Behavior Research Methods, Instruments, & Computers
1997, 29 (3), 313–321

RunScript: An extendable object-oriented program for computer-controlled psychology experiments

SILVIO P. EBERHARDT, MISHA NEVEROV, and OMAR HANEEF
Swarthmore College, Swarthmore, Pennsylvania

RunScript is a general-purpose authoring program that allows one to specify, in a script text file, the steps to be undertaken in the execution of an experiment. The script language supports fundamental operations, such as conditional branching to script-defined labels, manipulation of variables, sending the user messages, appending a string to a data file, and executing a subsidiary script file. A second command type allows runtime loading and execution of custom object-oriented software modules. A module's methods (or routines) are accessed from the script file simply by citing their text names, with optional variables likewise specified via a character string. Module development is simplified by requiring that new modules inherit existing code that implements protocols for passing variables and interacting with the script interpreter. These mechanisms allow the experimenter to add features of arbitrary complexity while maintaining simplicity, clarity, and consistency in the overall architecture of the software and the scripting language.

Programs that control real-time psychological experiments necessarily exhibit a tradeoff between ease of use and generality. Easy-to-use systems, with built-in functions that anticipate the user's needs, can significantly improve lab productivity (Ratcliff, 1994). However, customizing the program for nonsupported experimental paradigms and behaviors may be difficult and awkward at best and often impossible. More general programs allow a user to implement arbitrary functionality, but at the cost of extensive programming. It is not at all obvious that ease of use and generality can coexist in one software package (Dixon, 1991).

Several commercial applications have attempted to provide both generality and ease of use. A popular example is SuperLab (Cedrus Corp., Phoenix, AZ), which, rather than requiring explicit programming, guides the user gently through a series of windows to set up an experiment. Visual and auditory stimuli can be presented, and responses are subsequently collected and saved to a data file. However, the underlying scripting system does not allow conditional branching (at the time of this writing), so adaptive testing paradigms are not possible. Also, every stimulus token must be stored as a separate image or audio file. It is not possible to directly manipulate features of the underlying operating and windowing systems, and creation of studies with large numbers of stimulus tokens can be time-consuming and awkward.

This material is based on work supported by the National Science Foundation under Grant IRI-9309015. The authors gratefully acknowledge the kernel driver development efforts of Ye He and, for helpful comments on this paper, Lynne Bernstein and the reviewers. Correspondence should be addressed to S. P. Eberhardt, Department of Engineering, Swarthmore College, Swarthmore, PA 19081 (e-mail: silvio@jabberwock.swarthmore.edu).

Another popular package is MEL Professional (Psychology Software Tools, Inc., Pittsburgh, PA), which allows almost any experimental paradigm to be set up. It has a steeper learning curve than SuperLab, uses outdated software technology (it runs under PC DOS), and does require programming. As with SuperLab, facilities are available to link custom software routines with MEL.

The most general approach, but also the most time-intensive, is to develop custom software using standard software development tools. Many investigators, the present authors included, have pursued this avenue because existing products did not sufficiently meet their needs. Frequently, stimulus presentation or response collection requires the coordination of one or more custom computer peripherals—a situation that unavoidably requires custom code development and that can cause difficulties even if the custom code is linked to a commercial package. Our computer-interface studies, for example, require direct manipulation of the computer graphical interface, with custom mouse interrupts, direct manipulation of the event loop, and specification of the appearance and behavior of graphical objects, such as buttons, menus, and text. Our approach was to find the operating system that most flexibly supported manipulation of the user interface and then to develop RunScript to allow fine-grained control of underlying system resources.

Several risks are inherent in developing a full-custom application. One is that as the software "grows" over a series of experiments, the complexity of the package can increase dramatically, and it becomes difficult to track the interactions of variables and subroutines. Development of additional functions becomes increasingly difficult (Dixon, 1991), as does debugging and modifying the software. An alternative approach, creating individual soft-

ware packages for each experiment that incorporate only the necessary routines, requires maintaining multiple copies of the software, thereby making it difficult to maintain and upgrade the underlying software.

A reasonable balance between ease of use and generality would be exhibited by a program that (1) implements a basic set of commands that are frequently used across experiments, (2) allows arbitrary new functions to be added, and (3) enforces a structured mechanism for interfacing new functions that avoids runaway complexity as the program is extended. These requirements are likely best achieved by the use of object-oriented programming (OOP).

OOP requires structuring a program into semiautonomous units called *objects*, which interact via well-defined interfaces. An object consists of variables (including data structures), which are inaccessible outside of the object, and "methods," which manipulate the variables and provide object behavior. An object's methods can be accessed by other objects. Thus, a program generally consists of a hierarchically organized object structure in which a top-level controller object activates objects at the next level to carry out particular tasks, and these objects in turn access methods of objects at a lower level, and so on. Objects can also control entities external to the immediate program, such as equipment connected to the computer, icons on the screen, or operating system features.

OOP has several advantages over traditional procedural programming. First, well-crafted objects can be modified and extended without the likelihood of errors arising from interactions of object variables with code outside the object, since the internal implementation of an object is isolated from other objects by the method interface. Second, objects can be extended by adding methods without affecting preexisting methods. Third, (well-documented) objects are easy to debug, modify, and understand. Finally, many well-designed modules are sufficiently general that they can be reused in other applications at a later date. Indeed, the advanced OOP programmer usually finds that much of the tedium of programming vanishes, especially when a good object library is available, because low-level functional requirements can often be satisfied with preexisting objects rather than custom code. Software development proceeds at a much higher level of abstraction.

OOP does have a few drawbacks, beyond a minor-to-moderate level of computational overhead. The programmer must not only learn a new syntax but must significantly reorganize how software systems are conceptualized. OOP emphasis is spatial—involving interactions between static objects—rather than temporal, as is the case in procedural programming where the primary focus is on the sequences of subroutine calls. Objects do not disappear after they have been accessed; data contained within an object remain valid after the object's method returns. In fact, multiple instances of one object type, each with independent copies of variables and data, are frequently maintained in a program. For example, each icon, button, and window on an OOP-based graphical user interface (GUI) is associated with a separate object. Another drawback is that the documentation of a general object library may run to thousands of pages, thus requiring a substantial time investment for initial familiarization.

Finally, it is critical that the programmer carefully plan at the outset how a problem will be decomposed into objects. Poor initial planning often leads to extensive restructuring at a later date, whereas good designs can quickly result in well-structured programs, which are easy to understand and modify, and objects that can be reused.

With respect to real-time software for running experiments, there are several additional reasons for adopting OOP. First, some OOP languages—most notably, Smalltalk, Objective-C, and Java—have a feature called *dynamic binding* (Budd, 1991). Dynamic binding allows loading a new object into memory at runtime by specifying its text-string name (and directory location) and by subsequently accessing its methods by way of their text names. This powerful mechanism permits loading only the objects needed for a given experiment. Also, new objects can be developed without requiring changes in the script interpreter program. In contrast, OOP languages with static binding (such as C++) require the names of all objects and methods used at runtime to be specified at compile time. This means either that all objects used in any experiment must be compiled into the script interpreter program or that a different program must be compiled for each experiment.

A second reason for adopting OOP is that objects inherit all of the methods and variables of a parent object and all predecessors back to the "root object" (from which all objects inherit). The task of creating new custom objects for real-time experiments is greatly facilitated by requiring that all loadable objects inherit from a prototype object that implements standardized interactions with the script interpreter and with other custom objects.

Finally, current GUIs consist of many interrelated components, including icons of many types, sounds, video, and text strings. Application development frameworks, such as Microsoft Foundation Classes (MFC, Microsoft Corp., Redmond, WA), Object Windows Library (OWL, Borland Corp., Scotts Valley, CA), Abstract Window Toolkit (java.awt, Sun Microsystems, Mountain View, CA), and NEXTSTEP (Apple Computer Inc., Cupertino, CA), implement each such component as one or more objects that are interrelated and controlled in a logical and consistent manner and are carefully documented. This makes it possible for custom applications to manipulate even the most subtle aspects of preexisting GUI objects and to create new objects that inherit all of the behaviors of a preexisting object.

Unfortunately, few operating systems allow the user to exploit all these OOP benefits. One system is NEXTSTEP, which incorporates the Objective-C language (an extension of C) throughout the user interface. Each element on the NEXTSTEP graphic display (e.g., buttons, windows, sliders, etc.) and each sound is implemented as an object. NEXTSTEP has been used previously for psychology experiments. The high quality of the discontinued black-box NeXT Computer's audio capabilities have

made it a platform of choice for audio synthesis and output (e.g., Ratcliff, 1994). The StatTools statistics package (Anderson, 1995) takes advantage of dynamic loading and another NEXTSTEP capability that allows an object to communicate with other applications (in this case, the Mathematica engine and a spreadsheet). Cohen, Lamoureux, and Dunphy (1991) give a good introduction to the elegant features of NEXTSTEP, and a more complete treatment is found in Garfinkel and Mahoney (1993). The NEXTSTEP operating system currently runs on a variety of hardware platforms, including 386/486/Pentium-based machines and workstations from Sun Microsystems and Hewlett-Packard. Also, the new OpenStep package (Apple Computer, Inc.) allows layering most of the features of the NEXTSTEP interface over Windows NT (Microsoft Corp.).

Java is another object-oriented language that allows dynamic binding. The full-featured Java language is significantly simpler than the Objective-C and C++ extensions to the C language, because many features of C have been omitted (Ritchey, 1995)—most notably, pointers. Java code is more portable than any other language because objects, which are compiled to a special Java machine code format, can be transferred to and executed on any system for which there is a Java machine code translator. For our purposes, however, NEXTSTEP is preferable because it allows extremely fine-grained control of GUI objects.

Our work has centered on developing a script-based software package, RunScript, that takes advantage of the features of Objective-C and NEXTSTEP. While the goal of RunScript development was to provide a completely automatic data collection facility for behavioral experiments that compare the efficacy of different windows-based computer interfaces, it was decided that the RunScript script interpreter should remain as general purpose as possible. Experiment-specific code is added to RunScript by way of dynamically loadable modules. Each module includes a primary object that is loaded into memory when a script first references the module's name. The primary object's methods can then load, from the module, auxiliary objects and resources such as images or sounds. We have also developed a protocol for naming methods and passing variables in order to simplify the creation of new modules, to maintain consistency in the scripting language, and to simplify the interactions between each module and the script parser. We have made available "black-box" and 386/486/Pentium (NEXTSTEP) versions of RunScript that provide additional documentation and full source code for RunScript and our modules (see the uniform resource locator [URL] given in the Availability section at the end of this article).

An overview of RunScript is presented below, and features of the scripting language are described. Next, outlines of several modules that have been developed to date illustrate how an experiment can be orchestrated. Finally, we outline how we have solved the technical difficulties involved in taking response-time measurements on a multitasking operating system. Due to space considerations, we will not detail our particular experiments, which involve evaluating the relative efficacy of different pointer devices by comparing task-completion times for everyday GUI tasks such as button acquisition and menu-item selection (Eberhardt, Neverov, West, & Sanders, in press).

RUNSCRIPT OVERVIEW

The RunScript program, when initially launched, waits for the user to select a script file. It then parses and executes the command strings given in the script file. Each line of this text file contains either a complete command or a comment. Lines are interpreted in order, unless execution order is explicitly changed via a goto or control loop command. A sample command file is given in Listing 1.

Two distinct command types are implemented. *Core commands* implement script functions that are of a general nature; these are executed directly by the RunScript program. *Module method requests* are routed to user-supplied modules. A module's methods are accessed from the script by citing the text names of the module and the desired method, followed by an optional text string that comprises arguments to be passed to that method. RunScript will automatically load a module into memory the first time it is referenced in a script.

The object structure of RunScript is shown in Figure 1. The MainController object performs initialization and services user-selected menu requests, such as "load script" and "quit." The ModuleController object is responsible for all interactions with user-supplied modules. It loads modules, keeps track of all modules loaded into memory, passes module method requests to the specified module, and passes back the resulting return value.

The ScriptParser object iteratively obtains a line from the script file, interprets and executes all core commands, and passes module method requests to the ModuleController. An important feature of RunScript is that a script can cause the execution of a secondary script (herein called a *subscript*) in a manner similar to a subroutine call. When subscript execution completes, control returns to the higher level script. Consequently, an arbitrary number of scripts can be open at one time. By instantiating independent copies of the ScriptParser object for each script, independent variable spaces for each script are automatically generated, greatly reducing the possibility of variable interaction errors between scripts.

Since the main task of the ScriptParser object is to execute one line at a time from the script file, it was advantageous to insulate this object from the low-level details of script file data input and output. The FileController object was created to perform file functions, such as returning the next line from the script file, resetting to the first line, returning the last line, and appending a line to the file. Since these functions must be independently performed on all open files, a separate copy of the FileController is created for each script and data file.

RunScript object interactions are as follows. When the RunScript program is first executed, the MainController object is loaded. Its initialization sequence loads and ini-

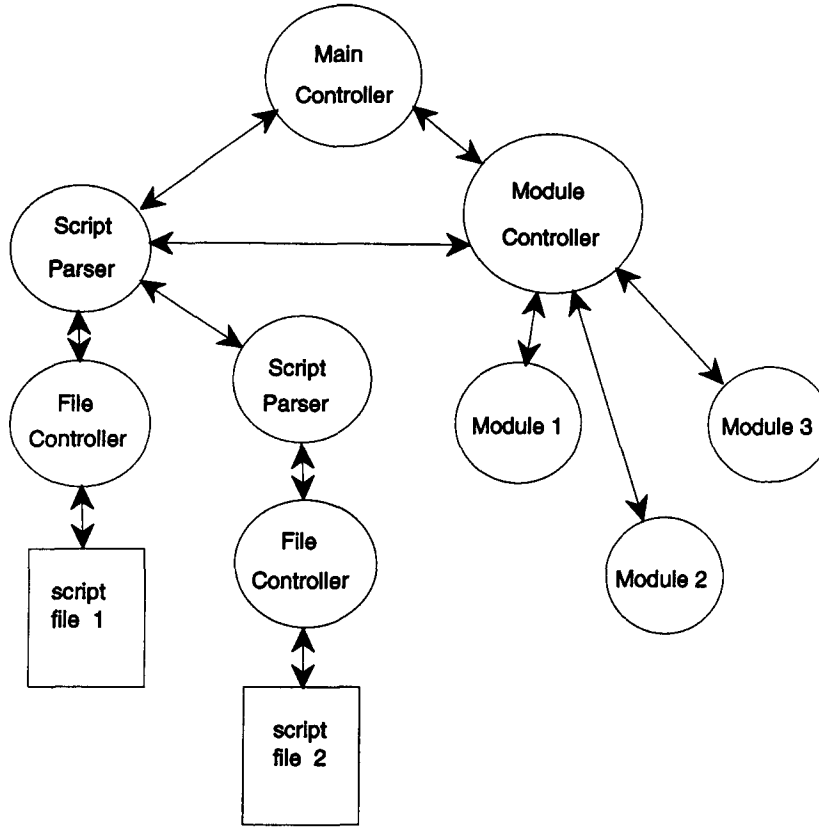

Figure 1. Object relationships of the RunScript system. The MainController object initializes the system and satisfies menu selections. The ModuleController dynamically loads script-requested modules and passes method invocation requests to the module's main object. In a way similar to subroutine execution, a script file can specify that a secondary (subscript) file be parsed. For each script file, separate instances of ScriptParser and FileController objects cooperate to sequentially read and execute the script file commands.

tializes the ModuleController and RunScript's menu bar. MainController then returns control to the operating system. When the user selects a menu item, the operating system invokes a MainController method associated with that menu item. If the menu item is "Open & Run," MainController prompts the user for the script file to be executed. If the file is valid, a ScriptParser and a FileController object are created for that file, and control is passed to the ScriptParser. ScriptParser immediately requests the first script file line from the FileController and decodes it. If the line specifies a core command, ScriptFile also executes it. If the line specifies a module method request, the whole line is passed to the ModuleController, which in turn reformats the request and passes it to the specified module (loading the module first, if necessary). Finally, the value returned by the module is passed back to the ScriptParser object, which then obtains the next script line from FileController.

RunScript COMMAND SYNTAX

Script file syntax is simple: A capital letter in the first character position of a line indicates a core command, and an "@" symbol in the first position specifies that a module method request follows. All other character types in the first position render the line a comment line. The basic syntax format for these operations is

```
CoreCommand [OptionalArgumentString]
@ModuleName methodName [OptionalArgumentString]
        this is a comment line
as is this
```

All arguments and return values (except null, indicating a fatal error) are required to be text strings. By this mechanism, ScriptParser need not determine the number and type of arguments, variable substitution is simplified, and return strings can be directly output to the user (simplifying script debugging).

Variables are identified by a positive integer that acts as variable index. (The next major version of RunScript will use arbitrary words to identify variables.) A variable may be explicitly set either by the SETVAR core command ("SETVAR 3 arbitrary string" will load the string "arbitrary string" into variable number 3) or by redirecting the return string from a core command or module method (e.g., "GETDATE > $5" will place the date and time into

variable 5). Any argument string specified in a script file may contain one or more variable identifiers, using the dollar-sign ($) notation; before executing the corresponding operation, RunScript replaces all variable identifiers with the variable contents (e.g., "MESSAGE the current time is $5" will output to the user the date and time obtained above). Listing 1 gives more examples of variable substitution. Finally, the special variable $0 is reserved for passing parameters to a subscript.

CORE COMMAND SET

The core command set, given in Table 1, currently consists of 25 commands that are implemented in less than 400 lines of Objective-C code. This set implements commands useful for running real-time experiments, although few of the commands are highly specialized.

Most of the commands are straightforward, and only a few of the more complex commands will be treated in more depth below. First, the primary conditional control-flow commands are IFGO and WHILE. IFGO has the syntax

IFGO condition label

Table 1
RunScript Core Commands

APPEND argString	Append string to data file
APPENDNL argString	Append string to data file with newline
GETDATE	Return current date and time
DEBUG	Print diagnostics as script executes
DECR var	Decrement specified (integer) variable
DOSCRIPT fileName <args>	Execute specified subscript, passing args
END	End conditional loop (while or if)
ENDFAIL	End failure code, resume normal operation
EVAL regularExpression	Evaluate a mathematical expression
GETLINE n	Get nth line from current data file
GETWORD n string	Return nth word in string
GOTO label	Unconditional jump to specified label
IFFAIL	If last method returned error, continue, else continue execution at ENDFAIL
IFGO condition label	Jump to label if relational clause is true
INCR var	Increment specified (interger) variable
LABEL labelName	Associate label with this line in script file
LOGFILE <fileName>	Specify data file to append to and read (if fileName is omitted, append to script file)
MESSAGE string	Output a message string to the user
MODULEPATH directoryPath	Set directory where modules are
QUIT	Abort RunScript altogether
RETURN returnString	Return from current script, pass returnString
SCRIPTPATH directoryPath	Set path to subscripts directory
SETVAR varString	Set variable to string
WAIT milliseconds	Delay by the specified interval
WHILE condition	Do to END while conditional clause true

Note—These commands comprise a simple scripting language that implements conditional execution and a convenient interface between module return strings, system variables, comments, and file and user input and output. All arguments, variables, and return values are character strings. Any argument string can incorporate variable identifiers; RunScript replaces the identifiers with the variable's content string. Core command return strings may be saved to a variable by the redirection (">") operator.

If the Boolean condition evaluates true, a branch to the specified label is taken. The condition is specified using the form arg1 connective arg2, with spaces separating the three parts. Arg1 and arg2 can be variables or literal strings; if numeric, the arguments are evaluated as integer or float types, and if alphabetic, as case-sensitive strings. Valid connectives are =, ==, <=, >=, !=, or <>.

The WHILE loop, using the same conditional format, has the syntax

```
WHILE <condition>
...
END
```

Currently, loops cannot be nested, except when inner loops are contained within subscripts.

The specialized control-flow command IFFAIL allows executing a block of commands if the last-invoked module method failed to execute properly, as evidenced by a NULL or "ERR" method return. Usually, a failed module method causes notification of the user; however, notification is suppressed if the IFFAIL command directly follows the failed module method request. The fail code terminates with the ENDFAIL command.

Another important function, DOSCRIPT, allows the execution of subscripts, or lower level script files. Once a lower level script file completes execution, control returns to the command following the DOSCRIPT command. DOSCRIPT takes as arguments the file name to be executed and a string that is passed to the subscript's $0 variable. The command returns the string that followed the subscript's RETURN command, or "ERR" if the subscript file was not found.

Subscripts serve the same purposes as subroutines: They allow command blocks that are frequently called to be written only once, they allow structuring the operation of a complex experiment, and, perhaps most importantly, they allow conditional execution of script files. To illustrate the last point, our current experiment setup uses a top-level script loop that requires subjects to enter their assigned code identifier. That identifier serves as the file name of the script that controls their particular experiment procedure. Once a subject completes a session, the upper level script reinitializes its modules in preparation for the next subject.

The EVAL function evaluates the string argument as a regular mathematical expression, with integer, floating point, hexadecimal, or octal values. The return type is consistent with the types in the expression, and mixed types give a consistent result (e.g., mixing integer and float types results in a correct float). An error is returned if a variable cannot be converted to one of the supported types. Currently, the four arithmetic primitive operations are supported, and parentheses can be used to group subexpressions and to overcome the strictly left-to-right evaluation order. The EVAL function was included to support adaptive experimental paradigms (such as two-interval forced choice), where stimulus characteristics and the number of experimental trials depends on subject responses.

Listing 1
Code Segment From a Script File Illustrating How Experiment Sessions Are Automatically Executed Sequentially

```
MODULEPATH /home/silvio/Apps/CHIRA/bundles
SCRIPTPATH /home/silvio/Expts
append all data to this control/data file
LOGFILE
activate module to blank screen of operating system icons
@CoverWindow on
put last line in control/data file in variable number 1
GETLINE last > $1 > $2
get second word from line—this is index of last session
GETWORD 2 $1 > $2
increment to next session index
INCR 2
then, execute code for that session
GOTO SESSION $2
execute failure code—come here only if GOTO failed
MESSAGE Can't find session, please notify attendant!
END
********** SESSION 1 **********
LABEL SESSION 1
get date and time
GETDATE > $2
and identify beginning of session in logfile
APPENDNL +SESSION 1 $2
...
GETDATE > $2
output that we're done to control file, IDENTIFY THIS SESSION
APPENDNL -SESSION 1 $2
RETURN
********** SESSION 2 **********
LABEL SESSION 2
...
```

Note—All data are appended as text to the end of the script file, with special characters placed in a data line's first character position in order to facilitate later data processing. The top code segment determines the last session's index, increments the index, and performs a GOTO to the next session's code. The CoverWindow module overlays the screen with an opaque layer to mask out all operating system icons.

Three commands implement the mechanism for archiving to a data file comments and strings returned from commands and module methods.The command LOGFILE is used to specify the file. If no filename argument is given, the currently executing script file is used for all subsequent data access. APPEND and APPENDNL are used to append a string (without and with a newline character) to this data file.

Two additional commands allow experiments to proceed completely automatically. First, a means is necessary for dividing a script file into sessions that are sequentially executed on different days. This is achieved by the code segment in Listing 1, which we include in each subject's script file. First, RunScript determines the last session executed by (1) reading the last line of the data file, which must contain the index of the last session executed, and (2) obtaining the session index from the line. The GETLINE command (with special argument "last") retrieves the last line of the datafile, and GETWORD returns the second word of that line (which specifies the session number). Second, RunScript must be able to branch to the code specifying the next session. This is achieved by the GOTO command, which takes as argument the word SESSION followed by the substituted, incremented session index.

MODULE DEVELOPMENT PROTOCOLS

While script execution control and some user messaging functions can easily be programmed using just the core commands, any custom behavior must be added by way of runtime-loadable modules. As argued above, development of custom modules is substantially simplified by the use of OOP, not only because the internal implementation details of each module are strictly isolated from RunScript and other modules but also because a module's main object inherits from a prototype object that confers useful methods for interfacing to the RunScript application. Additionally, the establishment of method-naming and variable-passing protocols brings consistency to the RunScript scripting language and the module development endeavor.

In this section, we discuss the rules for developing new modules and present the most important features of the prototype object, which is called RSClass. RSClass methods are intended to supply commonly used functions appropriate for many modules. The currently implemented functions include sizing and moving windows and other on-screen objects (using sizes and positions in screen millimeters), setting important module parameters, and sending error messages to the user. (It is straightforward to add additional methods to RSClass, although recompilation of all modules is subsequently required.)

A potential difficulty with using OOP for experiments is that many modules can be expected to have internal variables that need to be adjusted from the script file. In our studies, for example, graphical object parameters, such as window size and location, frequently need to be adjusted from within a script. While it is certainly possible to write script-accessible module methods that explicitly set such parameters, it is more efficient to provide a general service for modifying module variables. Thus, RSClass has a built-in mechanism for allowing specially registered internal variables to be changed from the script.

Variables may be registered within module code by using a one-line method call within the initialization code of the module. (Currently, a simple method must also be written to actually affect the change.) After registration, that variable may be set, using its text name, by the script-accessible method

```
@SomeModule setVar variableName variableValue
```

Text strings and any atomic variable types (such as character, integer, or floating point) may be set in this way. While this variable-setting mechanism muddies the interface between the RunScript core and custom modules, the implementation of setVar is clean and straightforward, and the module's interface specification document makes it obvious which variables are registered.

In another attempt to enforce uniformity between modules, it is suggested that all RunScript modules implement the following methods:

init	initialization code executed only at load time
reinit	code that returns the module to its initial (default) state
suspend	removes interface features from view, removes stimuli
quit	removes interface icons and operating system connections
exe	perform the primary function of the module (execute)
free	release resources in preparation for module destruction

It is not currently possible to make more than one instance of a module, since it is important that a given module maintain its state between method invocations. If necessary, RunScript will be extended to allow copies by allocating a unique name to each copy.

MODULE DESCRIPTION

A number of modules have been developed so far, at various levels of generality. For illustrative purposes, several of the more general-purpose modules are described here in some detail. Our special-purpose modules, geared toward our GUI studies, exhibit behaviors such as drawing arbitrary configurations of buttons on the screen and highlighting one of them (to determine how fast a subject can acquire the highlighted button), and interacting with an external MC68000 microcomputer system for real-time arbitrary signal generation.

The simplest general-purpose module, Random, returns a pseudorandom floating-point number that lies between two limits given as argument. One use for this module is to specify a random stimulus onset time interval. Only one instruction is needed to generate and collect a random number in a variable:

```
select a random float between 200 and 1000 ms
@Random exe 200.0 1000.0 > $5
```

A more complex module, RandStim, randomizes stimulus order. An arbitrary number of independent factors may be specified with the method *setNumberFactors*, and the number of elements for each factor must then be programmed with *setNextFactor*. The method *randomize* tosses the die, and *nextStim* can then be iteratively used during each trial to retrieve, without replacement, a string that contains the choices (or indices) for each factor. The script can archive this string to the data file, and pass it to the module that will generate the stimulus.

Another function commonly required in an experiment is display of a text document, and (possibly) input of a text response. TextPanel allows an arbitrary text document (with interleaved images) to be displayed in a scrollable NEXTSTEP text window. A response box can be used to collect from a subject a line of text (such as the code password that activates the subject's personalized script file). The positions and sizes of the window and response box are registered variables that can be altered from the script.

The Button module can be used for presenting visual stimuli and collecting subject responses. The *newButton* method initializes a new button, takes as argument the position (in screen millimeters) and default text title, and returns a unique button identifier that can be subsequently used to change one of many attributes. Each button can also display an arbitrary image. Display of all buttons is deferred until the *exe* method is executed; this method returns the button that was pushed by subjects and the (approximate) reaction time.

Finally, modules were developed for controlling a two-button, two-lamp response box (ButtonBox) and for obscuring the screen and disabling the menu system (CoverWindow).

REACTION TIME MEASUREMENT

A common difficulty with using computers for real-time experiments is that today's multitasking operating systems allocate computer resources among many different computational processes, thereby introducing time lags in individual programs that are unpredictable and over which the user may have little control. Such time lags, which can last hundreds of milliseconds or more, make it difficult or impossible to take precisely timed measurements. While some systems, including NEXTSTEP, have built-in mechanisms to reduce such time lags, these mechanisms are difficult to control and may not completely eliminate delays or may disallow using some of the very features for which NEXTSTEP was chosen in the first place.

An alternative solution, which we have adopted on our 486 machines, is to use an internal counter/timer board (CTR-CIO5, Computer Boards, Inc., Mansfield, MA) to time events. Since we are interested in measuring the response time between when a graphical trigger occurs on the video terminal (such as a button highlight) and when the subject responds by clicking on a mouse button, we have designed a simple circuit (shown in Figure 2) for starting the stopwatch when a video event occurs and stopping it when the mouse button is pushed. While the binary "mouse-button pushed" signal was easily extracted via a mouse-cable splice, detecting a video event was more difficult. Our solution was to illuminate a small square of pixels at the screen's lower left corner in synchrony with the video trigger event. By drawing both the trigger and the square offscreen before video update, an interrupt cannot intervene between display of the two objects. The output of a photocell positioned over these pixels was routed through an analog comparator to detect onset of illumination. The resulting binary signal enabled the event timer, and the subsequent mouse click disabled the timer. A programmable-logic chip was used to implement the dig-

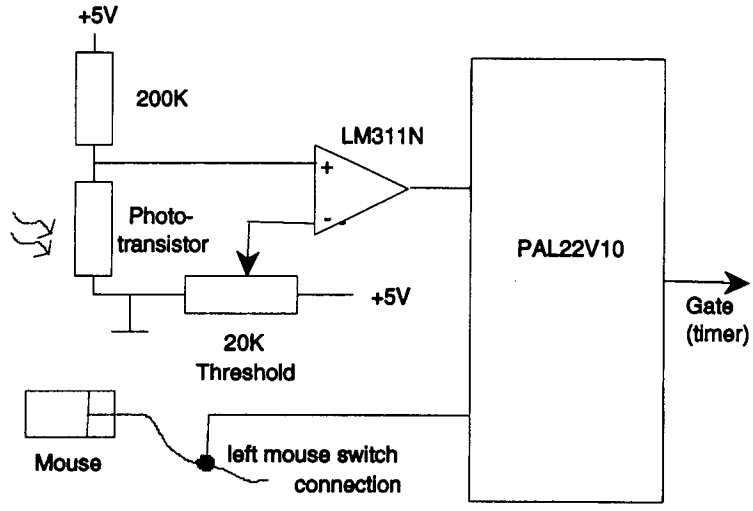

Figure 2. Reaction time measurement circuit detail. A group of pixels underlying a photocell is lit in synchrony with the graphical stimulus onset. This circuit detects the event via a comparator and a programmable-logic chip, and the derived binary trigger initiates counting on the internal CTR_CIO5 timer/counter board (Computer Boards, Inc., Mansfield, MA). Counting stops when the mouse button is pressed.

ital functions of this circuit; the ABEL code is included with the RunScript software as file timer.abl in the Timer module source.

A Timer module interfaced a CTR-CIO5 kernel driver to RunScript. Methods to control the timer include *arm* to initialize the stopwatch for a measurement, and *getTime* to retrieve the count in millisconds. The interval measurement has a temporal resolution of 1 μsec.

SUMMARY

RunScript, a platform for running real-time psychological experiments, was developed in an effort to find a compromise between simplicity and generality. Simplicity is obtained by controlling the behavior of the software via a general-purpose text-based scripting language that supports both general core commands and commands specifying the execution of custom (software) object module methods. The object-oriented nature of the interfaces to custom modules ensures simplicity. The system is very general in that modules can be developed to perform almost any computer-based task. Custom module development is simplified because new modules are built on a preexisting object framework that was designed to integrate seamlessly with the core RunScript script interpreter.

The use of script files allows experiments to be easily orchestrated and serves to document the precise sequence of events of an experiment, session by session. Indeed, as long as peripherals do not need to be switched or reconfigured between subjects, RunScript is capable of automatically collecting a day's worth of data, from the time the first subject "logs in" with a code that identifies his or her script file until the last subject of the day leaves. Documentation and subject impressions can also be automatically displayed and collected. By programming RunScript to automatically collect and record results to data files (complete with subjects' comments, date, and time), the likelihood of human error is greatly reduced.

Script files can be structured in much the same way as a high-level computer language, thereby reducing redundancy and improving readability. Blocks of code common for all subjects, as well as complex module initialization sequences, can be contained in subscript files that function similarly to subroutines. Conditional program execution based on variable values is possible, as is processing of previously collected data. Indeed, we have even used RunScript to generate stimulus files and to process data files.

The small core command set, which can be easily extended, cannot by itself run most experiments; it was designed to be able to carry out common support functions in executing an experiment. Significant effort was expended to ensure that the command set remained general and independent of particular experimental paradigms. Specific behavior is added by way of object modules loaded and accessed at run time. While we recognize that the development of new object modules is not a trivial task, this mechanism allows virtually any functionality to be added to RunScript.

So far, the philosophy underlying RunScript has proven sound. As the number of modules grows, the complexity of module interactions has not become appreciably more complex. On the contrary, efforts to keep modules and built-in commands simple have paid off well and have resulted in several modules that are useful for most experiments. Consistency within script method naming has

also simplified writing scripts and has rendered scripts easily understandable, even a year later. However, we recognize that the functionality we require of RunScript exercises only a fraction of the gamut of experimental paradigms, and we expect to extend the package as dictated by our needs and, to an extent, those of other investigators wishing to adopt RunScript.

Availability

RunScript may be obtained from http://www.engin.swarthmore.edu/~web/eberhardt/software.html.

REFERENCES

ANDERSON, D. E. (1995). Extensible programming: Beyond reusable objects. *Behavior Research Methods, Instruments, & Computers*, **27**, 131-133.

BUDD, T. (1991). *An introduction to object-oriented programming*. Reading, MA: Addison-Wesley.

COHEN, A., LAMOUREUX, M. P., & DUNPHY, D. A. (1991). NeXT in the psychology laboratory: An example of an auditory pattern tracking task. *Behavior Research Methods, Instruments, & Computers*, **23**, 523-536.

DIXON, P. (1991). The promise of object-oriented programming. *Behavior Research Methods, Instruments, & Computers*, **23**, 134-141.

EBERHARDT, S. P., NEVEROV, M., WEST, J. T., & SANDERS, C. (in press). Force reflection for WIMPS: A button acquisition experiment. In *American Society of Mechanical Engineers Winter Annual Meeting (Nov. 1997) Proceedings, Dynamic Systems and Control Volume*.

GARFINKEL, S. L., & MAHONEY, M. K. (1993). *NeXTSTEP programming: Step one, object-oriented applications*. Santa Clara, CA: TELOS.

RATCLIFF, R. (1994). Using computers in empirical and theoretical work in cognitive psychology. *Behavior Research Methods, Instruments, & Computers*, **26**, 94-106.

RITCHEY, T. (1995). *Java!* Indianapolis, IN: New Riders.

(Manuscript received January 24, 1996;
revision accepted for publication September 9, 1996.)