is then equal to `HalfTrials + 0`. We then take the former value of trial 0 and place it in `HalfTrials + 0`.

Disk and Console io

Formatted Input and Output

As mentioned earlier, not only do we need to specify which variables we are printing or reading in when using C io functions, but we also must specify the variables' types. This is accomplished by using formatting strings or placeholders.

Type	Printing format	Reading format
int	%d	%d
long int	%ld	%ld
short int	%hd	%hd
unsigned int	%u	%u
float	%f	%f
double	%f	%lf
short double	%f	%hf
exponential form of float	%e	%e
exponential form of double	%e	%le
exponential form of short double	%e	%he
long double	%Lf	%Lf
char	%c	%c
string	%s	%s

Most of the formatting strings are identical whether you are printing or reading in data. However, some of the *float* formatters change.

To specify the printing width, insert a number between the '%' and the letter specifier.

To print a *short* integer 3 numerals long:

```
%3hd
```

To print a *float* with two significant digits before the decimal and three significant digits after the decimal:

```
%2.3f
```

In addition, there are several characters that are difficult to type or are invisible. The most relevant ones are tabs and new lines:

```
tab          \t
new line     \n
backslash    \\
```

A quick overview of commands

printf()

```
int printf("formatting specifiers" , variables);
```

Prints a formatted string to the console window and returns the number of characters successfully printed. *Variables* is optional.

```
printf("Hello world!\n");
```

```
    Hello world!
```

```
printf("pi = %1.5lf\n", 3.14159);
```

```
    pi = 3.14159
```

```
float pi = 3.1415927;
```

```
printf("pi = %1.5lf\n", pi);           // limits pi to 6 digits (1+5)
```

```
    pi = 3.14159
```

fprintf()

```
int fprintf(FILE * filePointer, "formatting specifiers" , variables);
```

Used to print to a file. Identical to `printf()` except requires a pointer to the file as the first argument.

scanf()

```
int scanf("formatting specifiers" , * variables);
```

Works much like `printf()`, but in reverse. Typically used to read information from the keyboard. Instead passing `scanf()` the name of variable, you pass `scanf` pointers to the variables. For example:

```
short yourWeight;
```

```
scanf("%hd", &yourWeight);
```

Note: when you use the name of an array without the brackets [], the name is acting as a pointer to the array:

```
char yourName[10];
```

```
scanf("%s", yourName);
```

For more information about pointers, see the section *Definitions: Common Variable Types : Notes on pointers – when to use * or &*.

fscanf()

```
int fscanf(FILE * filePointer, "formatting specifiers", * variables);
```

The file counterpart to `scanf()`.

C++ *iostream*

C++ has two new methods for input and output: `cin` and `cout`. Personally, I prefer not to use them. However, you might encounter these methods sometimes, so it is good to know how they work.

cin

```
cin >> variable;
```

`cin` reads in data from the `iostream` (which can be from the file or from the console) and places it into *variable*.

cout

`cout` is the output counterpart of `cin`.

```
cout << "You typed in " << variable << '\n';
```

Words and Strings

What is a String?

Strings are simply arrays of characters. All C strings are terminated with a NULL '\0'. Pascal strings are length prefixed – that is, the first `char` indicates the length of the string. Since you will be programming in C/C++, you will be dealing with C strings most of the time.

Importing Strings from a text file

Let's say you are performing an experiment in which you will be presenting subjects with a series of words. You know that no words will be greater than 9 characters in length. Each word also has another word that is highly associated with it. So you might have a text file that is laid out like this:

```
cat    dog
house  home
doctor nurse
...
```

where each pair occurs on a separate line, and the words belonging to a pair are separated by tabs.

Our goal is to read those words from the text file “MyWords” so that we can use them in your our experiment.

```

char   Target[100][10];    // 100 target words, 9 characters long
char   Associate[100][10]; // 100 associated words, 9 characters long

void   LoadWords(void)
{
    short  i;
    FILE   *fp;           // the file pointer

    int     err;

    if ((fp=fopen (":MyWords", "r"))==NULL) // open MyWords for reading
    {
        cout << "Cannot open file: MyWords \n";
        exit(0); }
    for(i=0; i<100; i++)
    {
        fscanf(fp, "%s\t%s\n", &Target[i], &Associate[i]);
    }
    fclose(fp);
}

```

Remember: although our words are no more than 9 characters long, we must reserve space at the end of our strings for the NULL character. Therefore, our two arrays of strings, `Target` and `Associate`, must be at least 10 characters long.

Earlier, we showed that the name of an array (thus, the name of string) used without the brackets `[]` acts as a pointer to the array (or string). So, in our previous `scanf()` example, a `&` was not required before the name of the variable:

```

char   yourName[10];

scanf("%s", yourName);

```

However, in the `LoadWords()` example, we are passing *arrays* of strings to `fscanf()`. The reason we precede `Target[i]` with a `&` (yielding `&Target[i]`) is that we need to pass `fscanf()` pointers to which `Target` and which `Associate` we are reading in at the moment.

String Manipulation Functions

strcpy()

```
char * strcpy(char * to, const char * from);
```

`strcpy` copies the string *from* into the string *to*. Let, *from* = “ooh!”, *to* = “ahh!”.

```

printf("Before strcpy %s\n", to);
strcpy( to, from);
printf("After strcpy %s\n", to);

```

```

Before strcpy ooh!
After strcpy ahh!

```

Note: if *from* is greater in length than *to*, then *to* is cut short. Let, *from* = “Leopard”, *to* = “cat”.

```
char    from[8];
char    to[4];

printf("Before strcpy %s\n", to);
strcpy( to, from);
printf("After strcpy %s\n", to);

Before strcpy cat
After strcpy Leo
```

strcat()

```
char * strcat(char & to, const char * from);
```

We’ve already seen `strcat` used in `GetSubjectInfo()`. `strcat` concatenates (adds on to the end of) one string onto another. Let, *from* = “ooh!”, *to* = “ahh!”.

```
printf("Before strcat %s\n", to);
strcat( to, from);
printf("After strcat %s\n", to);

Before strcat ooh!
After strcat ooh!ahh!
```

atoi()

```
int atoi(const char * integerString);
```

`atoi` converts a string into an integer value (*alphanumeric to integer*?). It stops converting the string when it comes upon a character that can not form a number.

```
int    myInt;
char   myString[10];

strcat( myString, "123abc456");    // make myString = 123abc456
myInt = atoi(myString);           // convert myString to the int
printf( "%d\n", myInt);
```

123

atof()

```
double atof(const char * floatString);
```

`atof` is the floating point equivalent of `atoi`. Like `atoi`, `atof` stops converting the string when it comes upon a character that is can not form a number.

```
double myDouble;
char   myString[10];

strcat( myString, "123.4abc56");           // make myString = 123abc456
myDouble = atof(myString);                // convert myString to the int
printf( "%Lf\n", myDouble);

123.4
```

Graphics**Colors**

The MacOS allows you to define your own colors – up to 16 million different colors are possible, depending on the color depth of your screen.

RGBColor

```
typedef struct RGBColor {
    unsigned short red;
    unsigned short green;
    unsigned short blue;
} RGBColor;
```

An `RGBColor` is simply a struct made up of the three primary colors red, green, and blue. The values for red, green, and blue can range from 0 to 65535.

```
RGBColor RedRGB, YellowRGB;

RedRGB.red   = 65535;
RedRGB.green = 0;
RedRGB.blue  = 0;

YellowRGB.red   = 65535;
YellowRGB.green = 65535;
YellowRGB.blue  = 0;
```

RGBForeColor

```
void RGBForeColor(RGBColor * color);
```

RGBForeColor sets the foreground color to the closest match it can find for the RGBColor passed to it. If the current color depth is 24 bit color mode, there will be a perfect match. For lesser color modes, RGBForeColor will calculate the closest match. The foreground color controls the color of the pen, or whatever is being drawn in the foreground (such as a letter or line).

```
RGBForeColor( &YellowRGB);
```

RGBBackColor

```
void RGBBackColor(RGBColor * color);
```

RGBBackColor sets the background color to the closest match it can find for the RGBColor passed to it. The background color controls the color of the background and is used for filling polygons (Rects) and such.

```
RGBBackColor ( &RedRGB);
```

Indexed vs. Direct Colors

Both 16-bit and 24-bit color modes are *direct* color modes. That is, the MacOS will map your RGBColor directly to the screen. With 16-bit color, since the steps between the colors is coarser than 24-bit mode, there is some data loss, but it difficult to perceive.

On the other hand, 8-bit color mode is an *indexed* mode. You are limited to 256 different colors from the millions of different colors that are available in 24-bit mode. These 256 different colors are stored in CLUTs (Color Look-up Tables). Therefore, if you chose to do so, you could limit the colors in your display to 256 shades of red.

If you do not define your own CLUT, your RGBColors will be matched to the closest available color in the system's currently active CLUT. This is fine for most work outside of psychophysics.

Although using CLUTs is beyond the scope of this manual, I will leave you with some pointers. CLUTs are stored in your program's resource fork (see *Resources*, below). You probably want to use the function `NewPalette()` to load your CLUT. And of course, see *Think Reference* for more details.

Transfer Modes

In my own personal opinion, the coolest thing about the MacOS are the color transfer modes (I need to get out more often, don't I?). Transfer modes allow you to perform some truly phenomenal graphics manipulations.

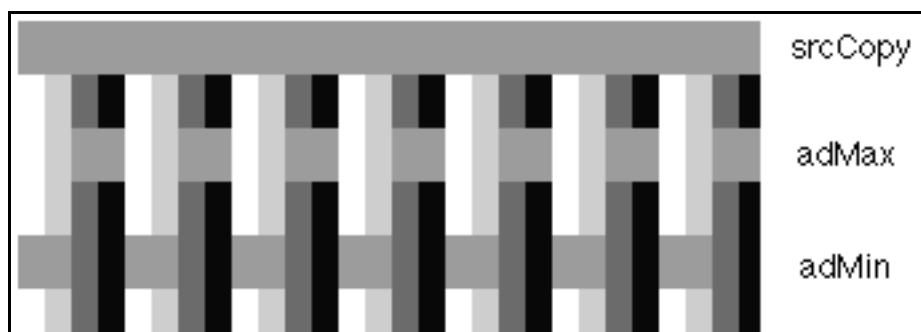
Transfer modes control how the color of one object mixes with another object. The Tcards all use the copy mode `srcCopy`, which tells the computer to completely overwrite the old *destination* image (e.g., what is on the screen) with the new *source* image (what is contained in the Tcard).

There are over ten transfer modes in all. Three often used **arithmetic** transfer modes are `adMax`, `adMin`, and `blend`. The arithmetic transfer modes take into consideration the red, green, and blue values of the *source* and *destination* colors.

adMax make the *destination* pixel equal to the maximum (RGBColor) value of the *source* and the *destination*. In the example below, when gray horizontal lines are draw to the screen, the lines only show up when they overwrite darker colors (e.g., gray > black).

adMin make the *destination* pixel equal to the minimum (RGBColor) value of the *source* and the *destination*. In the example below, when gray horizontal lines are draw to the screen, the lines only show up when they overwrite lighter colors (e.g., gray < white).

blend takes the average of the *source* and *destination* colors.



PenMode() and *TextMode()*

```
void PenMode( newTransferMode );
void TextMode( newTransferMode );
```

`PenMode()` and `TextMode()` control the transfer mode for objects drawn with the pen and text, respectively. `PenMode()` affects such functions as drawing lines.

CopyBits()

The function `CopyBits()` is used by the `Tcards` to copy images. If you want to use a different transfer mode when copying images, copy the appropriate `Tcard` function (e.g. `.Show()`), give it a new name, and change `srcCopy` to one of the other transfer modes.

Advanced Tcard functions

Masks

Not psychophysical masks, but Macintosh QuickDraw masks! Macintosh masks allow you to pass one image (the *source* or *from* image) through another image (the *mask*).

MaskToCard()

```
int Tcard.MaskToCard(Tcard ToCard, Tcard MaskCard);
```

MaskToCard is used to copy an image from a Tcard, pass it through a mask, and place it in ToCard.

```
FromCard.MaskToCard( ToCard, MaskCard);
```

CopyToCard()

```
int Tcard.CopyToCard(Tcard ToCard);
```

CopyToCard() copies the contents of a Tcard to another Tcard.

```
FromCard.CopyToCard( ToCard);
```

Resources

Every MacOS program consists of two parts - the data fork and the resource fork. The C/C++ code that you compile is transformed into the data fork. The resource fork can contain such things as fonts unique to the program, pictures, sounds, CLUTs, etc.

The file **Generic Psych.rsrc** is automatically contained in your program. It contains the definitions for the windows your program uses.

ResEdit

Double clicking on **Generic Psych.rsrc** while in *CodeWarrior* (it is probably located in **segment 3** of your project) automatically launches ResEdit.

Creating new resources

To create a new resource, go to the “Resource” menu in ResEdit and choose “Create New Resource K”. You will be presented with a variety of resource types to choose from. The two types that will be discussed in this manual are PICT and snd.

Each instance of a resource type has a unique ID #, starting with 128. To insert a new resource simply paste it into the appropriate (PICT or snd) resource window. Most programs, such as SuperPaint, Photoshop, or SoundEdit, allow you to copy items that can be pasted in as resources. You might be asked if you want the new resource to have a new unique ID. If you do not want to replace any of the old resources, select “Unique ID”.

Using resources in your program

```
Handle GetResource( 'resourceType' , resourceIDnumber );
```

The function `GetResource()` is used to access resources stored in the resource fork of your program. *resourceType* (must be in single quotes) is the type of resource your accessing, such as a PICT or a snd. *resourceIDnumber* is the ID number of your resource.

A Handle is a pointer to a pointer. Some functions that deal with your resources might require you to pass them the handle to your resource. For specific examples, see the section on *snd* resources below.

Picture (PICT) resources

```
PicHandle GetPicture( pictureIDnumber );
```

Although you can use `GetResource()` to access PICTs (the two functions are functionally equivalent), the preferred method is to use `GetPicture()`. Let’s say that we are going to use two different picture stimuli in our experiment. To accomplish this, we’ll make an array of PICTs:

```
PicHandle    stimPict[2]
...
void  InitStimPicts(void)           //gets our PICT resources
{    stimPict[0] = GetPicture(128);  // picture of dog
    stimPict[1] = GetPicture(129);  // picture of a cat
}
```

Displaying

```
void DrawPicture( PicHandle thepicture, Rect destinationRect);
```

The function `DrawPicture()` requires two pieces of data: 1) the PICT resource handle [we used `GetPicture()` to get this already], and the size of the rectangle. The size of the rectangle is preferably the size of your object. If it is not (and it does not have to be), `DrawPicture()` will stretch or shrink your image to fit the rectangle.

```
void DrawMyPICTs(void)
{
    stimCard.BeginDraw();
    MakeWhite();           // makes the pen color white
    PenMode(adMax);
    PenPat(&qd.white);     // makes the pen pattern "white"
    MakeWhiteBack();      // this is important
    //
    SetRect(&myRect, 310, 230, 330, 250); // L, T, R, B
    DrawPicture(stimPict[0], &myRect);
    stimCard.EndDraw();
}
```

Notice that `MakeWhiteBack()` is called before we draw the picture. I have found that sometimes the background color needs to be white. If you are having problems getting your PICTS to appear, try setting the background color to white before calling `DrawPicture()`.

Sound (snd) Resources

```
void CallSndPlay(short duration) // Plays prerecorded sounds stored in the
{
    Handle          mySndHandle;    // resource fork of a program.
    SndChannelPtr  mySndChan;      // handle to an 'snd' resource
    OSerr          myErr;          // pointer to a sound channel

    mySndChan = nil;              // Initialize channel pointer for error checking

    // Read in 'snd' resource from resource file
    mySndHandle = GetResource ('snd ', 128);
    ...
```

Above is the first half of an example function called `CallSndPlay`. This section of the function simply defines some local variables and gets `snd` resource # 128.

Playing

```
OSerr SndPlay( SndChannelPtr channel, Handle sndHandle, Boolean async);
```

```
myErr = SndPlay (mySndChan, (SndListHandle)mySndHandle, TRUE);
```

Above is an example of `SndPlay()`, and below is an example of `SndPlay()` used in the second half of the function `CallSndPlay()`. `SndPlay()` requires 3 items to be passed to it: 1) a

pointer to a `SndChannel`, 2) a handle to a `snd` resource, and 3) whether we want asynchronous (`TRUE`) or synchronous (`FALSE`) operation.

1) Making a handle to a `SndChannel` is easy: a) simply define it somewhere in your variable list, b) set it to `nil` before using it, and c) the computer takes care of the rest.

2) We already used `GetResource()` to get the handle to our `snd` resource. For some reason `mySndHandle` is typecast as a `SndListHandle`. I'm not sure why, but it does work! Perhaps it is due to Apple's change to the Universal Headers.

3) You should probably set the third parameter to `TRUE`. In asynchronous i/o, the computer issues the sound command, the command enters a queue for processing by the sound chips, and then the computer returns to the next task and eventually the sound chips carry out their commands. Synchronous (`FALSE`) i/o requires the computer to wait until the sound command has completed processing.

```
...
if ( mySndHandle != nil )    // if getting the sndHandle was successful...
{
    // ... then play that tune!
    myErr = SndPlay (mySndChan, (SndListHandle)mySndHandle, TRUE);
    WaitSynchs(1);          // pausing appears to be necessary
    if ( myErr )            // good etiquette
        DoError(myErr);
}
}
```

Keyboard Polling Accuracy Issues

It is notoriously difficult to get accurate timing from the Macintosh ADB port (as used by all Macintoshes since the late 1980's). One method of getting reasonable accurate results was to use the KeMo Reaction Time Utilities (Costin, 1993). However, the KeMo utilities only worked with Symantec C++ and Think Pascal, and these two development environments have fallen out of favor since the introduction of CodeWarrior (Metrowerks Corporation, 1995).

In order to get reasonably accurate timing while using Metrowerk's CodeWarrior, I had to develop my own routines. Much of my code was inspired by discussions recorded in the MacPsych archive and by *Inside Macintosh: Devices* (Apple Computer, 1994). The following is an empirical study of Macintosh timing issues, and was inspired by Mogg and Bradley (1995).

Method

Equipment

All tests were performed on a Macintosh Quadra 840av equipped with an Apple Keyboard 2. Stimuli were displayed using a Sony 17sf2 monitor set to 640x480 pixels (66 Hz.). A Radio Shack cadmium-sulfide photocell (catalog number 276-1657) was used to trigger the keyboard.

All code excluding the trials using the KeMo routines was compiled using Metrowerk's CodeWarrior 7 C++ (Metrowerks, 1995) with all 68k optimizations. The trials using the KeMo routines were compiled using Symantec's C++ for Macintosh 7.0.3 (Symantec, 1994).

The following routines were tested:

	Notes	Partial Extensions (Optimal Methods)		Full Extensions (Non-Optimal Methods)	
		Mouse On	Mouse Off	Mouse On	Mouse Off
KeMoWait()	KeMoWait() and KeyBoardPol() both use		X		X
KeyBoardPol()	Macintosh ADB polling routines.	X	X		
GetAKey()	Uses the Macintosh Event Manager GetNextEvent().		X	X	X
GetKeys()	Uses the Toolbox function GetKeys().			X	X

Design

The methods tested were divided into the optimal keyboard polling methods (KeMoWait() with the mouse turned off, KeyBoardPol() with the mouse turned off, KeyBoardPol() with the mouse turned on, and GetKeys() with the mouse turned off) and the non-optimal methods (KeyBoardPol() with the mouse turned off, GetAKey() with the mouse off, GetAKey() with the mouse on, GetKeys() with the mouse turned off, and GetKeys() with the mouse turned on). All non-essential extensions (e.g. QuickTime) were turned off for the optimal methods, whereas all extensions were left on for the non-optimal methods.

Procedure

In order to electrically trigger the keyboard, a photocell was wired to '/' key so that a flash of light would cause the circuit to be bridged. The photocell was attached to the upper left corner of

the monitor (see Mogg and Bradley (1995) for a similar methodology). All testing was done in a darkened room, and a 200x120 pixel white rectangle was flashed to the top left corner of the screen to trigger the photocell.

```

for(i=0; i< NumTrials; i++)          // 2600 or 600 trials
{
  WaitSyncs(10);                    // pause for 150 msecs
  StartTimer();                      // Start the timer
  CueCard.Show();                   // Flash the white rectangle
  PollReturn = KeyBoardPol();        // Wait until keyboard is triggered
                                      //   by the rectangle.
  gotTime = StopTimer();             // Stop the timer.
  myTime[Order] = gotTime;          // Record the time.
  BlankCard.Show();                 // Make the screen black.
}

```

The keyboard was sampled 2600 times for the optimal methods and 600 times for the non-optimal methods. Because attaching the wires from the photocell to the keyboard took some time to ensure that the wires were properly seated, the first 100 data points were excluded from each experiment, yielding 2500 and 500 for the optimal and non-optimal methods, respectively.

Results

Optimal Keyboard Polling Methods

Not surprisingly, the trials in which the mouse was turned off yielded the smallest standard errors. Although there are several reasons why having the mouse turned on might yield larger variances, the simplest explanation is to say that the more devices that share the same bus, the greater the variance. Interestingly, the KeMo utilities purport to correct for the time it takes to transfer information across the ADB bus, which should yield a mean response time of 0 milliseconds, rather than the 12.73 msec shown in Table 1. Furthermore, KeMo Test 1.5 reports that KeMo is accurate to within ± 1.7 msec, however as can be seen by the standard deviation of 2.47 msec in Table 1 and the distribution of response times in Figure 1, KeMo's accuracy is much less than the reported ± 1.7 msec.

	KeMo mouse-off	KeyboardPol mouse-off	KeyboardPol mouse-on	GetKeys mouse-off
Mean	12.73	17.22	17.06	19.71
Standard Deviation	2.47	2.61	5.20	3.96

Table 1. Mean response times and standard deviations for the optimal keyboard polling methods in milliseconds.

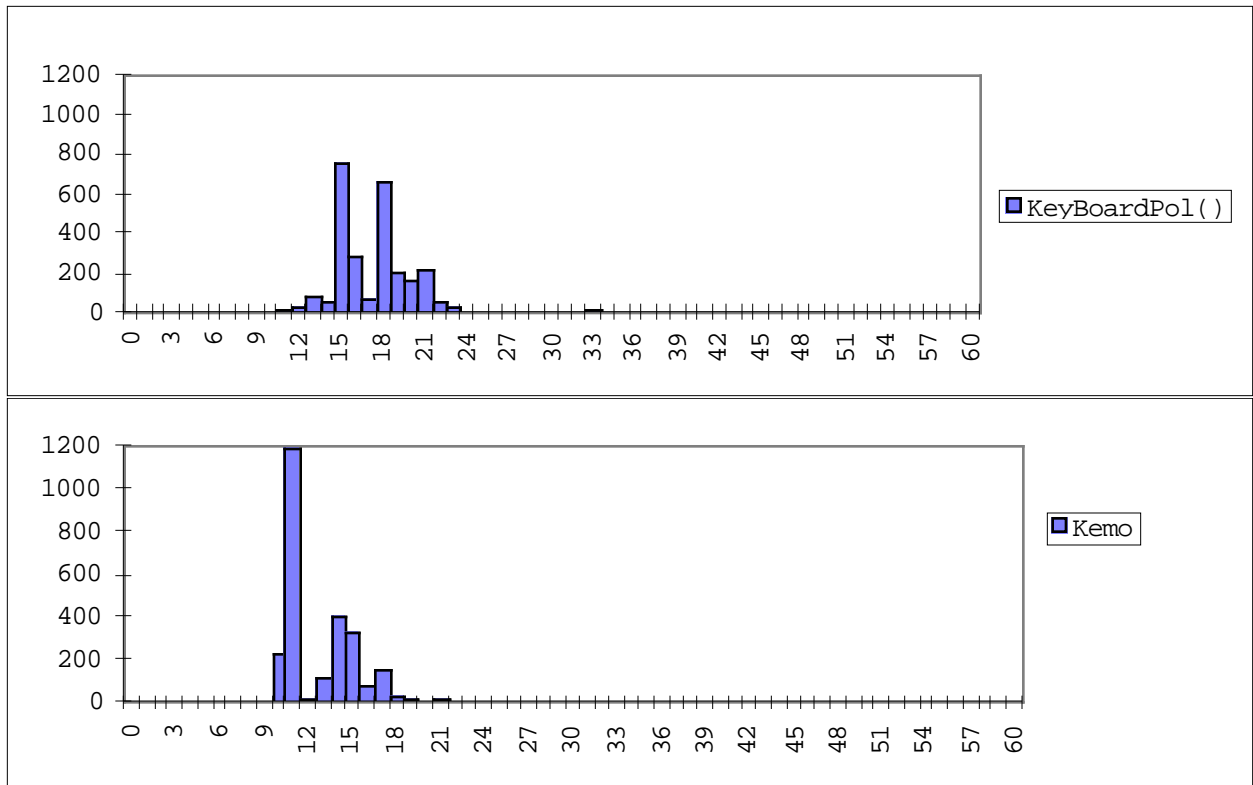


Figure 1. Distribution of response times for **KeyBoardPol()** and **KeMoWait()**. Response times are on the x-axis and counts are plotted on the y-axis. Data consists of 2500 trials.

Non-optimal Keyboard Polling Methods

As expected, the non-optimal keyboard polling methods fared much worse than the optimal keyboard polling methods. In particular, using the event manager (e.g. `GetAKey()`) to get response times is notoriously inaccurate, and this is reaffirmed in Table 2. Also, when comparing Table 1 with Table 2, it can be seen that the number of extensions that are running has an effect on keyboard accuracy, with less extensions providing the greater accuracy.

	KeyboardPol mouse-off	GetKeys() mouse-off	GetKeys() mouse-on	GetAKey() mouse-off	GetAKey() mouse-on
Mean	16.59	19.75	20.07	24.20	50.57
S.D.	3.73	4.00	4.61	3.98	12.38

Table 2. Mean response times and standard deviations for the non-optimal keyboard polling methods in milliseconds.

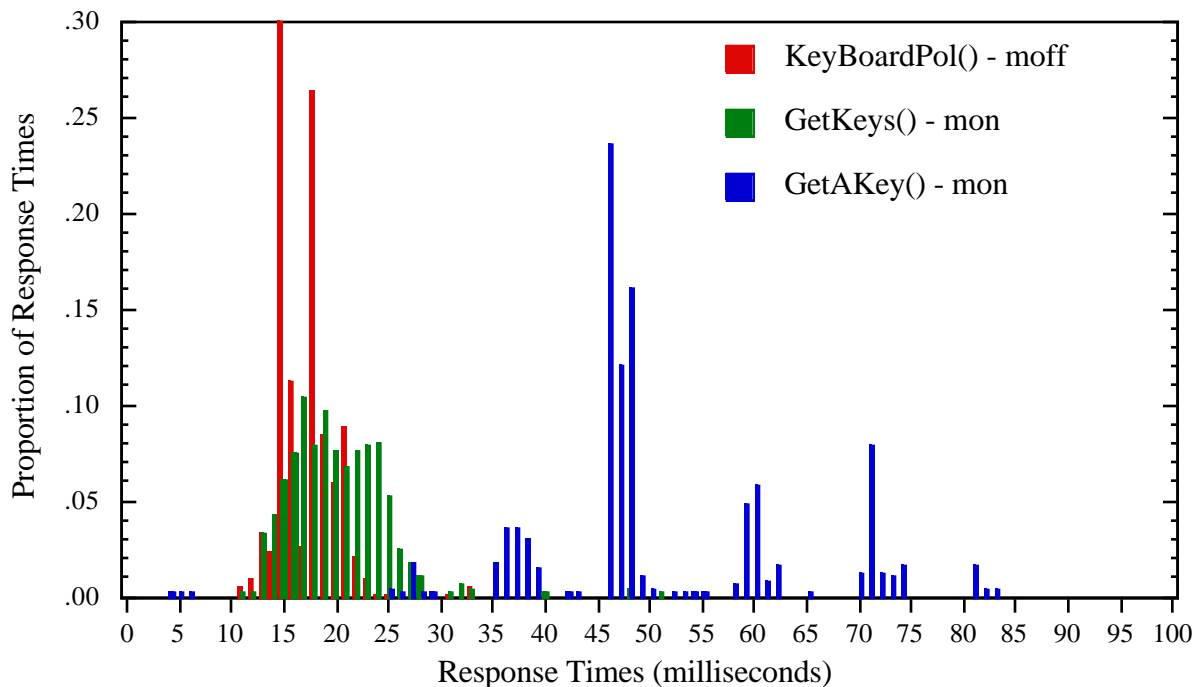


Figure 2. Distribution of response times for `KeyboardPol()` with the mouse turned off and minimal extensions and `GetKeys()` and `GetAKey()` with the mouse on and full extensions. Response times are on the x-axis and counts are plotted on the y-axis. Distributions are based on 2500 samples for `KeyboardPol()` and 500 samples for `GetKeys()` and `GetAKey()`.

Conclusions

`KeyboardPol()` provides an adequate substitution for the KeMo routines as long as the mouse is turned off and the system is using a minimal number of extensions.

Turning the MouseOff()

To turn the mouse off, call the function `MouseOff()` early in the program. You must remember to call the function `MouseOn()` before the program exists, or else the mouse will not work. It is a good idea to comment out `MouseOff()` while you are debugging, otherwise a harmless crash might leave you with the mouse turned off.

KeyboardPol() or GetaKey()?

In a nutshell, `KeyboardPol()` should be used with response time experiments, and `GetaKey()` should be used with accuracy experiments. From the above experiment, it is obvious that `KeyboardPol()` has an advantage for response time experiments, but why not use `KeyboardPol()` all of the time? The answer is simple: for some unknown reason, `KeyboardPol()` will not accurately record keystrokes that occur in close succession. Therefore, if you are performing an accuracy experiment and you are expecting more than one keystroke, you should use `GetaKey()`.

SyncAndPoll()

```
SyncAndPollData SyncAndPoll(short Syncs);
```

```
typedef struct SyncAndPollData
{   Byte    KeyPressed;
    long    KeyTime;
};
```

`SyncAndPoll()` is a Frankenstein routine: it is the illegitimate offspring of `WaitSyncs()` and `KeyboardPol()`. It is designed to detect keystrokes while animation is taking place on the screen. Use `SyncAndPoll()` whenever you might legitimately expect a keystroke during a `WaitSyncs()`.

To use `SyncAndPoll()`, pass it the number of refreshes to wait (like you do with `WaitSyncs()`). If a keystroke is detected, it returns which key was pressed and the response time in the `SyncAndPollData` structure.

```
SyncAndPollData  mySyncAndPoll;    // my SyncAndPollData variable

...
for(i=0; i< NumTrials; i++)
{   StartTimer();
    mySyncAndPoll = SyncAndPoll(10); // Wait 10 refreshes and check
    ...                             // for a keypress at the same time.
```

`SyncAndPoll()` is only accurate to within one refresh.

Definitions

Common Variable Types

Name	Size (bits)	Type	Range
char	8	signed byte	-128...127
Byte	8	unsigned byte	0...255
short	16	signed word	-32768...32767
*int	32	signed long word	-2,247,483,648...2,247,483,647
long	32	signed long word	-2,247,483,648...2,247,483,647
float	32	signed long word	1.17549e-38...3.40282e+38
†double - 68k	96		1.68105e-4932...1.18973e+4932
†double - PPC	64		2..22507e-308...1.79769e+308
Boolean	16	signed word	0...1, 0...non-zero, false or true, FALSE or TRUE

* int has been known to change with compilers. I recommend using short or long.

† The default settings for double is 96 bits when compiling 68k applications and 64 bits when compiling for PowerPC. However, for 68k machines, if “8 byte doubles” is turned on in the preferences, then double is 64 bits in length. When compiling for 68k machines without math coprocessors, double is 80 bits in length.

Notes on pointers – when to use * or &

Normally, variables represent some value. Pointers, on the other hand, represent a memory address. Most of the time you probably do not need to use pointers. However, you might come across functions that require pointers.

Instead of using pointer (variables defined with an asterisk ‘*’ in front of them), you might find it easier use normal variables and dereference them (place a ‘&’ in front of them) instead. This makes your code easier to read – you know that you are manipulating a memory address rather than a value because the variable has a ‘&’ in front of it. Note that this method is not always possible: some functions might require a specific type of system pointer (e.g. SNDHandle).

	variable	pointer
definition	short x;	short *x;
value of x	x	*x
pointer to x	&x	x

A Quick example of using pointers and dereferenced variables

short x;	Memory Address	Variable	Value
short *y; // a pointer	1000	x	0
short a;	1001	y	0
short b;	1002	a	0
	1003	b	0
void main(void);	Memory Address	Variable	Value
{ x = 10;	1000	x	10
*y = 19;	1001	y	19
a = 25;	1002	a	25
b = 30;	1003	b	30
.			
.			
*y = x; // y = value of x	Memory Address	Variable	Value
a = 25;	1000	x	10
&b = a; // b points to a	1001	y	10
.	1002	a	25
.	1003	b	25
.			
.			
.	Memory Address	Variable	Value
x++; // increment x	1000	x	11
a++; // increment a	1001	y	10
}	1002	a	26
	1003	b	26

Pointers, Arrays, and Strings

Pointers, arrays, and strings are intimately related. A string is an array of characters, with the last character equal to zero. The name of an array (or string) with out brackets is a pointer to the first item in the array. Thus, when you pass the name of a string to a function, you are really passing it a pointer to your string, not the contents of your string.

The example below illustrate this using the string “brown fox”

char myString[10]	// 9 characters + room for terminating NULL	
Memory Address	Variable	Value
1000	myString[0]	b
1001	myString[1]	r
1002	myString[2]	o
1003	myString[3]	w
1004	myString[4]	n
1005	myString[5]	
1006	myString[6]	f
1007	myString[7]	o
1008	myString[8]	x
1009	myString[9]	\0

C and Pascal Strings

On occasion, you might come across Apple Toolbox functions that require Pascal style strings. For example, the function `DrawString()` requires a Pascal string (the newer function `drawstring()` requires a C string). C strings are terminated by a null ‘\0’ character, whereas Pascal style strings are length prefixed (which limits string length to 255 characters).

For example:

C string	b	r	o	w	n		f	o	x	\0
Pascal String	9	b	r	o	w	n		f	o	x

```
char myCString[10];
char myPstring[10];
```

When passing quoted text to Macintosh toolbox function, the first character must be a “\p”. Since you are writing code in C, quoted text will normally be handled like C strings. Prefixing the text with “\p” tells the compiler to treat the text as a Pascal string.

```
DrawString("\pThe quick brown fox");
```

The *Psychological Toolbox* functions `C2Pstring()` and `P2Cstring()` are also available to convert between the two string formats, and are covered in more detail in the appendix.

Handles

Handles are pointers to pointers. In your normal course of programming, you will probably only encounter handles when using some Macintosh Toolbox functions. For example, the functions *Psychological Toolbox* function `SetColorDepth()` is passed a handle to a graphics device.

```
void SetColorDepth(GDHandle, short);
```

Common Structures and Variables

point 🍏

```
struct Point {
    short  v;           // vertical coordinate
    short  h;           // horizontal coordinate
};
typedef struct Point Point;
```

Rect 🍏

```
struct Rect {
    short  top;
    short  left;
    short  bottom;
    short  right;
};
typedef struct Rect Rect;
```

Although you can set the coordinates of a `Rect` by writing directly to its fields (e.g. `myRect.top = 0;`), the preferred method is to use `SetRect()`.

Str255 🍏

A Pascal style string 255 characters (256 bytes) in length. Often used by Macintosh Toolbox functions.

Common Macintosh Functions

PenMode() 🍏

```
void PenMode(newPenMode);
```

Sets the pattern transfer mode to be used with pen drawing and painting. The default setting is `patCopy`, which overwrites the background. For more details, see *Think Reference*.

PenPat() 🍏

```
void PenPat(newPenPattern);
```

Sets the pen pattern, which is used in all QuickDraw drawing and painting operations. The default setting is `black`. For more details, see *Think Reference*.

PenSize() 🍏

```
void PenPat(short width, short height);
```

Sets the width and height of the pen in pixels for the current `GrafPort` (e.g. for a single instance of a `Tcard`).

MoveTo() 🍏

```
void MoveTo(short h, short v)
```

Moves the drawing pen to the specified local coordinates. `MoveTo(0,0)` moves the pen to the top left pixel of the display.

LineTo() 🍏

```
void LineTo(short h, short v)
```

Draws a line from the current pen coordinate (see `MoveTo()`) to the coordinate specified in `LineTo()`.

SetRect() 🍏

```
void SetRect(Rect *r, short left, short top, short right, short bottom)
```

Sets the coordinates of a `Rect`.

```
Rect myRect;
```

```
SetRect(&myRect, 0, 0, 479, 639);
```

PaintRect() 🍏

```
void PaintRect(Rect *r)
```

Fills a `Rect` with the current background color

FrameRect() 🍏

```
void FrameRect(Rect *r)
```

Draws an outline just inside the border of a rectangle using the current mode, pattern, and pen size.

DrawString() 🍏

```
void DrawString(Str255 theString);
```

Draws a string in the current `GrafPort` (e.g. the current `Tcard` or the screen) using the current font, font size, style, and transfer mode. Characters are drawn at the current pen location at the baseline of the character (i.e. roughly the bottom-left corner). `DrawString()` uses Pascal style strings and has been superseded by `drawstring()`, which uses C style strings.

drawstring() 🍏

```
void drawstring(char *theString);
```

Identical to `DrawString()`, except accepts C style strings.

TextFont() 🍏

```
void TextFont(short fontNumber);
```

Sets the font of the current `GrafPort`. See *Think Reference* for a list of font numbers, or use the function `GetFNum()` to look up a font's number based on its name.

GetFNum() 🍏

```
void GetFNum(Str255 fontName, short *fontNum);
```

Returns the font number that matches a font name.

```
short myFontNum;
```

```
GetFNum( "\pGeneva", &myFontNum);
```

TextSize() 🍏

```
void TextSize(short newsiz);
```

Sets the font size for the current GrafPort (e.g. Tcard).

TextFace() 🍏

```
void TextFace(Style newface);
```

Sets the text face (e.g. normal, bold, italic) for the current GrafPort. See *Think Reference* for a list of text face codes.

Common C/C++ Functions**Input/Output Functions****printf()** ∞

```
int printf(const char *format,...);
```

Writes a formatted string to standard output (usually the console window). To draw characters as graphics (the normal Macintosh method), see `drawstring()`.

fprintf() ∞

```
int fprintf(FILE *stream, const char *format,...);
```

Writes a formatted to a stream (i.e. a FILE).

scanf() ∞

```
int scanf(const char *format,...);
```

Reads formatted text in from the console window. Useful for getting subject information before the experiment begins.

fscanf() ∞

```
int scanf(FILE *stream, const char *format,...);
```

Reads formatted text in from a file.

getchar() ∞

```
int getchar(void);
```

Reads the next character from standard input (the console window) and returns the integer value of the character.

```
MyChar = getchar();
```

getc() ∞

```
int getc(FILE *stream);
```

Reads the next character from a stream (a FILE) and returns the integer value of the character.

```
FILE *fp; // a pointer to a file
```

```
MyChar = getc(fp);
```

gets() ∞

```
char *gets(char *string);
```

Reads a line from standard input (e.g. the console window) and stores in *string*. `gets()` stops reading when it reads a newline or an EOF (end of file) character.

String Manipulation Functions***strlen()*** ∞

```
unsigned long strlen(char *myString);
```

Returns the length of a C style string, including all white characters. The count does not include the terminating NULL ('\0') character.

strcpy() ∞

```
char *strcpy(char *toString, char *fromString);
```

Copies the contents of *fromString* to *toString*.

strncpy() ∞

```
char *strncpy(char *toString, char *fromString, size_t n);
```

Copies *n* characters from *fromString* to *toString*.

strcat() ∞

```
char *strcat(char *toString, char *fromString);
```

Concatenates (adds to the end of) the contents of *fromString* to *toString*.

strncat() ∞

```
char *strncat(char *toString, char *fromString, size_t n);
```

Concatenates (adds to the end of) *n* characters from *fromString* to *toString*.

strcmp() ∞

```
int strcmp(char *String1, char *String2);
```

Compares two strings. If the two strings are equal, `strcmp()` returns a value of zero.

strstr() ∞

```
char *strstr(char *String, char *SubString);
```

Scans for a substring within a string. Returns a pointer to the first occurrence of the substring within the string, or a NULL (0) if the substring is not found.

atoi() ∞

```
int atoi(char *String);
```

Converts a string to an integer value. It stops converting the string when it comes upon a character that can not form a number.

```
int    myInt;
char   myString[10];

strcat( myString, "123abc456"); // make myString = 123abc456
myInt = atoi(myString);        // convert myString to the int
printf( "%d\n", myInt);
```

```
123
```

atof() ∞

```
double atof(char *String);
```

Converts a string to a double precision floating point value. It stops converting the string when it comes upon a character that can not form a number.

```
double myDouble;
char   myString[10];

strcat( myString, "123.4abc56"); // make myString = 123abc456
myDouble = atof(myString);      // convert myString to the int
printf( "%Lf\n", myDouble);
```

```
123.4
```

Psychological Toolbox Global Variables

onScreenGD 🍏

```
GDHandle onScreenGD;
```

A handle to a GDevice, in this case, the viewable computer screen. Used by the functions `SetColorDepth()` and `ReSetColorDepth()`.

onScreen 🍏

```
GWorldPtr onScreen;
```

A pointer to a CGrafPort (color graphics port), in this case, the viewable computer screen. Used by the functions `SetColorDepth()` and `ReSetColorDepth()`.

Psychological Toolbox Functions

Initialization and Environment Functions

InitPsychRoutines() 🍏

```
void InitPsychRoutines(void);
```

Initializes the environment, including seeding the random number generator, initializing the Macintosh Toolbox, initializing the monitor's graphics device, and defining colors. This function should be called very early in the program.

WindowInit() 🍏

```
void WindowInit(short windowNumber);
```

`WindowInit()` initializes the onscreen window. Two sizes of windows are currently available:

<i>windowNumber</i>	size in pixels	resource ID
0	640 x 480	128
1	832 x 624	129

The windows are defined in the file `Generic Psych.rsrc`. To define more window sizes, you must create a new WIND resource in `Generic Psych.rsrc` using *ResEdit*. If a window resource is not found that corresponds to the *windowNumber*, `WindowInit()` beeps three times.

CleanUp() 🍏

```
void CleanUp(void);
```

Called when the experiment is completed. Automatically resets the color depth, hides the graphics window, makes the cursor visible, makes the menubar visible, and resets the keyboard and mouse.

HideMenuBar() 🍏

```
void HideMenuBar(void);
```

Hides the menu bar. The menubar will still be visible during debugging.

ShowMenuBar() 🍏

```
void ShowMenuBar(void);
```

Shows the menu bar.

SetColorDepth() 🍏

```
void SetColorDepth(GDHandle onScreenGD, short bitDepth);
```

Sets the amount of colors that can be visible in the graphics device (usually the monitor). For best results, the monitor and Tcards should be set to the same color depth.

<i>BitDept</i> <i>h</i>	number of visible colors
1	2 (black and white)
2	4 (b, w, gray, & hilight)
4	16
8	256 (indexed colors)
16	32,767
32	16,777,216

ReSetColorDepth() 🍏

```
void ReSetColorDepth(GDHandle onScreenGD, short bitDepth);
```

Returns the color depth to *bitDepth*.

MouseOn() 🍏

```
void MouseOff(void);
```

Turns the mouse off. Turning the mouse off increases the accuracy of the keyboard polling routines. Not recommending during debugging (unless you want to debug without using the mouse!).

MouseOff() 🍏

```
void MouseOn(void);
```

Turns the mouse on. Must be called at the end of the program to ensure that the mouse works. `CleanUp()` calls `MouseOff()`, and may be used instead.

Timing and Response Functions**StartTimer()** 🍏

```
void StartTimer(void);
```

Starts the millisecond timer. On 68k machines (for compatibility with systems older than 7.5), the timer counts down from a preset time to zero. The default countdown setting is set to 4 seconds. See `SetTimeOut()` to increase the timing range. Timing on PowerPC machines is done by checking the number of milliseconds since system startup.

StopTimer()

```
double StopTimer(void);
```

Stops the millisecond timer and returns the value as a double precision floating point.

RestartTimer()

```
void RestartTimer(void);
```

Restarts the millisecond timer, without erasing the time that is left. Used for compatibility with 68k machines using pre-System 7.5.

SetTimeOut()

```
void SetTimeOut(long milliseconds);
```

Sets the initial countdown time in milliseconds on 68k machines. The default setting is 4000 milliseconds.

KeyboardPol()

```
Byte KeyboardPol(void);
```

Continuously polls the keyboard until a key has been pressed. Returns the keyboard character code (see *Think Reference: Keyboard Character Codes*). Character codes might change depending on which model keyboard is being used. Keyboard polling is more accurate if the mouse is turned off (see `MouseOff()`) and a minimal amount of extensions are in use.

`KeyboardPol()` occasionally has problems registering a second keypress if the keypress occurs within 200 msec or so of the first keypress. Because of this problem, `GetAKey()` is the preferred method to get responses if timing accuracy is not a consideration.

SyncAndPoll()

```
SyncAndPollData SyncAndPoll(short Syncs);
```

Effectively a combination of `KeyboardPol()` and `WaitSyncs()` – Polls the keyboard while counting the number of screen refreshes. Keyboard polling is much less accurate than `KeyboardPol()`, but allows the animation to take place while checking the keyboard for responses.

GetAKey()

```
char GetAKey(void);
```

Returns the ASCII value of the key that was pressed. Does not suffer from `KeyboardPol()`'s inability to properly register keystrokes when they occur in close succession. Because `GetAKey()`'s timing resolution is less accurate than `KeyboardPol()`, `GetAKey()` is the preferred method to use when timing accuracy is not an issue.

InitMicrophone() 🍏

```
void InitMicrophone(void);
```

Initializes the microphone routines. Must be used early in the program before calling any other microphone functions.

PollMicrophone() 🍏

```
void PollMicrophone(short threshold);
```

Continuously polls the microphone until the sound level exceeds some *threshold*. Using the program *MicMeterLevel* to get an idea of where to set the threshold.

Graphics Functions***DrawToScreen()*** 🍏

```
void DrawToScreen(void);
```

Makes the computer screen the current graphics device. All subsequent graphics functions (such as `Tcard.Show()`) will be performed on the screen.

WaitSyncs() 🍏

```
void WaitSyncs(short Syncs);
```

Waits *Syncs* refresh rates and then returns. When drawing to the screen, `WaitSyncs()` should be immediately followed by your drawing function:

```
WaitSyncs(10);           // wait 10 refreshes
myCard.Show();          // and then copy myCard to the screen.
```

Use the program *TimeVideo* (Pelli, 1996) or consult the monitor's reference manual to determine your monitor's refresh rate.

Color Functions

The `MakeColor()` functions set the forecolor to a predefined color. You can use these functions as prototypes for your own custom color functions. The components of each color are listed below the functions listings.

The `MakeColorBack()` functions set the backcolor to a predefined color. You can use these functions as prototypes for your own custom color functions. The components of each color are listed below the functions listings.

The `BackToColor(Rect theRect)` functions fill a `Rect` with a predefined color. Useful for erasing `Tcards` or setting the background color of the screen. You can use these functions as prototypes for your own custom color functions. The components of each color are listed below the functions listings.

MakeRed() 🍏

	hex	decimal	percentage
Red	FFFF	65535	100%
Green	199A	6554	10%
Blue	199A	6554	10%

MakeBlue() 🍏

	hex	decimal	percentage
Red	3333	13107	20%
Green	6666	26214	40%
Blue	FFFF	65535	100%

MakeDarkBlue() 🍏

	hex	decimal	percentage
Red	0	0	0%
Green	0	0	0%
Blue	BFFF	49151	75%

MakeGreen() 🍏

	hex	decimal	percentage
Red	0	0	0%
Green	B333	45875	70%
Blue	0	0	0%

MakeYellow() 🍏

	hex	decimal	percentage
Red	CCCC	52428	80%
Green	CCCC	52428	80%
Blue	0	0	0%

MakeWhite() 🍏

	hex	decimal	percentage
Red	FFFF	65535	100%
Green	FFFF	65535	100%
Blue	FFFF	65535	100%

MakeGrey() 🍏

	hex	decimal	percentage
Red	9FFF	40959	62%
Green	9FFF	40959	62%
Blue	9FFF	40959	62%

MakeBlack() 🍏

	hex	decimal	percentage
Red	0800	2048	3%
Green	0800	2048	3%
Blue	0800	2048	3%

MakeWhiteBack() 🍏

Sets the background color to the same color used in `MakeWhite()`.

MakeGreyBack() 🍏

Sets the background color to the same color used in `MakeGrey()`.

MakeBlackBack() 🍏

Sets the background color to the same color used in `MakeBlack()`.

BackToWhite() 🍏

```
void BackToWhite(Rect theRect);
```

Fills *theRect* with the same color used in `MakeWhite()`.

BackToGrey() 🍏

```
void BackToGrey(Rect theRect);
```

Fills *theRect* with the same color used in `MakeGrey()`.

BackToBlack() 🍏

```
void BackToBlack(Rect theRect);
```

Fills *theRect* with the same color used in `MakeBlack()`.

Randomization and Array Functions

SeedRandom() ∞

```
void SeedRandom(void);
```

Seeds the random number generator using the current date and time. This ensures that a unique set of numbers is pseudo-randomly generated for each experiment. Automatically called by `InitPsychRoutines()`.

RandomRange() ∞

```
short RandomRange(short myRange);
```

Returns a randomly generated short that is ≥ 0 and $< myRange$.

RandomBetween() ∞

```
short RandomBetween(short Bottom, short Top);
```

Returns a randomly generated short that is $\geq Bottom$ and $< Top$.

FloatRandom() ∞

```
double FloatRandom(double myRange);
```

Returns a randomly generated double that is ≥ 0 and $< myRange$.

RandomArray() ∞

```
void RandomArray(short *myArray, short ArrayLength);
```

Fills the array of shorts *myArray* with a series of randomly generated numbers without replacement from 0 to *ArrayLength*. *ArrayLength* is the length of the array.

```
short TrialOrder[100];
```

```
RandomArray(TrialOrder, 100);
```

OrderedArray() ∞

```
void OrderedArray(short *myArray, short ArrayLength);
```

Fills the array of shorts *myArray* with an ordered series of numbers from 0 to *ArrayLength*. *ArrayLength* is the length of the array.

```
short TrialOrder[100];
```

```
OrderedArray(TrialOrder, 100);
```

ScrambleArray() ∞

```
void ScrambleArray(short *myArray, short ArrayLength);
```

Randomly scrambles (or more accurately, shuffles) the contents of *myArray*.

ScrambleBetween() ∞

```
void ScrambleBetween(short *myArray, short Bottom, short Top);
```

Scrambles (or shuffles) the contents of only a portion of *myArray*, in the range from *Bottom* to *Top*-1.

String Functions**ScrambleString()** ∞

```
void ScrambleString(char *myString, short StringLength);
```

Scrambles (or shuffles) the contents of a string (i.e. an array of chars). `ScrambleString()` should be passed the length of the text inside the string, not the length of the array of characters. To find the length of a C string, use the function `strlen()`.

ClearString() ∞

```
void ClearString(char *String, short Length);
```

Clears the contents of an array of characters. Sets the value of all array cells to zero (NULL or '\0').

C2Pstring() ∞

```
void C2Pstring(char *PString, char *Cstring);
```

Transform the contents of a C string into a Pascal style string. See the section *Definitions: Common Variable Types: Pointers, Arrays, and Strings* for more information on C and Pascal style strings.

```
char PascalString[20], Cstring[20];
```

```
C2Pstring( PascalString, Cstring);
```

P2Cstring() ∞

```
void P2Cstring(char *CString, char *Pstring);
```

Transform the contents of a Pascal string into a C style string. See the section *Definitions: Common Variable Types: Pointers, Arrays, and Strings* for more information on C and Pascal style strings.

```
char PascalString[20], Cstring[20];
```

```
C2Pstring(Cstring, PascalString);
```

Math Functions**DegToRad()** ∞

```
double DegToRad(double myDegrees);
```

Converts *myDegrees* to radians, returning the value in radians.

RadToDeg() ∞

```
double RadToDeg(double myRadians);
```

Converts *myRadians* to degrees, returning the value in degrees.

Psychological Toolbox Objects**Tcard** 🍏

```
class Tcard {
private:
    GWorldPtr  tGWptr;           // pointer to the Tcard's GWorld
    GDHandle   tGDHand;         // handle to the Tcard's graphics device
    Boolean    defRect;         // flag tells whether the size of the GWorld has been defined
    Boolean    defWorld;        // flag tells whether Tcard.MakeWorld() has been called
public:
    Rect       tRect;           // Rect defining the size of the Tcard
    int        BeginDraw();     // prepares Tcard for drawing
    int        EndDraw();       // puts away Tcard when no longer in use
    int        Show();          // copies contents of Tcard to the current graphics device
    int        ShowToRect(Rect); // copies Tcard to a specified location
    int        ShowOff();
    int        CopyToCard(Tcard); // copies contents to another Tcard
    int        CopyToCardandRect(Tcard, Rect); // Same as CopyToCard, but
                                                // can specify ToRect.
    int        MaskToCard(Tcard, Tcard); // used for copying through a mask
    Tcard(); // constructor
    void       RectDef(Rect); // Defines the tRect of the Tcard.
    void       DefRect(short, short, short, short); // Identical to DefRect(), except
                                                // uses top, bottom, left, and right.
    int        MakeWorld(short); // Gives the Tcard a GWorld.
    int        ErrCheck(); // checks to make sure we have a rect(1) and world (2)
    MPGrafPage GiveGoodies(); // tells the secrets of the Tcard!
};
```

Psychological Toolbox Structures**MPGrafPage** 🍏

```
typedef struct MPGrafPage
{
    GWorldPtr myGWptr;
    GDHandle  myGDHand;
};
```

Structure used by Tcard.GiveGoodies() to hold the GWorldPtr and GDHandle of a Tcard.

SyncAndPollData 🍏

```
typedef struct SyncAndPollData
{
    Byte    KeyPressed;
    long    KeyTime;
};
```

Structure used by SyncAndPoll() to hold the response time and the keycode for the key that was pressed.

References

Products

Costin, D. (1993). *KeMo Reaction Time Utilities 1.5*

Metrowerks Corporation. (1995). *Metrowerk's CodeWarrior 7*

Symantec Corporation. (1992). *Think Reference 2.02*

Symantec Corporation. (1994). *Symantec C++ for Macintosh 7.0.3*

TimeVideo (Pelli, 1996)

Publications

Apple Computer Inc. (1994). *Inside Macintosh: Devices*, Addison-Wesley.

MacPsych archives, <ftp://stolaf.edu/pub/macpsych>

Mogg, K., & Bradley, B. (1995). Tachistoscopic applications of Micro Experimental Laboratory (MEL) used with IBM PC compatibles: Stimulus and response timing issues. *Behavior Research Methods, Instruments, and Computers*, *27*, 515-515.