SIS: A SORTING INSTRUCTION SIMULATOR

By

JAMES STEPHEN RAMLET

Bachelor of Science

Oral Roberts University

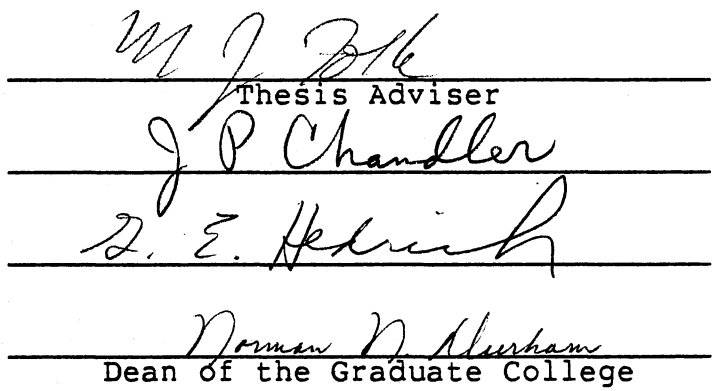
Tulsa, Oklahoma

1977

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
December, 1984

SIS: A SORTING INSTRUCTION SIMULATOR

Thesis Approved:



Thesis Adviser

Dean of the Graduate College

PREFACE

This paper is a discussion of a sorting instruction simulator (SIS). In addition to a description of the program modularity and simulation design, a study of the command language developed for the system is presented. Also included is a user's guide and module catalog.

I would like to express sincere gratitude to my major advisor, Dr. Michael J. Folk for his guidance, motivation, and invaluable help. I am also thankful to Dr. George E. Hedrick for his advisement in the course of this work. A special note of thanks is extended to Dr. John P. Chandler, not only for serving on my graduate committee, but also for his encouragement during my stay at Oklahoma State University.

My wife, Sandy, my parents, my father-in-law, and my mother-in-law deserve my deepest appreciation for their continual support, moral encouragement, and understanding.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

Impetus

This paper is a presentation of a Sorting Instruction Simulator (SIS), a computer simulation model of visual program execution for sorting algorithms. Many problems for beginning programming students relate to program comprehension. The purpose of SIS is to aid the student in his ability to conceptualize the abstraction of a given algorithm with the flow of the program code.

Chapter II is a discussion on how program visualization in computer-assisted instruction (CAI) has evolved.

Chapter III describes SIS in detail. Its menu displays, the screen layout, simulator execution, and program functions are included. The SIS command language and program modules are also covered. The chapter concludes with descriptions of the SIS User's Guide and SIS Module Catalog.

Chapter IV highlights the main features of the SIS system. Future project considerations are also suggested in this chapter.

Objectives

The goals of SIS parallel those of Herot (1982):

1. To aid programmers in the formation of clear and correct mental images of the structure and function of programs.
2. To illustrate the dynamic behavior of programs.
3. To "open the side of the machine" so that the user can form an accurate model of the program.

Herot's emphasis is placed upon the design of a program visualization environment to support builders and maintainers of large, complex software systems, whereas SIS is a visual display package of sorting routines. It allows a student to select a sorting algorithm he or she is interested in viewing. As he interactively investigates the principles and details of the algorithm, his comprehension is enhanced.

CHAPTER II

REVIEW OF LITERATURE

CAI Overview

History

Beginning in the late 1950's, Computer Assisted Instruction (CAI) was developed and applied to many problems in education ranging from elementary to secondary to university level (Suppes, 1978). Its history can be divided into time periods beginning with a "primitive age" of programmed instruction with workbooks, through today's "modern age" of intelligent CAI (Goldstein, 1977). The level of interaction between the student and computer has evolved from the limited, inflexible drill-and-practice systems to highly personalized interactive tutorial presentations (Suppes 1967, Goldstein 1974). The educational promise of CAI is reflected in two trends: (1) the increasing emphasis on individualized instruction, and (2) use of computers to simulate experiences not otherwise readily available (Magidson, 1978). Results indicate that in the future, CAI will continue to be geared toward an instructor supervised system. Supplementary written material and personal support from the

teacher are necessary in order to achieve the maximum benefits of CAI (Chambers, 1980).

Systems and Courses

CAI presentations can appear in many different forms. The most common types of lessons are: drill-and-practice, tutorial, simulation, and games.

In a drill-and-practice setting, the teacher introduces material in class. The student is then directed to work on related exercises offered by the drill-and-practice program. The degree of difficulty of the exercises changes according to the student's performance. These systems are strictly supplementary to the regular curriculum taught by the teacher. One of the first drill-and-practice programs was developed at Stanford for teaching mathematics in elementary schools (Suppes, 1972).

Tutorial systems take over the main responsibility for developing skill in the use of a given concept. They may be complete course sequences or special supplementary units incorporated by the teacher into his course program. Tutorial programs are capable of real time decisions, with branching contingent upon the student's responses. The SOPHIE system developed by Brown (1975), is an example of a CAI tutorial program.

Simulation systems provide the student with the illusion of experiencing a real life occurrence. These systems incorporate graphics routines for portraying a situation.

They are useful in a wide variety of subject areas, including scientific applications. For example, in a chemistry simulation the student can proceed through a potentially dangerous experiment in complete safety and without destruction to laboratory equipment. If he makes an error in judgement he is able to view the results of his error without experiencing physical harm. Examples of laboratory simulation programs are given by Lagowski (1970) and Gelder (n.d.). One of Gelder's simulators demonstrates the interrelationship of pressure, volume, and temperature in a closed system; another illustrates color titration.

The use of computer games for teaching has appeal (Goldstein, 1977). The knowledge gained by the student is learned actively, and with purpose (to improve his score). As in a simulation, the results of a decision are immediate. Computer games inspire motivation because they are enjoyable. However, Goldstein (1977) points out that a game has limitations. A student, when using a game on his own will reach a plateau of learning. At that time he needs a coach in order to further improve his performance. Therefore, Goldstein has combined a tutorial system with the computer game concept to propel the student forward in his learning endeavor.

The use of natural language processing within all forms of CAI has led to the introduction of courseware whereby students are not limited to responding in single words, word phrases, or by making multiple choice selections. Rather,

students may reply in a conversational manner. It is understood that success with natural language is limited (Shapiro, 1975). The user, though not specifically trained in the system's input language, must know the capabilities of the system and must adequately phrase input to the system for satisfactory handling. Courseware using natural language processing has been described by Brown (1975), Wexler (1970), and Carbonell (1970).

CAI has been implemented in a wide variety of subject areas. COALA is a computer-based learning system for an introductory electrical engineering network theory course (Gray, 1977). A case study is given by Peters (1982) on a tutoring system for organic chemistry. Interactive CAI programs have been described for teaching elementary mathematical logic (Suppes 1981, Goldberg 1972). Lantz (1983) has created a system for teaching equation solving. Other related uses of CAI include SIGI, a computer-assisted guidance system (Katz, 1978), and on-line consultation systems (Kehler 1981, Shapiro 1975). These are but a few examples. Thus CAI has progressed from a rough and relatively unexplored theory to a broad field that involves a diversity of systems tailored to the specific need at hand.

Course Design and Development

With any software development activity, risks are to be considered. The user is concerned with the quality of the software and the cost involved in getting the system "up and

running". On the other hand, the developer must have assurance that the hardware for which the software is developed will be widely used. He also needs protection through copyrights and other incentives which increase the probability of a return on his investment (Blaschke, 1979).

Regarding choice of hardware, Matthews (1978) believes that in the education market, the microcomputer is a definite contender. It is simpler and less expensive to operate than a minicomputer. Bork (1979) discusses two ingredients to consider when choosing a microcomputer: screen design capabilities, and programming languages. Although the display screen needs minimal resolution, it should be capable of displaying both alphanumeric information and pictorial or graphic information. Another plus is if the system has the ability to "turn on" a full complex picture rather than drawing it line by line. Bork regards BASIC as a less than adequate language for developing CAI courseware and prefers a PASCAL system. Schuyler (1979) agrees, noting that the best programming language for a given task is one whose structure allows an easy tranformation of the author's ideas into a finished, debugged program, and PASCAL is a good choice.

Once the feasibility of a software development project is established, the specific role of CAI must be clarified. Dimas (1978) suggests that the following questions may aid

in determining the direction that courseware development should take:

1. Should courseware be developed as a complete instructional package or as a supplement and enrichment to the present academic program?

2. Should the development of courseware be devoted to one particular type of lesson, i.e., drill-and-practice, tutorial, simulation, or games, or should different types of lessons be implemented?

When the role of CAI is established, the actual course design may begin. Dean (1983) advises top-down approach, that is, begin with general objectives and work down to detailed objectives. To accomplish this goal he provides the following strategy:

1. Begin with a course description.

2. Divide the course into sub-areas.

3. Divide each sub-area into tasks.

4. Divide each task into steps.

With each step or frame well defined, the job of preparing the computer dialogue is simplified and work can begin.

Many consider a team approach of two to five authors superior to a one man approach in lesson development (Bork 1981, Chambers 1980, Dimas 1978). A negative aspect of the one man approach is that the courseware may reflect the author's idiosyncrasies. Users other than the author may view the lesson as inappropriate for use with their students. Although it is believed that group authorship involves more man hours to create a given amount of material, less revision will be necessary after the material is used with the students. Another important aspect of group

development is that it makes possible the involvement of people with teaching experience, who may lack expertise in those skills necessary for courseware development.

It is speculated that courseware is not likely to improve in the immediate future (Sugarman, 1978). Therefore, the most important explorations should be aimed not at achieving specific learning goals, but at learning about learning through interactive computer use (Skinner 1961, Sugarman 1978). A current CAI issue centers around how to build concepts within the student's mind. Achieving this so-called "modeling" is explored by Suppes (1979). He states that at the present time it is unclear how to model the student and still retain a deep basis for individualization. Nor is it known, from a theoretical standpoint, how to approach the practicalities involved in modeling. However, Bayman (1983) reports progress in teaching concepts of BASIC. He discusses the formation of mental or conceptual models by the beginning programmer in the process of learning the language, also describing the misconceptions the student may acquire.

Critchfield (1979), from another perspective, suggests to proceed one step beyond CAI and provide a system for students to author their own programs. The computer is to be used as an intellectual tool for experimentation and creativity.

CAI in Computer Science

CAI courseware has been created for the specific purpose of teaching computer science principles. Lower level languages such as assembler and machine language are taught through systems devised by Ballaben (1975) and Koffman (1975), respectively. BIP, a BASIC instructional program (Barr 1975a, 1975b, 1976), BATS, a BASIC automatic teaching system (Santos, 1975), and MENO-II, an artificial intelligence based tutor for PASCAL (Soloway, 1983) are examples of CAI courseware for high level languages. Lorton (1981) has designed a system to develop computer literacy by offering tutelage in a wide range of programming languages. A program for teaching the fundamental principles and operations of software systems has been created by Su (1975). Courseware is not limited to any particular area within computer science, rather many aspects on the subject can be learned using CAI.

Dynamic Program Visualization

A major goal in computing is to be able to program (or design and implement algorithms) effectively. It is important to have a clear and in-depth understanding of the dynamic character of programs before attempting algorithm design and program implementation (Dromey, 1982).

The difficulties which arise in presenting an algorithm in the classroom are discussed by Mincy (1983). He expresses concern about how to convey the time factor when

an algorithm modifies the values or states associated with an entity over a period of time. In addition, during a lecture students understand material at different rates. Therefore, the pace of the lecture is often inappropriate for many students. Another problem with the classroom setting is the inability of a student to visualize what is happening.

None of the afore-mentioned programs allows the novice programmer to grasp fully what transpires during program execution. Students need to "see" programs written in a high level language executing. Two approaches have been taken in order to enable students to visualize program execution: animated films, and computer simulation.

Computer systems have been developed which facilitate the production of teaching films containing animated representations of program execution (Baecker 1975, Gross 1975). Sorting Out Sorting is an example of a teaching film produced with the aid of a computer (Baecker, 1981).

Computer simulation is preferred over animated teaching films because the dynamic visual sequence presented by a computer simulation has the following advantages:

1. The display screen is under the complete control of the individual student.
2. The display code can be followed in conjunction with the model animation.
3. The system allows the student to proceed through the algorithm at his own pace.
4. The material can be viewed by the student more than once.

5. There is a greater range of time when the system is accessible (vs. when the film is accessible) for viewing.

6. The simulator can provide a larger assortment of examples.

Examples of computer simulators include systems devised for the displaying of data structures within a program (Myers, 1983), an aid to program visualization (Herot, 1982), and the demonstration of dynamic events using Hypertext (Ward, 1981). Hypertext is a facility for creating, altering, and traversing information in a flexible manner.

CHAPTER III

METHODS AND PROCEDURES

Hardware

SIS is implemented on a GIGI (General Image Generator and Interpretor) terminal manufactured by Digital Equipment Corporation. The terminal is interfaced to a Perkin-Elmer 3230 Computer. Remote Graphics Instruction Set (ReGIS) commands are used to generate the screen displays on a GD 233 video monitor manufactured by BARCO Industries. The program package is written in the language C on the UNIX operating system.

Simulator Design

SIS, when developed, adhered to the guidelines for designing the display as outlined by Kosel (1982). Readability, variety, eye movement, timing, and visual imagery were the most important factors considered.

Menu Display

Three selections must be made by a student before a simulation can begin: choice of algorithm, mode of execution, and speed of execution. With all selections a similiar display format is used (Figure 1). The student is

provided with a blinking pointer to a menu which can be moved around via the up and down arrow keys. The pointer used is actually the graphics cursor which is made visible only during times of a selection process.

Figure 1. An Example of a SIS Menu Display

Originally the pointer was defined as a blinking box. Problems arose, however, in defining the cursor keys. It was difficult to put a bound upon their range due to limitations with ReGIS. Using a box was abandoned, and instead the graphics cursor was implemented as the menu pointer.

The graphics cursor provided all the benefits of the box without any of the disadvantages. With this approach the programming became cleaner. The only modification necessary was to widen the rows of the menu because the graphics cursor was a little taller than the box. This provided a beneficial side effect of improved readability in associating the menu pointer with the correct corresponding option.

The first decision to be made by the student is which sorting algorithm is to be viewed. The selection of sorting algorithms to choose from is:

1. bubblesort
2. insertionsort
3. quicksort
4. selectionsort
5. Shellsort

Selection is made as described above by moving the menu pointer to the desired algorithm and then pressing the return key.

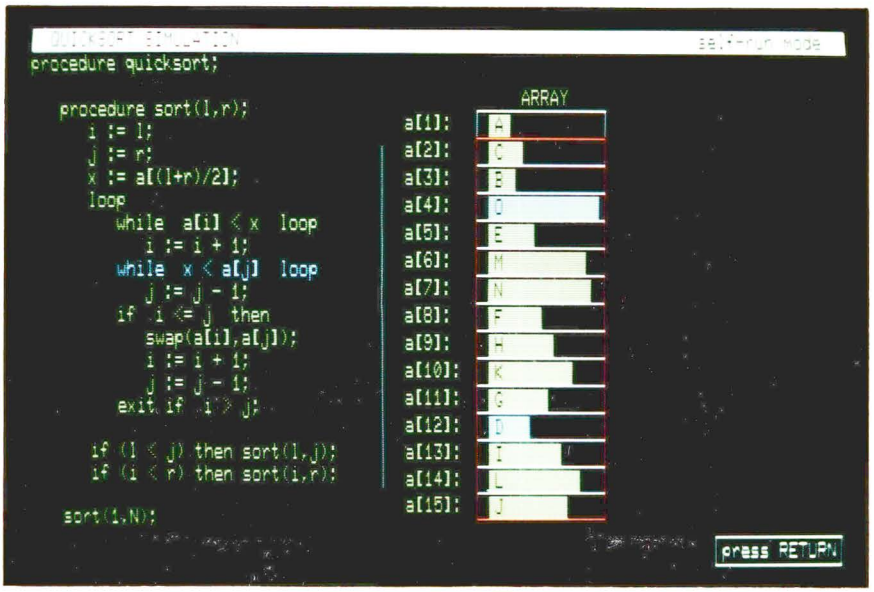
Next, the student must decide the mode of execution in which he would like to see the algorithm presented. There are two modes of execution: self-run mode and single-step mode. Self-run mode portrays the progress of the executing program in dynamic illustrations at a fixed rate of speed. In this mode the student is able to grasp the overall behavior of the particular algorithm in question. Main emphasis is placed on changes occurring within the list of keys to be sorted. He can see visually how the keys are

sorted. Single-step mode steps through the algorithm one instruction at a time. In this mode execution of the algorithm advances to the next instruction only as directed by the student. Such static illustrations portraying the program at some instant of execution time allow the viewer to examine the details and results of each instuction executed. Changes in the flow of control and variable assignments can be monitored along with results of conditional expressions.

If self-run mode is selected, a menu of relative speeds is presented: fast, medium, and slow. The option picked will determine the rate of instruction execution. Specific quantitative rates are not offered as options because the speed of algorithm simulation at a given time is dependent upon the availability of the CPU. In a multi-user environment the speed of simulation will bog down during times of CPU peak workloads.

Screen Layout

Once the student has selected an algorithm, mode of execution, and speed of execution, the simulator display is presented. The screen is divided into two partitions. The right partition holds the array of shuffled keys to be sorted and, if in single-step mode, the program variables. The left partition contains the program design language (PDL) of the algorithm. An example of the contrasting simulator displays of the two execution modes is shown in Figure 2.



(A) Self-Run Mode

(B) Single-Step Mode

Figure 2. Contrasting Simulator Displays of the Two Execution Modes

The layout is similiar to the design described by Dromey (1982). However, instead of illustrating the array of keys as a linear list of integers, a more pictorial representation has been developed (Figure 3). Letters are used as the keys rather than integers. The use of letters is an improvement over the use of integers because misunderstandings can arise when using integer keys in the array. Students often will confuse integers with the indices of the array. In addition, the letter keys within the array are represented by rectangles of proportional sizes as illustrated. The rectangles aid in eye recognition, as it is easy to check whether a[i] is greater than a[j] by comparing the sizes of the rectangles in both locations.

A PDL rather than a flowchart is placed in the left partition to follow the flow of the executing program. Results appear to provide a strong case for the use of a program design language in preference to flowcharts (Ramsey, 1983).

An ADA-like pseudocode is used as the PDL (Young, 1982). In order to fit the entire algorithm PDL on the display screen at one time, its length must be less than twenty-three lines. This limitation does not pose a problem except in the case of quicksort. To circumvent this problem a recursive version of quicksort is used.

Figure 3. Array of Keys Representation

Two other functional areas of the display are the return prompt and the header. The header rests at the top of the screen and provides orientation information. It informs the student of the algorithm currently being shown and its mode of execution. The lower right hand corner of the simulator display is reserved for a return symbol which prompts the student for a response. In the single-step mode, pressing the return key steps the simulation to the next instruction. Pressing the return key at the conclusion of a

simulation prompts SIS to ask, "Do you wish to make another selection?". Depending upon his response, the student is either redirected to the algorithm menu or logged out of SIS.

The Use of Color in Algorithm Simulation

Color is effective for highlighting any graphics entity. The color selection of the principle components in the running simulation is critical. Care must be taken in order to ensure maximum clarity within the ongoing presentation. The guidelines recommended by Heines (1984) were applied:

1. Make sure the adjacent colors do not clash.
2. Do not use colors that are too "hot".

In addition, the colors selected need to provide contrast. Contrasting colors are important to guide the reader's eye to important changes within the operating simulation.

Green was chosen as the primary display color. It is a soft color and easy to read. Unlike white, it is monochromatic and provides a sharper picture while not contributing to eye fatigue. The other colors picked to highlight simulation features are listed in Table I. Black was reserved for the background color because Foley (1982) advises that a neutral background be used with a display containing several colors.

By using a pre-defined color coding scheme, the PDL, array of keys, and program variables can be integrated to present the student with a detailed conceptual model. In

either execution mode, the student can associate what array key(s) are affected by the current executing instruction. Both the current PDL instruction and relevant array key(s) are highlighted in cyan.

TABLE I

SIS DISPLAY COLOR CODES

Color	Purpose
Green	General Display Color
Yellow	Highlights Previous Instruction
Blue (Cyan)	Highlights Current Instruction, Keys, and Variables
Red	Highlights Recursion
White	Highlights Display Header and Return Symbol
Dark	Background Color

Because single-step mode was designed to focus on the details of a single executing instruction, it provides additional information. Not only are the current PDL instruction and relevant array key(s) displayed in cyan, but so are the affected variables found within that instruction. For instance, if the current instruction is a conditional ex-

pression, the result of that condition is posted (either true or false). In addition, when in single-step mode the previous PDL instruction is color coded in yellow whichs aids in following the flow of control of the program.

The recursion imposed on the quicksort simulation introduces modifications in the color coding scheme. These modifications are necessary in order to make clear to the student when a recursive subroutine call is being made. On such an occasion the current PDL instruction and affected variables are highlighted in red. This deviates from the normal cyan highlighting. Within the array, a red box is drawn around the range of cells affected by the procedural call. This box is removed when the subroutine is exited. While inside the subroutine, all current components revert to cyan highlighting.

It should be pointed out that SIS does not teach recursion. Rather, it illustrates the performance of a recursive algorithm. It is assumed that the student has some notion of the concept of recursion before using SIS.

Program Functions

SIS offers a variety of program functions which may be called while the user is viewing a simulation. The break key may be pressed if the student elects to leave the current simulation prematurely. Such action prompts SIS to erase the screen and ask whether the student wants to view another sorting routine or prefers to exit the program.

SIS offers additional program functions when in self-run mode. For example, the viewer has the prerogative of changing the execution speed of the simulation by pressing the corresponding function keys. A pause function is also provided which can be used to halt program execution temporarily. A summary of the program functions available is given in Table II.

TABLE II

SUMMARY OF THE PROGRAM FUNCTIONS

Key	Function
'F' or 'f'	Change Execution Speed To 'Fast'
'M' or 'm'	Change Execution Speed To 'Medium'
'S' or 's'	Change Execution Speed To 'Slow'
'P' or 'p'	Pause Or Temporarily Halt Simulation, Press 'RETURN' Key To Resume Execution
'BREAK'	Abort Current Simulation

The terminal response time to depressed program function keys may not be immediate. The alarm macro provided by the UNIX operating system is methodically set to test for depressed function keys, but the time resolution of the

alarm is only one second. Therefore a small delay can occur between the time the function key is pressed and when the terminal responds.

In order to maintain clarity, the viewer is not allowed to change the mode of execution while a simulation is in progress. Because the objectives of the self-run mode and single-step mode are different, their respective PDLs vary slightly. Each PDL is tailor-made for a specific purpose. If the viewer were allowed to change the mode of execution, such an abrupt change within the currently displayed PDL could cause confusion. As an alternative, the viewer can abort the running simulation and restart it in the other execution mode.

Progam Description

The Program Modules

Due to the nature of the C programming language, SIS is constructed as a collection of interfacing procedures or modules. The principles set forth by Maynard (1972) are followed in modularizing the system. Each module is designed to fulfill a specific need within the program package.

While designing a system, care must be given when breaking the program code into modules. Within SIS there are three basic types of modules: control modules, process modules, and graphics modules. The function of a control module is to control the calling sequence of the modules

under its jurisdiction. A process module performs a single logical function or a series of small related logical functions. A graphics module carries out a specific graphics function whose results can be observed upon the screen display. Appendix A gives the schematics of the SIS module interfacing.

Modules of common purpose are grouped together into one file. The files constituting SIS can be divided into two categories: primary files containing primary modules, and secondary files containing secondary modules. All primary modules of a particular primary file serve together in a specific stage of the SIS program. They are not used at any other time. The following is a list of the primary files implemented by SIS and the functions they perform:

sis.c - The main driver of the system.

setup.c - Initializes both the screen and graphics parameters and produces the title of the package.

options.c - Displays all menu frames and processes all input in response to the menu frames.

screen.c - Displays the opening frame of the selected algorithm simulation.

run_sort.c - The driver of the algorithm simulation.

Secondary modules play a supplementary role to the primary modules in a SIS program. They exist as common functions which may be utilized many times throughout a simulation by primary modules of different primary files. Secondary modules are not limited by the number of times and places they are called. They are grouped into files accord-

ing to the functions they perform. The names of the secondary files used by SIS are:

globals.c - The routines which generate and remove the return symbol.

dspfns.c - The graphics modules which supplement the primary modules found in run_sort.c and screen.c.

code.c - The graphics display modules which supplement screen.c with the sorting algorithm PDLs.

sorts.c - The source code modules of the sorting algorithms supplementing run_sort.c.

kybdfns.c - The modules used for handling the simulation functions generated from the keyboard.

The interrelationship between the SIS files is diagramed in Figure 4.

The SIS Command Language

The SIS Command Language has been developed to simplify the programming of the algorithm simulation. Each command in the language can be conceived as an encapsulation of several ReGIS commands. The language acts as an interface between the sorting algorithm source code and the screen display. To the programmer, the SIS commands operate as visual tools and are implemented by being inserted into the existing source code of the sorting algorithms. The basic core of the source code is left unaltered. A complete summary of all available SIS commands is located in the SIS Module Catalog, Appendix B.

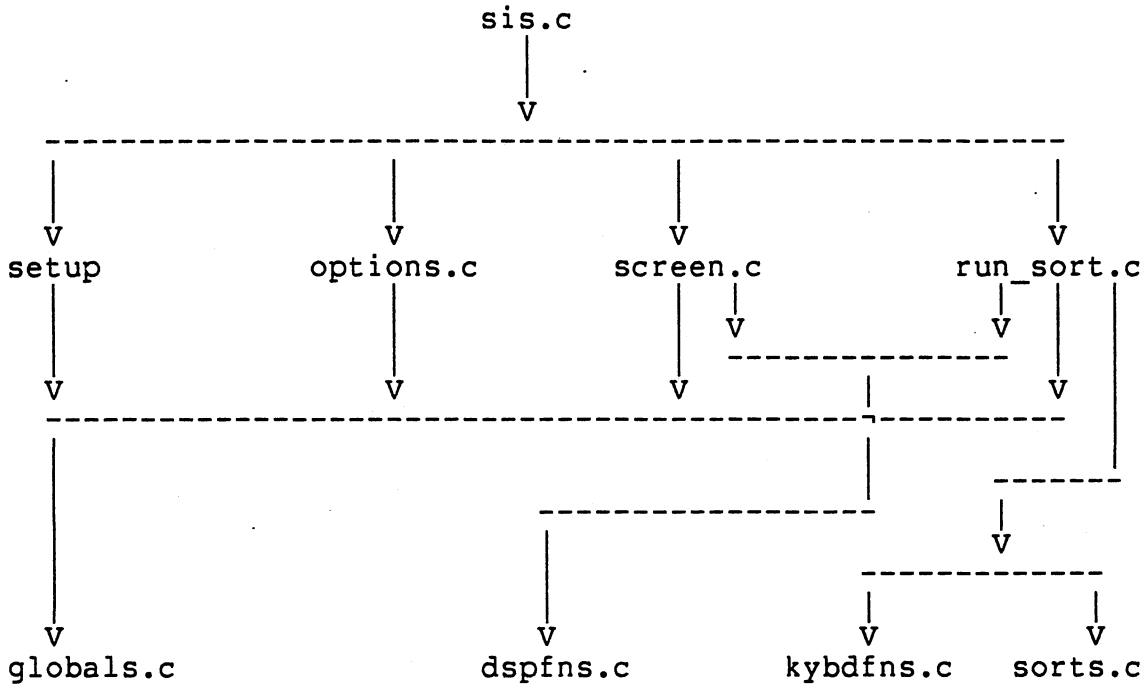


Figure 4. SIS File Interrelationship

The language can be broken down categorically into five types of commands. Each type of command controls one or more of the components to the simulation display: the PDL, the array of keys, or the variables. SIS commands are categorized as follows:

1. Line commands.
2. Variable commands.
3. Stack commands.
4. Condition commands.
5. Box commands.

1. Line commands display PDL instructions in the color indicated. The parameter passed is a PDL line number. For

example, a blue_line(2) command specifies that PDL line number two is to be displayed in blue (cyan). Similar results are obtained for red_line commands. Because green is the primary display color, a green_line command has an additional feature associated with it: if single-step mode is employed, the previously executed PDL instruction is displayed in yellow.

2. Variable commands function to highlight denoted variable displays and array display keys. A green_var('i') command dictates that the variable 'i' be displayed in green. A red_var('k') command directs the variable 'k' be highlighted in red. The format of variable commands only allows single character parameters to be passed. Variable names occurring in an algorithm that exceed one character are substituted with unique mnemonic symbols. Such substitutions are required for array variables and temporary variables. The symbols '-', 'J', and 'T' representing the variable names a[j-1], a[j], and temp respectively, are samples of mnemonic replacements. A blue_var('-') command orders the variable 'a[j-1]' and its associated key to be highlighted in blue (cyan).

3. Stack commands and variable commands function similarly. They both highlight variables and array keys, and pass the same set of parameters. However, unlike variable commands, stack commands use a display stack in their implementation. They are comprised of (1) push_dsp commands and (2) pop_dsp commands. A push_dsp command acts like a

blue_var command by highlighting in blue. In addition, it pushes its parameter onto the display stack. When a pop_dsp command is invoked, all stack elements are systematically popped off the display stack, and each becomes the object of a green_var command. Use of stack commands has advantages over variable commands in situations that merit many variable displays and key displays to be reset to green. Stack commands condense the SIS command encoding by replacing a list of green_var instructions with a single pop_dsp instruction.

4. Condition commands are issued only in single-step mode. Their function is to display the result of a conditional expression. 'TRUE' and 'FALSE' are posted in blue (cyan) for the condition commands cond(1) and cond(0) respectively. When a subsequent instruction is executed, the result of a condition command is automatically erased from the screen display.

5. Box commands are restricted to SIS presentations utilizing recursion. They appear in two varieties: as red_box commands or as white_box commands. The function of a red_box command is to encompass the range of cells in the array display affected by the recursive call. When returning from a recursive call, a white_box command is invoked to remove the red enclosure. Two integers are passed as parameters within box commands. These parameters designate the lower and upper bounds of the box to be superimposed upon a portion of the array display.

Two Examples of Source Code with SIS Commands

This section discusses the steps to follow in achieving the transformation of algorithm code into a SIS display which can be dynamically visualized. Two examples are furnished on how to transform bubblesort into (1) a self-run mode presentation and (2) a single-step mode presentation.

The six basic steps of a SIS algorithm creation are:

1. Establish the algorithm PDL.
2. Insert the line commands.
3. Insert the variable commands and stack commands.
4. Insert the condition commands.
5. Insert the box commands.
6. Replace the PDL instructions with program code.

These steps are followed regardless of the execution mode desired.

Step 1: Establish the algorithm PDL. The PDL serves as the framework for the SIS command insertions. It is identical to the one which will appear on the display screen during the simulation. Correctness of the PDL instructions is paramount because inaccuracies or mistakes will eventually lead to the conveying of erroneous information to the viewer. In this example, it was decided that the algorithm to be presented in self-run mode should be the one illustrated in Figure 5.A. This is contrasted with the single-step mode algorithm shown in Figure 5.B, in which it was

decided that the call to 'swap' should, for pedagogical reasons, be replaced by the three lines shown.

```
for i in 2..N
  for j in reverse N..i
    if a[j-1] > a[j] then
      swap(a[j],a[j-1])
```

(A) Self-Run Mode

```
for i in 2..N
  for j in reverse N..i
    if a[j-1] > a[j] then
      temp := a[j-1]
      a[j-1] := a[j]
      a[j] := temp
```

(B) Single-Step Mode

Figure 5. Bubblesort PDLs

Step 2: Insert the line commands. There are three different line commands used for displaying PDL instructions:

1. A blue_line command directed to highlight the particular PDL instruction.
2. A green_line command which resets the instruction (blue_line commands must always be followed by green_line commands).
3. A red_line command, which is substituted for the blue_line command whenever the PDL instruction is a recursive function call.

Figure 6 presents the bubblesort algorithms after the insertion of the line commands.

```
  for i in 2..N
*   blue_line(1);
*   green_line(1);

    for j in reverse N..i
*     blue_line(2);
*     green_line(2);

*     blue_line(3);
*     green_line(3);
      if a[j-1] > a[j] then

        swap(a[j],a[j-1])
*       blue_line(4);
*       green_line(4);
```

(A) Self-Run Mode

```
  for i in 2..N
*   blue_line(1);
*   green_line(1);

    for j in reverse N..i
*     blue_line(2);
*     green_line(2);

*     blue_line(3);
*     green_line(3);
      if a[j-1] > a[j] then

        temp := a[j-1]
*       blue_line(4);
*       green_line(4);

        a[j-1] := a[j]
*       blue_line(5);
*       green_line(5);

        a[j] := temp
*       blue_line(6);
*       green_line(6);
```

(B) Single-Step Mode

Figure 6. Insertion of Line Commands into Bubblesort

The ordering of the PDL instruction with its associated line commands is most important. Consider the first PDL instruction of Figure 6.A, which is a 'for loop' statement. If its related line commands were listed ahead of the 'for loop' statement instead, the line commands would execute only once while the first PDL instruction would execute at each iteration. Unlike the first PDL instruction of Figure 6.A, the third original PDL instruction (which is an 'if' statement) needs preceeding line commands. If the related line commands were placed beneath the 'if' condition in-

stead, they would not be executed when the result of the condition was false. It is urged strongly that the designer walk through his algorithm at the conclusion of Step 2 to assure himself that the ordering is correct.

Step 3: Insert the variable commands and stack commands. Variable commands and stack commands pertaining to a PDL instruction are inserted following the instruction's blue_line command. Stack commands are generally favored over variable commands because variable displays are easier to reset when using stack commands. One exception to stack command preference regards a variable appearing as an operand in consecutive PDL instructions. In such a situation the variable can be represented with one blue_var command in the beginning of the succession and one green_var command at the conclusion of the succession. Self-run mode algorithms have fewer variable command and stack command insertions than single-step mode algorithms. Unlike single-step mode algorithms, self-run mode algorithms include only those variable commands and stack commands which affect the array display keys. The bubblesort algorithms after the insertion of stack commands and variable commands are shown in Figure 7.

Step 4: Insert the condition commands. For each 'if statement' appearing as a PDL instruction in a single-step mode algorithm, a duplicate is made and placed ahead of the original. Condition commands are then inserted into the duplicate. Figure 8 gives the bubblesort algorithm in

```
  for i in 2..N
    blue_line(1);
    green_line(1);

    for j in reverse N..i
      blue_line(2);
      green_line(2);

      blue_line(3);
*     blue_var('J');
*     blue_var('-');
      green_line(3);
      if a[j-1] > a[j] then

        swap(a[j],a[j-1])
        blue_line(4);
*       blue_var('J');
*       blue_var('-');
        green_line(4);

*     green_var('J');

*   green_var('J');
```

(A) Self-Run Mode

```
  for i in 2..N
    blue_line(1);
*   push_dsp('i');
*   push_dsp('N');
    green_line(1);
*   pop_dsp();

    for j in reverse N..i
      blue_line(2);
*     push_dsp('j');
*     push_dsp('N');
*     push_dsp('i');
      green_line(2);
*     pop_dsp();

      blue_line(3);
*     blue_var('-');
*     push_dsp('J');
      green_line(3);
*     pop_dsp();
      if a[j-1] > a[j] then

        temp := a[j-1]
        blue_line(4);
*       push_dsp('T');
        green_line(4);
*       pop_dsp();

        a[j-1] := a[j]
        blue_line(5);
*       push_dsp('-');
*       blue_var('J');
        green_line(5);
*       pop_dsp();

        a[j] := temp
        blue_line(6);
*       push_dsp('J');
*       push_dsp('T');
        green_line(6);
*       pop_dsp();

*     else
*       green_var('-');
```

(B) Single-Step Mode

Figure 7. Insertion of Variable Commands and Stack Commands into Bubblesort

```
   for i in 2..N
     blue_line(1);
     push_dsp('i');
     push_dsp('N');
     green_line(1);
     pop_dsp();

   for j in reverse N..i
     blue_line(2);
     push_dsp('j');
     push_dsp('N');
     push_dsp('i');
     green_line(2);
     pop_dsp();

     blue_line(3);
     blue_var('-');
     push_dsp('J');
*    if a[j-1] > a[j] then
*      cond(1);              /* displays 'TRUE'  */
*    else
*      cond(0);              /* displays 'FALSE' */
     green_line(3);
     pop_dsp();
     if a[j-1] > a[j] then

       temp := a[j-1]
       blue_line(4);
       push_dsp('T');
       green_line(4);
       pop_dsp();

       a[j-1] := a[j]
       blue_line(5);
       push_dsp('-');
       blue_var('J');
       green_line(5);
       pop_dsp();

       a[j] := temp
       blue_line(6);
       push_dsp('J');
       push_dsp('T');
       green_line(6);
       pop_dsp();

     else
       green_var('-');
```

Figure 8. Bubblesort Algorithm in Single-Step Mode with the Condition Commands Inserted

single-step mode with the condition commands added. Since self-run mode algorithms do not display results of conditional expressions, they do not require condition commands.

Step 5: Insert the box commands. Box commands are used exclusively in recursive algorithms. If recursion does not exist within the algorithm (as in the case of the bubblesort examples), then this step is omitted. Red_box commands are inserted immediately before red_line commands. A white_box command is placed above each recursive function call and as the last executable statement within the called function.

Step 6: Replace the PDL instructions with program code. The final step entails the translation of all PDL instructions into actual program code. In addition to the original PDL instructions, all the duplicated 'if statements' from Step 4 must be changed as well. The completed bubblesort algorithms are listed in Figure 9.

An Example of Recursion and SIS Commands

All variables appearing in a SIS presentation are defined within the program as external variables. This poses a problem when simulating a recursive algorithm. Unavoidable errors result when SIS commands are inserted into an unaltered recursive function. The errors which occur pertain to the scope of recursively passed variables. Local copies of external variables are passed as parameters when the external variables are used later in SIS Commands. This

```
* for (i=2; i<=N; i++) {
    blue_line(1);
    green_line(1);

*   for (j=N; j>=i; j--) {
      blue_line(2);
      green_line(2);

      blue_line(3);
      blue_var('J');
      blue_var('-');
      green_line(3);
*     if (a[j-1] > a[j]) {

*       temp = a[j-1];
*       a[j-1] = a[j];
*       a[j] = temp;
        blue_line(4);
        blue_var('J');
        blue_var('-');
        green_line(4);
*     }
      green_var('J');

*   }
    green_var('J');
* }
```

(A) Self-Run Mode

```
* for (i=2; i<=N; i++) {
    blue_line(1);
    push_dsp('i');
    push_dsp('N');
    green_line(1);
    pop_dsp();

*   for (j=N; j>=i; j--) {
      blue_line(2);
      push_dsp('j');
      push_dsp('N');
      push_dsp('i');
      green_line(2);
      pop_dsp();

      blue_line(3);
      blue_var('-');
      push_dsp('J');
      if (a[j-1] > a[j])
        cond(1);
      else
        cond(0);
      green_line(3);
      pop_dsp();
*     if (a[j-1] > a[j]) {

*       temp = a[j-1];
        blue_line(4);
        push_dsp('T');
        green_line(4);
        pop_dsp();
*       a[j-1] = a[j];
        blue_line(5);
        push_dsp('-');
        blue_var('J');
        green_line(5);
        pop_dsp();
*       a[j] = temp;
        blue_line(6);
        push_dsp('J');
        push_dsp('T');
        green_line(6);
        pop_dsp();
*     } else
        green_var('-');
*   }
* }
```

(B) Single-Step Mode

Figure 9. The Completed Bubblesort Algorithms

harmonizes recursive parameter passing with external variable declarations by preventing external variables from being altered by function calls. Figure 10 provides an example of how the PDL of a recursive function can be altered to accommodate SIS commands.

```
procedure sort(l,r);

   i = l;

   j = r;

      .
      .
      .
      .


   if l < j then
      sort(l,j);

   if i < r then
      sort(i,r);
```

(A) Original PDL

```
*   procedure sort(left,right);

*       l = left;
*       r = right;

        i = l;

        j = r;

            .
            .
            .
            .

*       if left < j then
*          sort(left,j);

*       if i < right then
*          sort(i,right);
```

(B) Modified PDL

Figure 10. An Alteration in the PDL of a Recursive Function to Accommodate SIS Commands

After the original PDL has been altered, the insertion of SIS commands into the algorithm follows the procedure as outlined in the previous section. Figure 11 further illus-

trates the recursive function example with the insertion of SIS commands associated with recursive algorithms.

```
  procedure sort(left,right);

      l = left;
      r = right;

*     red_line(l);
*     green_line(l);

      i = l;
*     blue_line(2);
*     green_line(2);

      j = r;
*     blue_line(3);
*     green_line(3);

          .
          .
          .
          .


*     blue_line(15);
*     green_line(15);
      if left < j then
         sort(left,j);

*     blue_line(16);
*     green_line(16);
      if i < right then
         sort(i,right);
```

(A) Line Command Insertion

```
  procedure sort(left,right);

      l = left;
      r = right;
*     red_box(l,r);

      red_line(l);
      green_line(l);

      i = l;
      blue_line(2);
      green_line(2);

      j = r;
      blue_line(3);
      green_line(3);

          .
          .
          .
          .


      blue_line(15);
      green_line(15);
      if left < j then
*        white_box(l,r);
         sort(left,j);

      if i < right then
*        white_box(l,r);
         sort(i,right);

*     white_box(l,r);
```

(B) Box Command Insertion

Figure 11. SIS Command Insertion into a Recursive Function

User's Guide Description

The SIS User's Guide is a student's roadmap through a SIS presentation. The viewer will find in it the detailed information required to operate SIS. Illustrations of the major displays supplement the step-by-step instructions. The instructions are written at a low level so that they may be understood by persons with no computer experience. A copy of the SIS User's Guide is found in Appendix C.

Module Catalog Description

Specifications of all modules implemented in the simulator embody the SIS Module Catalog. Each module description is comprised of the following characteristics: module type, usage, input parameter(s), and returned value. The primary purpose of this catalog is assisting programmers who want to incorporate SIS modules into future systems. Appendix B contains a listing of the SIS Module Catalog.

CHAPTER IV

SUMMARY AND FUTURE WORK

SIS Synopsis

Many problems for beginning programming students relate to program comprehension, that is, the ability to conceptualize the abstraction of a given algorithm with the flow of the program code. SIS was developed to aid in this conceptualization process by providing the student a means to observe program execution via simulation. It offers the student a choice of five sorting algorithms to view in two modes of execution and three execution speeds. Program functions built into SIS allow the user to halt or adjust the execution speed, switch to another algorithm, or prematurely exit the simulator. The package was designed primarily as a demonstration tool for supplementing material presented in an algorithm course.

To assess the educational efficacy of SIS necessitates its evaluation in a classroom setting. Since the scope of this project was to design and implement SIS, no conclusions can be drawn regarding its performance or effectiveness. Such a study is left for future work.

Future Project Considerations

Natural extensions of SIS can be developed without requiring any changes in the screen format. If an array implementation is maintained, two such extensions would be a simulator for search algorithms, and a system to illustrate list structure operations. The most important modifications needed in SIS would be:

1. Altering the menu display for choosing the algorithm.
2. Displaying the relevant PDL and variables.
3. Inserting the necessary SIS commands into the source code.

Sequential search and binary search are examples of searching algorithms which could be presented. List structure operations which could be demonstrated include: the pop and push operations defined on a stack, and the remove and insert operations defined on either a deque or queue.

If the array display in SIS is replaced with a binary tree display, additional future projects may be considered. This extension would require the screen layout to be reformatted because a tree structure assumes a different shape than an array. In lieu of the tree display, the variable display fields would have to be repositioned. The major modification centers around the development of additional SIS commands to represent the tree display. Display modules for inserting and deleting the tree nodes as well as for labeling the tree nodes are needed. Heapsort, tree

traversals, and tree implemented sorting and searching algorithms are examples which could be simulated with a tree display.

SIS is designed to be implemented on GIGI. At institutions where GIGI is unavailable, some modifications can be performed to make the simulator compatible with a VT125 (VT125 User Guide , 1981). The changes which must be made regard the choice of colors or color commands used and the method of option selection. The changes are outlined in Appendix D.

The development of SIS points to several future projects. The SIS Command Language lends itself to adaptation regarding educational content and hardware compatibility. With minor modifications, SIS should be of value in teaching students additional computer science concepts.

SELECTED BIBLIOGRAPHY

Baecker, R. M. "Two Systems Which Produce Animated Representations of the Execution of Computer Programs." SIGCSE Bulletin 7, 1 (1975), 158-167.

Baecker, R. M. Sorting Out Sorting , 16 mm color, sound, 25 minutes (Dynamic Graphics Project, Computer Systems Research Group, University of Toronto, 1981).

Ballaben, G. and Ercoli, P. "Computer-Aided Teaching of Assembler Programming." In O. Lecarme and R. Lewis (eds.), Computers in Education , IFIP (Part 1). Amsterdam: North-Holland, 1975, 217-227.

Barr, A., Beard, M., and Atkinson, R. C. "A Rationale and Description of a CAI Program to Teach the BASIC Programming Language." Instructional Science 4 (1975), 1-31.

Barr, A., Beard, M., and Atkinson, R. C. "Information Networks for CAI Curriculums." In O. Lecarmi and R. Lewis (eds.), Computers in Education , IFIP (Part 1). Amsterdam: North-Holland, 1975, 477-582.

Barr, A., Beard, M., and Atkinson, R. C. "The Computer as a Tutorial Laboratory: The Stanford BIP Project." International Journal of Man-Machine Studies 8 (1976), 567-596.

Bayman, P. and Mayer, R. E. "A Diagnosis of Beginning Programmers' Misconceptions of BASIC Programming Statements." Communications of the ACM 26, 9 (1983), 677-679.

Blaschke, C. L. "Microcomputer Software Development for Schools: What, Who, How?" Educational Technology 19, 10 (1979), 26-28.

Bork, A. and Franklin, S. "Personal Computers in Learning." Educational Technology 19, 10 (1979), 7-12.

Bork, A. Learning With Computers. Bedford, MA: Digital Press, 1981.

Brown, J. S., Burton, R. R., and Bell, A. G. "SOPHIE: A Step Toward Creating a Reactive Learning Environment." International Journal of Man-Machine Studies 7 (1975), 675-696.

Brown, M. H., and Sedgewick, R. "A System for Algorithm Animation." Computer Graphics 18, 3 (1984), 177-186.

Carbonell, J. "AI in CAI: An Artificial-Intelligence Approach to Computer-Assisted Instruction." IEEE Transactions on Man-Machine Systems MMS-11, 4 (1970), 190-202.

Chambers, J. A. and Sprecher, J. W. "Computer Assisted Instruction: Current Trends and Critical Issues." Communications of the ACM 23, 6 (1980), 332-342.

Critchfield, M. "Beyond CAI: Computers as Personal Intellectual Tools." Educational Technology 19, 10 (1979), 18-25.

Dean, C. and Whitlock, Q. A Handbook of Computer Based Training. New York: Nichols Publishing Co., 1983.

Dimas, C. "A Strategy for Developing CAI." Educational Technology 18, 4 (1978), 26-29.

Dromey, R. G. How to Solve It by Computer. London: Prentice/Hall International, Inc., 1982.

Foley, J. D. and Van Dam, A. Fundamentals of Interactive Computer Graphics. Reading, MA: Addison-Wesley, 1982.

Gelder, J. I. Unpublished chemistry simulation programs. Oklahoma State University, Department of Chemistry, (n.d.).

Goldberg, A. and Suppes, P. "A Computer-Assisted Instruction Program for Exercises on Finding Axioms." Educational Studies in Mathematics 4 (1972), 429-549.

Goldstein, I. I. Training: Program Development and Evaluation. Monterey, CA: Brooks/Cole, 1974.

Goldstein, I. P. and Carr, B. "The Computer as Coach: An Athletic Paradigm for Intellectual Education." Proc. 1977 ACM Annual Conf., 1977, 227-233.

Gray, D. C., Hulskamp, J. P., Kumm, J. H., Lichtenstein, S., and Nimmervoll, N. E. "COAL-A - A Minicomputer CAI System." IEEE Transactions on Education E-20, 1 (1977), 73-77.

Gross, J. F. "Video Augmented Computer Science (VACS)." SIGCSE Bulletin 7, 4 (1975), 47-59.

Heines, J. M. Screen Design Strategies for Computer-Assisted Instruction. Bedford, MA: Digital Press, 1984.

Herot, C. P., Brown, G. P., Carling, R. T., Friedall, M., Kramlich, D., and Baecker, R. M. "An Integrated Environment for Program Visualization." In H. J. Schneider and A. I. Wasserman (eds.), Automated Tools for Information Systems Design. Amsterdam: North-Holland, 1982, 237-259.

Katz, M. R. and Chapman, W. "SIGI: An Example of Computer-Assisted Guidance." Educational Technology 18, 4 (1978), 57-59.

Kehler, T. P. and Barnes, M. "Design for an On-Line Consultation System." AEDS Journal 14, 3 (1981), 113-127.

Koffman, E. B. and Blount, S. E. "Artificial Intelligence and Automatic Programming in CAI." Artificial Intelligence 6 (1975), 215-234.

Kosel, M. and Jostad, K. "Designing the Display." PIPELINE 7, 1 (1982), 8-10,57.

Lagowski, J. J. "Computer-Assisted Instruction in Chemistry." In W. H. Holtzman (ed.), Computer-Assisted Instruction, Testing, and Guidance. New York: Harper & Row, 1970, 283-298.

Lantz, B. S., Bregar, W. S., and Farley, A. M. "An Intelligent CAI System for Teaching Equation Solving." Journal of Computer-Based Instruction 10, 1 & 1 (1983), 35-52.

Lorton, P. Jr. and Cole, P. "Computer-Assisted Instruction in Computer Programming: SIMPLER, LOGO, and BASIC, 1968-1970." In P. Suppes (ed.), University-Level Computer-Assisted Instruction at Stanford: 1968-1980. Stanford, CA: Stanford University, Institute for Mathematical Studies in the Social Sciences, 1981, 841-876.

Magidson, E. M. "Issue Overview: Trends in Computer-Assisted Instruction." Educational Technology 18, 4 (1978), 5-8.

Matthews, J. I. "Microcomputer vs. Minicomputer for Educational Computing." Educational Technology 18, 11 (1978), 19-22.

Maynard, J. Modular Programming Princeton: Auerbach Publishers, 1972.

Mincy, J. W., Tharp, A. L., and Kuo-Chung, T. "Visualizing Algorithms and Processes with the Aid of a Computer." SIGCSE Bulletin 15, 1 (1983), 106-111.

Myers, B. A. "INCENSE: A System for Displaying Data Structures." Computer Graphics 17, 3 (1983), 115-125.

Peters, H. J. and Daiker, K. C. "Graphics and Animation as Instructional Tools: A Case Study." PIPELINE 7, 1 (1982), 11-13,57.

Ramsey, H. R., Atwood, M. E., and Van Doren, J. R. "Flowchart versus Program Design Languages: An Experimental Comparison." Communications of the ACM 26, 6 (1983), 445-549.

Santos, S. M. dos and Millan, M. R. "A System for Teaching Programming by Means of a Brazilian Minicomputer." In O. Lecarmi and R. Lewis (eds.), Computers in Education, IFIP (Part 1). Amsterdam: North-Holland, 1975, 211-216.

Schuyler, J. A. "Programming Languages for Microprocessor Courseware." Educational Technology 19, 10 (1979), 29-35.

Shapiro, S. C. and Kwasny, S. C. "Interactive Consulting via Natural Language." Communications of the ACM 18, 8 (1975), 459-562.

Skinner, B. F. "Why We Need Teaching Machines." Harvard Education Review 31 (1961), 377-398. Reprinted in J. P. De Cecco (ed.), Educational Technology. New York: Holt, Rinehart, and Winston, 1964, 92-112.

Soloway, E., Rubin, E., Woolf, B., Bonar, J., and Johnson, W. L. "MENO-II: An AI-Based Programming Tutor." Journal of Computer-Based Instruction 10, 1 & 2 (1983), 20-34.

Su, S. Y. W. and Eman, A. E. "Teaching Software Systems on a Minicomputer: A CAI Approach." In O. Lecarmi and R. Lewis (eds.), Computers in Education, IFIP (Part 1). Amsterdam: North-Holland, 1975, 223-229.

Sugarman, R. "A Second Chance for Computer-Aided Instruction." IEEE Spectrum 15, 8 (1978), 29-37.

Suppes, P. "On Using Computers to Individualize Instruction." In D. D. Bushnell and D. W. Allen (eds.), The Computer in American Education. New York: John Wiley, 1967, 11-24.

Suppes, P., and Morningstar, M. Computer-Assisted Instruction at Stanford, 1966-68: Data, Models, and Evaluation of the Arithmetic Programs. New York: Academic Press, 1972.

Suppes, P. "Current Trends in Computer-Assisted Instruction." In M. C. Yovits (ed.), Advances in Computers (Vol. 18). New York: Academic Press, 1979, 173-229.

Suppes, P. and Macken, E. " The Historical Path from Research and Development to Operational Use of CAI." Educational Technology 18, 4 (1978), 9-12.

Suppes, P. and Sheehan, J. "CAI Course in Logic." In P. Suppes (ed.), University-Level Computer-Assisted Instruction at Stanford: 1968-1980. Stanford, CA: Institute for Mathematical Studies in the Social Sciences, 1981, 193-225.

VT125 User Guide. Maynard, MA: Digital Equipment Co., 1981.

Ward, D. L. and Irby, T. C. "Classroom Presentation of Dynamic Events Using Hypertext." SIGCSE Bulletin 13, 1 (1981), 126-131.

Wexler, J. D. "Information Networks in Generative Computer-Assisted Instruction." IEEE Transactions on Man-Machine Systems MMS-11, 4 (1970), 181-190.

Young, S. J. Real Time Languages: Design and Development. Chichester, England: Ellis Horwood Limited, 1982.

APPENDIX A

SIS MODULE SCHEMATICS

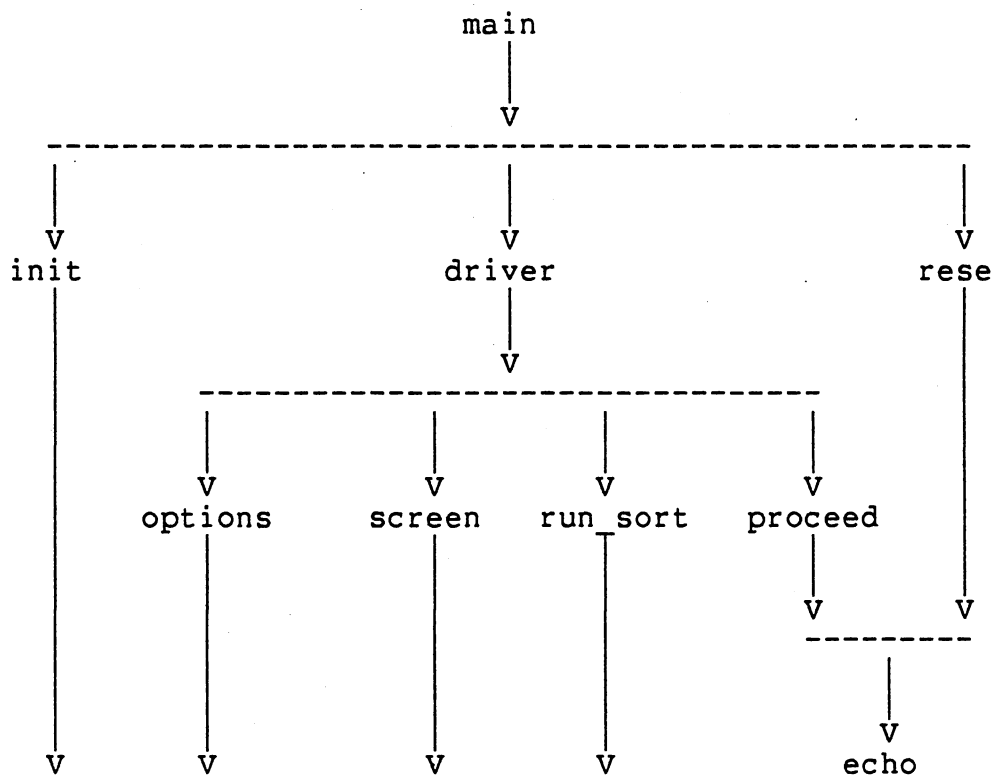


Figure 12. Main Module Schematics

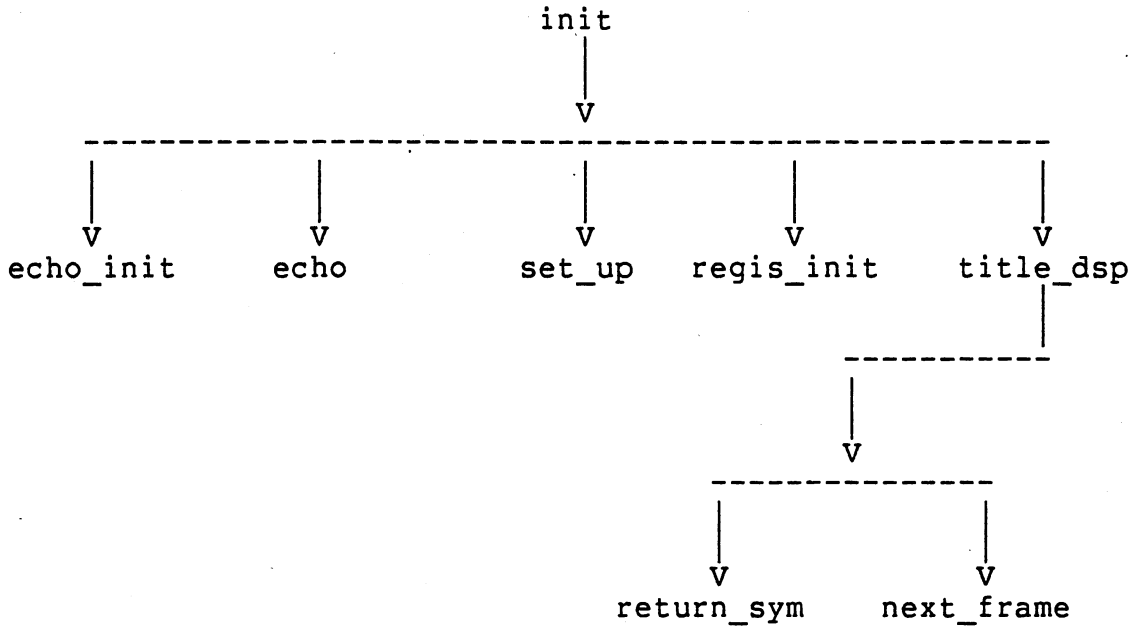


Figure 13. Init Module Schematics

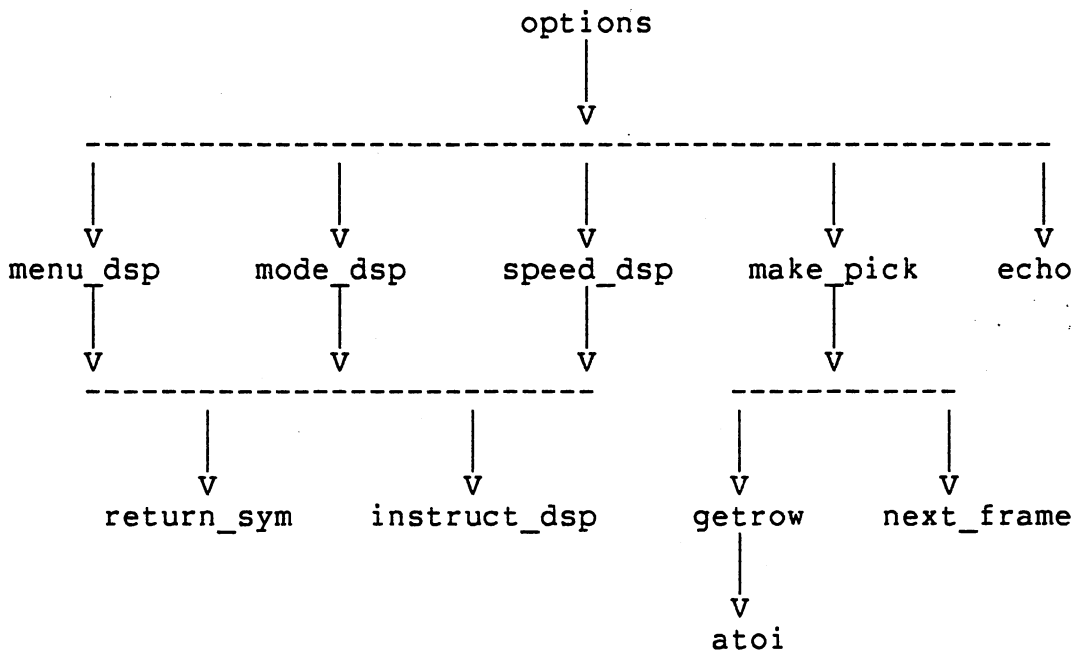


Figure 14. Options Module Schematics

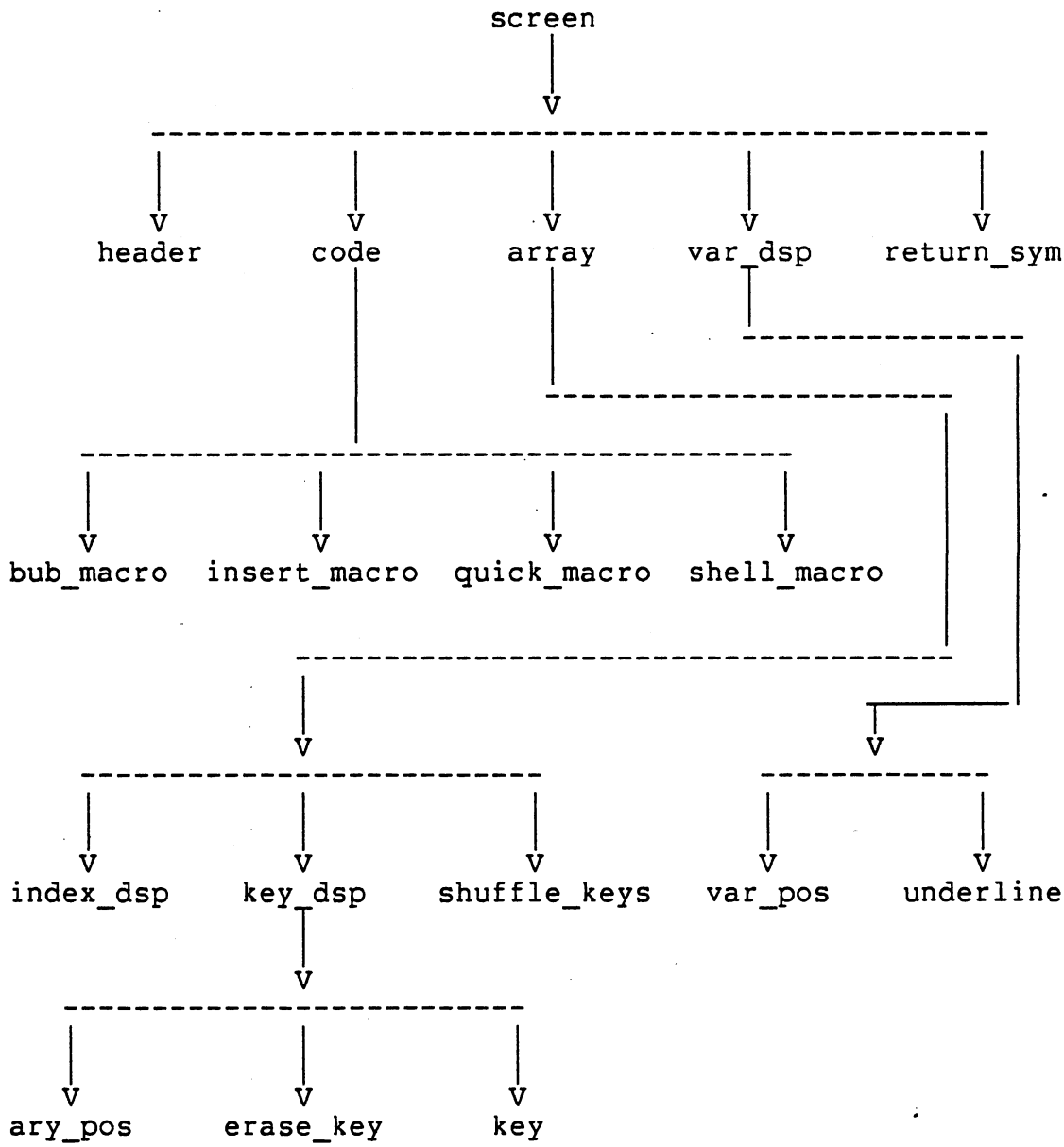


Figure 15. Screen Module Schematics

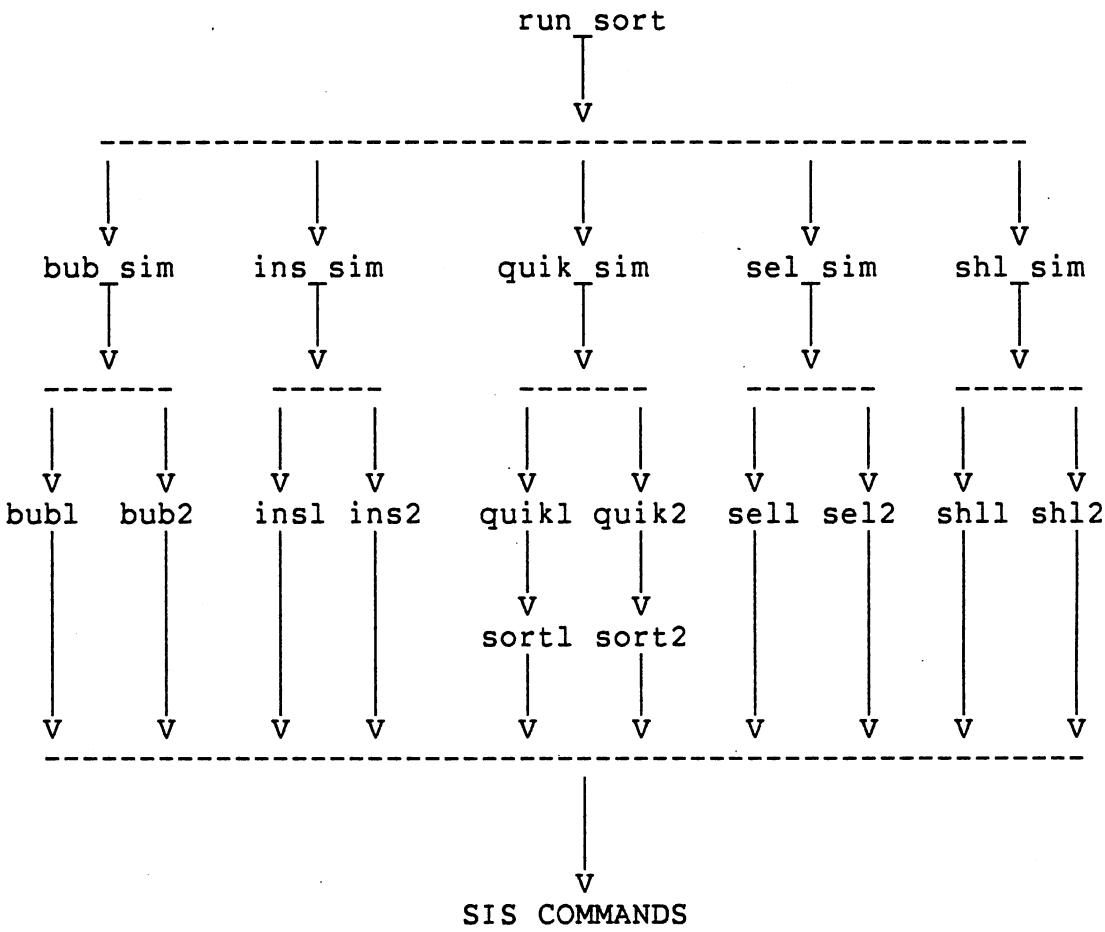


Figure 16. Run_sort Module Schematics

APPENDIX B

SIS MODULE CATALOG

The SIS Module Catalog contains a description of every module implemented in the SIS system. The modules are cataloged into directories based upon their usage within the SIS program. The directories and their members include:

1. Primary Control Modules

driver	main	run_sort
init	options	screen

2. Initialization Modules

echo_init	regis_init	set_up
init_var	reset	title_dsp

3. Menu Display Modules

atoi	make_pick	mode_dsp
getrow	menu_dsp	speed_dsp
instruct_dsp		

4. Screen Modules

bub_macro	insert_macro	shuffle_keys
code	quick_macro	underline
header	select_macro	var_dsp
index_dsp	shell_macro	var_pos

5. Run_sort Modules

bub1	ins_sim	sel_sim
bub2	quik1	shl1
bub_sim	quik2	shl2
erase_return_sym	quik_sim	shl_sim
ins1	sel1	sort1
ins2	sel2	sort2

6. SIS Command Modules

blue_line	green_var	red_line
blue_var	pop_dsp	red_var
cond	push_dsp	white_box
green_line	red_box	

7. SIS Command Utility Modules

array_dsp	green_dsp	push
blue_dsp	line_dsp	readkybd
box_dsp	par_dsp	red_dsp
clear	pause	sig_ign
gold_line	pop	value_dsp

8. Common Modules

ary_pos	interrupt	next_frame
echo	key	proceed
erase_key	key_dsp	return_sym

1. Primary Control Modules

The Primary Control Modules include the drivers to the major program components of the SIS system. Their function is to oversee the execution of the major program components by coordinating graphics modules and process modules.

Primary Control Modules

FUNCTION NAME: driver - main driving routine in SIS, invokes the menu display driver, screen driver, and run_sort driver.

MODULE TYPE: control module

USAGE: driver();

INPUT PARAMETER: N/A

RETURNED VALUE: N/A

FUNCTION NAME: init - initializes SIS and displays the title frame.

MODULE TYPE: control module

USAGE: init();

INPUT PARAMETER: N/A

RETURNED VALUE: N/A

FUNCTION NAME: main - main control module, invokes the initialization driver, the main driver, and the reset function.

MODULE TYPE: control module

USAGE: main()

INPUT PARAMETERS: a - array of keys.
N - size of array a.

RETURNED VALUE: N/A

Primary Control Modules

FUNCTION NAME: options - menu display driver.

MODULE TYPE: control module

USAGE: options();

INPUT PARAMETERS:
- sort - sorting algorithm selected.
- mode - execution mode selected.
- speed - execution speed.

RETURNED VALUE: N/A

FUNCTION NAME: run_sort - simulation driver, runs the selected sorting algorithm.

MODULE TYPE: control module

USAGE: run_sort();

INPUT PARAMETERS:
- sort - sorting algorithm selected.
- mode - execution mode selected.
- speed - execution speed.

RETURNED VALUE: N/A

FUNCTION NAME: screen - opening screen driver, displays the initial frame of the simulation.

MODULE TYPE: control module

USAGE: screen();

INPUT PARAMETER: sort - sorting algorithm selected.

RETURNED VALUE: N/A

---

2. Initialization Modules

The initialization modules constitute the first major program component. Their function is to set program parameters and display the title frame. These modules are driven by the init control module.

Initialization Modules

FUNCTION NAME: echo_init - sets echo flags.

MODULE TYPE: process module

USAGE: echo_init();

INPUT PARAMETER: N/A

RETURNED VALUE: N/A

FUNCTION NAME: init_var - sets array size and loads array with keys.

MODULE TYPE: process module

USAGE: init_var();

INPUT PARAMETERS: a - array of keys.
N - size of array a.

RETURNED VALUE: N/A

FUNCTION NAME: regis_init - initializes ReGIS for SIS.

MODULE TYPE: graphics module

USAGE: regis_init();

INPUT PARAMETER: N/A

RETURNED VALUE: N/A

Initialization Modules

FUNCTION NAME: reset - resets GIGI set-up parameters.

MODULE TYPE: process module

USAGE: reset();

INPUT PARAMETER: N/A

RETURNED VALUE: N/A

FUNCTION NAME: set_up - sets GIGI set-up parameters for SIS.

MODULE TYPE: process module

USAGE: set_up();

INPUT PARAMETER: N/A

RETURNED VALUE: N/A

FUNCTION NAME: title_dsp - displays SIS title frame.

MODULE TYPE: graphics module

USAGE: title_dsp();

INPUT PARAMETER: N/A

RETURNED VALUE: N/A

---

3. Menu Display Modules

The Menu Display Modules make up the second major program component. They are responsible for illustrating the menu displays and processing the menu selections. The modules are driven by the options control module.

Menu Display Modules

FUNCTION NAME:	atoi - converts a character string to an integer.
MODULE TYPE:	process module
USAGE:	number = atoi(string);
INPUT PARAMETER:	string - an array of characters.
RETURNED VALUE:	integer

Menu Display Modules

FUNCTION NAME: getrow - calculates current ordinate of cursor position.

MODULE TYPE: process module

USAGE: row = getrow();

INPUT PARAMETER: N/A

RETURNED VALUE: integer

FUNCTION NAME: instruct_dsp - displays the instructions for menu selection.

MODULE TYPE: graphics module

USAGE: instruct_dsp();

INPUT PARAMETER: N/A

RETURNED VALUE: N/A

FUNCTION NAME: make_pick - used when selecting options in a menu display.

MODULE TYPE: process module

USAGE: selection = make_pick(menu_number);

INPUT PARAMETER: menu_number - number of menu options.

RETURNED VALUE: integer (menu option selected)

Menu Display Modules

FUNCTION NAME: menu_dsp - displays the menu of sorting algorithms.

MODULE TYPE: graphics module

USAGE: menu_dsp();

INPUT PARAMETER: N/A

RETURNED VALUE: N/A

FUNCTION NAME: mode_dsp - displays the menu of available execution modes.

MODULE TYPE: graphics module

USAGE: mode_dsp();

INPUT PARAMETER: N/A

RETURNED VALUE: N/A

FUNCTION NAME: speed_dsp - displays the menu of available execution speeds.

MODULE TYPE: graphics module

USAGE: speed_dsp();

INPUT PARAMETER: N/A

RETURNED VALUE: N/A

---

4. Screen Modules

The Screen Modules draw the original simulation frame. They consist primarily as graphics modules. These modules compose the third major program component and are driven by the screen control module.

Screen Modules

FUNCTION NAME:	array - displays the array of keys.
MODULE TYPE:	graphics module
USAGE:	array();
INPUT PARAMETERS:	a - array of keys. N - size of array a.
RETURNED VALUE:	N/A

Screen Modules

FUNCTION NAME: bub_macro - contains the bubblesort PDL macrographs.

MODULE TYPE: graphics module

USAGE: bub_macro();

INPUT PARAMETER: mode - execution mode selected.

RETURNED VALUE: N/A

FUNCTION NAME: code - displays the selected PDL.

MODULE TYPE: graphics module

USAGE: code();

INPUT PARAMETER: sort - sorting algorithm selected.

RETURNED VALUE: N/A

FUNCTION NAME: header - display header of current algorithm.

MODULE TYPE: graphics module

USAGE: header();

INPUT PARAMETER: mode - execution mode selected.

RETURNED VALUE: N/A

Screen Modules

FUNCTION NAME: index_dsp - displays array indices.

MODULE TYPE: graphics module

USAGE: index_dsp();

INPUT PARAMETER: sort - sorting algorithm selected.

RETURNED VALUE: N/A

FUNCTION NAME: insert_macro - contains the insertion-sort PDL macrographs.

MODULE TYPE: graphics module

USAGE: insert_macro();

INPUT PARAMETER: mode - execution mode selected.

RETURNED VALUE: N/A

FUNCTION NAME: quick_macro - contains the quicksort PDL macrographs.

MODULE TYPE: graphics module

USAGE: quick_macro();

INPUT PARAMETER: mode - execution mode selected.

RETURNED VALUE: N/A

Screen Modules

FUNCTION NAME: select_macro - contains the selection-sort PDL macrographs.

MODULE TYPE: graphics module

USAGE: select_macro();

INPUT PARAMETER: mode - execution mode selected.

RETURNED VALUE: N/A

FUNCTION NAME: shell_macro - contains the Shellsort PDL macrographs.

MODULE TYPE: graphics module

USAGE: shell_macro();

INPUT PARAMETER: mode - execution mode selected.

RETURNED VALUE: N/A

FUNCTION NAME: shuffle_keys - mixes the order of the keys within the array.

MODULE TYPE: process module

USAGE: shuffle_keys();

INPUT PARAMETERS: a - array of keys.
N - size of array a.

RETURNED VALUE: N/A

Screen Modules

FUNCTION NAME: underline - used to underline display of algorithm variable.

MODULE TYPE: graphics module

USAGE: underline(num_char);

INPUT PARAMETER: num_char - number of characters to be underlined.

RETURNED VALUE: N/A

FUNCTION NAME: var_dsp - displays algorithm variables.

MODULE TYPE: graphics module

USAGE: var_dsp();

INPUT PARAMETER: mode - execution mode selected.

RETURNED VALUE: N/A

FUNCTION NAME: var_pos - moves cursor to indicated screen field.

MODULE TYPE: graphics module

USAGE: var_pos(field);

INPUT PARAMETER: field - number of a display field.

RETURNED VALUE: N/A

---

5. Run_sort Modules

The Run_sort Modules are the active participants in a running simulation. They contain the SIS algorithms which can be dynamically viewed. These modules comprise the fourth major program component and are driven by the run_sort control module.

Run_sort Modules

FUNCTION NAME: bubl - bubblesort algorithm in self-run mode.

MODULE TYPE: process module

USAGE: bubl();

INPUT PARAMETERS:
- a - array of keys.
- N - size of array a.
- temp - array key variable.
- i,j - array indices.

RETURNED VALUE: N/A

FUNCTION NAME: bub2 - bubblesort algorithm in single-step mode.

MODULE TYPE: process module

USAGE: bub2();

INPUT PARAMETERS:
- a - array of keys.
- N - size of array a.
- temp - array key variable.
- i,j - array indices.

RETURNED VALUE: N/A

FUNCTION NAME: bub_sim - invokes bubblesort algorithm in selected execution mode.

MODULE TYPE: process module

USAGE: bub_sim();

INPUT PARAMETER: mode - execution mode selected.

RETURNED VALUE: N/A

Run_sort Modules

FUNCTION NAME: erase_return_sym - erases the return symbol from the display screen.

MODULE TYPE: process module

USAGE: erase_return_sym();

INPUT PARAMETER: N/A

RETURNED VALUE: N/A

FUNCTION NAME: ins1 - insertionsort algorithm in self-run mode.

MODULE TYPE: process module

USAGE: ins1();

INPUT PARAMETERS:
a - array of keys.
N - size of array a.
temp - array key variable.
i,j - array indices.

RETURNED VALUE: N/A

FUNCTION NAME: ins2 - insertionsort algorithm in single-step mode.

MODULE TYPE: process module

USAGE: ins2();

INPUT PARAMETERS:
a - array of keys.
N - size of array a.
temp - array key variable.
i,j - array indices.

RETURNED VALUE: N/A

Run_sort Modules

FUNCTION NAME: ins_sim - invokes insertionsort algorithm in selected execution mode.

MODULE TYPE: process module

USAGE: ins_sim();

INPUT PARAMETER: mode - execution mode selected.

RETURNED VALUE: N/A

FUNCTION NAME: quik1 - quicksort algorithm in self-run mode.

MODULE TYPE: process module

USAGE: quik1();

INPUT PARAMETER: N - size of array.

RETURNED VALUE: N/A

FUNCTION NAME: quik2 - quicksort algorithm in single-step mode.

MODULE TYPE: process module

USAGE: quik2();

INPUT PARAMETER: N - size of array.

RETURNED VALUE: N/A

Run_sort Modules

FUNCTION NAME: quik_sim - invokes quicksort algorithm in selected execution mode.

MODULE TYPE: process module

USAGE: quik_sim();

INPUT PARAMETER: mode - execution mode selected.

RETURNED VALUE: N/A

FUNCTION NAME: sel1 - selectionsort algorithm in self-run mode.

MODULE TYPE: process module

USAGE: sel1();

INPUT PARAMETERS:
a - array of keys.
N - size of array a.
temp - array key variable.
i,j,k - array indices.

RETURNED VALUE: N/A

FUNCTION NAME: sel2 - selectionsort algorithm in single-step mode.

MODULE TYPE: process module

USAGE: sel2();

INPUT PARAMETERS:
a - array of keys.
N - size of array a.
temp - array key variable.
i,j,k - array indices.

RETURNED VALUE: N/A

Run_sort Modules

FUNCTION NAME: sel_sim - invokes selectionsort algorithm in selected execution mode.

MODULE TYPE: process module

USAGE: sel_sim();

INPUT PARAMETER: mode - execution mode selected.

RETURNED VALUE: N/A

FUNCTION NAME: shl1 - Shellsort algorithm in self-run mode.

MODULE TYPE: process module

USAGE: shl1();

INPUT PARAMETERS:
a - array of keys.
N - size of array a.
temp - array key variable.
i,j,s - array indices.

RETURNED VALUE: N/A

FUNCTION NAME: shl2 - Shellsort algorithm in single-step mode.

MODULE TYPE: process module

USAGE: shl2();

INPUT PARAMETERS:
a - array of keys.
N - size of array a.
temp - array key variable.
i,j,s - array indices.

RETURNED VALUE: N/A

Run_sort Modules

FUNCTION NAME: shl_sim - invokes selectionsort algorithm in selected execution mode.

MODULE TYPE: process module

USAGE: shl_sim();

INPUT PARAMETER: mode - execution mode selected.

RETURNED VALUE: N/A

FUNCTION NAME: sort1 - quicksort subroutine invoked in self-run mode.

MODULE TYPE: process module

USAGE: sort1();

INPUT PARAMETERS:
a - array of keys.
N - size of array a.
temp,x - array key variables.
i,j,l,r - array indices.

RETURNED VALUE: N/A

FUNCTION NAME: sort2 - quicksort subroutine invoked in single-step mode.

MODULE TYPE: process module

USAGE: sort2();

INPUT PARAMETERS:
a - array of keys.
N - size of array a.
temp,x - array key variables.
i,j,l,r - array indices.

RETURNED VALUE: N/A

---

6. SIS Command Modules

This section is a library of the SIS Commands used in the program. They are invoked only by the run_sort modules of the previous section. Their purpose is to simplify the programming of an algorithm simulation.

SIS Command Modules

FUNCTION NAME: blue_line - displays line of PDL in blue.

MODULE TYPE: process module

USAGE: blue_line(line);

INPUT PARAMETER: line - line number of PDL.

RETURNED VALUE: N/A

FUNCTION NAME: blue_var - displays variable in blue.

MODULE TYPE: process module

USAGE: blue_var('var');

INPUT PARAMETER: var - mnemonic character representing an algorithm variable.

RETURNED VALUE: N/A

SIS Command Modules

FUNCTION NAME: cond - displays the result of a conditional expression.

MODULE TYPE: graphics module

USAGE: cond(bool);

INPUT PARAMETER: bool - integer with a boolean value.

RETURNED VALUE: N/A

FUNCTION NAME: green_line - displays line of PDL in blue.

MODULE TYPE: process module

USAGE: green_line(line);

INPUT PARAMETER: line - line number of PDL.

RETURNED VALUE: N/A

FUNCTION NAME: green_var - displays variable in green.

MODULE TYPE: process module

USAGE: green_var('var');

INPUT PARAMETER: var - mnemonic character representing an algorithm variable.

RETURNED VALUE: N/A

SIS Command Modules

FUNCTION NAME: pop_dsp - pops all variables off of the display stack and highlights them in green.

MODULE TYPE: process module

USAGE: pop_dsp();

INPUT PARAMETER: N/A

RETURNED VALUE: N/A

FUNCTION NAME: push_dsp - pushes a variable onto the display stack and highlights the variable in blue.

MODULE TYPE: process module

USAGE: push_dsp('var');

INPUT PARAMETER: var - mnemonic character representing an algorithm variable.

RETURNED VALUE: N/A

FUNCTION NAME: red_box - encloses a portion of the array display in a red box.

MODULE TYPE: process module

USAGE: red_box(lower,upper);

INPUT PARAMETERS: lower - lower bound of the red box.
upper - upper bound of the red box.

RETURNED VALUE: N/A

SIS Command Modules

FUNCTION NAME: red_line - displays line of PDL in red.

MODULE TYPE: process module

USAGE: red_line(line);

INPUT PARAMETER: line - line number of PDL.

RETURNED VALUE: N/A

FUNCTION NAME: red_var - displays variable in red.

MODULE TYPE: process module

USAGE: red_var('var');

INPUT PARAMETER: var - mnemonic character representing an algorithm variable.

RETURNED VALUE: N/A

FUNCTION NAME: white_box - encloses a portion of the array display in a white box.

MODULE TYPE: process module

USAGE: white_box(lower,upper);

INPUT PARAMETERS: lower - lower bound of the white box.
upper - upper bound of the white box.

RETURNED VALUE: N/A

---

7. SIS Command Utility Modules

SIS Command Utility Modules perform the screen mechanics of the SIS Commands. They play a supplemental role to SIS Commands and remain unseen to the programmer. Many of these utility modules are invoked by more than one SIS Command.

SIS Command Utility Modules

FUNCTION NAME: array_dsp - branches to cell of array key display, invoked by value_dsp subroutine.

MODULE TYPE: process module

USAGE: array_dsp(var);

INPUT PARAMETERS:

a	-	array of keys.
N	-	size of array a.
i,j,r,l,k,s	-	array indices.
var	-	mnemonic representation of algorithm variable.

RETURNED VALUE: N/A

FUNCTION NAME: blue_dsp - sets text and graphics to blue and is invoked by blue_var and push_dsp cmds.

MODULE TYPE: process module

USAGE: blue_dsp();

INPUT PARAMETER: N/A

RETURNED VALUE: N/A

FUNCTION NAME: box_dsp - SIS box command utility func.

MODULE TYPE: graphics module

USAGE: box_dsp(lower,upper);

INPUT PARAMETERS:

N	-	size of the array.
lower	-	lower bound of the box.
upper	-	upper bound of the box.

RETURNED VALUE: N/A

SIS Command Utility Modules

FUNCTION NAME: clear - clears the display stack, called by pop and push cmds.

MODULE TYPE: process module

USAGE: clear();

INPUT PARAMETER: N/A

RETURNED VALUE: N/A

FUNCTION NAME: gold_line - displays line of PDL in gold (yellow), invoked by green_line commands.

MODULE TYPE: process module

USAGE: gold_line(line);

INPUT PARAMETER: line - line number of PDL.

RETURNED VALUE: N/A

FUNCTION NAME: green_dsp - sets text and graphics to green, invoked by green_var and pop_dsp commands.

MODULE TYPE: process module

USAGE: green_dsp();

INPUT PARAMETER: N/A

RETURNED VALUE: N/A

SIS Command Utility Modules

FUNCTION NAME: line_dsp - SIS line command utility func.

MODULE TYPE: graphics module

USAGE: line_dsp(line);

INPUT PARAMETER: line - line number of PDL.

RETURNED VALUE: N/A

FUNCTION NAME: par_dsp - posts the current value of a variable, invoked by value_dsp.

MODULE TYPE: process module

USAGE: par_dsp(var);

INPUT PARAMETERS:

a	-	array of keys.
N	-	size of array a.
i,j,r,l,k,s	-	array indices.
var	-	mnemonic repesentation of algorithm variable.

RETURNED VALUE: N/A

FUNCTION NAME: pause - regulates simulation speed, invoked by green_line command.

MODULE TYPE: graphics module

USAGE: pause();

INPUT PARAMETERS:

mode	-	execution mode selected.
speed	-	execution speed selected.

RETURNED VALUE: N/A

SIS Command Utility Modules

FUNCTION NAME: pop - pops value from top of display stack, invoked by pop_dsp.

MODULE TYPE: process module

USAGE: pop();

INPUT PARAMETERS: sp - stack pointer.
par - parameter stack.

RETURNED VALUE: character (value from top of stack returned or error symbol returned if stack empty).

FUNCTION NAME: push - push_dsp command utility func.

MODULE TYPE: process module

USAGE: push(var);

INPUT PARAMETERS: sp - stack pointer.
par - parameter stack.
var - mnemonic representation of an algorithm variable.

RETURNED VALUE: error symbol returned on stack overflow.

FUNCTION NAME: readkybd - looks for input from keyboard, invoked by pause function.

MODULE TYPE: process module

USAGE: readkybd();

INPUT PARAMETERS: mode - execution mode selected.
speed - execution speed selected.

RETURNED VALUE: N/A

SIS Command Utility Modules

FUNCTION NAME: red_dsp - sets text and graphics to red, invoked by red_var cmds.

MODULE TYPE: process module

USAGE: red_dsp();

INPUT PARAMETER: N/A

RETURNED VALUE: N/A

FUNCTION NAME: sig_ign - dummy function to process alarm interrupt, invoked by C func 'signal' in readkybd.

MODULE TYPE: process module

USAGE: signal(SIGALRM,sig_ign);

INPUT PARAMETER: N/A

RETURNED VALUE: N/A

FUNCTION NAME: value_dsp - variable graphics display driver, invoked by blue_var and pop_dsp commands.

MODULE TYPE: process module

USAGE: value_dsp(var);

INPUT PARAMETERS: mode - execution mode selected.
var - mnemonic character representing an algorithm variable.

RETURNED VALUE: N/A

---

8. Common Modules

Common Modules are defined as those modules which are called by modules of different major program components. Common Modules perform an assortment of functions and may be summoned frequently in a SIS presentation.

FUNCTION NAME: ary_pos - sets graphics cursor to the appropriate array position.

MODULE TYPE: graphics module

USAGE: ary_pos(pos);

INPUT PARAMETER: pos - position corresponding to an array index.

RETURNED VALUE: N/A

FUNCTION NAME: echo - regulates the terminal echo.

MODULE TYPE: process module

USAGE: echo(flag);

INPUT PARAMETER: flag - specifies terminal echo cond, values it can assume: 'ON' or 'OFF'.

RETURNED VALUE: N/A

FUNCTION NAME: erase_key - erases array key found in the current cell position.

MODULE TYPE: graphics module

USAGE: erase_key();

INPUT PARAMETER: N/A

RETURNED VALUE: N/A

FUNCTION NAME: interrupt - handles signal interrupts generated by the 'BREAK' key.

MODULE TYPE: process module

USAGE: signal(SIGINT,interrupt);

INPUT PARAMETER: N/A

RETURNED VALUE: N/A

FUNCTION NAME: key - generates specified array key at current location.

MODULE TYPE: graphics module

USAGE: key(array_key);

INPUT PARAMETER: array_key - array key variable.

RETURNED VALUE: N/A

FUNCTION NAME: key_dsp - array key display driver, calls ary_pos, erase_key, and key.

MODULE TYPE: process module

USAGE: key_dsp(pos,array_key);

INPUT PARAMETERS: array_key - array key variable.
pos - position corresponding to array index.

RETURNED VALUE: N/A

Common Modules

FUNCTION NAME: next_frame - an infinite loop, exited only when carriage return is received.

MODULE TYPE: process module

USAGE: next_frame();

INPUT PARAMETER: N/A

RETURNED VALUE: N/A

FUNCTION NAME: proceed - displays 'new selection' question and processes user response to question, invoked by interrupt function.

MODULE TYPE: process module

USAGE: proceed();

INPUT PARAMETER: N/A

RETURNED VALUE: the response to the question is returned: 'YES' or 'NO'.

FUNCTION NAME: return_sym - generates the return symbol display.

MODULE TYPE: graphics module

USAGE: return_sym();

INPUT PARAMETER: N/A

RETURNED VALUE: N/A

APPENDIX C

SIS USER'S GUIDE

Introduction

SIS is a sorting instruction simulator. Its purpose is to provide you, the viewer, the opportunity to witness the execution of a sorting algorithm. The design of the simulator allows you to choose:

1. One of five sorting algorithms
2. One of two modes of execution
3. One of three execution speeds

Briefly page through this accompanying guide before using SIS. Knowing the options available to you in advance will enhance your learning and increase your enjoyment. SIS is programmed with several convenient features. With a few simple keystrokes you can change the execution speed while the simulation is in progress, temporarily halt algorithm execution, or abort the current simulation and begin another.

Colorful graphics highlight important points in an executing sorting algorithm. Through color change, you can follow the actions resulting from each executing instruction. You are always aware of what instruction is executing, what variables are being affected, and the current

order of the keys being sorted as they are displayed in a rectangular array.

Login Procedure

SIS is currently available on the Perkin-Elmer 3230 Computer associated with the Computing and Information Science (COMSC) Department. Having already acquired an account name and password from the COMSC Department:

1. Secure a GIGI Terminal accessing the Perkin-Elmer. One is located on the second floor of the Mathematical Science Building. (SIS will only operate properly on GIGI Terminals).

2. Repeatedly press the "RETURN" key until "login:" appears on the screen.

3. Type your account name and press "RETURN". "Password:" should appear, if it does not then go back to step 2.

4. Enter your password and press the "RETURN" key. (Your password will not be displayed.) Wait for the system to respond with a "%" prompt.

5. To invoke the SIS program, type:

   /u/fac/mjf/sis.jsr/SIS

   and press the "RETURN" key.

If these steps are followed correctly, the SIS title will appear on the display screen (Figure 17).

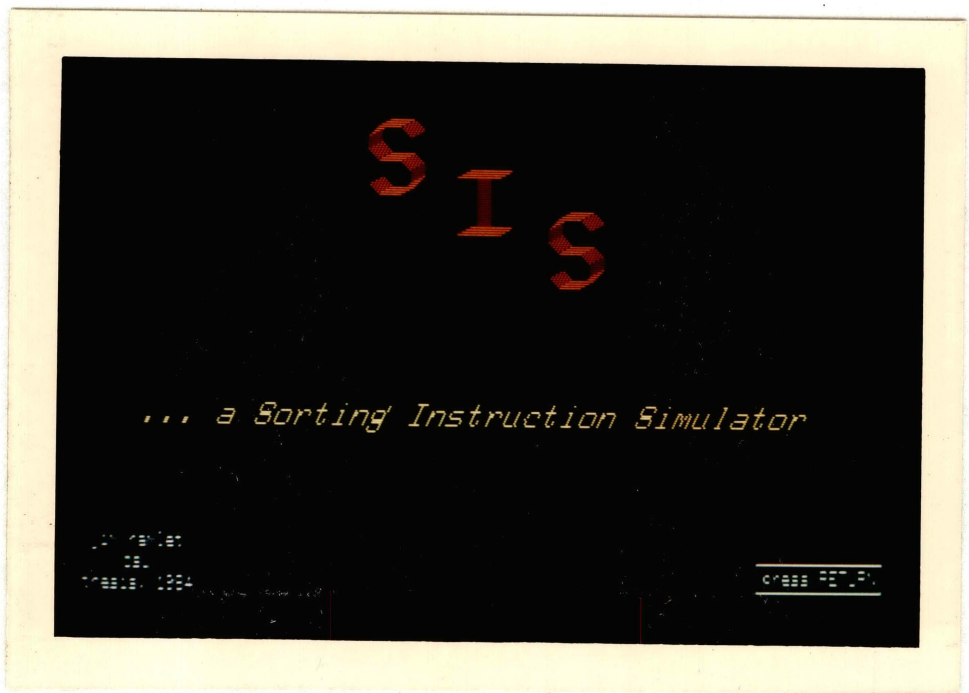


Figure 17. The SIS Title Display

Option Selection

With the SIS title presented on the display screen, press the "RETURN" key. You have now entered the menu selection portion of the SIS program. A menu of sorting algorithms should appear.

SORTING ALGORITHM - The blinking cursor situated near the center of the screen is the option selector. Locate the up and down arrow keys [↓] [↑] on the top row of the keyboard. Using these keys, align the option selector with the

sorting algorithm option that you desire to view. Press the "RETURN" key to enter your choice.

EXECUTION MODE - After choosing the sorting algorithm to simulate, you must decide how you want it to be presented. A menu of execution modes is displayed next with the options: 1) self-run mode and 2) single-step mode.

The self-run mode simulates the selected algorithm at a fixed rate of speed. This mode is beneficial when wanting to learn about the algorithm's overall behavior. Main attention centers on the array display where you will be observing the swapping characteristics of the algorithm.

If a more detailed account of the sorting algorithm is desired, then the single-step mode is recommended. Single-step mode proceeds through the algorithm one instruction at a time under your direction. Emphasis is placed upon variable values, program flow, and results of individual instructions.

As in the algorithm selection process, use the arrow keys in conjunction with the option selector to choose the execution mode. Press the "RETURN" key to enter your choice.

EXECUTION SPEED - If you have chosen the single-step mode of execution, omit this step and proceed immediately to Viewing the Simulation. Having elected to watch a simulation in self-run mode, you are presented with a menu of execution speeds: fast, medium, and slow. Following the previously mentioned menu selection procedure, choose an ap-

propriate speed and press the "RETURN" key to enter your choice.

Viewing the Simulation

You are now ready to begin the simulation. The simulator display should appear on the screen as shown in Figure 18 (self-run mode) and Figure 19 (single-step mode). At the top of the screen in white is the display header. It reminds you of your sorting algorithm and execution mode selections. The left portion of the screen holds the pseudocode of your sorting algorithm. The array display containing the keys to be sorted appears near the center of the screen. If you elected single-step mode, a display of algorithm variables is posted along the right side of the screen.

SIS employs a color scheme to aid you in algorithm comprehension. Table III describes the color codes used by SIS during a running simulation. Depending upon the sorting algorithm and execution mode, not all of the color codes will be used. For instance, yellow is not used in self-run mode simulations and red only appears when recursive algorithms (quicksort) are selected.

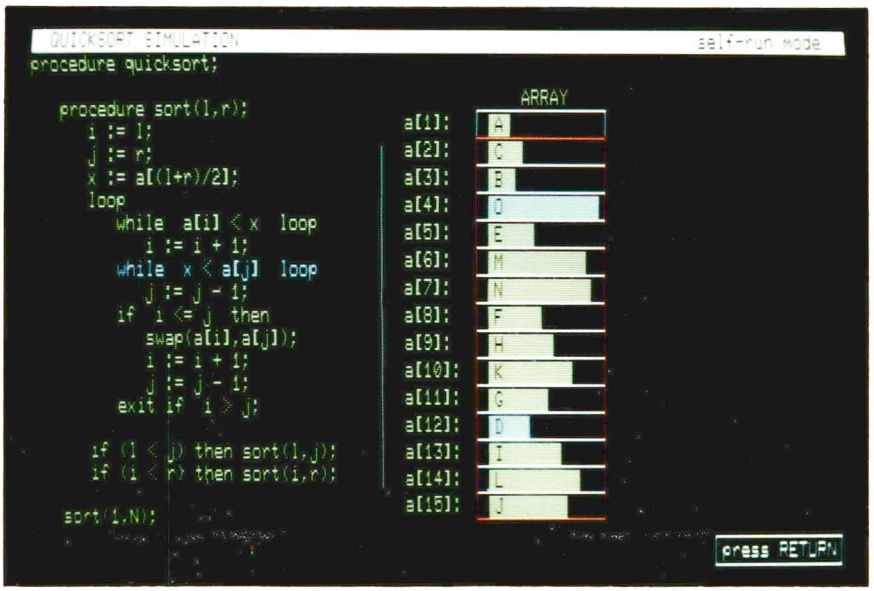


Figure 18. A Self-Run Mode Simulation Display

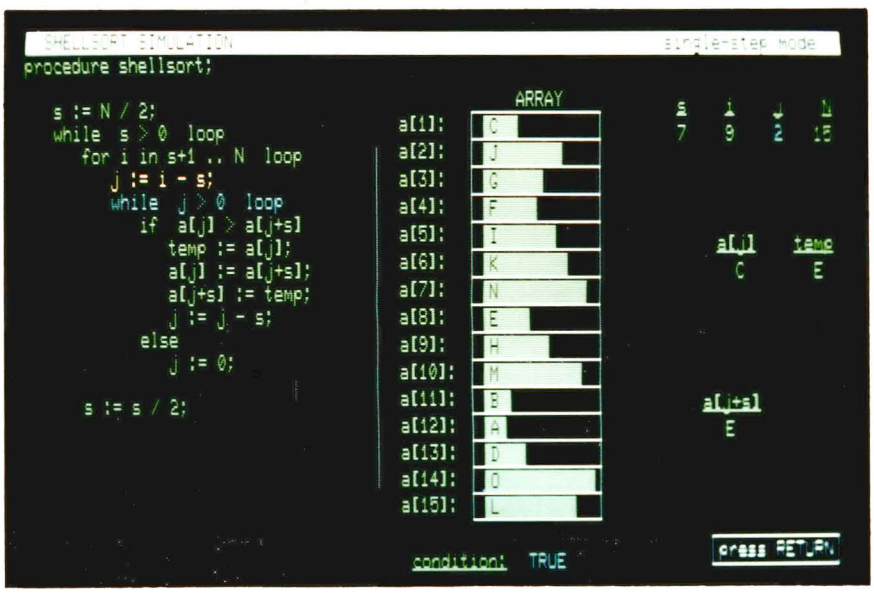


Figure 19. A Single-Step Mode Simulation Display

TABLE III

SIS DISPLAY COLOR CODES

Color	Purpose
Green	General Display Color
Yellow	Highlights Previous Instruction
Blue (Cyan)	Highlights Current Instruction, Keys, and Variables
Red	Highlights Recursion
White	Highlights Display Header and Return Symbol
Dark	Background Color

Please note the white return symbol found in the lower right hand corner of the display screen. Whenever this symbol appears, press the "RETURN" key when you are ready to continue the simulation.

The "BREAK" key is pressed when you want to leave the current simulation prematurely. You will then be given the option of either exiting the SIS program or choosing another sorting algorithm to view. Respond to the question by typing either "yes" or "no" and press the "RETURN" key. The "BREAK" key may be used anytime during the SIS program, even during menu selection.

Please proceed to the appropriate execution mode for further directions.

****

SELF-RUN MODE

****

In this mode, SIS provides you with a set of function keys. Use of these function keys during a simulation enables you to change execution speeds or pause the action of the algorithm demonstration. Please see Table IV for a complete description.

Press the "RETURN" key to initiate the simulation. Notice, that as each line of code is executed:

1. It is highlighted in blue.
2. All array keys affected by that instruction are also highlighted in blue.

Center your concentration upon the array display. Observing how the individual array keys are compared and exchanged will give you insight into the characteristics of the sorting algorithm. Questions to consider while viewing include:

1. What is the general direction of the comparisons being made? (up or down)
2. How far apart are the array keys being compared? Are they adjacent or several positions apart?
3. Is an additional array position required in this particular sorting algorithm?
4. How efficient is this sorting algorithm, that is, are there many unnecessary comparisons?
5. Do you see any general trends?

Remember, if you desire to leave the simulation before it reaches its conclusion, press the "BREAK" key. When you are finished watching SIS, please go to the Logout Procedure.

TABLE IV

SUMMARY OF THE PROGRAM FUNCTIONS

Key	Function
'F' or 'f'	Change Execution Speed To 'Fast'
'M' or 'm'	Change Execution Speed To 'Medium'
'S' or 's'	Change Execution Speed To 'Slow'
'P' or 'p'	Pause Or Temporarily Halt Simulation, Press 'RETURN' Key To Resume Execution
'BREAK'	Abort Current Simulation

****

SINGLE-STEP MODE

****

In this mode, progress through the simulation is accomplished by a repeated pressing of the "RETURN" key. With each keystroke, the next instruction is executed. Throughout the demonstration, use Table III as a guide for interpreting the various colored elements of the display.

Press the "RETURN" key. This initiates the simulation. While stepping through the demonstration focus your attention upon the blue highlighting in the display. The blue color indicates currently affected elements, including the variable values, line of code, and array keys. Questions to consider while viewing are:

1. What variable values changed when the current instruction was executed? Why?
2. What array keys are affected by the current instruction? Why?
3. Examine the program flow of execution. Why was the current instruction executed after the previous instruction?
4. If the current instruction is an "if statement", why is its posted result TRUE (or FALSE)?

Remember, if you desire to leave the simulation before it reaches its conclusion, press the "BREAK" key. When you are finished watching SIS, please go to the Logout Procedure.

Logout Procedure

When you have concluded viewing SIS, please do not leave the terminal without logging off the system. If you are still in the SIS program:

1. Press the "BREAK" key. The question "Do you wish to make another selection?" will then follow on the screen.

2. Respond to the question by typing "no" and press the "RETURN" key. A "%" prompt should appear.

You have now exited the SIS program. To log off the system type "logout" and press the "RETURN" key. If you are successful, "login:" will be displayed upon the screen.

APPENDIX D

VT125 CONVERSION REQUIREMENTS

The VT125 utilizes only four out of the eight colors provided by ReGIS: dark, blue, red, and green. This limited selection is inadequate for SIS, which requires six colors. Although a stripped down version of SIS could be built around four colors, replacing ReGIS color commands with the HLS color specifiers available on the VT125 is a better alternative. The HLS color system offers 64 different colors and would not impose any color restrictions on the presentation.

The up and down arrow keys, used to manipulate the pointer in a SIS menu display, are enabled with GIGI-specific device control strings (DCS). Because DCS are not acknowledged by a VT125, the menu display is rendered inoperable. Unfortunately, there is no simple conversion. Another method for choosing options could be devised to allow the student a way of entering simulation parameters.

An annoyance with running SIS on a VT125 is that the graphics cursor cannot be disabled. The cursor is present whenever the VT125 is processing ReGIS graphics commands. This is not a critical issue, but the cursor could become distracting during a simulation.

VITA 2

James Stephen Ramlet

Candidate for the Degree of

Master of Science

Thesis: SIS: A SORTING INSTRUCTION SIMULATOR

Major Field: Computing and Information Science

Biographical:

Personal Data: Born in Yankton, South Dakota, April 16, 1954, son of Robert L. and Dorothy A. Ramlet. Married to Sandra S. Burkhardt on March 20, 1982.

Education: Graduated from Larkin High School, Elgin, Illinois, in June, 1972; received Bachelor of Science Degree in Biology from Oral Roberts University in May, 1977; completed requirements for the Master of Science degree at Oklahoma State University, in December, 1984.

Professional Experience: Research Assistant, Oral Roberts University, October, 1977 to August, 1982; Teaching Assistant, Oklahoma State University, August, 1982 to January, 1982; Programmer, Los Alamos National Laboratory, June, 1983 to August, 1983.