Annual Progress Report

January-December 1983

THEORETICAL APPROACH TO MEASURING PILOT WORKLOAD

Cooperative Agreement NCC 2-228

between

Behavioral Institute for Technology and Science, Inc.
693 North 400 West
West Lafayette, IN 47906
Barry H. Kantowitz, Principal Investigator

and

NATIONAL AERONAUTICS AND SPACE ADMINISTRATION

Ames Research Center

(NASA-CR-173297) A THEORETICAL APPROACH TO MEASURING PILOT WORKLOAD Annual Progress Report, Jan. - Dec. 1983 (Behavioral Inst. for Technology and Science, Inc.) 113 p
HC A06/MF A01 CSCL 05H G3/54
N84-17859
Unclas
18343

Annual Progress Report

January-December 1983

A THEORETICAL APPROACH TO MEASURING PILOT WORKLOAD

Cooperative Agreement NCC 2-228

between

Behavioral Institute for Technology and Science, Inc.
693 North 400 West
West Lafayette, IN 47906
Barry H. Kantowitz, Principal Investigator

and

NATIONAL AERONAUTICS AND SPACE ADMINISTRATION
Ames Research Center

This is the second progress report submitted by BITS. The NASA Technical Officer for this Cooperative Agreement is S.G. Hart, Ames Research Center, Man-Vehicle Systems Research Division.

The work accomplished during this first year of the Cooperative Agreement can be grouped into three categories. First, and most important, are theoretical advances aimed at integrating the concepts of attention and workload. Second, are empirical studies performed at Ames Research Center. Third, are systems and operational software being written in West LaFayette for the Cromemco 68000 microcomputer to allow data collection and analysis.

Theoretical Accomplishments

The jewel in the crown for this year (and next year as well) was acceptance by The Journal of Mathematical Psychology of a lengthy review article written by BITS's Principal Research Wizard. This article, entitled "Stages and channels in human information processing: A limited review," discusses the theoretical assumptions used by researchers in the area of attention, with particular emphasis upon errors and inconsistent assumptions used by some researchers. It is based upon the original grant proposal submitted to NASA (see Appendix for manuscript as accepted by JMP). The article is currently being revised, with particular emphasis upon the use of z-score transformations and their theoretical implication for POC functions in timesharing tasks; the revised article will be appended in next year's annual progress report since the requisite computer simulations to generate POC functions with given performance-resource functions will take another 3-4 months of effort.

Empirical Accomplishments

Five experiments conducted at Ames Research Center were completed and/or presented this year: two GAT experiments conducted by Michael Bortolussi and two laboratory studies and one field experiment conducted by Jan Hauser.

Experiments conducted by Michael Bortolussi

The first GAT experiment using an asynchronous secondary reaction-time task was presented at the Annual Meeting of the Human Factors Society, Norfolk, 1983 and a complete written description is contained in the proceedings of that meeting (see Appendix). This experiment was important because it demonstrated quite successfully that pilot workload could be measured in a flight simulator with an appropriate secondary task that was derived from a theoretical model of attention and timesharing.

The second GAT experiment on building levels of workload (BLOW) built on this foundation and was equally successful. (Since the PI only claims a 50% hit rate on experiments, this implies that the next two experiments won't work.) Again the asynchronous secondary task was able to discriminate the different levels of workload associated with three hierarchical levels of flight tasks: baseline tasks (e.g., fly at constant speed), paired tasks (e.g., fly at constant speed and heading), and complex tasks (e.g., fly at constant speed, heading, and altitude). In this experiment certain degrees of freedom of the GAT were frozen for the lower hierchical tasks to obtain better workload estimates of performance uncontaminated by "automatic" corrections of an irrelevant task component (e.g., correcting heading when pilots were instructed to fly at constant speed and to ignore other aspects of flight). A complete description of this experiment is contained in the Appendix. It is anticipated that this experiment will be combined with another asynchronous secondary task experiment currently in progress using the GAT and that a single write-up of both experiments will be submitted for journal publication. Furthermore, the Principal Investigator has agreed to write a chapter entitled "Mental workload" in which this research is prominently featured (see publication list following).

Experiments conducted by Jan Hauser

The results of an experiment conducted earlier in the year were presented at the Annual Conference on Manual Control, Cambridge, 1983. The experiment examined the importance of the effect of feedback on the subjective assessment of workload and performance. Strong associations were found between actual and perceived performance, and perceived workload and performance. It was concluded that the nature of the task, pursuit tracking, was in part responsible for these associations, as constant feedback was inherent to the task.

A second experiment was then conducted to further examine the effect of feedback on the perception of performance and workload. Two tasks, the Sternberg memory task and a target acquisition task modelled on the Fitts'

Law paradign, were employed. Feedback and difficulty were manipulated on both tasks. Results of this experiment were presented at the Annual Meeting of the Human Factors Society, Norfolk, 1983. Although the associations found between actual and perceived performance, and perceived workload and performance, were weaker than those reported for the previous pursuit tracking experiment, the effect of feedback produced marked differences in these associations for the two tasks. This finding supports the notion that the effect of feedback on subjective assesments is to some extent dependent on the type of task. The results of this experiment will be incorporated in a NASA technical report, currently in preparation, that will include results of a number of experiments examining the same two tasks.

A third experiment was conducted to examine the usefulness of subjective rating scales during actual flight missions. This experiment was conducted over a period of several months in the NASA C-141 airborne observatory. Data was collected for a total of 11 missions, each mission was approximately seven and a half hours in length. The flight crew were asked to make subjective ratings of Stress, Mental/Sensory Effort, Time Pressure, Fatigue, Performance, and Workload, for each of seven flight segments. They also estimated the percent of workload experienced for type of activity, Flying the aircraft and Managing the Systems, Navigation, Communication, and Change in Procedures for each segment. In addition, a physiological measure of heart rate was recorded on each mission for each crew member. Although data was collected for all three crew members, the subject pool for the position of flight engineer was very limited, thus analyses will be largely directed towards comparisons of the left and right seats. Data analyses are still incomplete, but results indicate that for both left and right seat position, the subjective assessment of stress rather than workload, is associated with heart rate. No differences have been found for ratings of workload across the two seats, but large differences were found for the type of activity. A final report on this experiment will be presented at the Annual Meeting on Manual Control, 1984.

Software Accomplishments

Progress in this area has been slow but unsteady. Since the Cromemco 68000 microcomputer is quite inexpensive, it lacks a real-time operating system. Therefore, its Cromix system must be defeated before experiments can be conducted. (Cromix is designed to keep multiple users from clobbering each other.) This is an especially difficult problem because 68000 Cromix is actually a hybrid operating system with over 80% of its code using the resident Z80 co-processor chip. (The Cromemco 68000 CPU board, called the dual-processing unit or DPU contains both Z80 and 68000 processors to enable users to keep old Z80 software running while updating to 68000 code.) In particular, the interrupt handling routines of the 68000 simply hand off to the Z80. This could be handled but there is no easy way of discovering at the time an interrupt is encountered which processor, Z80 or 68000, is then in control. (Both processors never work simultaneously.) Our temporary solution has been to disable the interrupt mechanism and to use device polling techniques instead. Since BITS has funds for only part-time programming, our progress has been slow although the purchase of a hard disk for the Cromemco has accelerated progress. A program in the C language (see Appendix for listing) has been completed and will be ready for total (Gestalt) testing by February. Allowing a generous amount of time for full debugging still permits data collection to start before summer.

Publications

Hauser, J. & Hart, S.G. 1983. The effect of feedback on subjective and objective measures of workload and performance. Proceedings of the Human Factors Society, 27, 144. (Abstract)

Hauser, J. & Hart, S.G. 1983. Subjective workload experience during pursuit tracking as a function of available information. Proceedings of the Nineteenth Annual Conference on Manual Control, Cambrdige.

Kantowitz, B.H. Stages and channels in human information processing: A limited review. Journal of Mathematical Psychology, in press.

Kantowitz, B.H., Hart, S.G., & Bortolussi, M.R. (1983) Measuring pilot workload in a moving-base simulator: I.Asynchronous secondary choice-reaction task. Proceedings of the Human Factors Society, 27, 319-322.

Kantowitz, B.H. Mental workload. In P. Hancock (Ed.), Human factors and information processing. Holland: Elsveier, in preparation.

APPENDIX

1. KANTOWITZ. STAGES AND CHANNELS IN HUMAN INFORMATION PROCESSING. JOURNAL OF MATHEMATICAL PSYCHOLOGY.

2. KANTOWITZ, HART, & BORTCLUSSI. MEASURING PILOT WORKLOAD IN A MOVING-BASE SIMULATOR: I. ASYNCHRONOUS SECONDARY CHOICE-REACTION TASK. PROCEEDINGS OF THE HUMAN FACTORS SOCIETY.

3. KANTOWITZ, HART, BORTOLUSSI, SHIVELY, & KANTOWITZ. MEASURING PILOT WORKLOAD IN A MOVING-BASE SIMULATOR: II. BUILDING LEVELS OF WORKLOAD.

4. TRICE. LISTING OF C LANGUAGE PROGRAM FOR 68000 CROMEMCO SYSTEM.

Channels and Stages in Human Information Processing: A Limited Review

This article reviews the status of the theoretical construct of capacity. Four basic questions are discussed: (1) What is capacity? (2) How is capacity measured? (3) Is capacity limited? (4) If so, where is it limited? It is claimed that empirical answers to these questions have been unsatisfactory due to theoretical and methodological issues that need be resolved. Data are presented to illustrate such difficulties. It is concluded that the construct of capacity has become more and more vacuous and that caution is required whenever capacity is invoked to explain behavior.

To appear in Journal of Mathematical Psychology
(currently being revised)

Channels and Stages in Human Information Processing: A Limited Review[1]

Barry H. Kantowitz
Purdue University

Current models of attention and information processing that invoke channels and stages as key constructs share two closely related, but, nevertheless, logically distinct, concepts. The first is the assumption that human behavior can be best understood in terms of a hypothetical informational flow inside the organism (e.g., Broadbent, 1971, p. 7); this flow cannot be directly observed but clever manipulation of experimental conditions allows us to make reasonable inferences about this postulated flow. The second assumption is that a diagram of this internal flow is not by itself sufficient to explain behavior, that is, a second concept, that of capacity, must also be included. Some portions of the information flow proceed satisfactorily without the allocation of capacity but other portions suffer when the requisite amount of capacity cannot be supplied. While the flow of information within the organism and the allocation of flow of capacity within the organism may be highly correlated, they are seldom identical.

This paper is primarily concerned with the capacity construct, although as shall be seen, this cannot be discussed without also remarking upon the flow of information. The basic issue is simply to what extent does the use of the capacity construct increase our ability to explain behavior. Alas, there is no direct answer to this question and attempts to evaluate the utility of capacity necessarily raise additional questions.

In this paper attempts to understand and/or explain the basic theoretical nature of the concept or construct of capacity lead to four basic questions:

[1] This research was supported by Cooperative Agreement NCC 2-228 from The National Aeronautics and Space Administration, Ames Research Center.

1. What is capacity?

2 How is capacity measured?

3. Is capacity limited?

4. If so where is it limited?

The reader may feel that these questions are rather straightforward requiring only a few empirical results for clarification. A major purpose of the present paper is to deny this simplistic solution. Instead it is claimed that any resolution of empirical differences in this area will first depend upon clarification and resolution of theoretical issues not amenable to direct empirical test. This point will be made salient by reference to existing articles that reach apparently conflicting conclusions about capacity and by some new experimental results that highlight the source of this difficulty.

Capacity: What Is It?

For a clearer understanding of the psychological meaning of capacity we must start with the seminal work of Broadbent (1958; 1971) who was largely responsible for making psychologists aware of the general issue of capacity as expressed in his limited-capacity channel model of performance. In reviewing the earlier formulation Broadbent (1971) states that

> ". . . it was to some extent meaningful to regard the whole nervous system as a single channel, having a limit to the rate at which it can transmit information; such a limit is usually termed a limit to 'capacity' (page 8)."

This definition is precise specifying the system in question (the whole nervous system) and that the limitation is one of information transmission rate, i.e., bits/time. Broadbent also had the grace and foresight to bracket the term capacity with quotation marks, indicating his concern for an overly literal interpretation of the concept. It is clear that for Broadbent (1958; 1971) the concept of capacity has meaning only insofar as it relates to information theory and the communication model of Shannon (1948). While the term capacity has been otherwise used (e.g., Garner, 1962) to indicate a limit in the amount of information transmitted in perceptual discrimination, such usage neglects the temporal or rate-dependent aspects of a communication channel and is eschewed here.

Later workers found the quantitative concepts of information theory less useful (Garner, 1974) or even useless for psychologists (Neisser, 1967). While a small vanguard still maintains that information theory is indeed useful (Kantowitz, 1975; Moray & Fitter, 1974), most psychologists have been disappointed with the utility of information theory (e.g., Rosch, 1974). This rejection of information theory has quite interesting implications for the capacity concept. This concept was retained and amplified even though its original information-theoretic base was cast aside. But in this process capacity became more ambiguous and amorphous. Broadbent was able to define capacity quite precisely but later workers were not since they had rejected the information theory framework which first spawned the concept in a concrete manner. Thus later workers (Kahneman, 1973; Norman & Bobrow, 1975) had to rely upon some implicit meaning or pretheoretical assumption retained from information theory.

Norman and Bobrow (1975) offered an extensive discussion based upon a dichotomous classification of process limitations into data-limited and resource-limited processes. The important concepts of a performance-response function and a dual-task operating characteristic will be discussed later. Now we focus upon their definition of a <u>resource</u>--their neologism for capacity.

> "Resources are such things as processing effort, the various forms of memory capacity, and communication channels (p. 45)."

While this definition lacks the precision of the Broadbent example, it exemplifies current usage because many operational definitions could be linked to it; indeed the following section discusses operational definitions of capacity in detail. It is clear that the concept of capacity exists on a higher theoretical level than that of operational definition. The definition above offers three major concepts--processing effort, memory capacity, communication channels--each of which can be operationally defined in several ways. Yet the definition above implies that a resource is more than any one of this large number of possible operational definitions.

Townsend (1974) deserves credit for explicitly noting that semantic connotations of the term capacity were often misleading. Any answer to the question posed in the heading of this section necessarily implies some particular system as a locus for capacity. Metonomy is a term meaning the substitution of the container for what is contained as in the sentence "The pot is boiling." Here we are faced with a psychological reverse metonomy with the contents, --capacity --being substituted for the system containing the capacity. Most authors have tacitly assumed that the system is the entire organism. Townsend (1974, Table 1) has shown this to be a gross oversimplification since any specification of capacity is meaningless without a concommitant specification of other system properties. The third major section that considers the question "Is capacity limited?" will return to this issue.

Kahneman (1973) has provided an important book entirely devoted to the topics attention and effort. Yet it is difficult to find an explicit definition of capacity, although the term appears on virtually every page. The index directs us to attention, effort and spare capacity. It is hoped[2] that the following excerpt conveys the sense of what Kahneman means by these terms:

> "These observations suggest that the completion of a mental activity requires two types of input to the corresponding structure: an information

> input specific to that structure, and a non-specific input, which may be variously labeled 'effort,' 'capacity,' or 'attention.' To explain man's limited ability to carry out multiple activities at the same time, a capacity theory assumes that the total amount of attention which can be deployed at any time is limited (page 9)."

While this definition does contain one precise clue, that capacity is non-specific, a naive reader unversed in the lore of psychology and unacquainted with the hydraulic analogy, which likens the flow of capacity to the flow of some fluid (gasoline has been suggested due to its energizing properties), would be hard put to understand exactly what capacity is from this definition.

Many other imprecise definitions similar to the two cited above can be found in the literature. These two have been singled out because of the importance of the works that contain them and are examples of the best current efforts at defining capacity. They clearly reveal some tacit pretheoretical assumptions accepted by most psychologists working with the concept of capacity. Without these pretheoretical assumptions no one could understand these definitions.

These ambiguities are amplified when the hydraulic analogy is applied to psychological systems which are often less clearly specified than are water systems. Removing the information-theoretic base of the analogy stretches its credibility even more, so that heavy emphasis need be placed upon operational definitions of psychological capacity. If these definitions provide converging operations, capacity may still prove an extremely useful concept in psychology. So we now consider the problems encountered in measuring capacity.

Capacity: How is it Measured?

Most psychologists accept the dictum that anything that exists, exists in some quantity and this quantity can be measured. If capacity is more than a diaphanous analogy, amounts of capacity must be precisely measured. Psychologists have relied upon two measurement techniques to establish capacity requirements of different tasks. In single-stimulation paradigms the human must perform one task that has several levels of difficulties. Changes in capacity demands are inferred either indirectly from changes in the time needed to perform the task or directly from changes in working rate (Broadbent, 1965). In double-stimulation paradigms humans must perform two separate tasks simultaneously. Changes in capacity demands are inferred from observing decrements in secondary task or both secondary and primary task performance.

Single-Response Measures of Capacity

The most common measure of capacity has been lag (Broadbent, 1965) or reaction time. This is the time between the insertion of a signal into a system such as the human and its eventual emergence from the system in the form of a response. The key assumption is that tasks requiring greater capacity will traverse the system more slowly than tasks requiring lesser capacity. It is now well known that reaction time is a linear function of the information present in a set of alternatives (Hick, 1952; Hyman, 1955; Briggs, 1974). The reciprocal of the slope of this function has often been interpreted as measuring the channel capacity of the human (Welford, 1960). Broadbent (1971) has noted that this is not a satisfactory measure of channel capacity since restrictions upon input or output ends, such as altering S-R compatibility, affect the slope. Indeed with highly practiced subjects and extremely compatible S-R relationships the slope approaches zero implying that capacity approaches infinity. Thomas (1974) has noted that slope measures of capacity derived from speed-accuracy operating characteristics may be valid provided certain restrictions are met; this offers theoretical advantages lacking when the Hick's law slope is used to index capacity. Since a speed-accuracy operating characteristic relates two dependent variables, this procedure is quite different from using the slope of a Hick's law function.

It is not too astonishing that an indirect measure such as lag is inadequate as an index of capacity. However, a more direct measure, working rate (information/time) appears to be limited not by rate of information but rather by rate of responding (Broadbent, 1971). These factors led Broadbent (1971) to the following conclusion:

> "All these points make somewhat doubtful of the value of the original analogy with the speed at which messages could be perfectly encoded in our limited system (page 282)."

Such warnings, however, have not yet done much to dampen enthusiasm for the practice of inferring capacity from measures of reaction time. For example, Treisman, Squire and Green (1974) have interpreted earlier RT data (Treisman & Fearnley, 1971) to suggest that perceptual processing capacity is limited. (The meaning of a limited capacity system will be deferred until the following major section; the present example is cited only to demonstrate that RT is still used as a measure of capacity.) Treisman and Fearnley recorded RT in a digit classification task involving either one or two target items. Reaction time to pairs was slower than to single items and this increase in lag was taken by Treisman _et al_ (1974) as supporting a limit on capacity. A similar position was taken by Ninio and Kahneman (1974) who measured RT in dichotic listening when only a single response was required on each trial. Increased lag in a divided-attention condition was attributed to the greater capacity demands of divided- versus focused attention. Since change in RT does not always imply an impact on capacity, this interpretation, while not necessarily incorrect, should be accepted only with great caution. Ideally, lag should not be used as an index of capacity without strong converging observations.

While the two examples of lag given above are similar to those discussed by Broadbent, a much more sophisticated use of RT measures of capacity is illustrated in the work of Townsend (1974). The crucial distinction between Townsend's work and that described above lies in the detailed _a priori_ specification of system architecture that precedes Townsend's inferences about capacity.

ORIGINAL PAGE IS OF POOR QUALITY

Instead of dealing with the capacity of a relatively amorphous system equivalent to the entire human central nervous system, Townsend specified capacity at some particular level of processing, e.g., the level of an individual element where an element is defined as one of the finite inputs to a processing stage. (For now terms like element and stage will be used somewhat loosely; the last section of this paper attempts to be more precise regarding such concepts.) A specific assumption about capacity at one level usually implies strong constraints upon capacity at other levels. The level of processing is related to the number of elements with higher levels involving greater numbers of elements. Capacity is related to the rate parameters that specify the speed of processing for individual elements. Thus capacity becomes meaningful and measurable only when the size of the element is first specified. A simple example illustrates this. Suppose we wish to measure the capacity required to process the letter A in a perceptual identification stage. If the entire letter is one global element (or template) different conclusions about capacity would be drawn than if the letter consisted of three elements (or features): a horizontal line and two oblique lines.

This approach accepts the basic assumption, that operations requiring greater capacity take longer, as necessary but not as sufficient. Equal protraction of reaction times need not imply that operations require equal capacity. Models with unequal capacity parameters can easily produce equivalent reaction times by varying such structural arrangements as the level at which capacity is assigned, parallel vs. serial processing, self-terminating vs. exhaustive processing and independent vs. dependent processing (Townsend, 1974). There is no obvious and simple relationship between capacity and reaction time that holds for all systems when capacity is defined and measured in this precise mathematical manner.

Double-Response Measures of Capacity

Another paradigm that can be utilized to measure capacity requires the simultaneous performance of at least two independent tasks. Task independence implies that the qualitative nature of each of the component task remains unchanged in the dual-task environment (see also Garner & Morton, 1969, for a discussion of independence). In particular it is assumed that the component tasks are not reconstituted into some more molar single task as a result of practice and (hypothetical) operations such as stimulus and response grouping (Kahneman, 1973; Kantowitz, 1974).

The logic behind this assessment of capacity at first appears straightforward. Assume that performance on the primary task (i.e., the task arbitrarily defined by the experimenter by instructions or pay-off matrix as being most important) remains constant in both single- and double-task conditions. Then any decrement in secondary-task performance in the dual-task environment, relative to secondary-task performance by itself, can be attributed to capacity demands of the primary task provided that structural interference (Kahneman, 1973) is not the cause of the performance decrement. Structural interference occurs when contradictory or mutually impossible demands are placed upon a single processing system; again we will defer precise definition of the term "single system." This kind of structural interference is best illustrated by an example: It is impossible to simultaneously insert the index finger of your right hand into your ear and nose. The astute reader will have already noticed at least two difficulties with this means of measuring capacity: First, subjects are seldom sufficiently accommodating to equate primary task performance for single- and dual-task conditions. Second, how can one be sure that interference is not structural?

Equating Primary Task Performance. Despite pay-off matrices, instructions, temporary cessation of the experiment to excoriate subjects, good intentions and even practice, it is extremely difficult for subjects to maintain a constant level of performance on a primary task under both single- and dual-task environments. Indeed the more detail in which performance is examined (e.g., measuring speed and accuracy), the less likely that constant primary task performance is achieved. It is difficult for subjects to maintain constant performance in a single task despite great amounts of practice and explicit pay-off matrices; in research dealing with the speed-accuracy tradeoff this difficulty has been formally termed the micro-tradeoff (Thomas, 1974). In less mathematically oriented research this difficulty is usually called variability and is assumed to be part of the innate nature of (a) the human, (b) the experimenter's imprecise control of the environment, or both (a) and (b).

Since the desired condition of constant primary task performance is often approximated but seldom achieved, some specific model of primary task performance is usually required so that slight changes (relative to changes in the secondary task) in primary task performance can be equated. Some common models often used when reaction time and/or response accuracy are measures of primary task performance include the theory of signal detection and assorted models of speed-accuracy tradeoff. If the primary task involves tracking, control models are used to derive corrections. All these models introduce *post hoc* mathematical or statistical corrections after the data have been obtained. The utility of the correction depends, of course, upon the validity of the model as applied to the particular experimental situation at hand.

The recent availability of mini-computers in many psychological laboratories has given us another option in solving this problem in dual- or multi-task environments. Real-time (or on-line) computer capabilities can interactively shift primary (or secondary, or even both) task difficulty as a function of dual-task performance. Such a solution was used by Knight and Kantowitz (1974) to maintain a constant error rate in a double-stimulation speed-accuracy tradeoff paradigm. The mini-computer interactively shifted the criterion for a FAST response based upon the subject's previous behavior on immediately preceding trials.

While most experimental psychologists have been trained to prefer experimental to statistical control, the choice between these two solutions should be more than a reflex decision. With well-practiced subjects, it is likely that mathematical models such as signal detection theory fit nicely and so can be safely and usefully applied. Indeed with well-practiced subjects the interactive computer technique may backfire since subjects have ample opportunity to discover the algorithm guiding interactive changes and can use this information to defeat the purpose of the experiment. For example, in the Knight and Kantowitz (1974) experiment (which used naive unpracticed subjects), experienced subjects might have deliberately slowed their responses so that the computer would adjust the criterion for FAST responses upwards. Conversely, when naive, unpracticed subjects are used, their substantial variance often prevents adequate fitting of a precise mathematical model so that on-line techniques may be preferred. The key point is that instead of relying upon experimental designs that discourage switching between the dual tasks (Kerr, 1973), the tools are available to experimentally or statistically control and interpret slight changes in primary task performance. This seems more appropriate than restricting experimental design to those that appear to offer a sufficiently slight change in primary task performance so that this change may be ignored. Even small changes if consistent can have large implications (e.g., Pachella, 1974).

Structural interference. The introduction of this section noted that structural interference arises when contradictory demands are placed upon a single processing system. Kahneman (1973) measures structural interference by the following operations:

> "Tasks A and B are equated by difficulty or by a physiological measure of effort, when performed singly. If the combination of Task A with a new Task C is more demanding or difficult than the combination of tasks B and C, this result provides evidence for interference between A and C beyond what can be explained in terms of attention or capacity. The alternative interpretation, that Tasks B and C are mutually facilitating, also assumes a structural interaction (page 196)."

According to this operational definition, structural interference must be removed before capacity can be measured, i.e., structural interference is a confounding factor insofar as concern is upon capacity measurement.

Such a position is eminently reasonable when by structural interference is meant a limitation imposed by the physical arrangement of an effector or affector unit; this more specific type of structural interference will be termed "physical interference" in the present article. Examples of physical interference would be the eye's inability to image two stimuli upon the fovea if the stimuli are sufficiently separated and the hand's inability to simultaneously push a lever up and also push it down. Physical interference is a rather uninteresting phenomenon and any capacity-like effects that can be attributed to it are trivial and appropriately regarded as confounding factors. Kahneman's definition of structural interference goes much beyond physical interference and may prove too severe a restriction upon capacity measurement, i.e., such a broad definition may throw out the baby and the bath water.

For the moment let us assume that Tasks A and B can be successfully equated and that related difficulties discussed earlier have been surmounted or at least finessed. Let Task A be pushing a lever UP with the right hand and Task B be pushing the lever DOWN, also with the right hand. Further assume that both these tasks require equal reaction times and that this finding satisfies us that they are of equal difficulty. Now let Task C be pushing another lever UP with the left hand. It is more difficult to combine Tasks B and C than to combine Tasks A and C when reaction time is taken a measure of task difficulty (Way & Gottsdanker, 1968). So this combination of tasks satisfied the definition and demonstrates structural interference above and beyond capacity interference, i.e., the experiment does not prove that capacity limitations account for the relative inability of subjects to combine Tasks B and C. Now let us change the situation slightly by presenting the stimuli of Task C but no longer requiring an overt response. It is still more difficult to combine "non-task" C with Task B than with Task A (Kantowitz, 1973). In this latter situation it becomes considerably more difficult to dismiss the findings as an unimportant demonstration of structural interference, even though the operational definition of structural interference is satisfied. One cannot help but ask what "structures" are interfering with each other. In the Way and Gottsdanker (1968) study, one might be willing to state that some amorphous and generally unspecified "response structure" accounts for the structural interference. But when no response is required to Task C, the "structural" aspects of the situation are diminished to so great an extent that the operational definition need be questioned. Such a broad operationalization of structural interference precludes by definition any capacity explanation of a wide variety of interesting non-trivial findings. Again the issue reduces to the particular system involved in the interference. If the system is limited to observable effector and affector units, the definition shrinks down to physical interference. If, however, the system also includes any and all inferred processing stages, the definition expands to encompass and eliminate many interesting task combinations.

As if this difficulty were not sufficient by itself to eliminate a broadly defined structural interference as a confounding in the measurement of capacity, there is yet another perhaps even more serious difficulty with the operational definition cited above. Again let us ignore the practical difficulties encountered when trying to compare dual-task performance of different component tasks. The operational definition above strongly implies a reflexivity in dual-task conditions, i.e., if Task A interferes with Task C then Task C will also interfere with Task A. (While this reflexivity assumption is mentioned here, it has been a tacit assumption in many studies of dual-task performance.) As Norman and Bobrow (1975) have clearly shown, this assumption depends upon the location of both component tasks on their performance operating characteristics. Tasks A and B may be equal in difficulty as measured by some particular dependent variable or physiological concommitant of behavior and yet occupy quite different locations on a performance operating characteristic. The addition of Task C will then exhibit different effects despite the attempt at equating Tasks A and B.

These considerations argue that broadly defined structural interference forces capacity measurement into a Procrustean bed when structural interference is regarded solely as a confounding factor. The question, raised by Kerr (1973), as to when the limits of structural interference have been reached cannot be satisfactorily answered when structural interference is broadly defined as by Kahneman. The empirical solutions offered by Kerr regarding preferred paradigms do not reach the theoretical issue raised by this operational definition. The theoretical solution proposed here would limit structural interference to only physical interference, i.e., only physical interference would be regarded as an uninteresting confounding factor. Specification of the system being studied would determine capacity considerations regardless of structural interference as defined by Kahneman.

Grouping. The most basic assumption in time-sharing paradigms is that the two (or more) component tasks remain independent when performed in concert. Measuring the capacity demanded by a primary task by changes in secondary task performance implies that two tasks remain separate even though being performed simultaneously. This assumption is not easily tested empirically. The solution of requiring a third simultaneous task only pushes the problem back a level by offering more ways (3 to be exact) for two tasks to be combined to say nothing of all three tasks merging. There are at least two ways in which this independence assumption can be violated. First, the two tasks may be reconstituted into a more molar single task through stimulus or response grouping (Kahneman, 1973). Second, the two tasks may not be grouped but may nevertheless become time- or phase-locked so the elements of one component task cannot be completed until a "synchronization pulse" arrives.

The concept of stimulus grouping was used by Welford (1952) to explain findings in the psychological refractory period effect when the interval separating two successive stimuli was short. Welford believed that processing of the first stimulus was delayed so that both stimuli could be processed jointly. Later workers extended this idea to allow for the possibility of response grouping (Borger, 1963) where the first of two responses was delayed. Grouping is still a widely used explanatory device in the area of double stimulation, although it is often difficult to distinguish between stimulus and response grouping (Kantowitz, 1974).

It is difficult to find a behavioral index that will clearly identify examples of grouping. One possibility that can be used when both component tasks are discrete is the inter-response interval (IRI). If IRI is constant then grouping can be inferred. Since the temporal relation between both task components is unchanged, this may be taken as evidence that the two tasks have been merged into a single molar task. Kahneman (1973, Chapter 9) uses this approach to discredit conclusions contrary to his variable-capacity model reached by Schvaneveldt (1969). The arguments for and against such an interpretation are complex and will not be summarized here; the reader is referred to the original sources and to a discussion by Kantowitz (1974). The relevant difficulty for present purposes is that the same variation in IRI was interpreted by Kahneman as supporting response grouping and by Schvaneveldt and also by Kantowitz as failing to support grouping. Furthermore, IRI may reasonably be expected to vary with task complexity so that failure to obtain a constant IRI need not necessarily rule out a grouping explanation. A constant IRI criterion seems to be a reasonable first-approximation and until a better a priori specification is offered, this seems to be the best available criterion for discrete task components. Even this conservative definition will not serve for continuous tasks like tracking. The definition and measurement of response grouping is a major unsolved problem of considerable theoretical import. Perhaps the spectral analysis techniques of electrical engineering used for pulse trains may prove helpful.

Synchronization of the two component tasks is a problem closely related to grouping. The idea that basic periodicities in information processing characterize the human is not new (Craik, 1947; 1948) and has been generally accepted (Fitts, 1964). The human can be viewed as a sampled-data control system with periodic interruptions of feedback sampling. Studies of attention have attempted to relate this periodicity to some physiological concommitant such as the alpha rhythm (e.g., Kristofferson, 1967) or other evoked potentials (Posner, 1975). However, this general type of time-locked behavior does not necessarily influence the independence of the two component tasks in the same way as grouping violates the independence assumption. A more specific type of synchronization can occur when one component task is experimenter-paced and the other is self-paced. The self-paced task may fall into a fixed phase relation with the experimenter-paced task. A similar problem can arise when both component tasks are self-paced; in this case one task (usually the primary task) may temporally dominate the other which must fall into phase. While the spectral analysis techniques of engineering should prove useful in locating this kind of failure of the independence assumption, little work has been accomplished by psychologists working with double-response paradigms. Even having both component tasks paced by the experimenter may not avoid the issue since subjects do not always follow pacing requirements exactly; thus, again one task may become temporally subsidiary and phase-linked to the other task.

The related problems of grouping and synchronization present serious theoretical difficulties in the analysis of dual-task paradigms. These problems have been largely ignored in this context but since they bear directly upon a basic assumption of time-sharing logic, the independence assumption, solutions must be attempted.

Secondary Task Variables. In selecting a secondary task the first decision an experimenter need make is between continuous and discrete tasks. Kerr (1973) has argued that the discrete secondary task is preferable because (a) subjects are more likely to divert capacity from the primary task when faced with a continuous secondary task, and (b) increases (Welford, 1968) or decreases (Kahneman, 1973) in overall capacity due to insertion of a secondary task are less likely with a discrete task occurring on only a proportion of all trials. Although no evidence is given to bolster these contentions, they certainly are reasonable assertions that merit additional discussion. A possible rationale behind (a) and (b) above seems to be related to the duration or relative proportion of a trial taken up by the secondary task. Hence, a continuous secondary task must be performed for a longer time than a discrete task and so is more likely to divert capacity away from the primary task. Similarly, insertion of a discrete secondary task, especially if only on a proportion of trials rather than on all trials, is less likely to tamper with capacity than a continuous task present on every trial since the discrete task is present for a much smaller time. Unhappily this kind of rationale confounds definitions of capacity based upon time versus those based upon interference, or in Kerr's terminology time versus space. Allowing the duration of the secondary task to effect capacity removed from the primary task, illustrates this confounding. Such usage ignores the distinction between lag and working rate discussed earlier. While space and time are intended as independent theoretical concepts, any attempts at measuring space often involve coincident measure of time. In this paper I have tried to carefully distinguish between theoretical definitions (preceding section) of capacity and measurement of capacity. It is possible for theoretical concepts to have great value even if they cannot be directly measured. An example would be any mathematical model of behavior where parameters must be estimated all at once. Merely having data does not allow estimation of the particular parameter of theoretical interest without invoking the entire theoretical model. However, with amorphous concepts like capacity and space, measurement becomes crucial. The theoretical virtue of space is largely illusory when measurement is ignored. Thus, on a theoretical basis these reasons for preferring discrete secondary tasks are not entirely convincing. However, this preference may be valid on an empirical basis if data were obtained to demonstrate that primary task performance was more likely to remain unchanged when combined with a discrete secondary task. The difficulty in accomplishing this empirical justification lies in equating discrete and continuous secondary tasks in terms of capacity requirements. Any attempts to determine capacity demands of a (secondary) task take us immediately back to theoretical issues concerning the definition and measurement of capacity.

Arguments can also be made supporting the use of continuous secondary tasks. The modal argument states that any momentary diversion of capacity from the primary task may go undetected if no secondary task must be performed at that moment. This argument occurs most frequently with paradigms such as a dichotic listening where secondary task stimulation may be present without a requirement for momentary responses to the secondary task. Proponents of discrete secondary tasks in time-sharing paradigms could counter this argument by claiming that capacity would not be momentarily diverted from the primary task until occurrence of the discrete secondary task, i.e., there is no need to divert capacity when no secondary task is present. A better argument for continuous tasks might be that the sudden onset of an unpredictable discrete secondary task creates a momentary disruption that artificially increases the capacity demands of the secondary task. In engineering terminology, this would be equivalent to adding an impulse function to perturb a system. Impulse functions severely tax systems often resulting in distorted transient responses. This possibility could be evaluated by a parametric manipulation of secondary task characteristics; however, current research has emphasized manipulation of primary task parameters with little attention paid to systematic study of secondary task characteristics. It should be noted that problems in deciding between discrete and continuous secondary tasks can be adroitly evaded by requiring simultaneous responses to a complex stimulus (e.g., Schvaneveldt, 1969). This paradigm is methodologically equivalent to a psychological refractory period paradigm with a zero inter-stimulus interval. But, as has been previously mentioned, grouping problems may become more difficult to evaluate with this zero ISI technique. My own preference is for a continuous secondary task since a constant load diminishes expectancy effects and encourages subjects to maintain a steady-state strategy.[3]

Overlap between primary and secondary tasks. There are always two kinds of possible overlap, time and capacity. While the logical distinction between the two is simple and clear, discussions based upon measurement of the two overlaps can be obscured when time is measured so as to make inferences about capacity.

Overlap in time is a consequence most often of the operational requirements of two tasks. Time overlap acquires theoretical importance only when concern is focused upon the hypothetical processing stages inferred from performance so that serial vs. parallel processing and related issues are raised. Since a detailed discussion of stages is reserved for a later section, now only molar implications of time overlap will be mentioned. The clearest example of manipulation of time overlap is the psychological refractory period or double-stimulation paradigm in which two successive stimuli are separated by a brief temporal interval called the ISI. No processing of the second signal can occur until that signal has been presented. But this separation in time need not necessarily imply a complete separation in capacity. To make this point salient, a distinction between momentary capacity demands and expectancy or capacity allocation prior to stimulus onset must be offered. The second stimulus in a double-stimulation paradigm makes no momentary capacity demands until after the ISI, i.e., until it occurs. But it is certainly possible that some of the total system capacity is held in reserve or allocated for subsequent processing of the second signal. (This, of course, is not the only theoretical model possible since all capacity may be devoted to the first signal until second signal occurrence. In such a model there are only momentary capacity demands.) This allocation decision occurs prior to a trial and results from instructions, prior experience with the task, etc. To the extent that such a priori capacity allocation is made, the second signal may share a capacity overlap with the first signal, even though the second signal has yet to physically occur.

A more interesting form of capacity overlap occurs when both component tasks compete for momentary capacity. This implies a time overlap between the tasks. But although interest may center about measurement of momentary capacity requirements, attempts at empirically obtaining such measures often measure an amalgam of momentary and a priori allocation capacities. Although a clear theoretical distinction can be made between the two types of capacity, this distinction does not always appear in the operations used to measure capacity.

In particular operations that might be expected to effect a priori allocation, e.g., the proportion of trials on which a probe secondary task is inserted, relative pay-offs for the two tasks, etc., do not necessarily lead to conclusions about momentary capacity demands of a primary task. Furthermore, holding such an operation constant in the course of an experiment does not guarantee that conclusions about momentary capacity are justified. For example, a situation that greatly stressed the importance of the primary task by either pay-off or a low probability of a secondary task, might lead to an erroneous conclusion that the secondary task had no momentary capacity demand. Unless the experimenter had the foresight to include a single-stimulation secondary-task-only control condition, this state of affairs could go unnoticed.

Is Capacity Limited?

By now it should be quite clear that the answer to the above question is rather intimately related to definition and measurement of capacity. An algorithmic solution of attempting to answer the question for all combinations of definitions and types of measurement previously discussed would prove unduly lengthy. Instead the heuristic solution of applying the question mainly in the context of established models of capacity will be followed. This more limited review is not intended to minimize the points about definition and measurement previously made.

Channels in Information Processing Research

In psychological research, the existence of a channel can only be inferred from some measurement of capacity. Strictly speaking, the concept is then redundant and perhaps even unnecessary. I believe this insight is responsible for Kahneman's (1973) variable-allocation model of capacity in which any channel or locus of capacity limitation is vehemently denied. Capacity is the more important concept, since while capacity can exist without precise specification of a channel, a channel, at least in psychological research, can be inferred only from a measurement of capacity. Of course, a channel can be defined as, say, the entire central nervous system, but this definition increases our vocabulary rather than our understanding.

Nevertheless, the term "channel" while not strictly necessary does offer a certain convenience and has been often used. Provided one remembers that a channel is but a fiction inferred from an analogy (capacity) the term may prove helpful for some. In particular, so long as only a single molar channel is involved our habits of speech may not lead to undue confusion. But as soon as more than one channel is invoked, e.g., several parallel channels or independent channels, more care need be exercised with the concept. So long as channel means single-channel (Welford, 1952), a stage of information processing is clearly a smaller unit than a channel. Once channel means multi-channel, the distinction between a stage and a channel becomes more uncertain. In this paper by stage is meant a smaller unit of information processing and this definition will be expanded in the following major section.

The Limited-Capacity Channel

While many psychologists were influenced by communication theory and the single-channel concept (e.g., Munson & Karlin, 1954; Welford, 1952) the idea was brought to fruition in Broadbent's (1958) classic text, Perception and Communication. This model is so well-known and has engendered so much research that any complete review would require a tome of monumental proportions; indeed Broadbent (1971) found it necessary to omit or abbreviate some areas covered in detail in his earlier book. I will not illustrate predictions of the limited-capacity model since they are so well known (see Kantowitz, 1974, for such discussion). The basic prediction of the model is an interaction with task difficulty, and this interaction can take at least two distinct forms so that two related but operationally distinct meanings can be given to capacity limitation (Kantowitz & Knight, 1976, 1978b).

Variable-Allocation Models of Capacity

The finding that relative performance on two simultaneous tasks can be manipulated so that one task is performed better than the other regardless of the relative difficulties of the two tasks has caused many psychologists to infer that capacity can be not only divided or shared between two tasks but can also be allocated. Allocation is most often used to mean a deliberate or intentional assignment of capacity to a particular task component. Thus, in a dichotic listening task, for example, capacity can be allocated to the left ear but not to the right. Allocation can also be used to indicate assignment of capacity to some particular feature of a task. For example, in a dichotic listening task capacity might be said to be allocated to recognizing digits or letters of the alphabet regardless of the ear in which they appear. This feature-related use of allocation will not be used here, unless some specific stage (e.g., digit identification) is also postulated to receive this allocation of capacity. Allocation policies proposed range from several independent channels (Allport, et al. 1972) implying no total limitation in capacity to variable allocation models in which the "width" of the channel changes with task demands (Moray, 1967). The variable-allocation model proposed by Moray and by Triggs (1968) has been considerably expanded and generalized by Kahneman (1973) so that future discussion will be based upon this particular variable-allocation model.

There are two major distinctions which separate Kahneman's variable-allocation model from the limited-channel model. First, total amount of capacity expands with increasing demand, although at a slower rate. Second, allocation of capacity between competing demands is explicitly discussed, although allocation policies are very flexible. A third distinction is claimed in that no bottleneck is specified. However, proponents of limited-channel models have wavered on the locus of capacity limitation, usually stating that the limitation was in the channel itself. Since no one knows exactly where the channel can be found, in practice both limited- and variable-allocation models are vague about specifying where capacity is limited. While limited channel proponents have been embarrassed by relocation of this locus from earlier to later in the channel (e.g., Smith, 1967), proponents of the variable-allocation model have attempted to make a virtue of this ambiguity by vehemently denying the existence of any bottleneck. The issue of locating capacity limitations will be returned to in the following major section of this paper.

Although capacity is not fixed as in the limited-channel model, tasks still compete for a limited pool of capacity in Kahneman's variable-allocation model. The size of this pool is determined by task characteristics but once these are fixed, then so is total available capacity. So in any given situation capacity is still finite. While speaking of limited capacity in reference to the variable-allocation model may at first appear confusing, it is important to realize that this model also claims that capacity is fixed or limited. If capacity were unlimited, there would be no need for any allocation policies.

The allocation policy is potentially a great strength in the variable-allocation model. The concept is not well developed and except for the statement that tasks demanding high levels of attention or capacity tend to resist allocation to other tasks, little is stated so that a priori predictions can be made about allocation policy before data is collected. Kahneman (personal communication) has replied that what is needed is empirical knowledge and I do not dispute the benefits of more data. However, I doubt that rules of allocation policy can be found only by gathering more data without additional theoretical effort. For example, one can sweep out a POC function (see next section) by varying the payoff associated with each of two simultaneous tasks but will this tell us a great deal about allocation policies for other kinds of tasks? Without some theoretical statement linking task taxonomies to capacity, allocation policies can never be determined empirically without testing all possible task combinations. Kahneman does distinguish between two kinds of demand: demand_1 is a necessary condition

ORIGINAL PAGE IS OF POOR QUALITY

without which a task cannot be successfully performed, and demand$_2$ corresponds to requiring capacity in the sense that a mother demands$_2$ that her child pick up scattered toys. The relationship between allocation policy and both demand$_1$ and demand$_2$ is more implied than explicit. Presumably voluntary allocation due to instructions and pay-off controls demand$_2$. It is not clear if demand$_1$ automatically gets priority in allocation.[4] A clearer statement of how capacity gets allocated is needed. A step in this direction has been taken by Norman and Bobrow (1975) in their discussion of performance operating characteristics, the next topic to be considered.

Performance Operating Characteristics

The concept of an operating characteristic is well known in statistics (e.g., Stilson, 1966). Experimental psychologists are most familiar with a type of operating characteristic associated with the theory of signal detection: the receiver operating characteristic. Recently Norman and Bobrow (1975) suggested that plotting performance on one component of a timesharing task as a function of performance of the other component would be a useful way to portray these kinds of data. Such a function was termed a performance operating characteristic (POC).

The POC is merely another way of displaying data and as such has no more special implications for assessing capacity limitations than does, say, a bar graph. However, Norman and Bobrow place a special interpretation upon the POC especially when horizontal or vertical line segments best describe the data. Such segments are said to result from a data-limited process such that no improvement in performance could be expected regardless of additional allocation of capacity. Hence performance in these data-limited regions of the POC does not depend upon capacity limitations. By assuming that resources are allocated from one task to the other the POC function can be used to infer capacity demands of the component tasks. The basic assumption, termed the principal of complimentarity by Norman and Bobrow, is that a constant amount of capacity is always divided between the two component tasks so that the sum of capacities allocated to each remains fixed.

While this allocation rule has the great virtue of simplicity and clarity, it directly contradicts the variable-allocation model of Kahneman previously discussed. Kahneman's model claims that capacity expands with increasing task demands. Thus, Kahneman's interpretation of a POC function would differ greatly from that of Norman and Bobrow. What kinds of experimental outcomes, if any, might distinguish between these two interpretations?

A POC function that was montonically increasing over part of its range would reject both models. For Norman and Bobrow this would imply that the sum of resources increased rather than remained constant. At first it might appear that this outcome could be consistent with Kahneman's model since increasing capacity is postulated. However, for Kahneman capacity increases at a decreasing rate so that spare capacity always decreases.

A POC function consisting of discontinuous parallel horizontal (or vertical) line segments would at first appear to reject Norman and Bobrow's model. However, they explicitly allow for cases where increasing resources over a limited region need not result in a performance increment. Kantowitz and Knight (1976b) have discussed how this feature of the model makes it difficult to distinguish true data-limited processes from "step-limited" processes. However, a step POC function would not reject the Norman-Bobrow conceptualization. Since Kahneman posits smoothly accelerating functions with no step discontinuities in capacity allocation, this outcome would be inconsistent with his model.

Monotonically decreasing POC functions would be predicted by both models. Indeed most empirical studies would be expected to fall within this category. Thus, it appears that despite a crucial difference in assumptions concerning capacity allocation, the models can be readily distinguished in the POC space only because one of them posits smooth functions. Allowing discontinuities in allocation would not represent a major alteration in Kahneman's model and then the models could not be distinguished on the basis of operating characteristics. So far the theoretical discussion of POC space has ignored the kinds of empirical complications (changing error rates, etc.) mentioned earlier. When these additional difficulties are also taken into consideration it is not clear that strong theoretical conclusions may be inferred from empirical POC functions. In particular POC functions do not prove that capacity is limited; instead their interpretation is contingent upon some particular assumption about capacity limitation.

Summary

This section has discussed abstract limitations upon capacity without efforts at localizing such limitations. Of the three classes of models discussed only the early versions of the limited-channel model attempted to specify a locus of limitation and this attempt was not successful. As more and more data were gathered to test the model the locus of limitation got pushed back from some kind of stimulus filter or analyzer and retreated further into the depths of the organism (Kantowitz, 1974). The remaining two models deliberately do not try to specify any particular locus of limitation. While this has been claimed as an advantage of these newer models, skeptics could argue that upon close examination this advantage is based mostly upon ambiguity.

If the preceding analysis is correct, it appears that discussions of capacity limitations without serious concommitant efforts to specify precise loci for such limitations are of only limited value. This is unfortunate since it implies that very broad and general conceptualizations must give way to more specific formulations thus increasing the danger that models will be partitioned in such a way as to minimize predictions about common content leaving theoreticians with the considerable task of putting Humpty Dumpty together. Nevertheless, it seems likely that viable attempts based upon capacity as a major theoretical construct will first have to specify some details of the system architecture. The following section discusses such attempts with particular reference to stage models of information processing and capacity.

Where Is Capacity Limited?

The most common attempts at specifying of system architecture have been stage models of information processing where successive black boxes represent loci for transformations of information flow through the organism. Earlier stage models (Broadbent, 1958; Smith, 1968; Sternberg, 1969) concentrated upon serial processing stages where one transformation must be fully completed before the next stage can begin operation, but later theorists (Townsend, 1974; Taylor, 1976) have stressed the importance of parallel and hybrid configurations of stages. Nevertheless, proponents of both serial and parallel models have much in common when their views are contrasted with those of an earlier generation of experimental psychologists who represented the entire sequence of internal processing stages as a hyphen linking stimulus with response.

What Is a Stage?

Early proponents of stage analysis of reaction time were deliberately vague about the definition of a stage (Smith, 1968; Sternberg, 1969). Indeed, Sternberg explicitly refused to precisely define a stage and even violated his own informal definitions in one case by allowing an independent variable to influence the output, as well as the duration, of a stage. This relaxed approach was certainly defensible, and may be even wise, at the initial onset of this new theoretical tool, and in no way diminished the importance of stage analysis Indeed, more recent theoretical efforts at defining stages (Townsend, 1974; Taylor, 1976) have tried for greater precision but have achieved it only at the cost of a smaller unit--the element--that is ambiguously defined.

An element has been defined as the smallest unit of information processing. However, its physical correlate in the external world can often be observed without the aid of an electron microscope and may range from something relatively large like an entire word to something smaller like one stroke of a letter. Since the thrust of this paper is on capacity and not stages *per se*, I shall not pursue this point and direct the reader to discussions by Townsend (1974) and Taylor (1976). A stage is defined in terms of processing on the elements. For example, Townsend (p. 142) states: "stage *i* is the state of the system from the completion of the $(i-1)^{th}$ element to the completion of the i^{th} element." Taylor (1976) never even offers an explicit definition of a stage and is content to deal with definitions of stage time again based upon processing of elements. The reader who is accustomed to seeing flow diagrams with stages neatly labeled (e.g., memory-scan mechanism, push-down stack selector, stimulus normalizer, etc.) may be surprised to realize that the definition of a stage (when indeed it is offered) has little to do with such fanciful names. These labels are surplus values added by experimenters seeking to improve the generality of their models. The aptness and utility of such labels remain an empirical issue rather than a theoretical one, and so will not be pursued here (but see Kantowitz & Knight, 1978a, for one example of a more modest approach to labeling stages).

Defining a stage only scratches the surface of this theoretical concept since stages (or more precisely stage times) can never be measured in isolation, removed from possible influences of other stages. For example, how is the output of a stage communicated to the next stage? Taylor (1976) has noted that while Sternberg's original analysis called for instantaneous transmission (or at least a very brief impulse), logical possibilities range from this extreme to a very slow dribbling out of information as a stage commences to operate which is then speeded up as processing continues. Kantowitz (1969) called this first case all-or-none information exchange and the second case incremental exchange. Either assumption is viable, and Kantowitz's decision to prefer an incremental model was based upon ancillary considerations.

The purpose of this discussion has been to remind the reader that defining a stage requires more than drawing a rectangle and inserting an auspicious label. In view of recent summaries by Townsend and by Taylor a longer discussion is unnecessary. So the relationships between stage analysis and capacity can now be treated.

Capacity and Stages

The simplest and most straightforward assumption relating stages and capacity was made by Sternberg (1969): each stage has its own source of capacity that is independent of all other stages. The great success of the additive-factors method shows that even this simple assumption can be quite fruitful. Nevertheless, more recent efforts have examined alternative assumptions particularly about capacity allocation to elements, and thus also to stages. These efforts are more in tune with the present implied argument that capacity, rather than time, is the more useful analogy. Even so, this should not be interpreted as a blanket rejection of Sternberg's independent capacity stage assumption as being inconsistent with limited-capacity effects reviewed in preceding sections. If an incremental information exchange is assumed, even unlimited capacity stages can show limited-capacity effects in combination when the rate of exchange is limited. This is an extremely complex issue, mathematically as well as psychologically, and I am reluctant to state categorically that even Sternberg's original formulation with all-or-none exchange and independent stage capacities cannot show limited-channel effects.

Taylor (1976) considers the crucial distinction among stage models to lie along the dimensions of serial-parallel and exhaustive-self-terminating processing. Capacity, while certainly discussed, comes a poor third. This review shares the position of Townsend (1974, p. 135) and regards capacity as a key dimension.

Both Townsend and Taylor have a concise answer to the question posed at the head of this section: capacity is limited at the level of the element. Researchers in the general area of attention may have some difficulty in accepting this as a solution to their problems. First, although they have been able to tolerate ambiguity in finding a molar locus of limitation or attention bottleneck, ambiguity in the definition of an element may seem less acceptable. Indeed, some might claim that a proposed solution that merely exchanges one amorphous concept (capacity) for another arcane construct (the element) is rather unsatisfactory. There are two replies to this criticism. First, the element is potentially far more visible than either capacity or even a stage of processing. Elements can be mapped to particular stimulus events such as lines, curves, letters and words to mention a few possibilities. Second, as has been explicitly noted by Taylor (1976, 183-185), models differ as to their sensitivity concerning element identification, with some being totally insensitive, some being parameter sensitive, and some being prediction sensitive. It is only for this latter case that element identification presents a serious difficulty, whereas capacity "identification" is a severe problem for all of the general models of capacity previously discussed.

Second, many would prefer an answer that localized capacity limitations within some particular stage of information processing. While it is true that most researchers are sufficiently sophisticated not to expect a very specific locus (e.g., third stage from the left) since the number of stages depends upon the number and kind of independent variables manipulated by each experimenter (see Taylor, 1976, for implications of this for research strategies), they still would like a more global limitation like stimulus stage or response stage. Indeed, the present author is guilty of this oversimplification (Kantowitz, 1974) by claiming that response processes are a more important locus of limitation than are stimulus processes. While this may be true (e.g., I still believe it), it ignores implications of capacity limitations upon elements for limitations upon stages.

This point has been stressed by Townsend (1974, Table 1) and merits further emphasis. One can first specify capacity limitations upon elements as does Townsend and then, with some additional mathematical assumptions about distribution, elegantly derive capacity of models based upon different configurations of stage properties. Of course, one may question the selection of the Poisson distribution as being more from mathematical convenience than psychological validity--has a stage that is almost finished processing completed the same information transformations (none) as a stage that just started--but it has always been difficult to specify the distribution of unobservable entities. The complementary process of specifying limitations upon stages and then looking for implications of this at the level of elements has not been seriously pursued since those who view the stage as the basic unit don't care about elements. However, the stage as a basic unit is not as meaningful as the element. This is disconcerting for those who are fond of black box diagrams yet dislike the mathematics associated with elements.

The important point of this discussion is that the system architecture must be specified in some detail before assumptions about capacity limitations in general, and their loci in particular, have any meaning. Even if all the world's a stage, this doesn't help in finding the stage where capacity is limited, if indeed such a stage exists. Stating that some stage (even a molar stage such as response processing) has limited capacity has very strong implications for the architecture of the entire information flow postulated within the organism. Unless this architecture is specified, searching for a locus of limitation is like looking for the end of a rainbow: fun but not illuminating. There is as yet no methodology for first finding an empirical capacity limitation and then inferring system architecture. As the preceding sections of this paper have shown, all the methodologies that seek to determine how capacity is allocated first make assumptions about the structure of the system. While Sternberg and Knoll (1973) attempt something like this, that is, they use an additive-factors methodology to define independent channels in temporal order judgments, they do not claim that channels so defined have anything to do with attentional selectivity (p. 637) and issues of capacity.

As has been noted in the preceding section, the formidable problems associated with a rigorous application of stage analysis to issues of capacity have led some theorists to drop the stage concept entirely, and to conceive of capacity limitations in a more molar strain. This allows us to ignore system architecture to a large degree, but only at the price of the disadvantages mentioned earlier in connection with the models of Kahneman (1973) and Norman and Bobrow (1975). It seems preferable to attempt elucidating system architecture, even if certain configurations cannot be empirically distinguished (Townsend, 1974), rather than opt for global models that ignore inferred structure. However, it is only fair to admit that valuable insights can be gained from such molar strategies and perhaps they are better regarded as precursors to some more detailed analysis of system architecture than as being antithetical to the goals of stage methods of analysis.

Summary

A stage is difficult to define. Localizing capacity limitations within a stage is meaningful only to the extent that other stages and their relationships with each other (system architecture) are specified. More recent attempts at stage analysis focus upon the element as the basic unit of information processing. If an element can be properly identified (Taylor, 1976, p. 183), then capacity limitations can be meaningfully localized at the level of the element but this still leaves great flexibility in the arrangement and properties of stages.

An Experimental Demonstration

The basic assumption of the double-stimulation methodologies, as well as the stage analysis methodology discussed in preceding sections, is that the information flow by which some task is accomplished remains invariant when either a secondary task is added or when levels of independent variables are manipulated to determine factor interactions and additives. Even stronger assumptions are often made as when an experimenter decides a priori that some particular class of manipulation (say, changing stimulus brightness) produces changes within a particular processing stage (say, stimulus encoding) or does (or does not) demand capacity. There is an alternative to such a priori assumptions about similarity of information flow for tasks performed under different conditions or about localization of effects in particular processing stages. This is to examine experimental results with an open mind as to the nature of processing differences produced by manipulating experimental tasks. The following previously unpublished data (Snyder & Kantowitz, Note 1) illustrate some of the pitfalls discussed earlier in this regard.

Method

Subjects. In each of the two experiments to be presented, 13 different Purdue undergraduate students served to satisfy course requirements in Introductory Psychology.

Apparatus. In Experiment I a Psionix 1600 digital-logic system controlled stimulus presentation and recorded reaction times to the nearest millisec. An IEE in-line display was used to present digits as well as a digit-mask of checkerboard squares. In Experiment II an Automatic Data Systems 1800 minicomputer replaced the logic system and a Tektronix display with a more effective dot-matrix mask replaced the in-line display. In both experiments vocal reaction time was recorded by a microphone voice key system capable of responding at rates up to 10 HZ. A piano-type response key requiring a static force of 48 gm and a travel distance of .15 cm was used for manual responses.

Procedure. Procedures were the same for both experiments. A digit was visually presented. The vocal response required subjects to either name the digit (N) or to subtract 9 from it (9-N). The manual response was a simple (probe) reaction requiring subject to depress the response key when a digit appeared. On a random half of each 32-trial block, the digit was obscured by a mask. In dual-task conditions subjects performed both verbal and manual responses simultaneously. In single-task conditions, only a verbal or a manual response was required. Digit sets consisted of the digits 1-2, 1-4, and 1-8 with each digit appearing equiprobably within the 3 digit sets. Thus, there were 6 verbal-task conditions (3 digit sets X 2 verbal transformations) that could be performed either alone or in concert with a manual response. Adding a single stimulation manual-response only condition created a total of 13 experimental treatments with order of testing determined by a 13 x 13 Latin square.

Results and Discussion

Except for the effectiveness of the mask, results were identical for both experiments and so will be discussed together. Figure 1 shows verbal and manual RT for correct responses for dual-task conditions. Single-task simple manual RT is listed at the bottom of the figure.

For verbal RT all three independent variables yielded significant effects (at the .05 level or better) as well as a significant interaction of Task X Number. Previous studies (both probe RT and additive factors) have indicated that masking (stimulus degradation) affects a different processing stage than that influenced either by number of alternatives or task complexity, whereas these latter two share a common stage. So present results are entirely compatible with this interpretation.

For manual RT there were significant effects of Mask, Task and a marginal effect ($p < .10$) of Number and a marginal interaction between Task and Number. These data can be best summarized by noting how faithfully the manual RT mirrors the verbal choice RT.

Since according to additive-factors logic, Task and Number affect a common stage we should expect elevated probe RT for these variables since response selection has been widely believed to demand capacity (Hyman, 1953; Sternberg, 1969, among many others). Indeed, such are present results. Since the Mask variable by additive-factors logic affects a different (encoding) stage that does not require capacity according to probe methodology (Posner & Boies, 1971), Mask manual RT should not be elevated relative to the Mask-only control condition. But it is! How can this be? We know from additive-factors methodology that there is a separate encoding stage and we know from probe methodology that it requires no capacity. Yet two experiments yield identical results, suggestion that this dilemma arises from faulty logic rather than unreliable data.

The logical flaw is the tacit assumption that a stimulus variable (Mask) affects only a stimulus (encoding) processing stage in single- and dual-task conditions. But another plausible explanation can be offered.[5] Suppose that a degraded stimulus requires more time to be encoded than a clear stimulus. Then a response would have to be withheld until such encoding was completed. This requires a kind of response inhibition, that I have previously termed response interdiction (Kantowitz, 1974). Response interdiction can generalize to more than one effector mechanism. So on this basis we might expect manual RT to also be delayed. This is indeed the outcome shown in Figure 1: anything that delayed verbal RT also delayed manual RT.

More generally, there has been a change in the processing operations underlying the verbal task, as a result of degrading the stimulus digit. An extra step, that might be fancifully termed "inhibit response until degradation is removed," has been added to the information flow. This in turn influenced manual RT. According to this argument, a stimulus variable has indirectly influenced manual RT via a response process, rather than by capacity demands of stimulus identification.

So, it may be dangerous to assume that independent variables influence only those stages we know a priori they should. We have no guarantees that what looks like the same task always induces the same information flow within the organism. Methodology is never a substitute for thought. Any methodology is only as good as the thoughtful assumptions it carries with it.

Final Summary

In the preceding pages I have tried to evaluate the utility of the construct of capacity, and have freely criticized the manner in which some investigators and theorists have rendered the concept progressively more vague and ill-defined. Since it is nihilistic, and perhaps craven as well, to proffer criticism without taking a position oneself, I use this section to state my own views about capacity. While my opinions have already been tacitly expressed, some reviewers suggested that a cohesive summary would be beneficial.

First, while I have suggested that the concept be pruned, this is far removed from eliminating it entirely. Capacity has been used wisely by such investigators as Broadbent (1958, 1971) and Townsend (1974; Townsend & Ashby, 1978). Psychology is not the only social science to have multiple uses for capacity (see Winston, 1977, for a discussion of capacity in economics) and these can be successfully pared.

Ultimately, the utility of any theoretical concept hinges upon its ability to aid prediction of behavior. Since this paper has focused attention upon the unobservable concept of capacity, rather than upon observable behavior, some readers may have incorrectly inferred that I regard capacity as more important than behavior. For example, my discussion of POC functions (plots of behavior) centered about the difficulties of using them to distinguish among competing views of capacity. But the complimentary process is equally applicable in that such discussion can also be regarded as an examination of the predictions of capacity models about POC functions. I regard this latter point of view as more important.

My main point has been that the current trends towards a more ambiguous concept of capacity can and should be combatted by more conscientious search for converging operations. This is why I spent so many pages upon the various methodologies such as single and double response measures of capacity available to the researcher. I hope this paper encourages researchers to turn away from a single technique, be it probe methodology or the method of additive factors, and instead to expand the generality of their conclusions by using alternate methodologies as converging operations. It is my belief that a general concept of capacity that is applicable to a wide variety of experimental situations can be best achieved by seeking a wide variety of converging operations, and not by allowing the precision of the concept to lapse in order to expand its scope. Well-intentioned efforts at integrating the large amounts of published data in the area of attention have led to such a diffusion of the capacity concept that it is becoming rather difficult to devise empirical tests that can distinguish among these broadened conceptions.

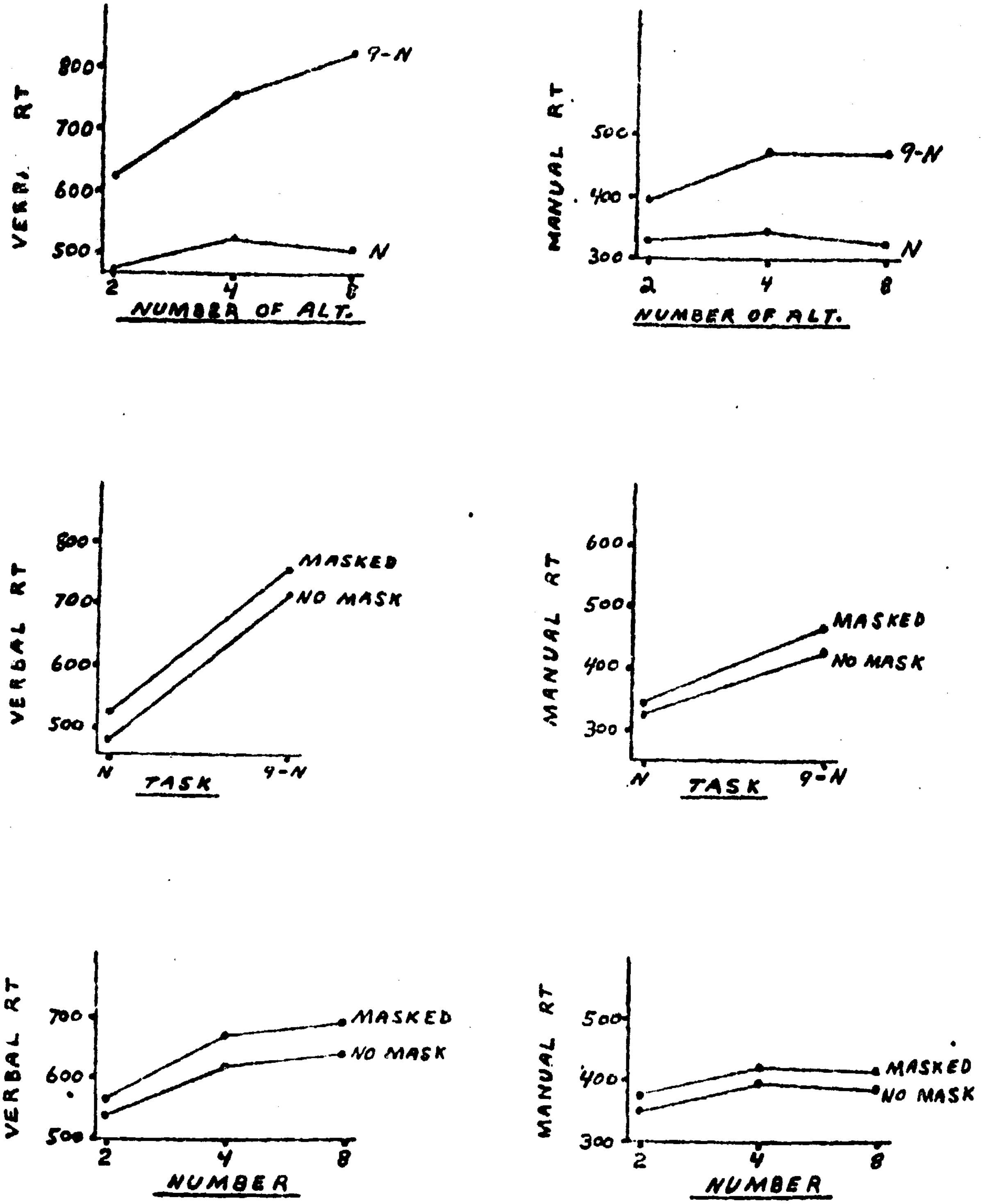
One example of this problem can be seen in the concept of an automatic process (Kantowitz & Knight, 1978a). Is a process automatic if it proceeds without demanding capacity, if it cannot be voluntarily inhibited, if it is controlled by a "motor program," if its variance does not increase when performed concurrently with other tasks? All of these definitions (and more) have been seriously proposed and investigated. Yet any investigation of automatic processes must consider many of the difficulties previously discussed in regard to capacity.

These problems can be surmounted by planning research with converging operations in mind. Capacity, carefully defined, carefully measured by converging operations, and carefully localized in a specific system architecture, can help us predict and explain behavior.

Figure Caption

Figure 1. Verbal and manual reaction times for Experiments I and II.

EXPERIMENT I

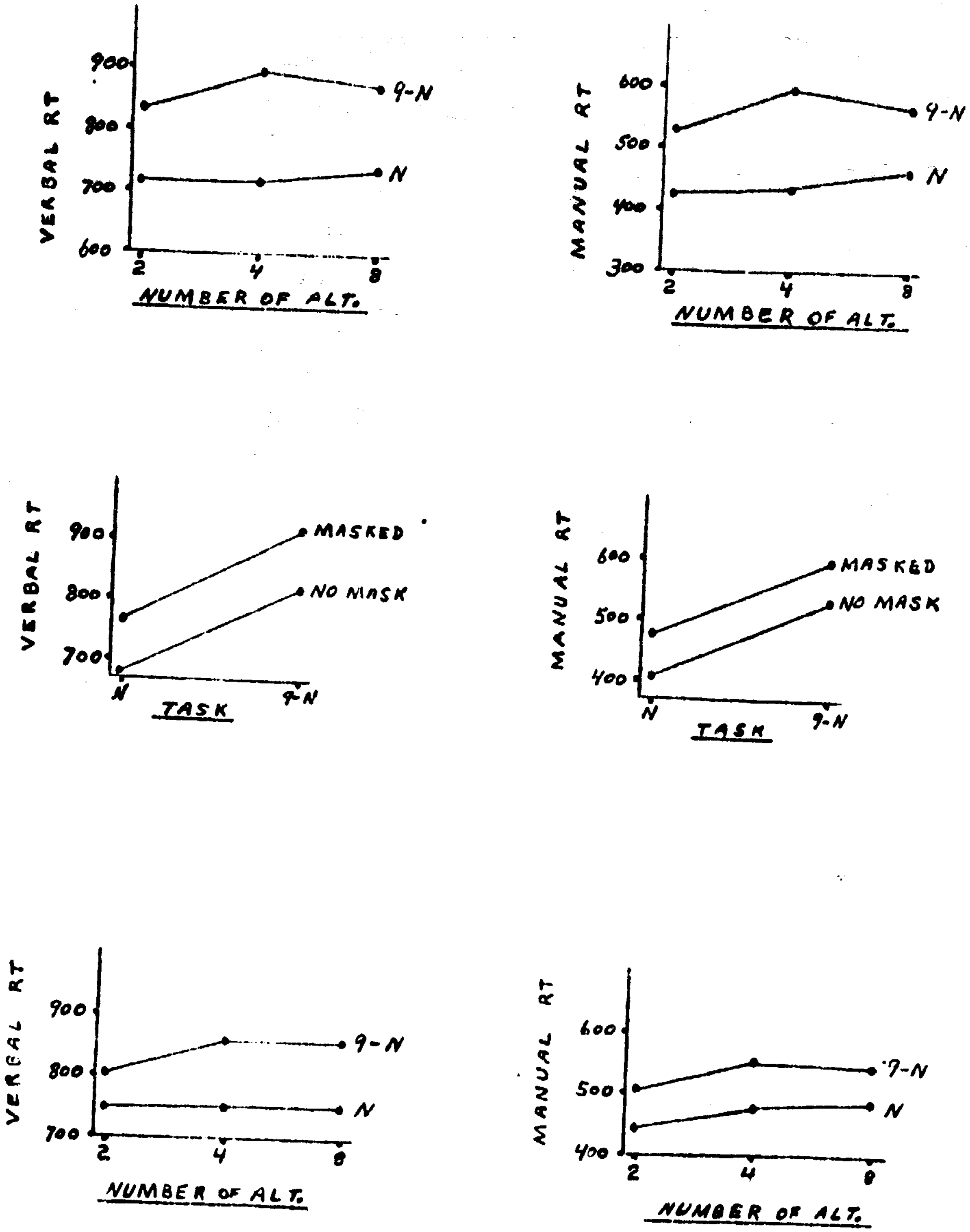


MANUAL ONLY: NO MASK = 256

MASKED = 259

Figure [illegible]

EXPERIMENT II



MANUAL ONLY: NO MASK = 266

MASKED = 300

Figure 7

ORIGINAL PAGE IS
OF POOR QUALITY

Reference Note

Snyder, C. R. R. & Kantowitz, B. H. Simple RT delay as a measure of processing load: Caveat emptor. Presented to Midwestern Psychological Association. Chicago 1975.

References

Allport, D. A., Antonis, B. & Reynolds, P. On the division of attention. A disproof of the single channel hypothesis. Quarterly Journal of Experimental Psychology, 1972, 24, 225-235.

Borger, R. The refractory period and serial choice-reactions. Quarterly Journal of Experimental Psychology, 1963. 15, 1-12.

Briggs, G. E. On the predictor variable for choice reaction time. Memory & Cognition, 1974, 2, 575-580.

Broadbent, D. E. Perception and communication. London, Pergamon, 1958.

Broadbent, D. E. Application of information theory and decision theory to human perception and reaction, 1965.

Broadbent, D. E. Decision and stress. London, Academic Press, 1971.

Craik, K. J. W. Theory of the human operator in control systems, I. The operator as an engineering system. British Journal of Psychology, 1947, 38, 56-61.

Craik, K. J. W. Theory of the human operator in control systems, II. Man as an element in a control system. British Journal of Psychology, 1948, 38, 142-148.

Fitts, P. M. Perceptual-motor skill learning. In A. W. Melton (Ed.) Categories of human learning. New York, Academic Press, 1964.

Garner, W. R. Uncertainty and structure as psychological concepts. New York, Wiley, 1962.

Garner, W. R. The processing of information and structure. Hillsdale, NJ: Lawrence Erlbaum Associates, 1974.

Garner, W. R. & Morton, J. Perceptual independence. Definitions models and experimental paradigms. Psychological Bulletin, 1969, 72, 233-259.

Hicks, W. E. On the rate of gain of information. Quarterly Journal of Experimental Psychology, 1952, 4, 11-26.

Hyman, R. Stimulus information as a determinant of reaction time. Journal of Experimental Psychology, 1953, 45, 188-196.

Kahneman, D. Attention and effort. Englewood Cliffs, NJ, Prentice Hall, 1973.

Kantowitz, B. H. Response force as an indicant of conflict in double stimulation. Journal of Experimental Psychology, 1973, 100, 302-309.

Kantowitz, B. H. Double stimulation. In B. H. Kantowitz (Ed.) Human information processing. Tutorials in performance and cognition. Hillsdale, NJ, Lawrence Erlbaum Associates, 1974.

Kantowitz, B. H. On the beaten track. Contemporary Psychology, 1975, 20, 731-733.

Kantowitz, B. H. & Knight, J. L. Testing tapping timesharing. Journal of Experimental Psychology, 1974, 103, 331-336.

Kantowitz, B. H. 4 Knight, J. L. Testing tapping timesharing, II: Auditory secondary tasks. Acta Psychologica, 1976, 40, 343-362. (a)

Kantowitz, B. H. & Knight, J. L. On experimenter-limited processes. Psychological Review, 1976, 83, 502-507. (b)

Kantowitz, B. H. & Knight, J. L. Testing tapping timesharing: Attention Demands of Movement Amplitude and Target Width. In G. E. Stelmach (Ed.) Information Processing in Motor Learning and Control. New York, Academic Press, 1978. (a)

Kantowitz, B. H. & Knight, J. L. When is an easy task difficult and vice versa? Acta Psychologica, 1978, in press. (b)

Kerr, B. Processing demands during mental operations. Memory & Cognition, 1973, 1, 401-412.

Knight, J. L. & Kantowitz, B. H. Speed-accuracy trade-off in double stimulation. Effects on the first response. Memory & Cognition, 1974, 2, 522-532.

Moray, N. Where is capacity limited? Acta Psychologica, 1967, 27, 84-92.

Moray, N. & Fitter, M. A. Theory and measurement of attention. In S. Kornblum (Ed.) Attention and Performance IV. New York, Academic Press, 1973.

Munson, W. A. & Karlin, J. E. The measurement of human channel transmission characteristics. Journal of the Acoustical Society of America, 1954, 26, 542-553.

Neisser, U. Cognitive psychology. New York, Appleton Century Crofts, 1967.

Norman, D. A. & Bobrow, D G. On data-limited and resource-limited processes. Cognitive Psychology, 1975, 7, 44-64.

Pachella, R. G. The interpretation of reaction time in information-processing research. In B. H. Kantowitz (Ed.) Human information processing. Tutorials in performance and cognition. Hillsdale, NJ, Lawrence Erlbaum Assoc., 1974.

Posner, M. I. Psychology of attention. In C. Blakemore & M. S. Gazzaniga (Eds.) Handbook of Psychobiology. New York, Academic Press, 1975.

Posner, M. I. & Boies, S. J. Components of attention. Psychological Review, 1971, 78, 391-408.

Rosch, E. Review of W. R. Garner. The processing of information and structure. Contemporary Psychology, 1975, 20, 100-102.

Schvaneveldt, R. W. Effects of complexity in simultaneous reaction time tasks. Journal of Experimental Psychology, 1969, 81, 289-296.

Shannon, C. E. The mathematical theory of communication. In C. E. Shannon & W. Weaver. The mathematical theory of communication. Urbana, University of Illinois Press, 1949.

Smith, E, E. Choice reaction time. An analysis of the major theoretical positions. Psychological Bulletin, 1968, 69, 77-110.

ORIGINAL PAGE IS
OF POOR QUALITY

Smith, M. C. Theories of the psychological refractory period. Psychological Bulletin, 1967, 67, 202-213.

Sternberg, S. The discovery of processing stages. Extensions of Donders' method. Acta Psychologica, 1969, 30, 276-315.

Sternberg, S. & Knoll, R. L. The perception of temporal order. Fundamental issues and a general model. In S. Kornblum (Ed.) Attention and Performance IV. New York, Academic Press, 1973.

Taylor, D. A. Stage analysis of reaction time. Psychological Bulletin, 1976, 83, 161-191.

Thomas, E. A. C. The selectivity of preparation. Psychological Review, 1974, 81, 442-464.

Townsend, J. T. Issues and models concerning the processing of a finite number of inputs. In B. H. Kantowitz (Ed.) Human information processing. Tutorials in performance and cognition. Hillsdale, NJ, Lawrence Erlbaum Assoc., 1974.

Treisman, A. M. & Fearnley, S. Can simultaneous speech stimuli be classified in parallel? Perception & Psychophysics, 1971, 10, 1-7.

Treisman, A. M., Squire, R. & Green, J. Semantic processing in dichotic listening? A replication. Memory & Cognition, 1974, 2, 641-646.

Triggs, T. J. Capacity sharing and speeded reactions to successive signals. Unpublished doctoral dissertation, University of Michigan, 1968.

Way, T. C. & Gottsdanker, R. Psychological refractoriness with varying differences between tasks. Journal of Experimental Psychology, 1968, 78, 38-45.

Welford, A. T. Fundamentals of skill. London, Methuen, 1968.

Welford, A. T. The "psychological refractory period" and the timing of high-speed performance--A review and a theory. British Journal of Psychology, 1952, 43, 2-19.

Winston, Gordon C. Capacity: An Integrated Micro and Macro Analysis. Proceedings of the American Economic Association, 1977, 67, No. 1, 418-422.

Measuring pilot workload in a moving-base simulator: II. Building levels of workload

Barry H. Kantowitz, Sandra G. Hart, Michael R. Bortolussi, Robert J.Shively
BITS, Inc. NASA BITS, Inc. Purdue University

and Susan C. Kantowitz
BITS, Inc.

Studies of pilot behavior in flight simulators often have used a secondary task as an index of workload (e.g.,Kantowitz, Hart, & Bortolussi, 1983; Wierwille & Connor, 1983). It is routine in such studies to regard flying as the primary task and some less complex task as the secondary task. Thus, flying is considered a unitary task much as the secondary task is considered to be a unitary task. While this assumption is quite reasonable for most secondary tasks used to study mental workload in aircraft (Williges and Wierwille, 1979), the treatment of flying a simulator through some carefully crafted flight scenario as a unitary task is less justified. While this is often a necessary simplification that can be easily forgiven since it yields useful information, it should be remembered that flying is a complex task that is likely to have an hierarchical organization. While researchers concerned with training have never forgotten this, researchers who are concenred with evaluating workload with skilled pilots tend to ignore the general complexity of flying and have been content to acknowledge only the general difficulty of a particular flight scenario with little regard to complexities that might be related to the hierarchical structure of the flight task.

The present research is a first step towards acknowledging that total mental workload depends upon the specific nature of the sub-tasks that an aircraft pilot must complete. As a first approximation, we have divided flight tasks into three levels of complexity. The simplest level (called the Base level) requires elementary maneuvers that do not utilize all the degrees of freedom of which an aircraft, or a moving-base simulator, is capable. Examples would be flying at a constant altitude or at a constant heading. The second level (called the Paired level) requires the pilot to simultaneously execute two Base level tasks, for example, flying on a constant heading while also maintaining a constant altitude. The third level (called the Complex level) imposes three simultaneous constraints upon the pilot. An example would be flying at a constant altitude, on a constant heading, and at a constant speed. Further example of Base, Paired, and Complex tasks used in this experiment can be found in Table 1. Note that even the Complex level is relatively elementary when compared to the actual demands of flight where other necessary tasks such as navigation and communication must also be performed. This additional complexity is addressed in Experiment II, currently in progress.

Workload is assessed by subjective ratings and by an asynchronous secondary choice-reaction task quite similar to those used by Kantowitz, Hart and Bortolussi (1983). Two general questions are asked. The first involves comparing secondary-task performance under single- and dual-task conditions. Since highly skilled pilots are being tested, one reasonable prediction would be that elementary maneuvers are so automatic and overlearned that they impose no workload on the pilot. Therefore, one would expect no differences between secondary-task performance regardless of whether or not the primary flying task was required. An alternate prediction, based upon the notion that training does not eliminate attentional requirements of flight (Johnson, Haygood & Olson, 1982), would expect faster reaction times and/or fewer errors under single-task conditions. The second general question arises only if the alternate prediction is correct. Given that even these elementary flight tasks create workload, one can then ask if the three different levels of task complexity defined a priori as Base, Paired, and Complex also produce different levels of pilot workload. One might expect that task differences, especially between Base and Paired, are so small that no workload differences should be produced or one might predict that workload should increase as levels go from Base to Complex. And of course, one can always ask the eternal question in workload studies by attempting to relate subjective and objective measures of pilot workload.

METHOD

Pilots

Seven male and five female instrument-rated pilots served as paid participants. Four pilots had a private pilot license, six had commercial licenses, and two had airline transport licenses. Pilots had from 500 to 6000 hours of total flight time (median=1025 hours) and from 30 to 1200 hours of actual instrument time (median=130 hours).

Flight Tasks

Each pilot flew 21 separate flight tasks (Table 1) twice, once with the

ORIGINAL PAGE IS
OF POOR QUALITY

TABLE 1

BASE LEVEL-TASK.	PITCH	ROLL	YAW	ALT	ASI
1. FLY HDG 360	F			F	F
2.MAINTAIN 2000FT.		F	F		F
3. "S" TURN	F			F	F
4. CLIMB AT 500FPM		F	F		F
5. DESCEND AT 500FPM		F	F		F
6.MAINTAIN 120KTS.		F	F	F	
PAIRED LEVEL-TASKS					
1. FLY HDG 360,MAINTAIN 2000FT.					F
2. MAINTAIN 2000FT.,"S" TURN					F
3. FLY HDG 360,CLIMB AT 500FPM					F
4. FLY HDG 360,DESCEND AT 500FPM					F
5. "S" TURN,CLINB AT 500FPM					F
6. "S" TURN,DESCEND AT 500FPM					F
7. FLY HDG 360,MAINTAIN 120KTS				F	
8. MAINTAIN 2000FT.,MAINTAIN 120KTS		F	F		
9. "S" TURN, MAINTAIN 120KTS.				F	

COMPLEX LEVEL-TASKS

1. FLY HDG 360,MAINT 2000FT.,MAINT 120KTS.
2. FLY HDG 360,DESC. AT 500FPM,MAINT 120KTS.
3. FLY HDG 360,CLIMB AT 500FPM,MAINT 120KTS.
4. "S" TURN 360,DESC. AT 500FPM,MAINT 105KTS.
5. "S" TURN,CLIMB AT 500 FPM,MAINT 105 KTS.
6. "S" TURN,MAINT 2000FT.,MAINT 120KTS.

secondary task and once by itself. Each flight task lasted three minutes. All flight tasks were flown in a Singer/Link GAT-1 instrument trainer with three degrees of freedom. As indicated in Table 1, certain degrees of freedom were frozen for certain flight tasks. This prevented the pilot from attempting to control irrelevant simulator motion. Freezing a task component also froze the corresponding instruments inside the simulator.

Secondary Task

Three positions of a helicopter trim switch ("coolie-hat" switch) mounted on the left side of the control yoke under the pilot's thumb were used for responses to auditory tones. A low tone (800 Hz) was paired with switch motion to the left, a medium tone (1500 Hz) with a forward switch motion, and a high tone (4000 Hz) with a right switch motion. Tones were 300 msec in duration and approximately 70 dB SPL, presented over headphones. An Apple II computer with a Cyborg Model 91A interface generated tones and recorded reaction time to the nearest millisecond as well as errors. Tones were presented asynchronously--that is, regardless of performance on the flying task--every eight seconds.

Normally, it is prudent to utilize two levels of difficulty in the secondary task (Kantowitz & Knight, 1976) to ensure that data can be theoretically interpreted. However, only one level (3-choice task) was used in this study because an earlier study using much the same secondary task (Kantowitz, Hart & Bortolussi, 1983) found no interaction with two- and four-choice auditory secondary tasks.

ORIGINAL PAGE IS OF POOR QUALITY

Procedure

Each of the 21 flight tasks were flown twice: with and without the secondary RT task. As a single-task control condition, the RT task was performe. alone in the GAT cockpit at the end of each flight level. All 31 orders of flight level were used with two subjects randomly assigned to each order. In each block half of the pilots flew the task with tone first (dual-task condition) and the other half flew first without tones (single-task condition).

All pilots were given approximately 30-40 minutes of simulator practice to learn the flight characteristics of the GAT before starting the experiment proper. Practice on the auditory choice-reaction task continued until a criterion of 95% -98% accuracy was achieved.

Immediately after each single-task flight condition, pilots completed bipolar rating scales for ten items. During all simulated flight airspeed, altitude, x-y position and rudder, elevator and aileron control deflection were continuously recorded.

RESULTS

Primary Task Performance

The major concern to be evaluated is a comparison of single- versus dual-task performance for the flying task. The relative performance for the 21 flight tasks of Table 1 is not of major interest, especially since it is not clear how to directly compare different tasks, e.g., how much rms error in altitude is equivalent to a given rms error in heading? It is, however, possible to compare Paired and Complex tasks with the appropriate Base tasks since here the units are comparable but Paired and Complex tasks cannot be contrasted.

Figure 1 shows rms error for single- versus dual-task performance for each of the three levels of complexity. Three separate analyses of variance (one at each level) revealed no significant differences between flying alone and flying plus responding to tones for the Base level (F(1,11) = 0.04), Paired level (F(1,11) = 0.54), and Complex level (F(1,11) = 0.18). Thus, adding the secondary-tone task did not alter flying performance.

A vector analysis was computed in order to contrast Base versus Paired and Base versus Complex flight performance. This is best illustrated by the Base versus Paired comparison which can be plotted in two-dimensional space but the extension to the three-dimensional space of the Base versus Complex comparison is straightforward. Let us select as an example a comparison of Base performance of flight tasks 1 and 2 in Table 1 with flight task 7 that demands simultaneous performance of tasks 1 and 2. In a two-dimensional space we can plot Base performance with rms error in heading as a point on the abscissa and rms error in altitude as another point on the ordinate. Paired performance can be represented by a single point in this vector space. We then calculate the length of the existing vector representing Paired performance and also the length of the implied vector formed by projecting the two Base points perpendicular to their respective axes until they meet. Note that this implies an equal weighting of the scales shown on the abscissa and ordinate and that such an assumption requires empirical justification which we shall soon provide. Figure 2 shows comparisons based upon vector length. As we would expect from Figure 1, there was no significant effect of single- versus dual-task for either the Base vs. Paired comparison, F(1,384) = .14, or the Base vs. Complex comparison, F(1,240) = 1.03. However, significant effects indicating reliably smaller

PRIMARY TASK

RMSE AS A function of Single vs. Dual task PERFORMANCE

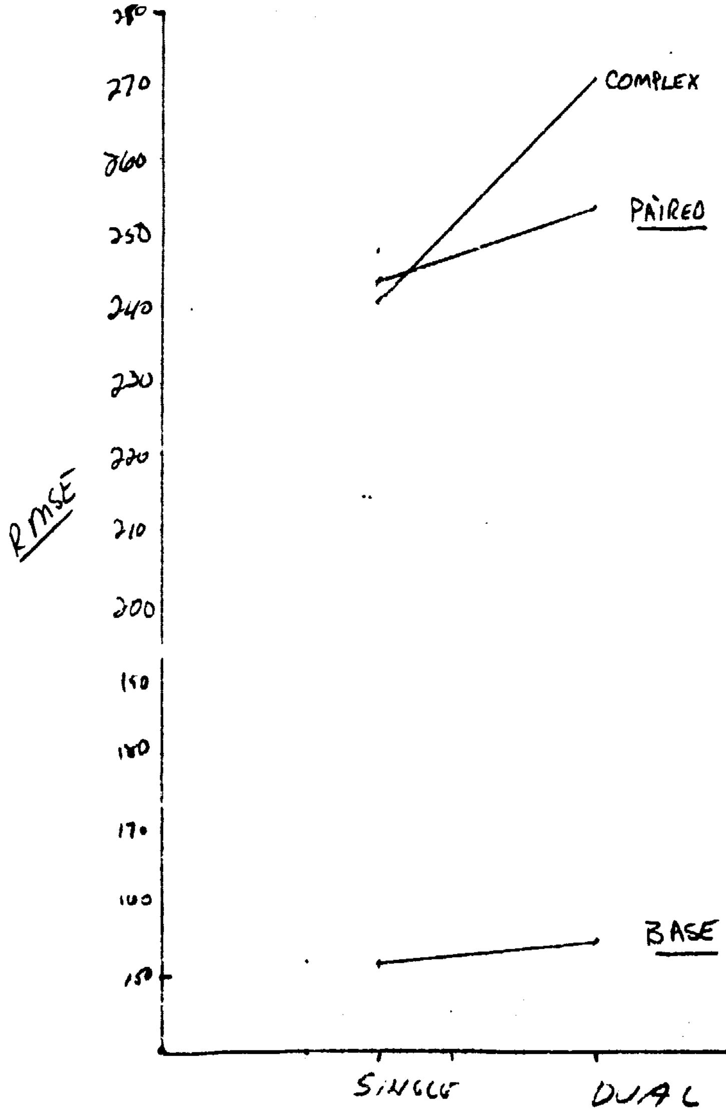


Figure 1

PRIMARY TASK

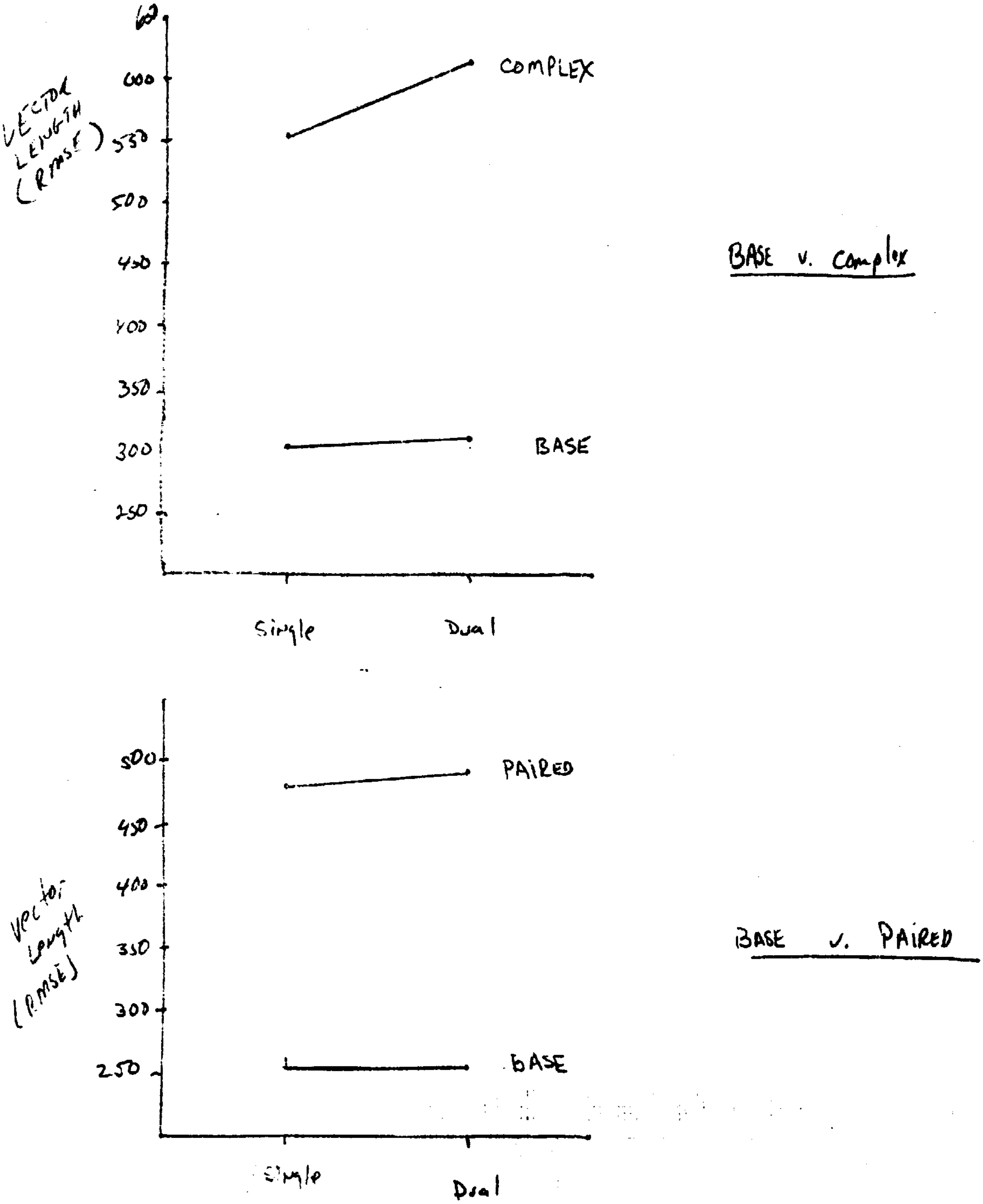


Figure 2

rms error in the Base condition were obtained for both Base vs. Paired, F(1,384) = 31.63, p<.001, and Base vs. Complex, F(1,240) = 32.55, p<.001, comparisons. No significant interactions were obtained for either comparison.

In order to check the validity of the equal-weighting assumption mentioned above, an additional analysis was performed whereby the length of a vector's projection upon an axis was compared to Base performance on that axis. If performance for the Base condition was worse than the corresponding vector projection, this might indicate a trade-off between task components where outstanding performance on one task component (i.e., performance better than that component performed singly during the Base condition) was achieved at the expense of performance on the remaining vector projection(s). There were 43 possible paired comparisons of this nature for single- and also for dual-task performance. Since there were 12 subjects a total of 1032 data points were examined (43 X 2 X 12). We searched for cells in which at least 9 subjects showed lesser vector projections since this would be a significant number of subjects by sign test. Of the total of 86 cells (43 single- and 43 dual-task) only three cells had 9 such deviant pilots and no cell had 10 or more deviant pilots. Hence, we conclude that an equal-weighting assumption is reasonable for these data.

To recapitulate, the tortuous analysis of primary task performance, required since the various rms error scales are not equivalent, showed that Base performance was better than either Paired or Complex performance. This is hardly an astonishng outcome and the detailed vector analysis should not detract from the more important result shown in Figure 1 that addition of a secondary task did not alter primary flight-task performance.

Secondary Task Performance

For each pilot and each flight task, transmitted information (bits/sec) was calculated for the secondary three-choice reaction task. Since this measure takes both speed and accuracy into account, it is the optimal index of secondary-task performance (Kantowitz, Hart, & Bortolussi, 1983). Figure 3 shows that transmitted information was highest for the Base level conditions and declined with higher flight-task levels, $F(2,22) = 8.23$, $p<.001$. As was expected, reliably more information was transmitted during the single-task control conditions, $F(1,18) = 39.6$, $p<.001$. However, while transmitted information was able to discriminate among levels of flight task, three separate analyses of variance performed within each level (Figure 4) were unable to detect any reliable differences.

Figure 5 shows the same results as Figure 3, except that reaction time and errors are plotted separately rather than combined as transmitted information. Effects of level were significant for both reaction time, $F(2,252) = 33.1$, $p<.001$, and errors, $F(2,252) = 4.12$, $p<.05$.

Subjective Ratings

Subjects were asked to rate each of 21 flight tasks using 10 bipolar rating scales. The results of the analyses of variance indicate that all the scales were able to distinguish between at least two of the 21 flight tasks (Table 2.).

Further analysis was done to determine the effect of flight task on rating behavior. The subjects gave a subjective rating of importance to

ORIGINAL PAGE IS
OF POOR QUALITY

Secondary Task Performance as a function of level of the primary task.

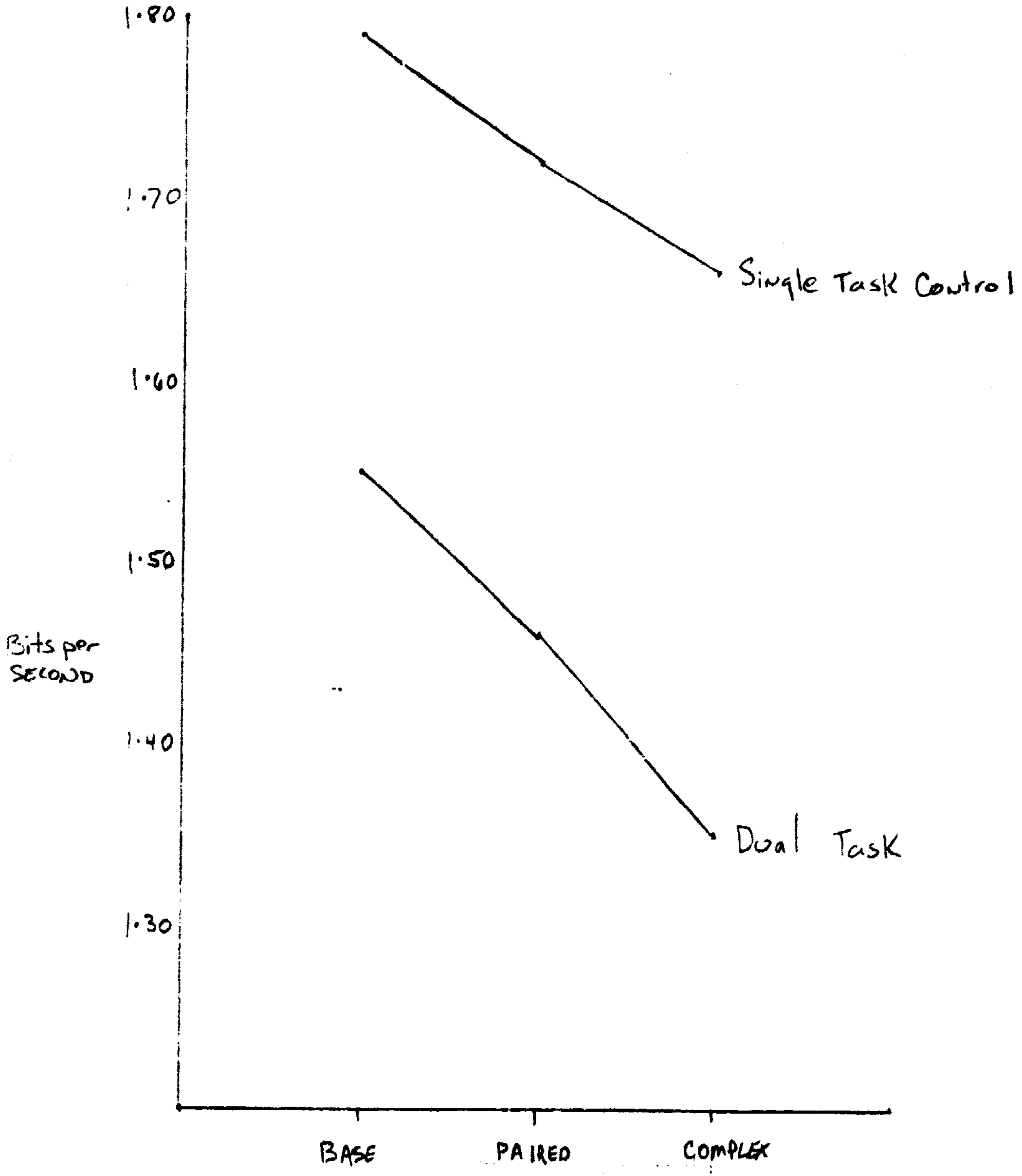


Figure 3

ORIGINAL PAGE IS OF POOR QUALITY

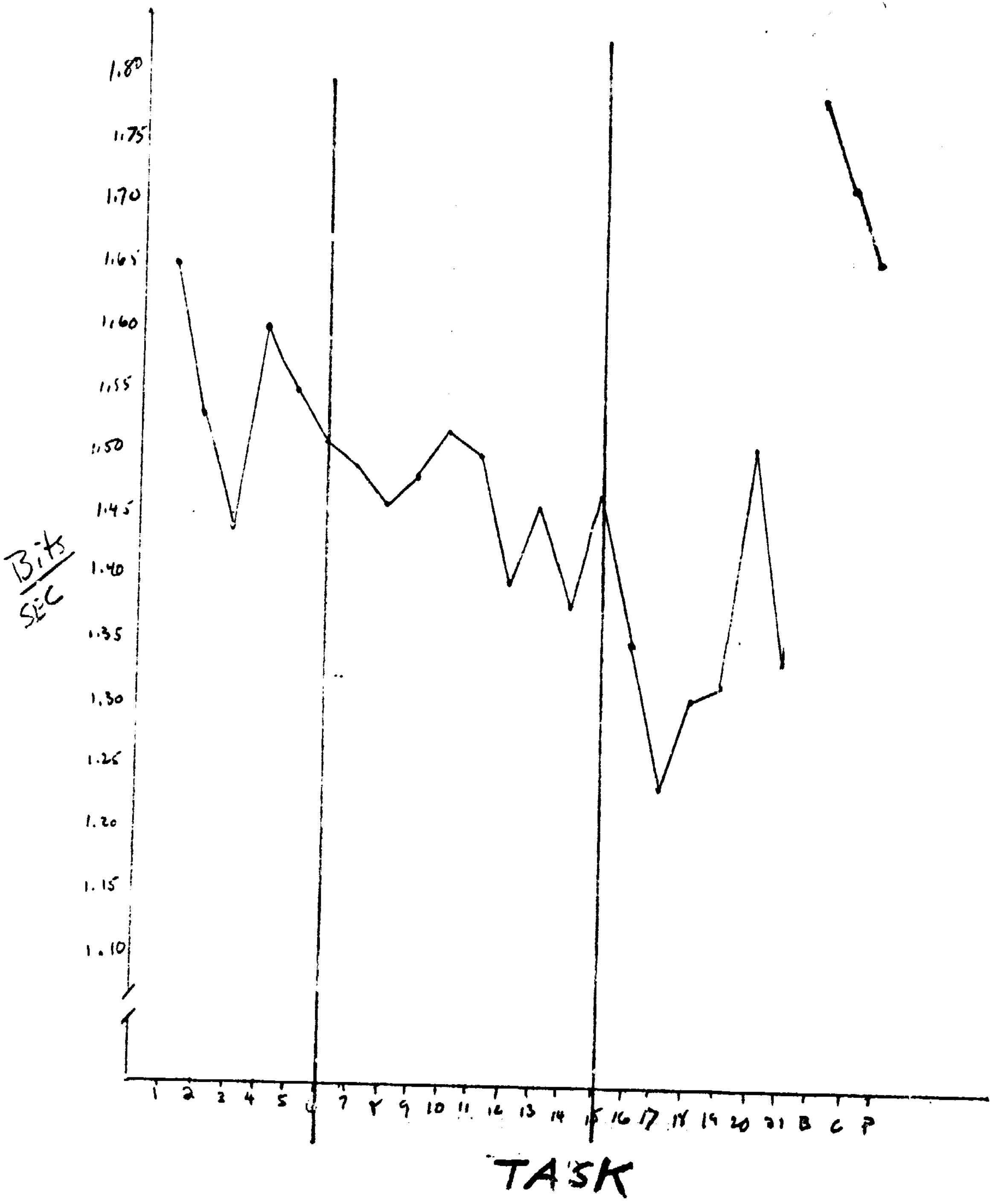


Figure 4

ORIGINAL PAGE IS
OF POOR QUALITY

Secondary task performance as a function of level of the primary task.

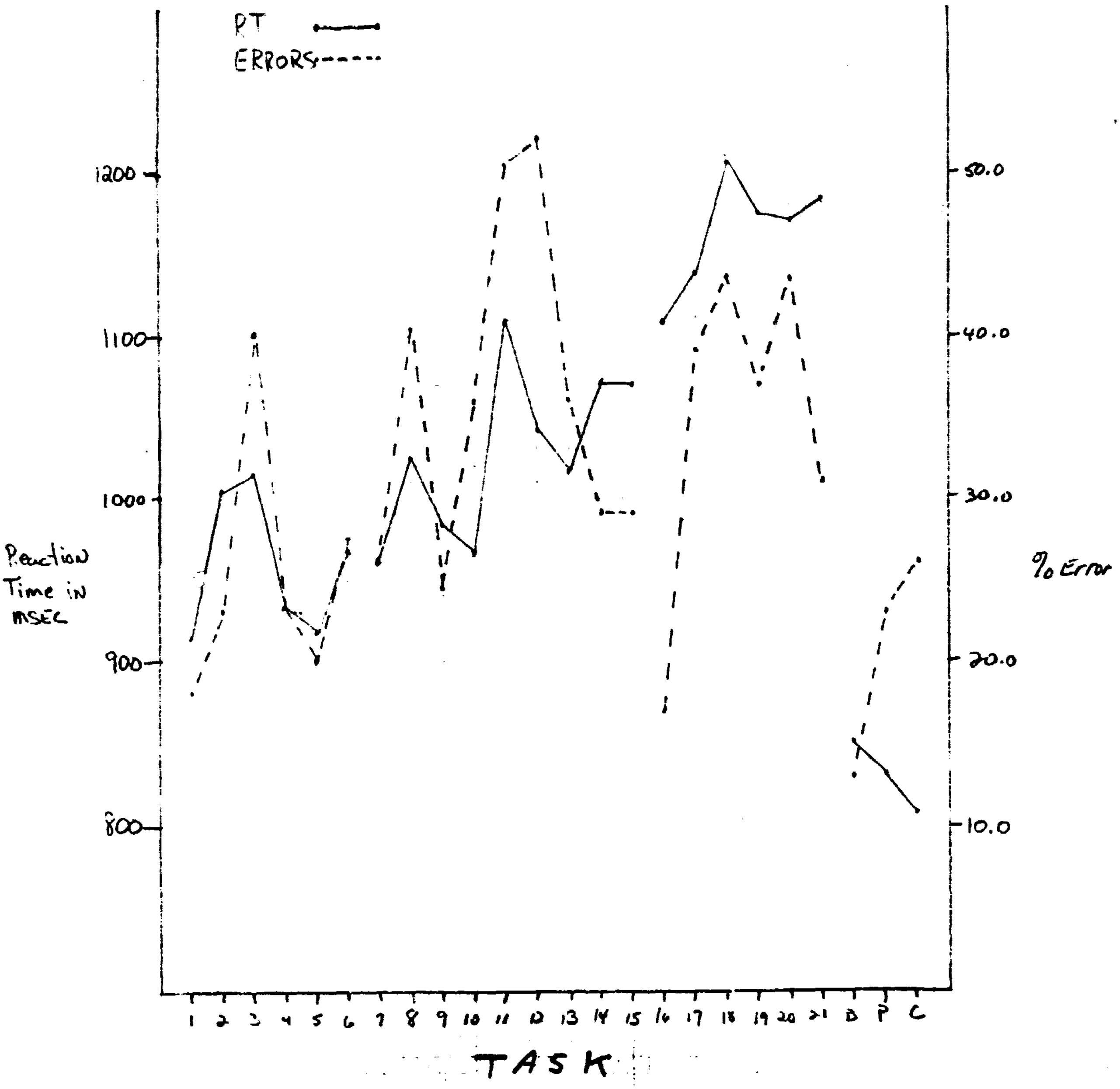


Figure 5

FLIGHT TASK	Task Difficulty	Time Pressure	Performance	Mental Effort	Physical Effort	Frustration	Stress	Fatigue	Activity Type	Overall Workload
1	12.917	8.833	29.917	34.000	13.583	21.917	15.583	32.167	9.250	22.667
2	19.333	13.000	31.250	30.500	16.583	26.667	19.917	29.833	10.000	24.167
3	24.667	14.667	32.583	41.750	25.167	30.583	23.167	33.167	8.000	28.750
4	17.750	10.750	31.833	31.833	17.917	26.000	23.583	35.250	9.833	19.583
5	22.000	13.417	36.667	36.250	18.833	28.000	23.833	35.000	9.833	25.750
6	15.583	9.167	32.333	36.500	12.917	23.417	17.667	35.000	6.833	17.750
7	32.583	17.417	44.083	51.417	24.500	35.417	24.333	37.000	9.333	39.167
8	41.583	24.750	53.333	57.417	25.750	42.583	30.583	41.917	14.000	43.833
9	38.667	19.083	49.333	48.250	20.583	37.500	25.583	42.250	15.583	46.500
10	37.000	19.083	45.917	48.500	25.333	31.417	24.417	45.333	10.500	45.583
11	45.750	20.750	44.250	56.000	24.333	29.583	23.333	43.083	11.583	51.750
12	50.167	19.583	50.833	52.667	23.167	38.667	28.917	48.083	12.667	52.833
13	31.167	12.167	46.750	45.667	21.833	29.917	25.000	47.000	10.667	38.000
14	30.583	13.083	40.167	39.167	20.167	28.667	21.583	39.750	6.667	32.750
15	36.667	13.417	54.583	50.250	18.333	36.750	26.333	41.333	8.250	42.667
16	41.667	15.833	40.167	44.833	23.750	27.167	25.667	39.500	13.750	46.333
17	48.167	16.750	48.083	50.750	32.583	37.333	31.417	46.833	15.167	52.333
18	52.000	17.917	51.167	57.500	25.167	36.750	33.750	49.333	15.583	61.250
19	53.333	19.500	49.917	60.333	32.000	45.167	40.167	51.583	15.250	59.750
20	57.083	22.417	54.333	60.833	35.083	45.333	36.833	52.083	16.417	58.500
21	53.000	21.500	45.833	54.500	30.667	29.417	28.583	50.417	14.667	54.500
F(20,220)	16.1***	1.78*	3.02***	9.06***	3.56***	2.44**	3.62***	2.96**	1.95*	17.1***

Note: ***=p .001
**=p .01
*=p .05

Table 2. Mean Subjective Ratings

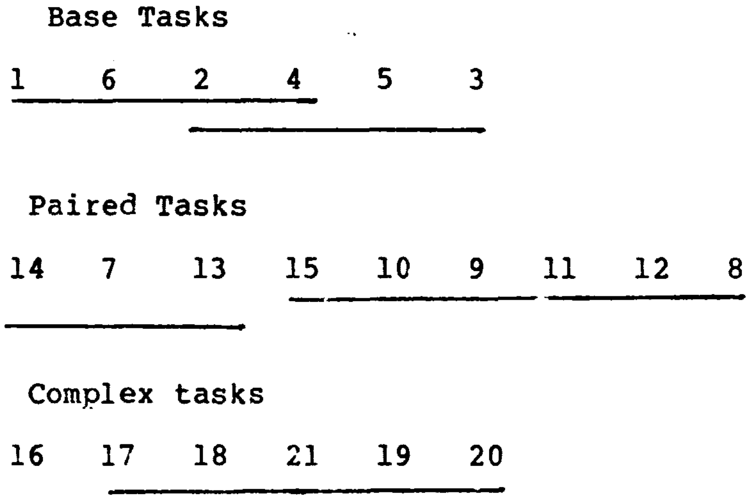
TASK	MEAN	df	F
Base tasks 1-6		5,55	2.73*
1	20.4		
2	22.2		
3	25.4		
4	22.3		
5	24.7		
6	20.6		
Paired tasks 7-15		8,88	2.74*
7	29.8		
8	35.9		
9	31.3		
10	31.3		
11	32.1		
12	34.5		
13	29.8		
14	25.8		
15	31.1		
Complex tasks 16-21		5,55	3.54**
16	29.5		
17	35.9		
18	36.2		
19	39.5		
20	41.3		
21	36.4		

Note: **=p .01
*=p .05

Table 3. Weighted Mean Subjective Ratings

each of the 10 scales. This importance rating was used to weight each subject's summed ratings on all the scales for each of the 21 flight tasks. Analysis of the weighted mean scores over all flight tasks indicates that at least 2 of the 21 flight task means are significantly different, F(20,220)=8.84, p<.001. The flight tasks were divided into 3 categories and separate analysis were calculated on the mean scores in each category. The results indicate that at least 2 of the means for each category differ significantly (Table 3).

To determine which flight task means differed, t-tests were calculated on all possible pairs of flight tasks within each category. The significant mean differences are summarized in Table 4.



Base Tasks

1 6 2 4 5 3

Paired Tasks

14 7 13 15 10 9 11 12 8

Complex tasks

16 17 18 21 19 20

Tasks are arranged in increasing mean value for each category. The line indicates those means that do not differ significantly at p<.05.

Table 4. Pairs of Flight Tasks

DISCUSSION

Results clearly showed that even the most elementary flying tasks (Base) produced measurable pilot workload using the objective secondary-task technique. Furthermore, as the flying tasks were made more complicated, progrᵓssing to Paired and Complex tasks, workload increased even more. These findings are impressive confirmation of the utility of the asynchronous choice-reaction secondary task used by Kantowitz, Hart and Bortolussi (1983). Primary task performance was unaffected by the addition of the secondary tone-task while transmitted information decreased with flight-task complexity.

Subjective ratings confirmed the objective results. Furthermore, using ratings that weighted the importance of the bipolar rating scale produced a metric that could distinguish workload within one of the three classes of flight tasks. Therefore, this improved subjective scale was more sensitive than the objective measure which could not discriminate within a class. Due to the short duration of each flight task, it is unlikely that the superiority of the weighted rating scale can be attributed to its measuring peak, rather than average, workload as suggested by Kantowitz et al (1983). Instead, weighted ratings may just be more sensitive measures. The use of such rating data is acceptable when confirmed by objective results.

The next step is to repeat this experiment using flight scenarios that combine more complex flight demands. Thus, instead of one of the present Base tasks, e.g., fly at constant speed, we would substitute a tracking task, e.g. VOR tracking. Then the corresponding Paired level would require VOR tracking while maintaining constant speed. Finally, an analog to the present Complex level would require VOR tracking, constant speed and controlled descent. We would anticipate results similar to the present with greater objective and subjective workload associated with increasing task compexity.

```
#include "stdio.h"

/* Add default values and range limits for all parameters                  */
#include "../c/cdev/defaults.h"

/* Reserve global variable and array storage                               */
#include "../c/cdev/storage.h"
/* Define file name for output data file.                                  */
#define OUTFILE "force.dat"


main ()  {
int blkcount;           /* counter for number of trial blocks */
extern struct par;

        openout (OUTFILE);      /* open output file           */

        /* Prompt operator for experimental parameters.                    */
        parameters ();

        /* Enter experimental process section. The program will terminate
        when the block count goes to zero.                                 */
        for (blkcount=par->numblks; blkcount>=0; blkcount--)
        {

                block (blkcount);       /* Execute block tasks             */

                /* Store data collected in output data file                */
                storedata (blkcount);

        }       /* End of block iterations                                 */
}       /*         End of main.                                            */

/* Define all constants and defaults: The program will retrieve default
   values for all parameters from this section.                            */

#define YES     1
#define NO      0

#define DEFPARSET       YES     /* Default parameter set selection. This
                                selection will allow the use of parameters
                                specified for the first block to be used
                                for subsequent blocks. A NO answer will make
                                the default option user specification of
                                all parameters for all experiment blocks.
                                */
#define DEFPARCHK       NO      /* Default parameter checking option.      */

#define DEFINST         YES     /* Default subject instruction display flag.
                                YES: Give instructions at start of each
                                block, no: No instructions on subject display .
                                                                           */

#define DEFDUR          10      /* Default display update rate. This
                                value gives the default interval
                                in milliseconds between subject
                                display scope updates.
                                Note that this figure must be an integral
                                multiple of 10 milliseconds.               */


#define MAXDUR          10      /* Maximum subject display update rate.
```

```
                                            the maximum rate at which the subject
                                            display is updated. Note that this figure
                                            is subject to physical constraints (e.g.,
                                            the bandwidth of the hardware link
                                            between the computer and the vector
                                            generator, etc).
                                            The response must be an integral multiple
                                            of 10 milliseconds.                     */

#define MINDUR               500            /* Minimum subject display update rate.
                                            This value gives the maximum interval,
                                            or the minimum rate at which the subject
                                            display is refreshed.
                                            The response must be an integral multiple
                                            of 10 milliseconds.                     */

#define DEFVS                1              /* Default Vector/Scalar mode flag (1=vector
                                            mode, 0=scalar mode.                    */

#define MAXTRIALS            100            /* Maximum allowable number of trials in
                                            a block. Specification of trial counts
                                            greater than this is not allowed.       */
#define DEFTPBLK             50             /* Default number of trials per block   */

#define DEFRFM               'B'            /* Default response feedback mode (Binary),
                                            meaning a correct/incorrect answer is the
                                            default. The allowable selections are
                                            b(binary), n(none), and a(running average).
                                            The last mode is a cumulative position error
                                            that is summed over all samples in a trial
                                            */
#define MINRSR               10             /* Minimum response sampling rate. This value
                                            defines the fastest rate at which the subject
                                            response data is taken.                 */

#define DEFRSR               10             /* Default response sampling rate; This value
                                            is in milliseconds, and must be a
                                            multiple of 10ms (Minimum)              */

#define DEFNUMBLKS           5              /* Default number of blocks in an experiment


                                                                                    */

#define MAXBLOCKS            20             /* Maximum allowable number of blocks in
                                            an experiment.                          */

#define MAXLENGTHS           8              /* Maximum allowable number of unique
                                            stimulus vector lengths. This number indicates
                                            only the greatest number of lengths an
                                            experiment can have; it says nothing about
                                            what those lengths are, or exactly how many
                                            there MUST be in an experiment.         */

#define MAXDIRS              8              /* Maximum allowable number of unique
                                            stimulus vector directions. For vector mode
                                            only, this value gives roughly the same type
                                            of information about stimulus vector direction
                                            as MAXLENGTHS gives about vector length. */

#define DEFRESCRIT           'M'            /* Default response criterion; allowable
                                            selections include s(small), m(medium),
                                            and l(large). These correspond to the size
                                            of a 'window' on the display screen that
```

```
                                        constitutes a 'hit' or 'miss'.        */

#define DEFTPT          3       /* Default maximum time per trial; dictates
                                   the time limit for a response to a
                                   stimulus.                                  */

#define MAXTPT          99      /* Maximum time per trial; the allowable
                                   time for a trial may not be set greater
                                   than this value.                           */

#define DEFITI          11      /* Default inter-trial interval (in seconds)*/

#define DEFWARN         1       /* Default trial start warning (0= .5sec tone
                                   1 second before stimulus onset, 1= green
                                   square on display for same length of time.*/

#define MAXITI          99      /* Maximum inter-trial-interval. This
                                   number will be used to evaluate the
                                   response for the iti prompt; no values
                                   greater than MAXITI will be accepted.      */


#define DEFDISPCOLOR    2       /* Default display color combinations         */
/*      The combinations are as follows:

0:      No colors for either stimulus or response.
1:      Green stimulus, red response
2:      green stimulus, yellow response
3:      red stimulus, green response
4:      red stimulus, yellow response
5:      yellow stimulus, green response
6:      yellow stimulus, red response

*/

#define RCSMALL         5  /* Small target area for response criterion  */
#define RCMEDIUM        10 /* Medium target area for response criterion */
#define RCLARGE         20 /* Large target area for response criterion  */

#define OUTFILE         'force.dat'
/*
   Stdio.h:  Cromemco 68000 C I/O header file
   Copyright (c) 1983 by Cromemco, Inc., All Rights Reserved

   This file is for inclusion in programs to be run under
   the Cromix operating system.

   18-May-83
*/

#define BUFSIZ  512             /* default buffer size for buffered I/O */
#define _NFILE  20              /* maximum # of open buffered files */

#ifndef FILE
extern  struct  _iobuf {        /* definition of the structure tab _iobuf */
        char    *_ptr;          /* next byte to access */
        int     _cnt;           /* characters left */
        char    *_base;         /* start of buffer */
        char    _flag;          /* see bit definitions */
        char    _file;          /* file number from Cromix */
} _iob[_NFILE];
#endif

#define _IOREAD 01              /* open for read */
#define _IOWRT  02              /* open for write */
```

ORIGINAL PAGE IS OF POOR QUALITY

```
#define _IOMYBUF        010     /* user-supplied buffer */
#define _IOEOF   020
#define _IOERR   040
#define _IOSTRG 0100
#define _IORW    0200            /* open for read & write */

#define NULL     0
#define FILE     struct _iobuf
#define EOF      (-1)

#define STDIN    0
#define STDOUT   1
#define STDERR   2

#define stdin    (&_iob[0])
#define stdout   (&_iob[1])
#define stderr   (&_iob[2])
#define getc(p)          (--(p)->_cnt>=0? (*(p)->_ptr++)&0377:_filbuf(p))
#define getchar()        getc(stdin)
#define putc(x,p) (--(p)->_cnt>=0? ((int)(*(p)->_ptr++=(unsigned)(x))): \
                                                  _flsbuf((unsigned)(x),p))
#define putchar(x)       putc(x,stdout)
#define feof(p)          (((p)->_flag&_IOEOF)!=0)
#define ferror(p)        (((p)->_flag&_IOERR)!=0)
#define fileno(p)        (p)->_file

#define backc(fp,c)       ungetc(fp,c)
#define ungetchar(c)      ungetc(stdin,c)
#define getline(buf,max) getl(STDIN,buf,max)
#define alloc(x)          malloc(x)


/* [illegible] used macros */

#define isalpha(c) (isupper(c) || islower(c))
#define isdigit(c) ((c) <= '9' && (c) >= '0')
#define islower(c) ((c) <= 'z' && (c) >= 'a')
#define isupper(c) ((c) <= 'Z' && (c) >= 'A')
#define isspace(c) ((c)==' ' || (c)=='\n' || (c)=='\r' || (c)=='\t')
#define max(a,b)   ((a) > (b) ? (a) : (b))
#define min(a,b)   ((a) < (b) ? (a) : (b))
#define toupper(c)    (islower(c) ? (c) - 32 : (c))
#define tolower(c)    (isupper(c) ? (c) + 32 : (c))


/* Functions which don't return int */

FILE    *fopen();
FILE    *freopen();
FILE    *fdopen();
long    ftell();
char    *fgets();
char    *fputs();

double abs();
double sqr();
double sqrt();
double exp();
double ln();
double pwrof10();
double log10();
double sin();
double cos();
double asin();
double acos();
```

ORIGINAL PAGE IS
OF POOR QUALITY

ORIGINAL PAGE IS
OF POOR QUALITY

```
double sinh();
double cosh();
double tanh();

/* Storage for global variables and arrays used throughout the program. */

struct param {  /* Define parameter set storage                            */

int *inst[MAXBLOCKS];    /* Flag for instruct display for each block       */
int *numblks;            /* Total number of trial blocks in experiment.    */
int *numtrials[MAXBLOCKS];        /* T  als per block variable.            */
char *vs[MAXBLOCKS];              /* vector/scalar mode flag               */
char *fdbk[MAXBLOCKS];            /* Feedback mode flag.                   */
char *rescrit[MAXBLOCKS];         /* Response criterion storage.           */
int *iti[MAXBLOCKS];              /* Inter-trial-interval storage.         */
int *warn[MAXBLOCKS];             /* Type-of-warning for trial onset flag */
int *dispcolor;                   /* Display color combinations...         */
int *rsr[MAXBLOCKS];              /* Response sampling rate.               */
int *tpt[MAXBLOCKS];              /* Time per trial.                       */
int *dur[MAXBLOCKS];              /* Storage for display update rate       */

} *par;

struct vector {

int *stimx[MAXBLOCKS];
int *stimy[MAXBLOCKS];
int *numdir;             /* Storage for number of stimulii directions.    */
int *directions[MAXDIRS];         /* Storage for stimulii direction info. */
int *numlengths;         /* Storage for number of stimulii vector lengths*/
int *lengths[MAXLENGTHS];         /* Storage for stimulii lengths info.    */
int *RC;                          /* Response criterion storage..          */

};
struct vector *vect;
struct response {
int *x[2000];            /* Storage for x coordinate of response vector   */
int *y[2000];            /* Storage for y coordinate of response vector. */
};
struct response *resp;

/*****************************************************************
 *                                                               *
 *       Openin(fname)                                           *
 *       Entry:  Input file name is passed as an arg.            *
 *       Exit:   Pointer to the opened file is returned; a       *
 *               null value is returned if the open was          *
 *               unsuccessfull.                                  *
 *       Calls:  none.                                           *
 *       Called by:      ?                                       *
 *       Calling sequence:       returnval=openin('fname');      *
 *                                                               *
 *****************************************************************/

openin(fname){
FILE *opencode, *fopen();


/*       Open for file read, check value returned for no good...          */

         if ((opencode=fopen(fname,'r'))==NULL){
                 error('ERROR-Can't open input file\n');
         }
         return(opencode);
}
```

```
 *                                                                        *
 *      Openout(filename);                                                *
 *      Entry:  Output file name is passed to the routine as an arg.      *
 *      Exit:   Returns a pointer to the opened file; pointer has null    *
 *              value if open was unsuccessfull.                          *
 *      Calls:  none.                                                     *
 *      Called by: ?                                 ORIGINAL PAGE IS     *
 *      Calling sequence:       x=(openout(fname));  OF POOR QUALITY      *
 *                                                                        *
 *************************************************************************/

openout (fname){
FILE *opencode, *fopen();           /* Specify pointers to (opened) file, and
                                       open macro.                          */

/* Open file for write only, then test for null returned value              */

        if ((opencode=fopen(fname,"w"))==NULL){
                error("ERROR-Can't open output file");
        }

        return(opencode);
}
#include "/usr/include/stdio.h"

/* BLOCK: subroutine to perform block-level tasks

        Entry:  List of entry arguments and conditions goes here.
        Exit:   List of exit arguments and conditions goes here.
        Calls:  Stimvect, Instructions,iti.
        Called by:      Main.                                               */

block(count)
int count;      /* Current block number;                                    */
{

extern struct par;      /* Declare pointers to parameter and data           */
extern struct vect;     /* structures.                                      */
int trialnum;


        /* Calculate stimulus vectors, store in arrau...                    */
        stimvect(count);

        respcrit(count);        /* Calculate response criterion.            */

        if (par->inst[count])   /* Send instructions if requested.          */
        {
                instructions(count);
        }

        /* Execute 1 block of trials.                                       */
        for (trialnum=0;trialnum<(par->numtrials[count]);++trialnum)
        {
                iti(count);     /* Wait for proper inter-trial interval.*/
                trial(count,trialnum);
        }
        return;
}       /* End of block routine.                                            */

/* Power (x,n): Routine to raise x to the nth power. This was
shamelessly taken from K+R, p23. The limitations are as follows:
integers only for both mantissa and exponent, no negative numbers.
*/
```

```
int x,n;
{
        int i,p;

        p=1;
        for (i=1;i<=n;++i)
                p = p * x;
        return (p);
}

#include"/usr/include/stdio.h"
#include "../c/cdev/subs/defaults.h

/**********************************************************************
 *                                                                    *
 *                           Respcrit(x)                              *
 *      This routine calculates the response criterion for the        *
 * current block. The response criterion is a measure of how close    *
 * a sample of the subject reponse is to the stimulus vector          *
 * coordinates. It consists of an integer which gives a deviation     *
 * range against which all collected subject response samples are     *
 * compared. The response criterion can take on 1 of three values     *
 * (small, medium, or large) in any given block. The actual integer   *
 * value is logarithmically related to the stated criterion.          *
 *                                                                    *
 * Entry:         Current block number passed as integer.             *
 * Exit:          RC range stored in *vect->RC as int.                *
 * Calls:         None.                                               *
 * Called by:     Block.                                              *
 *                                                                    *
 **********************************************************************/

respcrit(count)
int count;
{
        extern struct *par;
        extern struct *vect;

        /* Parse RC for current block.                                  */
        switch (*par.rescrit[count])
        {
                case 'S':                /* Small target area.
 *
                        *vect->rc=RCSMALL;
                        break;
                case 'M':                /* Medium target area.
*/
                        *vect->RC=RCMEDIUM;
                        break;
                case 'L':                /* Large target area.
*/
                        *vect->RC=RCLARGE;
                        break;
        }
}       /* End of respcrit.                                             */
#include "/usr/include/stdio.h"
#include "../c/cdev/subs/defaults.h"

/**********************************************************************
 *                                                                    *
 *                           Storedata()                              *
 *      This routine stores all sampled subject response data in      *
 * a disk output file previously opened for writing. This subroutine  *
 * is called at the end of each block, and saves the parameters data  *
 * at the head of the output file.
* Entry:        Integer passed containing current block number.          *
* Exit:         Current response data stored on output buffer.           *
* Calls:        None.                                                    *
* Called by:    Main.                                                    *
*                                                                        *
*************************************************************************/

storedata(count,datafile)
int count,*datafile;
{
extern struct *par;

/* If first call, store parameters information...                       */
        if (count == 1)
        {
                fprintf(datafile,"************** Response force experiment");
                fprintf(datafile,"  ********************\n\n\n");
        }
                fprintf(datafile,"                 Experimental parameters:\n\n");
                fprintf(datafile,"Instructions [%d]=%d\n",count,(*par).inst[coun
t]);
                fprintf(datafile,"Vector/scalar[%d]=%c\n,",count,*par.vs[count]
;
                fprintf(datafile,"Number of trials for block #%d =%d\n,",count,*
par.numtrials[count]);
                fprintf(datafile,"Feedback for block #%d =%c\n,",count,*par.fdbk
[count]);
                fprintf(datafile,"Response criterion for block #%d =%c\n,",count
,*par.rescrit[count]);
                fprintf(datafile,"Inter-trial interval for block #%d =%d\n,",cou
nt,par.iti[count]);
                fprintf(datafile,"Warning for block #%d =%d\n",count,*par.warn[c
ount]);
                fprintf(datafile,"Response sampling rate for block #%d =%d\n,",c
ount,*par.rsr[count]);
                fprintf(datafile,"Max time per trial in block #%d =%d\n,",count,
*par.tet[count]);
                fprintf(datafile,"Min display update rate for block #%d =%d\n,",
count,*par.dur[count]);
                fprintf(datafile,"\n\n\n,");
}
#include "/usr/include/stdio.h"
#include "../c/cdev/subs/defaults.h"
#include "../c/cdev/subs/storage.h"

/************************************************************************
*                                                                        *
*               Parameters(count)                                        *
*       Subroutine to prompt and set experimental parameters.            *
* This subroutine will, when called at the head of the experiment,       *
* allow the operator to set all parameters for the current experiment.   *
* It will do this by first prompting for the total number of             *
* experimental  blocks, collecting parsmeters for the first block,       *
* and offering as defaults for each subsequent block those               *
* parameters selected by the operator for block #1. At the end of        *
* the selection process for the last block the operator will be given    *
* the oportunity to change any parameter he has selected in an           *
* interactive dialog.                                                    *
*                                                                        *
* Entry:        None.                                                    *
* Exit:         Experimental parameters for all blocks set to values     *
*               selected by operator.                                    *
* Calls:        List of routines calleo by parameters goes here.         *
* Called by:    Block.                                                   *
*                                                                        *
```

ORIGINAL PAGE IS OF POOR QUALITY

```
parameters()
        int err;                        /* Define error trap for dialog.       */
        int resp;                       /* Space for response                  */
        int bcount;                     /* Temp block counter.                 */


/* Prompt for total block count for this experiment.                           */
        numblkset();

        for (bcount=1; bcount<= *par.numblks; bcount++)
        {
/* For block 1, no choice but to set all parameters...                         */
                if (bcount==1)
                {
                        promptnset(bcount);
                }
                else
                {
                        do
                        {
                        err=NO;
                        /* Give the option to use parameters already set*/
                        printf("\n\n\nReady to set parameters for block #");
                        printf("%d.\nDo you wish to use the parameters set",bcou
nt);

                        printf(" in block #1 \nfor the current block (y/n) ");
/* Offer default answer...                                                     */
                        printf("[%c]? ",(DEFPARSET?'Y':'N'));

                        resp=getchar();
                        switch(resp)
                        {
                                case '\n':
                                        printf("\n%d\n\n",DEFPARSET);
                                        if (DEFPARSET)
                                        {       /* Set to YES                  */
                                                defaultset(bcount);
                                        }
                                        else
                                        {       /* Set to NO                   */
                                                promptnset(bcount);
                                        }
                                        break;


                                case'Y':
                                case 'y':
                                        printf("Y\n\n\n");
                                        defaultset(bcount);
                                        break;
                                case 'N':
                                case 'n':
                                        printf("N\n\n\n");
                                        promptnset(bcount);
                                        break;
                                default:
                                        error("ERROR-Answer Y or N or <cr>\n");
                                        err=YES;
                                        break;
                        }
                } while(err);   /* Loop 'til no more incorrect responses.*/
                }
        }
```

ORIGINAL PAGE IS
OF POOR QUALITY

```
#include '/usr/include/stdio.h'
#include '../c/cdev/subs/defaults.h'
#include '../c/cdev/subs/storage.h'

/************************************************************************
*                                                                       *
*                       promptnset(block)                               *
*       This routine will set the eperimental parameters to those      *
* values selected by the operator in an interactive dialog. It is      *
* called with the number of the current block as an argument, and      *
* prompts the operator for each parameter to be set in that block.     *
*                                                                       *
* Entry:        Current block number passed for internal use.           *
* Eit:          None.                                                   *
* Calls:        instset, vsset, numtrialsset, dispcolorset, rsrset,     *
*               tptset, durset, warnset, rescritset.                    *
* Called by:    parameters()                                            *
*                                                                       *
************************************************************************/

promptnset(block)
{
                if (block == 1) /* Set display colors for block 1 only. */
                {
                        colorset(1);    /* Set color of display vectors.*/
                }
                warnset(block); /* Set trial onset warning.             */
                printf('In prompt, warn[%d]=%d\n',block,*par.warn[block]);

                itiset(block);  /* Set inter-trial interval.  .         */
                printf('In prompt, iti[%d]=%d\n',block,*par.iti[block]);

                tptset(block);  /* Set time per trial.                  */
                printf('In prompt, tpt[%d]=%d\n',block,*par.tpt[block]);

                rsrset(block);  /* Set response sampling rate.          */
                printf('In prompt, rsr[%d]=%d\n',block,*par.rsr[block]);

                instset(block); /* Call instruction set routine.        */
                printf('In prompt, inst[%d]=%d\n',block,*par.inst[block]);

                vsset(block);   /* Call vect/scalar set routine.        */
                printf('In prompt, vs[%d]=%c\n',block,*par.vs[block]);

                fdbkset(block); /* Call feedback mode set routine.      */
                printf('In prompt, fdbk[%d]=%c\n',block,*par.fdbk[block]);

                rescritset(block);      /* Call response criterion set routine.
*/
                numtrialsset(block) ;/*  Call trial count set routine.  */
                printf('In prompt, numtrials[%d]=%d\n',block,*par.numtrials[bloc
k]);
}

#include '/usr/include/stdio.h'
#include '../c/cdev/subs/defaults.h'
#include '../c/cdev/subs/storage.h'

/************************************************************************
*                                                                       *
*                       numblkset()                                     *
*       This routine will prompt the user for the total number         *
* of experimental trial blocks to be included in this experiment.      *
* assigns to the block counter a default value stored in the          *
* DEFNUMBLKS member of the defaults.h file. Note that the block       *
* count entered can not exceed the value stored in the constant       *
* MAXBLOCKS (also in the defaults.h file).                            *
*                                                                     *
**********************************************************************/

numblkset()
{

        int err,maxbytes,resp;/* Define error trap for dialog.           */
        int count;
        extern struct par;
        char buf[100];          /* Generalized character buffer.          */

        maxbytes = 80;          /* Define maximum input line length.      */

/* Prompt for total block count for this experiment.                      */

        do
        {
        err=NO;
                printf("\nEnter total number of trial blocks in this experiment
");
/* Offer default option.                                                  */
                printf("[%d]: ",DEFNUMBLKS);

                resp=getint(buf,maxbytes);
                if ( resp == -1 )       /* If non-integer response ...    */
                {
                        if ( buf[0] == '\n')    /* <cr> means default.    */
                        {
                                *par.numblks=DEFNUMBLKS;
                                printf("%d\n",DEFNUMBLKS);
                                err=NO;
                        }
                        else            /* Else bad non-integer answer.   */
                        {
                                error("ERROR-Response must be numeric or <cr>\n\
n");

                                err=YES;
                        }
                }
                else                    /* Else an integer response.      */
                {
                        if ( resp > MAXBLOCKS ) /* Response too large.    */
                        {
                                error("ERROR-Response must be less than ");
                                printf("%d blocks\n\n",MAXBLOCKS);
                                err=YES;
                        }
                        else            /* Else everything ok.            */
                        {
                                *par.numblks=resp;
                                err=NO;
                        }
                }

        }       while(err);

printf("Response stored in numblks=%d\n",*par.numblks);
}       /* End of numblkset routine.                                      */


/***********************************************************************
```

ORIGINAL PAGE IS
OF POOR QUALITY

```
*       Routine to set instruction flag for current block.                  *
* This routine is called with the current block number as an argument,      *
* and prompts the operator for instruction display. The allowable           *
* responses are y(yes), n(no), and carriage return, which can mean          *
* either yes or no, depending upon the setting of the flag in               *
* defaults.h. The inst[] member of the global structure par is then         *
* set to reflect the selection.                                             *
*                                                                           *
****************************************************************************/
instset(count)
{
        extern struct par;
        int err;                        /* Define error trap for dialog.        */
        int resp,bcount;                /* Space for response                   */

        do {    /* Loop 'til correct response is received                       */
        err=NO; /* Begin dialog with no errors.                                 */
                printf("\nDisplay subject instructions for block #%d? ",count);
                printf("(y or n)[%c]",DEFINST?'y':'n');

/* Collect response and test.                                                   */
                resp=getchar();
                switch (resp){

                        case '\n':      /* Default response                     */
                                *par.inst[count]=DEFINST;
                                printf("%c\n",DEFINST?'y':'n');
                                break;

                        case 'y':       /* yup...                               */
                        case 'Y':
                                *par.inst[count]=YES;
                                err=NO;
                                break;

                        case 'n':       /* Nope.                                */
                        case 'N':
                                *par.inst[count]=NO;
                                err=NO;
                                break;

                        default:        /* All other responses.                 */
                                err=YES;
                                error("\n\nERROR-Allowable responses are y or n
\n");
                                break;
                        }
        }       while (err);    /* Loop 'till no err.                           */

        printf("Response stored in *par.inst[%d]=%d\n",count,*par.inst[count]);
}       /* End of instset routine.                                              */

/****************************************************************************
*                                                                           *
*                       vsset()                                             *
*       This routine will prompt thr operator for selection of              *
* Either vector or scalar mode operation for the current block.             *
* The allowable responses are v or V(vector mode), s or S(scalar mode),     *
* or carriage return, whose meaning is determined by the status of          *
* the flag DEFVS in the file DEFAULTS.H. If the vector mode is              *
* selected, both the magnitude and direction of the stimulus and            *
* and response vectors can be changed, while in the scalar mode only        *
*                                                                          *
***************************************************************************/
vsset(count)
{

        extern struct par;
        int err;                /* Define error trap for dialog.          */
        int resp,bcount;        /* Space for response                     */

        do {    /* Loop 'til correct response is received                 */
                err=NO;
/* Prompt for vect/scalar mode                                            */
                printf("\n\nSelect vector or scalar mode for block #%d(v=vect, s
=sclr)",count);
/* Print character corresponding to default selection.                    */
                printf("[%c]",DEFVS?'v':'s');
/*      Collect response                                                  */
                resp=getchar();
                switch(resp){
                        case '\n':      /* Default selection              */
                                *par.vs[count]=DEFVS;
                                putchar(DEFVS?'V':'S');
                                break;
                        case 'v':       /* Vector selection               */
                        case 'V':
                                *par.vs[count]=1;
                                err=NO;
                                break;
                        case 's':       /* Scalar mode selection          */
                        case 'S':
                                *par.vs[count]=0;
                                err=NO;
                                break;

                        default:        /* Garbage response               */
                                error("\nERROR-Allowable responses are <cr>, v,
or s.\n\n");
                                err=YES;
                                break;
                        }
        }       while (err);    /* ...'til they get it right!             */

        printf("Response stored in =par.vs[%d]=%c\n",count,(*par.vs[count]?'V':'
S'));
}       /* End of vsset routine.                                          */


/**************************************************************************
*                                                                         *
* Prompt for and collect info on subject response criterion.              *
* The RC is used to specify the size of a 'target' area on the display    *
* screen within which a response vector would be scored as a 'hit'.       *
* Allowable responses to the RC prompt are s or S(small target), m or     *
* M(medium size target), l or L(large size target), and <cr>, whose       *
* meaning is assigned by the constant DEFRESCRIT in the defaults.h        *
* file. The small, nedium, and large qualifiers are in relative terms,    *
* and are logarithmically related.                                        *
*                                                                         *
**************************************************************************/

rescritset(count)
{

        extern struct par;
        int err;                        /* Define error trap for dialog.  */
        do
        {
        err=NO;
        printf("\n\nSelect response criterion for block #%d\n",count);
        printf(" (s=small, m=medium, l=large) [%c]: ",DEFRESCRIT);

        resp=getchar();
        switch (resp)
        {

                case '\n':      /* Default response.                            */
                        printf("%c\n\n",DEFRESCRIT);
                        *par.rescrit[count]=DEFRESCRIT;
                        break;
                case 's':       /* Small area.                                  */
                case 'S':
                        *par.rescrit[count]='S';
                        err=NO;
                        break;
                case 'm':       /* Medium area.                                 */
                case 'M':
                        *par.rescrit[count];
                        err=NO;
                        break;
                case 'l':       /* Large area.                                  */
                case 'L':
                        *par.rescrit[count]='L';
                        err=NO;
                        break;
                default:        /* All others...                                */
                        error("\nERROR-Allowable responses are <cr>, s,m,or l\n\
n");
                        err=YES;
                        break;
        }
}       while (err);


        printf("Response stored in *par.rescrit[%d]=%c\n",count,*par.rescrit[cou
nt]);
}       /* End of rescritset routine.                                   */





/***************************************************************************
 *                                                                          *
 *                      numtrialsset()                                      *
 *      This routine will prompt the user for the number is trials          *
 * to be included in the current block. A default valued ( stored in        *
 * DEFAULTS.H under DEFNUMTRIALS) is offered, and a response is taken.      *
 * This integer value is then stored in the numtrials member of the         *
 * global struct par.                                                       *
 *                                                                          *
 ***************************************************************************/

numtrialsset(count)
{

        char buf[100];                  /* General use character buffer.       */
        int err;                        /* Define error flag for dialog.       */
        do
        {
        err=NO;
                printf("\n\nEnter number of trials for block #%d:",count);
/* Insert default option.                                               */
                printf("[%d]",DEFTPBLK);

                resp=getint(buf,maxbytes);
                if (resp == -1)         /* -1 means newline, or default...*/
                {
                        if (buf[0]=='\n')
                        {
                                *par.numtrials[count]=DEFTPBLK;
                                printf("%d\n",DEFTPBLK);
                                err=NO;
                        }
                        else    /* Else an error                        */
                        {
                                error("\n\nERROR-Response must be numeric\n");
                                err=YES;
                        }
                }
                else
                {
                        if ( resp > MAXTRIALS )
                        {
                                error("\n\nERROR-Response must be less than ");
                                printf("%d trials\n\n",MAXTRIALS);
                                err=YES;
                        }
                        else
                        {
                        *par.numtrials[count]=resp;
                        err=NO;
                        }
                }
        }       while (err);

        printf("\nResponse stored in numtrials[%d]=%d\n",count,*par.numtrials[co
unt]);

}       /* End of numtrailsset routine.                                 */



/***********************************************************************
*                                                                      *
*                       fdbkset(count)                                 *
*       This routine will prompt for the type of subject feedback      *
* desired for the current block. Allowable answers include             *
* n or N(no feedback), b or B(Binary feedback, meaning correct         *
* incorrect), a or A(Running average, which is an error count          *
* that is cumulative over all samples in a trial), and <cr>, which     *
* gives the response determined by the setting of the DEFFDBK flag     *
* in the defaults.h file.                                              *
*                                                                      *
***********************************************************************/

fdbkset(count)
{

        extern char *fdbk[];    /* Assign storage for feedback mode flag*/
        int err;                /* Define error trap for dialog.        */
        int resp,bcount;        /* Space for response                   */
        do{
        err=NO;
/* Prompt and collect response for feedback mode.                         */

                printf("\n\nSelect subjec  feedback mode:\n\nb=binary (correct/i
ncorrect),\n");
                printf("a=Running average (Cumulative error summed over all");
                printf("samples in a trial),\n ");
                printf("n=No subject feedback.");
                printf("\nEnter selection [%c]: ",DEFRFM);
                resp=getchar();
                switch (resp)
                {

                        case '\n':      /* Default response.              */
                                printf("%c\n",DEFRFM);
                                *par.fdbk[count]=DEFRFM;
                                break;
                        case 'b':       /* Binary feedback.               */
                        case 'B':
                                *par.fdbk[count]='B';
                                err=NO;
                                break;
                        case 'a':       /* Running average.               */
                        case 'A':
                                *par.fdbk[count]='A';
                                err=NO;
                                break;
                        case 'n':       /* No feedback.                   */
                        case 'N':
                                *par.fdbk[count]='N';
                                err=NO;
                                break;
                        default:
                                error("\n\nERROR-Response must be <cr>,a,b,or n\
n\n");
                                err=YES;
                                break;
                }
        }       while (err);
        printf("\nResponse stored in fdbk[%d]=%c\n\n",count,*par.fdbk[count]);

}       /* End of fdbkset routine.                                        */


#include "/usr/include/stdio.h"
#include "../c/cdev/subs/defaults.h"
#include "../c/cdev/subs/storage.h"


/*************************************************************************
*                       colorset(count)                                  *
*                                                                        *
* Prompt for and collect info on subject display vector colors.          *
* The parameter collected here controls the colors of the stimulus      *
* and response vectors on the subject display scope. Allowable          *
* colors include red, green, and yellow; all mutually exclusive         *
* permutations are permitted. The color combinations and associated     *
* codes are as follows:                                                  *
*       0)      No colors for either stimulus or response.               *
*       1)      Green stimulus, red response.                            *
*       2)      Green stimulus, yellow response.                         *
*       3)      Red stimulus, green response.                            *
*       4)      Red stimulus, yellow response.                           *
*       6)      Yellow stimulus, red response.                                   *
*                                                                                *
*******************************************************************************/
```

```
colorset()
{

        extern int *dispcolor;/* Assign storage for display colors.      */
        int resp,count,err;     /* Space for response              */

        do
        {
        err=NO;
        printf("Select subject display colors: Allowable combinations are\n");
        printf("0)          No colors for either stimulus or response vector\n");
        printf("1)          Green stimulus, red response.\n");
        printf("2)          green stimulus, yellow response\n");
        printf("3)          Red stimulus, Green response\n");
        printf("4)          Red stimulus, Yellow response\n");
        printf("5)          Yellow stimulus, Green response\n");
        printf("6)          Yellow stimulus, Red response\n\n");
        printf("Enter code corresponding to desired colors [%d]: ",DEFDISPCOLOR)
;

        resp=getchar();
        switch (resp)
        {

                case '\n':      /* Default combination.                  */
                        printf("%d\n\n",DEFDISPCOLOR);
                        *par.dispcolor=DEFDISPCOLOR;
                        err=NO;
                        break;
                case '0':       /* No colors.                            */
                        *par.dispcolor=0;
                        err=NO;
                        break;
                case '1':       /* Green + Red.                          */
                        *par.dispcolor=1;
                        err=NO;
                        break;
                case '2':       /* Green + Yellow.                       */
                        *par.dispcolor=2;
                        err=NO;
                        break;
                case '3':       /* Red + Green.                          */
                        *par.dispcolor=3;
                        err=NO;
                        break;
                case '4':       /* Red + Yellow.                         */
                        *par.dispcolor=4;
                        err=NO;
                        break;
                case '5':       /* Yellow + Green.                       */
                        *par.dispcolor=5;
                        err=NO;
                        break;
                case '6':       /* Yellow + Red.                         */
                        *par.dispcolor=6;
                        err=NO;
                        break;
                default:        /* All others.                           */
                        error("\n\nERROR-Response must be <cr>, 1-6\n\n");
                        err=YES;
```

```
}        while (err);

printf("\n\n");

}        /* End of colorset routine.                                 */


/************************************************************************
*                                                                       *
*                        warnset(count)                                 *
*                                                                       *
*        This routine will prompt for and set the code for the          *
* selection of the subject trial-onset warning. Allowable responses     *
* include 1, which gives a .5 second warning tone on a system           *
* Sonalert beeper, 0, which gives a green square on the subjects'       *
* display scope for the same length of time, and <cr>, whose            *
* meaning is governed by the status of the DEFWARN flag stored in       *
* the defaults.h file.                                                  *
*                                                                       *
************************************************************************/

warnset(count)
{



        extern int *warn[];       /* Assign storage for warning tone.     */
        int err;                  /* Define error trap for dialog.        */
        int resp,bcount;          /* Space for response                   */

        do
        {
        err=NO;
        printf("Select Trial-onset warning tone. Responses include\n");
        printf("0=green square on display scope, 1=Sonalert tone.\n");
        printf("Enter 0 or 1 [%d]: ",DEFWARN);

        resp=getchar();
        switch (resp)
        {
                case '\n':        /* Default response.                    */
                        printf("%d\n\n",DEFWARN);
                        *par.warn[count]=DEFWARN;
                        err=NO;
                        break;
                case '0':         /* Green square.                        */
                        *par.warn[count]=0;
                        err=NO;
                        break;
                case '1':         /* Sonalert tone.                       */
                        *par.warn=1;
                        err=NO;
                        break;
                default:          /* All others.                          */
                        error("\nERROR-Response must be <cr>, 0, or 1\n\n");
                        err=YES;
                        break;
        }
}       while (err);

printf("\n\n");

        printf("Response stored in *par.warn[%d]=%d\n",count,*par.warn[count]);
}       /* End of warnset routine.                                    */
```

```
/************************************************************************
*                                                                       *
*                       itiset(count)                                   *
*       This routine will prompt for and set the inter-trial            *
* interval for the trial block specified. Allowable responses to        *
* the prompt include <cr>, which sets a default number of seconds       *
* whose value is stored in the defaults.h file, and any digit(s)        *
* less than the pre-determined maximum limit, whose value is also       *
* in the defaults.h file. The response to this prompt is stored in      *
* the .iti member of the global structure *par.                         *
*                                                                       *
************************************************************************/
```

```
itiset(count)
{


        char buf[100];          /* General use character buffer.        */
        extern int *iti[];      /* Assign storage for inter trial interval*/
        int err;                /* Define error trap for dialog.        */
        int resp,bcount,maxbytes;       /* Space for response
*/
do
{
        maxbytes = 80;  /* Define maximum input character string length */

        printf("Enter desired inter-trial interval length for block #%d: [%d]?
,count,DEFITI);
        resp=getint(buf,maxbytes);
        if (resp==-1)
        {
/* check for default answer.                                            */
                if (buf[0] == '\n')
                {
                        printf("%d\n",DEFITI);
                        *par.iti[count]=DEFITI;
                        err=NO;
                }
                else    /* else an error if 1 character (other than cr) */
                {
                        error("\nERROR-Response must be numeric\n\n");
                        err=YES;
                }
        }
        else    /* Else an integer response.                            */
        {
                if (resp > MAXITI)      /* Check for iti too long...    */
                {
                        error("\nERROR-ITI must be less than ");
                        printf("%d seconds\n\n",MAXITI);
                        err=YES;
                }
                else    /* Else everything ok...                        */
                {
                        *par.iti[count]=resp;
                        err=NO;
                }
        }
}
while (err);
```

```
        printf("Response stored in *par.iti[%d]=%d\n",count,*par.iti[count]);
}       /* End of itiset routine.                                              */
```

```
/*************************************************************************
*                                                                        *
*                         tptset(count)                                  *
*       This routine collects data on the desired maximum length of      *
* time a trial should last;it is the point at which the program will     *
* interrupt the subject response/display update cycle.                   *
* Allowable responses to the prompt include <cr>, which sets a default   *
* value which is set in the file defaults.h, and any integer value       *
* less than an absolute maximum, which is also stored in defaults.h      *
*                                                                        *
*************************************************************************/

tptset(count)
{



        extern int *tpt[];      /* Assign storage for trial time.      */
        char buf[100];          /* General use character buffer.       */
        int err;                /* Define error trap for dialog.       */
        int resp,bcount,maxbytes;       /* Space for response
*/
        maxbytes = 80;  /* Define maximum input character string length */

do
{
        err=NO;
        printf("Enter Subject time limit (in seconds) for trials in block #%d: [
%d]",count,DEFTPT);

        resp=setint(buf,maxbytes);
/* Check for default response.                                         */
        if ( resp==-1)
        {
                if (buf[0] == '\n')
                {
                        printf("%d\n",DEFTPT);
                        *par.tpt[count]=DEFTPT;
                        err=NO;
                }
                else    /* Else bad answer.                             */
                {
                        error("\nERROR-Answer must be numeric.\n\n");
                        err=YES;
                }
        }
        else    /* Else a valid integer response.                       */
        {
                if (resp > MAXTPT)      /* Can't be too big.            */
                {
                        error("\nERROR-response must be less than ");
                        printf("%d seconds\n\n",MAXTPT);
                        err=YES;
                }
                else    /* Else ok to store.                            */
                {
                        *par.tpt[count]=resp;
                        err=NO;
                }
}       while ( err );

printf("\n\n");

        printf("Response stored in *par.tpt[%d]=%d\n",count,*par.tpt[count]);
}       /* End of tptset routine.                                               */
```

ORIGINAL PAGE IS
OF POOR QUALITY

```
/***********************************************************************
*                                                                      *
*                          rsrset(count)                               *
*       This routine will prompt for and set the desired response      *
* sampling rate for the current block. Allowable answers include       *
* <cr>, which sets a default response sampling rate (exact default     *
* set in the file defaults.h), and any integer string which is in      *
* milliseconds, is a valid integer string greater than the minimum     *
* sampling interval, and is an integral multiple of 10 msec.           *
*                                                                      *
***********************************************************************/

rsrset(count)
{


        extern int *rsr[];       /* Assign storage for response
                                 sampling rate.                         */
        char buf[100];           /* General use character buffer.       */
        int err;                 /* Define error trap for dialog.       */
        int resp,bcount,maxbytes;        /* Space for response
*/
        maxbytes = 80;  /* Define maximum input character string length */

do
{
        err=NO;
        printf("Enter desired response sampling rate for block number %d\n",coun
t);
        printf("The sampling rate should be in milliseconds, and should be an \n
");
        printf("integral multiple of 10 milliseconds, which is the minimum.\n");
        printf("Sampling rate =[%d] ",DEFRSR);

        resp=getint(buf,maxbytes);

/* Check for default answer.                                            */
        if ( resp == -1)
        {
                if ( buf[0] == '\n')    /* carriage return ok.          */
                {
                        printf("%d\n",DEFRSR);
                        *par.rsr[count]=DEFRSR;
                        err=NO;
                }
                else    /* Else an error.                               */
                {
                        error("\nERROR-Response must be numeric\n\n");
                        err=YES;
                }
        }
        else    /* Else a valid integer response.                       */
        {
                if (resp < MINRSR )     /* Too low...                   */
                {
                        error("\nERROR-Response must be greater than ");
                }
                else
                {
                        if (( resp % 10 ) != 0 )        /* Multiple of 10?
*/
                        {
                                error("\nERROR-Response must be multiple of 10\n
\n");
                                err=YES;
                        }
                        else    /* Else take a number.
*/
                        {
                                *par.rsr[count]=resp;
                                err=NO;
                        }
                }
        }
}       while ( err );

printf("\n\n");
        printf("Response stored in *par.rsr[%d]=%d\n",count,*par.rsr[count]);
}       /* End of rsrset routine.                                           */




/***********************************************************************
*                                                                      *
*                       Defaultset(blocknum)                           *
*                                                                      *
*       This routine will set the parameters for a block whose         *
* number was passed as an argument to those values set by the          *
* operator for block #1. The routine utilizes the global structure     *
* containing the parameters, and returns nothing.                      *
*                                                                      *
* Entry:        Current block number.                                  *
* Exit:         None.                                                  *
* Calls:        None.                                                  *
* Called by:    Parameters.                                            *
* History:      11/xx/83:Complete and working.                         *
*                                                                      *
***********************************************************************/

defaultset(blknumber)
{
#include "../c/cdev/subs/defaults.h"
#include "../c/cdev/subs/storage.h"


        *par.inst[blknumber]=*par.inst[1];      /* Set instruction flag */
        *par.numtrials[blknumber]=*par.numtrials[1];
        *par.vs[blknumber]=*par.vs[1];
        *par.fdbk[blknumber]=*par.fdbk[1];
        *par.rescrit[blknumber]=*par.rescrit[1];
        *par.iti[blknumber]=*par.iti[1];
        *par.warn[blknumber]=*par.warn[1];
        *par.rsr[blknumber]=*par.rsr[1];
        *par.tpt[blknumber]=*par.tpt[1];
        *par.dur[blknumber]=*par.dur[1];

}       /* End of defaultset routine.                                       */
#include"../c/cdev/subs/defaults.h"
```

```
/* routine to display contents of global parameter structure.   */

disp(count)
{
extern struct par

printf("Instructions [%d]=%d\n",count,*par.inst[count]);
printf("Vector/scalar[%d]=%c\n",count,*par.vs[count]);
printf("Number of trials for block #%d =%d\n",count,*par.numtrials[count]);
printf("Feedback for block #%d =%c\n",count,*par.fdbk[count]);
printf("Response criterion for block #%d =%c\n",count,*par.rescrit[count]);
printf("Inter-trial interval for block #%d =%d\n",count,*par.iti[count]);
printf("Warning for block #%d =%d\n",count,*par.warn[count]);
printf("Response samp rate for block #%d =%d\n",count,*par.rsr[count]);
printf("Max time per trial in block #%d =%d\n",count,*par.tpt[count]);
printf("Min display update rate for block #%d =%d\n",count,*par.dur[count]);
printf("\n\n\n");
}
#include "/usr/include/stdio.h"
/*
ERROR(): Error subroutine. This routine is a generalized error reporting
subroutine.

Entry:          The entry argument is a phrase to be printed on the console.
Exit:           Argument directed to standard output.
Calls:          None.
Called by:      Any other routine.

*/

error (ptr)
char *ptr;

{
        int k;
        k=0;
        while (ptr[k++])        /* Trailing null in line will terminate routine*
/
                putchar (ptr[k-1]);
}

#include "/usr/include/stdio.h"
#include "../c/cdev/subs/defaults.h"

/**********************************************************************
*                                                                     *
*                         Getint(x,y)                                 *
*       Subroutine used to gather a line of input from the console,   *
* evaluate it, and return an integer corresponding to the             *
* collective value of the digits entered.                             *
*                                                                     *
* Entry:          Maximum line length passed as int, also pointer to  *
*                 character buffer.                                   *
* Exit:           Int returned giving value of digit(s) entered, or -1 *
*                 if an error occurred.                               *
* Calls:          Readin, error.                                      *
* Inclusions:     Calling routine must include the stdio header.      *
* Notes:          This routine should be run in a 'mode raw, -ec'    *
*                 environment.                                        *
**********************************************************************/

getint(buf,maxchars)
char *buf;
int maxchars;
{
        count=0;                /* Initialize running total.                */

/* Get a line of characters, check to see if EOF encountered in line.   */
        if ((x=readin(buf,maxchars))!=NULL)
        {
                if (x== 1)
                {
                        err=YES;
                }
                else
                {
                err=NO;
/* Step thru buffer returned from last character entered to first.      */
                        for (digitcount=x-2; digitcount>=0; --digitcount)
                        {
/* Check for valid digit.                                               */
                                if ((isdigit(buf[digitcount]))==NULL)
                                {
                                        err=YES;
                                }
                                else
/* If good digit, scale to proper power of ten for digit place, and add to
running total.                                                          */
                                        count+=((buf[digitcount]-'v')*(power(10,
(x-digitcount-2))));
                        }
                }
        }
        if (err)
                return (-1);    /* Signal error.                        */
        else
                return (count); /* Or return running total.             */
}



/***********************************************************************
*                                                                      *
*                         numblkset()                                  *
*       This routine will prompt the user for the total number        *
* of experimental trial blocks to be included in this experiment.     *
* The response may be 1 or more digits, or a carriage return, which   *
* assigns to the block counter a default value stored in the          *
* DEFNUMBLKS member of the defaults.h file. Note that the block       *
* count entered can not exceed the value stored in the constant       *
* MAXBLOCKS (also in the defaults.h file).                            *
*                                                                      *
***********************************************************************/

numblkset()
{

        int err;                /* Define error trap for dialog.        */
        int resp;               /* Space for response                   */
        int bcount;
        extern int *numblks;    /* Assign storage for total block count */


/* Prompt for total block count for this experiment.                    */

        do{
        err=NO;
                printf("\nEnter total number of trial blocks in this experiment
");
/* Offer default option                                                 */
```

```
        resp=getchar();
        switch (resp){

            case'\n':       /* Default response.            */
                printf("%d\n",DEFNUMBLKS);
                *par.numblks=DEFNUMBLKS;
                break;
            default:
                if ((isdigit(resp))==NULL)
                {
                    error("\nERROR-Response must be numerica
l\n");

                    err=YES;
                    break;
                }
                else{
                    if ((resp-'0')>MAXBLOCKS)
                    {
                        error("\nERROR-maximum allowable
 number of blocks is ");

                        printf("%d\n",MAXBLOCKS);
                        err=YES;
                        break;
                    }
                    else{
                        printf("%c\n",resp);
                        *par.numblks=resp-'0';
                        break;
                    }
                }
        }
    }   while(err);

}   /* End of numblkset routine.                        */

#include "/usr/include/stdio.h"
#include "../c/cdev/subs/defaults.h"

/*************************************************************************
*                                                                       *
*                   Function readin(buf,maxbytes)                       *
*   This routine will read a line of maximum length maxbytes from       *
* the standard input, and will place it in the buffer pointed to by the *
* argument *buf.The routine will return an int with the following       *
* meanings:                                                             *
* 1)If no errors occurred, a count of the number of characters read     *
* is returned.                                                          *
* 2)If EOF was encountered, or an error occurred, a NULL value is       *
* returned.                                                             *
*                                                                       *
* Entry:        A pointer to a character buffer must be passed, as well *
* as an int containing the maximum allowable byte count.                *
* Exit: An int is returned, giving exit status information.             *
* Calls:        getchar                                                 *
* Called by:    Any other routine.                                      *
* Inclusions:   The calling routine must include the stdio.h file.      *
* Notes:        The calling routine should execute in a 'mode raw' .    *
*               environment.                                            *
*                                                                       *
*************************************************************************/

readin(buf,maxbytes)
char *buf;
int maxbytes;
```

```
        int charcount;
        charcount=0;

/* Collect input until End of file encountered (aka control c, ^c)         */
        while((c=getchar())!=EOF)
        {
                *(buf+charcount++)=c;
/* Check for carriage return or maximum buffer count exceeded.              */
                if (c=='\n'||charcount>maxbytes-1)
                {
                        return(charcount);
                }
        }
        return(NULL);   /* Else EOF encountered                             */
}
/*************************************************************************
*                                                                        *
*                       parchange()                                      *
*       This subroutine will handle the mechanics of listing the         *
* parameter block contents, and will oversee any changes made.           *
*                                                                        *
* Entry:        None.                                                    *
* Exit:         None.                                                    *
* Calls:        Dispblk, Blkchange.                    *
* Called by:    Parcheck.                                                *
*                                                                        *
*************************************************************************/


#include "/usr/include/stdio.h"
#include "../c/cdev/defaults.h"


parchange()
{
        extern struct *par;
        int resp,err,blk,maxchar;
        char buf[80];

        maxchar=70;     /* Define maximum line length.                      */

        do      /* Error trapping loop.                                      */
        {
                printf("Enter the number of the block you wish to change :");

                resp=getint(buf,maxchar);
                switch ( resp)
                {

                        case -1:        /* Error generated in getint.
*/
                        case 0:
                                error("ERROR-Response must be numeric and non-ze
ro\n\n");
                                err=YES;
                                break;
                        default:        /* All others valid integer responses.
*/
                                if (resp > par->numblks) /* Response too high.
*/
                                {
                                        error("ERROR-Response must be less than
");
                                printf("%d \n\n",par->numblks);
```

```
                                        else    /* ok to change.
 */
                                        {
                                                disrblk(resp);  /* Display the contents
 of the selected block */
                                                blkchange(resp);        /* Change the pa
 rameters in the selected block */
                                                err=NO;
                                        }
                                        break;
                        }
        }       while ( err );
        return;
}
/******************************disrblk()******************************/

disrblk(){

printf("\nIn disrblk subroutine\n");
return;
}
/*********************************************************************
 *                                                                   *
 *                            parcheck()                             *
 *        This routine handles the correction of experimental        *
 * parameters as part of the parameters routine. In this subroutine, *
 * the operator is prompted for changes to be made. If there are any, *
 * the number of the block to be changed is requested, after which   *
 * a list of the parameters selected for that block is generated.    *
 * Parcheck then prompts for a list of parameters to change,         *
 * displaying each parameter after it is changed. The process is     *
 * repeated at each level until all parameters are correct.          *
 *                                                                   *
 * Entry:          None.                                             *
 * Exit:           None.                                             *
 * Calls:          parchange, error.                                 *
 * Called by:      parameters.                                       *
 *                                                                   *
 *********************************************************************/

#include "/usr/include/stdio.h"
#include "../c/cdev/defaults.h"

parcheck()
{
        extern struct par;
        int block, err, resp,pleasecheck;
        char buf[80];
        pleasecheck = NO;       /* Start off with Non-repetition.         */

        do      /* This loop for cyclic dialog-style checking.            */
        {
                do      /* This loop for response errors.                 */
                {
                        if (pleasecheck)        /* If 2nd time around...*/
                        {
                                printf("\nDo you wish to examine or make changes
  to additional blocks? [%c]: ",(DEFPARCHK?'Y':'N'));
                        }               /* Print different prompt.        */
                        else
                        {
                        printf("\nDo you wish to examine or change any parameter
 s? [%c]: ",(DEFPARCHK?'Y':'N'));
                        }
```

```
/* Examine and react to response.                                      */
                        switch (resp)
                        {
                                case '\n':      /* Default answer.      */
                                        printf("%c\n",(DEFPARCHK?'Y':'N'));
                                        err=NO;
                                        switch (DEFPARCHK) /* Case out possibili
ties */
                                        {
                                                case YES:
                                                        err=NO;
                                                        pleasechk=YES;
                                                        parchange();
                                                        break;

                                                case NO:
                                                        err=NO;
                                                        pleasechk=NO;
                                                        break;

                                                default: /* Bad option */
                                                        err=YES;
                                                        error("ERROR-Bad option
DEFPARCHK in Defaults.h!\n\n");

                                                        break;
                                        }

                                break;

                                case 'n':       /* Don't want to.       */
                                case 'N':
                                        err=NO;
                                        pleasecheck=NO; /* No more.     */
                                        break;
                                case 'y':       /* Yes, please.         */
                                case 'Y':
                                        err=NO;
                                        pleasecheck=YES;
                                        parchange();    /* Check +/or change a b
lock.*/
                                        break;
                                default:        /* All others (Wrong answer).*/
                                        error("ERROR-Response must be y,n,or <cr
>\n\n");

                                        err=YES;
                                        break;
                        }
                }       while ( err );  /* Repeat 'til no response errors*/
        }       while ( pleasecheck );  /* Repeat 'til no more changes requested
 */
        return;
}

#include "../c/cdev/subs/defaults.h
#include "/usr/include/stdio.h"

/*****************************************************************
*                                                                *
*               Blkchange.c                                      *
*       subroutine to execute a parameter change within a        *
* block. This subroutine will, when called with a block          *
* number as an integer argument, prompt for the number of        *
* a parameter to change, display the old value, and prompt       *
* and the routine will exit.                                          *
*                                                                     *
* Entry:          Integer holding number of valid exp block           *
*                 in which changes are requested.                     *
* Exit:           1 parameter is changed.                             *
* Calls:          any of the parameter set routines: Instset,         *
*                 vsset,rsrset,numtrialsset,colorset,tptset,          *
*                 durset,fdbkset,numblksset,warnset,                  *
*                 rescritset,and itiset. Also, error,setint.          *
* Called by:      parchange.                                          *
*                                                                     *
**********************************************************************/

blkchange(block)
{
        int resp,maxchars,err;
        char *responsebuf;
        extern struct *par;

        do      /* Loop 'til no incorrect responses.                          */
        {
                printf("Enter the number of the parameter to be changed [%s] :",
DEFPARNUM);

/* Get and check response.                                                     */
                resp=setint(responsebuf,maxchars);
                switch (resp)
                {
                        case '\n':      /* Default response.                  */
                                printf("%s\n",DEFPARNUM);
                                err=NO;
                                switch (DEFPARNUM) /* Check defaults.         */
                                {
                                        case '1':
                                                instset(block);
                                                break;
                                        case '2':
                                                numblksset;
                                                break;
                                        case '3':
                                                numtrialsset(block);
                                                break;
                                        case '4':
                                                vsset(block);
                                                break;
                                        case '5':
                                                fdbkset(block);
                                                break;
                                        case '6':
                                                rescritset(block);
                                                break;
                                        case '7':
                                                itiset(block);
                                                break;
                                        case '8':
                                                warnset(block);
                                                break;
                                        case '9':
                                                colorset;
                                                break;
                                        case '10':
                                                rsrset(block);
                                                break;
                                        case '11':
                                                tptset(block);
```

ORIGINAL PAGE IS
OF POOR QUALITY

```
                                        case '12':
                                                durset(block);
                                                break;
                                        default:
                                                return;
                                                break;
                                }
                                case '1':
                                        instset(block);
                                        break;
                                case '2':
                                        numblksset;
                                        break;
                                case '3':
                                        numtrialsset(block);
                                        break;
                                case '4':
                                        vsset(block);
                                        break;
                                case '5':
                                        fdbkset(block);
                                        break;
                                case '6':
                                        rescritset(block);
                                        break;
                                case '7':
                                        itiset(block);
                                        break;
                                case '8':
                                        warnset(block);
                                        break;
                                case '9':
                                        colorset;
                                        break;
                                case '10':
                                        rsrset(block);
                                        break;
                                case '11':
                                        tptset(block);
                                        break;
                                case '12':
                                        durset(block);
                                        break;
                                default:
                                        error("\nERROR-Bad selection. Please try
 again\n");

                                        return;
                                        break;
                        }
        }       while (err);    /* Repeat while in error condition.     */
                switch (resp)   /* Display parameter after change.      */
                {
                        case '1':
                                printf("Instruction flag for block #%d=%s\n",blo
ck,*par.inst[block]?'YES':'NO');
                                break;
                        case '2':
                                printf("Number of blocks =%d\n",*par.numblks);
                                break;
                        case '3':
                                printf("Number of trials in block #%d=%d\n",bloc
k,*par.numtrials[block]);
                                break;
                        case '4':
                                printf("Vector/scalar mode for block #%d=%c",blo
                                break;
                        case '5':
                                printf("Feedback mode for block #%d=%d\n",block,
*par.fdbk[block]);
                                break;
                        case '6':
                                printf("Response criterion for block #%d=%c\n",b
lock,*par.rescrit[block]);
                                break;
                        case '7':
                                printf("Inter-trial interval for block #%d=%d se
conds\n",block,*par.iti[block]);
                                break;
                        case '8':
                                printf("Trial-onset warning for block #%d=%s\n",
block,*par.warn[block]?'tone':'square');
                                break;
                        case '9':
                                printf("Subject display colors code =%d\n",*par.
dispcolor);
                                break;
                        case '10':
                                printf("Response sampling rate for block #%d=%d
milliseconds\n",block,*par.rsr[block]);
                                break;
                        case '11':
                                printf("Maximum time for each trial in block #%d
=%d seconds\n",block,*par.tpt[block]);
                                break;
                        case '12':
                                printf("Display update rate for block #%d=%d mil
liseconds\n",block,*par.dur[block]);
                }
        return;
}       /* End of blkchange subroutine.                                     */
/*****************************************************************************
*                                                                             *
*                       dispblk(block)                                        *
*       Subroutine to display values of experimental parameters               *
* associated with a particular block. This routine, when passed               *
* an integer corresponding to the number of an operator-selected              *
* block, will print the name of each parameter in the block, along            *
* with its current value.                                                     *
*                                                                             *
* Entry:        Block number passed as int.                                   *
* Exit:         Parameter values displayed on stdout.                         *
* Calls:        *
* Called by:    Parchange.                                                    *
*                                                                             *
*****************************************************************************/

#include "/usr/include/stdio.h"
#include "../c/cdev/defaults.h"

dispblk(block)
{
        extern struct par;

        printf("\n\n                        Parameters for block #%d:\n\n",block);

        printf("1) Subject instruction display=%c\n",(*par.inst[block]?'Y':'N'))
;
        printf("2) Number of experimental blocks =%d\n",*par.numblks);
        printf("3) Number of trials for block #%d =%d\n",block,*par.numtrials[bl
ock]);
```

ORIGINAL PAGE IS
OF POOR QUALITY

```
        printf("5) Feedback mode for block #%d =%c\n",block,*par.fdbk[block]);
        printf("6) Response criterion for block #%d =%c\n",block,*par.rescrit[bl
ock]);
        printf("7) Inter-trial interval for block #%d =%d\n",block,*par.iti[bloc
k]);
        printf("8) Trial-onset warning for block #%d =%d\n",block,*par.warn[bloc
k]);
        printf("9) Subject display colors are : ");
        switch (*par.dispcolor) /* Print the possible color combinations
*/
                {

                case 0: /* No colors                                        */
                        printf("No colors for either stimulus or response vector
s\n");

                        break;

                case 1: /* green + red                                      */
                        printf("Green stimulus, red response\n");
                        break;

                case 2: /* Green + Yellow                                   */
                        printf("Green stimulus, yellow response\n");
                        break;

                case 3: /* Red + green                                      */
                        printf("Red stimulus, green response\n");
                        break;

                case 4: /* Red + yellow                                     */
                        printf("Red stimulus, yellow response\n");
                        break;

                case 5: /* Yellow + green                                   */
                        printf("Yellow stimulus, green response\n");
                        break;

                case 6: /* Yellow + red                                     */
                        printf("Yellow stimulus, red response\n");
                        break;

                default:        /* All others                               */
                        error("ERROR-Bad DEFDISPCOLOR entry in defaults.h!!\n");
                        break;
                }

        printf("10) Response sampling rate for block #%d =%d\n",block,*par.rsr[b
lock]);
        printf("11) Maximum trial duration for block #%d =%d\n",block,*par.tpt[b
lock]);
        printf("12) Subject display update rate for block #%d =%d milliseconds\n
",block,*par.dur[block]);

}       /* End of dispblk.                                                  */

#include "/usr/include/stdio.h"
#include "../c/cdev/subs/defaults.h"

/*******************************************************************
*                                                                  *
*                       Stimvect()                                 *
*       Routine to calculate stimulus vector coordinates.          *
* This subroutine is called in the block subroutine, and provides  *
* the coordinates of each stimulus vector in a trial block.        *
* * Position to the new vector endpoints.                                 *
*                                                                          *
* Entry:          Int containing current block number, pointer to          *
*                 stimulus vector coordinate storage area.                 *
* Exit:           All stimulus vectors set for current block.              *
* Calls:          Scale,                                                   *
* Called by:      Block.                                                   *
*                                                                          *
***************************************************************************/

stimvect(blk,*vect)
int blk;
struct *vect;
{
/* Calculate the correct number of coordinate pairs for current block.     */
        for (x=0;x<*par.vs[count];++x)
        {
                /* Calculate x and y coordinates separately.               */
                *vect->stimx[x]=scale(x,*vect);
                *vect->stimy[x]=scale(y,*vect);
        }
        return(*vect);
}       /* End of Stimvect routine.                                        */
#include "/usr/include/stdio.h"
#include "../c/cdev/subs/defaults.h"

/***************************************************************************
*                                                                          *
*                            Trial(x,y)                                    *
*       Trial task subroutine.                                             *
* This subroutine executes each of the tasks required to complete 1        *
* trial in the response force experiment. These include sending the       *
* trial onset warning, displaying the stimulus vector, collecting          *
* and evaluating response data, and dispensing subject feedback.           *
*                                                                          *
* Entry:          Current trial number and block number passed as ints.    *
* Exit:           1 Trial executed, data stored in response data area.     *
* Calls:          warning,stimulus,response,compare,feedback               *
* Called by:      Block.                                                   *
*                                                                          *
***************************************************************************/

trial(blknum,tnum)
int tnum,blknum,runave;
{
        extern struct *par;
        extern struct *vect;
        extern struct *resp;
        int timeout,sampcnt,hit;

        /* Send appropriate warning to subject.                            */
        warning(blknum);

        startclk;       /* Start timing sequence.                          */

        stimulus(blknum,tnum);  /* Send stimulus vector.                   */

        sampcnt=0;
        timeout=NO;
        hit=NO;
        runave=0;       /* Initialize cumulative error count.              */
        /* Start response collection: this will stop when either the
        subject scores a hit, or the maximum time alloted has elapsed.     */
        while (( hit==NO ) && ( timeout == NO))
        {
                hit=compare(blknum,tnum,sampcnt,response(blknum,tnum,sampcnt));

                if ((sampcnt%10)==0) /* Dispense attaboy every 10 samples*/
                {
                        /* Dispense feedback as necessary.                 */
                        runave+=feedback(runave,blknum,hit,(sampcnt-10),sampcnt)

                }
                sampcnt +=1;     /* Update sample count.                   */
                stopclk;
        }
        return;
}
```

ORIGINAL PAGE IS
OF POOR QUALITY