118.

CONTROL DATA

PLATO

AUTHOR LANGUAGE REFERENCE MANUAL

CONTROL DATA

CONTROL DATA PLATO AUTHOR LANGUAGE REFERENCE MANUAL

CONTROL DATA

REVISION RECORD

REVISION	DESCRIPTION
01	Preliminary manual.
(7-18-74)	
A	Manual released. This printing obsoletes the previous edition.
(9-19-75)	
B	Manual revised to reflect changes and corrections resulting from product development. This revision includes
(3-31-77)	minor editorial changes and adds index. This edition obsoletes all previous editions.
C	Manual revised to reflect changes and corrections resulting from product development. This edition obsoletes all
(4-21-78)	previous editions.

Publication No.
97405100

REVISION LETTERS I, O, Q AND X ARE NOT USED

Printed in the United States of America

Address comments concerning this manual to:

Control Data Corporation
Publications and Graphics Division
4201 North Lexington Avenue
St. Paul, Minnesota 55112

or use Comment Sheet in the back of this manual.

LIST OF EFFECTIVE PAGES

New features, as well as changes, deletions, and additions to information in this manual, are indicated by bars in the margins or by a dot near the page number if the entire page is affected. A bar by the page number indicates pagination rather than content has changed.

PAGE	REV
Front cover	-
Title page	-
ii	C
iii/iv	C
v/vi	C
vii	C
viii	C
ix	C
x	C
1-1	C
1-2	C
2-1	C
3-1	C
4-1	C
5-1	C
5-2	C
5-3	C
5-4	C
5-5	C
5-6	C
5-7	C
5-8	C
5-9	C
5-10	C
5-11	C
5-12	C
5-13	C
5-14	C
5-15	C
5-16	C
5-17	C
5-18	C
5-19	C
5-20	C
5-21	C
5-22	C
5-23	C
5-24	C
5-25	C
5-26	C
5-27	C
5-28	C
5-29	C
5-30	C
5-31	C
5-32	C
5-33	C
5-34	C
5-35	C
5-36	C
5-37	C
5-38	C
5-39	C
5-40	C
5-41	C
5-42	C
5-43	C
5-44	C
5-45	C

PAGE	REV
5-46	C
5-47	C
5-48	C
5-49	C
5-50	C
5-51	C
5-52	C
5-53	C
5-54	C
5-55	C
5-56	C
5-57	C
5-58	C
5-59	C
5-60	C
5-61	C
5-62	C
5-63	C
5-64	C
5-65	C
5-66	C
5-67	C
5-68	C
5-69	C
5-70	C
5-71	C
5-72	C
5-73	C
5-74	C
5-75	C
5-76	C
5-77	C
5-78	C
5-79	C
5-80	C
5-81	C
6-1	C
6-2	C
6-3	C
7-1	B
7-2	C
7-3	C
7-4	C
8-1	C
8-2	C
8-3	C
8-4	C
8-5	C
8-6	C
8-7	C
8-8	C
8-9	C
8-10	C
8-11	C
8-12	C
8-13	C
8-14	C
8-15	C
8-16	C

PAGE	REV
8-17	C
8-18	C
8-19	C
8-20	C
9-1	B
9-2	C
9-3	C
9-4	C
9-5	C
9-6	C
9-7	C
9-8	C
9-9	C
9-10	C
9-11	C
9-12	C
9-13	C
9-14	C
10-1	B
10-2	C
10-3	C
10-4	C
10-5	C
10-6	C
10-7	C
10-8	C
10-9	C
10-10	C
10-11	C
10-12	C
10-13	C
10-14	C
10-15	C
10-16	C
10-17	C
11-1	C
11-2	C
11-3	C
11-4	C
11-5	C
11-6	C
11-7	C
11-8	C
11-9	C
11-10	C
11-11	C
11-12	C
11-13	C
12-1	C
12-2	C
13-1	C
13-2	C
13-3	C
13-4	C
13-5	C
14-1	C
14-2	C
15-1	C
15-2	C

PAGE	REV
15-3	C
A-1	B
A-2	B
A-3	C
A-4	B
B-1	B
B-2	C
B-3	C
B-4	C
B-5	C
B-6	C
B-7	C
C-1	B
C-2	B
C-3	B
C-4	B
D-1	B
D-2	C
Index-1	C
Index-2	C
Index-3	C
Index-4	C
Index-5	C
Index-6	C
Comment sheet	C
Back cover	-

PAGE	REV

PREFACE

The CONTROL DATA® PLATO (Programmed Logic for Automatic Teaching Operations) system is a multimedia computer-based educational delivery system.

This manual describes the PLATO author language used by the PLATO system. It is intended as a reference manual for use by persons familiar with the author language.

Sections 1 through 4 provide a general introduction to the PLATO system, lesson structure, and execution of the author language lessons.

Section 5 gives a brief, ready-reference description of each PLATO author language instruction.

Sections 6 through 14 explain in detail each of the author language instructions.

Readers with no previous knowledge of the author language are encouraged to direct their initial attentions to the following sections of the manual.

Sections 1 through 4	Introduction
Sections 6 and 7	Variables, expressions, and functions
Section 9	Displays
Section 10	Lesson structure
Section 11	Handling responses

Familiarity with the PLATO User's Guide may be helpful to the user of this manual.

DISCLAIMER

This product is intended for use only as described in this document. Control Data cannot be responsible for the proper functioning of undescribed features or parameters.

RELATED PUBLICATIONS

The following publications are referenced and related PLATO manuals.

Control Data Publication	Publication No.
PLATO Terminal User's Guide	97404800
PLATO Operator's Guide	97405200
PLATO User's Guide	97405900
PLATO Author Language Instruction Formats	97406600
PLATO System Overview	97406700

CONTENTS

APPENDIXES

INDEX

FIGURES

TABLES

The PLATO author language enables the development and presentation of course material using the PLATO system.

SYSTEM COMPONENTS

Two types of components necessary for operation of a PLATO system are the actual equipment and the programming that controls the equipment. The equipment consists of a central computer, extended core storage (ECS), mass storage, communications equipment, and PLATO terminals. The programming consists of the operating system, the PLATO system, and the PLATO author language.

The central computer is the component that actually executes the lessons, as well as performing other related tasks. It consists of a central processing unit, which performs the actual execution, and central memory, which contains the lesson being executed.

ECS is used to store records and any lessons that are being used, because information can be transferred quickly between ECS and central memory.

Mass storage has a larger capacity than ECS, but the associated information transfer rate is much less. Hence, mass storage is used primarily for storing lessons that are not currently in use. Mass storage consists of magnetic disks and is therefore sometimes called disk storage, disk memory, or simply disk.

The communications equipment allows the central computer and the PLATO terminals to exchange information.

The PLATO terminal is the device used by the author or student for constructing or executing lessons. Additionally, instructors use the terminal to construct the necessary courseware (lesson selection and lesson sequence lists). A description of the PLATO terminal and its use is given in the PLATO Terminal User's Guide.

The operating system performs such functions as handling the transfer of information between components of the system.

The PLATO system controls the actions of the computer as they relate specifically to PLATO. This programming allows the author to construct a lesson, specify information about the lesson, control lesson execution, and perform various associated tasks.

The author language condenser is the part of the PLATO system that converts author language instructions (also called source code) into instructions that can be interpreted or executed by the computer.

The PLATO author language is composed of a number of instructions with which an author can construct a lesson for presentation to students.

SYSTEM OPERATION

PLATO is a time-sharing system. This means that a lesson that is being used, whether by an author or a student, is not always in actual execution. Instead, each lesson is given a certain amount of time in the central processing unit and then waits while other lessons execute. Because of the speed of the computer, the delays are not usually discernible to the user. The period of time during which a lesson is using the central processing unit is called a time-slice.

A condensed copy of the lesson is kept in ECS whenever the lesson is being used. At the beginning of the time-slice, the unit to be executed is copied into the central memory for execution. The unit is not copied back to ECS at the end of the time-slice, since the ECS copy was not destroyed. The copy in central memory is simply overwritten by the next lesson using the central processor.

When a lesson is not being used, it is kept on mass storage, allowing room in ECS for lessons being used. Because the total amount of ECS is limited, all lessons cannot be kept in ECS. Hence, the mass storage is used for the lessons that are not currently needed. Mass storage is not suitable for the temporary storage of lessons between time-slices because of the longer access time and slower information transfer rate in comparison with ECS.

Only one copy of a lesson is required to be in ECS, regardless of the number of students using the lesson. This results in a considerable saving of ECS space when several students are using a lesson concurrently. Since there are records and other data maintained for each student, more ECS space is required when 15 students are executing the same lesson concurrently than is used when only one student is executing the lesson. However, the space required for additional users of a lesson is much less than would be required if each user had a separate copy of the lesson.

NOTATION

To facilitate differentiation between text and author language instructions, instructions are set off from normal text with hyphens (–).

Names of function keys are in full capitals, while author language instructions are always given in lowercase letters. For example, NEXT refers to the function key, located to the right of the alphabetic keyboard, while –next– refers to the instruction that specifies the unit to be executed after completion of the current unit. Where an uppercase function key does not have a specific name, it is referred to as in SHIFT BACK. Uppercase keys that have specific names, such as FONT and TERM, do not use the SHIFT prefix.

Author language instructions that refer to an uppercase function key that does not have a specific name have a 1 at the end of the key name. Thus, the instruction

help1 **unitname**

enables the SHIFT HELP keys, while the instruction

help **unitname**

enables the HELP key.

When reference to a full instruction (that is, both the command and the tag of the instruction) is made in text, the command and tag portions of the instruction are separated by spaces, with the hyphens enclosing the entire instruction. An example is the –long 1– instruction. Additionally, tags shown in **bold** typeface are abbreviated descriptions of the tags and not the actual tags to be entered. Actual tags to be entered are in regular typeface, the same as the associated commands. Tags enclosed in braces, { } , are optional.

Each PLATO author language lesson has two structures, physical and logical. The physical structure is the same for all lessons and is simply a series of instructions grouped into entities called units. The system stores these instructions sequentially in central memory, in ECS, on mass-memory disks, or in any combination of these. The logical structure is usually more complex and may bear little relation to the physical structure. The concern of this manual is primarily with the logical structure of lessons rather than the physical structure. Thus, when the structure of a lesson is mentioned, without other qualification, the logical structure is the structure meant.

The logical structure of the lesson is the presentation of the execution of the parts of the lesson. The basic part of a lesson is the unit. A unit is usually entered from only one point in its structure, the beginning. Since all references to code other than that being executed refer to entire units, it is reasonable to speak of units as black boxes. The structure of the lesson then depends upon what types of boxes are used and their connection with each other. This is developed more thoroughly, although implicitly, in section 10.

A lesson is basically composed of one or more sequences of main units, usually with some of the main units having auxiliary units.

The difference between main and auxiliary units is given in detail in section 3. The relevant difference in terms of lesson structure is that auxiliary units cannot be formed into a sequence and are invisible to the student as separate entities.

BRANCHING

There are two types of branching within a lesson, author-initiated and student-initiated. Author-initiated branching occurs when one of several paths is taken, with the lesson, rather than the student, making the selection. In many situations, this is based on the performance of the student, such as automatically branching the student to a remedial sequence. The use of auxiliary units can be considered a form of author-initiated branching.

Student-initiated branching occurs when the student chooses an option to execute. There are two major types, function key branching and author-provided branching. Function key branching occurs when the author gives the student a choice which is accessible by a function key. This includes the main unit sequences specified with the NEXT, SHIFT NEXT, BACK, and SHIFT BACK keys and the help sequences specified with the HELP, SHIFT HELP, LAB, SHIFT LAB, DATA, and SHIFT DATA keys. An example of author-provided branching is when the author gives the student a list of options and lets the student select the next option to execute.

EXECUTION

The basic flow of execution of an author language lesson, unless modified by the author, is sequentially through the physical lesson. Thus, if a particular sequence of units that is to be executed sequentially is also sequential in the physical lesson, no specification of the next unit to be executed is required. However, if any modification to this simple linear flow is desired, as with the use of branching, the author must explicitly sequence the units within the physical lesson in order to give the order of execution of the units in the logical lesson. The addition of branching capabilities allows the sequential flow to be modified in several ways.

The author-initiated branching modifies the sequence of execution of units and can be used to select one of several possible sequences of base units. The base unit sequences usually contain the basic information that the lesson is designed to convey.

The student-initiated branching usually adds a number of units to the order of execution, with execution resuming at the unit from which the help sequence was accessed. However, this can be modified by the author through the use of the –base– instruction so that either the student does not return from the help sequence (because it defines itself as a base sequence, once entered) or the student returns from the help sequence to a different unit than the unit from which he entered the help sequence. There are three types of student-initiated function key branching: two of them have similar effects in that when the student executes the sequence specified by the author, he is returned to that part of the lesson from which the branching was initiated; the third type of branching does not return the student to that part of the lesson from which branching was initiated (refer to section 10).

Student-initiated branching is made possible by explicit definition of the branching possibilities by the author. The student-initiated branching can be initiated by the use of the student help keys and the TERM key. Unless the author has specified a help sequence or a -term-ed unit, these keys have no effect on the lesson.

Auxiliary units are units that are executed under the control of a main unit. Sometimes the main unit resumes execution at the point where execution was suspended, but this is not necessarily the case. Whether execution of the main unit is resumed depends on the contents of the auxiliary unit and the method used for accessing the auxiliary unit.

The basic component of a PLATO author language lesson is the unit. A unit is a semi-independent block of author language instruction, beginning with a –unit– instruction and ending immediately before the next –unit– instruction. With one exception, the –unit– instruction must be present. The tag of the –unit– instruction is the name of the unit, which must be no more than eight characters in length. The unit is semi-independent because the unit can be called into execution from another unit or from several other units at different points in the lesson. To be meaningful, however, the unit must have some connection with the logical structure of the lesson.

The most general form of the unit has two parts. While it is common for a unit to have both parts, it is not necessary. Inclusion or exclusion of either part depends on the intended use of the unit and the preference of the author. When both parts are present, they are separated by the –arrow– and –endarrow– instructions.

One part of the unit is used for presentation of material to the student that is not dependent on a response and for some general calculation. This part of the unit consists entirely of regular instructions.

The –arrow– instruction indicates that student response is requested. The part of the unit following the –arrow– instruction is the response judging part of the unit. Other processing can also be done in this part of the unit.

Units without a response processing section are fairly common and are used for special purposes, repetitive processing, and simple display presentation. Units consisting entirely of a response processing portion are rare, and their execution can be complex.

INITIAL ENTRY UNIT

The initial entry unit (IEU) is the one exception to the requirement that a unit have a –unit– instruction as its first instruction.

The IEU must physically be the first unit in the lesson. It is differentiated by the lack of the –unit– instruction as its first instruction. The use of an IEU is not mandatory but is encouraged if the student is to be allowed to restart an uncompleted lesson at some point other than the beginning of the lesson.

When the student starts at some point in the lesson other than the beginning, the physically preceding units are skipped totally except the IEU. Thus, if these units load a character set, define a micro table, or initialize the value of some variables, these actions are not performed and the lesson does not execute correctly, as seen by the student, unless they are in the IEU.

The IEU, if present, is executed each time the student enters the lesson, except when a lesson does a –jumpout– to itself, or when the lesson is a router. Executing the IEU upon each student entry makes the loading of character sets and other initializations simple and localized.

```
at          912
write       Now LOADING CHARACTER AND
                MICRO SETS

            Please be patient - loading takes
            about 17 seconds
charset     sindarin, feanorian
micro       sindarin, mike
erase
unit        intro
```

This IEU causes a character set and a micro table to be loaded and performs a full-screen erasure when the loading is complete. This is necessary because the IEU is executed as if it were accessed by a –join– instruction from the first unit entered by the student. Thus, if an erase is not performed, the display (if any) generated by the IEU is not erased before the display from the actual unit entered is placed on the screen.

MAIN AND AUXILIARY UNITS

Since the definition of a unit is the same for all units, the units vary in usage rather than definition. Furthermore, it is possible, although uncommon, for the usage of a specific unit to change during lesson execution. Hence, a reference to a type of unit indicates only the way in which the unit is to be used at that time.

There are two major types of units, main units and auxiliary units. Main units are further subdivided into base units and help units. A full discussion of the types of units and their use is given in section 10.

An auxiliary unit is executed under the control of another unit. The auxiliary unit, as the name implies, furnishes auxiliary instructions to the unit from which it was accessed. No unit initialization is performed.

A main unit is any unit which is not an auxiliary unit. Such units usually lie in a sequence of units, which an auxiliary unit cannot do. There are two types of sequences, base sequence and help sequences.

A help sequence is a sequence of one or more units that is specified by the author as accessible by the student through the use of the special student help keys. The units composing such a sequence are called help units. A help sequence usually contains background, definitional, or review material. The help sequence is not executed unless accessed by the student. When execution of a help sequence is complete, the unit from which the help sequence is accessed is reexecuted, unless the author specifies otherwise within the help sequence.

The unit from which the student can access a help sequence serves as a base for that sequence. Hence, such units are called base units. A special marker, indicating that the current unit is the base unit, is updated each time a main unit, other than a help unit, is initialized. Therefore, a sequence of main units that are not help units is termed a base sequence. There can be more than one base sequence in a lesson, which is why the term base sequence is preferable to primary sequence.

The instructions in a unit are not necessarily executed in order of occurrence. Particularly during response processing, some instructions might be executed several times, while others might not be executed at all.

Part of the reason is the possibility of author- and student-initiated branching. However, it is usually because of the manner in which student responses are processed.

The author language executer operates in two major states, regular and judging. The author language instructions are divided into two classes, based on which of the two system-states allow their execution. Thus, display instructions, which can be done independently of the student response, are regular instructions, while those that determine whether the student's response will be accepted as correct (plus some instructions for manipulation of the student response) are judging instructions. Only one instruction, the –join– instruction, is executed in both the regular and judging states.

The processor ignores judging instructions that do not follow an –arrow– instruction.

When a unit is entered, any initialization is done first. If the unit is an auxiliary unit, no initialization is done. If it is a help unit, there is some initialization, usually including a full-screen erase, but not complete initialization. Full initialization is performed if the unit entered is a base unit.

After initialization, all regular instructions up to the first –arrow– instruction or the next –unit– instruction, whichever occurs first, are executed.

If the system encounters a –unit– instruction, it waits for the student to press the NEXT key or any active key, indicating readiness to continue. When the student presses an active key, the processor enters the new unit with the procedure as before.

If an –arrow– instruction is encountered, the system places the arrow character on the student's screen (unless this has been disabled by the author with an –inhibit– instruction) and then executes any regular instructions immediately following the –arrow– instruction. When the first judging instruction is reached, the system then switches into the judging state and waits for the student to respond.

When the student presses the NEXT key, indicating that he is finished typing his response, the system begins checking the response against the specified answers (answers may be specified as correct or incorrect). If an adequate match is not found, the response is judged incorrect by default. If a match is found, the response is judged ok or no (correct or incorrect), whichever is indicated by the instruction matched. Regular instructions following the matched answer are then executed, if any, until an –endarrow–, another –arrow–, a –unit–, or another judging instruction is encountered.

Unless the author specifically constructs the lesson otherwise, a student is forced to press NEXT or ERASE, type a different response, and enter it by pressing NEXT again, whenever the response is judged incorrect. (Editing keys are also available to the student under author control that prevent the necessity of retyping the entire response.)

When the –arrow– has been satisfied (that is, when the student has given a response that has been judged ok and all following regular instructions have been executed), the system then checks for an –endarrow–, –arrow– or –unit– instruction. These are then executed as before. If an –arrow– or –unit– instruction is found, the instruction is executed as previously.

This execution sequence can be modified by the –long– and –specs– instructions. For a complete discussion of lesson execution during response processing, refer to section 11.

This section contains a short description of each of the PLATO author language instructions. A more complete explanation is given in sections 6 through 14. Table 5-1 gives an alphabetic list of the instructions with their page numbers.

INSTRUCTION FORMAT

Each author language instruction has two parts, command and tag. The command is the same as the name of the instruction. The tag gives the necessary specifications and modifiers for execution of the command. If none are necessary, the tag can be left blank. The command field begins in column 1 of the display. The tag field begins in column 9. After entering the command, the TAB key positions the cursor to the ninth column. (The cursor is not visible.)

The tag consists of one or more arguments. Each argument is one specification for the command portion of the instruction. Arguments are usually separated by commas. Arguments enclosed in braces ({,}) in the instruction descriptions are optional and need not be included in the instruction. Multiline entries give alternatives; only one can be used in a single instruction.

For example:

$$\left\{\begin{matrix}\text{blank}\\ \text{n}\end{matrix}\right\}$$

TABLE 5-1. INSTRUCTION INDEX

Instruction	Description	Explanation	Instruction	Description	Explanation	Instruction	Description	Instruction
abort	5-23	8-15	char	5-35	9-7	delname	5-21	8-19
addlst	5-62	10-16	charset	5-36	9-8	delrecs	5-21	8-19
addname	5-20	8-19	chartst	5-36	9-9	delta	5-43	9-14
addrecs	5-21	8-19	circle	5-31	9-5	disable	5-38	9-10
add1	5-5	7-4	circleb	5-31	9-5	do	5-47	10-2
allow	5-61	10-16	clock	5-12	8-8	dot	5-29	9-4
altfont	5-37	9-9	close	5-65	11-4	doto	5-16	8-11
ans	5-73	11-10	codeout	5-37	9-10	draw	5-30	9-4
ansu	5-70	11-8	collect	5-57	10-11	edit	5-75	11-12
ansv	5-70	11-7	color	5-35	9-6	else	5-16	8-10
answer	5-66	11-5	comload	5-18	8-14	elseif	5-16	8-10
answerc	5-66	11-6	common	5-18	8-13	enable	5-38	9-10
area	5-76	13-2	commonx	5-18	8-14	end	5-49	10-6
arheada	5-63	11-2	compare	5-73	11-10	endarrow	5-63	11-1
arrow	5-63	11-1	compute	5-12	8-7	endif	5-16	8-10
arrowa	5-64	11-2	comret	5-19	8-15	endings	5-68	11-7
at	5-24	9-2	concept	5-67	11-6	endloop	5-17	8-12
atnm	5-24	9-2	copy	5-75	11-12	entry	5-48	10-4
audio	5-38	9-11	cpulim	5-60	10-14	erase	5-28	9-3
axes	5-39	9-12	cstart	5-54	10-8	eraseu	5-29	9-4
back	5-49	10-5	cstop	5-54	10-8	exact	5-69	11-7
backgnd	5-60	10-14	cstop*	5-55	10-9	exactc	5-69	11-7
backop	5-50	10-5	data	5-51	10-6	exactv	5-70	11-7
back1	5-50	10-5	datain	5-19	8-17	exit	5-47	10-4
back1op	5-50	10-5	dataoff	5-76	13-1	ext	5-39	9-11
base	5-53	10-7	dataon	5-76	13-1	extout	5-39	9-11
block	5-23	8-20	dataop	5-52	10-7	find	5-7	8-4
bounds	5-39	9-12	dataout	5-20	8-17	findall	5-8	8-5
box	5-32	9-5	dataset	5-19	8-16	findl	5-62	10-17
branch	5-15	10-4	data1	5-52	10-6	finds	5-9	8-5
bump	5-65	11-4	data1op	5-52	10-7	findsa	5-9	8-5
calc	5-5	7-3	date	5-12	8-8	finish	5-48	10-5
calcc	5-5	12-1	day	5-12	8-8	force	5-58	9-10
calcs	5-6	8-13	define	5-3	6-2	foregnd	5-60	10-14
catchup	5-35	9-6	delay	5-35	9-6	from	5-13	8-8
change	5-58	10-12	deletes	5-11	8-6	funct	5-43	9-14

TABLE 5-1. INSTRUCTION INDEX (Contd)

Instruction	Description	Explanation	Instruction	Description	Explanation	Instruction	Description	Explanation
gat	5-25	9-11	match	5-73	11-10	rvector	5-34	9-5
gatnm	5-26	9-11	micro	5-37	9-10	scalex	5-40	9-12
gbox	5-33	9-11	miscon	5-67	11-6	scaley	5-40	9-12
gcircle	5-32	9-11	mode	5-35	9-6	score	5-59	10-13
gdot	5-30	9-11	modperm	5-15	8-9	search	5-8	8-5
gdraw	5-31	9-11	move	5-12	8-7	seed	5-14	8-9
getloc	5-75	11-5	name	5-13	8-8	set	5-6	8-2
getmark	5-74	11-5	names	5-21	8-19	setdat	5-77	13-2
getname	5-20	8-19	next	5-45	10-5	setname	5-20	8-18
getword	5-74	11-5	nextnow	5-48	10-4	setperm	5-14	8-9
gorigin	5-25	9-11	nextop	5-45	10-5	show	5-26	9-2
goto	5-47	10-3	next1	5-45	10-5	showa	5-28	9-3
graph	5-42	9-13	next1op	5-45	10-5	showe	5-27	9-2
group	5-13	8-8	no	5-69	11-7	showo	5-27	9-2
gvector	5-34	9-11	notes	5-79	13-5	showt	5-27	9-2
hbar	5-42	9-13	noword	5-69	11-2	showz	5-27	9-2
help	5-50	10-6	ntouch	5-71	11-9	site	5-80	14-1
helpop	5-51	10-7	ntouchw	5-72	11-9	size	5-29	9-4
help1	5-51	10-6	ok	5-68	11-7	slide	5-37	9-10
help1op	5-51	10-7	okword	5-69	11-2	sort	5-10	8-6
hidden	5-28	9-3	open	5-64	11-4	sorta	5-10	8-6
htoa	5-7	8-4	or	5-73	11-10	specs	5-73	11-10
iarrow	5-63	11-2	otoa	5-7	8-4	station	5-80	14-1
iarrowa	5-64	11-2	outloop	5-17	8-12	status	5-55	10-9
if	5-15	8-10	output	5-77	13-2	step	5-58	10-12
iferror	5-48	10-4	outputl	5-77	13-2	stoload	5-23	8-16
ignore	5-69	11-7	pack	5-6	8-3	stop	5-50	10-5
imain	5-44	10-5	packc	5-6	8-4	storage	5-23	8-15
in	5-59	8-8	pause	5-57	10-11	store	5-71	8-7
inhibit	5-56	10-10	play	5-38	9-11	storea	5-64	11-3
initial	5-59	10-13	plot	5-36	9-7	storen	5-71	11-8
inserts	5-11	8-6	polar	5-44	9-14	storeu	5-71	11-8
itoa	5-7	8-4	press	5-55	10-9	sub1	5-5	7-4
jkey	5-64	11-3	put	5-65	11-4	tabset	5-37	9-10
join	5-46	10-2	putd	5-65	11-4	term	5-54	10-8
judge	5-74	11-11	putv	5-66	11-4	termop	5-54	10-8
jump	5-46	10-2	randp	5-14	8-9	text	5-28	9-3
jumpout	5-56	10-10	randu	5-14	8-9	time	5-75	11-12
keylist	5-56	10-11	rat	5-25	9-5	timel	5-49	10-5
keytype	5-57	10-11	ratnm	5-26	9-5	timer	5-49	10-5
lab	5-52	10-6	rbox	5-33	9-5	touch	5-72	11-9
labelx	5-41	9-12	rcircle	5-32	9-5	touchw	5-72	11-9
labely	5-41	9-12	rdot	5-30	9-5	transfr	5-24	8-20
labop	5-53	10-7	rdraw	5-30	9-5	unit	5-44	10-1
lab1	5-53	10-6	readd	5-78	13-3	use	5-58	10-12
lab1op	5-53	10-7	readr	5-79	13-4	vbar	5-43	9-13
leslist	5-61	10-16	readset	5-78	13-3	vector	5-33	9-5
lessin	5-13	8-8	record	5-38	9-11	vocab	5-68	11-7
lesson	5-59	10-13	release	5-22	8-15	vocabs	5-68	11-6
lineset	5-36	9-9	reloop	5-17	8-12	window	5-34	9-5
list	5-67	11-6	remove	5-15	8-9	write	5-26	9-1
lname	5-62	10-17	removl	5-62	10-17	writec	5-26	9-1
loada	5-65	11-4	rename	5-21	8-19	wrong	5-66	11-5
long	5-64	11-3	reserve	5-22	8-15	wrongc	5-67	11-6
loop	5-17	8-12	restart	5-55	10-9	wrongu	5-70	11-8
lscalex	5-40	9-11	return	5-55	10-9	wrongv	5-70	11-7
lscaley	5-40	9-11	rorigin	5-25	9-5	zero	5-5	7-4
markup	5-74	11-10	rotate	5-29	9-4	*list	5-81	15-1
markx	5-42	9-12	route	5-60	10-14	*,c,$$	5-54	10-8
marky	5-42	9-12	routvar	5-61	10-15			

CALCULATION INSTRUCTIONS

The following instructions are used to manipulate information.

define	compute	endloop
calc	clock	common
addl	date	commonx
subl	day	comload
zero	name	comret
calcc	group	dataset
calcs	from	datain
set	lessin	dataout
pack	randu	setname
packc	setperm	getname
itoa	randp	addname
otoa	seed	delname
htoa	remove	rename
find	modperm	names
findall	branch	addrecs
search	if	delrecs
finds	elseif	reserve
findsa	else	release
sort	endif	abort
sorta	doto	storage
inserts	loop	stoload
deletes	outloop	block
move	reloop	transfr

DEFINE

Instruction format:

define {**name** {,**sets**}}
definitions

Tag definitions:

name Name of define set (optional)

sets Name of other define sets to be included (names separated by commas)

definitions Definitions of any or all of the forms:

variable

name=var

name Name assigned to a variable

var Variable of type n, v, nc, or vc

constant

name=num

name Name assigned to a constant

num Number or mathematical expression

function

fn(x)=expr(x)

fn Name assigned to an author-defined function

x Dummy variable

expr Definition of a function

array

array,**name(num)=var**

$$† one-dimensional array

array,**name(1num;nnum)=var**

$$one-dimensional array with offset index

array,**name(row,col)=var**

$$two-dimensional array

array,**name(1row,1col;nrow,ncol)=var**

$$two-dimensional array with offset index

array,**name=var**

$$zero-dimensional array

name(x1,x2,...,xn)=var(expr)

$$author-defined array

name Array name

num Number of elements

var Starting variable of array

1num First element

nnum Last element

row Number of rows

col Number of columns

1row First row

1col First column

nrow Last row

ncol Last column

x1,x2,...,xn Index variables specifying number of dimensions in array

expr Expression using index variables, specifying number of **var**

† Two consecutive dollar signs indicate a comment.

vertically segmented array

arraysegv,**name(num)=var,startbit,size** {,s / ,signed}

$$one-dimensional vertical array

arraysegv,**name(1num;nnum)=var,startbit,size** {,s / ,signed}

$$one-dimensional vertical array with offset index

arraysegv,**name(row,col)=var,startbit,size** {,s / ,signed}

$$two-dimensional vertical array

arraysegv,**name(1row,1col;nrow,ncol)=var,startbit,size** {,s / ,signed}

$$two-dimensional vertical array with offset index

name	Array name
num	Number of elements
var	Starting variable of array
startbit	Starting bit for segment
size	Number of bits in each element
s or signed	Segment elements have a sign bit (optional)
1num	First element
nnum	Last element
row	Number of rows
col	Number of columns
1row	First row
1col	First column
nrow	Last row
ncol	Last column

segment

segment,**segname=start,size** {,s / ,signed}

segname	Segment name
start	Starting variable of segment
size	Number of binary bits in each segment element
s or signed	Segment elements have a sign bit (optional)

vertical segment

segment,vertical,**name=var,startbit,size** {,s / ,signed}

name	Segment name
var	Unindexed variable such as n1 or nc (constant)
startbit	Starting bit for segment
size	Number of binary bits in each segment element
s or signed	Segment elements have a sign bit (optional)

units of measure

units,**prim1,prim2,...,equiv1,equiv2,...**

$$ must be in student define set

prim1, prim2,...	Primary units anticipated in student response for use with –ansu–, –storeu–, and –wrongu– instructions
equiv1, equiv2,...	Acceptable equivalent units

Effect:

Assigns meaningful names to variables.

Comments:

Defined names can be from 1 to 7 characters long. Names cannot begin with a number and cannot contain mathematical operators, FONT characters, or backspaces.

Previously defined quantities cannot appear on the left side of the equal sign but can appear on the right side. More than one definition can be placed on a single line if separated by commas. Names must be defined before use. Placing definitions in initial entry units is suggested. The define set named student defines variable names that can also be referenced by the student. An array may not contain more than 255 elements. The maximum value an array index may have (when offset) is $2^{13}-1$. The indexes, offsets, and base locations in array definitions must be literals or previously defined constants.

CALC

Instruction format:

calc	**var ⇐ expr**
calc	**var1 ⇐ var2 ⇐ ... ⇐ varn ⇐ expr**

Tag definitions:

var,var1, var2,..., varn	Variable names (primitive or defined)
expr	Mathematical expression

Effect:

Value of expression is placed in the variable(s) on the left of the assignment symbol(s) (⇐)

Comments:

The –calc– instruction can be a continued instruction. The variable can be an element of an array or segment. If **var** is the unindexed name of an array, the indicated operation is performed on each element of the array.

ADD1

Instruction format:

add1	**var**

Tag definition:

var	Variable name

Effect:

Adds one to the value of the named variable.

SUB1

Instruction format:

sub1	**var**

Tag definition:

var	Variable name

Effect:

Subtracts one from the value of the named variable.

ZERO

Instruction format:

zero	**start** {,**num**}

Tag definition:

start	Starting variable of block to be zeroed
num	Number of variables to be zeroed

Effect:

Sets indicated variables to numeric zero.

Comments:

One-argument tag sets only the named variable to zero.

CALCC

Instruction format:

calcc	**expr,assign1,assign2,...**

Tag definition:

expr	Expression whose value determines action to be taken
assign1, assign2,...	–calc– assignments, separated by commas

Effect:

Performs one of several –calc– type actions, depending on the value of the expression.

Comments:

First assignment done if expression is negative; second if expression is zero, third if expression is one, fourth if expression is two, and so on. Value of expression larger than that corresponding to a position uses last position. An x cannot be used to indicate no action; a null entry (successive commas) is used instead.

CALCS

Instruction format:

calcs **expr,var⇐value1,value2,...**

Tag definition:

expr	Mathematical expression
var	Variable to which a value is to be assigned
value1, value2,...	Expressions one of whose value is to be assigned to **var**

Effect:

Performs one of several possible assignments, depending on the value of the expression.

Comments:

First value is assigned if expression is negative, second if expression is zero, and so on. Value of expression larger than that corresponding to a position uses last position. An x cannot be used to indicate no action; a null entry (successive commas) is used instead.

SET

Instruction format:

set **var⇐expr1,expr2,...**

Tag definition:

var	Starting variable or array element of block to be –set–
expr1, expr2,...	Variables or mathematical expressions

Effect:

Allows several values to be entered into consecutive whole-word variables. The variables are set during execution of the –set– instruction.

Comments:

The –set– instruction cannot be used to assign values to consecutive elements of a segment.

PACK

Instruction format:

pack **nam, {len} ,str**

Tag definition:

nam	Variable in which characters are stored
len	Optional variable which contains character count
str	Character string to be stored

Effect:

Places specified character string in the named variable, left-justified, with zero-fill.

Comments:

String storage can cross variable boundaries. Using two successive commas specifies an omitted **len.**

PACKC

Instruction format:

packc **expr,nam, {len} ,str–,str0,str1,...,strn**

Tag definition:

expr	Mathematical expression
nam	Variable in which character strings are stored
len	Optional variable in which character count for specified string is stored
str–,str0, str1,..., strn	Character strings to be stored in **nam**

Effect:

Based on the value of the mathematical expression (**expr**), places specified character string in the named variable, left-justified, with zero-fill. –packc– is the conditional form of the –pack– instruction.

Comments:

The following may be used as separators: , ; or ǂ. The separator after **len** is the only allowable string separator. End-of-line also acts as a separator. If the first tag line ends with **len,** the only allowable separator is end-of-line.

ITOA

Instruction format:

itoa **val,loc {,num}**

Tag definition:

val	Integer variable to be converted
loc	Location of converted string
num	Optional variable for storing character count

Effect:

Value of the first variable is converted to an alphanumeric string. String is left-justified with zero-fill starting in **loc.**

Comments:

Only integer variables should be used. All arguments must be variable names.

OTOA

Instruction format:

otoa **val,loc {,num}**

Tag definition:

val	Octal variable to be converted
loc	Location of converted string
num	Optional variable for number of digits to be converted

Effect:

Value of the first variable is converted to an alphanumeric string. String is left-justified in a 1- or 2-word buffer named by second argument.

Comments:

If **num** is not specified, the default value is 20.

HTOA

Instruction format:

htoa **val,loc {,num}**

Tag definition:

val	Hexadecimal variable to be converted
loc	Location of converted string
num	Optional variable for number of digits to be converted

Effect:

Value of the first variable is converted to an alphanumeric string. String is left-justified in a 1- or 2-word buffer named by second argument.

Comments:

If **num** is not specified, the default value is 15.

FIND

Instruction format:

find **obj,start,length,return {,increment} {,mask}**

Tag definition:

obj	Variable containing object to be found
start	Starting variable of search
length	Number of variables to be searched
return	Variable in which number of variable containing object is stored
increment	Optional variable which specifies only every nth word for comparison with **obj**
mask	Optional mask of variable containing object

Effect:

Searches specified number of variables for bit configuration in object (possibly modified by **mask**). The number of the variable in which the object first occurs (relative to the starting variable) is placed in the return variable. If the object is not found, the value of the return variable is set to –1.

Comments:

The search can be made backwards through the list by specifying the length as negative. The value of **return,** however, is the number relative to the starting variable (that is, if there is only one occurrence, a negative search gives the same result as a positive search). If **mask** is used, **increment** must be specified.

FINDALL

Instruction format:

findall **obj,start,length,return,follow {,increment} {,mask}**

Tag definition:

obj	Object to be found
start	Starting variable of list to search
length	Length of list
return	Starting return variable (cannot be a segment)
follow	Number of following variables for storage of found locations (0 for count only)
increment	Optional variable which specifies only every nth word for comparison with object (assumed to be 1 if not given)
mask	Optional mask of variable containing object (full-word search if not given)

Effect:

Searches a list of variables to find a specified object, returns a count of the number of matches, and lists the locations of the matches. Increment between variables searched can be specified to be other than 1.

Comments:

If object matches the first variable, the location value returned is 0; if there is no match, return is –1. If **mask** is used, **increment** must be specified (1 in normal case). If **increment** specified exceeds the length, an execution error occurs. The search can be made backwards from the last variable in the list by specifying **increment** as negative.

SEARCH

Instruction format:

search **str,strlen,start,len,char,ret {,count}**

Tag definition:

str	Variable containing character string to be found
strlen	Number of characters in **str**
start	Starting variable of string to be searched
len	Number of variables to be searched
char	Starting character position of string to –search– within **start** (1 indicates first character in string)
ret	Variable storing result
count	Variable or mathematical expression

Effect:

Searches the specified variables for the object string specified in **str. ret** contains the relative character number of the first occurrence if an occurrence is found; otherwise, **ret** is –1. If the optional **count** is present and is 0, the entire string is searched for all occurrences of the object and returns a count of the number of occurrences in **ret.** If **count** > 0, each of that number of variables following **ret** contains the character position of one of the **ret** occurrences within the string searched. The **ret**+ 1th variable is set to –1 to indicate the end of the list of occurrences, and all following variables are unchanged. Does a forward search if **len** is positive, and does a backward search if **len** is negative.

Comments:

The object string must be left-justified in the variable. Search crosses word boundaries, so **ret** is the character position in which the occurrence begins. The string searched for (contents of **str**) can be anywhere within a word. The string searched for can be no longer than 10 characters. **ret** cannot be a –segment–ed variable.

FINDS

Instruction format:

finds **object,list;length,inc,1stbit,numbits, return {,mask}**

Tag definition:

object Variable containing object to be found

list Starting location of list can be:

Student variables	Either n(x) or defined name
CM variables	nc(x)
ECS common	common,2000 (=c,2000)
ECS storage	storage,123 (=s,123)

length Number of entries in list; one entry can be several words

inc Number of words per entry; must be $\leq$ 500

1stbit First bit of search field

numbits Number of bits in search field

return Variable containing the list entry number if object is found; otherwise, negative of list entry number where object should be

mask Optional mask of variable containing object

Effect:

Finds an object in a sorted numeric list.

Comments:

Does not check if the list is sorted. Each entry in the list can occupy one or more words, but must not be partial words. The numeric field cannot cross word boundaries. List **length** of 0 sets **return** to -1, and **inc** must be at least 1.

FINDSA

Instruction format:

findsa **object,list;length,inc,1stchar,numchars, return {,mask}**

Tag definition:

object Variable containing object to be found

list Starting location of list can be:

Student variables	Either n(x) or defined name
CM variables	nc(x)
ECS common	common,2000 (=c,2000)
ECS storage	storage,123 (=s,123)

length Number of entries in list; one entry can be several words

inc Number of words per entry; must be $\leq$ 500

1stchar First character of search field

numchars Number of characters in search field

return Variable containing the list entry number if object is found; otherwise, negative of list entry number where object should be

mask Optional 1-word mask of variable containing object

Effect:

Finds an object in a sorted alphabetic list.

Comments:

Does not check if the list is sorted. Each entry in the list can occupy one or more words, but must not be partial words. The character field can cross word boundaries. List **length** of 0 sets **return** to -1, and **inc** must be at least 1.

SORT

Instruction format:

sort **list1;length,inc,1stbit,numbits {,mask}**
list2;inc

Tag definition:

list1 Starting location of the list to be sorted in numerical order:

Student variables	Either n(x) or v(x)
CM variables	Either nc(x) or vc(x)
ECS common	common,2000 (=c,2000)
ECS storage	storage,123 (=s,123)

list2 Starting location of the optional associated list to be sorted in numerical order

length Number of entries in list; one entry may be several words

inc Number of words per entry

1stbit First bit of field for sorting

numbits Number of bits in sort field

mask Optional mask for sort field

Effect:

Places a list of variables (n or v variables) into numerical order, smaller to larger. The second tag line is optional; it causes **list2** to be sorted at the same time as **list1**, thus sorting paired lists.

Comments:

A semicolon is required as a separator after the list tag. Each entry in the list may occupy one or several whole variables. Entries cannot be partial words. The numeric field for sorting may not cross word boundaries. For example, it is allowable to sort on bits 43 through 53, but it is not allowable to sort on bits 55 through 67. The command field for the paired list (the optional second line for the instruction) must be blank. Only two lists can be paired.

SORTA

Instruction format:

sorta **lista;length,inc,1stchar,numchars {,mask}**
listb;inc

Tag definition:

lista Starting location of the list to be sorted in alphabetical order

listb Starting location of the optional associated list to be sorted in alphabetical order

length Number of entries in list; one entry may be several words

inc Number of words per entry

1stchar First character of sort field

numchars Number of characters in sort field

mask Optional 1-word mask for sort field

Effect:

Places a list into alphabetical order according to the internal codes for the characters. The second tag line is optional; it causes **listb** to be sorted at the same time as **lista**, thus sorting paired lists.

Comments:

A semicolon is required as a separator after the list tag. Each entry in the list may occupy one or several whole variables. Entries cannot be partial words. The character field may cross one word boundary, such as characters 8 through 14. The command field for the paired list (the optional second line for the instruction) must be blank. Only two lists can be paired.

INSERTS

Instruction format:

inserts **object1,list1;length,inc1,loc** {**,num**}
{**object2,list2;inc2**}

Tag definition:

object1 Variable containing object to be inserted

object2 Variable containing object to be inserted in associated list (optional)

list1 Starting location of list can be:

Student variables	Either n(x) or defined name
CM variables	nc(x)
ECS common	common,2000 (=c,2000)
ECS storage	storage,123 (=s,123)

list2 Starting location of optional associated list

length Number of entries in list before insertion; one entry can be several words

inc1, inc2 Number of words per entry; must be $\leq$ 500

loc Position objects are inserted into

num Optional number of inserted entries

Effect:

Inserts any number of new entries into a list. The second tag line is optional; it inserts **object2** in the optional associated **list2** in the same position as **object1** is inserted in **list1**.

Comments:

Each entry in the list can occupy one or more words, but must not be partial words. List entry size determines object size. **inc** must be at least 1, and **loc** must be ≥ 1 and $\leq$ **length** + 1.

DELETES

Instruction format:

deletes **list1;length,inc1,loc** {**,num**}
{**list2;inc2**}

Tag definition:

list1 Starting location of list can be:

Student variables	Either n(x) or defined name
CM variables	nc(x)
ECS common	common,2000 (=c,2000)
ECS storage	storage,123 (=s,123)

list2 Starting location of optional associated list

length Number of entries in list before deletion; one entry can be several words

inc1, inc2 Number of words per entry; must be $\leq$ 500

loc Position of entry to be deleted

num Optional number of deleted entries

Effect:

Deletes any number of entries from a list. The second tag line is optional; it deletes an entry in the optional associated **list2** from the same position as the entry in **list1** is deleted.

Comments:

Fills the last entry with zeros after moving the list together. Each entry in the list can occupy one or more words, but must not be partial words. List entry size determines object size. **inc** must be at least 1, and **loc** must be ≥ 1 and $\leq$ **length** + 1.

MOVE

Instruction format:

move **ostart,ochar,fstart,fchar {,len}**

Tag definition:

ostart Starting variable of object to be moved

ochar Character location within variable **ostart** of object to be moved

fstart Starting variable of new location

fchar Character location within variable **fstart** of new location

len Optional character count; if omitted, one character is moved

Effect:

Copies a character string from one location to another. A maximum of 1500 characters can be moved within one –move– instruction.

COMPUTE

Instruction format:

compute **val,start,len,var**

Tag definition:

val Variable to contain value of expression

start Starting variable of expression (left-justified)

len Number of characters in expression; must be ≤ 100

var Variable to contain a pointer to compile code

Effect:

Evaluates value of expression stored as a character string and saves compile code for later use of the –compute– for the same string.

Comments:

The first time a character string is compiled, **var** must have the value zero. On following evaluations of the same string, **var** must have the value assigned during the first compilation. Must use a student define set with this instruction, if defined variables are in the string.

CLOCK

Instruction format:

clock **name**

Tag definition:

name Variable name

Effect:

Places time (to nearest second) in variable in alphanumeric form with format such as

13.20.58

(that is, 24-hour clock).

DATE

Instruction format:

date **name**

Tag definition:

name Variable name

Effect:

Places current date in variable in alphanumeric format with the format month/day/year.

DAY

Instruction format:

day **vname**

Tag definition:

vname Floating-point variable name

Effect:

Places number of days (whole and fractional) since the starting date and time (set by installation).

NAME

Instruction format:

name **name1**

Tag definition:

name1 Name of first of two consecutive variables to be used

Effect:

First 10 characters of student sign-on name placed in first variable, remaining eight in second variable, left-justified, zero-filled.

GROUP

Instruction format:

group **name**

Tag definition:

name Variable name

Effect:

Places the student's group name in variable name, left-justified, zero-filled.

FROM

Instruction format:

from **var;lname** {**,uname**} ;
lname1 {**,uname1**} **;.;.;**

$$lesson name/unit name pairs

from **var;<expr>** {**,uname**} ;
<expr1> {**,uname**} **;.;.;**

$$leslist

from **lvar** {**,uvar**}

Tag definition:

var Variable

lname;
lname1;... Lesson names

uname;
uname1;... Unit names or variables containing unit name, left-justified, zero-filled

expr Variable or mathematical expression that references a leslist number enclosed in <>

lvar Variable in which lesson name is placed

uvar Variable in which unit name is placed

Effect:

Permits author to determine the lesson (and unit) from which his lesson was entered via a –jumpout– instruction.

Comments:

If the current lesson was entered from the first lesson name/ (unit name), the variable is set to 0; if from the next pair, it is set to 1, etc. If the lesson is not found in the list (lesson name/unit name pair list), the variable is set to –1. Each line of the –from– instruction must end with a semicolon. Last form of –from– places names of lesson and unit from which student entered current lesson into **lvar** and **uvar.**

LESSIN

Instruction format:

lessin **lesson**

Tag definition:

lesson Actual lesson name or leslist reference in <>

Effect:

Checks to see if lesson named is credited to the logical site and sets system-reserved word zreturn accordingly (zreturn is –1 if lesson is in ECS or in use at site; zreturn is 0 otherwise).

Comments:

Leslist reference may be a constant, a variable, or an expression.

RANDU

Instruction format:

randu **name** {**,lim**}

Tag definition:

name Variable name in which value will be stored

lim Limit value

Effect:

One argument: Returns a fraction between zero and one (0 < **name** < 1)

Two argument: Returns an integer between one and value of limit, inclusive

Comments:

The variable storing the number should be a floating-point variable and must be such in the one-argument case. Sampling is done with replacement.

SETPERM

Instruction format:

setperm **lim**{**,loc**}

Tag definition:

lim Upper bounds of integer set used (1 ≤ **lim** ≤ 120)

loc Location for storing permutation

Effect:

Sets up a permutation of integers between one and **lim** inclusive. One-argument form stores the permutation in the standard system location and creates a second copy. Two-argument form stores the permutation in the user variables specified and does not create a second copy. In the two-argument form, **lim** can be greater than 120.

Comments:

Second copy is for use by the –remove– and –modperm– instructions. When the two-argument form is used, the number of variables required to store the permutation is the value of the author language expression 2 + int ((**lim** - 1)/60).

RANDP

Instruction format:

randp **store** {**,loc**}

Tag definition:

store Variable used to store result

loc Starting location of permutation is in user bank variables

Effect:

Selects an integer from the permutation without replacement and stores the result in **store.** One-argument form uses system location; two-argument form uses a permutation located in the user variables specified. When the permutation is exhausted, a value of zero is returned.

SEED

Instruction format:

seed **loc**

Tag definition:

loc Location of seed value for use with the –randu– and –randp– instructions

Effect:

Allows specification of a starting value for the variable used in the algorithm for generating random numbers. As long as the algorithm is unchanged, any string of random numbers can be repeated by using the same seed.

Comments:

A –seed– instruction with a blank tag indicates that the system seed is to be used. When a student enters a lesson, the system seed is assumed until a –seed– instruction is executed. The seed indicated by the –seed– instruction is then used until a –seed– instruction with a blank tag is executed or the student leaves the lesson.

REMOVE

Instruction format:

remove **value** {,**loc**}

Tag definition:

value Number to be removed from the permutation

loc Starting location of permutation is in user bank variables

Effect:

Removes specified value from permutation. One-argument form affects second system copy, and two-argument form affects the user variable copy specified.

Comments:

The instruction is ignored if **value** specified is less than or equal to zero, greater than the largest element of the permutation, or previously removed by a –remove– instruction.

MODPERM

Instruction format:

modperm

Effect:

Replaces first system copy of permutation (accessed by a one-argument –randp–) with second system copy (affected by –remove–).

BRANCH

Instruction format:

branch **label**

branch **expr,label1,label2,...**

Tag definition:

expr Mathematical expression

label,label1, Line label, x, or q
label2

Effect:

Branches to different lines of code within the same unit. A line label appears in the command field, starts with a number, and is seven or less characters in length. x causes the –branch– to have no effect; q causes branching to the next non–calc– instruction. The second form of the –branch– instruction is the normal conditional form.

Comments:

If the –branch– is preceded by a –calc– instruction, the –branch– continues the preceding –calc– and can, if desired, be put in the tag field. Otherwise, the –branch– must be in the command field and initiates calculation.

IF

Instruction format:

if **expr**
. **instr**

Tag definition:

expr Logical or arithmetic expression

instr Indented author language instructions (must be regular)

Effect:

Begins an –if– structure. If the value of **expr** is true, the author language instructions immediately following –if– are executed. If the value of **expr** is false, execution resumes at the next –elseif–, –else–, or –endif– instruction at the current level of indenting.

Comments:

Every –if– structure must end with an –endif– instruction. Instructions between –if– and –endif– must be indented (except for –elseif– and –else–).

ELSEIF

Instruction format:

elseif	**expr**
.	**instr**

Tag definition:

expr	Logical or arithmetic expression
instr	Indented author language instructions (must be regular)

Effect:

If the value of **expr** is true, the author language instructions immediately following –elseif– are executed. If the value of **expr** is false, execution resumes at the next –elseif–, –else–, or –endif– instruction at the current level of indenting.

Comments:

Evaluated when the preceding –if– or –elseif– is false. Can have more than one –elseif– instruction in an –if– structure.

ELSE

Instruction format:

else	
.	**instr**

Tag definition:

instr	Indented author language instructions (must be regular)

Effect:

When all preceding –if– and –elseif– expressions are false, the author language instructions immediately following –else– are executed.

Comments:

Must have a blank tag.

ENDIF

Instruction format:

endif

Effect:

Ends an –if– structure.

Comments:

Execution jumps to –endif– after the execution of author language instructions following a true **expr.**

DOTO

Instruction format:

doto **label,var⇐beg,end {,inc}**

Tag definition:

label	Statement label
var	Index variable to be incremented
beg	Initial value of index variable (constant, variable, or mathematical expression)
end	Final value of index variable (constant, variable, or mathematical expression)
inc	Size of increment for index variable (default is 1; negative increments are permitted)

Effect:

Allows an interactive loop within the same unit, with the –doto– loop extending from the –doto– command to the statement label named in the tag. The statement label is in the command field, starts with a number, is seven or less characters in length, and has a blank tag. The –doto– loop is executed the specified number of times indicated in the iterative loop (**var⇐beg**). When the value of the index variable exceeds the final value (**end**) of the index variable, the –doto– loop is complete, and the command following the line label is executed.

–doto– loops can be nested within –doto– loops. A negative **inc** value gives a decreasing loop.

Comments:

–doto– in the command field initiates a –calc– function; if the –doto– begins the tag field, it continues a –calc–. Non–calc– commands are allowed within a –doto– loop. The statement label ending the –doto– may not contain a –calc– expression in the tag of the statement label. The statement label of the –doto– loop must be in the same unit as the –doto– statement. –doto– can only be used in the iterative form. An –entry– in the middle of a –doto– loop is illegal. When nesting –doto– loops, the inner –doto– loop cannot extend beyond the statement label of the outer –doto– loop.

LOOP

Instruction format:

```
loop      [expr]
.         instr
```

Tag definition:

expr Logical or arithmetic expression

instr Indented author language instructions (must be regular)

Effect:

Begins a loop. If the value of **expr** is true, the author language instructions immediately following –loop– are executed. If the value of **expr** is false, execution resumes following the –endloop– instruction.

Comments:

A blank tag is equivalent to a true expression. Every loop starting with a –loop– instruction must end with an –endloop– instruction. Instructions between –loop– and –endloop– must be indented (except for –outloop– and –reloop–).

OUTLOOP

Instruction format:

```
outloop   expr
```

Tag definition:

expr Logical or arithmetic expression

Effect:

Provides a conditional exit from a loop. If the value of **expr** is true, execution resumes following the –endloop– instruction. If the value of **expr** is false, the author language instructions immediately following –outloop– are executed.

Comments:

Must be at same level of indenting as last –loop– instruction. Must have a tag.

RELOOP

Instruction format:

```
reloop    expr
```

Tag definition:

expr Logical or arithmetic expression

Effect:

Provides a conditional branch to the –loop– instruction. If the value of **expr** is true, execution resumes with the previous –loop– instruction. If the value of **expr** is false, the author language instructions immediately following –reloop– are executed.

Comments:

Must be at same level of indentation as last –loop– instruction. Must have a tag.

ENDLOOP

Instruction format:

```
endloop
```

Effect:

Ends a loop. Branches immediately to the –loop– instruction which began the loop.

Comments:

Must have a blank tag.

COMMON

Instruction format:

common **num** {,**opt**}

common {**lesson**},**block,num** {,**opt**}

Tag definition:

num Number of variables declared (maximum of 8000 words)

lesson Lesson containing permanent common; optional if **lesson** is same lesson containing –common– instruction

block Name of block or blocks containing permanent common

opt Options

Effect:

One argument: Specifies the number of variables in temporary common

Three argument: Obtains specified number of variables from permanent common located in lesson **lesson,** block **block**

Options: Read only, no load, ronl (combined read only and no load), or checkpt (permanent common is returned to disk approximately every 8 minutes)

Comments:

This nonexecutable instruction can appear anywhere in a lesson because common is allocated at condense time. Can be used once per lesson. If common is less than 1500 words, no –comload– instruction is required after –common–. If common is more than 1500 words, a –comload– instruction after –common– is required.

COMMONX

Instruction format:

commonx {**lesson**},**block,num,**{**code**} {,**opt**}

Tag definition:

lesson Lesson containing permanent common; optional if lesson is same lesson containing –commonx– instruction

block Name of block or blocks containing permanent common

num Number of variables requested

code Codeword of common (optional)

opt Options (read only, no load, ronl, and checkpt)

Effect:

Common variables requested by –commonx– are allocated when –commonx– is executed.

Comments:

An executable form of the –common– instruction. All arguments can be variables. Can be used once per lesson.

COMLOAD

Instruction format:

comload

comload **cm,ecs,num**

Tag definition:

cm Starting position in central memory nc or vc variables

ecs Starting position in extended core storage common

num Number of variables (maximum of 1500 words per instruction)

Effect:

Loads **num** common variables from extended core storage (set by preceding –common–) into central memory nc and vc variables at the start of each time-slice and returns the updated values at the end of each time-slice.

Comments:

Unless different variables are to be loaded, a single instruction is sufficient. Can be a continued instruction (up to three lines) with one load specification per line. Subsequent –comload– instructions cause unload before load. Only one –comload– can be in effect at one time. A –comload– must be preceded by a –common– instruction. Blank tag stops –comload– execution.

COMRET

Instruction format:

comret

Effect:

Copies current permanent common from ECS to disk.

Comments:

ECS copy of common is not affected.

DATASET

Instruction format:

dataset

dataset **fn** {**,access**} {**,code**}

Tag definition:

fn Dataset file name

access Type of access desired (optional):

-1	⇒	Read/write
≠ -1	⇒	Read only
blank	⇒	Read/write

code Common code word (for read/write access) or inspect code word (for read-only access) of dataset (required if common code word of lesson does not match corresponding code word of dataset); can be a variable or a literal; optional

Effect:

Establishes connection between lesson and dataset or nameset file. After successful execution, records may be read from or written to the dataset by using –datain– or –dataout–. A blank tag terminates the connection to the current dataset.

Comments:

Operates in the same manner as the –readset– instruction but successful only if the named file is a dataset or a nameset. System-reserved word zreturn is set to:

-1	Connection made (lesson and dataset code words matched)
0	If the file is not found
1	If codes do not match
2	If file is in use elsewhere (file is being edited)
≥3	System disk errors

The system-reserved words zdsname, zrecs, and zwpr are also set following successful execution of the –dataset– instruction. If the dataset is a nameset, the system-reserved words znscpn, znsmaxn, znsmaxr, znsnams, and znsrecs are also set. Refer to appendix B.

DATAIN

Instruction format:

datain **srec;loc** {**;num**}

Tag definition:

srec Number of the first record to be read

loc Location to which the first block is written as follows:

Location	Format
storage	storage,x or s,x
common	common,x or c,x
student variables	nx

num Number of records to be read (optional); if omitted, 1 is assumed

Effect:

Reads **num** records from the dataset file, beginning at **srec**, into storage, common, or student variables.

Comments:

Up to 10 dataset-type instructions may be used in a –finish– unit; requires a previous –dataset– instruction. Delimiters must be semicolons.

DATAOUT

Instruction format:

dataout **srec;loc {;num}**

Tag definition:

srec Number of the first record to which data will be written

loc Beginning location from which data will be written as follows:

Location	Format
storage	storage,x or s,x
common	common,x or c,x
student variables	nx

num Number of records to be written (optional); if omitted, 1 is assumed

Effect:

Writes **num** records from storage, common, or student variables into the dataset file, beginning at **srec.**

Comments:

Up to 10 dataset-type instructions may be used in a –finish– unit; requires a previous –dataset– instruction. Delimiters must be semicolons.

SETNAME

Instruction format:

setname **{var}**

setname nextname

setname backname

Tag definition:

var Variable in which name starts

nextname Keyword nextname must be used in the –setname nextname– instruction

backname Keyword backname must be used in the –setname backname– instruction

Effect:

Selects a named set of records from a nameset for future reference. Selects the next name in alphabetical order when the tag is nextname. Selects the preceding name in alphabetical order when the tag is backname. A blank tag clears the current name.

Comments:

The argument **var** cannot be a constant or a literal.

GETNAME

Instruction format:

getname **name{,var}**

Tag definition:

name Starting variable for return of the name

var Variable where the information associated with the name is stored (optional)

Effect:

Returns the name currently selected from a nameset and its 24 bits of associated information. Returns a full word of zero if no name is currently in effect.

Comments:

Returns only full words.

ADDNAME

Instruction format:

addname **name{,num{,var}}**

Tag definition:

name Starting variable of name to add to nameset

num Number of records to add with **name;** default is 1

var Variable containing 24 bits of associated information to add with name

Effect:

Adds a new name and new set of records to a nameset.

Comments:

When **var** is specified, rightmost 24 bits are taken.

DELNAME

Instruction format:

delname

Effect:

Deletes the current selected name and its records.

Comments:

This is the only method of deleting a name from a nameset.

RENAME

Instruction format:

rename **name** {**,var**}

Tag definition:

name Starting variable of name to replace old name

var Variable containing 24 bits of new associated information

Effect:

Changes the name of the currently selected name. Can also change the 24 bits of associated information.

Comments:

If **var** is not specified, the current associated information stays the same.

NAMES

Instruction format:

names **pos,var,num,return**

Tag definition:

pos Starting position in list of names

var Starting variable for storing names

num Maximum number of words available for storing names

return Variable returning number of names actually obtained

Effect:

Returns a set of names from the current list in the nameset.

Comments:

Each name entry uses 1, 2, or 3 words for the name plus 1 word for the associated information.

ADDRECS

Instruction format:

addrecs {**record,**} **num**

Tag definition:

record Record number assigned to first added record

num Number of records to add

Effect:

Adds records to the current named set of records. One-argument form adds records to end of set of records.

DELRECS

Instruction format:

delrecs {**record,**} **num**

Tag definition:

record Starting record number to be deleted

num Number of records to delete

Effect:

Deletes records from the current named set of records. One-argument form deletes records from end of set of records.

Comments:

If all records are deleted, the name remains with no records.

RESERVE

Instruction format:

reserve	common
reserve	dataset
reserve	records,**record,num**

Tag definition:

common	The tag common must be used in the –reserve common– instruction
dataset	The tag dataset must be used in the –reserve dataset– instruction
records	The tag records must be used in the –reserve records– instruction
record	Starting record number of dataset to –reserve–
num	Number of records of dataset to –reserve–

Effect:

Sets a flag to indicate that this terminal is using common or dataset records.

Comments:

System-reserved word zreturn is returned equal to –1 if common is reserved successfully or is returned equal to the station number of the terminal which has common reserved. zreturn is –2 if the terminal already has common reserved. After a –reserve dataset– instruction, zreturn is set as follows:

-2	Records already reserved by this terminal
-1	–reserve– successful
0	No preceding –dataset– instruction
1	No such record number(s) in dataset
2	Terminal has read-only access to dataset
5+n	Record(s) of dataset reserved by terminal number n

The –reserve common– instruction does not lock out access to the common variables. However, appropriate branches can be used on the value of zreturn such that a lock-out will occur in the lesson. The –reserve dataset– instruction prevents other users from writing into the entire dataset or nameset. The –reserve records– instruction prevents other users from writing into the reserved records(s) of the dataset or nameset. Dataset records which are reserved by another terminal can still be read.

RELEASE

Instruction format:

release	common
release	dataset
release	records,**record,num**

Tag definition:

common	The tag common must be used in the –release common– instruction
dataset	The tag dataset must be used in the –release dataset– instruction
records	The tag records must be used in the –release records– instruction
record	Starting record number of dataset to –release–
num	Number of records of dataset to –release–

Effect:

Clears flag set by –reserve–. Not cleared if common or dataset records are not reserved by this station.

Comments:

System-reserved word zreturn is returned equal to –1 if common is released successfully or is returned equal to the station number of the terminal which has common reserved. zreturn is –2 if common is not reserved by any terminal. After a –release dataset– instruction, zreturn is set as follows:

-2	Dataset record(s) not reserved by any terminal
-1	–release– successful
0	No preceding –dataset– instruction
1	No such record number(s) in dataset
5+n	Record(s) in dataset are reserved by terminal number n

Common variables are automatically released if an execution error occurs in the lesson or if the user exits by pressing SHIFT STOP. Dataset records are automatically released when the user exits from the lesson by any means, when another –dataset– instruction is executed, or when a –dataset (0)– instruction is executed.

ABORT

Instruction format:

abort	common	
abort	record	
abort	autocheck	
abort	common, record, autocheck	(Up to three tags can be used with a single –abort– command.)

Effect:

The –abort common– instruction transforms a permanent common into a temporary common. Common is not returned to the disk (updated) when the last user leaves student mode. Lesson(s) containing common continue to function as before, but all recent changes to common are never returned to the disk. Execution errors occur if there is no common or read-only common.

The –abort record– instruction causes student records not to be returned back to the disk when the student signs off. However, CPU time, number of sessions, etc. are updated at sign-off time. Autocheckpoint function is included with –abort record–. An execution error occurs if user is not a registered student with the –abort record– and –abort autocheck– commands.

Autocheckpoint of common does not occur when an –abort autocheck– is in effect.

STORAGE

Instruction format:

storage	**num**

Tag definition:

num	Number of words of storage

Effect:

Reserves the requested number of words in ECS.

Comments:

Only one –storage– instruction can be used in a lesson. Maximum length of storage is 1500 words.

STOLOAD

Instruction format:

stoload	
stoload	**cm,stor,num**

Tag definition:

cm	Starting variable position in central memory nc or vc variables
stor	Starting variable position in storage
num	Number of variables (maximum of 1500 words per instruction)

Effect:

Loads **num** variables from storage (set by a preceding –storage–) into central memory nc and vc variables at the beginning of each time-slice and returns the updated values at the end of each time-slice.

Comments:

Unless different variables are to be loaded, a single –stoload– is sufficient. Can be a continued instruction (up to three lines) with one load specification per line. A blank tag cancels previous –stoload–. A –stoload– must be preceded by a –storage– instruction.

BLOCK

Instruction format:

block	**start,store,num**

Tag definition:

start	Starting location of block to be moved
store	Starting location of block destination
num	Number of variables to move (can be 0)

Effect:

Copies **num** variables from the block starting at **start** to the block starting at **store.**

TRANSFR

Instruction format:

transfr **start;store;num**

Tag definition:

start Starting location of block to be moved

store Starting location of block destination

num Number of variables to move (can be 0)

Effect:

Copies **num** variables from the block starting at **start** to the block starting at **store.**

Comments:

More general than the –block– instruction but with the same type of effect. The arguments must be separated with semicolons. The locations can have any of the following forms.

nx vx	Student variables
ncx vcx	Central memory variables
common,x c,x	ECS common variables
storage,x s,x	Storage variables

In all cases, x is the variable number.

DISPLAY INSTRUCTIONS

The following instructions are used to display information.

at	showz	rdraw
atnm	showe	gdraw
rorigin	showa	circle
gorigin	text	circleb
rat	hidden	rcircle
gat	erase	gcircle
ratnm	eraseu	box
gatnm	size	rbox
write	rotate	gbox
writec	dot	vector
show	rdot	rvector
showt	gdot	gvector
showo	draw	window
mode	tabset	lscalex
color	slide	lscaley
embed	enable	labelx
catchup	disable	labely
delay	audio	markx
char	play	marky
plot	record	graph
charset	ext	hbar
chartst	extout	vbar
lineset	axes	delta
altfont	bounds	funct
micro	scalex	polar
codeout	scaley	

AT

Instruction format:

at **loc**

at **finex,finey**

Tag definition:

loc Coarse-grid location

finex Fine-grid horizontal location

finey Fine-grid vertical location

Effect:

Sets position and margin for display instructions.

ATNM

Instruction format:

atnm **loc**

atnm **finex,finey**

Tag definition:

loc Coarse-grid location

finex Fine-grid horizontal location

finey Fine-grid vertical location

Effect:

Sets position for display instructions but does not set a left margin for continued lines (second and subsequent lines) of the display. Second and subsequent lines are displayed at the leftmost margin (default left margin).

RORIGIN

Instruction format:

rorigin	**loc**
rorigin	**finex,finey**
rorigin	

Tag definition:

loc	Coarse-grid location
finex	Fine-grid horizontal location
finey	Fine-grid vertical location
blank	Sets origin to current wherex and wherey

Effect:

Specifies an origin for the relocatable instructions.

Comments:

Remains in effect until another –rorigin– is executed. If no –rorigin– is specified when entering a lesson, –rorigin– is set to 0,0.

GORIGIN

Instruction format:

gorigin	**loc**
gorigin	**finex,finey**
gorigin	

Tag definition:

loc	Coarse-grid location
finex	Fine-grid horizontal location
finey	Fine-grid vertical location
blank	Sets origin to current wherex and wherey

Effect:

Specifies an origin for the relative graphics instructions.

Comments:

Remains in effect until another –gorigin– is executed. If no –gorigin– is specified when entering a lesson, –gorigin– is set to 0,0.

RAT

Instruction format:

rat	**loc**
rat	**finex,finey**
rat	

Tag definition:

loc	Coarse-grid location
finex	Fine-grid horizontal location
finey	Fine-grid vertical location
blank	Sets screen position to current –rorigin–

Effect:

The –rat– (relocatable at) instruction determines a new screen position relative to the specified –rorigin–. The current state of the –size– and –rotate– instructions is taken into account. The –rat– instruction is used with text display instructions and with relocatable instructions.

GAT

Instruction format:

gat	**x,y**

Tag definition:

x	Scaled horizontal position
y	Scaled vertical position

Effect:

Places cursor (position) at specified location on the screen for subsequent graphics instructions.

Comments:

Equivalent to the fine grid –at– instruction, except that the location is specified in scaled graph coordinates relative to the specified –gorigin–.

RATNM

Instruction format:

ratnm **loc**

ratnm **finex,finey**

Tag definition:

loc	Coarse-grid location
finex	Fine-grid horizontal location
finey	Fine-grid vertical location

Effect:

Sets position for display instruction with respect to –rorigin– but does not set a left margin for continued lines (second and subsequent lines) of the display. Continued lines are displayed at the leftmost margin.

GATNM

Instruction format:

gatnm **x,y**

Tag definition:

x	Scaled horizontal location
y	Scaled vertical location

Effect:

Sets position for graphics instruction with respect to –gorigin– but does not set a left margin for continued lines (second and subsequent lines) of the display. Continued lines are displayed at the leftmost margin.

Comment:

Equivalent to the fine grid –atnm– instruction, except that the location is specified in scaled graph coordinates relative to the specified –gorigin–.

WRITE

Instruction format:

write **message**

Tag definition:

message	Text to be displayed (may include embedded instructions)

Effect:

Displays text, starting at current screen position (may be specified by an –at–, –rat–, –gat–, –atnm–, –ratnm–, or –gatnm– instruction).

Comments:

Tag can be more than one line in length. Refer also to embed.

WRITEC

Instruction format:

writec **expr,mess1,mess2,...**

writec **expr†mess1†mess2†...**

Tag definition:

expr	Mathematical expression
mess1, mess2,...	Text to be displayed (may include embedded instructions)

Effect:

Displays one of several possible messages on the screen.

Comments:

Second form uses universal delimiters (ACCESS ,). Omitted arguments use successive delimiters, not x or q. Tag can be more than one line.

SHOW

Instruction format:

show **var {,num {,absval}}**

Tag definition:

var	Variable whose value is to be displayed
num	Number of significant figures desired
absval	Minimum absolute value to be displayed (0 to 1)

Effect:

Displays value of first argument, which can be an expression. If the value to be displayed has more than **num** + 4 digits before the decimal point, it is displayed in exponential format. If the value is less than 10^{-4}, it is also displayed in exponential format. If the value is less than the absolute value (**absval**), 0 is displayed.

Comments:

The default value of **num** is 4. The default value of **absval** is 10^{-9}.

SHOWT

Instruction format:

showt **var {,format}**

Tag definition:

var	Variable whose value is to be displayed
format	Format of number (l.r, or l,r)

Effect:

Displays the specified variable at the current screen position, right-justified, with the length or format specified.

Comments:

A value too large for the specified format displays asterisks (****). Integer default length is eight digits. Floating-point default format is 4,3.

SHOWO

Instruction format:

showo **var {,num}**

Tag definition:

var	Variable whose value is to be displayed
num	Number of octal digits to be displayed (default is 21)

Effect:

Displays value of specified variable in octal.

Comments:

If	**num** $\geq$ 20	(**num**-20) leading blanks are supplied
	num $\geq$ 64	**num** is set to 64
	num = 0	Instruction is ignored
	num < 0	Execution error is generated

If variable cannot be displayed in the specified number of digits, **num** asterisks are displayed.

SHOWZ

Instruction format:

showz **var {,num}**

Tag definition:

var	Variable whose value is to be displayed
num	Number of significant figures desired

Effect:

Displays value of first argument, which can be an expression. If value does not fit in the length specified in the second argument, exponential format is used, displaying precisely the number of places specified.

SHOWE

Instruction format:

showe **var {,num {,format}}**

Tag definition:

var	Variable whose value is to be displayed
num	Number of digits to be displayed (optional); default is 4
format	Type of scientific notation (optional); default is 0

Effect:

Displays specified number of digits in exponential format. If **format** (optional third argument) is 0, the standard format is displayed (for example, 3.00×10^2). If **format** is nonzero, the ** format is displayed (for example, 3.00*10**2). Leading space or - is automatically supplied to permit tabular displays.

Comments:

Width of the display field depends on exponent size. If the optional third argument is used, the optional second argument (**num**) must be specified.

SHOWA

Instruction format:

showa **var,num**

Tag definition:

var	Variable whose value is to be displayed
num	Number of characters to be displayed

Effect:

Displays contents of specified variable in alphanumeric form. Assumes character string is left-justified.

Comments:

Contents of more than one (consecutive) variable are displayed if **num** is greater than 10. Most shifted and access characters must be counted as two characters.

TEXT

Instruction format:

text **var,num**

Tag definition:

var	Starting variable of alphanumeric buffer
num	Number of computer words to be displayed (must be $\leq$ 468)

Effect:

Displays contents of an alphanumeric buffer, automatically starting new lines.

Comments:

Does not allow literals in the tag, and is not affected by the –size– or –rotate– instructions. Execution of the –text– instruction does not update the system-reserved word where.

HIDDEN

Instruction format:

hidden **var** {**,num**}

Tag definition:

var	Variable containing alphanumeric characters
num	Number of characters to be displayed (optional)

Effect:

Displays contents of **var**, showing all hidden 6-bit codes.

Comments:

The default value of **num** is 10. After execution of the –hidden– instruction, the value of system-reserved word where is unreliable. Does not work well with –mode rewrite–.

ERASE

Instruction format:

erase

erase **num** {**,lines**}

erase abort

Tag definition:

0	Zero value
num	Positive value
lines	Number of lines
abort	The keyword abort must be used with the –erase abort– instruction

Effect:

blank	Entire screen erased and pending output not aborted
0	Instruction ignored
num	Number of characters erased
num,lines	Block of **num** characters by **lines** erased
abort	Entire screen erased and pending output aborted

ERASEU

Instruction format:

eraseu

eraseu **uname**

eraseu **expr,uname1,uname2,...**

Tag definition:

uname	Unit name
uname1, uname2,...	Unit names, x, or q
expr	Variable or mathematical expression

Effect:

Causes unit named in tag to be executed when any of the following keys are pressed after an ok or no judgment.

ERASE
SHIFT ERASE
NEXT (not after ok)
EDIT
SHIFT EDIT

Comments:

After execution, –eraseu– remains in effect for that entire main unit. Conditional form x causes the –eraseu– to have no effect; q or –eraseu– with no tag clears previous –eraseu–.

SIZE

Instruction format:

size **s**

size **sizex,sizey**

Tag definition:

s	Size of text
sizex	Horizontal character size
sizey	Vertical character size

Effect:

Sets size of text to tag value times normal size. –size 0– returns to normal size.

Comments:

When size is other than zero, the text is displayed slower than normal text. The –size– instruction remains in effect between unit boundaries. Alternate characters display the associated nonprogrammable character if the size is not zero and no –lineset– instruction is in effect. If a –lineset– instruction is in effect, the size is nonzero, and alternate-character text is used, the appropriate –lineset– character is displayed.

ROTATE

Instruction format:

rotate **angle**

Tag definition:

angle	Angle of rotation (in degrees)

Effect:

Rotates text through **angle** specified before display.

Comments:

Normal (size zero) and alternate character text unaffected by –rotate– instruction. –lineset– characters are affected by the –rotate– instruction.

DOT

Instruction format:

dot **loc**

dot **finex,finey**

Tag definition:

loc	Coarse-grid location
finex	Horizontal fine-grid location
finey	Vertical fine-grid location

Effect:

Lights up one screen dot at specified location.

Comments:

Coarse-grid location lights dot in lower-left corner of character space.

RDOT

Instruction format:

dot	**loc**
dot	**finex,finey**

Tag definitions:

loc	Coarse-grid location
finex	Horizontal fine-grid location
finey	Vertical fine-grid location

Effect:

Lights up one screen dot at specified location relative to –rorigin–.

Comments:

Coarse-grid location lights dot in lower-left corner of character space.

GDOT

Instruction format:

gdot	**finex,finey**

Tag definition:

finex	Horizontal fine-grid location
finey	Vertical fine-grid location

Effect:

Lights up one screen dot at specified location relative to –gorigin–.

Comments:

The tag for the –gdot– instruction must use fine-grid coordinate form.

DRAW

Instruction format:

draw	**loc1;loc2; ...**
draw	**;loc1;loc2; ...**

Tag definition:

loc1; **loc2; ...**	Coarse- or fine-grid locations or keyword skip separated by semicolons

Effect:

Draws lines between locations specified, in the order specified. Initial semicolons start from current screen position; skip moves to next position without a line being drawn.

Comments:

Fine- and coarse-grid coordinates can be mixed in a single tag. Fine-grid coordinates give horizontal and vertical positions, in that order, separated by commas. System-reserved words where, wherex, and wherey are not updated until the instruction completes execution.

RDRAW

Instruction format:

rdraw	**loc1;loc2; ...**
rdraw	**;loc1;loc2; ...**

Tag definition:

loc1; **loc2; ...**	Fine-grid locations relative to –rorigin– or keyword skip separated by semicolons

Effect:

Allows specification of a figure with respect to –rorigin–. Allows sizing and rotation of the figure using the –size– and –rotate– instructions.

GDRAW

Instruction format:

gdraw **loc1;loc2; ...**

gdraw **;loc1;loc2; ...**

Tag definition:

loc1; **loc2; ...**	Fine-grid locations relative to –gorigin– or keyword skip separated by semicolons

Effect:

Draws lines between locations specified. Initial semicolons start from current –gorigin– position.

Comments:

The tag for the –gdraw– instruction must use the fine-grid coordinate form.

CIRCLE

Instruction format:

circle **radius**

circle **radius,beg,end**

Tag definition:

radius	Radius of circle in fine-grid dots
beg	Starting angle from the horizontal (in degrees); required when drawing arcs
end	Ending angle from the horizontal (in degrees); required when drawing arcs

Effect:

Draws a circle, as defined in the first instruction format, or partial circle (arc), as defined in the second instruction format. The center of the circle or arc is defined by the current system-reserved words wherex and wherey (usually set by a preceding –at– or –atnm– instruction). wherex and wherey are unchanged by the whole circle instruction but are reset to the last point drawn on the circumference of an arc.

Comments:

The –circle– instruction also performs automatic windowing at the end of the screen.

CIRCLEB

Instruction format:

circleb **radius**

circleb **radius,beg,end**

Tag definition:

radius	Radius of circle in fine-grid dots
beg	Starting angle from the horizontal (in degrees); required when drawing arcs
end	Ending angle from the horizontal (in degrees); required when drawing arcs

Effect:

Draws a broken (dashed) circle or ellipse, as defined in the first instruction format, or partial circle (arc), as defined in the second instruction format. The center of the circle or arc is defined by the current system-reserved words wherex and wherey (usually set by a preceding –at– or –atnm– instruction). wherex and wherey are unchanged by the whole circle instruction but are reset to the last point drawn on the circumference of an arc.

Comments:

The –circleb– instruction also performs automatic windowing at the edge of the screen.

RCIRCLE

Instruction format:

rcircle **radius**

rcircle **radius,beg,end**

Tag definition:

radius Radius of circle in fine-grid dots

beg Starting angle from the horizontal (in degrees); required when drawing arcs

end Ending angle from the horizontal (in degrees); required when drawing arcs

Effect:

Draws a circle or ellipse, as defined in the first instruction format, or partial circle (arc), as defined in the second instruction format relative to the current –rorigin–. The center of the circle, ellipse, or arc is defined by the current system-reserved words wherex and wherey (usually set by a preceding –rat– or –ratnm– instruction). Ellipses are drawn only if the sizex and sizey scales are different. wherex and wherey are unchanged by the whole circle instruction but are reset to the last point drawn on the circumference of an arc. The –rcircle– instruction is affected by the –size– and –rotate– instructions.

Comments:

The –rcircle– instruction also performs automatic windowing at the edge of the screen.

GCIRCLE

Instruction format:

gcircle **radius**

gcircle **radius,beg,end**

Tag definition:

radius Radius of circle in fine-grid dots

beg Starting angle from the horizontal (in degrees); required when drawing arcs

end Ending angle from the horizontal (in degrees); required when drawing arcs

Effect:

Draws a circle or ellipse, as defined in the first instruction format, or partial circle (arc), as defined in the second instruction format relative to the current –gorigin–. The center of the circle, ellipse, or arc is defined by the current system-reserved words wherex and wherey (usually set by a preceding –gat– or –gatnm– instruction). Ellipses are drawn only if the x and y scales are different. wherex and wherey are unchanged by the whole circle instruction but are reset to the last point drawn on the circumference of an arc.

Comments:

The –gcircle– instruction also performs automatic windowing at the edge of the screen.

BOX

Instruction format:

box {{**loc1** } ;} **loc2** {;**thick**}

Tag definition:

loc1 One corner (coarse- or fine-grid coordinates) of box to be displayed (optional)

loc2 Opposite corner (coarse- or fine-grid coordinates) of box to be displayed

thick Thickness of the box wall in fine-grid dots (optional)

Effect:

Draws a rectangular box whose opposite (diagonal) corners are at the two locations specified (**loc1** and **loc2**). If the first location is omitted (for example, –box ;**loc2**–), the current screen position is used for the other corner. If only one location is specified (for example, –box 910–), a box is drawn with one corner at the specified location and the other corner at location 0,0.

Comments:

If the thickness value (**thick**) is positive, the box wall is built in an outward direction from the corners. If thickness is a negative value, the buildup is in an inward direction. Thickness values of –1, 0, 1, and a blank (**thick** argument not specified) mean that the box wall is to be one line thick. This value must be less than 50.

RBOX

Instruction format:

rbox {{**loc1** } ; } **loc2** {;**thick**}

Tag definition:

loc1 One corner (coarse- or fine-grid coordinates) of box to be displayed (optional)

loc2 Opposite corner (coarse- or fine-grid coordinates) of box to be displayed

thick Thickness of the box wall in fine-grid dots (optional)

Effect:

Draws a rectangular box whose opposite (diagonal) corners are at the two locations specified (**loc1** and **loc2**). If the first location is omitted (for example, –rbox ;**loc2**–), the current screen position is used for the other corner. If only one location is specified (for example, –rbox 812–), a box is drawn with one corner at the specified location and the other corner relative to the current –rorigin–. –rbox– locations are affected by the –size– and –rotate– instructions.

Comments:

If the thickness value (**thick**) is positive, the box wall is built in an outward direction from the corners. If thickness is a negative value, the buildup is in an inward direction. Thickness values of –1, 0, 1, and a blank (**thick** argument not specified) mean that the box wall is to be one line thick. This value must be less than 50.

GBOX

Instruction format:

gbox

gbox {{**floc1** } ; } **floc2** {;**thick**}

Tag definition:

floc1 One corner (fine-grid coordinates) of box to be displayed (optional)

floc2 Opposite corner (fine-grid coordinates) of box to be displayed

thick Thickness of the box wall in fine-grid dots (optional)

Effect:

Draws a rectangular box whose opposite (diagonal) corners are at the two locations specified (**floc1** and **floc2**). If the first location is omitted (for example, –gbox ;**floc2**–), the current screen position is used for the other corner. If only one location is specified (for example, –gbox 200,200–), a box is drawn with one corner at the specified location and the other corner relative to the current –gorigin–. The –gbox– instruction uses scaled coordinates as specified by previous –scale– and –bounds/axes– instructions. A –gbox– instruction with a blank tag draws a box around the current –bounds–.

Comments:

If the thickness value (**thick**) is positive, the box wall is built in an outward direction from the corners. If thickness is a negative value, the buildup is in an inward direction. Thickness values of –1, 0, 1, and a blank (**thick** argument not specified) mean that the box wall is to be one line thick. This value must be less than 50.

VECTOR

Instruction format:

vector {{**loc1** } ; } **loc2** {;**size**}

Tag definition:

loc1 Horizontal and vertical location (coarse- or fine-grid coordinates) of the arrow's tail (optional)

loc2 Horizontal and vertical location (coarse- or fine-grid coordinates) of the arrow's head

size Size of arrow's head; default is 10.5 (optional)

Effect:

Draws a vector (pointer or arrow) with its tail at the first location (**loc1**) and its head at the second location (**loc2**). If the first location is omitted (for example, –vector ;**loc2**–), the tail is at the current screen position. If only one location is given (for example, –vector 910–), a vector is drawn with the head at the specified location and the tail at location 0,0.

Comments:

A positive-size arrowhead is a closed triangle. A negative-size arrowhead is open (barbed). When size ≥ 1, it specifies the absolute size of the arrowhead in dots. When size < 1, it specifies the size of the arrowhead relative to the length of the vector. This headsize changes with –size–, –scalex–, and –scaley– instructions.

RVECTOR

Instruction format:

rvector {{**loc1**} ;} **loc2** {;**size**}

Tag definition:

loc1 Horizontal and vertical location (coarse-or fine-grid coordinates) of the arrow's tail (optional)

loc2 Horizontal and vertical location (coarse-or fine-grid coordinates) of the arrow's head

size Size of arrow's head; default is 10.5 (optional)

Effect:

Draws a vector (pointer or arrow) with its tail at the first location (**loc1**) and its head at the second location (**loc2**). If the first location is omitted (for example, –rvector ;**loc2**–), the tail is at the current screen position. If only one location is given (for example, –rvector 1210–), a vector is drawn with the head at the specified location and the tail at the current –rorigin–. Vectors drawn with the –rvector– instruction are affected by the –size– and –rotate– instructions.

Comments:

A positive-size arrowhead is a closed triangle. A negative-size arrowhead is open (barbed). When size ≥ 1, it specifies the absolute size of the arrowhead in dots. When size < 1, it specifies the size of the arrowhead relative to the length of the vector. This headsize changes with –size–, –scalex–, and –scaley– instructions.

GVECTOR

Instruction format:

gvector {{**floc1**} ;} **floc2** {;**size**}

Tag definition:

floc1 Scaled horizontal and vertical location (fine-grid coordinates) of the arrow's tail (optional)

floc2 Scaled horizontal and vertical location (fine-grid coordinates) of the arrow's head

size Size of arrow's head; default is 10.5 (optional)

Effect:

Draws a vector (pointer or arrow) with its tail at the first location (**loc1**) and its head at the second location (**loc2**). If the first location is omitted (for example, –gvector ;**loc2**–), the tail is at the current screen position. If only one location is given (for example, –gvector 220,220–), a vector is drawn with the head at the specified location and the tail at the current –gorigin–. The –gvector– instruction uses scaled coordinates as specified by previous –scale– and –bounds/axes– instructions. The tag for the –gvector– instruction must use the fine-grid coordinate form.

Comments:

A positive-size arrowhead is a closed triangle. A negative-size arrowhead is open (barbed). When size ≥ 1, it specifies the absolute size of the arrowhead in dots. When size < 1, it specifies the size of the arrowhead relative to the length of the vector. This headsize changes with –size–, –scalex–, and –scaley– instructions.

WINDOW

Instruction format:

window

window {{**loc1**} ;} **loc2**

Tag definition:

loc1 One corner (coarse- or fine-grid coordinates) of window (optional)

loc2 Opposite corner (coarse- or fine-grid coordinates) of window

Effect:

Limits display area within rectangle bounded by specified corners (**loc1** and **loc2**). If the first location is omitted (for example, –window ;**loc2**–), the current screen position is used for the other corner. If only one location is specified (for example, –window **loc2**–), one corner is at **loc2** and the other corner is at 0,0. Blank tag turns off previous –window– instruction.

Comments:

The –window– instruction remains in effect across unit boundaries. Size 0 text and –dot– are unaffected. Fine-grid coordinates are separated by a comma.

MODE

Instruction format:

mode **type**

mode **expr,type1,type2,...**

Tag definition:

type, type1, type2,... Keywords write, rewrite, or erase

expr Variable or mathematical expression

Effect:

Sets the mode in which the terminal operates.

Comments:

Mode rewrite erases an entire character space before displaying text.

COLOR

Instruction format:

color **type**

Tag definition:

type Keyword orange means write; keyboard black means erase

Effect:

The –color orange– instruction is equivalent to the –mode write– instruction, and the –color black– instruction is equivalent to the –mode erase– instruction.

EMBED (Not an Instruction)

	Normal Form	Embedded Form
show	**a1,a2,a3**	⊲s,a1,a2,a3⊳
showt	**a1,a2,a3**	⊲t,a1,a2,a3⊳
showz	**a1,a2**	⊲z,a1,a2⊳
showo	**a1,a2**	⊲o,a1,a2⊳
showe	**a1,a2** {,**format**}	⊲e,a1,a2 {,format}⊳
showa	**a1,a2**	⊲a,a1,a2⊳
at	**a1,a2**	⊲at,a1,a2⊳
atnm	**a1,a2**	⊲atnm,a1,a2⊳
size	**a1**	⊲size,a1⊳
rotate	**a1**	⊲rotate,a1⊳
mode	**tag**	⊲m,tag⊳

Effect:

Causes specified display action within a –write– or –writec– instruction.

Comments:

Normal default options are in effect, except that no leading blank is specified for the embedded –showo– instruction. The embed mode feature does not work with alternate font.

CATCHUP

Instruction format:

catchup

Effect:

Halts lesson execution until all display material previously specified has been shown on the terminal.

DELAY

Instruction format:

delay **num**

Tag definition:

num Number less than or equal to one (floating-point format)

Effect:

Causes execution delay specified by tag (in seconds).

CHAR

Instruction format

char **slot,a1,a2,a3,a4,a5,a6,a7,a8**

Tag definition:

slot Memory slot number

a1,...,a8 Dot specifications

Effect:

Defines a character associated with the specified slot of the alternate character memory. Each dot specification specifies one column of the character matrix, each binary one signifies a lighted dot, and each binary zero signifies an unlighted dot.

PLOT

Instruction format:

plot **slot**

Tag definition:

slot Memory slot number

Effect:

Displays alternate character in the specified memory slot.

Comments:

Can only display one character per instruction.

CHARSET

Instruction format:

charset

charset {**less,**} **name**

Tag definition:

less Lesson containing character set or variable containing lesson name; optional if current lesson contains character set

name Name of character set or variable containing name

Effect:

Loads specified character set into alternate character memory of the terminal. Blank tag clears information about which character set is loaded.

Comments:

Loading requires about 17 seconds for a full character set. Loading is not done if the character set has already been placed in the terminal.

CHARTST

Instruction format:

chartst {**less,**} **name**

Tag definition:

less Lesson containing character set or variable containing lesson name; optional if current lesson contains character set

name Name of character set or variable containing name

Effect:

Checks if character set named in the tag (**name**) is currently loaded into the terminal. System-reserved word zreturn is –1 if character set is loaded; it is 0 if character set is not loaded.

Comment:

The –chartst– instruction cannot determine if the character set was properly loaded, but only if an attempt has been made by using the –charset– instruction.

LINESET

Instruction format:

lineset

lineset {**less,**} **name**

Tag definition:

less Lesson containing lineset or variable containing lesson name; optional if current lesson contains lineset

name Name of lineset or variable containing name

Effect:

Obtains specified lineset from specified lesson. Blank tag clears information about which lineset is loaded.

Comments:

The –lineset– instruction may be used when sizing and rotating alternate characters; however, it can only be used if the current size does not equal zero. As with normal characters whose size is other than zero, characters are plotted slower than size-0 alternate characters. Following a –lineset– instruction, system-reserved word zreturn is set as follows:

-1	Loaded successfully
0	Lineset not found
1	Error in lineset

ALTFONT

Instruction format:

altfont **val**

Tag definition:

val	on, 1, alt, off, 0, normal

Effect:

Switches displayed characters into alternate character set for both student and author if tag is on, 1, or alt. Returns to normal character set if tag is off, 0, or normal.

Comments:

Source code remains in standard character set.

MICRO

Instruction format:

micro

micro **less,}name**

Tag definition:

less	Lesson containing micro table; optional if current lesson contains micro table
name	Name of micro table

Effect:

Obtains micro table from specified lesson. Blank tag loads the system micro table.

Comments:

System-reserved word zreturn is set to −1 if the micro table is loaded; zreturn is 0 if the micro table is not loaded.

CODEOUT

Instruction format:

codeout **val**

Tag definition:

val	Octal value between 010 and 017, inclusive

Effect:

Sends specified octal code to the terminal when executed.

TABSET

Instruction format:

tabset **o a1 a2 a3 a4 a5 a6 a7 a8 a9 a10**

Tag definition:

a1,...,a10	Octal two-digit fields

Effect:

Defines columns that student can use with the TAB key.

Comments:

Ten fields must be used, with lowercase o preceding first field only. Commas are not used.

SLIDE

Instruction format:

slide **num**

Tag definition:

num	Number of slide selected

Effect:

Selects designated slide from microfiche and displays it.

Comments:

There are three additional options.

512+**n**	Selects the slide but leaves bulb off
256+**n**	Selects the slide but closes shutter
noslide	Selects slide 0, turns bulb off, and closes shutter

ENABLE

Instruction format:

enable	touch
enable	ext
enable	touch,ext

Tag definition:

touch	Keyword touch must be used in the –enable touch– instruction.
ext	Keyword ext must be used in the –enable ext– instruction.

Effect:

Allows input from touch panel or external devices.

DISABLE

Instruction format:

disable	touch
disable	ext
disable	touch,ext

Tag definition:

touch	Keyword touch must be used in the –disable touch– instruction.
ext	Keyword ext must be used in the –disable ext– instruction.

Effect:

Allows no further input from touch panel or external devices.

AUDIO

Instruction format:

audio	**var**

Tag definition:

var	Variable or mathematical expression (0 to 32767)

Effect:

If the terminal is equipped with the audio disk feature, the –audio– instruction activates the prerecorded message specified in tag (**var**).

Comments:

The –audio– instruction may be used instead of the –play– instruction.

PLAY

Instruction format:

play	**track,sector,length**

Tag definition:

track	Audio disk track number (0 to 127) or mathematical expression
sector	Audio disk sector number (0 to 31) or mathematical expression
length	Number of consecutive sectors to be played (0 to 4095) or mathematical expression

Effect:

If the terminal is equipped with the audio disk feature, the –play– instruction activates the prerecorded message specified by the tag.

RECORD

Instruction format:

record	**track,sector,length**

Tag definition:

track	Audio disk track number (0 to 127) or mathematical expression
sector	Audio disk sector number (0 to 31) or mathematical expression
length	Number of consecutive sectors to be recorded (0 to 4095) or mathematical expression

Effect:

If the terminal is equipped with the audio disk feature, the –record– instruction activates recording capability of the device at the location specified by the tag.

Comments:

Audio messages entered with the –record– instruction are replayed with either an –audio– or –play– instruction.

EXT

Instruction format:

ext **var** {**,station**}

Tag definition:

var	Variable (type n or nc) or mathematical expression containing information in rightmost 15 bits
station	Number of other station to receive –ext– instruction.

Effect:

Sends rightmost 15 bits to external terminal accessory. Two-argument form checks if the other station wishes to receive –ext– instructions.

Comments:

Value of mathematical expression in **var** truncated to 15 bits. System-reserved word zreturn is set to -1 if the data is sent or to 0 if data is not sent.

EXTOUT

Instruction format:

extout **var** {**,num**}

Tag definition:

var	Variable (type n or nc) containing information in rightmost 16 bits
num	Number of words to be sent out; default value is 1

Effect:

Sends rightmost 16 bits of **num** words starting at **var** to the external jack of the terminal.

Comments:

An –erase abort– or –jump– instruction aborts pending –extout– output. To prevent this, use a –catchup– instruction before –erase abort– or –jump–.

AXES

Instruction format:

axes

axes **x+,y+**

axes **x–,y–,x+,y+**

Tag definition:

x+	Positive length of horizontal axis in fine-grid dots
y+	Positive length of vertical axis in fine-grid dots
x–	Negative length of horizontal axis in fine-grid dots
y–	Negative length of vertical axis in fine-grid dots

Effect:

Draws axes of the graph and sets graph limits. Blank tag redraws previously specified axes.

BOUNDS

Instruction format:

bounds **x+,y+**

bounds **x–,y–,x+,y+**

Tag definition:

x+	Positive horizontal boundary of graph in fine-grid dots
y+	Positive vertical boundary of graph in fine-grid dots
x–	Negative horizontal boundary of graph in fine-grid dots
y–	Negative vertical boundary of graph in fine-grid dots

Effect:

Establishes boundaries of graph without drawing the axes.

SCALEX

Instruction format:

scalex **max** {**,offset**}

Tag definition:

max Maximum horizontal value of graph

offset Value of horizontal axis at origin (default is 0)

Effect:

Scales the horizontal axis, allowing later references to be given in graph values, rather than fine-grid dots; **offset** allows graph origin to be other than (0,0).

SCALEY

Instruction format:

scaley **max** {**,offset**}

Tag definition:

max Maximum vertical value of graph

offset Value of vertical axis at origin (default is 0)

Effect:

Scales the vertical axis, allowing later references to be given in graph values, rather than fine-grid dots; **offset** allows graph origin to be other than (0,0).

LSCALEX

Instruction format:

lscalex **max** {**,offset**}

Tag definition:

max Maximum horizontal value of graph

offset Value of horizontal axis at origin (default is 1)

Effect:

Scales axes of the graph according to the common logarithm (base 10) for later reference; **offset** allows author determination of value at origin.

Comments:

Logarithms of negative numbers cannot be used. If a negative portion of the axis is specified, it is used for negative logarithms (logarithms of numbers between 0 and 1). Maximum value can be expressed as 10^4, instead of 10000.

LSCALEY

Instruction format:

lscaley **max** {**,offset**}

Tag definition:

max Maximum value of vertical axis

offset Value of vertical axis at origin

Effect:

Scales vertical axis according to the common logarithm (base 10) for later reference; **offset** allows the author to specify axis value at the origin.

Comments:

Logarithms of negative numbers cannot be used. If a negative portion of the axis is specified, it is used for negative logarithms (logarithms of numbers between 0 and 1). Maximum values can be expressed as 10^4, instead of 10000.

LABELX

Instruction format:

labelx **major** {,**minor**,**size** {,**format**}}

labelx **0** {,**minor**,**size** {,**format**}}

Tag definition:

major	Major mark interval
minor	Minor mark interval
size	Size of major and minor marks
format	Format of labels (l.r or l,r)

Effect:

Places major and minor marks on horizontal axis at specified scaled intervals and gives numeric labels of value.

Comments:

If axis is normally scaled, the first form is used. If no major mark interval is specified, a choice is automatically made. The second form is used if the axis is logarithmically scaled. In this case:

Minor Mark Argument	Marks Occur At:
- (minus sign)	none
0, 3, or none	1, 2, 5
5	1, 2, 3, 5, 7
10	1, 2, 3, 4, 5, 6, 7, 8, 9, 10

within each decade. Major marks are automatically made each decade.

The size may be:

0 or omitted	Normal length
1	Major marks extend to graph boundaries
2	Major and minor marks extend to graph boundaries

LABELY

Instruction format:

labely **major** {,**minor**,**size** {,**format**}}

labely **0** {,**minor**,**size** {,**format**}}

Tag definition:

major	Major mark interval
minor	Minor mark interval
size	Size of major and minor marks
format	Format of labels (l.r or l,r)

Effect:

Places major and minor marks on the vertical axis at the specified (scaled) intervals and attaches numeric value labels.

Comments:

If the axis is normally scaled, the first form is used. If no major mark interval is specified, a choice of interval is made automatically. The second form is used if the axis is scaled logarithmically. The minor mark intervals can be:

Minor Mark Argument	Marks Occur At:
- (minus sign)	none
0, 3, or none	1, 2, 5
5	1, 2, 3, 5, 7
10	1, 2, 3, 4, 5, 6, 7, 8, 9, 10

within each decade. Major marks are made automatically at each decade.

The size may be:

0 or omitted	Normal length
1	Major marks extend to graph boundaries
2	All marks extend to graph boundaries

MARKX

Instruction format:

markx **major {,minor,size}**

Tag definition:

major Major mark interval

minor Minor mark interval

size Size of major and minor marks

Effect:

Places major and minor marks on the horizontal axis at the specified (scaled) intervals.

Comments:

Equivalent to –labelx–, except that no numeric labeling is done.

MARKY

Instruction format:

marky **major {,minor,size}**

Tag definition:

major Major mark interval

minor Minor mark interval

size Size of major and minor marks

Effect:

Places major and minor marks on the vertical axis at the specified (scaled) intervals.

Comments:

Equivalent to –labely–, except that no numeric labeling is done.

GRAPH

Instruction format:

graph **x,y {,string}**

graph **x,y;var {,num}**

Tag definition:

x Scaled horizontal position

y Scaled vertical position

string Character string to be displayed (optional)

var Variable containing string to be displayed

num Number of characters to be displayed (optional)

Effect:

Two-argument form places a dot at specified location. Three-argument form displays third argument, beginning at specified location. Four-argument form displays **num** characters.

Comments:

Character string (if used) can be no longer than nine character codes if **string** is used or 10 character codes if **var** is used. The location of the string is moved somewhat down and to the right of the specified location to center the first character of the string.

HBAR

Instruction format:

hbar **x,y {,string}**

hbar **x,y {,var}**

hbar **x,y {,var,num}**

Tag definition:

x Scaled horizontal location of end of bar

y Scaled vertical location of center of bar

string Character string used in plotting bar

var Variable containing character string used in plotting bar

num Number of characters from variable to be used

Effect:

Draws horizontal bar from vertical axis to location specified.

Comments:

If a **string** is the third argument, it must be no longer than nine character codes. If the third argument is a variable name, the alphanumeric contents of the variable is used. All 10 character codes are used if there is no fourth argument. The fourth argument gives the number of characters, starting from the leftmost, to be used from **var.**

VBAR

Instruction format:

vbar	**x,y {,string}**
vbar	**x,y {,var}**
vbar	**x,y {,var,num}**

Tag definition:

x	Scaled horizontal location of the center of the bar
y	Scaled vertical location of the end of the bar
string	Character string used in plotting bar
var	Variable containing character string used in plotting bar
num	Number of characters from variable to be used in plotting bar

Effect:

Draws vertical bar from horizontal axis to location specified.

Comments:

If a **string** is the third argument, it must be no longer than nine character codes. If the third argument is a variable name, the alphanumeric contents of the variable is used. All 10 character codes are used if there is no fourth argument. The fourth argument gives the number of characters, starting from the leftmost, to be used from **var.**

DELTA

Instruction format:

delta	**inc**

Tag definition:

inc	Increment size for following –funct– instruction

Effect:

Specifies the increment size for each iteration of a following –funct– instruction that does not specify its own increment size.

Comments:

If –delta– instruction is omitted, default value is 1.

FUNCT

Instruction format:

funct	**f(x),x {<= xbeg,xend,dx}**

Tag definition:

f(x)	Function to be plotted
x	Independent (horizontal axis) variable
xbeg	Initial value of x
xend	Final value of x
dx	Increment size during iterations

Effect:

Plots lines connecting values of the function at specified increments.

Comments:

Iteration of the independent variable is done automatically. If the independent variable does not appear, explicitly or implicitly (through definitions) in the function, the function is plotted as a constant quantity. Independent variable should be floating-point, not integer.

Two-argument form	Uses increment specified by previous –delta– instruction.
Five-argument form	Specifies beginning and ending value of the independent variable plus an increment size; no previous –delta– instruction is required.

POLAR

Instruction format:

polar	
polar	**scale**
polar	**scx,scy**
polar	**neg**

Tag definition:

scale	Mutual scale in x and y directions
scx	Scale in x direction ($\theta = 0$ or $\theta = \pi$)
scy	Scale in y direction ($\theta = \pi/2$ or $\theta = 3\pi/2$)
neg	Negative number of any magnitude

Effect:

Replaces Cartesian coordinates with polar coordinates, with or without scaling. Negative tag turns off polar conversion and returns to Cartesian coordinates.

Comments:

Blank tag does not affect scaling. Single argument nonnegative tag scales both horizontal and vertical directions equally. Two-argument tag scales horizontal direction to first value and vertical direction to second value. Conversion remains in effect past –unit– bounds. All graphing commands are affected by –polar–.

LESSON CONTROL INSTRUCTIONS

The following instructions are used to control lesson execution.

unit	helpop	keylist
imain	help1	pause
next	help1op	collect
nextop	data	keytype
next1	dataop	force
next1op	data1	change
jump	data1op	use
join	lab	step
do	labop	in
goto	lab1	initial
exit	lab1op	lesson
nextnow	base	score
iferror	term	backgnd
entry	termop	foregnd
finish	*, c, $$	cpulim
time1	cstop	route
timer	cstart	routvar
end	cstop*	allow
back	restart	leslist
backop	status	addlst
back1	return	remov1
back1op	press	lname
stop	jumpout	findl
help	inhibit	

UNIT

Instruction format:

unit	**name (arg1,arg2,...,arg10)**

Tag definition:

name	Unit name
arg1, arg2,..., arg10	Argument list (maximum of 10)

Effect:

Denotes start of unit. Variables in the argument list of the –unit– instruction are given the value of the expression in the corresponding argument list of an instruction referencing that unit.

Comments:

Unit name must be no longer than eight characters. An instruction referencing an argumented unit may have fewer arguments than defined in the –unit– instruction but not more. When fewer arguments are referenced than defined, the nonreferenced arguments are unchanged.

IMAIN

Instruction format:

imain	**uname**
imain	**expr,uname1,uname2,...**

Tag definition:

uname	Unit name
expr	Variable or mathematical expression
uname1, uname2,...	Unit names, x, or q

Effect:

Executes unit named in tag at the start of every main unit in a lesson. The –imain– instruction is in effect for all main units executed after the unit containing the –imain–. Place –imain– instruction in the lesson's initial entry unit to have the imain unit in effect for the entire lesson.

Comments:

Later occurrences of –imain– override any earlier setting; q or –imain– with no tag clears previous setting; x leaves the previous setting in effect.

NEXT

Instruction format:

```
next
next        name
next        expr,name1,name2,...
```

Tag definition:

name	Unit name
expr	Variable or mathematical expression
name1, name2,...	Unit names, x, or q

Effect:

Specifies unit which follows the present one when all –arrow–s are satisfied in the present unit and the NEXT key is pressed.

Comments:

Both unconditional and conditional forms. Specification of x causes instruction to have no effect; specification of q clears the marker. The –next– instruction with a blank tag clears a previously specified –next– instruction.

NEXTOP

Instruction format:

```
nextop      name
nextop      expr,name1,name2,...
```

Tag definition:

name	Unit name
expr	Variable or mathematical expression
name1, name2,...	Unit names, x, or q

Effect:

Initiates a new main unit (specified by **name**) when all –arrow–s are satisfied in the present unit and the NEXT key is pressed.

Comments:

There is no full-panel erasure when –nextop– is executed. Any graphics or text specified by the –nextop– instruction is added to the current display and remains on the panel. The –nextop– instruction can have both unconditional and conditional forms. Specification of x causes instruction to have no effect; specification of q clears the marker.

NEXT1

Instruction format:

```
next1
next1       name
next1       expr,name1,name2,...
```

Tag definition:

name	Unit name
expr	Variable or mathematical expression
name1 name2,...	Unit names, x, or q

Effect:

Specifies unit which follows the present unit when the SHIFT NEXT keys are pressed. –arrow–s in present unit need not be satisfied.

Comments:

Both unconditional and conditional forms. Specification of x causes instruction to have no effect; specification of q clears the marker. The –next1– instruction with a blank tag clears a previously specified –next1– instruction.

NEXT1OP

Instruction format:

```
next1op     name
next1op     expr,name1,name2,...
```

Tag definition:

name	Unit name
expr	Variable or mathematical expression
name1, name2,...	Unit names, x, or q

Effect:

Initiates a new main unit (specified by **name**) when the SHIFT NEXT keys are pressed. –arrow– instructions in the present unit need not be satisfied.

Comments:

There is no full-panel erasure when –next1op– is executed. Any graphics or text specified by the –next1op– instruction is added to the current display and remains on the panel. The –next1op– instruction can have both unconditional and conditional forms. Specification of x causes instruction to have no effect; specification of q clears the marker.

JUMP

Instruction format:

jump **name**

$$unconditional

jump **expr,name1,name2,...**

$$conditional

jump **name(arg1,arg2,.,arg10)**

$$argument

Tag definition:

name	Unit name
expr	Mathematical expression
name1, name2,...	Unit names, x
arg1,arg2,., arg10	Variable or mathematical expressions that are passed to the unit as arguments

Effect:

Immediately transfers control to the specified new unit, which is fully initialized as a main unit.

Comments:

An x in the conditional form allows execution to continue past the instruction.

JOIN

Instruction format:

join **name**

$$unconditional

join **expr,name1,name2,...**

$$conditional

join **name,var⇐beg,end {,inc}**

$$iterative

join **expr,name1,name2,...,var⇐beg,end {,inc}**

$$conditional-iterative

join **name (arg1,arg2,.,arg10)**

$$argument

Tag definition:

name	Unit name
expr	Mathematical expression
name1, name2,...	Unit names, x, or q
var	Variable to be incremented
beg	Initial value of variable
end	Final value of variable
inc	Size of increment (default is 1)
arg1,arg2,., arg10	Variables or mathematical expressions that are passed to the unit as arguments

Effect:

The unconditional form of the –join– instruction executes the named (**name**) unit. The conditional form evaluates the specified expression: if negative, the first unit is executed; if 0, the second unit is executed, and so on. The iterative form executes a unit the specified number of times indicated in the iterative loop. The conditional-iterative form executes a specified unit from the list (**name1,name2,...**) per the current value of the expression (**expr**) by the number of times indicated in the iterative loop. The argument form passes specified arguments to the named unit and executes the named unit. After execution of the named unit, execution continues with the instruction following the –join– instruction.

Comments:

Conditional form x performs no action; q halts further execution of current unit, except in the conditional-iterative form (terminates iterative loop and continues execution of current unit). The –join– instruction is both a regular and a judging instruction. It never ends judging when the system is in the judging state.

DO

Instruction format:

do	**name**
	$$unconditional
do	**expr,name1,name2,...**
	$$conditional
do	**name,var<=beg,end {,inc}**
	$$iterative
do	**expr,name1,name2,...,var<=beg,end {,inc}**
	$$conditional-iterative
do	**name (arg1,arg2,.,arg10)**
	$$argument

Tag definition:

name	Unit name
expr	Mathematical expression
name1, name2,...	Unit names, x, or q
var	Variable to be incremented
beg	Initial value of variable
end	Final value of variable
inc	Size of increment (default is 1)
arg1,arg2,., arg10	Variables or mathematical expressions that are passed to the unit as arguments

Effect:

The unconditional form of the –do– instruction executes the named (**name**) unit. The conditional form evaluates the specified expression: if negative, the first unit is executed; if 0, the second unit is executed, and so on. The iterative form executes a unit the specified number of times in the same manner as a –join– instruction. The conditional-iterative form executes a specified unit from the list (**name1,name2,...**) per the current value of the expression (**expr**) by the number of times indicated in the iterative loop. The argument form passes specified arguments to the named unit and executes the named unit. After execution of the named unit, execution continues with the instruction following the –do– instruction.

Comments:

Conditional form x performs no action; q halts further execution of current unit, except in the conditional-iterative form (terminates iterative loop and continues execution of current unit). –do– is a regular instruction only.

GOTO

Instruction format:

goto	**name**
	$$unconditional
goto	**expr,name1,name2,...**
	$$conditional
goto	**name(arg1,arg2,.,arg10)**
	$$argument

Tag definition:

name	Unit name
expr	Mathematical expression
name1, name2,...	Unit names, x, or q
arg1,arg2,., arg10	Variables or mathematical expressions that are passed to the unit as arguments

Effect:

Executes specified unit without performing any initialization; does not return to accessing unit unless used as part of response processing.

Comments:

Conditional form x specification permits execution of instruction following –goto–. Conditional form q specification halts unit execution.

EXIT

Instruction format:

exit	{**expr**}

Tag definition:

expr	Mathematical expression

Effect:

Blank tag terminates auxiliary unit structure. Numeric tag backs out specified number of levels.

NEXTNOW

Instruction format:

nextnow **name**

nextnow **expr,name1,name2,...**

Tag definition:

name	Unit name
expr	Mathematical expression
name1, name2,...	Unit names, x, or q

Effect:

Disables all terminal keys except the NEXT key. When NEXT is pressed, the unit specified in the tag is entered. Stops execution of remainder of unit.

Comments:

Initialization of the new unit is the same as if the current unit were completed.

IFERROR

Instruction format:

iferror **name**

iferror **expr,name1,name2,...**

Tag definition:

name	Unit name
expr	Mathematical expression
name1, name2,...	Unit names, x, or q

Effect:

Specifies unit to –goto– if an error occurs in a –calc– instruction.

Comments:

Marker is cleared by a tag of q or when a new main unit is entered.

ENTRY

Instruction format:

entry **name**

Tag definition:

name	Unique name

Effect:

Supplies alternate entry point to a unit.

Comments:

name can be no longer than eight characters.

FINISH

Instruction format:

finish **name**

finish **expr,name1,name2,...**

Tag definition:

name	Unit name
expr	Mathematical expression
name1, name2,...	Unit names, x, or q

Effect:

Specifies a unit to be executed when student presses the SHIFT STOP keys.

Comments:

This instruction does not execute if lesson is ended by an –end lesson– or a –jumpout–. Some instructions cannot be included in a –finish– unit. Only one –finish– instruction is necessary (in IEU). Marker is held past unit boundaries.

TIMEL

Instruction format:

timel

timel **expr,name**

Tag definition:

expr	Mathematical expression (in seconds)
name	Unit name within current lesson

Effect:

Provides a –helpop– type branch to the specified unit when the given time limit has expired. A unit to be branched to with a –timel– instruction is cleared if the student exits from the current lesson or if a –timel– instruction with a blank tag is executed. If the time limit expires while the student is executing instructions, the branch occurs when the student reaches a waiting point (–pause–, –arrow–, or end of the unit).

Comments:

The time limit is given in seconds, and the minimum time limit is 0.75 second. The –timel– instruction is not affected by any –time– instructions.

TIMER

Instruction format:

timer

timer **expr,name**

Tag definition:

expr	Mathematical expression (in seconds)
name	Name of unit within router lesson

Effect:

Causes a return to the specified unit of the router lesson when the time limit expires. This branch remains in effect until the student signs off the system or executes a –timer– instruction with a blank tag. The branch does not occur if the student is using a system TERM feature (for example, TERM-calc) when the time limit expires. The IEU of the router is not executed when the student returns to the router.

Comments:

The time limit is given in seconds, and the minimum value is 300 seconds (5 minutes). The –timer– instruction cannot be used unless the author is writing his own router.

END

Instruction format:

end {help}

end lesson

Tag definition:

help	Optional tag equivalent to blank tag; both end help sequences
lesson	Logical end of lesson

Effect:

A blank tag or keyword help tag end help sequences. Keyword lesson tag ends lesson execution and sets system-reserved word ldone to -1.

Comments:

An –end– instruction with a blank tag in a base unit has no effect. An –end– instruction in a help sequence delimits the unit as well as ending the sequence.

BACK

Instruction format:

back **name**

back **expr,name1,name2,...**

Tag definition:

name	Unit name
expr	Variable or mathematical expressions
name1, name2,...	Unit names, x, or q

Effect:

Initiates a new main unit (specified by **name**) when the BACK key is pressed.

BACKOP

Instruction format:

backop **name**

backop **expr,name1,name2,...**

Tag definition:

name	Unit name
expr	Variable or mathematical expression
name1, name2,...	Unit names, x, or q

Effect:

Initiates a new main unit (specified by **name**) when the BACK key is pressed.

Comments:

There is no full-panel erasure when –backop– is executed. Any graphics or text specified by the –backop– instruction is added to the current display and remains on the panel.

BACK1

Instruction format:

back1 **name**

back1 **expr,name1,name2,...**

Tag definition:

name	Unit name
expr	Variable or mathematical expression
name1, name2,...	Unit names, x, or q

Effect:

Initiates a new main unit (specified by **name**) when the SHIFT BACK keys are pressed.

BACK1OP

Instruction format:

back1op **name**

back1op **expr,name1,name2,...**

Tag definition:

name	Unit name
expr	Variable or mathematical expression
name1, name2,...	Unit names, x, or q

Effect:

Initiates a new main unit (specified by **name**) when the SHIFT BACK keys are pressed.

Comments:

There is no full-panel erasure when –back1op– is executed. Any graphics or text specified by the –back1op– instruction is added to the current display and remains on the panel.

STOP

Instruction format:

stop **name**

stop **expr,name1,name2,...**

Tag definition:

name	Unit name
expr	Variable or mathematical expression
name1, name2,...	Unit names, x, or q

Effect:

Specifies unit which follows the present unit when the STOP key is pressed.

HELP

Instruction format:

help **name**

help **expr,name1,name2,...**

Tag definition:

name	Unit name
expr	Mathematical expression
name1, name2,...	Unit names, x, or q

Effect:

Specifies beginning unit of a help sequence and enables the HELP key for entry to that unit.

HELPOP

Instruction format:

helpop	**name**
helpop	**expr,name1,name2,...**

Tag definition:

name	Unit name
expr	Mathematical expression or variable name
name1, name2,...	Unit names, x, or q

Effect:

Specifies the beginning unit of a help-on-the-same-page sequence and enables the HELP key for entry to that unit.

Comments:

There is no full-panel erasure when –helpop– is executed. Any graphics or text specified by the –helpop– instruction is added to the current display and remains on the panel when control is returned to the main unit following execution of –helpop–.

HELP1

Instruction format:

help1	**name**
help1	**expr,name1,name2,...**

Tag definition:

name	Unit name
expr	Mathematical expression
name1, name2,...	Unit names, x, or q

Effect:

Specifies the beginning unit of a help-on-the-same-page sequence and enables the SHIFT HELP keys for entry to that unit.

HELP1OP

Instruction format:

help1op	**name**
help1op	**expr,name1,name2,...**

Tag definition:

name	Unit name
expr	Mathematical expression or variable name
name1, name2,...	Unit names, x, or q

Effect:

Specifies the beginning unit of a help sequence and enables the SHIFT HELP keys for entry to that unit.

Comments:

There is no full-panel erasure when –help1op– is executed. Any graphics or text specified by the –help1op– instruction is added to the current display and remains on the panel when control is returned to the main unit following execution of –help1op–.

DATA

Instruction format:

data	**name**
data	**expr,name1,name2,...**

Tag definition:

name	Unit name
expr	Mathematical expression
name1, name2,...	Unit names, x, or q

Effect:

Specifies the first unit of a help sequence and enables the DATA key for entry to that unit.

DATAOP

Instruction format:

dataop	**name**
dataop	**expr,name1,name2,...**

Tag definition:

name	Unit name
expr	Mathematical expression or variable name
name1, name2,...	Unit names, x, or q

Effect:

Specifies the first unit of a help-on-the-same-page sequence and enables the DATA key for entry to that unit.

Comments:

There is no full-panel erasure when –dataop– is executed. Any graphics or text specified by the –dataop– instruction is added to the current display and remains on the panel when control is returned to the main unit following execution of –dataop–.

DATA1

Instruction format:

data1	**name**
data1	**expr,name1,name2,...**

Tag definition:

name	Unit name
expr	Mathematical expression
name1, name2,...	Unit names, x, or q

Effect:

Specifies the first unit of a help sequence and enables the SHIFT DATA keys for entry to that unit.

DATA1OP

Instruction format:

data1op	**name**
data1op	**expr,name1,name2,...**

Tag definition:

name	Unit name
expr	Mathematical expression or variable name
name1, name2,...	Unit names, x, or q

Effect:

Specifies the first unit of a help-on-the-same-page sequence and enables the SHIFT DATA keys for entry to that unit.

Comments:

There is no full-panel erasure when –data1op– is executed. Any graphics or text specified by the –data1op– instruction is added to the current display and remains on the panel when control is returned to the main unit following execution of –data1op–.

LAB

Instruction format:

lab	**name**
lab	**expr,name1,name2,...**

Tag definition:

name	Unit name
expr	Mathematical expression
name1, name2,...	Unit names, x, or q

Effect:

Specifies the first unit of a help sequence and enables the LAB key for entry to that unit.

LABOP

Instruction format:

labop **name**

labop **expr,name1,name2,...**

Tag definition:

name	Unit name
expr	Mathematical expression or variable name
name1, name2,...	Unit names, x, or q

Effect:

Specifies the first unit of a help-on-the-same-page sequence and enables the LAB key for entry to that unit.

Comments:

There is no full-panel erasure when –labop– is executed. Any graphics or text specified by the –labop– instruction is added to the current display and remains on the panel when control is returned to the main unit following execution of –labop–.

LAB1

Instruction format:

lab1 **name**

lab1 **expr,name1,name2,...**

Tag definition:

name	Unit name
expr	Mathematical expression
name1, name2,...	Unit names, x, or q

Effect:

Specifies the first unit of a help sequence and enables the SHIFT LAB keys for entry to that unit.

LAB1OP

Instruction format:

lab1op **name**

lab1op **expr,name1,name2,...**

Tag definition:

name	Unit name
expr	Mathematical expression or variable name
name1, name2,...	Unit names, x, or q

Effect:

Specifies the first unit of a help-on-the-same-page sequence and enables the SHIFT LAB keys for entry to that unit.

Comments:

There is no full-panel erasure when –lab1op– is executed. Any graphics or text specified by the –lab1op– instruction is added to the current display and remains on the panel when control is returned to the main unit following execution of –lab1op–.

BASE

Instruction format:

base

base **name**

base **expr,name1,name2,...**

Tag definition:

name	Unit name
expr	Mathematical expression
name1, name2,...	Unit names, x, or q

Effect:

Specifies a base unit.

Comments:

Blank tag or q clears base unit marker.

TERM

Instruction format:

term

term **name**

Tag definition:

name Author term (8-character maximum)

Effect:

When student presses TERM and types **name,** the unit containing the –term **name**– instruction is entered as the initial unit in a help sequence. Units containing –term–s can be accessed from any part of the lesson. The –term– instruction with a blank tag matches any term entered that is not matched by a specified author term.

Comment:

More than one –term– instruction can be used in a single unit, but same tags must not occur in more than one unit per lesson.

TERMOP

Instruction format:

termop

termop **name**

Tag definition:

name Author term (8-character maximum)

Effect:

When the student presses TERM and types **name,** the unit containing the –termop– instruction is entered as the initial unit in a –term– sequence; the screen is not erased. Units containing –termop–s can be accessed from any part of the lesson.

Comments:

More than one –termop– instruction can be used in a single unit, but tags must not occur in more than one unit per lesson.

*, c, $$

Instruction format:

*** comment**

c comment

command tag $$comment

Effect:

Allows comments to be placed in source code.

Comments:

An asterisk or the letter c in the first space specifies the entire line as a comment. The double-dollar sign specifies remainder of a line as a comment.

CSTOP

Instruction format:

cstop

Effect:

Causes condensing to stop.

Comments:

Condensing can be restarted later in the lesson.

CSTART

Instruction format:

cstart

Effect:

Causes condensing to be started at current source code line.

Comments:

Only necessary following a –cstop– instruction.

CSTOP*

Instruction format:

cstop*

Effect:

Prevents rest of lesson from being condensed.

Comments:

Any –cstart– instructions following –cstop*– have no effect.

RESTART

Instruction format:

restart

restart **uname**

restart **lname,uname**

Tag definition:

uname	Unit name
lname	Lesson name

Effect:

Specifies lesson and unit in which the student will restart lesson execution if lesson terminated before completion. Sets the system-reserved words rstartl and rstartu.

Comments:

Blank tag specifies current lesson and main unit for restart. One-argument tag specifies unit of current lesson for restart. Two-argument tag specifies unit and lesson for restart.

STATUS

Instruction format:

status **var**

Tag definition:

var	Variable or mathematical expression

Effect:

Sets the system-reserved word lstatus to the value in **var.**

Comments:

Noninteger values in **var** are rounded to integers.

RETURN

Instruction format:

return

Effect:

Ends the current time-slice.

Comments:

Ensures an entire time-slice for execution of following instructions.

PRESS

Instruction format:

press **code** {,**station**}

press **name** {,**station**}

press **expr** {,**station**}

Tag definition:

code	Character code
name	Key name, such as next or a
expr	Variable or mathematical expression
station	Station number

Effect:

Places the specified key in the student's input buffer. Can do only one keypress per second. The optional second argument (**station**) is the station at which the key is to be –press–ed. Two-argument form is executed only if both stations are in the same lesson. System-reserved word zreturn is set to –1 if –press– was executed; it is set to 0 if not executed. A router lesson may –press– a key at any station being used by a routed student.

JUMPOUT

Instruction format:

jumpout

jumpout **lname** {**,uname**}

jumpout **expr;lname** {**,uname**} **;lname1** {**,uname1**} **;.;.;.**

jumpout **lname, (var)**

jumpout **opt**

Tag definition:

lname;
lname1;... Actual lesson names or leslist number (constants, variables, or expressions) in <>, x, or q

uname,
uname1,,, Actual unit names or variable containing unit name in ( )

expr Variable or mathematical expression

var Variable containing a unit name

opt Options

Effect:

Transfers student to lesson and unit specified upon execution. The following optional tags have special meanings.

Tag	Description
resume	Takes the student to the restart lesson and unit, thus allowing the student to continue where he left off
router	Returns the student to the router lesson
return	Returns the student to the first unit of the lesson from which this lesson was entered
return, return	Returns the student to the unit following the unit from which this lesson was entered
q or blank tag or (0)	Returns the student to his router if he has one; returns an author to author mode page
x	–jumpout– has no effect

Comments:

If no unit is specified, execution starts at the beginning of the named lesson. The lesson named can be the lesson currently being executed. The IEU, if any, of the lesson to which the –jumpout– is performed is always executed unless the –jumpout– goes to the same lesson that it is in. If a unit name is specified, the –jumpout– codes of the lesson must match.

INHIBIT

Instruction format:

inhibit {**action**}

Tag definition:

action arrow, erase, dropstor, edit, jumpchk, term, charclear, blanks, from, anserase, dropset

Effect:

Prevents normal actions from being taken.

Comments:

Blank tag or a new main unit cancels all inhibits.

KEYLIST

Instruction format:

keylist **name,list**

Tag definition:

name Name of the keylist (maximum of seven characters)

list Single character and/or previously defined list names

Effect:

Establishes a list of keys to be used in the –pause– and –keytype– instructions.

Comments:

The –keylist– instruction is a nonexecutable instruction and should be placed in the IEU. System-defined lists available are:

numeric	Digits (0, 1, 2, ..., 9)
alpha	Alphabet (a through z and A through Z)
funct	Function keys (HELP, LAB, NEXT, and so on)
touch	Touch panel input (256 < key < 511)
ext	External input (512 < key < 767)

PAUSE

Instruction format:

pause

$$pause until any key is pressed

pause **n**

$$pause for n seconds

pause keys=**k1,k2,...**

$$pause until one of the keys
$$in list is pressed

pause **n**, keys=**k1,k2,...**

$$pause for n seconds or until
$$one of the keys in the list is
$$pressed

Tag definition:

n	Number of seconds of pause (n $\geq$ 0.75)
k1,k2,...	Key names

Effect:

Halts lesson execution until condition specified by tag is satisfied.

Comments:

An entry in the key list may be a group name (defined by –keylist– instruction) or one of the following system-defined groups.

numeric	Digits (0,1,2,...,9)
alpha	Alphabet (a through z and A through Z)
funct	Function keys (HELP, LAB, NEXT, and so on)
touch	Touch panel input (256 < key < 511)
ext	External input (512 < key < 767)

COLLECT

Instruction format:

collect **var,num**

Tag definition:

var	Beginning user variable (n) where key information is to be stored
num	Number of keys to accept

Effect:

Collects keys from external sources. Lesson execution is halted until the condition specified by **num** is satisfied or time expires as a result of a previous –time– command. The keys are stored one per variable.

Comments:

An –enable– instruction must precede –collect– to receive touch or external input. The upper limit is 20 keys.

KEYTYPE

Instruction format:

keytype **var,list**

Tag definition:

var	Variable where result is to be stored
list	Single character, previously defined group names, external input, variable, and/or touch panel input

Effect:

The –keytype– instruction compares the key pressed by the student to the entries in **list.** If the key is not found in **list, var** is set to –1. If the key is found, **var** is set to 0 when the first entry in **list** is matched. **var** is set to 1 if the second entry in **list** is matched, and so on. Possible arguments for **list** are as follows:

Single keys: a, b, c, $, %, ?, =

System-defined key names: next, back, data1

Your own defined groups: mygroup, w, d, x, a

System-defined groups: funct, alpha, touch

ext(nx): value of external key in nx

(nx): value in nx

t(finex,finey): touch input, fine address

touch(coarse address): touch input, coarse address

touch(finex,finey,xtol,ytol): tolerance in dots

t(coarse;xchars,ychars): tolerance in characters

Comments:

If a –keylist– name is used, the –keytype– **var** is set to the same value when any one of the –keylist– arguments is matched.

FORCE

Instruction format:

force {**oplist**}

Tag definition

oplist One or more of the character strings font, micro, long, clear, left, and first-erase (separated by commas).

Effect:

Forces specified action to be taken.

Comments:

Keyword font forces use of alternate characters, micro forces use of micro table, long forces judging initiation when response length limit is reached, clear or a blank tag clears previous –force– setting, left forces response to appear right to left, and firsterase forces an ERASE key after a no judgment so the student can enter a new response without pressing NEXT or ERASE first.

CHANGE

Instruction format:

change command **cname** to **nname**

change symbol **character** to **ncharacter**

Tag definition:

cname Instruction command (for example, at, end, write)

nname New instruction command to be substituted

character A single character

ncharacter New character to be substituted

Effect:

The –change command– instruction causes instances of the new name to condense as if it were the old name. The –change symbol– instruction changes the values of characters in the judging state.

Comments:

Both forms of the change instruction must be in the IEU and both produce lesson-wide changes. The –change symbol– instruction is processed in linear order. It is used only in instructions in the judging state.

USE

Instruction format:

use **bname**

Tag definition:

bname Block name

Effect:

Inserts code from specified block into current lesson at condense time.

Comments:

The lesson to be accessed with the –use– instruction is specified in the Author Information page of the lesson containing the –use– instruction. Only one lesson can be used per lesson; a block containing a –use– cannot be used. Use codes of the lesson must match.

STEP

Instruction format:

step on

step off

step {**expr**}

Tag definition:

on Keyword on must be used in the –step on– instruction and turns on the step feature

off Keyword off must be used in the –step off– instruction and turns off the step feature

Effect:

Enables author to step-through his lesson, instruction by instruction. When lesson is stepped through, certain information is displayed on the lower lines of the panel. Next instruction to be executed is shown (press NEXT key to execute), and base, main, and current units are listed. Student variables may be examined at any time.

Comments:

When executing the lesson, step mode can also be entered by pressing the TERM key and typing step. The –step– mode is not operable for students. Common variables or storage cannot be inspected. Pressing the BACK key exits from step mode. User editing security code must match lesson change code to enter step mode.

IN

Instruction format:

in	**stanum**

Tag definition:

stanum	Station number in the range of 0 to 1023

Effect:

Indicates if a station is using the current lesson by setting system-reserved word zreturn to –1.

Comments:

Current station number is obtained with system-reserved word station.

INITIAL

Instruction format:

initial	lesson,**lessu**
initial	common,**commu**

Tag definition:

lesson	Keyword lesson must be used in the –initial lesson– instruction
lessu	Name of unit to be executed when –initial lesson– is encountered by first user
common	Keyword common must be used in the –initial common– instruction
commu	Name of unit to be executed when –initial common– is encountered by first user

Effect:

Specifies a unit which is executed when a lesson or common is brought into ECS and encountered by first user.

Comments:

If the lesson or common is already in ECS, the initial unit is not executed. The initial unit is inserted (like a –do–) at the location of the –initial– instruction.

LESSON

Instruction format:

lesson	completed
lesson	incomplete
lesson	no end
lesson	**expr**,completed,incomplete,...

Tag definition:

completed	Lesson completed
incomplete	Lesson not completed
no end	Lesson has no logical end
expr	Mathematical expression

Effect:

Assigns a value to system-reserved word ldone. Keyword completed sets ldone to –1, keyword incomplete sets ldone to 0, and keyword no end sets ldone to 1.

Comments:

The system router uses the value of ldone. If ldone equals –1, an asterisk is placed next to the lesson on the student's index (sequence). Upon entering a lesson, the system router checks if the lesson has been completed, and if so, sets ldone to –1. If not, ldone is set to 0.

SCORE

Instruction format:

score	
score	**val**

Tag definition:

val	Value to be placed into lscore

Effect:

Assigns a value (constant or any expression from –1 to 100) to the system-reserved word lscore. Value of lscore can then be stored in any data base or status bank (student, common, router, etc.) for the student.

Comments:

The –score– instruction with no tag assigns a value of –1 to lscore. Values are rounded to the nearest integer. Negative score is interpreted as "do not store any score". Score which rounds to a value greater than 100 produces an execution error.

BACKGND

Instruction format:

backgnd

Effect:

Allows the user more processing time during each time-slice if the system is not busy.

Comments:

If the system is busy, the lesson receives less than the average processing time. The –backgnd– instruction should not be used with lessons which are used by the students in an instructional setting. The –foregnd– instruction cancels the effect of –backgnd–.

FOREGND

Instruction format:

foregnd

Effect:

Cancels the effect of the –backgnd– instruction. Normal execution state is restored.

Comments:

When used, –foregnd– normally follows a –backgnd– instruction.

CPULIM

Instruction format:

cpulim **var**

Tag definition:

var	Variable or mathematical expression specifying time limit in thousand instructions per second (TIPS) (maximum of 10)

Effect:

Allows author to place limit on CPU time for a given lesson while that lesson is being used by students. CPU time in TIPS is listed in the student records and at sign-off time.

Comments:

The value for CPU TIPS remains in effect until sign-off time. A maximum limit on the CPU time allows the lesson author to test a lesson at a low CPU time maximum and decide if there is any effect on a lesson. If –cpulim– is set to a negative or zero time, an execution error occurs.

ROUTE

Instruction format:

route end lesson,**endlesu**

route finish,**finishu**

route error,**erroru**

route resignon {, **resignu**}

Tag definition:

end lesson	Keyword end lesson must be used in the –route end lesson– instruction and indicates an end-of-lesson exit from the instructional lesson
endlesu	Entry unit which is executed when the student leaves the instructional lesson because of an –end lesson– or –jumpout q– instruction
finish	Keyword finish must be used in the –route finish– instruction and indicates a finished exit from the instructional lesson
finishu	Unit to be executed when a student leaves the instructional lesson by pressing the SHIFT STOP keys
error	Keyword error must be used in the –route error– instruction and indicates an execution-error exit from the instructional lesson
erroru	Unit which is executed when an execution error occurs in the instructional lesson
resignon	Keyword resignon must be used in the –route resignon– instruction and gives the options of signing off or of returning to the router
resignu	Router unit to which student is returned (optional)

Effect:

Specifies which units of the router are to act as reentry units when the student exits from an instructional lesson in the router.

Comments:

The –route– instruction must be executed each time a student is in the router in order for the specified units to be functional. Placing the –route– instruction in the router's index or decision unit sets these flags each time a lesson is selected by or for the student. The IEU of the router is only executed at sign-on time or when –route resignon– without a specified unit is executed.

ROUTVAR

Instruction format:

routvar **num**

Tag definition:

num Mathematical expression specifying the number of router student variables (maximum of 50); 50 locations are referenced by nr or vr

Effect:

Specifies the number of student router variables to be made part of the permanent student bank and retained between sessions in addition to the 150 student variables. Fifty router student variables can be added to the amount of ECS used at a logical site.

Comments:

Student router variables can be altered only in the router. With an –allow read rvars– instruction in effect, the values can then be read in an instructional lesson.

ALLOW

allow

allow read

allow write

allow read rvars

Tag definition:

read Keyword read must be used in the –allow read– instruction and specifies read-only access to the ECS router common

write Keyword write must be used in the –allow write– instruction and specifies read and write access to the ECS router common

read rvars Keyword read rvars must be used in the –allow read rvars– instruction and specifies read-only access to the student router variables

Effect:

Permits instructional lessons to reference the router's common variables. The –allow– instruction is only meaningful when executed by students in a course using that router and in the router itself (not instructional lessons).

Comments:

An –allow– instruction with no tag clears –allow– settings. A –transfr– instruction must be used following the –allow– to read and write the ECS router common or to read router student variables. (Refer to the –transfr– instruction.)

LESLIST

Instruction format:

leslist {**lesson,**} **block**

Tag definition:

lesson Lesson containing the lesson list; optional if **lesson** is same lesson containing –leslist– instruction

block Actual block name or variable containing block name

Effect:

Allows access to the specified leslist. A leslist is a special block used to maintain a list of lesson names.

Comments:

Only one leslist may be used at a time, and the –leslist– instruction must be executed before any references are made to the leslist. The –leslist– instruction may be used with the –jumpout–, –from–, –restart–, and –lessin– instructions. A leslist can be altered with the –addlst– and –removl– instructions. After a –leslist– instruction, zreturn has the following values.

-1 leslist obtained successfully

0 leslist not found

1 Code word error

2 Already a leslist in use

ADDLST

Instruction format:

addlst **var** {,**position**}

Tag definition:

var First of a set of three consecutive variables specifying a valid lesson name

position Variable or mathematical expression specifying the list position the lesson should occupy

Effect:

One-argument form adds lesson names to a leslist (next open slot on the list). In the two-argument form, **position** specifies the leslist slot which the name is to occupy.

Comments:

An –addlst– instruction with the actual lesson named in the tag (not a variable) produces errors. Additions to a leslist should be done by storing the information with a –storea– using a character count of 30. A –storea– instruction must precede the –addlst– instruction. After execution of the –addlst– instruction, system-reserved word zreturn is set as follows:

-1 Lesson name added successfully

0 No leslist specified

1 Illegal lesson name

2 Lesson name already in leslist

3 leslist is full

4 Specified position is in use

REMOVL

Instruction format:

removl **lesson**

Tag definition:

lesson Number, mathematical expression, or variable specifying the lesson to be removed from the list

Effect:

Deletes lessons from a list, with a blank entry left in that list position.

Comments:

After execution of the –removl– instruction, system-reserved word zreturn is set as follows:

-1 Lesson name deleted successfully

0 No leslist specified

LNAME

Instruction format:

lname **var,position**

Tag definition:

var First of three consecutive variables required to place the leslist information

position Constant, variable, or mathematical expression specifying the lesson number in the leslist

Effect:

Places lesson name at **position** into the three consecutive variables beginning with **var.** leslist information can be displayed with a –showa– instruction using a character count of 30.

Comments:

After execution of the –lname– instruction, system-reserved word zreturn is set as follows:

-1 Lesson information transferred successfully

0 No leslist specified

FINDL

Instruction format:

findl **initial,return**

Tag definition:

initial Initial variable for a set of three consecutive variables specifying the lesson name

return Return variable containing the position of the named lesson in the –leslist–

Effect:

Determines if a specified lesson name is included in a leslist. If the specified lesson is not found or a leslist is not specified, the value of the return variable is set to –1.

RESPONSE HANDLING INSTRUCTIONS

The following instructions are used to process student responses.

arrow	list	storen
endarrow	concept	storeu
iarrow	miscon	ntouch
arheada	vocab	ntouchw
arrowa	vocabs	touch
iarrowa	endings	touchw
long	ok	match
jkey	no	or
storea	okword	ans
open	noword	compare
loada	ignore	specs
close	exact	markup
bump	exactc	judge
put	exactv	getword
putd	ansv	getmark
putv	ansu	getloc
answer	wrongv	edit
answerc	wrongu	copy
wrong	store	time
wrongc		

ARROW

Instruction format:

arrow **loc**

arrow **finex,finey**

Tag definition:

loc	Coarse-grid location of arrow
finex	Horizontal fine-grid location of arrow
finey	Vertical fine-grid location of arrow

Effect:

Places arrowhead on student screen at specified point, indicating a desire for student response. Serves to delimit response handling portion of the unit.

Comments:

Sets some default parameters.

ENDARROW

Instruction format:

endarrow

Effect:

Ends instructions that process response to preceding arrow.

IARROW

Instruction format:

iarrow

iarrow **name**

iarrow **expr,name1,name2,...**

Tag definition:

name	Unit name
expr	Mathematical expression
name1, name2,...	Unit names, x, or q

Effect:

Inserts named unit with a –join– after the first –arrow– of a unit and just before the first judging command for that –arrow–.

Comments:

The –iarrow– instruction with no tags (or q) clears previous settings.

ARHEADA

Instruction format:

arheada **char**

Tag definition:

char	Any character

Effect:

Specifies a character to act as an alternate arrow.

Comments:

The alternate arrow is limited to five character codes. The –arheada– instruction is used with the –arrowa– and –iarrowa– instructions.

ARROWA

Instruction format:

arrowa **loc**

arrowa **finex,finey**

Tag definition:

loc Coarse-grid location of alternate arrow

finex Horizontal fine-grid location of alternate arrow

finey Vertical fine-grid location of alternate arrow

Effect:

Identical in operation to the –arrow– instruction, except that the alternate character previously specified in an –arheada– instruction is used instead of the regular arrowhead.

IARROWA

Instruction format:

iarrowa **name**

Tag definition:

name Unit name

Effect:

Operates analogously to the –iarrow– instruction.

LONG

Instruction format:

long **num**

Tag definition:

num Number of characters in maximum student response

Effect:

Specifies a length limit for student response.

Comments:

A –long 1– instruction initiates judging after the first keypress. Default response length (with no –long– instruction) is 150 characters.

JKEY

Instruction format:

jkey **names**

Tag definition:

names One or more function key names separated by commas (for example, help1, back)

Effect:

Specifies function keys besides the NEXT key that will initiate judging. Cannot specify SHIFT STOP.

STOREA Judging Instruction

Instruction format:

storea **var** {,**cnt**}

storea **var,var1**⇐jcount

Tag definition:

var Variable in which student response is stored

cnt Character count (default is 10)

var1 Variable in which character count is stored

jcount System-reserved word jcount

Effect:

Stores number of characters specified of the student response, starting in indicated variable. Never ends judging.

OPEN Judging Instruction

Instruction format:

open **var**

Tag definition:

var Starting variable for storage

Effect:

Places student response in variable beginning with the variable specified. One character, right-justified, with zero-fill, per variable. Never ends judging.

LOADA Judging Instruction

Instruction format:

loada **var,cnt**

Tag definition:

var Starting variable of character string

cnt Character count

Effect:

Replaces the judging copy of the student response with the specified character string.

Comment:

Assumes the string is left-justified in the variable, packed 10 character codes per word. Never ends judging.

CLOSE Judging Instruction

Instruction format:

close **var,cnt**

Tag definition:

var Starting variable

cnt Character count

Effect:

Takes rightmost character code of number of variables specified in **cnt**, beginning with the specified variable, and uses the resulting string to replace the judging copy.

Comments:

Reverse of the –open– instruction. Never ends judging.

BUMP Judging Instruction

Instruction format:

bump **char**

Tag definition:

char Characters (up to eight) to be removed

Effect:

Removes indicated characters from the judging copy. Never ends judging.

PUT Judging Instruction

Instruction format:

put **str1=str2**

Tag definition:

str1 Character string (not including an equal sign) to be replaced

str2 Character string replacing **str1**

Effect:

Replaces all occurrences in the student response of the string on the left with the string on the right. Never ends judging.

PUTD Judging Instruction

Instruction format:

putd **char str1 char str2 char**

Tag definition:

char Any character not appearing in **str1** or **str2**

str1 String to be replaced

str2 String replacing **str1**

Effect:

Replaces all occurrences of first character string in the student response with the second character string.

Comments:

Allows the specification for replacement of strings containing an equal sign. Never ends judging.

PUTV Judging Instruction

Instruction format:

putv **beg,lengthchar,initial,lengthrep1**

Tag definition:

beg Beginning location of character string (left-justified)

lengthchar Length of character string

initial Variable specifying initial location of replacement string (left-justified)

lengthrep1 Length of replacement string (constant or variable)

Effect:

Replaces all occurrences of character string contained in location **beg** of length **lengthchar** with replacement string contained in location **initial** of length **lengthrep1.** Never ends judging.

ANSWER Judging Instruction

Instruction format:

answer **{<optwords> (synwords)} resp**

Tag definition:

optwords Optional words that are allowed in the student response

synwords Synonymous words that are required in the student response

resp Answer to be matched

Effect:

Judges the student response ok if it adequately matches the tag.

Comments:

Unless inhibited by the –specs– instruction, some marking of the answer may be done. Tag cannot contain punctuation; however, the student may use punctuation in his response. Cannot match any student response which contains more than 39 words.

ANSWERC Judging Instruction

Instruction format:

answerc **expr;resp1;resp2;...respn**

Tag definition:

expr Variable or mathematical expression

resp1, resp2,... respn Answers to be matched; may contain optional words or synonymous words

Effect:

Based on the value of **expr,** judges the student response ok if the response adequately matches that portion of the tag. For example, if **expr** is 0, the –answerc– instruction has the same effect as the instruction –answer **resp2**–.

Comments:

The –answerc– instruction is the conditional form of the –answer– instruction and is designed for use in drill-and-practice lessons. Optional words are enclosed in angular brackets. Synonymous words are separated by commas and are enclosed in parentheses. Maximum number of tags is approximately 500.

WRONG Judging Instruction

Instruction format:

wrong **{<optwords> (synwords)} resp**

Tag definition:

optwords Optional words that are allowed in the student response

synwords Synonymous words that are required in the student response

resp Answer to be matched

Effect:

Judges the student response no if it adequately matches the tag.

Comments:

Identical in operation to the –answer– instruction, except that a matched response is judged no instead of ok.

WRONGC Judging Instruction

Instruction format:

wrongc	**expr;resp1;resp2;...respn**

Tag definition:

expr	Variable or mathematical expression
resp1, **resp2,...** **respn**	Responses to be matched; may contain optional words or synonymous words

Effect:

Based on the value of **expr**, judges the student response no if the response adequately matches that portion of the tag. For example, if **expr** is 0, the –wrongc– instruction has the same effect as the instruction –wrong **resp2**–.

Comments:

The –wrongc– instruction is the conditional form of the –wrong– instruction. The –wrongc– instruction is identical in operation to the –answerc– instruction except that a matched response is judged no instead of ok.

LIST

Instruction format:

list	**name,wlist**

Tag definition:

name	Name of the list (maximum of seven characters)
wlist	List of associated words with words separated by commas

Effect:

Specifies list of equivalent words for synonyms or optional words in –answer– and –wrong– instructions.

Comments:

The list is referenced by name. When specifying synonymous or ignorable words, the markers setting off the words are double for the list (that is, <<**name**>> for ignorable words and ((**name**)) for synonyms).

CONCEPT Judging Instruction

Instruction format:

concept	**sen1** **sen2** . . .

Tag definition:

sen1, **sen2,...**	Equivalent concepts

Effect:

Judges student response ok if it matches one of the concepts specified.

Comments:

Words used must occur in a previous –vocab– or –vocabs– instruction. The –concept– instruction is used for evaluation of complex responses or any of a number of divergent responses. Judging is based on word order only. Cannot match any student response which contains more than 39 words.

MISCON Judging Instruction

Instruction format:

miscon	**sen1** **sen2** . . .

Tag definition:

sen1, **sen2,...**	Equivalent concepts

Effect:

The –miscon– instruction is identical in operation to the –concept– instruction except that the student response is judged no if the response matches one of the specified concepts.

VOCAB

Instruction format:

vocab **name**
<opt words>
(syn1)
(syn2)
.
.
.
words
words with suffixes

Tag definition:

name	Name of –vocab– (maximum of seven characters)
opt words	List of ignorable words
syn1, syn2,...	List of synonymous words
words	Nonignorable words without synonyms
words with suffixes	Keyword/number or keyword/suffix

Effect:

Constructs a list of useable and ignorable words for use with –concept– instructions that follow –vocab–.

Comments:

More than one –vocab– can be used, but only one is in use at any time. A previously defined but superseded –vocab– can be brought into use by a –vocab– instruction with only the name in the tag.

VOCABS

Instruction format:

vocabs **name**
<opt words>
(syn1)
(syn2)
.
.
.
words
phrases
words with suffixes

Tag definition:

name	Name of –vocabs– (maximum of seven characters)
opt words	List of ignorable words
syn1, syn2,...	List of synonymous words
words	Nonignorable words without synonyms
phrases	Keywords separated by *
words with suffixes	Keyword/number or keyword/suffix

Effect:

Constructs a list of useable and ignorable words, with spelling and capitalization checks, for use with –concept– instructions that follow –vocabs–.

Comments:

More than one –vocabs– can be used, but only one is in use at any time. A previously defined but superseded –vocabs– can be brought into use by a –vocabs– instruction with only the name in the tag.

ENDINGS

Instruction format:

endings **num,list**

Tag definition:

num	Number (0 through 9) identifying suffixes list
list	Actual list of up to eight suffixes

Effect:

The –endings– instruction must precede the –vocab– or –vocabs– instruction. Adds suffixes to words defined in vocabulary (–vocab– or –vocabs–).

Comments:

The notation rootword/number in a vocabulary (–vocab– or –vocabs–) defines as synonymous the root word and all associated words formed in the suffix list identified by **num** in the –endings– instruction. However, the notation rootword//number does not include the root word as one of the synonyms. Root words can be no longer than 30 characters, and suffixes can be no longer than 10 characters.

OK Judging Instruction

Instruction format:

ok

Effect:

Judges student response ok (correct). Always ends judging.

NO Judging Instruction

Instruction format:

no

Effect:

Judges student response no (incorrect). Always ends judging.

OKWORD

Instruction format:

okword **message**

Tag definition:

message Replacement message (maximum of nine characters), including shift and font codes)

Effect:

Permits the changing of the standard ok message. The –okword– instruction can be inserted after the first judging instruction and a different –okword– after another judging instruction for the same –arrow– instruction. The –okword– instruction remains in effect until changed.

Comments:

The –okword– instruction with a blank tag inhibits the ok message, but not the no message. A space code is automatically provided before any specified message.

NOWORD

Instruction format:

noword **message**

Tag definition:

message Replacement message (maximum of nine characters, including shift and font codes)

Effect:

Permits the changing of the standard no message. The –noword– instruction can be inserted after the first judging instruction and a different –noword– after another judging instruction for the same –arrow– instruction. The –noword– instruction remains in effect until changed.

Comments:

The –noword– instruction with a blank tag inhibits the no message. A space code is automatically provided before any specified message.

IGNORE Judging Instructon

Instruction format:

ignore

Effect:

Erases student response and causes wait for new student response.

Comments:

Regular instructions following an –ignore– instruction are not executed. Always ends judging.

EXACT Judging Instruction

Instruction format:

exact **resp**

Tag definition:

resp Answer to be matched

Effect:

Judges student response ok if it exactly matches the tag.

Comments:

Exact match includes spelling, punctuation spaces, and so on.

EXACTC Judging Instruction

Instruction format:

exactc **expr,resp1,resp2,...**

Tag definition:

expr Mathematical expression

resp1, resp2,... Possible answers to be matched

Effect:

Judges student response ok if it exactly matches the answer selected by the expression.

Comments:

Conditional form of the –exact– instruction.

EXACTV Judging Instruction

Instruction format:

exactv **start {,num }**

Tag definition:

start Starting variable of string

num Number of characters (optional)

Effect:

Judges the student response ok if it exactly matches the characters specified in the variables.

Comments:

Can be used as a conditional –exact– . If **num** argument is omitted, comparison ends after the tenth character or at the first occurrence of a 0 character (six bits of zero). Segmented variables cannot be used.

ANSV Judging Instruction

Instruction format:

ansv **val {,tol }**

Tag definition:

val Value to be matched

tol Tolerance permitted

Effect:

Judges algebraic student response ok if value of response is equal to the specified value, within tolerance.

Comments:

One-argument form specifies no tolerance allowed. Tolerance can be numeric or percentage. Value to be matched can be an expression.

ANSU Judging Instruction

Instruction format:

ansu **ans {,tol }**

Tag definition:

ans Exact answer required or mathematical expression

tol Tolerance permitted

Effect:

Judges numeric student responses with scientific units ok if answer is equal to the specified answer (**ans**), within specified tolerance.

Comments:

If argument **ans** is a mathematical expression and argument **tol** is absent, the exact answer must be given for an ok judgment. To store the numeric and dimensional parts of the response, the –storeu– instruction should precede –ansu–. The –ansu– tags must be previously defined in a –define student– instruction. Tolerance can be numeric or percentage.

WRONGV Judging Instruction

Instruction format:

wrongv **val {,tol }**

Tag definition:

val Value to be matched

tol Tolerance permitted

Effect:

Judges algebraic student response no if the value of the response is equal to the specified value, within specified tolerance.

Comments:

Operates like the –ansv– instruction but judges no instead of ok.

WRONGU Judging Instruction

Instruction format:

wrongu **ans {,tol}**

Tag definition:

ans Exact answer required or mathematical expression

tol Tolerance permitted

Effect:

Judges numeric student responses with scientific units no if answer is equal to the specified answer (**ans**), within specified tolerance.

Comments:

Operates like the –ansu– instruction but judges no instead of ok.

STORE Judging Instruction

Instruction format:

store **var**

Tag definition:

var Variable in which value of student response is to be stored

Effect:

Stores value of algebraic student response in **var.** The response is judged no and judging ended only if the student response cannot be evaluated.

STOREN Judging Instruction

Instruction format:

storen **var**

Tag definition:

var Variable in which value is to be stored

Effect:

Searches student response for simple numeric element, and if found, evaluates the element and stores the value in **var.** The numeric element is removed from the judging copy.

Comments:

If a simple numeric element is not found, judging is ended with a judgment of no. Variable names are not permitted in the numeric element.

STOREU Judging Instruction

Instruction format:

storeu **var,array**

Tag definition:

var Variable in which numeric part of student response is to be stored

array Ten consecutive variables in which dimensional part of the student response is to be stored. Must be v variable and not n.

Effect:

Stores the numeric part of the student response in the specified variable (**var**) and stores the dimensional part of the student response in the 10 consecutive v variables (**array**). The response is judged no and judging ends only if the student response cannot be evaluated.

Comments:

If the student response can be evaluated, judging is not ended. The –storeu– tags must be previously defined in a –define student– instruction. The –ansu–, –wrongu–, –ansv–, and –wrongv– instructions should always follow the –storeu– instruction.

NTOUCH Judging Instruction

Instruction format:

ntouch

ntouch **area1;area2;...;arean**

Tag definition:

area1, area2,... arean Touch areas of the forms:

coarse grid

loc {,charx,liney}

loc Coarse-grid location

charx Number of characters wide

liney Number of lines high

fine grid

finex,finey {,dotx,doty}

finex Fine-grid horizontal location

finey Fine-grid vertical location

dotx Number of dots wide

doty Number of dots high

Effect:

Judges a student touch of the touch panel ok if it lies within any of the specified **areas;** otherwise, the system stays in the judging state and searches for another judging instruction. Blank tag judges all touches ok.

Comments:

Touch panel must be activated with a previous –enable– instruction or –pause keys=touch– instruction and must be deactivated with a subsequent –disable– instruction or a full screen erase. If widths or heights are 0 or omitted, the default value is 1. If they are negative, the system ignores that area. Cannot split an **area** designation between lines.

NTOUCHW Judging Instruction

Instruction format:

ntouchw

ntouchw **area1;area2;...;arean**

Tag definition:

area1, area2,... arean Touch areas of the forms:

coarse grid

loc {,charx,liney}

loc Coarse-grid location

charx Number of characters wide

liney Number of lines high

fine grid

finex,finey {,dotx,doty}

finex Fine-grid horizontal location

finey Fine-grid vertical location

dotx Number of dots wide

doty Number of dots high

Effect:

Judges a student touch of the touch panel no if it lies within any of the specified **areas.** Blank tag judges all touches no.

Comments:

The –ntouchw– instruction is identical in operation to the –ntouch– instruction except that a matched response is judged no instead of ok.

TOUCH Judging Instruction

Instruction format:

touch **loc,tol**

touch **loc,width,height**

touch **list**

Tag definition:

loc Coarse-grid location

tol Tolerance (in touch squares)

width Number of touch squares wide

height Number of touch squares high

list List of touch locations in either two- or three-argument form, separated by semicolons.

Effect:

Judges a student touch of the touch panel ok if it lies within the specified area; otherwise, the system stays in the judging state and searches for another judging instruction.

Comments:

loc in the three-argument form is the lower-left corner of the sensitive area. Touch panel must be activated with a previous –enable– instruction. Up to 20 touch panel locations may be specified in the tag of a single –touch– instruction. Judging is ended with an anticipated ok if the screen is touched in any one of the elements specified by **list.** The –ntouch– instruction will replace –touch–.

TOUCHW Judging Instruction

Instruction format:

touchw **loc,tol**

touchw **loc,width,height**

touchw **list**

Tag definition:

loc Coarse-grid location

tol Tolerance (in touch squares)

width Number of touch squares wide

height Number of touch squares high

list List of touch locations in either two- or three-argument form, separated by semicolons

Effect:

Judges a student touch of the touch panel no if it lies within the specified area.

Comments:

The –touchw– instruction is identical in operation to the –touch– instruction except that a matched response is judged no instead of ok. The –ntouchw– instruction will replace –touchw–.

MATCH Judging Instruction

Instruction format:

match **var,list**

Tag definition:

var Variable in which position number is to be stored

list List of possible responses, separated by commas

Effect:

Places position of matched response (0 for first possible response, 1 for second, and so on) in **var.** Places –1 in **var** if no match is found.

Comments:

Always ends judging.

OR Judging Instruction

Instruction format:

or

Effect:

Defines following judging instruction as equivalent to preceding judging instruction.

Comments:

System-reserved word ansent is the same whichever of the two (or more) instructions is matched. Instructions judging ok and no can be specified as equivalent. Never ends judging.

ANS Judging Instruction

Instruction format:

ans

Effect:

Executes regular instructions immediately following if the student presses the ANS key.

Comments:

The –ans– instruction must be the first judging instruction following the –arrow– instruction.

COMPARE Judging Instruction

Instruction format:

compare **word1,word2,return**

Tag definition:

word1 Variable containing first of two words to be compared

word2 Variable containing second word to be compared with first word

return Location where result of word comparison is to be stored

Effect:

Compares **word1** with **word2** and returns the result in **return** as follows:

-1 If they are different words

0 If they are the same word

+n If the words could be misspellings of one another; the smaller the n, the closer the spelling

Comments:

The words in **word1** and **word2** are ended by the first character that ends author language judging.

SPECS Judging Instruction

Instruction format:

specs {**opt list**}

Tag definition:

opt list List of options desired

Effect:

Turns off special standard options or turns on specified nonstandard options. Regular instructions following a –specs– instruction are executed after every response judgment.

Comments:

A list of the available options is given in table 11–2. Never ends judging.

MARKUP

Instruction format:

markup

Effect:

Marks the student's answer with the markup saved by –specs holdmark–.

Comments:

Used only with –specs holdmark–. Has no tag.

JUDGE

Instruction format:

judge **opt**

judge **expr,opt1,opt2,...**

Tag definition:

opt,opt1, opt2,...	ok, no, wrong, exit, continue, rejudge, x, ignore, okquit, noquit, quit
expr	Mathematical expression

Effect:

Specifies an action to be taken regarding the judging process.

Comments:

Executed in regular state only.

GETWORD

Instruction format:

getword **numword,loc,num** {**,maxlength**}

Tag definition:

numword	Ordinal number of word in response
loc	Location where word is to be stored (packed 10 characters per word)
num	Number of characters in word
maxlength	Maximum allowable character length; default is 10 (optional)

Effect:

Seeks a word (specified by **numword**) in a student response; stores the word in a specified location (**loc**); and then stores the number of characters in the word (**num**) in another location.

Comments:

System-reserved word wcount contains the number of words in the student response. Words are defined as strings of characters separated by spaces, punctuation characters, or letter/number boundaries.

GETMARK

Instruction format:

getmark **numword,loc**

Tag definition:

numword	Number of the word from which markup information is desired
loc	Location where return information about desired word is stored

Effect:

The –getmark– instruction can be used after judging a student answer to return information about the words within the student answer. This instruction gives the author the exact information that the system uses to mark up the student answer from the –answer– and –concept– instructions.

Comments:

The value of the word at **loc** is as follows:

-2	No markup possible (for example, no –answer– instructions)
-1	Word out of bounds (for example, **numword** is greater than the number of words in the student response)
0	Perfect word
>0	Various markup information

Various errors set bits in the word at **loc** as follows:

Bit Set	Error
60 (rightmost)	A word is missing before this word
59	Word out of order (move it left)
58	Capitalization incorrect
57	Bad spelling
56	Part of broken phrase
55	Extra word
54	Word missing at end (only for last word)

GETLOC

Instruction format:

getloc **numword,sfinex,sfiney{,efinex,efiney}**

Tag definition:

numword	Number of word whose location is desired
sfinex	Starting x location of word (fine-grid)
sfiney	Starting y location of word (fine-grid)
efinex	Ending x location of word (fine-grid), optional
efiney	Ending y location of word (fine-grid), optional

Effect:

Returns the starting touch panel coordinates of a word in the student's answer, and optionally, the ending coordinates of the word. If the desired word is out of bounds (**numword** is greater than the number of words in the student answer), **sfinex** is set to –1.

Comments:

If the judging copy of the student answer is changed via –put–, –bump–, and so on, the –getloc– instruction attempts to return a best approximation of the touch panel origin of the current word.

EDIT

Instruction format:

edit **loc**

Tag definition:

loc	Starting variable of response buffer

Effect:

Sets up a response buffer where the student can temporarily store his response, returning portions with the EDIT key.

COPY

Instruction format:

copy **loc,len**

Tag definition:

loc	Starting variable of character string
len	Number of characters in string

Effect:

Allows the student to copy the author-specified string into his response.

Comments:

String can only be used once per –arrow– instruction. The string is not destroyed by use.

TIME

Instruction format:

time **num**

Tag definition:

num	Number of seconds specified (must be greater than or equal to 0.75)

Effect:

Requires student to respond within specified time.

Comment:

The value of the system-reserved word key is –timeup– if the student does not respond within the time limit.

STUDENT DATA INSTRUCTIONS

The following instructions are used to collect and access lesson execution data.

dataon	output	readd
dataoff	output1	readr
area	readset	notes
setdat		

DATAON

Instruction format:

dataon

dataon **optlist**

Tag definition:

optlist List of data options to be turned on

Effect:

Starts data collection for current lesson and student when executed.

Comments:

A –dataon– instruction is necessary in every lesson that collects data. The options of the –dataon– instruction can temporarily (that is, for the remainder of the lesson) override individual student data options but cannot turn on course record data options which are turned off. Possible data options are ok, no, unrec no, vocab, area, output, help, help no, term, term no, errors, and signin. Blank tag turns on course record data options.

DATAOFF

Instruction format:

dataoff

dataoff **optlist**

Tag definition:

optlist List of data options to be turned off

Effect:

Stops the collection of data for current student and lesson.

Comments:

Blank tag stops all data collection. Data options cannot override course record data options. Possible data options are ok, no, unrec no, vocab, area, output, help, help no, term, term no, errors, and signin.

AREA

Instruction format:

area

area **name**

area **(expr)**

area incomplete

area cancelled

Tag definition:

name Name of area being entered (maximum of 10 characters); cannot start with a number

(expr) Variable or mathematical expression indicating the area name; expression must be placed in parentheses

incomplete The tag incomplete must be used in the –area incomplete– instruction

cancelled The tag cancelled must be used in the –area cancelled– instruction

Effect:

Delimits previous area for summary data collection and begins new area. –area incomplete– terminates collection of data for current area and marks that area as incomplete. –area cancelled– clears all information in the current area without putting an area summary in the data file, and no data collection is done until a new –area– instruction is encountered.

Comments:

Blank tag delimits previous area, but no new data is collected. Execution of an –area– instruction with the same name as the current area stops data collection until an –area– instruction with a different tag is encountered.

SETDAT

Instruction format:

setdat **resword expr**

Tag definition:

resword One of the following system-reserved words pertaining to areas:

aarea
atime
aarrows
aok
aokist
asno
auno
ahelp
ahelpn
aterm
atermn

expr Mathematical expression or variable

Effect:

Allows alteration of the value of system-reserved words pertaining to areas.

Comments:

Any of the area-reserved words may be set by the –setdat– instruction. They may contain only integers and cannot have a value greater than 511.

OUTPUT

Instruction format:

output **message**

Tag definition:

message Message to be placed in data file

Effect:

Places the tag of the instruction into the datafile, together with overhead information on student, lesson, time, and so on.

Comments:

Variable contents can be included by a form of embedding: ⋖t,var⋗.

t Type (a, n, o, or v)

var Variable name

Tag can be longer than one line, but each line has all associated overhead information written.

OUTPUTL

Instruction format:

outputl {**label,**} **start,num**

Tag definition:

label Instruction label

start Starting variable of block to be placed in datafile

num Number of variables to be stored (limit is 20 variables)

Effect:

Places specified variable contents into datafile.

Comments:

Seven words of overhead information are placed in the datafile unless **label** is absent.

READSET

Instruction format:

readset **fn** {**,access**} {**,var**}

Tag definition:

fn	Datafile name or course file name
access	Access code word of datafile or course file
var	Variable name containing number of students in the course or number of unused records remaining in a datafile

Effect:

Establishes a link between the specified datafile or course file and the lesson so that the lesson can read data from the datafile using –readd– or from the course file using –readr–.

Comments:

Access code word is necessary only if change or inspect code words for the file and the lesson are different. System-reserved word zreturn is returned with the following values.

Value	Meaning
-2	Connection to course file made successfully
-1	Connection to datafile made successfully
0	File does not exist or is not a course or datafile
1	Code words do not match
2	File empty
3	No room in ECS for disk buffer
4	System disk error

READD

Instruction format:

readd area,**var,num**

readd outputl,**var,num**

readd signoff,**var,num**

Tag definition:

area	Keyword area must be used in the –readd area– instruction
var	First of block of variables to receive data
num	Expression giving number of words to transfer
outputl	Keyword outputl must be used in the –readd outputl– instruction
signoff	Keyword signoff must be used in the –readd signoff– instruction

Effect:

Reads appropriate records sequentially from student datafile.

Comments:

A –readset– instruction must be successfully executed before attempting a –readd– instruction. An execution error occurs if an attempt is made to read past the end of the datafile. System-reserved word zreturn is set to –1 if there is more data and to 0 if the end of the file is encountered (no more data).

READR

Instruction format:

readr name,**n**
stats,**start;destin;num** $$One to
svars,**start;destin;num** $$five tags
rvars,**start;destin;num** $$are used.
ldone,**start;destin;num**
lscore,**start;destin;num**

readr sequential
stats,**start;destin;num** $$One to
svars,**start;destin;num** $$five tags
rvars,**start;destin;num** $$are used.
ldone,**start;destin;num**
lscore,**start;destin;num**

readr roster,**start;destin;num**

Tag definition:

name	Keyword name must be used in the –readr name– instruction
n	Two contiguous variables specifying a student name
sequential	Keyword sequential must be used in the –readr sequential– instruction
roster	Keyword roster must be used in the –readr roster– instruction
stats	Student statistics (maximum of 10)
svars	Student variables (maximum of 150)
rvars	Router variables (maximum of 50)
ldone	System-reserved word ldone information (3-bit signed segments)
lscore	System-reserved word lscore information (8-bit signed segments)
start	Starting location in student statistics, student variables, router variables, lesson number in "mrouter", or student roster
destin	Destination into which student statistics, student variables, router variables, ldone or lscore information, or student roster is read
num	Number of variables (stats, svars, or rvars) or student names to be read

Effect:

The –readr name– instruction reads specified student information (student statistics, student variables, router variables, or ldone or lscore information from "mrouter" lessons) into the work space (student variables or common) for inspection.

The first –readr sequential– instruction in the lesson reads the first student's course file (student statistics, student variables, router variables, or ldone or lscore information from "mrouter" lessons) from the roster into the work space (student variables or common) for inspection. Each –readr sequential– reads the next student's information until the roster is exhausted.

The –readr roster– instruction reads a list of names from the student roster into the work space (student variables or common) for inspection.

Comments:

The –readset– instruction must precede the –readr– instruction. The tag for –readset– must specify the course to be read. The ldone and lscore information is available only when "mrouter" is in use.

NOTES

Instruction format:

notes

notes **var,len** {,send}

Tag definition:

var	Student variables (n or v) containing heading information
len	Word length of heading information
send	Keyword send sends note immediately

Effect:

Blank tag initiates TERM-comments for the student. Two-argument tag allows the student to write a comment and title it. The system heads the note with the information stored in **var.** Three-argument tag sends a note consisting of the information in **var** and does not allow the student to write a comment or to title it.

Comments:

System sends notes to the student note file named on the course information page or to the lesson note file named on the lesson information page.

RESOURCE MANAGEMENT INSTRUCTIONS

The following instructions are used to manage terminals and ECS.

site
station

SITE

Instruction format:

site	set,**name**
site	info,**return**
site	active,**start,return,num**
site	stations,**start,return,num**

Tag definition:

name	Variable containing name of logical site or name in quotes
return	Variable in which information is stored
start	First station number
num	Number of stations

Effect:

The results of the tags are listed.

set	Enables other –site– instructions for site **name**
info	Obtains current site ECS information for site **name**
active	Finds active station numbers for site **name**
stations	Finds station numbers permanently in site **name**

Comments:

Used by site lessons.

STATION

Instruction format:

station	info,**statnum,return**
station	status,**statnum**
station	send,**statnum,loc,text,len**
station	stop1,**statnum**
station	logout,**statnum**
station	off,**statnum**
station	on,**statnum**

Tag definition:

statnum	Number of physical station or variable containing number
return	Variable in which information is stored
loc	Coarse-grid coordinates
text	Message to be sent
len	Length of message in characters

Effect:

The results of the tags are listed.

info	Obtains information on station **statnum**
status	Returns current status of station **statnum** in zreturn
send	Sends a message in mode rewrite to station **statnum**
stop1	Presses SHIFT STOP keys on station **statnum**
logout	Signs off station **statnum**
off	Turns off station **statnum**
on	Turns on station **statnum**

Comments:

Used by site lessons. Requires a previous –site set– instruction.

PRINTING INSTRUCTION

The following instruction is used to specify the format of the printout of a lesson.

*list

*LIST

Instruction format:

*list	label,**string**
*list	title,**string**
*list	eject
*list	text
*list	ignore
*list	info
*list	symbols
*list	commands,**list**
*list	off {,**blocks**}
*list	parts
*list	charset {,**(db)**}
*list	leslist
*list	micro
*list	vocabs
*list	mods
*list	deleted
*list	common,**comname,words,format**

Tag definition:

string	Alphanumeric strings used for headings
list	List of commands
blocks	List of blocks
(db)	Characters used to mark dots and background in a charset
comname	Name of common
words	Number of words of common
format	Code for printing datasets and common blocks

Effect:

The results of the tags are listed.

label	Puts specified **string** in printout
title	Prints subheading **string** for a page
eject	Performs a page eject
text	Prints only tags of –write– and –writec– instructions
ignore	Ignores subsequent –*list– instructions
info	Prints lesson information page
symbols	Prints a cross-reference table of symbols
commands	Prints a table listing where instructions specified in **list** are used
off	Prevents printing of named **blocks** or stops source printing if no **blocks** are named
parts	Prints only condensed blocks
charset	Prints any charset in the lesson
leslist	Prints any –leslist– block in the lesson
micro	Prints any –micro– block in the lesson
vocabs	Prints any –vocab– block in the lesson
mods	Prints mod words
deleted	Prints deleted lines
common	Prints a common block or a dataset

Comments:

The –*list– instructions specify options when printing a lesson on hardcopy. The error directory printed toward the end of the listing gives the line numbers on which there were errors. The only errors flagged are *list option errors and duplicate unit names. Putting the –*list– options at the beginning of block B is usually the safest.

It is often necessary to store information during execution of a lesson. As a simple example, a unit that sends the student to a remedial sequence of units after answering the question incorrectly four times must keep track of the number of times the student has answered incorrectly.

The PLATO system provides variables for storing such information. Normally, the student cannot access these variables, although the author can specify some of them for use by the student. The author has access to all the variables.

Each variable consists of one computer word of 60 binary digits (bits). This is equivalent to 20 octal digits or approximately 15 decimal digits. (For a discussion of the binary, octal, and decimal systems and their relationship, refer to appendix C.) All information is stored in the form of numbers. Alphanumeric information is converted to character codes (numbers) for storage. The character codes are each 6 bits so that 10 characters can be stored in one variable (word). Numbers are stored by magnitude only so that the number stored can be quite large.

Information on manipulation of individual bits and groups of bits is contained in sections 7 and 8.

STUDENT VARIABLES

The PLATO system provides each user with a bank of 150 student variables (sometimes called user bank variables) for storing information. If the user is a student, the system stores these variables on disk whenever the student is not using the system. If the user is not a student, the system does not store these variables between sessions.

The primitive names for these 150 student variables are nxx and vxx, where xx represents the variable number. The n and v indicate an integer variable or a floating-point variable, respectively.

A single variable in the student variables can be referenced as both a floating-point variable and as an integer variable in different parts of the lesson. This can be done either by referring to the variable with the primitive names (for example, n67 and v67) or by a defined name. However, the defined names must be different.

INTEGER VARIABLES

Integer variables are used to store integers up to 59 binary digits (2^{59} -1). The first bit, which is the leftmost bit, is used in indicating the sign of the number. If the sign bit is 0, the number is positive; if the sign bit is 1, the number is negative.

Negative numbers are stored as the complement of the positive number with the same absolute value. In binary, this means that all bits that are 1 in the positive number are set to 0, and all bits that are 0 in the positive number are set to 1. Thus, an integer variable containing the number 1 has the octal representation

0000000000000000001

while the value -1 has the representation:

7777777777777777776

A variable is selected as an integer variable from the student variables by prefixing the number of the variable in the bank with an n. For example, the name n27 indicates that the 27th of the 150 variables in the student variables is an integer. Any variable in the student variables can be designated as an integer variable.

FLOATING-POINT VARIABLES

If the number to be stored is larger than 2^{59}-1 or contains a fraction, an integer variable is not suitable. Instead, a floating-point variable is used.

The term floating-point is used because the decimal point is not fixed but floats to the right or left when operations are performed. The number is stored in the form of an exponent and mantissa. The format of the word is shown in figure 6-1.

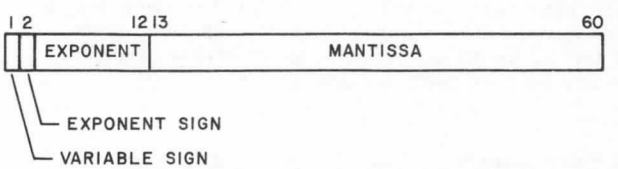


Figure 6-1. Format of Floating-Point Variables

The mantissa is 48 bits long, so numbers that would require more than 48 bits, such as large numbers with digits used in the lower places or extended fractions, lose some accuracy. The rightmost (least significant) digits of such numbers are lost. Usually, the accuracy provided by 48 bits is sufficient.

The exponent is 10 bits long, plus a sign bit for the exponent. Since the mantissa is treated as an integer but with the number shifted so that the first bit of the mantissa is set to 1, the number 1, for example, has an exponent of -47 (decimal). The octal representation of the value 1 in floating-point is:

1720400000000000000

The mantissa begins with the 4, which is 100 in binary, and causes the leftmost bit of the mantissa to be set to 1.

Additionally, there is a sign bit for indicating the sign of the entire number. This is the leftmost bit in the word. Negative floating-point numbers are handled in the same way as negative integers; that is, the negative number is the complement of the positive number with the same absolute value. Thus, the form of -1 would be

6057377777777777777

when stored as a floating-point variable.

Floating-point variables are referenced by prefixing the variable number with the letter v. For example, v23 refers to the 23rd variable in the student variables and specifies that it is a floating-point variable.

Floating-point numbers are sometimes called real numbers because they can approximate the real numbers rather than just the integers.

Variables can be indexed so that the actual variable used depends upon the value of the index. An example is v(n3). The value contained in n3 determines which variable is referenced. For example, if the value of n3 is 47, the reference v(n3) is equivalent to v47 but changes if the value of n3 changes. An expression, rather than a simple constant or variable name, can be used as the index.

NC AND VC VARIABLES

The PLATO system also provides each lesson with a bank of 1500 variables for storing information. The system does not store these 1500 variables with the student variables, because these 1500 variables are temporary variables in central memory. They are used with common and storage (refer to section 8).

The primitive names for these 1500 variables are ncxx and vcxx, where xx represents the variable number. The n and v indicate an integer variable or a floating-point variable, respectively, just as with the student variables. For example, variables nc1 and vc1 are the same variable. The way the author wants to use that variable determines if the author should define it as an integer variable (nc1) or as a floating-point variable (vc1).

ASSIGNING NAMES TO VARIABLES

While it is possible to write an entire lesson using only the primitive variable names, it is neither necessary nor desirable to do so. If there is more than one author of a single lesson or if the author is coming back to the lesson after an absence, it can be difficult to keep track of the significance of the various variables used. Thus, it is preferable to assign meaningful names to the variables. This is done with the –define– instruction.

The –define– instruction is a continued instruction. Each time the command –define– is encountered, a new set of variable names is defined. As a result, the command should only occur in the first line of the instruction.

The name of the variable set (define set) is also on the first line of the instruction. When it is used, it appears as the first argument in the tag of the first line. The definition of variable names may begin on the second line or after the set name on this first line.

```
define   set1,try=n1
         right=n5
         radius=v3,angle=v4
         person=nc1,topscor=vc2
```

The effect of the –define– instruction during lesson execution is to replace each occurrence of the item on the left of the equal sign with the quantity on the right. While this effect is not important in defining variable names, it is important when the –define– instruction is used to define functions, arrays, and segments (covered in sections 7 and 8).

As indicated in the preceding example, more than one definition can be made in a single define set. If more than one definition is to be made on a single line, the definitions are separated by commas. Definitions on separate lines do not require delimiters between definitions. It is not permitted to extend a single name definition over more than one line; hence,

```
define   set
         upper=v1
         lower=v7,medium=v8
         mean=v149
```

is a permissible instruction, but

```
define   set
         upper=
         v1
         lower=v7,medium=v8,mean=v149
```

is not permissible.

The variable names assigned can be no longer than seven characters. The first character cannot be a number, and the name cannot contain mathematical operators, FONT characters, or backspaces.

Because a define set is assigned a name itself, it is possible to specify that a define set includes, besides those definitions explicitly given, the definitions of one or more previous sets. This is done by giving the names of the sets whose definitions are to be included as arguments in the tag of the first line of the instruction, following the name of the new define set. For example, if the name one has already been assigned to a define set, the instruction

```
define   two,one
         zot=n89
```

includes all of the definition in set one, as well as the new definition of the variable name zot as set two.

The student define set is a special set of variable names. This set defines names that can be used by the student. For example, the instruction

```
define   student
         x=v1,y=v124
```

allows the student to type expressions such as 3x+y/4 and have the response understandable by the system. The student set is also necessary when using instructions such as –ansu–, –wrongu–, –storeu–, –ansv–, –wrongv–, –store–, and –compute– to define units useable by the student.

The student define set can redefine system functions. For example, cos(x)=cos(x°) allows the student to use degrees instead of radians.

A lesson can have any number of define sets, with up to five define sets active at any one time. When a sixth define set is created, all previous sets (except the define set named student, if used) are discarded. The purge option of the –define– instruction allows the author to selectively discard define sets. This option is exemplified by the instruction

```
define    purge,one
```

which renders the define set named one unavailable. If the second argument is omitted, as in

```
define    purge
```

all define sets except the set named student are discarded. Only one set can be purged by name in a –define– instruction. A –define– instruction using the purge option cannot be a continued instruction.

It is possible to make an available set active by simply giving the name as the tag of a one line –define– instruction. As an example, if the define set grelp was previously defined and made inactive (although available), but it is now desired to use the set grelp, the instruction performing this operation is:

```
define    grelp
```

Definition of a variable name must be done earlier in the physical (not logical) lesson than any use of that name, because –define– is not an executable instruction. Therefore, a good place to put define sets is at the beginning of the lesson.

All information on the PLATO system is stored in the form of numbers. However, there are two methods of interpreting the numbers, numeric and nonnumeric. The numeric method assumes the value of the number is of primary importance. The nonnumeric method assumes the number is a coded form of alphabetic information, such as a name. The difference is in the method of interpretation. Thus, a variable with the integer value 3 can be interpreted either as the number 3 or as the coded representation of the character c. (Character codes are given in appendix A.) All student input is in the form of characters. Instructions are available to convert student input to numeric form (refer to section 11).

Since all information is stored in the form of numbers, all information is manipulatable by means of expressions and functions. However, caution should be exercised in the use of expressions and functions on information that is to be used as nonnumeric information. Methods of dealing with specifically nonnumeric information are given in section 8.

CONSTANTS

There are two ways of expressing a constant in the author language. The number can be given explicitly, such as 143.52, or it can be a defined constant.

A defined constant is specified in the –define– instruction by assigning a specific value to a name. For example,

```
define   byte=8
         char=6
```

defines two constants equal to 8 and 6. These can usually be used within an author language lesson interchangeably with explicit constants.

EXPRESSIONS

An expression can usually be used in the author language wherever a numeric value is required. Expressions are composed of variables, functions, and constants separated by operators. The simplest form of an expression is a single constant, such as 3 or 237.36, or a single variable, such as x. Real and integer variables can be intermixed in an expression, with the result being rounded if an integer value is required. An example of rounding with integer variables is

```
v(2.5+6.8)
```

which is equivalent to

```
v(9)
```

Parentheses are used as in standard algebraic notation. The extensive use of parentheses is encouraged, even where not strictly necessary, for greater readibility and to ensure the expression is evaluated in the manner desired.

There are four types of operators in the PLATO author language: arithmetic, logical, bit, and array. All four yield numeric values, but the usage is different. An arithmetic operator returns a numeric value in the normal arithmetic manner. Logical operators return a value of -1 (true) or 0 (false). Bit operators affect the bit setting within a specified computer word. Array operators manipulate arrays. Operator precedence is given in table 7-1. The four types of operators can be mixed in a single expression, but because of the differing precedences, great care should be exercised, and parentheses should be used freely.

TABLE 7-1. OPERATOR PRECEDENCE

Precedence†	Operator	Operation
Binary		
9	(superscript)	Exponentiation
8	*,X	Multiplication
7	÷,/	Division
6	+,-	Addition, subtraction
5	cls,ars, $mask$,$diff$	Refer to Bit Operations
4	$union$	Refer to Bit Operations
3	>,<,≥,≤,=,≠	Logical relationals
2	and	Boolean AND
1	or	Boolean OR
0	⇐	Replacement
Unary		
2	-,+	Arithmetic sign

†Operations with a higher precedence are performed first.

For example, if y has a value of 22, the expression

```
100 -25(y<25)
```

is equivalent to 125 [y is less than 25, so the logical operation has a value of -1 (true)].

However, the expression

```
100-25y<25
```

because of the lower precedence of the logical operators is equivalent to

```
(100-25y)<25
```

which, in this example, gives a value of -1 (true), since 100-25y is -450 (quite a bit less than 25).

ARITHMETIC OPERATIONS

Arithmetic operations are performed as in standard algebra, with some differences in operator precedence from many computer languages.

In the author language, multiplication has a higher precedence than division, while many computer languages place the two operations on the same precedence level. The reason for giving multiplication a higher precedence is that experience has shown that most students evaluate an expression in that manner.

Because of the capabilities of the PLATO terminal, there is no operator in the author language to indicate exponentiation. Instead, the exponent is specified as a superscript through use of the SUPER (for superscript) key. Pressing SUPER allows the next character, which may be a number, to be written as a superscript. Pressing the SHIFT SUPER keys locks the terminal into the superscript mode, so expressions of more than one character can be written as a single superscript without the repeated use of the SUPER key. The mode is switched back from SHIFT SUPER by pressing the SHIFT SUB keys.

There are two ways of indicating multiplication. There is a special multiplication key in the cluster to the left of the alphabetic portion of the keyboard that produces a character resembling an x, called the multiplicative X . The alphabetic x cannot be used to indicate multiplication. The second method is the use of an asterisk (*). This functions like the multiplication key and is included because this character is used to indicate multiplication in many other computer languages. The use of the multiplicative X is preferred to the use of the asterisk.

Because of the operation of the author language condenser and executer, an author must indicate multiplication explicitly. Thus, while an expression such as $r\cos\theta$ is permitted in algebra, the author must use r*cos(θ) in the author language. That is, multiplication must be explicitly indicated, and the arguments of functions must be enclosed in parentheses. These restrictions do not apply to the user executing the lesson.

When an explicit constant is to be multiplied by a named variable, concatenation to indicate multiplication is available to the author. For example, the author can use 23theta rather than 23*theta. However, theta23 cannot be used since the condenser would not know whether the 23 is a part of the name or a constant.

Similar to multiplication, there are two operators that indicate division, ÷ and /. The ÷ symbol is produced, like the X symbol, by a special key to the left of the alphabetic keyboard. The use of the / symbol is permitted because of its use in other computer languages and because it is commonly used to indicate division in algebra.

Addition and subtraction have the same precedence and are executed from left to right. The keys used to indicate addition and subtraction are the + key and the – key, both of which are near the multiplicative X and ÷ keys.

LOGICAL OPERATIONS

There are two types of logical operators, relational and Boolean. Relational operators have a higher precedence.

The relational operators are used to compare the numeric magnitude of expressions. They return a value of true (-1) or false (0), depending on whether the relation is true or false. The relational operators are the standard mathematical ones: > (greater than), < (less than), ≥ (greater than or equal to), ≤ (less than or equal to), = (equal to), and ≠ (not equal to).

The Boolean operators permit the combination of logical expressions. There are three Boolean operators: and, or, and not. Note that the Boolean not (the unary negation) is not enclosed in dollar signs ($). The not is actually a system-defined function that changes the value of a logical expression. As a result, it is necessary to enclose the expression to be affected in parentheses.

The other two Boolean operators are binary and combine two logical subexpressions into a single logical expression. The and operator gives the expressions a value of true (-1) if and only if both subexpressions are true. The or operator gives a value of true if either or both of the subexpressions are true (inclusive OR).

Although many computer languages give logical expressions a value of 1 if true or 0 if false, the author language assigns -1 and 0, respectively. This is to allow the use of logical expressions in the conditional form of some instructions (refer to section 12).

An expression other than a logical expression (such as 3x+5 or vname $mask$ 73) always has a logical value of true (-1) when used as part of a logical expression.

BIT OPERATIONS

Bit operations manipulate the information contained in a single word. On the PLATO system, one computer word consists of 60 binary digits or bits. For convenience, octal numbers can be used. One octal digit is equal to three binary digits (refer to appendix C for comparisons of binary, octal, and decimal numbers) and has the added advantage of being quite close to decimal notation. An octal number is identified in the author language by a lowercase o preceding the number. For example, o753 is an octal number equivalent to 111 101 011 in binary and 491 in decimal.

All bit operators are enclosed in dollar signs ($) to distinguish them from variables. Except for the $union$ operator, all bit operators have the same precedence. Therefore, the result of an expression depends not only on the operators and operands but on their order of occurrence. For example,

```
o77$mask$o70$ars$3
```

gives a value of o7, while

```
o77$ars$3$mask$o70
```

gives a value of zero. As a result, parentheses are usually necessary when using bit operations.

The author language provides two shift operators, cls and ars, which are acronyms for circular left shift and arithmetic right shift, respectively.

The circular left shift causes the contents of the word to the left of the cls operator to be shifted to the left the

number of bit positions indicated by the expression to the right of the operator. The bits that are shifted from the word on the left are brought into the word on the right. A circular left shift of 60 would leave all bits in precisely the same position they originally occupied (hence, the term circular in the name).

The arithmetic right shift operates in the same general manner as the circular left shift, with three exceptions. The shift is to the right rather than to the left. Bits that are shifted from the word on the right are lost. The sign is extended from the left, so the number remains positive or negative, whichever it was originally.

The remaining bit operators ($union$, $mask$, and $diff$) are used to compare the bit configurations of two computer words. The $union$ operator sets the bits of the result to 1 wherever either word being compared has a 1. The $mask$ operator sets a bit in the result to 1 only if both words have the bit in the same position set to 1. The $diff$ operator sets a bit in the result to 1 only if the bits in that position in the two comparison words have different values.

For example, if the lower four bits of variable test are 1010 and the lower four bits of variable check are 1100, the result of

```
test$union$check
```

has 1110 as its lower four bits,

```
test$mask$check
```

has 1000 as its lower four bits, and

```
test$diff$check
```

has 0110 as its lower four bits.

The $union$ operator has a lower precedence than the other bit operators. The other operators all have the same precedence and are executed from left to right if the order of execution is not modified by parentheses.

The system function, bitcnt (x), counts the number of bits set to 1 in the variable named in the argument.

The system functions, lmask(x) and rmask(x), generate left- or right-justified masks, respectively, of length x bits.

ARRAY OPERATIONS

Array operations modify system-defined arrays (refer to section 8). Because the system treats system-defined arrays as ordinary elements, all the arithmetic and logical operations are allowed (except exponentiation). When referencing an array with indexes, the indicated operation affects only the element referenced. When referencing an array without indexes, the indicated operation affects all the elements of the array. Using array-scalar operations, such as

```
A ⇐ 4
B(3,4) ⇐ n5
C ⇐ DX10
E ⇐ a2 X E
```

is legal. All the elements of array A are set to four, the element B(3,4) of array B is set to the value stored in n5, all the elements of array C are set to values 10 times the corresponding elements of array D, and all the elements of array E are increased by a^2.

Element-to-element array operations do not require indexes. For example:

```
A ⇐ B/C
D ⇐ sin (E)
```

Here, each element in array A is set to the result of the division of the corresponding elements in arrays B and C. Each element in array D is set to the sine of the corresponding element in array E. These operations require that the arrays have the same dimensions.

Two operators are used only with arrays. One is ∘ , the vector dot product or matrix multiplication, produced by pressing MICRO X . The other is × , the vector cross product, produced by pressing MICRO SHIFT X .

```
var ⇐ vec1 ∘ vec2
C ⇐ A ∘ B
vec3 ⇐ vec1 × vec2
```

In this example, var contains the vector dot product of the vectors vec1 and vec2. If var is not a scalar, all its elements are set to vec1 ∘ vec2. Array C contains the result of the matrix multiplication of arrays A and B. Vector vec3 contains the vector cross product of the vectors vec1 and vec2.

The author must be cautious when chaining together more than one array operation, because the computer might lose the temporary storage buffers if too many operations are attempted in one instruction.

-CALC- INSTRUCTION

For assigning a value to a variable, the author language provides the –calc– instruction. This can be a continued instruction, as is the –define– instruction. The tag consists of the assignment desired.

Assignment is specified by having a variable name (either an assigned or a primitive name) separated by an ⇐ from the expression giving the value to be assigned the variable. For example:

```
calc    inum ⇐ 3X theta + y ÷ 2
```

The assignment key (⇐) is located to the left of the alphabetic keyboard. A variable may appear on both the left and right sides of an assignment. In this case, the current contents of the variable is used when evaluating the expression. This value is then assigned to the variable. For instance, if variable fiawol contains the value 7, then

```
calc    fiawol ⇐ fiawol2
```

assigns fiawol a value of 49.

A single statement can contain more than one assignment, as in

```
calc    fout ⇐ v1 ⇐ jwc ⇐ 362 ÷ x2
```

which assigns the value of the expression $36^2 \div x^2$ to all three variables: fout, v1, and jwc.

An error in a –calc– instruction gives an execution error.

There are three instructions that can be used for specific calculation. The –add1– instruction adds one to the value of the variable in the tag. The –sub1– instruction subtracts one from the value of the named variable. The –zero– instruction can set an individual variable, or an entire block of variables, to zero. The –zero– instruction, when it is used to set a block of variables to zero, is much faster than using a –calc– instruction. The –add1– and –sub1– instructions are slower than a –calc– instruction.

The tag of the –zero– instruction can have either one or two arguments. The first (or only) argument specifies the starting location of the block of variables to be set to zero. If the tag has one argument, only this variable is zeroed. If the tag has two arguments, the second argument gives the number of variables to be set to zero. As an example, the instruction

```
zero    v23,17
```

sets 17 variables to zero, beginning with variable 23.

FUNCTIONS

Functions are used to perform lengthy, complex, or repeated calculations. They allow the calculational statements to be made once and then to be referenced by a name.

Several useful functions, called system functions, are built into the author language and need not be defined by the author. An example is sin(x), which gives the value of the trigonometric sine of x, where x (in radians) can be an expression. The system functions which begin with a capital letter deal specifically with system-defined arrays (section 8). These array functions may be used with the other system functions. (For a list of system functions, refer to appendix B.)

Often the function desired is not available as a system function. Thus, the author language allows the author to define his own functions. This is done in a –define– instruction in a manner similar to defining a variable. The method is probably best explained by the example

```
define  circle
        diam(a)=2π X a
```

where a, acting as a dummy variable, has not been previously defined. When used with a variable, such as radius, this function gives the diameter of the circle whose radius is contained in the variable. A variable that has been previously defined cannot appear on the left of the equal sign in a function definition, although it can appear on the right side, as in:

```
define  set1
        x=v1
        func(t)=t^4+sin(x)
```

A function can be used on the right side. The author language allows this nesting of definitions to go down seven levels (if the function is very simple). A function can also be an assignment, as in

```
define  set2
        x=v1
        cube(a)=a^3
        root(a)=a^1/4
        value=(x ⇐ root (cube(x)))
```

which permits the instruction

```
calc    value
```

where the change in the value of x is implicit because of the definition of the function value. Note that value needs no arguments.

Any legal expression can be the definition of a function, and as shown, an assignment is also possible. Since an operator by itself is not a permissible expression, a definition such as

```
define  neg = not
```

is not legal, but

```
define  neg(b) = not(b)
```

is legal.

A function can have up to six arguments. As an example of a function of three arguments, consider:

```
define  root1 = v1
        root2 = v2
        soln1 (a,b,c) = (root1 ⇐ (–b+
                         sqrt(b^2–4a X c))/(2a))
        soln2 (a,b,c) = (root2 ⇐ (–b–
                         sqrt(b^2–4a X c))/(2a))
```

Using these functions, the instruction

```
calc    soln1 (3,5,1)
        soln2 (3,5,1)
```

puts the two solutions of the equation $3x^2+5x+1=0$ into variables root1 and root2. Of course, some means of determining if the equation has real roots should be used; otherwise, an error is generated and lesson execution is stopped when the roots are not real.

All computation necessary in a PLATO author language lesson can usually be done using the instructions described in sections 6 and 7. However, the lessons that result are often extremely complex in structure and clumsy in execution. To further simplify the task of information manipulation and to make lessons less complex and more comprehensible, several more sophisticated data handling capabilities are built into the author language.

SYSTEM-DEFINED ARRAYS

Ordered sets of data often can be usefully arranged into arrays. The location of the data then bears a clear relationship to the information which the data represents. Arrays may be either system-defined or author-defined. Author-defined arrays are discussed later. System-defined arrays may be zero, one, or two dimensions. Several operations and system functions (refer to section 7 and appendix B) are available to manipulate system-defined arrays.

The –define– instruction names an array. The indexes, offsets, and base locations in the definition must be literals or defined constants, because the array size must be known at condense time. At this time, the system sets aside enough storage to contain the array. Then, whenever the lesson references the array, the system does a bounds check on the array. Referencing values outside the specified index limits causes an execution error. The two types of system-defined arrays are full-word arrays and vertically segmented arrays.

FULL-WORD ARRAYS

Full-word arrays use one entire variable for each element in the array. Variables may be n, v, nc, or vc type variables. A full-word array can be one, two, or zero dimensions. An example of a one-dimensional array is:

```
define   array,list(5)=v20
```

This defines a vector of five elements, starting with list(1) in v20. The last element, list(5), is in v24. The elements of list are floating-point numbers.

At times, the author may desire to begin the index with a value other than one. Therefore, the author language allows index offsetting. For example,

```
define   array,year(1860;1880)=vc50
```

defines year (1860) as the first element in this 21-element vector. Both the beginning and ending index values are given, separated by a semicolon.

Two-dimensional arrays are defined similarly to one-dimensional arrays; however, the size of both dimensions must be specified. For example,

```
define   array,A(4,4)=n1
         array,B(0,0;4,5)=nc10
```

Array A is a 16-element square matrix. The elements of a matrix are stored by rows; therefore, A(1,1) equals n1, A(1,2) equals n2, A(1,3) equals n3, A(1,4) equals n4, A(2,1) equals n5, A(2,2) equals n6, and so on.

Array B is a 30-element array with five rows and six columns. The initial index is offset to B(0,0). The maximum value an offset may be is $2^{13}-1$.

An array can also be defined to be of zero dimension, as in:

```
define   array,scale=vc10
```

An array of zero dimension is called a scalar and consists of a single element. It is essentially the same as defining a constant; however, the array operations discussed in section 7 sometimes make the use of scalars desirable.

The –define– instruction allows the defining of individual elements of an array after the array as a whole is defined. For example:

```
define   array,group(15)=v25
         beg=group(1)
         mid=group(8)
         end=group(15)
```

VERTICALLY SEGMENTED ARRAYS

Vertically segmented arrays use one vertical segment for each element in the array. That is, an element uses only the specified bits in the computer word, rather than the entire word, with consecutive elements in consecutive computer words. Vertically segmented arrays can decrease the number of variables necessary, because the same variables can contain more than one vertically segmented array. As with full-word arrays, the variables may be n, v, nc, or vc type variables; however, vertically segmented arrays are restricted to one and two dimensions.

An example of a one-dimensional vertical array is:

```
define   arraysegv,name(15)=n12,5,25
```

Array name has 15 elements stored in integer variables n12 through n26, occupying bits 5 through 29. Vertical arrays may also specify initial index offset, as in:

```
define   arraysegv,grade(101;115)=n12,31,11,signed
```

Array grade also has 15 elements stored in integer variables n12 through n26; however, its elements occupy bits 31 through 41, including one bit for the sign.

Two-dimensional vertical arrays, with or without initial index offset, are defined similarly to one-dimensional arrays, except that both dimensions must be specified.

The system stores elements of segmented arrays as integers. To store a floating-point value, the author must first scale the value up or down and then store it as an integer.

AUTHOR-DEFINED ARRAYS

Sometimes an author wants more than two dimensions in an array. In this case, the author can define arrays of more than two dimensions; however, these author-defined arrays cannot be used with the system array functions and operations. There are three types of author-defined arrays: full-word arrays, horizontally segmented arrays, and vertically segmented arrays.

FULL-WORD ARRAYS

Arrays can be defined by indexing the primitive variable name (v, n, vc, and nc). This is done by putting the number or expression in parentheses, like v(num+10). The value of the indexing expression is rounded, if necessary.

If the array starts in variable 37, the instruction

```
define    mat(n)=v(36+n)
```

defines the array, with the assumption that n is not previously defined.

If the student is allowed to specify an element of an array, the array must be defined by name with subscripts. The definition must be in the student set of variables.

Because this definition of an array does not specify a limit on the size of the array, a certain amount of care must be taken to ensure that the index variable is within the desired bounds. For instance, if the array mat has been defined as given previously and is to contain 20 elements, a value of n that is greater than 20 does not cause an execution error but simply accesses a variable beyond the bounds of the array. Similarly, if n has the value -10, a reference to mat(n) references variable v26, which is previous to the defined start of the array.

An example of a two-dimensional 3 by 15 array with index variables i and j is:

```
define    quad(q,m) = v(19+15(q-1)+m)
```

The actual index variables cannot be used in the definition. As in the definition of a function, dummy variables must be used.

The array starts in variable 20 (assuming that both q and m are never less than one). For example, the fifteenth element of the second row is then accessed by quad (2, 15). The expression (q-1) in the definition is used to start the array at variable 20, rather than 35, as would happen if simply q were used.

There is an alternative way to define the array so that no arguments are visible. The previous example, defined in this manner, would become

```
define    set1
          i=n1,j=n2
          quad=v(19+15(i-1)+j)
```

and once values for i and j have been set, the use of the word quad automatically gives the value of the proper variable. This is especially useful when a long routine that uses an array name quite often is used. The form of array with no explicit arguments simplifies the writing of the routine. Just as in function definition, previously defined variables can appear on the right side of the array definition. In fact, if the array is to be referenced without specific arguments, as shown, it is necessary that the indexes given in the definition be previously defined.

The –set– instruction is used to assign values to consecutive variables or to previously defined arrays. It assigns values or mathematical expressions to full words, and therefore, may not be used to assign segments. If array boundaries or student or central memory (nc or vc) variable boundaries are exceeded using a –set– assignment, condense or execution errors occur.

The following example –set–s n5 equal to 4, n6 equal to sin(x), and n7 equal to 14.7.

```
set       n5 <4,sin(x),14.7
```

Arrays may be partially or fully assigned.

```
set       B <2,4

set       B<4,8,19
             1,0,1
             2,3,5
```

If B is defined as a 3X3 array, B(1,1) equals 2, and B(1,2) equals 4. The second –set– reassigns the array to B(1,1) equals 4, B(1,2) equals 8, B(1,3) equals 19, B(2,1) equals 1, B(2,2) equals 0, and so on.

SEGMENTS

A segment is a special form of an array that uses less than a full computer word for storing the values. For example, if it is known that all the values to be stored are less than 200, that they are all positive integers, and there are 200 such scores to be saved, the student variables cannot store all the values. Common variables can be used instead, but a large amount of space is wasted since only part of each word is used.

The instruction

```
define    segment,list=v1,8
```

sets up an array composed of eight bit units, or bytes, starting in variable 1, with the array having the name list. As in an ordinary array definition, the quantities on the right of the equal sign can be previously defined quantities. For example, the instruction

```
define   start=v1
         byte=8
         segment,list=start,byte
```

has precisely the same effect as the previous definition of a segment list. An indexed variable cannot be used in the definition of the starting variable, and the byte size must be specified by a constant.

A segment is referenced with a single index [for example, list(3)].

If the bytes of a segment are not necessarily positive, a sign option is available. This is done by writing signed or s as a final argument after the rest of the segment definition. However, if signed bytes are to be used, an extra bit must be included. This bit is used to indicate the sign of the byte. Continuing the example of the segment named list, this means that the definition should be:

```
define   segment,list=v1,9,s
```

The size of the bytes in a segment can be from 1 to 59 bits. However, a byte larger than 30 bits occupies a full word. The bytes of a segment do not cross computer word boundaries. Thus, if the size of a byte is defined to be 16 bits, byte 4 of the segment starts at the left of the second variable used.

A segment cannot be stored into by instructions such as –store– and –do–. In general, a segment reference is illegal anywhere a nonstorable expression such as n1+1 is illegal, except when assigning a value in a –calc– instruction. Thus, the instruction.

```
pack     list(1)
```

is not legal, while the instruction

```
at       list(1)
```

is legal.

VERTICAL SEGMENTS

A vertical segment defines a segment where consecutive bytes are in consecutive words.

For example, the instruction

```
define   segment,vertical,list=n1,8,10
```

defines a vertical segment list such that list(1) occupies bits 8 through 17 (where bit 1 is the sign bit of the word) of n1. list(2) occupies bits 8 through 17 of n2, and so on.

Vertical segments execute faster than regular segments.

NONNUMERIC INFORMATION

One of the major uses for manipulation of bytes is working with nonnumeric information. This is usually information given by the student as a response. Some nonnumeric information, however, is obtainable from the system. Additionally, the author can deal with nonnumeric information of his own specification, if desired.

Nonnumeric information is also called alphanumeric information, since the data objects of concern are not the numbers themselves but the alphabetic, numeric, and special characters that the numbers represent.

In the PLATO system, all alphanumeric information is represented by codes of six bits per character. Thus, the character a has code 01, the character g has code 07, and so on. There is a system-reserved word, called key, that contains a code for the last key pressed. However, these key codes are not the same as the actual internal representation of the keys (called the character codes). A complete list of the key codes, character codes, and the characters they represent is given in appendix A.

A capital letter (for example, A) requires two character codes. The first code is for the shift, and the second code is the actual character code.

Some instructions, such as the –move– instruction, are designed for manipulating alphanumeric information. In addition, the bit operators described in section 7 can be used for manipulating individual character codes, or groups of character codes, as well as other bit operations.

ENTERING ALPHANUMERIC INFORMATION

There are two methods for the author to specify alphanumeric information in a lesson. The first method is entering values into a variable which correspond to the characters desired. This has the disadvantage of being slow, prone to error, and difficult to interpret when rereading the lesson. An alternate method is the use of literals. An example of this use is the instruction:

```
calc     n47 ⇐ "funk"
```

The character string funk is the literal. This instruction puts the character codes for the literal in variable 47, right-justified and zero-filled; that is, the k is at the right end of the computer word, and the left part of the word is set to zero. Since most instructions work with left-justified strings, a single quote (apostrophe) stores a literal left-justified. The previous example would, in this case, become:

```
calc     n47 ⇐ 'funk'
```

Literals must be no longer than 10 character codes. Capital or access characters usually require two or three character codes. The extra codes are for the shift and access characters. These extra codes must be counted as part of the literal. Thus, the literal alpha is 5 character codes long, while the literal ALPHA is 10 character codes long.

Each variable, that is, each computer word, can contain up to 10 character codes. It is good practice when storing alphanumeric information to have the variable in which the information is to be stored an integer rather than a floating-point variable. In the case of storing a literal, this is necessary for proper execution.

The –pack– instruction is used to store a string of alphanumeric characters (usually referred to as simply a string).

The instruction has three arguments. The first argument is the starting location where the string is to be stored. The second argument receives the character count of the string being stored, and the third argument is the string to be stored. The string is stored left-justified. These strings can cross word boundaries, unlike literals. The second argument can be omitted by using successive commas.

Partial words (that is, words that are not completely filled by the character string) have the unspecified portion of the word set to zero.

The –packc– instruction is the conditional form of the –pack– instruction. Depending on the value of the expression in the first argument, different strings may be left-justified –pack–ed into the variable specified. The –packc– may have up to 100 arguments and may use the various forms of embedded –show–. The separators allowed are , ; ǂ and end-of-line. The separator which immediately follows the length variable is the only one allowed to separate the strings. If the length variable is not used, successive separators are used, as in:

```
packc   n2–2;n4;; ⊲a,count⊳ ;number;; ⊲s,n50⊳
```

The –itoa– instruction is used to convert integer values to the character string that the integer represents. The instruction can have either two or three arguments. In the two-argument form, the value in the first argument variable is converted to a string and placed in the variable name in the second argument. In the three-argument case, the first two arguments function in the same manner, and the third argument receives the numeric character count of the string. In both cases, all arguments must be variable names. Only integer variables should be used. The converted string is left-justified, with any excess set to zero.

As an example, the instructions

```
calc    n27⇐135
itoa    n27,n35
```

would put the character code for the string 135 in variable 35. Similarly, the instructions

```
calc    n27⇐135
itoa    n27,n35,n2
```

would place the same values in variables 27 and 35 and would put the (numeric) value 3 in variable 2.

The –otoa– and –htoa– instructions are similar to the –itoa– instruction except for the third argument. The –otoa– instruction converts the octal representation of a number to the alphanumeric representation of the number by converting each 3-bit octal code to a 6-bit alphanumeric code. The value in the first argument variable is converted to a string and left-justified in the first word of a 1- or 2-word buffer where the second argument variable names the first word. The optional third argument is a variable containing the number of octal digits to be converted (counting from the right). If this value is < 10, the second word in the buffer does not change. The default for the third argument is 20.

The –otoa– instruction is useful when the –showo– instruction cannot be used; for example, when printing blocks of alphanumeric characters.

The –htoa– instruction converts the hexadecimal representation of a number to the alphanumeric representation of the number, by converting each 4-bit hexadecimal code to a 6-bit alphanumeric code. The –htoa– instruction works exactly like the –otoa– instruction with one exception; the default for the third argument of the –htoa– instruction is 15.

LOCATING SPECIFIC INFORMATION

Five instructions that check for the occurrence of an object are –find–, –findall–, –search–, –finds–, and –findsa–. The first two instructions look for an object which can be any sequence of bits of arbitrary length in a variable. The –search– instruction looks for an object which is a character string. These instructions can use sorted or unsorted lists. The –finds– and –findsa– instructions look for numeric or alphabetic objects in sorted lists.

The –find– instruction has four to six arguments. The first four arguments are the same in all cases.

The first argument specifies the variable that contains the object for which the other computer words are searched. The second argument specifies the variable at which the search is to start. The system searches through the number of variables specified by the third argument. The number of the word in which the object is found (relative to the starting variable) is placed in the variable specified by the fourth argument. If the object is not found, the value of the variable specified in the fourth argument is –1. The optional fifth argument specifies that every nth variable is to be compared with the object; the argument is assumed to be 1 if not present. The fifth argument must be present if the optional sixth argument is used.

The object for which the search is made need not be the entire variable specified by the first argument. It can be modified by the optional sixth argument, which is a mask. It should be specified in octal for clarity. Thus, if the object for which the other variables are searched is contained in the lower seven bits of variable n5, the instruction

```
find    n5,start,length,return,1,o177
```

is equivalent to the instructions:

```
calc    n5⇐n5$mask$o177
find    n5,start,length,return
```

For a specific example, consider the instruction

```
find    n5,n30,10,n17,2,o177
```

which attempts to find the object (bit pattern) in the lower seven bits of variable n5 in every second variable of the 10 variables starting with variable 30 (that is, variables n30 to n39). If the object is found in variable 30, the value of variable 17 is 4. The return value is independent of the value of the optional increment argument. If the object is not found at all, variable 17 is assigned the value -1.

The search can be made backwards through the variables by specifying a negative length. The search then starts at the high end of the variables to be searched and proceeds

back down the list. However, the variable number returned if a match is found is then the same as in the forward search. Hence, if in the previous example the length was -10, the results would be precisely the same, provided that only one occurrence of the object for which the search is being made is contained in the variables searched. If there is more than one occurrence of the requisite pattern, a forward –find– finds the first instance, and a backward –find– finds the last instance.

The –findall– instruction is similar to the –find– instruction in that it searches a list of variables to find a specified object. The –findall– instruction returns a count of the number of matches, and it then lists the location of the matches. The increment between variables searched can be specified to be other than one.

The tag for the –findall– instruction consists of five required and two optional arguments.

Argument	Definition
1	**object** to be found
2	**starting** variable of list to search
3	**length** of list
4	Starting **return** variable (cannot be a segment)
5	Number of **following** variables for storage of found locations (0 for count only)
6	Optional variable **increment** specifies only every nth word for comparison with object (assumed to be 1 if not given)
7	Optional **mask** of variable containing object (full-word search if not given)

For example, instruction

```
findall   n3,n50,n2,return,0
```

counts the number of times object is matched.

Variable	Return
n50 and n53	2
n53 only	1
no match	0

In contrast, the instruction

```
find      n3,n50,n2,return
```

places in return the relative location of the first match of the object.

The locations of matches are specified as the offset of the location from the starting variable. Therefore, if the object matches the first variable, the location value returned is 0. If there is no match, the location value returned is -1.

If the mask argument is used, the increment must be specified (1 in normal case).

If either the increment or length is 0, no search is performed, a count of 0 is returned, and the first entry of the following list (if any) is set to -1. If the increment specified exceeds the length, an execution error occurs. The search can be made backwards from the last variable in the list by specifying the increment as negative.

The –findall– instruction does not work with segmented variables.

When the object to be found is a character string, the –search– instruction is more efficient than the –find– instruction. The –search– instruction functions in the same manner as the –find– instruction, but it does not allow a mask argument. It does, however, require an argument specifying the length of the object string. When the count is specified, –search– operates like a –findall–.

The object string is left-justified in the variable containing it. The length of the object string can be no longer than 10 characters.

The search finds the object string regardless of the position of the string within a variable or whether the string crosses word boundaries.

As an example, the instruction

```
search    n1,5,n37,15,1,n90
```

searches for an occurrence of the first five characters in variable n1 in the set of variables n37 through n51, starting at the first character position of n37. If an occurrence is found starting in n40, the variable n90 contains the value 3. If no occurrence is found, n90 contains the value -1. In the example above, however, if the sixth argument (n90) were followed by the seventh argument (**count**), the –search– instruction would operate like a –findall– and give the locations of all occurrences found.

If the fourth argument is negative, the search is backwards, from the end of the list to the beginning of the list. The absolute value of the fourth argument gives the number of variables to search. Occurrences are stored in the order they are encountered, that is, in descending order.

The –finds– and –findsa– instructions find objects in sorted lists. The –finds– instruction works for numerically sorted lists, and the –findsa– instruction works for alphabetically sorted lists. Because these instructions do not check if the list is sorted, the return on an unsorted list is unpredictable. The –finds– and –findsa– instructions have either seven or eight arguments and are similar to the –sort– and –sorta– instructions.

The first argument of their tags is a variable containing the object to be found. The second argument is the starting location of the list which is to be searched. This list can be in student variables n(x), in central memory variables nc(x), in ECS common, or in ECS storage. This argument must be followed by a semicolon (;) as a separator. The other arguments are separated by commas.

The third argument specifies the number of entries in the list. The fourth argument specifies the number of words per entry. If this argument is 0, it causes an execution error. The fifth argument gives the first bit of the numeric field for -finds- (bits 1 through 60) or the first character location of the character field for -findsa-. The following argument gives the number of bits or characters in the search field. The numeric field of the -finds- instruction cannot cross word boundaries, but the character field of the -findsa- instruction can cross word boundaries if it doesn't cross into another entry of the list.

The seventh argument is a variable returning the list entry number of the object that was found. This is unlike the -find- instruction which returns the CM position of the found object. If -finds- or -findsa- cannot find the object, the seventh argument is the negative of the position where it would be if it had been found. This is useful for later insertion.

The optional eighth argument, a mask, modifies the object in the first argument. It can be used to clean up the bits before and after the bits specified in the fifth and sixth argument. A mask consisting of all zeros causes a condense error. For the -findsa- instruction, the mask can be at most one word, masking only the first word of the object.

If variables n30 through n33 contain these values:

Variable	Contents
n30	1
n31	4
n32	9
n33	16

the instruction

```
finds    n10,n30;4,1,1,60,n11
```

returns the following values in n11:

n10	n11
1	1
5	-3
9	3
25	-5

SORTING ROUTINES

The -sort- and -sorta- instructions are used to sort numbers and words, respectively, and are very similar. The -sort- instruction arranges a list of numbers in numerical order, from smaller to larger. The -sorta- instruction arranges a list in alphabetical order according to the numeric codes for letters. The first argument of their tags is the location of the list which is to be sorted. This list can be in student variables n(x) or v(x), in central memory variables nc(x) or vc(x), in ECS common, or in ECS storage. The entries in the lists may be one or more words in length but may not be partial words. The numeric field in the list may not cross word boundaries. For example, the sort may be on bits 24 through 34 but not on bits 54 through 64. The character field in the list may cross one word boundary.

The first argument in the tag must be followed by a semicolon (;) as a separator. The other arguments are separated by commas. The second argument is the length of the list, which is the number of entries in the list. The number of words per entry is specified in the third argument. This must be an integer. The fourth argument gives the first bit of the numeric field for -sort- or the first character location of the character field for -sorta-. The argument following this gives the number of bits or characters in the sort field.

The optional sixth argument, a mask, modifies the entry implied by the fourth and fifth arguments. A mask consisting of all zeros causes a condense error. For the -sorta- instruction, the mask can be at most one word, masking only the first word of the object.

An option for paired list sorting allows an author to sort linked lists. With this option an author may sort a list of names and automatically sort the corresponding list of addresses. To do this, the starting location of the associated list is given in the second line of the tag, followed by the number of words per entry, separated by a semicolon. The command field of this line must be blank.

If common or storage variables in ECS are being loaded by -comload- or -stoload-, a -sort- or -sorta- in ECS is not permissible. An execution error results if this is attempted.

CHANGING LIST CONTENTS

The -inserts- and -deletes- instructions change the contents of lists. The -inserts- instruction inserts entries into lists, and the -deletes- instruction deletes entries from lists. The instructions work with either sorted or unsorted lists and with either numeric or alphabetic lists. Their arguments are similar to the arguments of the -sort- and -finds- instructions.

The first argument of the -inserts- instruction is a variable containing the object to be inserted. This argument must be a variable and must be the same word size as the other entries in the list. The second argument is the starting location of the list. This list can be in student variables n(x), in central memory variables nc(x), in ECS common, or in ECS storage. This argument must be followed by a semicolon (;) as a separator. The other arguments are separated by commas.

The third argument specifies the number of entries in the list before the insertion occurs. The fourth argument specifies the number of words per entry. If this argument is 0, it causes an execution error. The fifth argument is the position in the list where the object is to be inserted. This argument cannot be less than 1 or greater than 1 + the length of the list (the third argument). This allows the insertion of an entry at the beginning or at the end of the list. The optional sixth argument specifies the number of entries to insert.

If variables n30 through n33 contain these values:

Variable	Contents
n30	18
n31	22
n32	16
n33	19

the instruction

```
inserts   n5,n30;4,1,3,1
```

with n5=21, returns this list:

Variable	Contents
n30	18
n31	22
n32	21
n33	16
n34	19

The arguments of the –deletes– instruction are the same as the arguments of the –inserts– instruction except for the first argument. The –deletes– instruction does not specify the object to be deleted; it only specifies the list position of the object. Therefore, the tag of the –deletes– instruction starts with the starting location of the list.

The –deletes– instruction deletes an entry, moves the list together, and fills the last position with zeros. For example, using the instruction

```
deletes   n30;4,1,3,1
```

on the original list in the example above, returns this list:

Variable	Contents
n30	18
n31	22
n32	19
n33	0

The –inserts– and –deletes– instructions have options for associated list insertion and deletion, allowing an author to insert or to delete entries from the same relative position in two associated lists. To do this, the instructions add another line in their tags. The arguments in the second line of the –inserts– instruction are the variable containing the object to be inserted, the starting location of the associated list, and the number of words per entry. The arguments in the second line of the –deletes– instruction are the starting location of the associated list and the number of words per entry. The instructions

```
inserts   n5,n30;10,2,5,4
          n7,n100;3
deletes   n30;14,2,5,4
          n100;3
```

insert 4 entries in each associated list and then delete those same entries. The entries are inserted starting in position 5. The entries in the first list are 2 words long, and the entries in the second list are 3 words long.

MOVING CHARACTER STRINGS

The –move– instruction copies a string from one location to another. The first argument gives the variable in which the string to be moved starts. The second argument gives the location in that variable of the start of the string (that is, the appropriate character location). The third and fourth arguments perform the analogous functions for the destination in the same order. The final argument, which is optional, gives the length of the character string to be moved. If not otherwise specified, a single character is moved.

The character locations (the second and fourth arguments) can cross word boundaries (that is, be larger than 10) but must be greater than 0. Hence, the instruction

```
move      n1,13,n123,2,6
```

moves the six characters starting in character location 3 of n2 (location 13 of n1) to the location beginning with character position 2 of variable 123.

Up to 1500 characters can be moved in a single –move– instruction.

The moved characters are merged with the characters already in the destination word; that is, characters that are not overlaid with the moved characters are unaffected by the instruction. Similarly, the characters at the origination point are left unchanged. Thus, the –move– instruction can be used to create a copy of a character string without destroying or modifying the original string.

COMPILING CHARACTER STRINGS

The –compute– instruction can be used to compile a character string representing an expression. The tag has four arguments. The first specifies the variable that receives the value of the expression; the second specifies the variable in which the expression begins; the third specifies the length of the string (in character codes); and the fourth is a variable that receives a pointer to the compiled code, so the expression need not be recompiled on reevaluations of the expression. If the value returned in the pointer is zero, the compiled code was not saved. The expression must then be recompiled if it is reevaluated.

The result of the –compute– instruction is similar to that of the –store– instruction, but there are differences (refer to section 11). The –compute– instruction is a regular instruction, while the –store– instruction is a judging instruction. As a judging instruction, the –store– instruction works only with the student's response, while the –compute– instruction can compile and evaluate any character string in the variable bank (but not a student response, unless the response has been stored previously with a –storea– instruction). Finally, the –compute– instruction saves the compile code so reevaluation can be done much more rapidly, while the –store– instruction does not save the compile code past the time-slice, which means that it is useable only once.

The student define set must be used so that the string compiles during execution.

Because the compile code is saved, if the string is changed and the –compute– instruction is to be reexecuted, the pointer (argument 4) must be set to zero to ensure proper recompilation. If the expression is to be reevaluated at a later time, the pointer should be saved so the expression need not be recompiled.

String compiling sets system-reserved words opcnt, varcnt, and formok.

SPECIAL INFORMATION

The –clock– instruction stores the time, to the nearest second, in the variable named in the tag. The author can use the –showa– instruction to display the time, which is in alphanumeric form hr.min.sec, based on the 24-hour clock. For example, the time stored at 2:15 PM is 14.15.08. The system-reserved word clock contains the current time in numeric form (in seconds since the PLATO system was loaded that day) and is accurate to 0.001 second (1 millisecond).

The –date– instruction places the current date, in alphanumeric form, in the variable named in the tag. Ten characters are used, and the format is month/day/year with a preceding and a trailing blank. For example, on June 13, 1975, the instructions

```
date    n1
showa   n1
```

would display:

```
06/13/75
```

The –day– instruction gives the number of days since the starting date and time, as set by the installation. As a result, the value depends on the installation in which the system is running. The value is in numeric, not alphanumeric form, and gives both the number of whole days, and as a fractional part, the portion of a day (accurate to the nearest 0.1 second) since the value was initialized by the installation. Because of the fractional part, the variable named in the tag must be a floating-point variable, not an integer variable.

The other special information instructions deal with information about the student, course, and lessons.

The –name– instruction puts the name under which the student signed on in two consecutive variables, with the tag giving the name of the first of these variables. Two variables are necessary since the student's sign-on name can be up to 18 characters long. The first 18 characters of the two variables contain the name, left-justified and zero-filled.

The –group– instruction performs the equivalent function for the student's group. The group name, however, must be no more than eight characters in length, so only one variable is used. As with the –name– instruction, the name is left-justified in the variable, with the remainder set to zero.

The –from– instruction gives the lesson or lesson and unit from which the student entered the current lesson.

The unit name used is the name of the last main unit that was executed in the lesson from which the student entered. Thus, if an auxiliary unit contained the –jumpout– instruction from which the student entered the current lesson, that unit is not given as the unit from which the student came. Rather, the main unit that used the auxiliary unit is given (refer to section 10 for a full discussion of unit types).

A student entering the lesson immediately after signing on to the system, rather than entering from another lesson, is –from– lesson "plato".

A student can be returned to the lesson from which he entered by use of instructions such as:

```
from      n10;les1;les2
jumpout   n10;x;les1;les2
```

If the student came from les1, he is returned to les1, if he came from les2, he is returned to les2, and so on.

The 2-argument form of the –from– instruction places the name of the lesson and the name of the unit from which the student entered the current lesson into the variables specified in the tag. The system-reserved words zfroml and zfromu also contain the lesson and unit names, regardless if the lesson executes –from–. Executing –inhibit from– in the lesson prior to a –jumpout– to another lesson inhibits both the –from– instruction and the system-reserved words zfroml and zfromu from containing information referencing that lesson.

The –lessin– instruction is used to check if the lesson named in the tag (either in a variable or as a literal) is credited to the logical site. After the instruction is executed, the result is given in the system-reserved word zreturn. The value of zreturn is -1 if the lesson named is in ECS and is in use by someone in the student's logical site. The value is 0 otherwise. This allows a check before a –jumpout– instruction to determine if the lesson to which the student is sent is already condensed and available, or if it must be brought into the system and condensed before the student can execute it.

The –in– instruction is used to indicate whether a particular station is executing the current lesson. The tag (variable or expression) for the instruction specifies a station number (0 to 1023), with the current station number obtained with the system-reserved word station.

Following execution of the –in– instruction, the system-reserved word zreturn has one of the following values.

Value	Meaning
-2	Station was routed by current lesson (routers only)
-1	Station is in your lesson
0	Station is not in your lesson

The system-reserved word usersin contains the number of students currently in the lesson. Students routed to another lesson (via a router lesson) are considered to be in the router lesson and the current instructional lesson.

RANDOM NUMBERS

There are two methods of accessing random numbers in the PLATO author language, sampling with replacement and sampling without replacement.

Actually, the numbers generated in either method are not truly random but are termed psuedo-random. The numbers are generated by the computer and cannot be

truly random since, by definition, there is a mechanical (logical) sequence that generates the numbers. However, the difference on the practical level is inconsequential.

Sampling with replacement means that the values are treated as if they were returned to the pool of values from which they were drawn. Thus, if the value 3 were returned in one sample, the value 3 could occur again in another sample. In fact, the value could occur in the next sample.

Sampling with replacement is done in the author language with the –randu– instruction. The tag has either one or two arguments, with the first (or sole) argument being a variable name. This is the variable in which the value is stored. It must be a floating-point variable if the one-argument tag is used.

The one-argument tag returns a fractional number between 0 and 1. A two-argument tag returns an integer between 1 and the value of the second argument, inclusive.

Sampling without replacement means that once a value has occurred, it does not occur again. The author language does sampling without replacement on permutations of integers. The numbers are from 1 to the limit specified in the –setperm– instruction, inclusive. There are five instructions dealing with these permutations. They provide two methods of sampling, using the system permutation locations or using locations in the variable bank specified by the author.

The –setperm– instruction sets up the permutation, with the first (or only) argument specifying the limit. If the single-argument tag is used, the system location for the permutation is used. This consists of two locations of three words each. The first location is the working copy of the permutation. The second copy is not changed by ordinary sampling and can be used to regenerate the permutation. The first word of each location contains the number of elements remaining. Each bit of the other two words signals a single element. Thus, permutations with limits up to and including 120 can use the system locations.

If the two-argument form of the –setperm– instruction is used, the second argument specifies a starting location of contiguous words in the user bank variables that are used for holding the permutation. These words contain one word (the first) that specifies the number of elements remaining in the permutation and one word for every 60 elements in the permutation. If the number of elements is not an even multiple of 60, the number of words required is the same as for the next larger multiple of 60. Thus, a permutation with 50 elements requires two words, a permutation of 60 elements requires two words, and a permutation of 70 elements requires three words (the same as a permutation of 120 elements). The words following the first word contain flag bits that indicate whether an element has been sampled. The bit for each element is 1 if the element is still available for sampling and 0 if the element is not available. Since the location is specified, more than one permutation can be set up, so sampling from different permutations can be done. A second copy is not generated; if the author desires such a copy, he must generate it himself. The –block– instruction (described later in this section) can be used to create the second copy.

The –randp– instruction is used to sample the permutation previously specified. Again, two forms are available, with the single-argument form referring to the system location for a permutation and the two-argument form referring to a permutation located in the bank of variables.

The first (or only) argument specifies the variable in which the element obtained is to be stored. If the one-argument tag is used, this is obtained from system permutation location using the first copy of the permutation only. If the two-argument tag is used, the permutation starting in the user variable specified in the second argument is used to obtain the element. In either case, the corresponding bit is set to zero, so the element is not selected again. If the system permutation location is used, only the first copy of the permutation is affected. The second system copy is left unchanged for purposes of regenerating the permutation. When the permutation is exhausted, a value of 0 is returned.

The –remove– instruction is used to modify the second system copy. The element contained in the variable specified in the first argument is removed from the permutation.

If the system location is used, a single argument is sufficient. If a second copy of a permutation in an author-specified location is to be modified, the second argument gives the starting location in the student variables of the second copy. The –remove– instruction differs from the –randp– instruction in that it simply removes a specified element from consideration for selection, while the –randp– instruction selects a new element and then eliminates it from further consideration.

The –modperm– instruction is used only with the system location for a permutation and has a blank tag. The effect is to replace the first copy of the permutation (accessed and modified by the –randp– instruction) with the second copy (which can be modified by the –remove– instruction).

An example of the advantage of having two copies of the permutation is when the author wishes to give a number of problems in random order, requiring the student to retry the problems initially answered incorrectly. The problems are selected using the –randp– instruction from the first copy of the permutation. If the student answers correctly, the corresponding value is deleted from the second copy with the –remove– instruction. When the first copy is empty (that is, all problems have been tried), the –modperm– instruction moves the second copy, containing only the numbers of problems originally answered incorrectly, into the first copy so that the –randp– instruction can be used to reselect from among these numbers for representing the problems to the student, again in random order.

The –seed– instruction is used with –randu– and –randp– to specify the location of the variable used in beginning the generation of a series of random numbers. If there is no –seed– instruction or if the –seed– instruction has no tag, the system seed is used. Pseudo-random numbers are generated by the system using an algorithm which needs an initial number, a seed, to begin. Thus, it is possible to repeat a sequence of pseudo-random numbers using the –seed– instruction; however, the algorithm used by the system is subject to change, which will also change the sequence of random numbers initiated by a specific seed.

BRANCHING AND LOOPING WITHIN A UNIT

It is possible to do branching and looping using units or –entry– instructions as the point to which the lesson branches; however, such construction is clumsy and slow, and it can interfere with the screen display unless the author takes repetitive and tedious precautions. Four types of instructions are available for branches and loops within a unit. These are the –branch– instruction, the –if– structure, the –doto– instruction, and the –loop– structure.

BRANCHING WITHIN A -CALC- INSTRUCTION

Branching can be done within a single –calc– instruction with a multiple line tag (that is, a continued –calc–). A line to go to must be specified and identified by a name in the command field. To distinguish the name from ordinary author language instructions, the name must begin with a numeric rather than an alphabetic character.

The branching is done with the –branch– instruction. This instruction is not necessarily in normal form but can have the command portion begin in the first character space of the tag field in a line of the –calc– instruction. The command portion consists of the character-string branch. After the command portion, a space should be left, and the label (name) of the line to which the branch is to take place is then given. Alternatively, a conditional form is available, operating in the normal manner except for the relocation of the fields.

As an example of the use of the –branch– instruction, consider the problem of computing the factorial of a number. The instructions

```
calc    v2⇐ v3⇐ 1
10      v2⇐ v2+1
        branch v2> v1,20,x
        v3⇐ v3*v2
        branch 10
20
```

compute the factorial of the contents of v1 and place the result in v3.

This can also be written as:

```
calc        v2⇐ v3⇐ 1
10          v2⇐ v2+1
branch      v2>v1,20,x
            v3⇐ v3*v2
branch      10
20
```

The x in the tag part of the –branch– instruction in the example causes processing to fall through the branch and continue executing the –calc– instruction as though the –branch– had not been encountered. A tag of q in a –branch– causes all calculation to halt when the conditional selects the q. Processing then continues at the first instruction following the –calc– instruction containing the –branch–, or processing halts and waits for a control key, if the –calc– instruction was the last instruction in a main unit.

A –calc– instruction with a branch containing a q in the tag portion should not be followed by another –calc– instruction without other intervening instructions. If the q is selected, a –calc– instruction immediately following is ignored. Therefore, the second –calc– should be part of the first, with an appropriate branching. For example, the instructions

```
calc    .
        .
        .
branch  v2>v1,q,20
20      .
        .
        .
calc    v2⇐ v1
```

should be replaced by the single instruction:

```
calc    .
        .
        .
branch  v2>v1,30,20
20      .
        .
        .
30      v2⇐ v1
```

If a –branch– instruction is not part of a continued –calc–, it begins a –calc– sequence. It can occur in the command field whether it is part of a continued –calc– or not.

The –branch– instruction can be used to branch around non–calc– instructions within a single unit (refer to section 10).

-IF- STRUCTURE

The –if– structure is useful when the author wants the lesson to do different actions based on a logical or arithmetic expression. Four instructions form the –if– structure: –if–, –elseif–, –else–, and –endif–, in that order. The –if– and –endif– instructions are required; the –elseif– and –else– instructions are optional. The –if– structure usually executes faster than an equivalent –branch– structure.

The tags of the –if– and –elseif– instructions are logical or arithmetic expressions. Logical expressions have two values: true (-1) and false (0). Arithmetic expressions are evaluated and rounded before testing. If the expression is less than 0, it takes the value true. If the expression is greater than or equal to 0, it takes the value false. When the tag of an –if– or –elseif– instruction is evaluated as true, the author language instructions immediately following it are executed up to the next –elseif– or –else– instruction. Execution then goes directly to the instruction following the –endif– instruction, leaving the –if– structure. The –else– instruction always has the value true.

The –if–, –elseif–, and –else– instructions must be followed by at least one line of indented author language instructions. The indented instructions can be any regular instructions. They cannot be judging instructions. To indent, press ACCESS period (.), (or type a period, then

TAB) and start the command of the instruction in the ninth space. The tag of the instruction starts in the sixteenth space. The period followed by 7 spaces is an implied then. (If the tag is true, then the next instruction is executed.) A simple –if– structure is

```
if      n1≤n2
.       box     1244;1550;5
else
.       box     2240;2550;4
endif
```

In this example, the first box is drawn when n1<n2 and the second box is drawn when n1>n2.

An –if– structure can be nested within another –if– structure. A nested –if– can be inserted anywhere in an outer –if–; however, the entire nested –if– must be between the two outer –if– statements. Each nested instruction has a period followed by 7 spaces for each level of indenting. There is no limit to the number of levels of indenting as long as each instruction fits on one line.

The following is an example of a nested –if– structure.

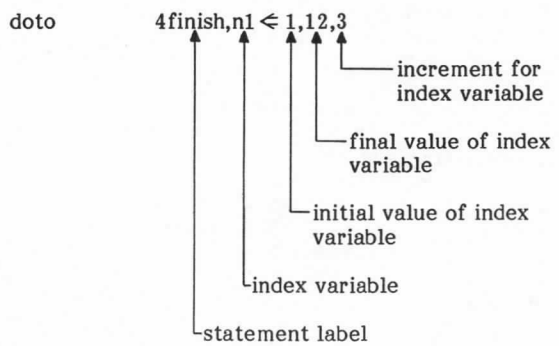
```
if      n1≤10
.       box     1244;1550;5
elseif  n1≤20
.       box     2240;2550;4
*       $$if 10<n1≤20
.       if      n1=15
.       .       erase   20
.       else
.       .       box     2344;2446
.       endif
else
.       box     0235;0550;3
*       $$if n1>20
endif
```

An –if– structure can be nested within a –doto– loop or a –loop– structure, or it can nest a –doto– loop or a –loop– structure within itself.

-DOTO- INSTRUCTION

The –doto– instruction allows an author to have an iterative loop within the same unit. The –doto– loop extends from the –doto– to the statement label named in the tag of the –doto– instruction.

The format of the –doto– instruction is analogous to the format of the iterative –do–.



The statement label must begin with a number and contain seven or less characters (no shift, access, or fonts). The initial and final values of the index variable may be constants, variables, or expressions. The increment for the index variable may be omitted if it equals 1. A negative increment for the index variable is permitted.

For example, the following instructions

```
doto    3finish,n1 ⇐ 1,12,3
        area ⇐area+n1^2
        circ ⇐ circ+4*n1
3finish         $$must have a blank tag
at      1513
write   the sum of the area is ⊲s,area⊳
        the sum of the circumferences is
        ⊲s,circ⊳
```

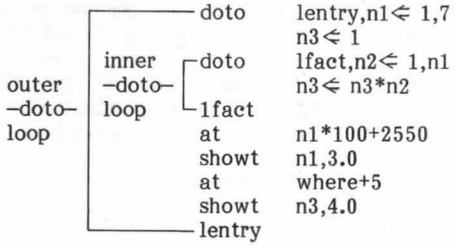
perform the following.

The –doto– loop is executed four times with n1 equal to 1, 4, 7, and 10. When the current value of the index variable (plus the increment) is greater than the final value of the index variable, as specified in the tag of the –doto– instruction, the loop is completed and the instruction following the statement label is executed. In this example, the –at 1513– instruction is executed. After the loop, the value of the index variable is undefined.

When using the –doto– instruction, the following considerations should be noted.

- A –doto– in the command field initiates a –calc– function; if the –doto– begins in the tag field, it continues a –calc–.
- Non –calc– instructions are allowed within a –doto– loop.
- The statement label ending the –doto– may not contain a –calc– expression.
- If a –doto– statement continues a –calc– instruction, it can be moved to the tag field. When the –doto– is in the tag field, at least one space must be left between the doto and the statement label named in the tag of the –doto– instruction.
- The –doto– instruction can be used only in the iterative form. The –branch– instruction should be used for conditional and unconditional branches.
- The statement label of the –doto– loop must be in the same unit as the –doto– statement and must not be referenced by any other branching instructions in that unit. The same statement label can be used in different units because the –doto– (or –branch–) loops operate only within units.
- An –entry– instruction within a –doto– loop is illegal.

It is also possible to have –doto– loops nested within a –doto– loop. For example, the following instructions



construct a table of numbers and their factorials. The outer loop is executed seven times, and the inner loop is executed n1 times (n1 is the outer loop index and goes from 1 to 7) as part of each outer loop.

The following list of factorials (N!) is produced from the above -doto- loops.

N	N!
1	1
2	2
3	6
4	24
5	120
6	720
7	5040

When nesting -doto- loops, the inner loop cannot extend beyond the statement label of the outer loop.

The -branch- instruction can be used in a -doto- loop to:

- Branch within loops
- Branch to the end of -doto- loops
- Branch out of a -doto- loop
- Fall through a -branch- with an x
- Branch to the next non-calc- instruction with q

It is possible to branch into a -doto- loop; however, this should not be done because the current value of the index variable may contain an unexpected value, resulting in an unexpected number of iterations of the -doto- loop. If the value of the index variable is assigned prior to the entry into the loop, the results are predictable.

LOOPING

The -loop- structure allows an author to have a loop based on logical or arithmetic expressions. Four instructions form the -loop- structure: -loop-, -reloop-, -outloop-, and -endloop-. The -loop- and -endloop- instructions are required; the -reloop- and -outloop- instructions are optional. Unlike the -doto- instruction, the -loop- instruction does not initiate or continue a -calc- instruction. The -loop- structure usually executes faster than an equivalent -branch- structure.

The tags of the -loop-, -reloop-, and -outloop- instructions are logical or arithmetic expressions. Refer to the -if- structure for comments on logical and arithmetic expressions. The tag of the -loop- instruction is evaluated at the beginning of each pass through the loop. When the tag of the -loop- instruction is evaluated as true, the author language instructions immediately following it are executed. If it is evaluated as false, execution resumes following the -endloop- instruction. When execution reaches the -endloop- instruction from within the loop, the loop is executed again beginning at the -loop- instruction.

The author language instructions within the loop must be indented. These instructions can be any regular instructions. They cannot be judging instructions. Refer to the -if- structure for directions on indenting.

The following loop moves data from array temp to array final as long as n1 is less than 20. The loop begins with the -loop- instruction and ends with the -endloop- instruction.

```
define   array,final(20)=n40
         array,temp(20)=n100
calc     n1 ⇐ 0
         n2 ⇐ 1
loop     n1 < 20
.        calc     n1 ⇐ n1+1
.                 final(n2) ⇐ temp(n1)
.                 n2 ⇐ n2+1
endloop
```

The -reloop- instruction allows execution to go back to the beginning of the loop without going all the way to the -endloop- instruction. If the tag of the -reloop- instruction is true, execution goes back to the -loop- instruction; otherwise, it continues with the next line. Using the same array definitions and adding the -reloop- instruction, the previous loop now also deletes entries which are 0.

```
calc     n1 ⇐ 0
         n2 ⇐ 1
loop     n2 < 20
.        calc     n1 ⇐ n1+1
reloop   temp(n1)=0
*        $$do not move a zero entry
.        calc     final(n2) ⇐ temp(n1)
.                 n2 ⇐ n2+1
endloop
```

The -outloop- instruction allows execution to jump out of the loop. If the tag of the -outloop- instruction is true, execution goes to the instruction following the -endloop- instruction; otherwise, it continues with the next line. Using the same array definitions and adding the -outloop- instruction, the previous loop now also sums the entries and stops moving the data if the total would exceed 100.

```
calc     n1 ⇐ 0
         n2 ⇐ 1
         n3 ⇐ 0
loop     n1 < 20
.        calc     n1 ⇐ n1+1
reloop   temp(n1)=0
outloop  (n3+temp(n1)) > 100
*        $$exit if sum > 100
.        calc     final(n2) ⇐ temp(n1)
.                 n3 ⇐ n3+temp(n1)
*                 $$add to sum
.                 n2 ⇐ n2+1
endloop
```

The –reloop– and –outloop– instructions can appear anywhere in a loop, and each can appear more than once in the same loop.

A loop can be nested anywhere within another loop; however, the entire nested loop must be between the two outer –loop– instructions. Each nested instruction has a period followed by seven spaces for each level of indenting. There is no limit to the number of levels of indenting as long as each instruction fits on one line. A loop can be nested within an –if– structure or a –doto– loop, or it can nest an –if– structure or a –doto– loop within itself.

-CALCS- INSTRUCTION

There are two instructions that perform calculations in a conditional form. The –calcc– instruction functions in the manner normal to conditional instructions and is the counterpart of the –writec– instruction. Both of them are dealt with in section 12. The other conditional calculation instruction is the –calcs– instruction.

The –calcs– instruction is used to perform one of a number of variable assignments, depending on the value of an expression. The variable to which a value is to be assigned must be the same in all cases. Other than the fact that it gives an assignment of value based on the expression, it functions in the same manner as other conditional instructions. Up to 61 specific values can be entered in the –calcs– and –calcc– instructions. An example of the use of the –calcs– instruction is:

```
calcs        3+filk,grun ≤ 17,filk^2,grun/2
```

COMMON

Common is a type of variable in which users can store information. In contrast to the 150 student variables, common stores information which is common to all users of a lesson. The system keeps only one copy of the common variables for each lesson, regardless of the number of users executing the lesson concurrently. Several users can read from or can write to the same common locations, allowing one user to pass information to another user when both are executing the lesson. When a lesson is executing, a copy of that lesson is in ECS. Also, any common that the lesson accesses is in ECS.

TYPES OF COMMON, -COMMON-, AND -COMMONX-

Two types of common are available, temporary common and permanent common.

Temporary common is in ECS while the lesson is in execution, but it is not saved on disk between sessions. This type of common is specified by the –common– instruction with a one-argument tag. The tag indicates the number of variables that are to be used as temporary common. As an example, the instruction

```
common      257
```

specifies that common is a block of 257 variables, numbered from 1 to 257. The tag may be a constant or an expression but not a variable.

Permanent common is on disk and is saved between sessions. The system creates an ECS copy of the common whenever it executes a lesson which uses the common. This type of common is useful for storing statistics about the lesson, such as the number of times the lesson has been executed or for functioning as an extension to the standard variable bank through author manipulation of variable assignments and use of the –comload– instruction.

Permanent common is formed by specifying the space for the variables as a portion of a lesson. The lesson containing the common need not be the same lesson as that using the common, providing the two lessons have the same common access code. For each set of 320 or fewer words, a new block is specified. This block should be placed at the end of the lesson. The block should be specified as a common block upon block creation, and all of the common blocks should be given the same name.

As an example of the creation of permanent common, consider the following example: lesson vrap requires 410 words of permanent common. However, all possible blocks of the lesson are in use. If lesson vlunge is available to the author and has the same common access code as lesson vrap, two common blocks (since the number of variables necessary is greater than a single block, but smaller than two blocks) can be created at the end of lesson vlunge, and both blocks are given the name scrounge. The required permanent common can then be accessed by lesson vrap with the instruction:

```
common      vlunge,scrounge,410
```

As implied by the example, permanent common is accessed by means of a –common– instruction with a three-argument or four-argument tag. The first argument is the lesson containing the common blocks. If the referenced common blocks are in the lesson that uses them, this argument is optional. In either case, a comma must appear before the second argument. The second argument is the name of the common blocks, and the third is the number of words contained in the common. As with the one-argument tag, the number of variables (indicated by the third argument) may be a constant or an expression, but not a variable. All lessons which reference the same common must have the same length of common (third argument) defined in their –common–s. The –common– instruction allows a maximum of 8000 words for actual use.

There are four additional options with the –common– instruction. If one of these options is desired, it is specified as a fourth argument to the –common– instruction. The first option is specified by having the last argument consist of the two words no load. This prevents normal loading and unloading of common, so the –comload– instruction must be used by the author to specify these actions. If the last argument consists of the two words read only, changes to common are not allowed. If the last argument is the four characters ronl, both the no-load and the read-only options are used. The fourth option is checkpt which returns permanent common to disk approximately every 8 minutes. The checkpt option ensures system recovery after a system failure. (Refer also to the –comret– instruction.)

Only one –common– instruction is permitted in a lesson. The IEU is a convenient place to put this nonexecutable instruction.

The –commonx– instruction is an executable form of the –common– instruction. The –commonx– instruction allows a lesson to determine what common it needs after it condenses, using information that is not available until the lesson begins execution.

The –commonx– instruction can have up to five arguments. The first argument is the lesson containing the common blocks. If the referenced common blocks are in the lesson that uses them, this argument is optional. In either case, a comma must appear before the second argument. The second argument is the name of the common blocks, and the third is the number of variables contained in the common.

The fourth argument is the codeword of the common. If this codeword is present, the system compares it with the common codeword of the lesson that contains the common. If this codeword is not present, the system compares the common codeword of the lesson containing the –commonx– instruction with the common codeword of the lesson that contains the common. Unlike some instructions, if the codeword is present but does not match the common codeword of the lesson that contains the common, the system does not check the common codewords of the lessons against each other.

The fifth argument can be one of four options which are the same as the four options in the –common– instruction. This argument can also be blank.

The –commonx– instruction makes heavy demands on the system, and therefore, is restricted as follows:

- A user can execute one –commonx– instruction per lesson.
- After a user executes three –commonx– instructions (resulting from jumpouts), the system limits execution of the –commonx– instruction to one per minute.
- When a student leaves a router lesson, the router releases common declared by a –commonx– instruction. This means it is not possible to allow the reading of common variables with the –allow– instruction in a router lesson.

After execution of the –commonx– instruction, the system-reserved word zreturn is set to the following values:

Value	Meaning
-1	Common ok
0	Common not found
1	Code words do not match
2	Already have a common
3	Common in ECS has different length
4	Bad length

USING COMMON

The author usually does not reference common locations directly. Instead, the author accesses a portion of the common (up to 1500 words) using the nc and vc variables and the –comload– instruction. The –comload– instruction transfers a portion of ECS common to the nc and vc variables at the beginning of each time-slice. At the end of the time-slice the contents are transferred back from the nc and vc variables to the ECS common. Therefore, the author can refer to nc and vc variables without regard to when time-slices occur during lesson execution.

The –comload– instruction has three arguments in its tag. The first is the starting position in nc or vc variables; the second is the starting position of the block of variables in the ECS copy of the entire set of common variables; and the third specifies the number of variables in the block to be loaded and unloaded. As an example, the instruction

```
comload     nc53,40,10
```

transfers the contents of the 10 variables starting with position 40 in the ECS copy of common variables to the 10 variables starting with nc or vc variable 53 at the beginning of each time-slice. At the end of each time-slice, the process is reversed. This occurs for each student, individually, although all students share the same variables.

If a –comload– instruction with a blank tag is encountered during the lesson, the common variables are unloaded as if the time-slice has expired. No further loading or unloading of common variables is done until or unless another –comload– instruction is encountered.

If a second –comload– instruction is encountered while the first is still in effect, the variables specified in the first –comload– are unloaded before the second –comload– instruction is executed.

An error is generated if an attempt is made to –comload– into the user variable bank, as in the following instruction.

```
comload     v14,12,10
```

The –comload– instruction can also be used to set aside specific portions of common for individual students, while allowing the author to use simple common variable references. This is possible because the second argument in the tag of the –comload– instruction can be an expression. In particular, it can be an expression based on a student identification. Thus, if ID represents the student identification, the instruction

```
comload     nc1,500+200*ID,200
```

effectively gives each student his own copy of 200 variables, located in the common variable area, which is as fully individualized as the standard variable bank (providing, of course, that no other –comload– instructions or references to these variables are made). For example, at the same time, the author can deal with variable nc1 rather than having to use the cumbersome expression nc(500+200*ID).

Up to three separate automatic loads and unloads can be done for each time-slice by using a continued –comload– instruction. Each tag line consists of one normal

–comload– tag. The command comload appears only on the first line. An example is the following instruction.

```
comload     nc24,10,5
            nc30,50,12
            nc48,293,300
```

This is not the same as:

```
comload     nc24,10,5
comload     nc30,50,12
comload     nc48,293,300
```

These three instructions (not a single instruction) perform three loads and two unloads. Only the third –comload– is in effect thereafter. In the first example, all three blocks are loaded and unloaded each time-slice.

If the common has a length of 1500 words or less, the author does not need to include a –comload– instruction in the lesson, because the system inserts a default –comload– at the beginning of the lesson. For example, if a lesson has a common named mycom of length 650, the following –comload– is automatically in effect at the beginning of the lesson, although it does not appear in the lesson source code.

```
comload     nc1,1,650
```

The author can prevent the execution of this default –comload– with the no load option of the –common– instruction.

The –comret– instruction copies the current permanent common from ECS to disk. Execution of the –comret– instruction has no effect on the ECS copy of common; therefore, –comret– cannot delete common from ECS. The –comret– instruction has a blank tag. After execution of the –comret– instruction, the system-reserved word zreturn is set to the following values:

Value	Meaning
-1	Common returned
0	No common found
1	Unable to return common

In a –finish– unit, the execution of a –comret– instruction counts as one of the 10 disk accesses allowed.

The –comret– instruction is not needed when the number of users of common drops to zero, because the system automatically copies common from ECS to disk at that time. The –comret– instruction is useful only when the author wants to preserve ECS common on disk in its state at the moment –comret– is executed.

-RESERVE- AND -RELEASE- INSTRUCTIONS

The –reserve– and –release– instructions make the protection of common and dataset records easier. Both instructions have keyword tags of 'common', 'dataset', and 'records'.

The –reserve– instruction sets a flag to indicate that this terminal is using common or dataset records. The flag is not set if common or dataset records are already –reserve–d by another terminal. The system-reserved word zreturn is set to –1 if the –reserve common– and –reserve dataset– instructions have executed successfully.

The –release– instruction clears the flag set by the –reserve– instruction. The flag does not clear if this terminal does not have common or dataset records –reserve–d. The system-reserved word zreturn is set to –1 if the –reserve– instruction has executed successfully.

-ABORT- INSTRUCTION

The –abort– instruction is available in three forms.

```
abort       common
abort       record
abort       autocheck
```

Up to three keyword tags also may be used with a single –abort– instruction.

```
abort       common,record,autocheck
```

The –abort common– instruction causes permanent common not to be returned to the disk (updated) when the last user leaves the student mode. The permanent common, in effect, is transformed into a temporary common. The lesson(s) containing common continue to function as before, but all recent changes to common are never returned to the disk.

Student records are not returned to the disk when the student signs off as a result of an –abort record– instruction. However, the CPU time, the number of sessions, and so on, are updated at sign-off time. This option requires more access time, and for that reason, should not be used when CPU time is at a premium.

Autocheckpoint (automatic return of records) is included as a function of the –abort record– instruction.

The automatic return of records does not occur when the –abort autocheck– instruction is in effect.

An execution error occurs if a user is not registered as a student and the –abort record– or –abort autocheck– instruction is in effect.

STORAGE

Storage is essentially a variety of temporary common, in that the contents of storage cannot be preserved between sessions unless some special action is taken (refer to datasets). An author cannot reference storage locations directly.

-STORAGE- INSTRUCTION

The –storage– instruction, which creates storage, is comparable to the –common– instruction used to create

temporary common. An example is the following instruction.

```
storage     400
```

This instruction creates 400 storage variables for use by the lesson. Unlike common variables where there is only one copy of the variables per lesson, the storage is specific to the user in the same manner as the student variables. As a result, some caution should be exercised in the amount of storage used. The -storage- instruction in the example uses an extra 400 words of ECS for each person executing the lesson. There is a limit on the amount of ECS a logical site is allowed, so storage can use up the ECS allotment rather quickly.

The maximum length of storage available to a user (that is, the maximum size that can be specified in a -storage- instruction) is 1500 words.

The -storage- instruction can only be executed once in a lesson, much like the -common- instruction. Therefore, it is not possible to destroy the storage if it is no longer needed.

-STOLOAD- INSTRUCTION

In order to use storage, there must be a mechanism for bringing the storage variables to a location where they can be referenced. This is done with the -stoload- instruction.

The -stoload- instruction is similar to the -comload- instruction, but it references storage locations rather than locations in the ECS copy of common. The format of the instruction is precisely the same, and the -stoload- also brings the storage into the nc and vc variables. As a result, if both -stoload- and -comload- instructions are used in a lesson, the author must make sure that they do not load the variable values into the same nc and vc variables. If storage and common locations overlap in the nc and vc variables, the storage variables overwrite the common variables, because -comload- executes before -stoload- at the beginning of each timeslice.

All of the capabilities and restrictions of the -comload- apply to the -stoload-. This includes the behavior when a second -stoload- is executed and the fact that a -stoload- with no tag cancels loading and unloading of variables.

DATASET FILES

A means for saving information, other than permanent common, is available to the author. This method uses another type of file as an auxiliary to the lesson. This auxiliary file is a dataset.

A dataset is not the same as a student datafile. A student datafile stores lesson execution data. The data is collected automatically by the PLATO system, with the author simply specifying the data to collect. A dataset is under complete control of the author (with the exception of the first block of the file). It is more like permanent common than a student datafile.

It is not possible to execute a dataset. The file is used entirely to store information, and as such, is not considered to contain author language instructions.

All transfer of information to or from a dataset is done by records; that is, a minimum of one record of data can be transferred, and the data transferred is an integer number of contiguous records. The number of records contained in a dataset depends on the number of words per record and the length of the dataset. The number of words per record is determined when the dataset is created and must be between 64 and 512 words, with the default as 320 words.

A maximum of 10 dataset accesses, including the -dataset- instruction may be executed in a finish unit. Because of code word checks, the -dataset- instruction counts as two accesses.

-DATASET- INSTRUCTION

Before a lesson can use a dataset, the system must be informed of the name of the dataset. The -dataset- instruction gives the name to the system. The -dataset- instruction performs a function similar to that performed by the -readset- instruction (refer to section 13) except that a dataset, rather than a student datafile, is the file processed.

The tag of the -dataset- instruction may have three arguments. The first argument must be the name of the dataset. The second argument (optional) is the type of access desired, read/write or read only. The third argument (also optional) is the code word. If the code word is specified, it must match the dataset common code word for read/write access, or it must match the dataset inspect code word for read only access. If these code words do not match, the system-reserved word zreturn is set to 1 and the dataset is not attached. If no code word is specified, the code words of the dataset are compared to the code words of the lesson in which the -dataset- instruction occurs. If the common code words match, either read/write or read only access is granted, and if only the inspect code words match, read only access is granted, if requested. If these code words do not match, the system-reserved word zreturn is set to 1 and the dataset is not attached.

The -dataset- instruction sets system-reserved word zreturn to -1 if the link is successful. The author can check zreturn to ensure that the file has been located.

Since -dataset- is an executable instruction, the link to the named dataset is effective until another -dataset- instruction is executed. For operations such as copying from one dataset to another, the author can keep records reserved in two datasets at the same time: the current active dataset and an inactive dataset. If a second -dataset- instruction is executed while a previous dataset is still attached, the first dataset becomes inactive with all its record reservations preserved. The author can reactivate the first dataset with its record reservations intact by reexecuting the -dataset- instruction. The system can hold only one dataset in the inactive state. If a third -dataset- instruction is executed, the link to the current inactive dataset is closed and all its reservations are released. A -dataset- instruction with a blank tag

closes the current dataset without preserving its record reservations and without affecting the current inactive dataset, if any. A SHIFT STOP exit or a –jumpout– releases all datasets unless an –inhibit dropset– is in effect.

-DATAIN- INSTRUCTION

Data is transferred from the dataset to storage, common, or student variables with the –datain– instruction. The instruction has two or three arguments in the tag.

The first argument is the number of the first record to be transferred. The second argument gives the location in storage, ECS common, or student variables to which the record is to be transferred. For example, the fourth word of storage is formatted as storage,4 or s,4. The 90th word of ECS common is formatted as common,90 or c,90. Student variables are formatted as n15 or n24. CM nc and vc variables cannot be used.

The optional third argument gives the number of records to be transferred. The total size of the records to be transferred cannot exceed the amount of space available following the beginning location given by the second argument. Hence, if the size of storage is 500 words, two 320-word records cannot be read in with a single instruction, regardless of the starting position of the first record. Similarly, if the size of common is 1000 words and the first record begins at location 100, the maximum number of 320-word records that can be transferred is two, rather than the three that can be transferred if the starting location in storage is word 1. If a third argument is not specified, a value of 1 is assumed.

As an example of the –datain– instruction, the instruction

```
datain      5;n2;2
```

reads records 5 and 6 of the dataset into student variables, beginning with variable 2.

After a –datain– instruction, the system-reserved word zreturn is set to the following values.

Value	Meaning
-1	Transfer successful
0	System pack error
1	System file error
2	Record numbers are out of range
3	Addresses (storage, common, or student variable) are out of range
6	System disk error

-DATAOUT- INSTRUCTION

Data is sent to the dataset from storage, ECS common, or student variables with the –dataout– instruction. The format of the instruction is precisely the same as the format of the –datain– instruction; however, instead of reading in the data from the dataset, the data is written out to the dataset.

As an example, the instruction

```
dataout     2;s,30;1
```

transfers the contents of storage locations 30 to (30 plus record length) to the second record of the dataset file.

The values for system-reserved word zreturn are the same as the –datain– instruction with these additions.

Value	Meaning
4	No write permission
5	Record is –reserve–d or dataset is being edited

NAMESET FILES

A nameset is a dataset that contains sets of records. Each set has an alphanumeric name and some records for storing data. Namesets are useful for the storage and retrieval of easily-categorized data because each set of records can be one set of information.

Namesets are easier to use than datasets for categorizing information because all the instructions dealing with records reference the records relative to the name of the set. The nameset keeps track of all its record pointers so that it automatically finds information in the nameset when given a name and a record number.

STRUCTURE

A nameset contains sets of names and records. Each set has an alphanumeric name of up to 30 characters. After the name, each set has a computer word of associated information. In the first 15 bits of this word, the nameset stores the number of records associated with the name. In the last 24 bits, the user can store any alphanumeric information associated with the name. In addition to the name and the associated information, each set has a variable number of records. Figure 8-1 shows a nameset with three names (three sets).

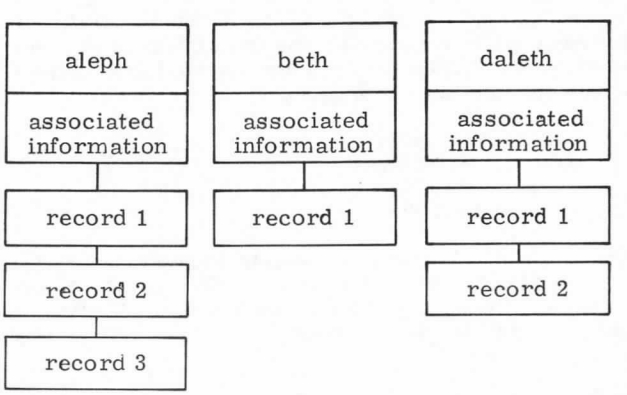


Figure 8-1. Example of a Nameset

A nameset has a directory that stores the names and the associated information of each set. The system reads this directory into ECS during execution of the –dataset– instruction referencing the nameset and keeps the directory in ECS throughout the use of the nameset. The system accesses information in the directory quickly and efficiently.

At creation of a nameset, the author must specify the parameters of the nameset. The author can set the maximum length of the names in a nameset between 10 and 30 characters. (The actual length can be less.) The record size can be between 64 and 512 words. The number of parts in a nameset can be between 1 and 63. (The number of words per part is 7 X 320.) The author also must specify the maximum number of names in a dataset. More names means fewer records are available for each name; conversely, fewer names means more records are available for each name.

DATASET INSTRUCTIONS USED WITH NAMESETS

Before using a nameset, the author must establish a connection to the nameset with the –dataset– instruction. The tag of the –dataset– instruction is the name of the nameset. After execution of the –dataset– instruction, the system-reserved word zreturn is set to the same values that are used for connections with a dataset. These values are:

Value	Meaning
-1	Connection made
0	No such nameset
1	Codewords do not match
2	File directory being edited
≥3	System disk errors

The system-reserved word znscpn tells if the current connection is with a nameset (value is nonzero) or with a dataset (value is 0).

After the –dataset– instruction connects a nameset, the lesson can manipulate the records in a nameset and the information in the records. The –datain– instruction reads data from the nameset, and the –dataout– instruction writes data into the nameset. These instructions function for namesets in the same way as for datasets except that references to nameset records are relative to the current named set of records. For example:

```
dataset  hebrewset
calc     n20 ⇐ 'daleth'
setname  n20
datain   1;c,100;2
```

The –dataset– instruction connects nameset hebrewset. The –setname– instruction accesses the set of records named daleth, and the –datain– instruction reads records 1 and 2 of set daleth into common, beginning in location 100.

Refer to the –dataset–, –datain–, and –dataout– instructions for more information.

NAMESET INSTRUCTIONS

The author language has eight nameset instructions: –setname–, –getname–, –addname–, –delname–, –rename–, –names–, –addrecs–, and –delrecs–. The –setname– instruction selects a named set of records from a nameset for further reference. It must precede –datain– and –dataout– instructions referencing the nameset. After executing a –setname– instruction which has a variable tag storing the name of a set, such as –setname n20–, the system-reserved word zreturn is set to:

Value	Meaning
-1	Name matches exactly
0	Name matches as far as possible with an existing name; set to reference it
1	Name matches as far as possible with more than one existing name; set to reference the first one
2	Name does not match any existing name; reference cleared
3	No nameset in effect

If the list of names in the nameset is

```
aleph
beth
daleth
he
heth
```

then the instruction

```
setname n20
```

with n20 storing the following names, returns the following values of zreturn and produces these actions:

n20	zreturn	Result
aleph	-1	References aleph
dal	0	References daleth
h	1	References he
mem	2	Clears reference

When the –setname– tag is nextname, the instruction selects the next name in alphabetic order. If no name is currently in effect, the –setname nextname– instruction selects the first name in the list. The system-reserved word zreturn is set to -1 upon selection of a new name and to 2 when the last name has already been selected.

When the –setname– tag is backname, the instruction selects the preceding name in alphabetic order. If no name is currently in effect, the –setname backname– instruction selects the last name in the list. The system-reserved word zreturn is set to -1 upon selection of a new name and to 2 when the first name has already been selected.

A –setname– instruction with a blank tag clears the current name and sets system-reserved word zreturn to -1. The author can select the names in a nameset in alphabetical or reverse alphabetical order by using –setname– with a blank tag and the nextname and backname tags.

Because the system allows sequential selection and partial matches, the author might not always know what name is in effect. To find out, the author can use the –getname– instruction. This returns the current name in 1, 2, or 3 full words, depending on the size of the name. Any additional part of the last word after the end of the name is zero-filled. Segmented variables are not allowed. If the –getname– instruction has two arguments in its tag, it returns the 24 bits of associated information in the rightmost bits of another word, clearing the leftmost bits.

The –addname– instruction adds a new name and a new set of records to a nameset. After a successful addition, the name added is the current name, eliminating the need for a –setname– instruction. After an –addname– instruction, the system-reserved word zreturn is set to the following values.

Value	Meaning
-1	Completed ok
0	No nameset in effect
1	No write access
2	New name duplicates an existing name
3	Number of records out of range
4	Nameset reserved; additions not permitted

The –delname– instruction deletes the current name and its records from a nameset. After a –delname– instruction, the system-reserved word zreturn is set to the following values.

Value	Meaning
-1	Completed ok
0	No nameset in effect
1	No write access
2	No name selected
4	Records reserved; deletions not permitted

The –rename– instruction changes the name of the current name. It can also change the 24 bits of associated information. If the new name is the same as the current name, then only the associated information changes. After a –rename– instruction, the system-reserved word zreturn is set to the following values.

Value	Meaning
-1	Completed ok
0	No nameset in effect
1	No write access
2	No name selected
3	New name duplicates an existing name
4	Records reserved in current name

To get a set of names from the current list in the nameset, the author can use the –names– instruction. This returns each name in 1, 2, or 3 full words, depending on the size of the names. Following each name is a word containing its associated information. The first 15 bits contain the number of records associated with the name and the last 24 bits contain the associated information. A mask or a defined vertical segment allows easy reading of the last 24 bits. After a –names– instruction, the system-reserved word zreturn is set to the following values.

Value	Meaning
-1	Completed ok
0	No nameset in effect
1	Invalid starting position

The –addrecs– and –delrecs– instructions add records to and delete records from the current named set of records. For example:

```
calc    n35 ⇐ 'aleph'
        n36 ⇐ 'heth'
setname n35
delrecs 1,2
setname n36
addrecs 1
```

This deletes the first two records from the set named aleph and adds one record after the last record in the set named heth. The addition or deletion of records from a name does not affect the content of the records. Therefore, the record added to heth might be one of the records deleted from aleph.

After an –addrecs– instruction, the system-reserved word zreturn is set to the following values.

Value	Meaning
-1	Completed ok
0	No nameset in effect
1	No write access
2	No name selected
3	Record number not in name
4	Records reserved; additions not permitted
5	Not enough records available

After a –delrecs– instruction, the system-reserved word zreturn is set to the following values.

Value	Meaning
-1	Completed ok
0	No nameset in effect
1	No write access
2	No name selected
3	Record number not in name
4	Records reserved; deletions not permitted

RESERVING NAMESETS

When changing information in a nameset record, the author can reserve and release records to prevent access to them by other users. Reserving a record or a nameset allows you to access it, but prevents access by other users. The –reserve– and –release– instructions reserve and release records relative to the current named set of records. For example:

```
dataset  hebrewset
calc     n53 ⇐ 'heth'
setname  n53
reserve  records,1,4
```

reserves the first 4 records of the set named heth in the nameset hebrewset.

```
reserve  dataset
```

reserves all the records in the nameset hebrewset, and it prevents the addition of new names or new records to the end of existing names by another user. Refer to the –reserve– and –release– instructions for more information.

For operations such as copying from one nameset to another, the author can keep records reserved in two namesets at the same time: the current active nameset and an inactive nameset. If a second –dataset– instruction is executed while a previous nameset is still attached, the first nameset becomes inactive with all its record reservations preserved. The author can reactivate the first nameset with its record reservations intact by re-executing the –dataset– instruction. The system can hold only one nameset in the inactive state. If a third –dataset– instruction is executed, the link to the current inactive nameset is closed and all its reservations are released. A –dataset– instruction with a blank tag closes the current nameset without preserving its record reservations and without affecting the current inactive nameset, if any. A SHIFT STOP exit or a –jumpout– releases all namesets unless an –inhibit dropset– is in effect.

MOVING BLOCKS OF DATA

When a number of variables continuous in the student variables or nc and vc variables are to be copied to another contiguous location within the student or nc and vc variables, the –block– instruction can be used.

The –block– instruction has three arguments in the tag. The first argument is the variable in the bank at which the block of values to be transferred begins. The second argument gives the first variable in the block of variables to which the values are to be transferred. The third argument gives the number of variables (or words) that are to be transferred. Thus, the instruction

```
block    v22,v50,7
```

transfers the contents of variables v22 to v28 into variables v50 to v56.

The transfer of values is nondestructive; that is, the values of the variables from which the values are transferred are left unchanged. Thus, the transfer is actually the creation of a copy rather than the removal of the values from the point or origin and placement of the values in a different location. An attempt to transfer more than 1500 variables results in an execution error.

The –transfr– instruction is a more general form of the –block– instruction. The arguments are the same as in the –block– instruction but are separated by semicolons and can have a more general form.

The first two arguments, specifying the origin and destination of the block of variables moved, can have any of several forms. If the form is nx or vx (where x is a number), the location is in the student variables. If the form is ncx or vcx, the location is in central memory. If the form is nrx or vrx, the location is in the user router variables. If the form is common,x or c,x, the location is in the copy of lesson common in ECS. Finally, if the form is either storage,x or s,x, the location is in the storage for the lesson (set by a preceding –storage– instruction). Thus, the –transfr– instruction allows transfer of variable values between any of the variables available to the lesson.

The CM variables which are transferred must be previously loaded and unloaded by a –comload– or a –stoload– instruction.

The PLATO author language includes a large number of instructions for creating displays. The author can, if desired, create pictures, show slides, create animated sequences, or simply display text.

COARSE AND FINE GRIDS

Before material, in whatever form, can be displayed on the screen, it is necessary to specify just where on the screen it should appear. There are two ways of specifying a screen position, coarse grid and fine grid. Coarse grid is usually sufficient for text display; in fact, it is designed specifically for text display. Fine grid is used to give more accurate positioning of display material.

Coarse grid divides the screen into 32 lines of 64 characters per line. The lines are numbered from top to bottom, beginning from 1. The character positions on each line are numbered from 1 to 64, running from left to right.

A coarse grid specification is a three- or four-digit number, with the last two digits specifying the character position (or column) and the first one or two digits specifying the line number. Thus, the position 214 indicates the second line and the fourteenth character position on that line. Similarly, 3210 specifies the bottom line of the screen, tenth character position (column) from the left of the screen.

The fine grid is somewhat different. The screen is divided into a 512-by-512 matrix, with each point pair representing a single dot on the screen. The numbers run from 0 to 511, both horizontally and vertically. The horizontal numbering is from left to right, as with the coarse grid, but vertical numbering is from bottom to top rather than from top to bottom.

A grid position is indicated by giving first the horizontal position and then the vertical position (the opposite of the coarse grid), with the positions separated by a comma. Thus, 234,175 indicates the 235th dot from the left of the screen at the 176th line above the bottom of the screen (the extra number is due to numbering starting at 0 rather than 1).

Some instructions permit more than one position to be specified in the tag. When this is done, the position numbers are separated by semicolons, as in:

```
draw        1230;702;153,254;1230
```

There are three system-reserved words that give the current screen position (cursor location). The reserved word where gives the coarse-grid screen position, and the reserved words wherex and wherey give the fine-grid horizontal and vertical position, respectively. The values of these system-reserved words are automatically updated by the system upon completion of a text display instruction.†

If it is desired to convert coarse grid to fine grid, or vice versa, the formulas are

finex = 800 frac (coarse/100)-8

finey = 512-16 int (coarse/100)

coarse = 100 (1+int((511-finey)/16)) + int (finex/8)+1

where

int(x) = the largest integer whose value is $\leq$ x

frac(x) = x-int(x)

For the current screen position, wherex, wherey, and where can be substituted for finex, finey, and coarse, respectively. The functions frac(x) and int(x) are the system functions.

BASIC DISPLAY PRESENTATION

What an author most commonly wants to place on a student's screen is text. The –write– instruction is used for this purpose.

The –write– instruction prints whatever is contained in the tag of the instruction on the screen. The tag can be more than one line long, provided that the lines after the first line are in the tag field (that is, they do not occupy the command field). For example, the instruction

```
write       The purple unicorn
            rides high on a sea of
            tuna fish.
```

displays the text just as it appears, including placing the text on three separate lines.

The –writec– instruction is a conditional instruction which acts as a conditional –write– instruction (refer to section 12).

---

†The system-reserved words where, wherex, and wherey are not always properly updated, unless special action is taken. For example, the instructions

```
calc    ni ⇐ 'xxxx'
showa   n1,6
```

cause the system-reserved words to be set as if six, rather than four, characters were displayed. Proper update to the reserved words is made if a –return– instruction is executed immediately following the –showa– instruction.

Unless otherwise specified, the first writing in a unit begins at coarse-grid position 101 (upper-left corner of the screen), and subsequent writing starts wherever the last writing ended. To specify a specific position on the screen where text is to be written, the –at– instruction is used.

The –at– instruction has as its tag a single screen position, which can be in either coarse or fine grid. When used for a –write– instruction (or some other display instructions discussed later), it specifies the position at which text presentation begins. It also sets a margin, so subsequent lines of text start at the same character position.

Because the tag refers to a screen position, the tag must be a permissible screen position. Therefore, if an expression or variable is used in the tag, the value of the expression or variable cannot, for example, be less than 101 (for a coarse-grid position). Similarly, a coarse-grid position such as 375 is not permitted.

The example of the –write– instruction previously given would appear in the upper-left corner of the screen, beginning at coarse-grid position 101, if it were the first such instruction in a unit and no –at– instruction was used. However, the instructions

```
at      1520
write   The purple unicorn
        rides high on a sea of
        tuna fish.
```

would position the text near the center of the screen, with all three lines starting at character position 20.

Screen locations for displaying text, data, and drawings may also be set with an –atnm– instruction. This instruction works like the –at– instruction except that it does not change the left margin for continued lines of the display. Continued lines are aligned with the margin previously set by an –at–. If no –at– has been specified, the default for the margin is character position 1 (left edge of the screen).

DISPLAYING VARIABLES

The –show– instruction displays the value of the first argument in a form dependent upon the contents of the next two optional arguments. For example, the instruction

```
show    v1,3
```

displays the contents of v1. If the value to be displayed has more than seven digits before the decimal point or is less than 10-4, it is displayed in exponential format. If the value is less than 10-9, a zero value is displayed. If a third argument (specifying minimum absolute value to be displayed) is used, a zero value is displayed if v1 is less than the absolute value.

The –showt– instruction displays the numeric value of a variable in tabular form. The first argument of the tag gives the variable to be displayed, and the second gives the number of digits displayed. This argument is an integer or a floating-point number of the form l.r or l,r where l specifies the number of digits to the left of the decimal point, and r specifies the number of digits to the right of the decimal point. If r equals 0 or is omitted, it specifies an integer number. If r is less than or equal to 9, either l.r or l,r is allowed. If r is greater than 9 or if l and r are variables rather than constants, then only l,r is allowed. If the value of the variable cannot be displayed in the specified length, the display is a line of asterisks. As an example, if the value of variable check is 625.38 and the instruction is

```
showt   check,2,5
```

the display is:

```
********
```

The existence of more digits beyond those specified to the right of the decimal point in the case of a floating-point number does not cause asterisks to be displayed. The value is rounded to the number of digits required.

If the second argument of the tag is omitted, the default number of decimal digits, in the case of an integer variable, is assumed to be 8. The default format, in the case of a floating-point variable, is assumed to be 4,3.

The –showo– instruction displays the value of the variable in octal. The first argument of the tag specifies the variable, as with the –showt– instruction, and the second specifies the number of octal digits that are to be displayed. If the second argument is omitted, a default value of 21 is assumed. Since one computer word is precisely 20 octal digits on the PLATO system, this allows an extra space, so tabulated results can be displayed without the necessity of worrying about space to be left between entries.

When the lesson author cares only about the most significant digits of a value, the –showz– instruction is most appropriate. This instruction has either one or two arguments. The first argument gives the value the instruction displays. The second argument specifies the number of significant figures of the value of interest to the author. If the second argument is omitted or zero, a default of four significant figures is used.

However, the number of significant figures desired and the number of digits displayed are not necessarily the same. The reason is the space required to display a value. If the value is 193752.675, displaying only the four most significant figures would give

$$1.937 \times 10^{5}$$

where the scientific (exponential) notation is used to give the magnitude of the number. The use of the period, multiplicative X , and 105, however, require five character spaces in addition to the significant figures. Since the full integer value uses only seven characters, as opposed to the nine characters used with the exponential format, it is shorter to display the full integer value, rounded if necessary, as in:

```
193753
```

The –showz– instruction checks whether full integer or exponential format is shorter and uses the shorter of the two formats.

When exponential form is desired, the –showe– instruction is used. As with the other forms of –show–, the first

argument gives the variable name and the second argument gives the number of digits. The third argument indicates the format. Standard format (for example, 4.00×10^3) is indicated by a 0 or by omitting the argument, and ** format (4.00*10**3) is indicated by a nonzero third argument. The width of the display field (that is, the width of the line used in the display on the screen) depends on the size of the exponent. A leading blank or a negative (–) is automatically provided to assist in the creation of the tabular displays. If the exponent power is zero, 100 is not displayed. If the second argument is omitted, four digits are displayed.

If the data in the student variable is in alphanumeric form, it can be displayed by using the –showa– instruction. In this case, the second argument of the tag gives the number of characters to be displayed. If omitted, the second argument is assumed to be 10. The instruction assumes that the information is packed left (left-justified) in the variable. This is the case if it was stored with a –pack– or –storea– instruction. Otherwise, it may be necessary to shift the data to the left of the word using the bit operator cls (refer to section 7).

Character spaces containing (numeric) zeros are ignored when displaying the variable, but the system-reserved word where is updated as if the characters were displayed. Thus, the instructions

```
calc      n1 ⇐ 'xxx'
          n2 ⇐ 'abc'
at        1510
showa     n1,20
```

display

```
xxxabc
```

beginning at screen location 1510. After display, the system-reserved word where has a value of 1530, although the actual current screen position is 1516. The system updates where after each line of writing and at the end of a timeslice. It does not update where after each –showa–.

The –text– instruction displays the contents of an alphanumeric buffer, automatically executing carriage returns at the end of each line of text. The –text– instruction displays lines of text faster than a loop of –at– and –showa– instructions can. CDC software identifies an end-of-line by two 6-bit zero codes at the end of the computer word. The –text– instruction converts these lowest 12 bits of zero into a carriage return and displays the entire buffer, ignoring any extra zero codes. If the variables n1 through n10 contain the following codes

Variable	Octal code
n1	24101123551411160555
n2	03171624011116230000
n3	34430000000000000000
n4	03100122010324052200
n5	00000000000000000000
n6	03170405235502050617
n7	22055524100555061122
n8	23245534355502112423
n9	00000000000000000000
n10	17065532052217570000

the instructions

```
at        1010
text      n1,10
```

display the following with the t from the word this at screen position 1010.

```
this line contains
18
character
codes before the first 12 bits
of zero.
```

The –hidden– instruction displays the contents of a variable, showing all the internal 6-bit codes, including the characters that normally are not displayed (hidden characters). Following are the symbols used to designate these characters.

Symbol	Code	Character
o	o00	Zero
_	o55	Blank
u	o66	Subscript
n	o67	Superscript
↑	o70	Shift
↵	o71	Carriage return
←	o74	Backspace
●	o75	Font
□	o76	Access

ERASING

It is sometimes desirable to erase text that has already been displayed. Entering a new main unit erases the entire screen, unless the author specifies otherwise with the –inhibit– instruction (refer to section 10), but entering a new unit is not necessarily the way the author wants to construct the lesson. Additionally, it may be that only part of the displayed text is to be erased. This type of erasing is done through use of the –erase– instruction.

There are three forms of the –erase– instruction; the form used depends on the type of erasing desired.

If the tag is blank or negative, the –erase– instruction erases the entire terminal screen. If the tag is zero, the instruction is ignored.

If the instruction tag is a positive value n, the instruction

```
erase     n
```

erases n characters, starting at the current cursor position. A character is an 8-by-16 dot area. If the value n is not an integer, it is rounded before erasing is done.

An –erase– instruction with a tag of two arguments specifies a block of character positions to be erased. The first argument specifies the number of character positions on each line that are to be erased (starting from the current screen position), and the second argument specifies the number of lines that are to have these characters erased.

For example, the instructions

```
at          1520
erase       20,5
```

erase character positions 20 through 39 of lines 15 through 19. The two arguments are separated by commas. The one- and two-argument –erase– instructions do not affect the system-reserved word where.

The –erase– instruction with the tag abort erases the entire screen and aborts any pending output in the buffer.

To erase a complicated display after a response is judged no, an –eraseu– instruction is used. The unit named in the tag of this instruction is executed (and remains in effect for the entire main unit) when any of the following keys are pressed after an ok or no judgment.

ERASE
SHIFT ERASE
NEXT (not after ok)
EDIT
SHIFT EDIT

Default erasing (occurs when one of the above keys is pressed after a no judgment) is done in addition to the eraseu unit. If a –write– instruction with a blank tag is the last writing done after a no judgment, default erasing is turned off.

If the tag for the –eraseu– instruction is blank or q, previous –eraseu– instructions are cleared.

LARGE AND ANGLED WRITING

It is sometimes desirable to use writing that is larger than normal to emphasize a point, to give a heading, and so on. This can be accomplished by means of the –size– instruction.

The form of the tag is a number or expression specifying the size of the writing. Normal writing is called size 0, instead of 1. Oversize writing is slower than normal writing, so whenever oversize writing has been used, a –size 0– instruction should be used to return to normal size. An example is:

```
unit      header
size      3          $$writing is 3 times normal size
at        1010
write     PSYCHOHISTORY
size      0
  .         .
  .         .
  .         .
```

Any writing (assuming no further modifications of size) that is done after these instructions is normal in size and speed. If a –size 1– instead of a –size 0– instruction were used, writing would be at normal size but at the slower speed of large size writing.

The –size– instruction can have either one or two arguments. If one argument is used, it is the size of both the horizontal and the vertical directions. If two arguments are used, the first gives the size in the horizontal direction and the second gives the size in the vertical direction. Since the –size– instruction is affected by the –rotate– instruction, the terms horizontal and vertical refer to the directions when the –rotate– instruction is not used.

The instructions which are affected by –size– are –write–, –writec–, –rat–, –rdraw–, –rcircle–, –rbox–, –rvector–, –labelx–, –labely–, –graph–, and –erase–.

The system-reserved words size, sizex, and sizey are set to the current values of size specified in the one- and two-argument tags, respectively.

The –size– instruction without a tag is equivalent to the –size 0– instruction.

It is also possible to write at an angle to the normal, horizontal presentation. This is specified by the –rotate– instruction. The tag gives the angle, in degrees, through which the line of text is to be rotated. Normal (size 0) text is unaffected by the –rotate– instruction, so if it is desired to write normal-sized text at an angle, the –size–1– instruction must be used prior to writing.

The zero angle is the point directly to the right of the center of the circle, with the angle increasing in a counterclockwise direction. This is standard mathematical notation.

Alternate character sets cannot be used when either the –size– or –rotate– instruction is in effect.

GRAPHICS INSTRUCTIONS

There are seven basic instructions for creating drawn displays on the student's screen.

The –dot– instruction places a single dot on the student's screen. The tag of the instruction can be either a coarse- or a fine-grid location. A fine-grid location specifies the dot to be placed explicitly. If a coarse-grid location is used, the dot is placed in the lower-left corner of the character block specified. It is possible to draw entire figures using the –dot– instruction; however, it is not recommended. The procedure is slow, difficult to read in the lesson code, and liable to error. A single, isolated dot on the screen will not always light, especially when the screen is blank.

For drawing figures, the –draw– instruction should be used. The tag of this instruction specifies a number of points, with separate points being separated by a semicolon. The point specifications can be either coarse or fine grid, and the two types can be mixed in a single instruction. The skip option is also available.

For example, the instruction

```
draw      2015;100,200;1500;skip;205;2015
```

draws a line from the lower-left corner of character position 2015 (coarse-grid) to dot position 100,200 (fine-grid), and from there to the lower-left corner of character position 1500 (coarse-grid, again). The cursor then skips

to coarse-grid position 205 without drawing a line and draws a line from character position 205 to character position 2015.

The system-reserved words where, wherex, and wherey are not updated until the end of execution of a –draw– instruction. Hence, an instruction such as

```
draw    131;1015;where+504
```

is not the same as:

```
draw    131;1015;1015+504
```

Drawing starting from the current cursor position (at the beginning of execution of a –draw– instruction) can be done by having a semicolon as the first character of the tag. The line is then drawn from the current character position to the location specified in the next argument. An example of this usage is

```
draw    ;where-6
```

This instruction underlines the last six character positions. If the –draw– instruction is given with a single argument, it functions as a –dot– instruction. The single argument is evaluated to give the point.

Each argument of a –draw– instruction can be an expression rather than a number or variable name. A maximum of 63 arguments can be used in a single –draw– instruction.

For drawing circles, there are two instructions, –circle– and –circleb–. The –circle– instruction draws a solid circle or arc, while the –circleb– instruction draws a broken circle or arc.

Both instructions have the same form of tag. The tag can have either one or three arguments. The first argument specifies the radius of the circle in fine-grid dots. If the three-argument tag is used, the second and third arguments specify the beginning and ending angles, respectively, for drawing an arc. The angles are in degrees rather than radians, and the degree sign is not used.

The zero angle is the point directly to the right of the center of the circle, with the angle increasing in a counterclockwise direction. This is standard mathematical notation.

The center of the circle or arc is set with a preceding –at– or –atnm–. If these instructions are not present, the center is set at the current value of wherex,wherey. The edge of the screen provides automatic windowing.

The –window– instruction limits the display area of the screen for display of anything except size 0 text display. The effect is to show only that part of the display that lies within the bounds of the window. Thus, a –circle– instruction executed after a –window– instruction shows the entire circle only if it lies completely within the window. Otherwise, it shows only a part of the circle, or if the circle lies entirely outside the window, nothing at all.

The –window– instruction limits the display area within a rectangle specified by opposite corners at the two locations given. If only one location is given, the window has opposite corners at 0,0 and the specified location. If the first location is omitted but the semicolon precedes the second location, the opposite corners are at the current screen position and the specified location.

The window specified is active until superceded by another –window– specification or until turned off by a –window– instruction with a blank tag. Entering a new unit does not turn off the window, just as it does not turn off the size specification.

The instructions affected by the –window– instruction are all of the display instructions except text display of size 0, the –dot– instruction, and author-designed characters. This includes the graphing instructions discussed later in this section.

The –box– instruction draws a rectangular box with opposite corners at the two locations given. If only one location is given, the box is drawn with opposite corners at 0,0 and the specified location. If the first location is omitted but the semicolon precedes the second location, the opposite corners are at the current screen position and the specified location. Wherex and wherey are set to the last point drawn. An optional third argument specifies the thickness of the border of the box.

The –vector– instruction draws a vector with the tail at the first location specified and the head at the second location specified. If only one location is specified, the tail is drawn at 0,0. If the first location is omitted but the first character in the tag is a semicolon, the tail is at the current screen position. If the size of the head is specified, the full three-argument form of the instruction must be used. If the vector is smaller than the size of the head, the arrowhead is automatically reduced in size. At some angles the arrowheads may look unusual because of the dot patterns that form the arrowheads. Often, this can be corrected by use of a different arrowhead size.

RELOCATABLE INSTRUCTIONS

Certain display instructions are relocatable with respect to a specified origin. With this feature, a set of instructions may display a drawing in one section of the screen at the beginning of a lesson, and then using the same set of instructions, display the drawing in another section of the screen later in the lesson. The relocatable origin is specified with the –rorigin– instruction. It serves as a reference point for the relocatable instructions. An –rorigin– setting remains in effect until another –rorigin– instruction is executed. If no –rorigin– is specified when entering a lesson, it is set to 0,0. Fine-grid coordinates are preferable over coarse-grid coordinates.

The relocatable instructions which –rorigin– affect are –rat–, –ratnm–, –rdot–, –rdraw–, –rcircle–, –rbox–, and –rvector–. These instructions work in the same manner as the corresponding instructions obtained by deleting the beginning r's, except that all are affected by the –size– and –rotate– instructions. For example, figures made with –rdraw– may be sized larger or smaller or may be rotated. If the figure is rotated, the size may be 0.

The –rat– instruction relocates screen positions relative to –rorigin–. When the –ratnm– instruction is used, continued lines are aligned with a margin previously set by –rat–. The default margin, in this case, is set by –rorigin–.

The –rcircle– instruction draws an ellipse if the values of sizex and sizey are different. The –rbox– instruction uses the current –rorigin– as a corner when only one location has been specified. The –rvector– instruction uses the current –rorigin– as the tail position if only one location has been given.

MODE CONTROL

The PLATO terminal operates in three modes under control of the author. The standard mode is the write mode. This is the mode the terminal is in when the student signs on. In this mode, new text is written over the character spaces without any display that might be there being affected. Hence, if text was already present, the new text is written without the original text being erased.

The rewrite mode erases a character space and then writes the new text in the space. This is useful for presenting new text on a portion of the screen without disturbing the rest of the screen. However, care should be exercised if this mode is used for line displays using such instructions as –draw– and –dot–. Writing must be size 0 in rewrite mode.

A method of erasing presented texts with a minimum effect on any background displays is through the use of the erase mode. In this mode, anything written or drawn is actually erased, but only those screen dots that would, in a different mode, be lighted upon execution are cleared. By changing the erase mode and reexecuting display instructions (whether graphics or text), only the dots of a character are affected. Thus, background displays are less disturbed than they would be by the –mode rewrite– instruction.

The mode of the terminal is set by the author using the –mode– instruction. The three keyword tags permissible are write, rewrite, and erase for the write, rewrite, and erase modes, respectively. A conditional form of the –mode– instruction is available (refer to section 12).

The –color– instruction may be used in much the same manner as the –mode– instruction for writing and erasing. In fact, the –color orange– instruction is equivalent to the –mode write– instruction, and the –color black– instruction is equivalent to the –mode erase– instruction.

EMBEDDING

For simplicity of presentation and manipulation of material, several of the display instructions can be embedded in a –write– (or –writec–) instruction. As an example, the instruction

```
write    The value of the expression is
         ◁t, result ▷.
```

places the value of variable result on the screen in the appropriate place.

The left delimiter of an embedded instruction is obtained by pressing ACCESS 0, and the right delimiter by pressing ACCESS 1. A list of the instructions that can be embedded, together with their embedded form, is given in table 9-1. The a1, a2, and so on, are the arguments of the tags. Some of these are optional for certain instructions. The standard default options for nonspecified parameters are in effect, except that no leading blank is supplied for the embedded form of the –showo– instruction.

The embedded –mode– instructions are rather different from the normal –mode– instructions. Rather than switching the mode of the terminal unconditionally, the embedded –mode– is in effect only until the end of the character text string that follows. The end of the –write– instruction or another embedded instruction of any kind returns the terminal to the mode the terminal was in prior to the beginning of the –write– instruction.

TIMING

To control the timing of displays, the author language provides two instructions, –catchup– and –delay–.

TABLE 9-1. EMBEDDED INSTRUCTIONS

Normal Form		Embedded Form	Comments
show	a1,a2,a3	◁s,a1,a2,a3▷	No trailing zero
showt	a1,a2,a3	◁t,a1,a2,a3▷	Tabular - fixed field
showz	a1,a2	◁z,a1,a2▷	No leading blanks
showo	a1,a2	◁o,a1,a2▷	Octal number
showe	a1,a2	◁e,a1,a2▷	Exponential format
showa	a1,a2	◁a,a1,a2▷	Character strings
at	a1,a2	◁at,a1,a2▷	Writing position
atnm	a1,a2	◁atnm,a1,a2▷	Writing position
size	a1	◁size,a1▷	Size of writing
rotate	a1	◁rotate,a1▷	Angle of writing
mode	write	◁m,w▷	Write mode
mode	erase	◁m,e▷	Erase mode
mode	rewrite	◁m,r▷	Rewrite mode

TABLE 9-2. RELATIVE TIMING EXAMPLE

Instruction		t_x †	t ††
unit	drill	.000	***
randu	n1	.001	***
randu	n2,50	.002	***
size	2	.003	***
at	510	.004	***
write	Add these numbers	.005	.005
at	1010	.015	***
showt	n1,3	.016	2.006
at	1208	.018	***
write	+	.019	2.087
at	1210	.020	***
showt	n2,3	.021	2.017
draw	;1210	.024	2.127
catchup		.025	***
arrow	1410	2.129	2.129
time	5	2.130	***
.	.	.	.
.	.	.	.
.	.	.	.

†tx is an approximate time of execution of the instruction relative to the time the unit was entered (in seconds).
† †t is the approximate time the event indicated by the instruction would be observed to begin relative to the time the unit was entered (in seconds).

The –catchup– instruction has a blank tag. It is used to coordinate the timing of execution of a lesson and presentation of material. The computer, in general, processes instructions at a much faster rate than material can be displayed on the screen. The –catchup– instruction causes the lesson to stop execution until all material sent to the terminal for display has been displayed. An example of this usage is given in table 9-2. (The times, tx and t, vary depending on system load and other factors.)

The –delay– instruction can be used to give precise specification of output delays. The tag is a number less than or equal to 1. The tag specifies the time delay in seconds (1 second maximum). This instruction is especially useful for blinking or animations.

CONSTRUCTING ALTERNATE CHARACTERS

Besides the 126 characters that the author language provides for all lessons, there are 126 characters that can be programmed by the author. For special characters in a lesson that use few such characters, the –char– instruction can be used. If many characters are used, it is possible and usually desirable to create an alternate character set. This is the case when teaching a language that does not use the Roman alphabet, such as Russian, or when complex but repetitious displays of small size are used.

Each character possesses an area 8 fine-grid dots wide and 16 dots high. The dots that are lighted when the character is plotted determines the appearance of the character.

The –char– instruction specifies a slot in the alternate character memory and the dots within the 8 by 16 pattern that are to be lighted. The slot can be specified either by the actual number or by a defined constant (refer to section 7) as the first argument in the tag. The instruction clears the alternate-character-set flag that indicates which alternate character set is loaded in the terminal, unless prevented by the –inhibit charclear– instruction (refer to section 10).

The specification of the character is usually done in the tag field of the two lines following the –char– command. The tag is nine arguments which can be entered in a single line if space permits. However, the multiple line tag is clearer and easier to edit in case of a change. Four of the columns of the character space are specified in each of these rows. The specification is done by identifying the positions in the column that are to be lighted with a set bit, that is, a value of 1 wherever the dot is to be lighted. For this reason, it is usually best (but not necessary) to give the specifications in octal, which has an easily interpreted relationship to binary (refer to appendix C). The specifications for each column are separated by commas. An example of a character definition is given in figure 9-1.

When the character has been defined, the –plot– instruction is used to show the character on the screen. The tag of the instruction is the name (or memory slot number) of the character to be plotted. The –plot– instruction displays only one character but otherwise functions as a –write– instruction functions. That is, the cursor must be placed prior to plotting, with the default of plotting the character in either location 101 (if no display instructions were previously executed in the unit) or just after the last

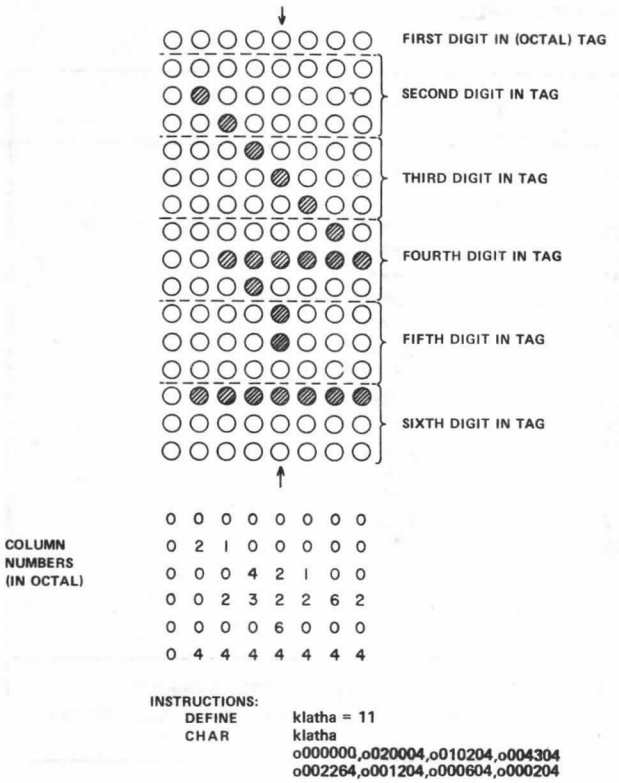


Figure 9-1. Sample of Character Definition

display (that is, at the value of the system-reserved word where). As an example, if character klatha has been defined as in figure 9-1, the instructions

```
at      1010
write   This is character klatha: $$with a space
plot    klatha                    $$following the:
```

would display

This is character klatha: ≥

on the screen.

The –plot– instruction can also be used to display a character from the nonprogrammable character set. In such a case, the value of the tag is between 0 and 126, depending on the character to be displayed.

CHARSETS

When a large number of alternate characters are used in a lesson, it is easier to specify a character set. This can be done by creating a new block in the lesson, specifying it as a –charset– block, and then following instructions. With this method, you can associate each character with a key or shifted key by simply specifying which key is to be used.

Once the character set is defined, it is necessary to load it into the alternate character memory of the terminal. This is done in the regular lesson code by the use of the –charset– instruction.

The –charset– instruction has two arguments in its tag. The first argument gives the lesson in which the character set is contained, and the second argument gives the name of the character set. As an example, the instruction

```
charset philk,blach
```

would obtain set blach from lesson philk and load the characters into the alternate character memory of the terminal. Additionally, it sets a flag identifying the character set to be used with the –chartst– instruction. If the character set is in the same lesson that uses the –charset– instruction, the first argument is optional.

A blank tag clears the alternate character set flag.

New character sets do not blank out the alternate character memory but merely replace those characters specified by the new character set. Hence, if an alternate character is not replaced by a new alternate character, the character is still contained within the terminal memory and can be used by the student or author.

After a –charset– instruction, the system-reserved word zreturn allows the author to check if the character set was properly loaded. The system-reserved word zreturn has the following values.

Value	Meaning
-1	Character set is successfully loaded
0	Character set is not found
1	STOP key is pressed
2	Character set is loaded improperly
5	No character set name is given

It requires about 17 seconds to load a complete character set, so it is usually desirable to write a message on the screen informing the student of the reason for the delay. If the required character set is already in the terminal memory, the terminal remembers this and loading does not occur.

Because different character sets can be moved into the terminal between sessions, the best location for the –charset– instruction is in the IEU. For example, the IEU

```
at        1221
write     Now loading character set.
          Please be patient, loading requires
          about 17 seconds.
charset   philk,blach
erase
```

loads the desired character set and performs an erase when done to prevent cluttering (or blocking) the display of the unit in which the student begins lesson execution.

The –chartst– (character test) instruction determines if the character set named in the instruction tag is currently loaded into the terminal. The first argument of the instruction tag specifies the lesson containing the character set, and the second argument is the name of the character set. If the character set is in the same lesson that uses the –chartst– instruction, the first argument is optional.

Sometimes it is confusing to a student when a 'loading characters' message quickly flashes on and off the screen. The system-reserved word zreturn has the following values.

Value	Meaning
-1	Character set is loaded
0	Character set is not found

After a –chartst– instruction, the author can use the value of system-reserved word zreturn to determine whether or not to place a message on the screen.

Alternate characters associated with a key are accessed by use of the FONT key. Unlike the SHIFT or ACCESS keys, the FONT key need not be pressed before each character to type the alternate character set. Instead, the FONT key switches the terminal back and forth between the two character sets. When the FONT key is pressed, the alternate character set is used until the terminal is switched back to the standard character set by again pressing the FONT key.

Alternate characters, whether of a full character set or constructed through use of the –char– instruction, are not affected by –size– or –rotate– instructions. In fact, using alternate font characters following a –size– instruction with a tag other than 0 (if no lineset is in use) results in the keys associated with the alternate font characters being written at the larger size rather than the alternate characters.

If all the text, both by the author and the student, is to be in an alternate character set, the –altfont– instruction can be used. This instruction switches the terminal into the alternate font if the tag is on, 1, or alt and switches back to the normal character set if the tag is off, 0, or normal. After an –altfont– instruction, the normal character set can be accessed temporarily by use of the FONT key, but a new line in a –write– instruction causes the terminal to return to the alternate character set. The effect of placing an –altfont on– instruction in the lesson is much the same as the author pressing the FONT key and using a –force font– instruction for the student responses. A difference is that the lesson code is in the standard character set, which can make editing difficult, unless there is a clear relationship between the normal characters and the alternate characters.

A pseudo-conditional form of the –altfont– instruction can be constructed by using a variable or simple expression as the tag. However, only integer variables should be used. Great care must be taken in the construction of the expression, since a value other than exactly 0 or 1 causes an execution error. (Hence, a floating-point variable assigned the value 0 cannot be used, since the value is not precisely 0).

LINESETS

A lineset is similar to a charset. Both are character blocks designed by the author; however, the line-drawn characters in a lineset may be sized and rotated using the –size– and –rotate– instructions. In fact, the size must be nonzero to display lineset characters; otherwise, charset characters are displayed instead. This allows a lesson to use a lineset and a charset at the same time. Since lineset characters are line drawings, they take as much time to plot as sized writing does.

The –lineset– instruction loads a lineset in a lesson. The first argument of the tag is the name of the lesson which the lineset is in, and the second argument of the tag is the name of the lineset. The lineset block does not need to be in the same lesson in which it is used; however, if it is, the first argument is optional. Since the –lineset– instruction is executable, different linesets may be used in one lesson. A blank tag cancels a previous –lineset–.

Lineset characters take only as much space as is necessary because they are variable length characters. A one-block lineset can have 128 small lineset characters or 30 intricate ones. A lineset may be from one to three blocks long and automatically expands or contracts depending upon the space it needs.

When an author is using the editor, pressing FONT and a letter shows a charset character rather than a lineset character because the editor is size 0.

MICROS

Micros are used to save work for the author or student. They allow two keypresses to do the work of up to 40 keypresses. In the student response buffer, the codes look exactly as if the student has performed each keypress manually instead of using the micro option. Thus, use of the micro does not affect judging of student responses. When a –long– is in effect at an –arrow–, the system does not substitute a micro if it is longer than the –long– specification. Where the author has used a –long 1– instruction, the micro substitution is made if it is less than or equal to 8 characters.

Each micro is contained in a micro table. A micro table has a maximum length of 256 words, with one word capable of holding up to 10 characters. As a result, the maximum number of micro definitions possible is 256, with a maximum of 10 characters per micro. This is because each micro requires at least one word. Micros can be defined as longer, with a corresponding decrease in the number of micros available. A micro table with 20-character micros has a maximum of 128 micros available, and a micro table with 40-character micros has a maximum of 64 micros available.

A micro table is contained in a special block of a lesson. Upon creation of the new block, the author must specify that it is to be a micro block and follow the directions given. When the table is first being edited, the author must choose the length of each micro. All micros in the same block have the same maximum length (although all available space is not necessarily used for each micro). Thus, if the author needs one micro that is 37 characters

long, all of the micros must be 40 characters long. The available lengths for a micro definition are 10, 20, and 40 characters per micro.

A micro table is obtained for lesson use by the –micro– instruction. This instruction is similar to the –charset– instruction. The tag specifies the lesson in which the micro table is located in the first argument and the name of the micro table in the second argument. If the micro table is in the same lesson that uses the –micro– instruction, the first argument is optional. A blank tag loads the system micro table.

The micro table is used by pressing the MICRO key (refer to section 1) and then the key associated with the micro. For example, if the key K is associated with the micro Kinnison, typing MICRO K places the word Kinnison on the screen, whether it is the student or the author who accesses the micro. The system micro table is always available with the ACCESS key, even if another micro table is loaded.

The –force– instruction can be used to force the student to use the alternate character set or to force all student keypresses to be sent through the micro table. The keyword tag of the instruction is font and micro for the respective cases. Additionally, the –force– instruction can be used to force response judging when the student response-length-limit is reached (refer to section 11). Options can be combined within a single –force– instruction by giving the desired tags, separated by commas, as in:

```
force       font,micro
```

-CODEOUT- INSTRUCTION

It is sometimes desirable to perform some special operations when writing on the screen. These can be done by pressing the appropriate key in a –write– instruction, but these can be difficult to spot when editing the lesson. Therefore, the –codeout– instruction is provided.

The tag of the –codeout– instruction is an octal number that specifies the action that is to take place. The permissible values of the tag are given in table 9-3.

TABLE 9-3. PERMITTED –codeout– TAG VALUES

Value	Action
o10	Backspace
o11	Tab
o12	Line feed, with no carriage return
o13	Vertical tab (up one line)
o14	Form feed (to upper left corner)
o15	Carriage return (to left of screen)
o16	Locking superscript
o17	Locking subscript

-TABSET- INSTRUCTION

The –tabset– instruction enables the author to give a set of tabs that the student can use in his response. The tag consists of exactly 10 two-digit fields, with a preceding o. All values are in octal. For example, the instruction

```
tabset      o 10 20 30 40 50 00 00 00 00 00
```

sets tabs for student use at columns 8, 16, 24, 32, and 40 (in decimal). The fields are not separated by commas. They can be made contiguous, as in:

```
tabset      o10203040500000000000
```

NONSCREEN DISPLAY INSTRUCTIONS

These are instructions for controlling the nonscreen capabilities of the PLATO terminal. Not all of them apply to every terminal or every installation.

The –slide– instruction selects a slide from the microfiche and projects the slide on the terminal screen. The tag is the number of the slide to be shown. There are 256 slides on each microfiche sheet, so slide numbers can range from 0 to 255.

There are two additional options with the –slide– instructions. If slide n is the slide desired, a tag of 512+n selects the slide but leaves the bulb of the projector turned off. If the tag is 256+n, the slide is selected, the bulb is turned on, but the shutter remains closed.

The locations of slides on the microfiche sheet can be specified by row and column (refer to figure 9-2). If the row (y) and column (x) of a particular slide are known, it can be referenced by an instruction of the following form.

```
slide       y+16*x
```

The use of defined constants (refer to section 7) can simplify the manipulation of slides so that the action taken is apparent from the code. For example, the instruction

```
define      shut=256
            off=512
            rowcol=16
```

enables the use of instructions such as

```
slide       y+rowcol*x
slide       shut+n1
slide       off+n1
slide       shut+y+rowcol*x
```

where shut, off, and rowcol give mnemonics for closing the shutter, turning the projector bulb off, and locating a slide by its row and column position.

The –enable– and –disable– instructions apply to the touch panel and to the external input devices of the terminal. Caution should be used when constructing lessons using external devices because not all terminals have these capabilities. Therefore, an alternate method of student response is usually desirable, such as using the –or– instruction (refer to section 11) when using the touch panel option.

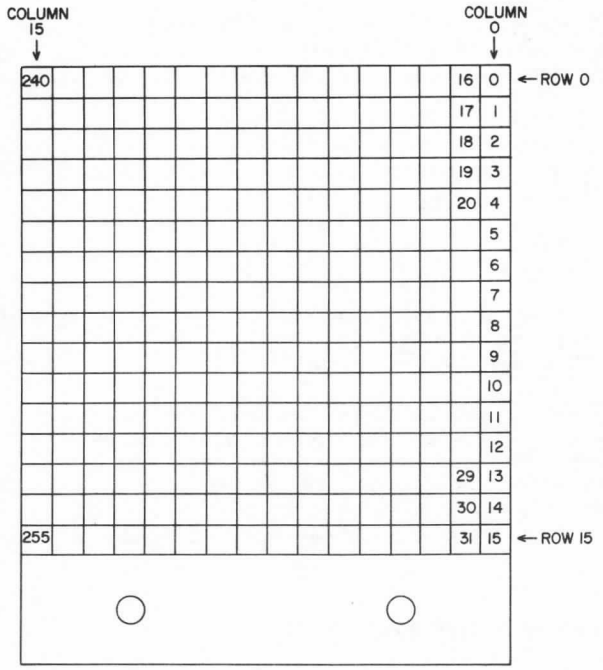


Figure 9-2. Microfiche Layout

The –enable touch– instruction allows input from the touch panel. It remains in effect until a –disable touch– is executed, an –endarrow– or another –arrow– is encountered, or a new main unit is entered with a full screen erase. (An exception is that –inhibit erase– allows –enable touch– to remain in effect when starting a new unit.) In a unit with an –arrow– instruction, the –enable touch– instruction should follow the –arrow– instruction. In a unit with no –arrow– instruction, the –enable touch– instruction allows the touch panel to function in the same manner as the NEXT key.

The –enable ext– instruction allows input from all other external devices. Its effect is cancelled only by –disable ext–.

The –play–, –record–, and –audio– instructions are used with the optional audio disk feature of the terminal. The audio disk has 128 tracks, each of which is 32 sectors long, with each sector lasting about 0.33 second. One track, therefore, is about 10 to 12 seconds long.

Playing and recording (up to 32 sectors with one command) can start anywhere on a track. However, recording separate words and then putting them together to form messages should not be tried; phrasing and pitch will sound unusual, and messages over track breaks will not reproduce.

Both the –play– and –record– instructions have a tag consisting of three arguments: track number, sector starting place, and number of sectors to be played or recorded. For example:

```
play        2,0,23
```

activates the prerecorded message on track 2, sector 0, which is 23 sectors long. All arguments may be mathematical expressions.

The –audio– instruction has a single-argument tag (variable or expression) that identifies the prerecorded message to be activated.

The –play– and –record– instructions, in most cases, should be used instead of the –audio– instruction.

The one-argument form of the –ext– instruction sends the rightmost 15 bits of the rounded integer value of the expression or variable in the tag to any external device connected to the terminal. The two-argument form checks if another station wishes to receive –ext– instructions. The –extout– instruction sends the 16 rightmost bits of a word to any external device connected to the terminal. It sends the rightmost bits of each word for as many words as specified in the second argument, starting with the variable in the first argument. Because the result of sending –ext– and –extout– values depends on the device attached, it is necessary to be familiar with the device and to know which device is connected.

RELATIVE GRAPHICS INSTRUCTIONS

The –gorigin– instruction works in a parallel fashion to the –rorigin– instruction. The –gorigin– instruction affects the graphing instructions and the relative graphics instructions. The graphing instructions are –axes–, –bounds–, –scalex–, –scaley–, –lscalex–, –lscaley–, –labelx–, –labely–, –markx–, –marky–, –graph–, –hbar–, –vbar–, –funct–, –delta–, and –polar–. These are described later in this section. The relative graphics instructions are –gat–, –gatnm–, –gdot–, –gdraw–, –gcircle–, –gbox–, and –gvector–. These instructions work in the same manner as the corresponding instructions obtained by deleting the beginning g's, with the following exceptions.

The –gat– instruction tag specifies scaled units relative to –gorigin–. If no –scalex– or –scaley– instructions have been previously executed, –gat– locates a position x dots to the right and y dots above the gorigin. In the absence of a –gat– instruction, continued lines of –gatnm– default to –gorigin–.

The –gdraw– instruction tag also specifies scaled units relative to –gorigin–. Ellipses may be drawn with the –gcircle– instruction if the x and y scales are different.

The –gbox– and –gvector– instructions use scaled coordinates as specified by previous –scale– and –bounds– or –axes– instructions. If –gbox– has a blank tag, it draws a box around the current –bounds–. If a –polar– instruction precedes a –gvector– instruction, the vector is drawn assuming the tag specifies radians.

The tags of the relative graphics instructions must specify fine-grid coordinates.

CREATING GRAPHS

The graph-creating capabilities of the author language are extensive and make the construction of graphs extremely simple.

The horizontal axis (abcissa) of the graph is usually referred to as the x axis, thus following the general algebraic convention. Similarly, the vertical axis (ordinate) of the graph is usually referred to as the y axis. In some of the graphing instructions, such as –markx– and –lscaley–, there is an instruction pair that performs in identical fashion, except that they refer to different axes. These instructions are differentiated by the final character in the command portion of the instruction. Instructions with an x as the final character in the command refer to the abcissa. Those with a y as the final character refer to the ordinate. It is not necessary that the variables which the axes represent, if any, be named x and y.

SETTING BOUNDARIES OF A GRAPH

The –gorigin– instruction specifies the location that serves as the origin of the graph. This is not necessarily the 0,0 point of the graph, but it is the point where the axes cross. The tag of the –gorigin– instruction is the screen location of the origin. A default –gorigin– instruction of 0,0 is executed at lesson initialization. A –gorigin– instruction remains in effect past unit boundaries.

The axes of the graph are drawn with the –axes– instruction. The tag of this instruction has two forms. If it is desired to draw the axes only in the positive direction starting at the origin, the tag is the length of the axes, with the x axis specified first, in fine-grid dots. If it is desired to have the axes extend in the negative direction as well, a tag with four arguments is used. The first two arguments give the length of the negative portions of the axes and are negative numbers. The second two arguments give the length in the positive direction. The x axis is the first argument in each pair. Both forms of tag have all arguments separated by commas. A blank tag redraws the axes after erasure.

If it is desired to draw a graph without showing the axes, the –bounds– instruction should be used instead of the –axes– instruction. The format of the instruction tag is the same as for the –axes– instruction, and it has the same effect but does not draw the axes. In both instructions, the x and y values refer to the number of fine-grid dots.

SCALING THE GRAPH AXES

The two instructions for scaling the graph, –scalex– and –scaley–, allow later references to specific points in terms of the graph coordinates rather than the normal screen grids. Both instructions have a similar tag, which can have either one or two arguments.

A tag with one argument gives the maximum value of the axes. For example, the instruction

```
scalex      300
```

specifies that the maximum value that the x (horizontal) variable can attain on the graph is 300. Since the length of the axes should have been specified previously, this instruction indicates the scale of the graph (for the x axis). After both –scaley– and –scalex– instructions, following references to points on the graph, refer to the coordinate values of the graph rather than to the fine-grid locations on the screen. This form of the tag assumes that the value of the variable at the origin is 0. If this is not desired, the two-argument form of the tag is used. The second argument, separated from the first by a comma, specifies the value of the relevant axis at the origin. Thus, the instructions

```
scalex      300,50
scaley      200
```

indicates that the maximum value of x is 300, the value of x at the origin is 50, and the maximum value of y is 200, with a value of 0 at the origin.

There are also instructions to scale the axes logarithmically. These are the –lscalex– and –lscaley– instructions. They are equivalent to the –scalex– and –scaley– instructions but scale the axes accordingly to the common logarithm of the variable value. If an offset value is not specified, a value of 1 (10^0) at the origin is assumed.

LABELING THE AXES

It is usually desirable, if axes are drawn, to mark units on the axes. There are two methods of marking axes in the author.

The –labelx– and –labely– instructions mark the axes and provide appropriate numeric labels. The tag can have either of two general forms depending on whether the axis has been scaled or log-scaled.

If the axis has been normally scaled, the tag specifies the location of major and minor marks, together with the size of these marks. The first argument gives the spacing of major marks, and the second gives the spacing of the minor marks. This argument can be omitted. An execution error occurs if either of these arguments are greater than 100. The third argument specifies the length of the marks. If the third argument is 0 or omitted, the marks are of normal length. If the third argument is 1, major marks are extended to the boundaries of the graph. If the third argument is 2, all marks, both major and minor, are extended to the graph boundaries. The fourth argument is an integer or a floating-point number of the form l,r or l.r where l specifies the number of digits to the left of the decimal point and r specifies the number of digits to the right of the decimal point. If r equals 0 or is omitted, no decimals are shown. If r is less than or equal to 9, either l,r or l.r is allowed. If r is greater than 9 or if l and r are variables rather than constants, then only l,r is allowed.

As an example, the instruction (assuming that the x axis has been scaled)

```
labelx      25,5,1
```

places a major mark every 25 scaled units and a minor mark every 5 units. The major marks extend to the top of the graph, but the minor marks are of normal length.

The numeric labels are placed by the major marks whenever possible.

If a major mark interval of 0 is specified, as in the instruction

```
labely      0
```

the author language executer attempts to make a reasonable decision on where marks should be placed, but the result is not always optimum.

If the axis has been log-scaled, a slightly different tag is used. In this case, the first argument specifies the format and must be 0. The second argument specifies the interval of minor marks (major marks are automatically placed each decade). If the second argument is -, no minor marks are made. If the second argument is 0, 3, or omitted, minor marks are placed at 1, 2, and 5 within each decade. If the second argument is 5, minor marks are placed at 1, 2, 3, 5, and 7 within each decade. The third argument specifies the mark size, the fourth argument specifies the format, and both function as in labeling of normally scaled axes.

If it is not desirable to give the numeric labels on the axes, the –markx– and –marky– instructions are used. These instructions function exactly like the –labelx– and –labely– instructions but omit the numeric labeling.

It is, of course, possible to use a –label– instruction on one axis and a –mark– instruction on the other.

WRITING ON THE GRAPH

There are two approaches to writing on the graph. The first uses a standard –write– instruction. The location can be specified by an –at– instruction, as in other lesson creations. Additionally, there is the –gat– instruction, which functions in the same manner as the –at– instruction, but which is relative to –gorigin– and uses scaled coordinates.

The other approach is the use of the –graph– instruction. The tag of this instruction can have up to four arguments. If the tag contains two arguments, they are assumed to be an x,y location, and a dot is placed at that location. If the graph has been scaled previously, the x and y are assumed to be scaled quantities. Otherwise, the –graph– instruction is the same as a fine-grid –dot– instruction.

If the –graph– instruction has three arguments, the third argument can be a text string and is written on the screen, with the first character placed at the location specified by the x and y arguments. The string can be up to nine character codes long. A capital letter uses two character codes rather than one. The instruction

```
graph       x,y,string
```

is equivalent to the instructions

```
gat         x,y
write       string
```

except that the starting location of the string in the –graph– instruction is moved somewhat down and to the right of the location specified so that the first character is centered on the location.

If the third argument is a variable containing the string to be written, the string can be up to 10 character codes long. This form can have an optional fourth argument, specifying the number of characters to be plotted.

DRAWING BARS ON THE GRAPH

There are special facilities for drawing bars on a graph, the –hbar– and –vbar– instructions. The –hbar– instruction draws a horizontal bar (that is, parallel to the x axis or abcissa), and the –vbar– instruction draws a vertical bar (that is, parallel to the y axis or ordinate). The two instructions function identically except for the difference in direction. The tag of these instructions can have two, three, or four arguments. The two-argument tag specifies a location (x,y) to which the bar is to be drawn. As an example, the instruction

```
vbar        35,20
```

draws a vertical bar, centered on abcissa location 35, that is 20 units in height. Similarly, the instruction

```
hbar        35,20
```

draws a bar from the ordinate axis to the same location, with the bar centered over ordinate position 20. (These instructions display bars as in figure 9-3). The units are fine-grid dots, unless a –scalex– or –scaley– instruction has been executed. When an axis has been scaled, any later references to that axis use the scaled units.

A tag with three arguments uses the third argument as a string to be plotted and plots the bar using this string. The string can be up to nine character codes long. For example,

```
vbar        x,y,***
```

draws a vertical bar to location (x,y) (assuming that values for these have been previously specified) that is three characters wide. The leftmost character is centered (horizontally) over the abcissa specified. The lower part of the character is usually the actual value, so the bar is somewhat higher than it should be.

If the third argument in a three-argument tag is a variable name, the contents of that variable, interpreted as a string of characters, is used to plot the bar and must be separated by a semicolon. All 10 character codes contained in the variable are used.

Because of the use of double dollar signs ($$) for comments (refer to section 10), a string of two or more dollar signs cannot be used to plot a bar, as they are treated as a comment.

A four-argument tag assumes that the fourth argument, whether a variable name or a number, specifies the number of characters to be written from the variable named in the third argument. Thus, if the variable groaci contains the characters hisop, and variable bolo contains the number five, the instruction

```
vbar        100,150;groaci,bolo
```

draws the vertical bar using the characters hisop.

The three- and four-argument forms cannot be used to draw bars in a negative direction. Hence, instructions such as

```
hbar        -10,13,*
```

are not allowed.

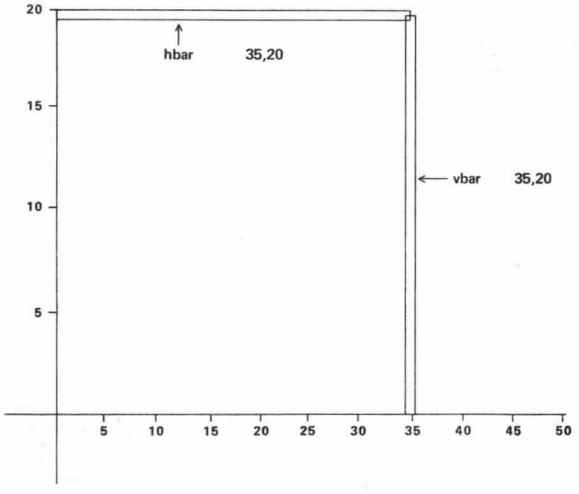


Figure 9-3. Example of –hbar– and –vbar– Instructions

GRAPHING FUNCTIONS

If it is desired to plot a given function of one variable, the –funct– instruction can be used. Associated with this instruction is the –delta– instruction. The –delta– instruction specifies an increment size in its tag. This is the increment size that is to be used unless the extended form of the –funct– instruction is used.

The simplest form of the –funct– instruction has two arguments in its tag. The first argument is the function to be plotted, and the second is the independent variable that is to be used. If the independent variable does not appear in the function, either explicitly or implicitly through –define– definitions, the function is plotted as a constant quantity. In this form of the instruction, the bounds of plotting are the bounds of the independent (abcissa) variable.

An extended form of the –funct– instruction allows greater flexibility in controlling the plotting of the function. The instruction takes the form

```
funct       f(x),x⇐xbeg,xend,dx
```

which is similar to the iterative –do– instruction (refer to section 10). The first argument is the function to be plotted and can be any compilable expression. The x is the independent variable. Therefore, if the variable specified is not present, explicitly or implicitly in the function to be plotted, the function is plotted as a constant quantity. The third argument (immediately to the right of the ⇐) specifies the initial value of the independent variable. The fourth argument gives the final value, beyond which the function is not plotted. Finally, the last argument gives the size of the increment to be used in incrementing the independent variable. The sign of the last argument controls the direction of plotting.

Plotting of complicated functions, in either form of the –funct– instruction, can cause time-slice errors in the lesson if the increment specified is too small. As a general guide, the increment should be larger than 0.02 X (xmax-xmin). The independent variable should also be a floating-point rather than an integer variable.

When the graph to be plotted does not fit well into the form of a function or when evaluating it as a function causes time-slice errors, the –gdraw– instruction can be used.

The –gdraw– instruction is exactly like the ordinary –draw– instruction, but it operates relative to the current –gorigin–, can use scaled units, and only uses fine-grid form.

POLAR COORDINATES

Up to this point, all of these instructions have assumed the use of Cartesian (that is, rectangular) coordinates. If desired, polar coordinates can be used in all except the –hbar– and –vbar– instructions. The use of polar coordinates is enabled by the –polar– instruction.

When a –polar– instruction with a nonnegative or blank tag is executed, all coordinate references following are assumed to be in polar coordinates. Polar conversion is turned off, and Cartesian coordinate usage resumed, upon execution of a –polar– instruction with a negative tag. The magnitude of the negative tag is irrelevant.

If the tag of the –polar– instruction is blank, polar conversion is turned on, and any scaling that has been done is unaffected and is used in the polar coordinates. A tag consisting of a single argument scales both the x and y axes to the value of the tag. (In terms of polar coordinates, this means that the actual length of a radius is independent of the angle.) A tag with two arguments scales the x and y axes separately, with the x axis being scaled by the value of the first argument.

Polar conversion remains turned on, even past unit bounds. Hence, the author should always include a –polar– instruction with a negative tag when he is finished using polar coordinates, or unanticipated results can occur.

When polar coordinates are used, the first argument is the radius, and the second argument is the angle (in radians), in all cases. Thus, if the radius is represented by r and the angle by θ , r and θ replace x and y, respectively, in all instructions. The relationship between the two coordinate systems is:

$$x = r \times \cos(\theta)$$
$$y = r \times \sin(\theta)$$

The PLATO author language provides a number of methods for branching within a lesson. The resulting structure can be complex. The instructions fall into two major categories, author-initiated branching and student-initiated branching. In addition, several instructions for lesson control other than branching are available. Many of these also have a conditional form (refer to section 12).

Nearly all lesson sequencing is done in terms of sections of source code. These sections are called units. The –unit– instruction delimits the sections (units) of the lesson. The tag of the instruction is the name of the unit, and it must be no more than eight characters in length. Because of the use of characters x and q in branching instructions, neither of these can be used as the name of a unit. The block of source code following the –unit– instruction until the next –unit– instruction or the end of the lesson, whichever comes first, can generally be thought of (in terms of lesson structure) as a single box that performs some specified actions. This is an over-simplification, since it is possible to enter a unit at a point other than the beginning and to leave at a point other than the end. However, the concept has utility as a conceptual device, provided the limitations are kept in mind.

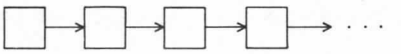
Several methods of accessing a unit are available. The execution of the unit depends, to an extent, on the method of access. The system has several reserved words that store information about the unit currently being executed. When a new unit is accessed, some or all of these reserved words are changed. If all of them are changed, the new unit is called a base unit. If only some of them are changed, the new unit is usually a help unit. Another type, the auxiliary unit, changes only one of the reserved words. (The terms base unit, help unit, and auxiliary unit are explained in detail later in this section.) Besides changing the reserved words, accessing help units and base units usually erases the entire terminal screen. A base unit is said to have full initialization as the reserved words are changed and the screen erased. Help units have partial initialization as the screen is usually erased, but only some of the reserved words are changed. An auxiliary unit is said to have no initialization (although this is not, strictly speaking, true), because only one reserved word is changed and the screen is not erased.

AUTHOR-INITIATED BRANCHING

The author can use several sequencing methods to impose a logical structure on a lesson. Each method of branching within a lesson specifies a certain type of execution, as well as structure, on the lesson.

USE OF THE -NEXT- INSTRUCTION

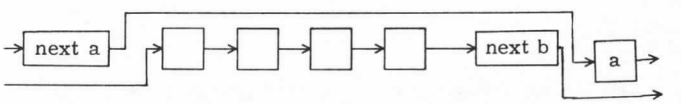
The simplest structure of an author language lesson containing more than one unit is a simple sequence of the units.



This simple structure requires no effort at sequencing on the part of the author, since the author language assumes, unless otherwise specified, that the next unit in the physical lesson is also the next unit in the logical lesson.

If it is desired to execute the units in a different sequence, the –next– instruction is used. The tag of the instruction is the name of the unit that is to be executed after the current unit. If more than one –next– instruction is used in a single unit, the last one encountered is used.

The –next– instruction is most often used to skip over units that are part of a different sequence or auxiliary and help sequences. As an example, the structure might be similar to:

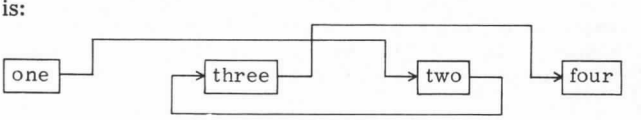


Here the logical lesson is quite different from the physical lesson, since the physical order of the units is not the order in which they are executed.

The –next– instruction can be used to completely reorder the sequence of execution. The physical lesson structure for

```
unit        one
next        two
 .
 .
 .
unit        three
next        four
 .
 .
 .
unit        two
next        three
 .
 .
 .
unit        four
 .
 .
 .
```

is:



but the logical lesson structure is:

For the remainder of this section, any sequence of units is given in logical order rather than physical order. This order is not necessarily the same as the physical order, but it is the sequence in which the units are executed.

-JUMP- INSTRUCTION

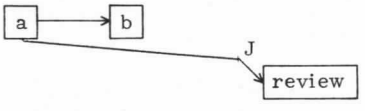
It is sometimes desirable to start a new unit immediately, although the current unit has not been completely executed. The –jump– instruction causes the unit named in the tag to be executed immediately. All normal initializations are made, including erasing the screen, unless the –inhibit erase– instruction is in effect.

For example, if part of the unit being executed is

```
wrong       hamilton
jump        review
  .
  .
  .
wrong       burr
  .
  .
  .
```

the –jump– instruction causes unit review to be started if the student answered hamilton.

The logical structure might be similar to



where a is the unit containing the jump instruction and b is the next unit to be executed if the –jump– instruction is not executed.

The –jump– instruction also has a conditional form. As with most conditional forms in the author language, a unit is selected on the basis of the value of an expression. Conditional forms of instructions are covered in detail in section 12.

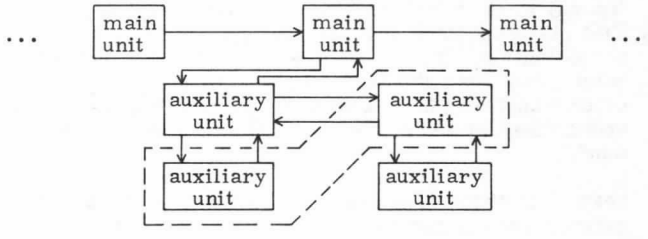
AUXILIARY UNIT STRUCTURES

The –join– and –do– instructions allow one unit to be executed inside another. (All statements made in the following paragraphs about the –join– instruction apply equally to the –do– instruction, unless otherwise noted.) No unit initialization takes place. Rather, the –join– instruction usually acts as if the –join–ed unit (minus the –unit– instruction) replaced the –join– instruction. This text-insertion characteristic of the –join– instruction makes it useful for eliminating repetition of groups of identical instructions in various places within the lesson. Instead, a single copy of each group of code is written and then –join–ed to other units when needed.

A unit that is –join–ed to another is an auxiliary unit. Program control causes auxiliary sequences to be only one unit deep. However, an auxiliary unit can have auxiliary units of its own, and these also can have auxiliaries. In fact, the author language allows the author to go as many as 10 levels deep in –join–s. Thus, while it is not possible to have an auxiliary unit sequence, it is possible to have an auxiliary unit structure attached to a unit.

Some care must be taken concerning –join–ed units, however. Since no unit initialization is done, all instructions in the –join–ed unit are executed as if they were in the main unit, except for the –goto– instruction. For example, a –next– instruction is valid and sends the student to the unit named in the –next– instruction after the current main unit is executed. Since a unit may be –join–ed from a number of other units, this is not necessarily desirable. In general, any instruction that could affect unit and lesson execution in a main unit acts similarly in an auxiliary unit.

Graphically, an auxiliary unit (with possible auxiliaries of its own) might be represented by:



The effect of an auxiliary unit depends on the value of the variables (if any) used in the unit. As an example, the unit

```
unit        assign
calcs       n3,n17 ⇐ 0,1,15,27,2,8
```

assigns different values to variable n17, depending on the value of variable n3 when the unit is –join–ed. The instructions

```
calc        n3 ⇐ -1
join        assign
```

assign variable n17 a value of 0, while the instructions

```
calc        n3 ⇐ 2
join        assign
```

assign variable n17 a value of 27.

The –join– instruction is unique in that it is executed both in the judging state and in the regular state. This makes the –join– instruction more useful and permits the use of auxiliary units anywhere within a unit. The –do– instruction is executed only in the regular state.

While the –join– instruction can usually be thought of as a text insertion device, there is an exception. If the unit that is –join–ed contains a –goto– instruction, the –goto– does not stop execution of instructions following the –join– instruction, as would be the case if the –goto– were contained in the main unit. Instead, when the auxiliary

units are completed, they are un–join–ed, and the instructions following the –join– instruction are executed. This permits looping among auxiliary units through use of the –goto– without affecting execution of the main unit. Furthermore, the –goto– instruction does not affect the level of –join–s, so auxiliary units can be attached to a depth greater than 10, providing that some of these are accessed by a –goto– instruction.

ITERATIVE -JOIN- AND -DO- INSTRUCTIONS

(All statements made in the following paragraphs about the –do– instruction apply equally to the –join– instruction.)

The format of the iterative –do– instruction is

```
do          name,var⇐beg,end,inc
```

where **name** is the name of the unit the author wants to execute repeatedly, **var** is the name of the variable that is used as an index for the repetitions, **beg** is the initial value this variable is given, **end** is the final value of the variable for which execution of the unit is desired, and **inc** is the amount the value of the variable is incremented at each repetition. **inc** is optional, with a default of 1, and may be negative or a fraction. **beg**, **end**, and **inc** may be mathematical expressions. The instruction

```
do          filk,index⇐10,20,10
```

executes unit filk twice, with the variable named index having values 10 and 20, whereas the instruction

```
do          filk,index⇐10,20
```

executes unit filk 11 times, with the variable named index having values 10, 11,...,19, 20.

Just as a –join–ed unit can contain a –do– instruction, a unit accessed by a –do– instruction can contain a –join–.

Units accessed with a –do– instruction are auxiliary units.

CONDITIONAL-ITERATIVE -DO- AND -JOIN- INSTRUCTIONS

(All statements made in the following paragraphs about the –do– instruction apply equally to the –join– instruction.)

The conditional-iterative form of the –do– instruction combines the features of iterative execution of a unit with the ability to conditionally specify which unit is executed as the iteration proceeds. For example, the instruction

```
do          exp,u1,u2,u3,u4,index⇐0,4,1
```

is executed the same as the iterative –do– instruction, except the value of the conditional expression (exp) is evaluated every time the variable index is incremented and the proper unit (u1,...,u4) is executed.

If exp is negative, unit u1 is executed. If exp is zero, unit u2 is executed, and so on. Since exp is evaluated each time index is incrememted, a different unit may be executed as the iteration continues.

Unit names x and q have special meanings: x means do nothing, and q means stop execution of this instruction and continue execution with the instruction that follows.

-GOTO- INSTRUCTION

The –goto– instruction is a hybrid of the –jump– and –join– instructions. Instruction execution is done in the same manner as with the –jump– instruction, but as with the –join– instruction, a new unit is not initialized. For example:

```
unit        first
goto        third
at          1012
write       This is never written
*
unit        second
 .
 .
 .

 .
 .
 .
*
unit        third
 .
 .
 .

 .
 .
 .
*
```

While the –goto– instruction generally does not return to the unit in which it occurred, there are two cases in which it does return.

If the –goto– instruction is part of the response to a matched answer, the fact that markers are not changed causes control to return to the calling unit and search for another –arrow– instruction. If there is another –arrow–, it is processed and execution follows the same pattern as would be executed in the absence of the –goto–. If there is no other –arrow–, no actual return to the unit for execution is made.

If the –goto– is contained in an auxiliary unit, after the –goto– unit has been executed, control returns to the auxiliary unit that contains the –goto– instruction. The remainder (if any) of the auxiliary unit is not executed, but the auxiliary unit is un–join–ed, and control returns to the unit that called the auxiliary unit.

The –goto– instruction also can have a conditional form. If the action specified by the conditional expression is the tag q, execution of the current unit stops and goes to the end of that unit. For a full discussion of the conditional form of instructions, refer to section 12.

The practice of sorting units into different types is somewhat artificial (refer to section 3). The –goto– instruction furnishes an excellent example of the reason. Since no unit initialization is done, a unit accessed by a –goto– instruction could be considered an auxiliary unit. However, the instructions following a –goto– are not executed, and the unit from which access was done is not returned to. Because it is not a base unit (the base unit marker is not initialized), it is not in a sequence (the next unit executed, unless modified by other instructions, is the unit following that from which access was obtained). In particular, it is not part of a help sequence, and it is usually considered an auxiliary unit.

ARGUMENTED UNITS

The –do–, –jump–, –goto–, and –join– instructions can pass arguments to the unit branched to at execution. As an example, the unit

```
unit        assign(n3)
calcs       n3,n17⇐ 0,1,15,27,2,8
```

assigns different values to variable n17 depending on the value of variable n3 when the unit is executed. The instruction

```
join        assign(-1)
```

assigns variable n17 a value of 0, while the instruction

```
join        assign(2)
```

assigns variable n17 a value of 27. A unit may have up to 10 arguments.

The branching instructions (–do–, –jump–, –goto–, and –join–) may specify fewer arguments than the unit has defined, but they may not specify more arguments.

-EXIT- INSTRUCTION

The –do– and –join– structures can be complex and can achieve their desired end before normal termination. The –exit– instruction allows termination of these structures before normal completion.

There are two types of tags that can be used in the –exit– instruction, blank and numeric. A blank tag or a tag of –1 causes the entire structure to terminate, with control returning to the main unit. A numeric tag causes the structure to back out from the structure the specified number of levels. If the tag is 0, the instruction is ignored. As usual, a variable or expression can be used instead of a constant in the numeric tag.

-NEXTNOW- INSTRUCTION

The –nextnow– instruction terminates processing of the unit and disables all terminal keys except the NEXT key. It is usually used to give the student feedback on the correctness of his answer without allowing him to change his answer if wrong.

There are two tags that can be used with the –nextnow– command. If the tag is a unit name, control is transferred to that unit when the student presses the NEXT key. The tag can also be in conditional form, in which case it follows normal conditional form conventions (refer to section 12).

-IFERROR- INSTRUCTION

The –iferror– instruction specifies the unit to which a –goto– is made if an error is detected in a –calc– instruction. The –goto– is performed by the –iferror– instruction upon error detection, not by the author.

The unit specification marker clears in the same manner as the –next– marker; that is, it clears when a main unit is entered or by the following instruction.

```
iferror     q
```

-ENTRY- INSTRUCTION

The –entry– instruction specifies an alternate entry point to the unit. The tag, as with the –unit– instruction, is a unique name of eight or fewer characters. The name is then used as a unit name is used.

When a unit is entered from an alternate entry point, initialization is performed in the same manner as if the entry point were a –unit– instruction; however, it does not do a full screen erase, and it does not initiate a new main unit. The actual initialization performed, as usual, depends on the manner of access. Thus, an entry point accessed with a –jump– has full initialization, an entry point accessed as a help sequence has partial initialization, and an entry point accessed with a –join– instruction has no initialization.

Use of the –entry– instruction permits looping within a unit. A faster looping method is to use the –branch–, –doto–, or –loop– instructions.

BRANCHING WITHIN A UNIT

A –branch– instruction branches to different lines of code within the same unit; the branching is to a line with a statement label. A statement label is a name that starts with a number, is seven or less characters in length, and begins in column 1 of the line. The following instructions draw a box in one of two locations, depending on the values of n1 and n2.

```
branch      n1<n2,x,100
box         1244; 1550
branch      2out
100
box         2240; 2550
2out
```

Branching cannot be done between units or around –entry– statements.

Two statement labels in the same unit cannot have the same name, but the same statement label can be used in more than one unit.

A conditional form of the –branch– instruction is available (refer to section 12).

-FINISH- INSTRUCTION

The –finish– instruction causes one more unit to be executed when the student exits a lesson by pressing the SHIFT STOP keys. This allows the author to process data before the student leaves the lesson. No input or output to the screen can be performed in this unit. The –finish– instruction need only occur once in a lesson. Therefore, the logical location for it is the IEU, although this is not mandatory. The tag of the instruction is the name of the unit to be executed upon lesson exit. A conditional form is available.

Some instructions are not legal in a –finish– unit (the unit to be executed) and result in an execution error display. These instructions include the –pause–, –catchup–, and –return– instructions.

The execution of an –end lesson– instruction or a –jumpout– instruction does not cause the –finish– unit to be executed.

A –finish– unit allows only 10 disk accesses or 10 time-slices.

-IMAIN- INSTRUCTION

The –imain– instruction causes the unit named in the tag to be executed at the start of every main unit in a lesson. For example, with this instruction an author can specify the help-type keys he wishes to have active throughout a lesson.

The –imain– instruction is in effect for all main units executed after the unit containing –imain–. Placing the –imain– instruction in the lesson's initial entry unit causes the imain unit to be effective for the entire lesson. A later occurrence of an –imain– instruction overrides any earlier settings.

A conditional form is available. The –imain– instruction with a blank tag clears previous setting. The imain unit is not executed in auxiliary units.

TIMED BRANCHING

Two instructions allow an author to branch a student to another unit or to the router after a specified time, the –timel– and the –timer– instructions.

The first argument in the tags of these instructions is an expression which specifies the length of time in seconds. For the –timel– instruction, the second argument is the name of a unit in the current instructional lesson. For the –timer– instruction, the second argument is the name of a unit in the router lesson.

The –timel– and –timer– instructions with blank tags clear the existing –timel– and –timer– instructions, respectively. The –timel– instruction also clears when the student exits from the current lesson, and the –timer– instruction clears when the student signs off the system.

The –timel– instruction provides a –helpop– type branch to the specified unit when the time limit is reached. It allows timing to continue over several units in a lesson and may be used in any unit of the lesson. The –timel– instruction is not affected by any –time– instruction which may be used in any of the units. The minimum time limit which can be specified is 0.75 second.

The –timer– instruction causes a branch to the specified unit of the router lesson when the time limit is reached. This instruction may only be used when an author is writing his own router lesson. When the student is branched to the router, the IEU of the router is not executed. The minimum time limit which can be specified is 300 seconds.

STUDENT-INITIATED BRANCHING

The PLATO author language allows the author to give the student the ability to branch within the lesson. Student-initiated branching permits the lesson to better fit the needs of the individual student by providing additional information or background (remedial) information. Further, all of the branching capabilities of the author language are available so that the student, if the author desires, can be sent back to review material already covered or to an entirely different sequence of main units. Student-initiated branching can be achieved three different ways with function keys. The first is a branch to a new main unit sequence, the second is the help sequence, and the third is the use of the TERM key.

BRANCHING TO A NEW MAIN UNIT SEQUENCE

Five instructions branch to a new main unit sequence when the appropriate key is pressed: –next–, –next1–, –back–, –back1–, and –stop–. The corresponding keys are: NEXT, SHIFT NEXT, BACK, SHIFT BACK, and STOP.

These instructions take a unit name as their tags. When the student presses one of these keys, execution goes to the unit named in the tag. These instructions may also be used in the conditional form (refer to section 12).

If –next1–, –back–, and –back1– instructions have not been included in a unit, pressing these keys has no effect. If the –next– instruction is not in a unit, pressing NEXT branches the student to the next unit in the physical sequence. When STOP is pressed with no –stop– instruction in the unit, the screen display stops.

The instructions –nextop–, –next1op–, –backop–, and –back1op– also provide branching to a new main unit. These instructions work exactly like –next–, –next1–, –back–, and –back1– except that the screen is not erased when the student is branched to the unit named in the tag of the instruction. The values of system-reserved words where, wherex, and wherey do not change when the new unit is entered. These instructions also have conditional forms.

The system evaluates expressions in the conditional forms of these instructions when it executes the instruction, not when the student presses the corresponding key.

HELP SEQUENCES

A help sequence is an author-supplied sequence of units available to the student through use of the HELP, LAB, DATA, or other specified help keys. The names help sequence and help key are used because these units provide the student with additional information or other help. Since the help sequence is accessed from certain keys, these are called help keys.

There are a number of structural differences between help sequences and auxiliary unit structures. The most significant is that a help sequence is just that: a sequence of one or more units that are executed in order. An auxiliary unit structure, on the other hand, can have a number of units, but they are executed by –join–, –do–, and/or –goto– commands. The help sequence is executed as a series of main units, not auxiliary units. Therefore, a help sequence can be arbitrarily long rather than having a predetermined cutoff point.

-END- INSTRUCTION

A consequence of this arbitrary length is the necessity to specify where the help sequence ends. This is done with the –end– instruction. Where the instruction is

```
end          lesson
```

it indicates the final unit in the logical lesson. After this unit is executed, the student is returned to the system. This form of the instruction is necessary because the end of the logical lesson is often not the end of the physical lesson. The keyword tag is lesson, as given, not the name of the lesson. More than one –end lesson– instruction can occur in a single lesson. The –end lesson– instruction sets ldone to –1 (refer to –lesson– instruction). An –end lesson– instruction in the IEU stops execution of the IEU and begins execution of the first unit of the lesson. The –end lesson– instruction does not execute the finish unit of a lesson.

The form used to end a help sequence has a blank tag or the keyword tag help. The instruction

```
end
```

halts processing in the help sequence. The instruction is executed regardless of the state of the system. No instruction following the –end– instruction is executed. The –end– instruction delimits the unit. Thus, if an –end– instruction follows a response judging instruction, no response judging instructions (or regular instructions) following are perceived by the system as belonging to the same unit.

The –end– instruction with a blank tag affects only help units, not base or auxiliary units. This allows a unit to be in a help sequence, base sequence, and auxiliary unit structure within the same lesson. An –end lesson– instruction in a help sequence behaves like an –end– instruction.

Both forms should be the last instruction in the relevant unit. Since it is simply a place marker, there are no difficulties caused by response judging in the unit.

SPECIFYING A HELP SEQUENCE

The unit from which a help sequence is accessed is referred to as the base unit, since it is the base of the help sequence and is the unit to which the help sequence returns when completed (or when the student presses the SHIFT BACK keys). Only main units can be the base unit for a help sequence, since the base unit marker is not reset for auxiliary units. However, the help sequence can be specified in an auxiliary unit. Although a help sequence can have another help sequence accessible from it, the base unit marker is not reset, so the base unit remains the same. Consequently, a help unit cannot be a base unit, although it is considered a main unit.

In a main unit sequence (other than a help unit sequence), the base unit marker is automatically set to zero. Because such units can act as base units (if a help sequence is accessed), such main sequences are called base unit sequences.

Six help sequences can be attached to any unit. The keys that can be used are the HELP, DATA, and LAB keys. The HELP, DATA, and LAB keys can also be used with a shifted form, such as SHIFT HELP.

The commands all have the same form, exemplified by the –help– instruction.

```
help         unitname
help1        unitname
```

A conditional form is also available (refer to section 12). The –help– instruction enables the HELP key, and the –help1– instruction enables the SHIFT HELP keys.

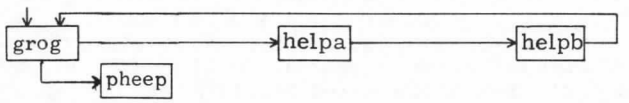
If an x is the instruction tag, the instruction has no effect; if a q is the instruction tag, the relevant marker is cleared.

The SHIFT BACK keys cause a student in a help sequence to return to the base unit. When the student is in a help unit that does not have a –back– instruction, the BACK key operates in the same manner as the SHIFT BACK keys; that is, the student is returned to the base unit.

For example, in the portion of the lesson

```
unit        grog
next        pheep
lab         helpa
  .
  .
  .
unit        helpa
  .
  .
  .
unit        helpb
  .
  .
  .
end
unit        pheep
  .
  .
  .
```

the logical structure is:



If the student presses the LAB key while in unit grog, he is sent to unit helpa. If unit helpa does not contain a –back– instruction, pressing either the BACK or SHIFT BACK keys causes the student to return to unit grog. The NEXT key sends him to unit helpb. Again, the BACK or SHIFT BACK keys returns him to unit grog, but since helpb is the last unit in the help sequence, pressing the NEXT key also returns the student to unit grog. This feature makes help sequences flow back to their base unit.

Additionally, six author language instructions provide help on the same page: –helpop–, –labop–, –dataop–, –helplop–, –lablop–, and –datalop–. (All references to –helpop– apply equally to the other five instructions.)

All six instructions have the same formats as the –helpop– (help on page) instruction.

```
helpop      unitname

helpop      expression,unitname1,unitname2,...
*           $$conditional form
```

A conditional form (second format above) is available (refer to section 12).

The –helpop– instruction enables the HELP key, and the –helplop– instruction enables the SHIFT HELP keys.

Once a –helpop– instruction is encountered, its function remains in effect for the rest of the current main unit. A –helpop q– or –helpop– with no tag clears the previous setting. If another –helpop– or –help– instruction is encountered in the same main unit, the unit named in the tag of the most recent –help– or –helpop– instruction is executed when the HELP key is pressed.

The unit named in the tag of one of these instructions is executed when the corresponding key is pressed. After the helpop unit is executed, control returns to the main unit. There is no full screen erasure when any of these instructions are executed. Any graphics or text in the helpop unit is added to the current display. After the helpop unit is executed, control returns to the unit containing the –helpop– instruction. The graphics and text remain on the panel when control is returned to the main unit.

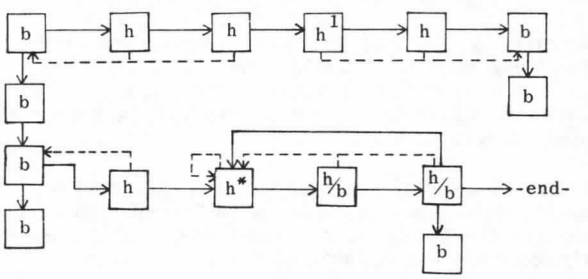
If successive keys are pressed (each activating a help-on-the-same-page sequence), the last displayed text (via –write– or –writec–) from the helpop unit is automatically erased before the new text is displayed from the next helpop unit. If the helpop unit contains more than one –write– or –writec–, only the last displayed text is automatically erased when successive helpop units are accessed.

-BASE- INSTRUCTION

The base unit of a help sequence is not necessarily fixed. It can be changed by use of the –base– instruction. The instruction has the form

```
base        name
```

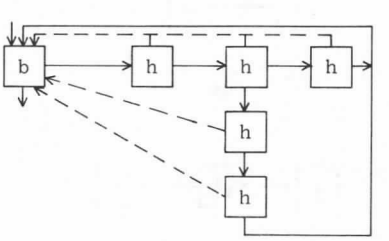
where **name** is the name of the unit which is the base unit. If the tag is blank, the unit being executed is stipulated as the base unit, and execution continues as a sequence of base units. The following are graphic examples.



b indicates base units and h indicates help units. Dotted arrows represent the path of control if the SHIFT BACK keys are pressed, h1 is a help unit defining another main unit as the base unit, and h* is a help unit defining itself as the base unit.

h/b is a unit that is executed once as a help unit (with h* as the base unit) and again as a base unit when the help sequence is terminated by an –end– instruction (or student pressing the SHIFT BACK keys).

As previously mentioned, a help unit possesses the full branching capabilities of the author language. This includes the capability of having help sequences of its own. If such sequences are appended to a help sequence, the base unit does not change. Hence, pressing the SHIFT BACK keys returns control to the same main unit. Graphically, this is exemplified by:



A help sequence attached to another help sequence does not return to the first help sequence but to the base unit. If it is desired to return to the original sequence, the sequencing must be made explicit by the author. This can be done by the use of a –goto– instruction or by a student-initiated branch. Use of the –next– instruction is ignored, as the –end– instruction has precedence.

USE OF THE -TERM- INSTRUCTION

A unit containing an instruction of the form

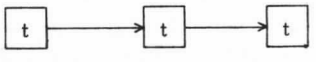
term **name**

can be accessed by the student from anywhere in the lesson by pressing the TERM key and typing the name specified in the tag. If the –term– instruction has a blank tag, the student accesses the unit containing it by pressing the TERM key and typing any name which is not a previously defined term. Previously defined terms include these system terms: consult, talk, calc, cursor, grid, operator, reject, step, comment(s), and time. Descriptions of these terms are in the PLATO User's Guide.

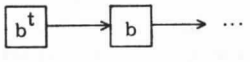
If there is no –base– instruction in the unit thus accessed, the unit functions as a help sequence. In this form, it is especially useful for defining terms used elsewhere in the lesson with which the student may not be familiar. As a help sequence, an –end– instruction is necessary.

The other form of the unit is most useful when a lesson has several base sequences and the student is to be provided an index to these sequences. The use of the –term– instruction allows the student to return to the index from anywhere in the lesson.

Because of the dual nature of the –term– instruction, two representations are possible when drawing the block structure of a lesson. A –term– sequence that is used as a help sequence is represented by:

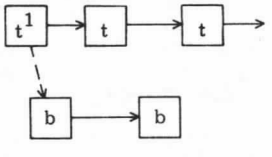


A –term– unit that defines itself as a base unit is represented by:

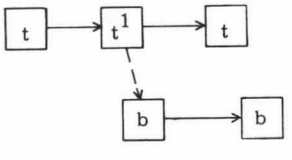


t indicates that the unit is accessible by the TERM key.

Analogously to a help unit, a –term– unit can define another unit as the base unit. This can be represented by:



or



The dashed arrows are necessary in this case to show the unit to which control returns on completion of the –term– sequence.

The –termop– instruction works like the –term– instruction except that the screen is not erased when the student branches to the unit containing the –termop–. The execution of a termop unit is identical to that of a helpop unit.

OTHER CONTROL INSTRUCTIONS

Several instructions permit the author to control other aspects of lesson execution.

COMMENTS

It is often useful to have comments in a lesson explaining the purposes of various portions of the lesson or of specific instructions. The author language contains three methods for inserting comments. While the comments have no effect on lesson execution, they make the source code much more understandable to either another author or the original author after a period of time. As a result, the extensive use of comments is encouraged.

Comments used to explain a large section of code or a complex set of manipulations generally should use one or more full lines. An asterisk or the letter c (followed by at least one space) placed in the first column of a line indicates that the entire line is a comment and is ignored by the condenser. Such comments can be inserted anywhere in a lesson because they are ignored and do not affect lesson execution.

An example of the use of a comment line for purposes of increasing readability, rather than actual comments, is separating units by a line (or partial line) of asterisks. This makes unit boundaries easier to locate, and because of the asterisk in the first column, has no effect on unit execution.

For comments on a single instruction, it is not always necessary to use an entire line. Two consecutive dollar signs after the tag indicate that what follows is a comment and should be ignored.

CONDENSING CONTROL

Three instructions indicate the parts of a lesson that are to be condensed, if it is not desired to condense the entire lesson. All three have a blank tag.

Since the instructions can be somewhat difficult to see in the source code, it is preferable to use the partial condense option of the editor.

The –cstop– instruction causes condensing to stop. Thus, when the lesson is executed, any instructions following a –cstop– instruction are not available for execution.

The –cstart– instruction causes condensing to be done after it has been halted by a –cstop– instruction. All instructions, until the next –cstop– instruction, are condensed. Hence, the –cstart– and –cstop– instructions can

indicate portions of source code in a lesson that are not to be condensed for execution. Since condensing starts automatically when the lesson is called for execution, a –cstart– before the first –cstop– is not meaningful.

The –cstop*– instruction halts all further condensing, regardless of any following –cstart–s. No instructions whatsoever are condensed following a –cstop*– instruction.

REENTERING LESSONS

The –restart– and –status– instructions allow an author to permit the student to reenter a lesson at a point other than at the beginning.

The –restart– instruction allows the author to specify where a student starts the lesson if he did not complete it the first time. If not otherwise specified, the student must reexecute the entire lesson. There are three forms of the instruction.

If the tag is blank, the student starts in the unit containing the instruction when he signs on again. If the tag has one argument, that argument is the name of the unit where the student starts. This unit must be in the same lesson as the instruction. If the tag has two arguments, the first specifies the lesson name and the second the unit name where the student starts. It is not necessary that the lesson be the one containing the instruction. The two arguments are separated by a comma.

For example, if the student is in unit add of lesson math1, and the instruction

```
restart
```

is encountered, the student can then quit, and when he enters the lesson again, start at unit add. The student starts in this unit regardless of whether further units were executed before the original sign-off, as long as another –restart– instruction was not encountered. As a result, the student can be forced to execute the same unit more than once.

If the instruction

```
restart        addhelp
```

is executed, the student, upon entering the lesson math1 again, starts in unit addhelp, regardless of whether he has or has not previously executed this unit.

If the instruction is

```
restart        algebra,add1
```

the student is automatically sent to unit add1 of lesson algebra. The IEU of lesson algebra is executed prior to unit add1.

Since the student is not automatically notified of the execution of a –restart– instruction, the author must so inform the student if it is desired that the student be notified.

The –status– instruction allows the author to specify where a student starts the lesson again after the student has worked partially on the lesson and then executed other lessons. The difference between –restart– and –status– is that –restart– is used when a student quits a lesson and then resumes working on that lesson without executing any other lesson, and –status– is used when a student quits a lesson, executes some other lesson(s), and then resumes working on the first lesson. When using –restart–, the system saves all the student variables and the student status if the student does not execute any other lesson. If the student does execute another lesson, the system saves only a limited amount of information; therefore, the author should not use –status– unless there is a real need for the student to come back to a lesson.

The –status– instruction sets the system-reserved word lstatus to the value of the variable in the tag. The student's router lesson can save the lstatus value and can reset lstatus to that saved value later when the student reenters that lesson. The lesson must check lstatus upon entry of the student and must do its own branching to restart the student correctly.

The lesson author and the router author should agree on the initial value of lstatus. Zero is recommended. Then if lstatus is equal to zero, the lesson does not do any special restarting for the student. When the student completes the lesson, the author can execute –status 0– so that lstatus equals 0.

The system router "mrouter" saves up to eight nonzero lstatus values, allowing students to start again in eight lessons.

-RETURN- INSTRUCTION

This instruction affects the time-sharing aspects of the PLATO system. The tag is left blank. When a –return– instruction is encountered, the lesson relinquishes the central processor. The system behaves as if the lesson has exceeded its current time-slice. Thus, when execution resumes, an entire time-slice is available.

-PRESS- INSTRUCTION

The –press– instruction controls lesson execution by simulating a student response keypress. The tag of the instruction specifies the key. The tag can be either the key code or the name of the key. Use of the key name is preferable, as the meaning is clearer when the lesson code is read.

As a sample usage of the instruction, an author might want a student to enter a help sequence if a certain response to a question is given (for example, pressing the ANS key). Rather than using a –jump– instruction, with the necessity of determining the proper unit to return to when the sequence is completed, the author can use the instruction

```
press          lab
```

(or whatever key name is appropriate). The help sequence is executed as if the student had pressed the appropriate key.

Most of the function keys have predefined names that can be used, as in the previous example. However, most keys do not have such names. In such a case, the character desired is given, placed in double quotation marks. For example:

```
press       "h"
```

This instruction is equivalent to

```
press       o10
```

but is easier to interpret. The named keys do not need the quotation marks, as they are interpreted in the same manner as defined constants.

Each –press– instruction enters a single key code into the student's input buffer. Only one such press can be done per second. As a result, if it is desired to enter two consecutive keys with –press– instructions, the two instructions should be separated by a –pause 1– instruction to ensure proper execution.

The –press– instruction has an optional two-argument form. The optional second argument is the station number at which the key is to be –press–ed. The two-argument form is executed only if both stations are in the same lesson or if the station is routed by the lesson executing the –press–.

System-reserved word zreturn is set to –1 if the –press– instruction is executed; it is set to 0 if –press– is not executed.

-JUMPOUT- INSTRUCTION

The –jumpout– instruction is actually a drastic form of author branching. Execution of a –jumpout– causes the student to leave the current lesson and to enter the lesson specified in the instruction tag.

For a number of reasons, care must be exercised in the use of the –jumpout– instruction. If the lesson specified is not currently condensed, it will be condensed for execution. However, this can cause a delay of up to several seconds. A delay this long is apparent to the student.

If a unit name is specified, the –jumpout– codes of the lesson must match. If the lesson to which the –jumpout– is performed has no –jumpout– code, any lesson can perform a successful –jumpout–.

Any character set used in the original lesson is still used, unless the new lesson initializes a new set. The standard micro table is loaded between –jumpout–s.

In contrast, any common or storage used is not saved. This requires some care if information contained in common is to be retained. Storage may be retained by using the –inhibit dropstor– instruction.

If sufficient ECS for the new lesson is not available, the –jumpout– is ignored. The author can then test, in the instructions following the –jumpout– instruction, if the –jumpout– has occurred.

The ECS check can be stopped by the instruction

```
inhibit     jumpchk
```

but caution should be used, since the new lesson cannot be used if insufficient ECS is available.

The –jumpout– instruction, particularly when used in conjunction with the –restart– instruction, allows much more flexibility in lesson construction by allowing lessons to be interconnected. Because of initialization factors, caution should be exercised in the use of these instructions.

-INHIBIT- INSTRUCTION

Various standard processes of the author language can be disabled by the use of the –inhibit– instruction. For example, if it is desired to request a student response without showing the arrow at the position where the response appears, the instruction

```
inhibit     arrow
```

placed before the –arrow– instruction suppresses the arrow on the screen.

The tag of the instruction determines which option or options are to be inhibited. If the tag is blank, all previous inhibits are cancelled. When a new main unit is entered, all –inhibits– are cancelled.

An –inhibit erase– instruction prevents the usual full-screen erase when a new main unit is entered. Since all previous inhibits are cancelled in the new main unit, erasure occurs following the new unit, unless a new –inhibit– instruction is given.

An –inhibit dropstor– instruction retains storage after a –jumpout– instruction.

An –inhibit edit– instruction deactivates the EDIT key, and an –inhibit term– deactivates the TERM key.

The –inhibit jumpchk– instruction ignores ECS check before a –jumpout– instruction.

If it is not desired to accept a blank answer to a request for student response, the –inhibit blanks– instruction is used.

The –inhibit from– instruction prevents setting the from-reserved words (refer to the –from– instruction).

The –inhibit anserase– instruction prevents the erasing of any answer-contingent writing after a no judgment when the student presses the NEXT, ERASE, or SHIFT ERASE keys.

The –inhibit charclear– instruction prevents the charset flag from being cleared. This is used preceding a –char– instruction to prevent the necessity to reload an alternate character set.

The –inhibit dropset– instruction retains the current dataset connection and prevents the release of any reserved records over a –jumpout–.

-KEYLIST- INSTRUCTION

The –keylist– instruction is useful when establishing a list of keys to be used in the –pause– and –keytype– instructions. Since –keylist– is a nonexecutable instruction, it should be placed in the IEU.

The instruction's tag consists of two or more arguments: the first is the actual name of the list, and the second and/or successive arguments are the keys to be in that list.

List names must be from two to seven characters in length, and lists can be combined.

Five system-defined lists are available.

numeric	Digits (0 through 9)
alpha	Alphabet (a through z and A through Z)
funct	Function keys (BACK, HELP, DATA, and so on)
touch	Touch panel input (256 ≤ key ≤ 511)
ext	External inputs (512 ≤ key ≤ 767)

-PAUSE- INSTRUCTION

The –pause– instruction has four forms. If the tag is blank, processing is suspended until the student presses any key on the keyboard. If the tag is a number, execution halts for the number of seconds specified. Thus, the instruction

```
pause    3
```

causes lesson execution to be suspended for 3 seconds, while the instruction

```
pause
```

causes suspension of execution until the student presses any key on the keyboard. The minimum time that can be specified in the tag of a –pause– instruction is 0.75 second.

The third form of the –pause– instruction uses a key list. For example:

```
pause    keys=next,back
```

With this example, lesson execution is suspended until one of the tested keys (NEXT or BACK) is pressed. If a branching key such as BACK or HELP is listed, and if that branching instruction is in effect in the unit, pressing that key branches the student to the unit specified in the instruction tag. An exception is the NEXT key which terminates the pause and does not branch.

The fourth form of the –pause– instruction is a combination of a numeric tag and a key list. For example, the instruction

```
pause    3,keys=touch
```

suspends lesson execution for 3 seconds or until one of the keys in the group touch is pressed.

An entry in the key list can be a list name. Five system-defined list names are available (refer to the –keylist– instruction above). In addition, the keyword all specifies that all keys terminate the pause.

If the author wishes to perform actions that use the key the student pressed to decide the particular course of action, the system-reserved word key contains the key code of the key pressed. The –keytype– instruction may also be used.

-COLLECT- INSTRUCTION

The –collect– instruction collects several keys at the same time. It executes faster than an equivalent loop using the –pause– instruction. The key inputs are stored one per variable, and as many as 20 may be stored. A preceding –enable– instruction is required to receive touch or external inputs. The –collect– instruction terminates when the specified number of keys have been received or when the keyname timeup occurs as a result of a previous –time– instruction. In the later case, the timeup key is also stored.

-KEYTYPE- INSTRUCTION

The –keytype– instruction provdes an easy method for determining which key from a specified list has been pressed. For example:

```
group      mine,a,b,c,d,w
keytype    n10,mine,funct,z,o
```

If the pressed key is in the tag of –keytype–, the variable named as the first argument is set as follows:

Key Pressed	n10	Comment
z	2	Listed key
d	0	Defined with –group–
NEXT	1	System-defined group
w	0	Defined with –group–
LAB	1	System-defined group
o	3	Listed key

The search for a listed key is made from left to right, stopping when the key is found. If no key is found, the variable is set to –1.

If a –keylist– name or system-defined list is used, a value is assigned to the variable for any of the entries in the –keylist–.

If external keys or variables are to be checked, use an instruction similar to

```
keytype    n7,ext(n3),(n14)
```

This checks the 10 bits of reserved word key. If the first bit is 1, an external input is indicated and the remaining bits are compared with the value in n3. The key may also be compared with the value in n14. The value in n14 may be any type of input: external, touch, function, numeric, or alpha. Parentheses are required in this instruction, and any of these values may be expressions.

If touch panel input is used, the instruction may look like:

```
keytype      n7,touch(1419;4,2)
```

The fine- or coarse-grid tolerances are optional and are separated from the address by a semicolon. Either t or touch may be used.

-FORCE- INSTRUCTION

The –force– instruction initiates actions in the author language. The instruction can have six arguments in its tag.

A –force font– instruction causes any characters the student enters to be in the alternate character set, unless the student presses the FONT key. A –force micro– instruction causes all keypresses to be sent through the micro table. A –force long– instruction causes judging to be initiated when the student response length-limit is reached. The –force left– instruction causes any response the student enters to appear right to left on the screen.

The –force firsterase– instruction allows the student to type a new response after a no judgment without first pressing the NEXT or ERASE key. The first character of the new response erases the entire old response and any wrong-response-contingent writing. Additionally, the –force firsterase– instruction executes an –eraseu– unit if required.

All –force– options are cancelled by a –force– instruction with a blank tag, by –force clear–, or when a new main unit is entered.

-CHANGE- INSTRUCTION

The –change command– instruction allows the author to redefine the names of normal author language instructions to names of his own devising. This can be especially useful in the case of language lessons, where use of the –change command– instruction at the beginning of the lesson (in the IEU) can allow virtually the entire lesson to be written in the language with which the lesson deals (refer to section 11 for the –change symbol– instruction).

The tag of the instruction consists of the word command, followed by the old instruction command, followed by the word to, and ending with the new name for the instruction command, all set off by spaces. An example is:

```
change       command at to wo
             command write to schreib
```

These instructions could be used in a lesson teaching German (or a lesson teaching English as a second language with German as the basis) to replace the –at– and –write– instructions. The lesson is then more readable to persons familiar with German and more closely resembles its subject matter.

One of the disadvantages of using the –change command– instruction in this manner is that it makes the lesson difficult to understand for persons not familiar with the language to which the instructions are changed.

When instructions have been altered by use of the –change command– instruction, it is imperative that all subsequent uses of the instructions use the new definitions. Otherwise, the lesson does not execute properly and may cause a fatal execution error. It is, of course, possible to use the –change command– instruction to return the instructions to their original definitions.

-USE- INSTRUCTION

It is often desirable to use parts of other lessons rather than duplicate the source code. This can be done with the –use– instruction.

The effect of the –use– instruction is to insert the source code from the accessed lesson at the position of the –use– instruction when the lesson is being condensed.

The tag of the –use– instruction specifies the name of the block which is to be condensed as part of the current lesson. The name of the lesson to be –use–d is specified on the Author Information page of the lesson which contains the –use– instruction. Only one lesson may be –use–d per lesson.

Consecutive –use– instructions allow more than one block to be accessed. If consecutive blocks share the same name, a single –use– instruction accesses all of them.

The –use– instruction is particularly helpful when using large vocabularies. By means of the –use– instruction, the vocabularies need only be written once, thus saving the author time while writing the lessons and also saving storage space.

The –use– security codes of the two lessons must match. If the code of the lesson –use–d is zero, the –use– instruction is permitted.

-STEP- INSTRUCTION AND TERM-STEP OPTION

Both the –step– instruction and the TERM–step option enable an author to execute his lesson instruction by instruction. Executing a lesson one instruction at a time helps the author find program errors. When stepping through a lesson, the lower lines on the screen display the next instruction to be executed, the current state (regular or judging), and the base, main, and current unit. Student variables may be examined at any time.

To enter the step mode while viewing a lesson in the student mode, the author presses the TERM key and then types the word step. The bottom lines on the screen are erased, and the step information is plotted.

The step mode is also entered by the author placing a –step– instruction in his lesson. The –step– instruction can have a keyword tag of on or off, or the tag can be an expression with 0 equal to off and non-0 equal to on.

While in step mode, the instruction listed is executed when the NEXT key is pressed. At each press of NEXT, the instruction presently listed is executed, and the succeeding instruction is then displayed.

The waiting for key message means the system is waiting for a student's response or keypress. A response should be entered and the NEXT key should be pressed to have the response judged and step mode continued.

The arrow character at the bottom of the screen enables the author to see the current values of any of the student variables. The author should press the a, o, v, or n character to indicate the format and then the number of the specified variable.

By typing the character s and a number at the arrow character, the lesson is skipped forward by the number of instructions specified before reentering the step mode.

Step mode is exited by pressing the BACK key.

Only authors can use step mode. The author's security code must match the change code of the lesson. Step mode cannot be entered nor is it operable when a student enters TERM-step or when a student encounters a –step on– instruction.

Common variables and storage variables cannot be inspected in the step mode.

-INITIAL- INSTRUCTION

The –initial– instruction specifies a unit for execution when a lesson or common is entered by the first user. If the lesson or common is already in ECS, the –initial– unit is not executed.

The formats for the –initial– instruction are as follows:

initial	lesson,**unit**
initial	common,**unit**

The tag for the instruction consists of two arguments. The first argument is a keyword indicating the type of –initial– command (lesson or common) and is not the name of the lesson or the name of the common. The second argument specifies the unit to be executed after the –initial lesson– or –initial common– is executed.

The first user to execute an –initial– instruction causes the initial unit (lesson unit or common unit) to be inserted (like a –do– instruction) at the location of the –initial– instruction. Any subsequent encounters of –initial lesson– or –initial common–, whichever applies, are ignored.

If the initial unit contains a –jump– or –jumpout– instruction, the completion flag for the –initial– instruction is never set, thus preventing all other users from continuing in the lesson.

-LESSON- INSTRUCTION

The –lesson– instruction assigns a value to the system-reserved word ldone. Upon entering a lesson, the system router checks to see if the lesson is completed. If the lesson is completed, ldone equals –1, and an asterisk is placed next to the lesson on the student's index (sequence). However, if the lesson is not completed, the value of ldone is set to 0.

The tag for the instruction consists of a single keyword argument indicating completed, incomplete, or no end. The tag no end is used for instructional lessons which have no logical end, and it sets ldone to 1.

A conditional form of the –lesson– instruction is available (refer to section 12).

-SCORE- INSTRUCTION

It is often necessary for an instructor to determine if a student has worked on a particular lesson and what relative progress the student has made on that lesson. The –score– instruction can be used for that purpose.

The –score– instruction assigns a value to the system-reserved word lscore. The value of lscore can then be stored in any data base or status bank (student, common, router, and so on) for the student.

The tag for the instruction can be either a constant or any expression from –1 to 100 indicating the value to be placed in system-reserved word lscore. Values are rounded to the nearest integer.

Any negative score is interpreted as do not store any score. Any score that rounds to a value greater than 100 produces an execution error.

A –score– instruction with a blank tag assigns a value of –1 to system-reserved word lscore. If lscore already has a value and a –score– instruction is then executed, the new value overwrites the previous value; lscore contains the new score. However, if a –score– instruction has not been executed for a lesson, lscore has a value of 0.

If the students are routed by the system router, scores are stored as part of the student's permanent record. These scores may be displayed through an option of the student roster. If the score is negative, no score is displayed. The scores are kept only as part of the permanent record if the system router is used. When the system router is not used, the scores may be stored in some status bank (common, router variables, and so on). Only the most recent score for each lesson is stored as part of the permanent record.

When creating a router lesson, the –score– instruction can be used to place a value in lscore, with the value representing the status for the lesson. Branching could then be done on that condition.

-BACKGND- AND -FOREGND- INSTRUCTIONS

The –backgnd– (background) instruction allows the user to obtain more CPU time during each time-slice if extra CPU time is available (PLATO system not busy). With a –backgnd– instruction in effect, sections of lessons receive more CPU time if it is available.

If the system is busy and –backgnd– is in effect, the lesson receives less than the average processing time. This instruction should therefore not be used with lessons that are to be used by students in an instructional setting.

The –foregnd– (foreground) instruction cancels the effect of a –backgnd– instruction and indicates the end of a section of a lesson that is to be run on the background priority.

The foreground priority is the normal state. Both –foregnd– and –backgnd– are executable instructions, so portions of lessons can be specified to run on either priority. It is possible to switch between background and foreground priorities in the same lesson by executing a –backgnd– or –foregnd– instruction.

No tags are used with either the –backgnd– or the –foregnd– instruction.

-CPULIM- INSTRUCTION

The –cpulim– instruction should be used if an author wishes to place a limit on the CPU time for a given lesson while that lesson is being used by students. The CPU time in milliseconds/second is listed in the student records and at sign-off time. If this instruction is not used, the CPU limit is 10 TIPS.

A maximum limit on the CPU time allows the lesson author to test a lesson at a low CPU time maximum and decide if there is any effect on a lesson.

LESSON ROUTING

A router allows the management of student progress in a course by sequencing lessons according to student performance. An author should write a router for a course only after thoroughly studying the features of the system routers and after determining that the course requires additional capabilities.

Author-initiated routing and system routing are described and their use explained in the PLATO User's Guide. The following is a summary of router features.

- Automatic entry into the router at sign-on time.
- Automatic return to the router when a lesson is complete.
- All –route– instructions are functional.
- All –allow– instructions are functional (no –transfr–s with router variables or common without –allow–).
- Router variables can be used (maximum of 50 per student)

Routers remain in ECS as long as a student from the course is signed-on. The system router, maintained by system-support personnel, requires a minimum of ECS.

A user-written router should execute the following instructions before doing a –jumpout– to a lesson for the first time.

```
restart  (0), (0)
lesson   incomplete
score
status   0
```

These instructions initialize the values of the system-reserved words rstartl, rstartu, ldone, lscore, and lstatus. If the router does not initialize these system-reserved words, the lesson receives values from a previous lesson, possibly causing errors. If the router saves the new values of these system-reserved words upon return from a lesson, the router can give the values back to the lesson the second time the student enters it.

-ROUTE- INSTRUCTION

The –route– instruction is used in the router to specify which units of the router are to act as reentry units when the student exits from one of the instructional lessons in the router.

The instruction tag for –route– consists of two arguments: the first argument is a keyword (end lesson, finish, error, or resignon) which specifies the type of exit from an instructional lesson, and the second argument specifies the entry unit in the router when the corresponding exit occurs. The –route– instruction has five forms.

```
route    end lesson,unit
route    finish,unit
route    error,unit
route    resignon,unit
route    resignon
```

The **unit** specified in the second argument of the instruction is executed as follows:

Keyword	When executed
end lesson	When the student leaves the instructional lesson as a result of –end lesson– or –jumpout q–, the router unit, **unit**, is executed.
finish	When the student leaves the instructional lesson as a result of SHIFT STOP, the finish unit in the instructional lesson is executed first and then the router unit, **unit**.
error	When an execution error occurs in the instructional lesson, the router unit, **unit**, is executed. (The system-reserved word errtype is also set.)

Keyword	When executed
resignon	Same sequence as for keyword finish. If **unit** is not specified, the IEU and the first unit of the router are executed. The student is also given the option of returning to the router or of signing off the system.

The –route– instruction must be executed each time the student is in the router for the specified units to be functional. Placing the –route– instructions in the router's index or decision units sets the corresponding flags each time a lesson is selected by or for the student.

When an execution error occurs and a –route error– instruction is in effect, system-reserved word errtype is set to one of the following values.

Value	Meaning
0	Unknown error
1	Execution error
2	Fatal condense error
3	Memory allocation exceeded
4	Error in –finish– unit of instructional lesson
5	SHIFT STOP from the condense queue

A –jumpout– to another router causes an execution error (errtype set to 2).

If errtype is set to 4, the student is returned to the unit of the router named in the second argument of the –route finish– instruction.

An errtype of 5 occurs only if the student exits from the condense queue with a SHIFT STOP and the student has been waiting in the queue for more than 30 seconds.

When a student returns to the router, the router's IEU is not usually executed. The router's IEU is only executed when the session begins (sign-on time), when –route resignon– without a specified unit is executed, and when a lesson does a –jumpout– to the router.

For the –route– instructions to be in effect at all times, the student must execute the –route– instructions each time he is in the router.

Some of the key locations for placing the –route– instructions are illustrated in figure 10-1.

-ROUTVAR- INSTRUCTION

In addition to the 150 student variables available, there can be up to 50 additional variables called router student variables. Like student variables, each student in an instructor's course has these router student variables.

The –routvar– instruction is used by the instructor to specify the number of router student variables to be made part of the permanent student bank and retained between sessions. Authors may not have router student variables.

The router student variables are referred to as nr or vr locations and can be used in the router in any manner as the n or v variables can be used. The –routvar– instruction should be executed before the nr or vr variables are referenced.

The tag for the –routvar– instruction is an expression specifying the number of router student variables (50 maximum). These variables are added to the amount of ECS used at an instructor's logical site; therefore, 20 students with a –routvar 40 – instruction in effect require 800 additional words of ECS.

Router student variables can be altered only in the router. With an –allow read rvars– instruction in effect, the values can be read by an instructional lesson.

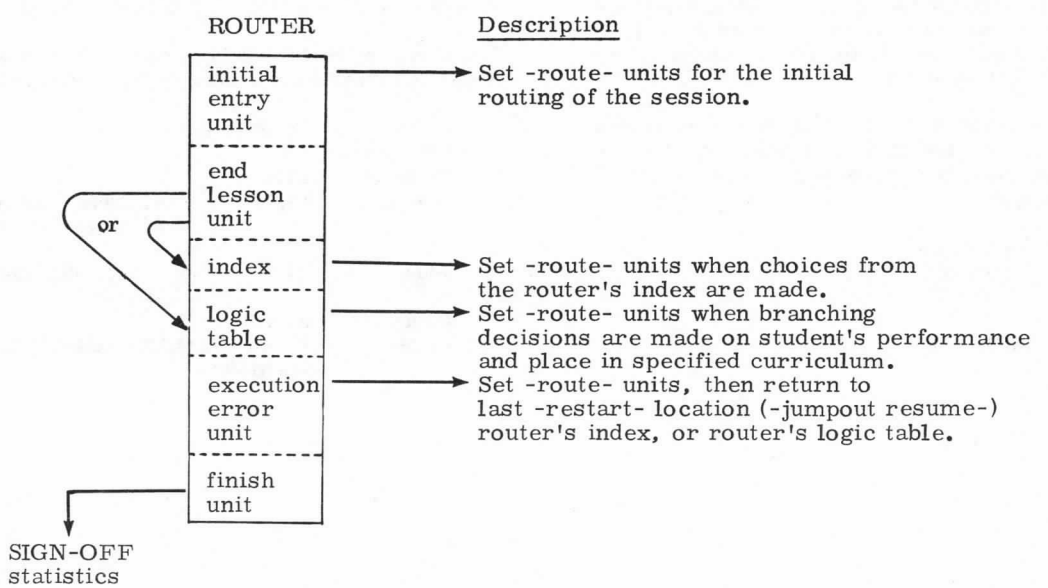


Figure 10-1. Key Locations for Placing –route– Instructions

-ALLOW- INSTRUCTION

Instructional lessons can reference common variables that the router uses. To allow this, an –allow– instruction must be previously executed in the router.

The instruction tag for –allow– consists of a single keyword argument (read, write, or read rvars). The –allow– instruction with a blank tag clears –allow– settings. An asterisk merges successive –allow–s.

An –allow read– instruction permits read-only access to the router common. An –allow write– instruction, however, permits read and write access to the router common. The –transfr– instruction is then used to write the router common.

The –allow read rvars– instruction permits read-only access for the router student variables, as established by a –routvar– instruction. The router student variables can be read by using the –transfr– instruction.

The –allow– instruction is meaningful only when executed by students in a course using that router and in the router itself (not instructional lessons).

If an –allow– instruction is not present in the router lesson, an attempted –transfr– with router common or router variables yields an execution error.

In the router, nr, vr (student), nc, vc, and c,1 (common) locations should be referenced directly with a –transfr– instruction.

LESSON LISTS

A lesson list (leslist) is a special block used to maintain a list of lesson names. This list can be referenced by variables used with the –jumpout–, –from–, –restart–, and –lessin– instructions.

The leslist is two blocks long and stores 320 lesson names. The lessons are numbered 0 to 319.

Leslists are created so authors do not have to be concerned with system lesson-naming conventions, which are subject to change. Since lessons are accessed by position in the list rather than by name, the actual format of the leslist is unimportant for users.

A lesson list for a user's file (one of the type-of-block options) is created by the author language editor. The contents of the block is displayed as a list of lesson names. For example:

```
0       spanish202
1       german13
2       german7
3
4       german3
5
6
```

Directions for editing the list are in the leslist block. Lessons are added at the first empty slot. Deleting a lesson leaves a blank entry. In the example above, lesson 3 is empty. The next lesson to be added is placed in the first empty slot, 3.

Leslists can also be altered (edited) by using the –addlst– and –removl– instructions from the student mode.

The leslist editor does not do a lesson-name check; it assumes the lesson names entered actually exist.

-LESLIST- INSTRUCTION

The –leslist– instruction allows user access to the specified leslist. Only one leslist at a time may be used. The –leslist– instruction must be executed before any references are made to the lesson list. The –leslist– instruction is an executable instruction.

The tag for –leslist– consists of two arguments: the first specifies the lesson name or a variable containing the lesson name, which contains the leslist. If the referenced leslist is in the lesson that uses it, this argument is optional. The second argument specifies the actual block name or variable containing the block name.

For example:

```
leslist     mylesson,mylist
*common codes must match
```

The common access code words for the current lesson and the lesson containing the leslist blocks must match.

-ADDLST- INSTRUCTION

The –addlst– instruction has a one- and two-argument form.

The one-argument form adds a lesson name to the next open slot on the leslist. With either form, the first argument must be the first of a set of three consecutive variables (n or v type) specifying a valid lesson name.

For example, the following code adds a lesson to the leslist and finds the position in which it was added.

```
leslist     myless,myllist
* more code
arrow       1430
storea      n4,30       $$reserves three
*                       $$consecutive variables
ok
addlst      n4          $$n4, n5, and n6 are
*                       $$used
findl       n4,n130
write       lesson ⊲a,n4,30⊳ added as number
            ⊲s,n130⊳
```

The two-argument form adds a lesson name to the specified slot (**position**) on the leslist. The second argument (**position**) may be a variable or expression. Blank slots may be left, but the lesson named in the first argument is not added to the list if that slot (specified in the second argument) is occupied.

An –addlst– instruction with the actual lesson named in the tag (not a variable) produces an error. Tags such as "lesson", 'lesson', <lesson>, <'lesson'>, or <"lesson"> are illegal.

Additions to a leslist should be done by storing the information with a –storea– instruction using a character count of 30. The –storea– instruction must precede the –addlst–.

-REMOVL- INSTRUCTION

The –removl– instruction is used to delete lessons from a leslist; a blank entry is left in that list position.

The tag for the –removl– instruction specifies the lesson to be removed from the list; the tag can be a number, an expression, or a variable. Tags such as "lesson" or 'lesson' do not remove the lesson from the list. The tag is interpreted as a number, and the lesson corresponding to that number is removed.

For example, the instruction

```
removl      8
```

removes lesson number 8 from the leslist.

-LNAME- INSTRUCTION

The –lname– instruction permits the user to place entries from his leslist into variables.

The first argument of the tag specifies the first of three consecutive variables required to place the leslist information. The second argument specifies the lesson number in the leslist and can be a constant, a variable, or an expression. The lesson number can be from 0 to 319 for a two-block lesson list.

The leslist information is displayed with a –showa– instruction using a character count of 30.

For example, the following instructions

```
lname       n10,1
showa       n10,30
```

place the information from position number 1 of the leslist in variables n10, n11, and n12.

-FINDL- INSTRUCTION

The –findl– instruction determines for a user whether a specified lesson name is included in his leslist.

The first argument of the tag is an initial variable for a set of three consecutive variables (n or v type) specifying the lesson name.

The second argument is the return variable containing the position of the named lesson in the leslist.

If the specified lesson is not in the user's leslist or a leslist is not in use, a value of -1 is returned.

All lesson names used with the –findl– instruction should be specified by storing the lesson name with a –storea– instruction using a character count of 30.

The heart of a computer assisted instruction (CAI) system is judging student responses. This is what differentiates a CAI system from a textbook, because a textbook cannot give immediate feedback to student responses.

The first necessity is to inform the student and the system that student input is desired. This is done with the –arrow– instruction. The tag of the instruction is the location at which the student's answer is to appear; it can be in either fine or coarse grid. The author language executer automatically plots an arrow indicating the location, unless the author specifies otherwise with an –inhibit– instruction (refer to section 10).

Besides indicating the desire for a student response, the –arrow– instruction also sets a default length for the student response of 150 characters, disables the copy option, and specifies the NEXT key as the only key that starts the system judging the student response. All three features can be overridden by the author through the use of the appropriate instructions (–long–, –copy–, and –jkey–, respectively). Finally, the –arrow– instruction is used by the author language executer as a marker for the beginning of the response judging portion of the unit. Since this portion can be executed several times, it is important to have a marker that prevents the entire unit from being reexecuted.

At times it is desirable to have more than one student response in a single unit (accomplished with more than one –arrow– instruction in the unit) or to perform some regular instructions at the end of the unit, regardless of how the –arrow– instruction was satisfied. For this purpose, the –endarrow– instruction is provided. The tag of the instruction is blank. The –endarrow– instruction itself serves as an indicator that the author language executer should not go past this instruction in trying to satisfy the –arrow–; it also serves as a location point for beginning execution when the –arrow– has been satisfied. If no –endarrow– instruction is present, the PLATO system, if necessary, searches to the end of the unit in trying to satisfy the –arrow– instruction. Because of the text-insertion characteristic of the –join– and –do– instructions, an –endarrow– should always be used if arrow processing is contained in an auxiliary unit.

EXECUTION OF RESPONSE HANDLING

In the most common form, a student must give an answer that the lesson recognizes as correct before the student leaves the part of the lesson containing the instructions following the –arrow– instruction and preceding the –endarrow– instruction, or the end of the unit if no –endarrow– instruction is present. Since there are many more wrong answers than right, it is often necessary for the instructions to be reexecuted.

When the –arrow– instruction is encountered, the limits and actions previously described are done. All regular instructions immediately following the –arrow– instruction are then executed. When the first of the judging instructions is encountered (refer to table 11-1), the system is changed from the regular to the judging state. Henceforth, with one exception described later, no regular instructions are executed. The system waits, upon changing states, for the student to type in his response. When this has been done, the system searches the judging instructions, in order of their occurrence, to try and match the response. If no match is found, the response is judged no, and the student must enter a different response. If a match is found, any regular instructions immediately following the matched instruction are executed. This, combined with the ability to recognize specific wrong responses, enables the author to comment on the student response. Display instructions such as –write– are automatically positioned following a matched response, unless overridden by an –at– instruction. The position is three lines below the student response, beginning in the same column position.

TABLE 11-1. JUDGING INSTRUCTIONS

Instruction	Effect†	Instruction	Effect†
ans	s	ok	a
ansu	s	open	n
ansv	s	or	n
answer	s	put	n
answerc	s	putd	n
bump	n	putv	n
close	n	specs	n
concept	s	store	s
exact	s	storea	n
exactc	s	storen	s
exactv	s	storeu	s
ignore	a	touch	s
join	n	touchw	s
loada	n	wrong	s
match	a	wrongc	s
miscon	s	wrongu	s
no	a	wrongv	s

†a Always ends judging
n Never ends judging
s Sometimes ends judging

The exception mentioned previously occurs with the –specs– instruction. The –specs– instruction is a judging instruction that can be used to disable standard judging options or to enable others. In addition, it serves as a locator. After a student response has been judged, whether it is judged correct or incorrect, all regular instructions immediately following the –specs– instruction are executed. This is done after the regular instructions, if any, following the matched answer are executed. Since this execution is done for all matched answers, it is done

for responses that have been judged ok as well as for those judged no. If the response has been judged ok, the arrow is satisfied when the regular instructions, if any, following the matched answer and the -specs- instruction (if present) have been executed.

Unless prevented by the tag of the -specs- instruction, when all instructions following a matched judging instruction and all regular instructions, if any, following the -specs- instruction have been executed, an ok or no is placed on the screen following the student response. This is done last so that any modification of the original judgment (made possible by the -judge- instruction) is reflected in the message displayed. Thus, a response can be judged ok, processing done on the response, and the response judged again, yielding a judgement of no, and only the no message appears on the student's screen.

The standard ok and no messages can be changed with the -okword- and -noword- instructions, respectively. The tag for these instructions specifies the new message desired by the author. The tag with the new message, however, must be less than nine characters. Shift and font characters are allowed and are counted as part of the nine characters.

When the -arrow- has been satisfied (that is, the student response judged ok) and if there is no -endarrow- instruction, the next unit is entered. If an -endarrow- instruction is present, any regular instructions immediately following the -endarrow- instruction are executed. If another -arrow- instruction is then encountered, processing occurs as previously. If there is no other -arrow- instruction, the unit has been completed.

When the unit completes execution, whether through satisfaction of an -arrow- with no corresponding -endarrow- or through the execution of the regular instructions following the last -arrow- and -endarrow- pair in the unit, the system then waits for the student to press the NEXT key. When this is done, the next unit is initialized.

This execution can be complicated by the presence of a -join- instruction. The -join- instruction is unique, since it is both a regular and a judging instruction. In fact, it is also executed in the search state when the system is looking to see if another -arrow- instruction should be executed before leaving the unit. As a result, a unit containing judging instructions that is -join-ed to another can cause the execution of the originating unit to be somewhat different than intended by the author.

As an example, a -join- following a -specs- instruction is executed when first encountered, since the -join- instruction is a judging instruction. The contents of the unit that has been -join-ed is then examined, and any judging instructions are executed as if they were contained in the original unit. This includes any -join- instructions in the -join-ed unit. If a response-judging instruction matching the student response is found in the -join-ed unit, the appropriate action is taken, as if the answer were in the original unit. Then, if there is a -specs- instruction, all regular instructions immediately following the -specs- instruction are executed, including any -join- instructions. This can include the -join- that contains the answer that was matched to the student response. Any regular instructions in a -join-ed unit following a -specs- instruction are executed, down to the first judging instruction, if any. If the -join-ed unit contains no judging instructions, the entire unit is executed, and any regular instructions following the -join- instruction are then executed. If the -join-ed unit contains any judging instructions, execution of regular instructions halts when the first is encountered. Regular instructions following the -join- instruction are not executed.

In general, the execution of the response judging area of a unit that contains a -join- instruction can be understood by thinking of the -join- instruction as a text-insertion device that executes in all states of the system (refer to section 10).

Additionally, the -iarrow- instruction inserts the unit named in the tag after the -arrow- instruction and just before the first judging instruction for that -arrow-. For example,

```
unit     test
iarrow   iarrow
arrow    1115
-->
specs    bumpshift
*more judging instructions
```

inserts unit iarrow (with -join-) after the -arrow- instruction and before the first judging instruction (-specs-) for the -arrow-.

The iarrow unit is in effect for all -arrow- instructions encountered after the -iarrow- instruction and while still executing in the same main unit. However, the -iarrow- instruction clears when a new main unit is started. A later occurrence of an -iarrow- instruction in the same main unit overrides any earlier setting.

An -iarrow- instruction with a blank tag (or q) turns off the insertion feature.

Placing the -iarrow- instruction in the imain unit causes an iarrow unit to be effective for an entire lesson.

A conditional form of the -iarrow- instruction is available (refer to section 12).

An alternate arrow character may be set up so that a lesson can use two different arrows. The -arheada- instruction specifies the alternate character. This is usually placed in the IEU. The -arrowa- instruction displays the alternate character at the specified location and continues executing like an -arrow- instruction. The -iarrowa- instruction inserts the specified unit after the -arrowa- instruction and operates analogously to the -iarrow- instruction.

INITIATING JUDGING

When a student types in a response, the PLATO system makes two copies. One copy is stored for later reference and for possible use in conjunction with instructions such as -edit- and -copy-, while the other copy, called the judging copy, is used for determining how well the response fits the answer (or answers) specified by the author. It is extremely important to remember that this form of answer judging is not the same as determining if the student response is correct. If the student responds

with a word or phrase that, while correct, was unanticipated by the author, the response is still judged no (incorrect).

Judging usually starts when the student presses the NEXT key, indicating that he has completed typing his response. The maximum length of the student response permitted is 150 characters, unless otherwise specified by the author through use of the –long– instruction. If this instruction is used, the tag gives the number of characters that the student response is allowed to contain as a maximum. The default limit of 150 characters can be either decreased or increased to as many as 300 characters. Since the –arrow– instruction resets the default length limit, the –long– instruction, if used, must follow and not precede the –arrow– instruction to be effective.

Specification of a single character as the maximum student response length causes special action to be taken by the system. First, despite the limit, a capital letter can be used as a response. A capital letter is stored as two characters (refer to section 9), so that a limit of one character would, except for this special handling, not allow the use of capital letters. If the limit is greater than one, a capital that would cause the response to exceed the limit is ignored, so it does not appear on the screen. Second, judging is started automatically when a response length limit of one character is given. In other cases, the student must press the NEXT key to start judging, but this is done automatically by the system if the limit has been set to one by a –long– instruction.

The number of characters in the student response is kept in the system-reserved word jcount. As with all system-reserved words, jcount can be used in expressions, so it is possible for the author to test the student response for length.

Although response judging normally is not automatically initiated when the response length limit is reached (except in the case where the limit is one), judging can be forced to start when the limit is reached by use of the –force– instruction (refer to section 10 for a full description of the –force– instruction). The instruction is:

```
force        long
```

The –jkey– instruction allows the author to specify keys other than the NEXT key for initiating judging. The NEXT key remains active. The tag of the instruction gives the keys that initiate judging. The keys are specified by name. For example:

```
jkey         back,help1
```

help1 represents a shifted HELP key. Only function keys, such as ERASE, BACK, and so on, can be used as the named keys in a –jkey– instruction. The function keys are those with key codes greater than o200. If more than one –jkey– instruction is used with a single –arrow– instruction, the –jkey– instructions are merged, so any key specified by any one of the –jkey– instructions initiates judging. The –jkey– instruction is a regular instruction. The –arrow– instruction resets the –jkey– options, so the instruction should follow the –arrow– instruction and must be included following each –arrow– instruction for which the options specified are desired. (Refer to –iarrow– for another method.)

MANIPULATING THE STUDENT RESPONSE

The response made by the student is not necessarily in the form that the author desires it to be for judging. As a result, instructions are included for modifying the student response prior to actual judging of the response. However, these instructions, since they work with the student response, are judging and not regular instructions. The author should exercise some caution in their use, since it is possible to alter the student response beyond recognition.

These instructions never end judging. Therefore, if the author desires to perform more complex operations on the student response, it is necessary to use one of the judging instructions described later in the section to end judging before the manipulation can be done. The –judge– instruction (covered in detail later in this section) allows judging to be restarted when the manipulations are finished.

The manipulation judging instructions fall into three classes: storing a student response, making a judging copy of stored characters, and directly altering the judging copy.

STORING A STUDENT RESPONSE

There are two ways of storing a student response. The most often used is the –storea– instruction, which stores the student response in the variable specified with any excess stored in the following variables. The –open– instruction puts one character into each variable.

The –storea– instruction specifies the variable in which the response is stored as its first argument. Since the student response is often more than 10 characters in length (the maximum number of characters that can be stored in a single variable), the variables immediately following the one specified hold the remainder of the response, if necessary. The –storea– instruction produces left-justified, zero-filled storage.

For example, if the student response is 23 characters long, the instruction

```
storea       n11,30
```

stores the response in variables n11, n12, and n13. The second argument, 30 in this case, gives the number of characters to be stored. The last seven character codes, in this case, are zero. If the second argument is omitted, up to 10 characters are stored. An alternative allows the author to store only the exact number of characters in the response.

```
storea       res,n1 ⇐ jcount
```

This instruction stores the number of characters in the student response and also stores the value of jcount in variable n1. Either integer (n type) or floating-point (v type) variables can be specified as the starting storage location in the –storea– instruction. The action taken is the same in either case.

The –open– instruction puts one character of the student response into each variable. The starting location is the only argument in the tag. Each character is right-justified within the variable. The remainder of each variable is filled with zeros. Only integer variables should be specified as the starting location of an –open– instruction.

MAKING A JUDGING COPY

The –loada– and –close– instructions are used to place characters already stored in the bank of variables in the judging copy. The judging copy is blanked before the new characters are put in, so the characters constitute the judging copy in its entirety. The –loada– and –close– instructions are the complementary instructions to the –storea– and –open– instructions, respectively.

The –loada– instruction places the character starting in the location given in the first argument in the judging copy. The number of characters is given as the second argument and can be a variable reference. An execution error results if a character code of o00 is encountered (neither a blank nor a zero have o00 as a character code). It assumes that the characters are packed 10 per variable, left-justified (as from a –storea– or –pack– instruction).

The –close– instruction takes the rightmost character code of the number of variables specified in the second argument of the tag, beginning with the variable specified as the first argument of the tag. These characters then become the judging copy.

ALTERING THE JUDGING COPY

Two operations can be performed on the judging copy without requiring that it be stored, manipulated, and then replaced. These operations are performed by the –bump–, –put–, –putd–, and –putv– instructions.

The –bump– instruction removes any of the characters in its tag (up to eight characters can be specified in a single –bump– instruction) from the judging copy. Spaces can be included but should be between two other characters for clarity.

All occurrences of any of the characters specified are removed from the judging copy. Thus, the judging instructions following a –bump– instruction should not contain any of the –bump–ed characters as a part of the specified answer.

As an example, the instructions

```
bump      etr cond
answer    5fps
```

allow the student to respond with 5 feet per second and still judge the response as correct.

The –put– instruction substitutes strings. Any occurrence of the string on the left of the first equal sign in the tag is replaced by the string to the right of the same equal sign. As with the –bump– instruction, all occurrences of the string to be replaced are replaced with the execution of the single instruction. As an alternative example to that given for the –bump– instruction, the instructions

```
put       fps = feet per second
answer    5 feet per second
```

allow the student to answer with 5fps and still match the tag of the –answer– instruction.

Problems arise if the character string to be replaced contains one or more equal signs. In this case, the –putd– instruction should be used.

The tag of the –putd– instruction gives the two character strings along with delimiters. As with the –put– instruction, occurrences of the first character string in the student response are replaced by the second string. The major difference is the use of delimiters for the character strings. The first character in the tag is the delimiter used. The next occurrence of the same character indicates the end of the first string and the beginning of the second. The third occurrence gives the end of the second string.

An example of the –putd– instruction is:

```
putd      ,irregardless, regardless,
```

Because the delimiter is defined as the first character in the tag field, this instruction could also be

```
putd      "irregardless"regardless"
```

and have precisely the same effect. The restriction on the delimiters allowed is the obvious one: the delimiter used must not appear in either of the character strings.

The –putv– instruction is similar to the –put– instruction except that the original and replacement character strings are in variables.

The tag field for the –putv– instruction consists of four arguments specifying the following.

Argument	Definition
1	Original character string (left-justified)
2	Length of original string
3	Replacement character string (left-justified)
4	Length of replacement string

An example of the –putv– instruction is:

```
putv      n1,4,n5,n8
```

A maximum of 50 characters can be replaced with each –putv– instruction. If the replacement of characters causes the judging copy to exceed 300 characters, an automatic no judgment is made. The limit of the judging copy is 300 characters.

If successive –putv– instructions are used, they are executed in the order of occurrence and may possibly modify the previous –putv– instructions.

INFORMATION FROM A STUDENT RESPONSE

Sometimes an author needs information about the words in a student response. The –getword–, –getmark–, and –getloc– instructions supply information about a specified word in the student response by specifying the ordinal number of the word in the first argument of the tag. The instructions are regular rather than judging instructions and are executed only in the answer-contingency state which is the regular state following judging and preceding an –endarrow–, an –arrow–, or end-of-unit.

The system-reserved word wcount contains the number of words in the student response. A word is defined as a string of characters separated from other strings by spaces, punctuation, or letter/number boundaries.

The –getword– instruction finds the word in the student response which is specified by number in the first argument of the tag. The word is stored in the location given in the second argument of the tag, and the number of characters in the word is stored in the location given in the third argument of the tag. The optional fourth argument specifies the maximum character length allowed. When this value is not given, the default character length is 10. If the value in the first argument is greater than wcount, the third argument containing the number of characters is zero, and the second argument is unaffected. The third argument always contains the number of characters in the requested word regardless of limits set by the fourth argument.

The –getmark– instruction returns markup information about the word specified by number in the first argument of the tag. The markup information is stored as a number in the location specified in the second argument of the tag.

The –getloc– instruction returns the screen coordinates of the word specified by number in the first argument of the tag. The second and third arguments are the starting x and y coordinates, respectively, and the optional fourth and fifth arguments are the ending x and y coordinates, respectively.

The following instructions store the second word of the student's response in storage locations n10 and n11 and the length of the word in n9. Location n8 contains markup information about that word. The starting screen position is in n12 and n13, and the ending screen position is in n14 and n15.

```
getword     2,n10,n9,20
getmark     2,n8
getloc      2,n12,n13,n14,n15
```

RESPONSE JUDGING

The most important part of response handling is judging whether the response was anticipated by the author and is the one that the author will accept as a correct answer. Any response not specified by the author is judged no (incorrect).

NONNUMERIC RESPONSE JUDGING

Two instructions that are usually used for judging student responses when the response is most apt to be a word, phrase, or short sentence are the –answer– and –wrong– instructions. They operate in the same manner, except that the –answer– instruction judges the response ok if it adequately matches the tag of the instruction, and the –wrong– instruction, if matched, judges the response no. The –wrong– instruction allows the author to give the student a message following the wrong answer, and if desired, to do other processing. The –answer– or –wrong– instruction with a blank tag is matched by a NEXT keypress, a space bar press, or a punctuation mark.

The tag of the –answer– or –wrong– instruction is the answer to be matched. Three options are available for words within the tag.

Words that are simply written in the tag, as in a –write– instruction, must be present in the student response and in the order in which they appear (unless –specs noorder– is used).

Words within a set of parentheses are considered synonyms, and while it is necessary for one of the words to be present in the proper position, any one of the words within parentheses is judged correct. Each set of parentheses should have a list of two or more words, separated by commas.

Optional words are enclosed in angular brackets (<,>). This list can occur anywhere in the tag, and the tag may contain more than one optional word list. Any words in the student response that are included among the optional words are ignored, whatever their position in the student response.

A phrase is indicated in the tag by using an asterisk between the words of the phrase. Phrases must be completely on one line. The system-reserved word phrase is set to –1 if no errors are in the phrase, and it is set to 0 if a phrase is incomplete.

The answer tag cannot contain punctuation except for the commas within a bracket or parentheses set. Punctuation is allowed in the student's response, but it is ignored. The words in the tag are separated by spaces only and are in the order in which the student's response must occur for a match. Thus, the instruction

```
answer      <the,is> (square, rhomboid) blue
```

accepts the square is blue or rhomboid blue as a correct response, but it does not accept blue rhomboid as correct. It is possible to accept keywords in any order as a correct response to an –answer– or –wrong– instruction through the use of the –specs– instruction with the tag noorder. The –specs– instruction is discussed in detail later in this section.

The –answer– and –wrong– instructions also check for extra words, spelling, and numeric tolerance, as well as the word order. These options, like the word-order check, can be cancelled by the author thorugh the –specs– instruction.

The student response is marked for words not contained in the –answer– tag, for correct words out of place, for misspelled words, and for numeric answers that are close to that required but not quite accurate. Extra words are underlined with x's, misspelled words and improper numeric values are indicated with equal signs, and misordered words are indicated with arrows. Because of the complexities of spelling and spelling errors, a misspelled word is not always recognized as misspelled but might instead be indicated as an extra word. The response is not marked unless the response is close to a match or if a –no– instruction (described later in this section) is used. Thus, the instruction

```
answer      Johann Kepler
```

does not accept Leonardo da Vinci as a correct response nor is the response marked to indicate an error. The tolerance for numeric answers or numeric portions of answers is 10 percent for marking the answer (that is, the response is labeled as misspelled if the answer is within 10 percent of being correct). Numeric answers that are more than 10 percent from the correct value are not underlined with x's but are left unmarked. All of the options for marking a student response can be modified or disabled by the author with the –specs– instruction (described later in this section).

The –wrong– instruction performs the same sort of marking of the answer, which can be a disadvantage, since it leads the student to answer incorrectly by following the markings. However, messages and actions taken following a matched –wrong– answer can be helpful to the student, and these are only executed when the tag of the –wrong– instruction has been adequately matched.

The –answerc– and –wrongc– instructions are conditional forms of the –answer– and –wrong– instructions, respectively. The variable in the first argument of the tag indicates which response is anticipated. The responses may be anything which is legal in an –answer– or –wrong– instruction. A response for a condition may extend onto the next line since end-of-line is not a delimiter for these instructions.

At times, the author wishes to allow a number of ignorable words or wants to specify a number of synonyms for words. This can make the –answer– and –wrong– instructions unwieldy because of the number of words that must be specified in the tag. One alternative is the use of the –list– instruction.

The –list– instruction sets up a list of synonyms that can be used interchangeably. The synonymous words are given in the tag of the instruction, separated by commas. The first argument in the tag, however, is not one of the synonyms but the name to be associated with the list. This is necessary because more than one –list– can be used, even within a single –answer– or –wrong– instruction. Each list name must be unique with respect to other list names in the same lesson, and it must be no more than seven character codes long.

The synonyms in the list can be used as either synonymous important words or as a list of ignorable words. The list is referenced in the tag of the –answer– instruction by the name (first argument) of the list. The name of the list must be enclosed in either parentheses or angular brackets, whichever is appropriate to the usage, contained within another matching set of parentheses or brackets. These outer brackets allow further synonyms to be defined for use during a single –answer– or –wrong– instruction. As an example of the use of the –list– instruction, consider the following instructions.

```
list        extra,a,the
list        saurian, brontosaurus, stego-
            saurus, diplidocus
at          510
write       Name a dinosaur
arrow       1010
answer      <<extra>> ((saurian), tyrannosaurus,
            trachodon)
```

If the lesson concerns dinosaurs, a –list– instruction specifying synonymous names is much more readable and easier to write than spelling out all the alternatives in each relevent judging instruction.

If the vocabulary to be used in judging student responses is quite large, or if something approaching a dialog is to be allowed between the student and the lesson, the –answer– and –wrong– instructions might not be appropriate. Instructions that make the judging of complex responses faster and easier are the –concept–, –miscon–, –vocabs–, –vocab–, and –endings– instructions. These instructions allow student responses to match any of a number of anticipated responses.

The tag of the –concept– instruction specifies the concept to be matched, judging the student response ok if the tag is matched. The –vocabs– or –vocab– instructions must be used with the –concept– instruction. All of the words in a –concept– instruction must be in a preceding –vocabs– instruction to prevent condense errors. Using a number of –concept– instructions is fast and efficient because the –concept– instruction converts the student response to a sequence of numbers. If more than one line is used in the tag of the –concept– instruction, each line is taken to represent an equivalent concept. This allows the author to enable the student to use grammatically different but semantically identical responses and to allow those responses to match the same judging instruction. It also requires the author to keep each concept less than one line long. Every word in a –concept– instruction must be part of an active vocabulary established by a previous –vocabs– or –vocab– instruction. A –concept– instruction with a blank tag is matched when the student's response consists only of ignorable words.

The analog of the –wrong– instruction for judging concepts is the –miscon– instruction. It judges the student response no if the tag is matched. All the features of –concept– can be used with –miscon–.

The –vocabs– instruction specifies the vocabulary to be used by a –concept– instruction. Just as a –define– set has a name so that the set can be referenced by name at other points in the program, the first line of the tag of the –vocabs– instruction gives a name to be associated with the vocabulary that the instruction specifies. Succeeding

lines may be comprised of ignorable words, keywords, synonymous words, and phrases. Ignorable words are separated by commas and enclosed in angular brackets. Keywords are separated by commas. Synonymous words are separated by commas and enclosed in parentheses. Synonymous words with suffixes may be used (refer to –endings– instruction in this section). Phrases are designated with an asterisk between the words of the phrase. A phrase may not be split between lines and cannot use plurals and –endings–. If a word begins with a number and contains letters also, treat it as a phrase (for example, 3rd does not work, but 3*rd does work). The –vocabs– instruction also has spelling and capitalization checks.

The –vocab– instruction is almost identical to the –vocabs– instruction except that it does not allow phrases, has no spelling checks, and has no capitalization checks; therefore, it uses less space than a –vocabs– instruction.

The –endings– instruction adds suffixes to words defined in a vocabulary (–vocab– or –vocabs–) and must precede the –vocab– or –vocabs– instruction. The tag of the –endings– instruction consists of two arguments: the first argument is a number (0 through 9) identifying the suffixes list, and the second argument is the actual list of up to eight suffixes. For example, the instructions

```
endings     0,ed,ing,s
endings     1,ed,ing
```

define two suffix lists as shown below.

```
vocab       work/0
            fir//1
            hire/s/d
```

The first line of the –vocab– instruction defines as synonymous the words work, worked, working, and works. The second line defines as synonymous the words fired and firing but not fir. The third line defines as synonymous the words hire, hires, and hired.

The –concept– instruction uses the vocabulary specified by the last vocab type instruction encountered prior to the –concept– instruction. If more than one –vocab– type instruction has been encountered and the author wishes to use one of the earlier ones in judging a response, the relevant vocabulary is called into activity by a –vocab– type instruction with only the name of the desired vocabulary in the tag of the instruction. Because of this ability to recall vocabularies to activity, the name associated with each –vocab– vocabulary must be unique within the lesson.

The –concept– instruction functions by reducing the synonyms given in the –vocab– instruction to a positional reference within the list, associating the important words in the tag of the instruction with the same numbers, and checking the student response to see if a similar reduction yields the same result. Thus, if the important words specified in the tag of the –concept– instruction are in the order taken from the first, sixth, and second sets of synonyms in the –vocab– instruction, the –concept– instruction reduces the answer to be matched to the numbers 162. The student response is subjected to a similar reduction and judged ok if the reduction of the student response also yields 162. Words specified in the –vocab– instruction as ignorable do not affect judging of the student response, regardless of the position of occurrence or number of appearances in the student response.

The –ok– instruction judges the student response ok and ends judging. It can be used for acceptance of arbitrary criteria or any other purpose in which the exact form of the response is immaterial, such as storing a name by which to address the student. In such an instance, the –storea– instruction is probably the best instruction to use, but the –storea– instruction does not end judging so that further processing can be done. The –ok– instruction can be used for this purpose.

The –no– instruction functions in the same manner as the –ok– instruction, except that the student response is judged no rather than ok. As with the –ok– instruction, the tag of the –no– instruction is blank.

The –ignore– instruction, which also possesses a blank tag, has a somewhat different effect. The student response, rather than being judged ok or no, is ignored. The response is erased, processing stops, and the system waits for a new student response. Regular instructions following the –ignore– instruction are not executed, nor are the regular instructions, if any, following a –specs– instruction.

At times it is necessary to judge the student response correct only if it exactly matches the specification given by the author. The –answer– instruction is too general for this purpose. Three instructions are supplied for requiring exact matches: –exact–, –exactc–, and –exactv–.

The –exact– instruction judges the student response ok only if it precisely matches the tag of the instruction. This includes punctuation, spaces between letters, and all other characteristics of the tag. The –exactc– instruction is equivalent, but it is a conditional form. The first argument in the tag is the variable or expression whose value determines which argument is to be used as the tag. The remaining arguments are the possible responses the student can be required to match. Since the arguments are separated by commas, the responses may not contain commas. This can be altered by a –change symbol– instruction (described later in this section), or else an –exactv– instruction can be used.

The –exactv– instruction has two arguments in its tag. The first argument specifies a variable name, and the second argument specifies the number of characters to be matched. The instruction compares the student response against the specified number of characters in the variable bank, starting with the specified variable. The response is judged ok if an exact match is achieved. This allows a form of conditional judging with punctuation, since the characters to be matched can change either by change of characters or by change of variable reference. If the character count in the second argument is omitted, the comparison ends after 10 characters or after a zero string of six bits. If the second argument is zero, the student response is judged correct if no keys are entered.

NUMERIC JUDGING

When a purely numeric response is required, which may be an algebraic expression, several instructions are available for processing the response.

The –answer– instruction is limited, since only simple numeric responses, such as 200+32, are permitted. The –wrong– instruction has the same limitation. The –ansv– and –wrongv– instructions are specifically designed for

judging numeric and algebraic responses. The tags of these instructions have either one or two arguments. The one-argument form requires that the value of the student response be precisely equal to the value specified by the author. The optional second argument specifies a range within which the response is judged ok. The first argument can be an expression, if desired. The second argument can either specify an actual value range or a percentage range.

For example, the instruction

```
ansv        23+y^2,5%
```

judges the student response correct if it is within 5 percent of the value given by the expression 23+y2. An example of the use of a specific numeric range is the instruction

```
wrongv      237,4
```

which judges the response no if it is between 233 and 241, inclusive.

The –store– instruction also judges algebraic responses. The manner of evaluation is somewhat different from that of the –ansv– and –wrongv– instructions. The response is evaluated, and the value is stored in the variable which is the tag of the instruction. No value is specified as correct. The response is judged no if the response cannot be evaluated. If the response can be evaluated, the response is not judged ok, and the judging is not stopped. Thus, an instruction such as the –ok– instruction is necessary to end judging with an ok judgment.

The system-reserved word formok is set by the –store–, –ansv–, and –wrongv– instructions. The value of formok is –1 if the response can be evaluated. If the response cannot be evaluated, the value of formok is set to any of various values, depending on the type of error. A list of these values, together with the type of error they represent, is given in appendix B.

The –storen– instruction searches the student response for a numeric element, and if found, evaluates the element and stores the value in the variable specified in the tag. Judging, in this case, is not ended. If no numeric element is found, the response is judged no, and the variable is given the value 0. Only simple numerics are permitted in the student response, such as –2/3 or 4.75. Variable names cannot be used in the response. The numeric element can be embedded in text but must, in this case, be set off by spaces or punctuation. Only the first numeric element is searched for, but that element is removed from the judging copy so that responses with more than one element can be broken down by multiple –storen– instructions.

The –ansu–, –wrongu–, and –storeu– instructions are useful when judging student responses that involve both numbers and scientific units, such as 3 .7 kg-m/sec2.

These instructions are similar to the –ansv–, –wrongv–, and –store– instructions except that the dimensionality (units) of the student response is handled, not just the numerical element.

The dimensions used in the –ansu– and –wrongu– tags must be previously defined in a –define student– instruction. For example:

```
define      student
            units,kg,m,sec      $$primary units:
*                               $$the keyword units
*                               $$is required
            cm=m/100,           $$equivalent units
            min=60xsec
*
ansu        3.7 m/sec
*
wrongu      10.9 kg-m/sec^2
```

The –ansu– instruction judges numeric student responses (with scientific units) ok if the answer is equal to the specified answer within a specified tolerance. The –wrongu– instruction judges numeric student responses (with scientific units) no if the answer is equal to the specified answer within a specified tolerance.

When a tolerance is specified as a deviation, it is taken to be a primary unit from the student define set, no matter what units were specified in the –ansu– instruction. To make sure the deviation is what is intended, the deviation units must be specified explicitly. When the tolerance is a percentage, the units used when none are specified are the units specified in the first argument of –ansu–.

To store the numeric and dimensional elements of the –ansu– or –wrongu– response, the –storeu– instruction is used. The –storeu– instruction must precede the –ansu– or –wrongu– instruction.

The numeric element of the student response is stored in the variable specified in the first argument of the –storeu– instruction tag, and the dimensional element of the response is stored in the 10 consecutive variables specified in the second argument of the instruction tag.

The response is judged no, and judging ends if the student response cannot be evaluated. Judging does not end, however, if the student response can be evaluated.

Again, the –storeu– tags must be previously defined in a –define student– instruction. The –ansu–, –ansv–, –wrongu–, and –wrongv– instructions must follow the –storeu–.

TOUCH PANEL JUDGING

The –ntouch– and –touch– instructions are used to judge student responses when the optional touch panel is used. Since not all terminals have the touch panel capability, these instructions should usually be accompanied by a keyboard-input-response-judging instruction. The instructions can be linked with the –or– instruction discussed later, if necessary.

The touch panel is composed of 256 squares. Each square is two coarse-grid rows in height and four coarse-grid columns in width. Thus, the screen has 16 rows and 16 columns of touch squares, each 32 dots by 32 dots.

The fine-grid coordinates of the center of the last square touched are returned in system-reserved words ztouchx and ztouchy. This information is also returned in the system-reserved word key in the bit format 1xxxxyyyy, where xxxx is the horizontal location and yyyy is the vertical location of the touch square, both expressed in binary. Since the key code for the touch panel is greater than 255 (o377), it is simple to test if touch panel input has occurred by simply testing the magnitude of key. The keyboard input codes are all less than 255 (o377).

The touch panel, if enabled (refer to section 9), terminates a -pause- instruction and can also function as a NEXT key if a touch instruction is not included in the judging instructions of the current -arrow- instruction or in units that do not contain an -arrow- instruction.

The -ntouch- instruction judges a touch ok if it lies within one of the areas specified in the tag. The arguments in the tag specify rectangles either in coarse-grid coordinates or in fine-grid coordinates. Touch squares which are partially or completely covered by these rectangles are the areas used to match touches on the touch panel.

Areas specified in coarse-grid coordinates can have one or three arguments. The first argument is the coarse-grid location of the lower left corner of the area. The optional second and third arguments are the number of characters wide and the number of lines high, respectively. Areas specified in fine-grid coordinates can have two or four arguments. The first two arguments are the fine-grid location of the lower left corner of the area. The optional third and fourth arguments are the number of dots wide and the number of dots high, respectively. Default values for widths and heights are 1 and 1. Commas separate the arguments.

When the -ntouch- instruction specifies more than one area in the tag, semicolons separate the areas. The inequality

(3 X number of coarse-grid areas) + (4 X number of fine-grid areas) < 62

gives the maximum number of areas which can be specified in an -ntouch- instruction. Omitting the width and height of an area does not increase the maximum number of areas possible in an -ntouch- instruction.

Touch panel input in a location other than that specified by an -ntouch- instruction is judged no.

The -ntouchw- instruction judges anticipated no responses. It functions exactly like the -ntouch- instruction except that a matched response is judged no instead of ok. If the response is an anticipated no, system-reserved word judged is set to 0. If a square defined by -ntouch- overlaps a square defined by -ntouchw-, the square defined last is ignored.

The -touch- and -touchw- instructions are early forms of the -ntouch- and -ntouchw- instructions and are being phased out.

The -touch- instruction specifies the location to be touched as the first argument of the tag. This location is a single coarse-grid screen coordinate. There can be either one or two more arguments. If there is one more argument, it specifies the tolerance in terms of touch squares and in all directions from the touch square containing the coordinate reference.

The three-argument form is used when the area that is to give a judgment an ok is not square. The first argument again specifies a coarse-grid coordinate position. This position is the lower-left corner of the area to be delimited. The second argument specifies the width of the area in touch squares. The third argument specifies the height of the area, also in touch squares. Thus, the instruction

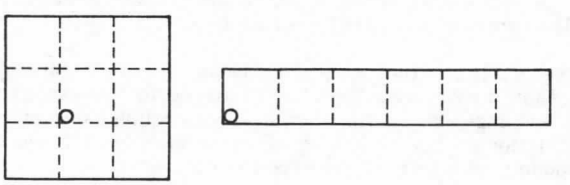
```
touch       2021,1
```

judges ok any touch panel response occurring in the square delimited by coarse-grid coordinates 1717, 1728, 2228, and 2217. The instruction

```
touch       1013,6,1
```

judges a touch panel response ok if it lies within the rectangle bounded by coarse-grid coordinates 913, 1013, 1036, and 936. Since the first argument specifies the lower-left corner of the sensitive area, the location requires an even line number and a column number obtained by the formula: (4 X integer) + 1. Any coarse-grid location can be used in the tag, but the system activates the square corresponding to a proper touch location as defined previously.

An important difference between the two- and three-argument forms of the -touch- instruction is that the two-argument form specifies the center touch square of the desired touch area, whereas the three-argument form specifies the lower-left corner of the desired touch area. These are the areas specified by the above -touch- instructions.



The -touch- instruction may also have a list in the tag of up to 20 areas as correct responses. These areas may be either two- or three-argument forms and are separated by semicolons. Judging is ended with an anticipated ok if any one of the specified areas is touched.

Touch panel input in a location other than that specified by a -touch- instruction is judged no.

The -touchw- instruction judges anticipated no responses. It functions exactly like the -touch- instruction except that a matched response is judged no instead of ok. If the response is an anticipated no, system-reserved word judged is set to 0.

OTHER JUDGING INSTRUCTIONS

When one of several possible actions expressed in one or more conditional instructions is to be executed, depending on the student response, the –match– instruction is usually the best way to determine which action to take.

The first argument of the –match– instruction is a variable name that gives the variable that will contain the number of the answer matched. Succeeding arguments give the possible responses, separated by commas. The value of the variable specified is -1 if no match is found, 0 if the student response matches the first item in the list of responses, 1 if it matches the second, and so forth. Continued lines (that is, lines immediately following, with the command field blank) are considered part of the same –match– instruction. Judging is ended in any case, with a judgment of ok if a match is found and no if otherwise.

The –or– instruction is used to define consecutive judging instructions as equivalent. The regular instructions, if any, following the last such judging instruction are executed if any one of the instructions in that set of equivalent instructions has been matched. Similarly, the system-reserved word ansent, which counts the number of judging instructions that have been scanned in searching for a match, has the same value regardless of which of the instructions is matched. The tag of the –or– instruction is blank.

As an example of the use of the –or– instruction, the instructions

```
answer      1
or
touch       515
```

allow the student to specify either a number or a location on the touch panel as the response. Either is judged as equivalent. The value of ansent is the same, and any regular instructions immediately following the –touch– instruction are executed if either the –touch– instruction or the –answer– instruction is matched.

However, the actual judging ok or no is done separately for each instruction. Thus, it is possible to mix –answer– and –wrong– instructions in a single equivalent set of instructions. Actual judging ok or no is done as it would be done if the instructions were not connected.

It is sometimes desirable to furnish the student with the correct answer; this can be done with the –ans– instruction.

The effect of the –ans– instruction is to enable the ANS key on the student's keyboard. If the student then presses the ANS key, judging is terminated, the response is judged ok, and any regular instructions immediately following the –ans– instruction are executed. These would ordinarily be the display instructions for informing the student of the correct answer.

The only restriction on the –ans– instruction is that it must be the first judging instruction to occur following the relevant –arrow– instruction. This means that the –ans– instruction must precede the –specs– instruction if both are used.

If a unit does not contain an –ans– instruction, the ANS key has no effect. Similarly, if there is an –ans– instruction following an –arrow– instruction and the student does not press the ANS key, the instruction has no effect. The value of ansent is not incremented by the –ans– instruction.

The –compare– instruction allows an author to compare the spelling of two words. These words may be either alphabetic or numeric. The –compare– instruction functions according to the system judging symbols.

The tag consists of three arguments. The first two arguments are the storage locations of the two words to be compared. The third argument is the location of the result of the comparison. This result is:

-1	If the words are different
0	If the words are the same
+n	If the words are misspellings of each other, where a smaller n indicates less of a misspelling

If the result of the comparison is +n, the system-reserved words capital and spell are set. (Refer to appendix B.)

-SPECS- INSTRUCTION

The –specs– instruction has already been mentioned earlier in this section because of its effect on instruction execution during response judging. This instruction modifies the normal judging processes for the current –arrow–. It also acts as a marker which is returned to after each ok or no judgment.

The tag of the instruction consists of specifications of standard author language options that are to be turned off during response judging or nonstandard options that are to be turned on during judging. Each specification is represented by an appropriate word. When more than one specification is to be given, the specifications are separated by commas. If the –specs– instruction is to be used only as a marker, a blank tag may be used.

A list of the options, with a short description of the effect of each, is given in table 11-2. Options affect only the student response.

The –markup– instruction marks the student's answer with the markup saved by the –specs holdmark– instruction. If –specs holdmark– is not specified, the system automatically marks the student's answer during judging, unless the markups are canceled by the –specs nomark– instruction. Student answer markups consist of the following.

Mark	Meaning
⇐	Word out of correct order, move left
===	Misspellings
xxx	Unknown words
***	Broken or incomplete phrases
↑	Capitalization errors
Δ	Missing items

TABLE 11-2. –specs– OPTIONS

Option	Effect
allwords	Numbers are interpreted as words, not numeric quantities.
alphxnum	Word-number boundary is treated as punctuation.
bumpshift	All shift codes in the response are ignored.
exorder	The order of ignorable words is important.
holdmark	Withholds an answer markup until –markup– instruction is executed.
nodiff	The numeric approximation judger is turned off.
nomark	The normal marking-up of an answer is not done.
nookno	The ok or no is not displayed following the response when judging is complete.
noops	No arithmetic operations are permitted.
noorder	The order of occurrence of key words is unimportant.
nospell	Spelling judger is turned off (no misspellings recognized).
novars	No variable references are permitted.
okassign	The student can assign values to variables which are defined in set student.
okcap	Capitalization of a word is ignored in a response if, and only if, the word is not capitalized in its –vocabs– entry.
okextra	Extra words in the response are permitted.
okspell	Recognizable misspellings are permitted as a match.
toler	Numeric answers within 1 percent of value given are judged ok.

REGULAR INSTRUCTIONS AFFECTING RESPONSE HANDLING

Some regular instructions also affect either the judging copy of the student response or the judging process. Since these are not judging instructions, judging must have been terminated or not yet begun for these instructions to be executed.

The –judge– instruction can be used to alter the judging process or the final judgment. The –edit– and –copy– instructions can be used to increase the ease with which the student can modify a response. The –time– instruction sets a time limit within which the system waits for the student response. The –change– instruction changes the judging values of characters.

-JUDGE- INSTRUCTION

The –judge– instruction can be used to restart judging with or without changes made in the judging copy, to cancel a previous judgment, or to specify the judgment to be made. The instruction has both a conditional and an unconditional form except that an argument of q is not permitted. The –judge– instruction is only executed in the regular state.

There are 11 possible specifications.

judge	ok	$$set judgment to ok, continue processing regular instructions
judge	no	$$set judgment to no, continue processing regular instructions, answer is considered unanticipated
judge	wrong	$$set judgment to no, continue processing regular instructions, answer is considered anticipated
judge	exit	$$rescind previous judgment, wait for further keys of response
judge	continue	$$restart judging, using judging copy as modified by –bump–, –put–, and so forth
judge	rejudge	$$replace modified judging copy with original response and restart judging
judge	x	$$no action taken
judge	ignore	$$student response erased, system waits for another response
judge	quit	$$stops the processing of regular instructions without changing judgment
judge	okquit	$$set judgment to ok, stops processing of regular instructions
judge	noquit	$$sets judgment to no, stops processing of regular instructions

The judge rejudge option uses the other copy of the student response, which is not modified by any author language instructions. This is why two copies are made, so the second copy can be used for reference or rejudging.

The continue and rejudge options start judging, so the system is switched into the judging state following execution of either of these options. The –ok– and –no– instructions are the equivalent instructions used to switch from the judging state to the regular state.

An example of the conditional form is:

```
judge        exp,no,x,ok,ignore,no
```

-EDIT- AND -COPY- INSTRUCTIONS

The student can be allowed to place words into his input buffer by means of the –edit– and –copy– options. They function in a similar but not identical manner.

The –edit– instruction sets up a buffer starting at the variable location specified in the tag of the instruction, and it continues until the length limit of the student response is reached. Unless modified with a –long– instruction, this is 150 characters or 15 variables.

A default –edit– buffer is always active unless a –long– instruction with a tag greater than 150 is used. If a –long– instruction with a tag greater than 150 is used and the –edit– feature is desired, the –edit– instruction must be used.

The EDIT key causes operation of the –edit– option. When the student has typed in a response, pressing the EDIT key removes the response from the display (and from the input buffer). Each press of the EDIT key after this brings in one word of the student response, both on the screen and into the input buffer, until the entire response is again on the screen. A press of the EDIT key following full display of the response causes the entire cycle to repeat. Pressing SHIFT EDIT copies the remaining portion, at any point in the cycle, into the input buffer and onto the screen.

The square key (□), when pressed, brings in one character of the student response, both on the screen and into the input buffer.

A word, in the case of the –edit– and –copy– instructions, is not a computer word, which would consist of 10 characters. Rather, it is a string of alphanumeric characters bounded by punctuation. In this case, punctuation also includes spaces, as well as periods, commas, parentheses, and so on.

Modifications that the student makes to the displayed response are reflected in the –edit– option. That is, if a word is added or deleted partway through the original response, use of the EDIT key includes the new change in the –edit– copy.

The starting location, which is the only argument of the tag, must be a variable name (or be reducible to a variable name by the system). The variable must be in the student variables and not in common or storage variables. In addition, the author must be sure that a sufficient number of variables are available to contain an entire response to ensure proper operation of the edit option. Thus, an instruction such as

```
edit         n140
```

should be used only if the permitted length of the student response (set by a –long– instruction) is no more than 100 characters.

The –copy– option functions similarly to the –edit– option, but it uses a character string given by the author rather than the student response. The –copy– instruction has a two-argument tag. The first argument gives the starting location of the character string, and the second gives the total length of the character string in number of characters. As with the –edit– instruction, the starting location must be in the student bank of 150 variables.

The COPY key activates the copy option. When the key is first pressed, the first word from the copy buffer is placed in the input buffer and displayed on the screen. As with the –edit– instruction, a word, in this context, is not a computer word but an alphanumeric string bounded by punctuation. Repeated presses of the COPY key cause the entire character string in the copy buffer to be copied into the input buffer. Again, similar to the –edit– option, pressing the SHIFT COPY keys copies the remainder of the string into the input buffer and displays it on the screen. The square key operates the same as the –edit– option.

However, unlike the –edit– option, the string can only be used by the student once. The string is not destroyed, so the author can specify the same string as the object of the –copy– option for a different –arrow– instruction. However, the student can only access the copy string once in each –arrow–.

For both the –copy– and the –edit– instructions, a blank tag clears the associated buffer, if placed after an –arrow–, and disables the option. Thus, if the author wishes to prevent the student from using the edit option, even though the student response length limit is less than 151 characters, an –edit– instruction with a blank tag should be used. The edit option may also be cancelled with an –inhibit edit– instruction.

The copy option is turned off when the system encounters an –arrow– instruction, so the –copy– instruction should follow, not precede, the appropriate –arrow– instruction.

-TIME- INSTRUCTION

The –time– instruction is used to limit the time the student has to give a response. The tag of the instruction is the time allowed in seconds. If the time is exceeded, the system-reserved word key is assigned the value timeup and processing is done as if the NEXT key had been pressed. The system-reserved word key can be tested to find if the student was timed out or entered a response. An example is:

```
time         10
ok
writec       key=timeup, Time limit exceeded,,
```

Other conditional instructions, such as the –judge– instruction, can also be used to test system-reserved word key, and if desired, start or modify judging on that basis.

-CHANGE- INSTRUCTION

The –change symbol– instruction is used in the IEU to change the values of characters when they are used in the judging state. The change is lesson-wide.

The tag of the instruction consists of the word symbol, followed by a character, which is followed by the word to, and ending in a new character to be substituted or the word letter. An example is:

```
change      symbol [ to (
            symbol ] to )
            symbol ( to letter
            symbol ) to letter
```

The order is important in a –change symbol– instruction. In the example above, [and] are read by the system as (and), respectively, in all judging instructions in the lesson. The symbols (and) are then treated as letters. This permits students to have symbols (and) in their answers. For example,

```
answer  [aluminum sulfate,A1₂(SO₄)₃·17H₂O]
```

accepts aluminum sulfate as a student answer, and it also accepts $A1_2(SO_4)_3 \cdot 17H_2O$.

Several of the PLATO author language instructions have a conditional form. In this form, the action taken depends on some condition. The condition is the value of an expression. The general form is

command expr,a1,a2,...

where **expr** is the expression whose value is the determinant of the action, and **a1,a2,...** are the different options to be taken. The first option is selected if the value of the expression is negative, the second if the value is 0, the third if the value is 1, and so forth through the list of options, in increments of one. If the expression is larger than the number associated with the last option specified, the last option is selected. The option is usually, but not always, a name to be associated with the instruction. Thus, the instruction

```
jump    funge/2-3,first,blat,zilch,threlp
```

jumps to unit first if the value of the expression funge /2-3 is negative, to unit blat if the value is 0, to unit zilch if the value is 1, and to unit threlp if the value is greater than 1. The value of the expression is rounded, if necessary. The effect is as if the option selected were the entire tag of the instruction.

Two special characters can be used to take no action or to clear markers. Usually, if an x is encountered as one of the actions, no action is taken, and the instruction has no effect. When the instruction is one that sets markers (such as the –base– and –next– instructions), the character q clears the marker, and the effect is that of having the instruction with a blank tag. As an example, the instruction

```
base    2x-2,un1,x,q,grilk
```

has no effect if the value of the expression 2x-2 is 0, and it specifies the current unit as the base unit if the value of the expression is 1.

A q as the action to be performed in a conditional –goto– or –join– instruction does not execute any other unit, but execution of the current unit is halted and not resumed. If the current unit is a main unit, the student must press the NEXT key to continue, and the next unit is then initialized. If the current unit is an auxiliary unit, execution goes back to the main unit immediately. Similarly, an x does not cause a unit to be executed, but the current unit is continued so that the instructions following the –join– or –goto– instructions are executed.

Since the expression can be any legal expression, a logical expression can be used. In this case, only two possible actions should be specified, since there are only two values that the expression can take; -1 (true) and 0 (false). It is because the first action specified must correspond to a negative value of the expression that the values of a logical expression are not 0 and 1 as in the semistandard notation of Boolean algebra.

There are some special cases of the conditional form.

The –writec– instruction is the conditional form of the –write– instruction. Because the actions to be taken are separated by commas in normal conditional form, the messages to be written by the –writec– instruction ordinarily cannot contain commas or other punctuation. However, there is a universal terminator available with the –writec– instruction which, when used, allows punctuation in the messages. This terminator is obtained by pressing the ACCESS key and then a comma; it looks like ↕.

If it is desired to write nothing in some cases, the appropriate position should be indicated by successive delimiters, either commas or the universal delimiter. This is necessary since an x, used to indicate no action to be taken in other forms, would be treated as text to be displayed. If the last condition is not to display a message, successive delimiters should be used rather than simply omitting use of the position. For example:

```
writec  y < 3, Too small,,
```

If the final comma were not present, the message

```
Too small
```

would be displayed regardless of the value of y.

The –calcc– instruction is the conditional –calc– instruction and performs the appropriate –calc– type action. The possible actions consist of different assignments of value, with the variable to which the value is assigned having the potential of being different in each action. For example:

```
calcc   expr, v2⇐ 31,gun⇐ sin(blet)+2,v7⇐ π /4
```

When one of several possible values is to be assigned to a single variable, it is more efficient to use the –calcs– instruction. The –calcs– instruction assigns one of several possible values to a single, specified variable. For example:

```
calcs   expr,fout⇐ en(grun),36,2y+7
```

Neither the –calcc– nor the –calcs– instruction can use an x to indicate no operation. The –calcc– can use a 0, and both the –calcc– and –calcs– can use successive commas, as in:

```
calcs   expr,glitch⇐ 1,5,,23
```

The instructions that can have a conditional form are listed in table 12-1.

TABLE 12-1. INSTRUCTIONS WITH A CONDITIONAL FORM

answerc†	datalop	iferror	match†
back	do	imain	mode
backop	eraseu	join	next
back1	exactc†	judge	nextop
back1op	finish	jump	next1
base	from	jumpout	next1op
branch	goto	keytype†	nextnow
calcc†	help	lab	packc†
calcs†	helpop	labop	stop
data	help1	lab1	writec†
dataop	help1op	lab1op	wrongc†
data1	iarrow	lesson	

†These instructions are only used in the conditional form.

The author or instructor can collect data on the execution of a lesson or lessons in his course by using a datafile, a file that is used exclusively for the collection of student data. All data stored is course-, lesson-, and student-specific. That is, the author or instructor can collect data only for students in his course, only for lessons that specify data collection, and only for students that have the data collection option turned on through their course records. Further, the lesson and student are indicated on the data collected as overhead information.

Various types of information can be collected. This includes summaries of areas of the lesson, requests by the student via the TERM and help keys (found or not found), execution error information, and student answers, whether judged ok, no, or u-no (unrecognized no judgment). Unrecognized words associated with a –concept– instruction can also be stored in the datafile.

Any or all of these options are specified in the records for each student in the course records. It is not recommended that all data be collected for all students, as this causes the datafile to fill quickly. The datafile can be cleared when full, but this loses any earlier information, and data can be lost between the time the datafile fills and the time the author empties the datafile. In particular, it is not recommended that all of the student responses be stored, as this can use a great deal of space in a short time. While it may be useful to store all responses when the lesson is still being intensively developed, it later is probably sufficient to collect the u-no judged responses. These are the responses that are not anticipated by the author, so they are judged no by default.

SPECIFYING DATA COLLECTION

As previously mentioned, the data options in the course records must be turned on for each student who is to collect data during lesson execution. Additionally, the course must have an associated datafile. This is done by going to the course records, specifying the proper student, and once the student's records have been obtained, choosing the proper option to find the data option specifications. These can then be modified by the author or instructor. Pressing the letter or number associated with any of the options switches the state of that option. That is, if the option is off, it is switched on, and vice versa. The data collection option number 1 must be on for data collection to occur.

Every lesson for which data is to be collected must contain a –dataon– instruction. The options of the –dataon– instruction can override student data options for the remainder of the lesson, but they cannot turn on course-wide data options that are turned off. The available options for data collection, with the tag associated with each, are given in table 13-1. More than one option can be specified in a single –dataon– instruction, separating the arguments representing the different options with commas. Thus, the instruction

```
dataon   no, unrec no
```

places all student responses that were judged no into the datafile, whether the answer was anticipated by the author or not.

TABLE 13-1. LEGAL TAGS FOR –dataon– AND –dataoff–

Tag	Stores
ok	Responses judged ok
no	Responses judged no that match an author-specified wrong answer
unrec no	Responses judged no that were unanticipated by the author
errors	Execution error
vocab	Unrecognized words in response to –concept–
area	Area summary information
output	–output– and –outputl– instructions
help	Requests for help keys (HELP, LAB, and so on) that were satisfied
help no	Requests for help keys that were not satisfied
term	TERM requests found
term no	TERM requests not found
signin	Time and data student entered lesson

The –dataoff– instruction stops data collection for that lesson. It can be restarted later in the lesson by another –dataon– instruction. If the tag of –dataoff– is blank, no more data is collected. If the tag has arguments (the same as the arguments permissible for the –dataon– instruction), only those options specified in the tag are no longer collected, provided that the options were originally specified in a –dataon– instruction.

SPECIFYING DATA TO BE COLLECTED

The author must, in some manner, indicate the data that is to be collected. For most types of data, such as storing the student responses or the number of TERM requests that were not satisfied, this is done through the course records or with the –dataon– instruction, as described previously. In fact, it is necessary to use these areas in

all cases. However, some types of data, specifically area summary information and the –output– and –outputl– instructions, require more work on the part of the author.

Area summary information refers to specific portions of the lesson. The areas are indicated by the author with the –area– instruction. If the tag is blank, the data for the area just completed is put in the datafile, and no more area data is collected until an –area– instruction with a nonblank tag is encountered. The tag of the instruction gives the name to be associated with the area of the lesson following. An area is delimited by the –area– instruction as encountered during lesson execution. Hence, if two students take different paths through an area of the lesson (for example, one executes more or different help sequences), the area summary data appears to refer to different areas, as the number of –arrow– instructions encountered can quite easily be different. Similarly, if a student is branched to a new area but at a later unit than that containing the –area– instruction, the new area is not recognized, and the summary data for the area is included as part of the previous area.

If an –area– instruction with a nonblank tag is encountered while data is still being collected for an area summary, one of two things happens. If the names of the two areas are different, the summary data for the previous area is written into the datafile, and collection of data in the new area begins. If the names of the two areas are the same, the second –area– instruction is ignored. No data is written to the datafile until an –area– instruction with either a blank tag or a different tag is encountered during execution.

The name of an area, that is, the tag of the –area– instruction, must be no longer than 10 characters and cannot start with a numeral. The tag can be a variable, in which case the alphanumeric contents of the variable is taken as the name. This also means that a (limited) expression can be used as the tag of the –area– instruction, as in

```
        area    n3$union$o73
```

or

```
        area    name $union$ var9
```

where var9 is a variable name, and name is an alphanumeric string.

The tag for the –area– instruction can also be one of two keywords available, incomplete and cancelled. The keyword incomplete ends data collection for current area and marks the area as incomplete, and cancelled ends data collection for the current area but does not enter any data in the datafile.

The –setdat– instruction allows an author to alter the value of system-reserved words pertaining to areas. Sometimes the numbers returned in area summaries are not meaningful because of the way a particular lesson operates. The –setdat– instruction makes it possible to collect more meaningful data. The system-reserved words which may be altered by –setdat– are:

aarea	auno
atime	ahelp
aarrows	ahelpn
aok	aterm
aokist	atermn
asno	

These reserved words may contain only integers and may not have values greater than 511. An exception is atime. It cannot be set to a value greater than the time signed on for the current session. It is accurate only to 1/10 second.

When the data the author desires is not contained in any of the standard data options, the –output– and –outputl– instructions are used. This is the case particularly where the author wishes to store the contents of one of the variables or wishes to insert a comment in the data, or both.

The –output– instruction functions in much the same manner as a –write– instruction except the text is placed in the datafile. That is, whatever character string is contained in the tag of the instruction is placed into the datafile.

The contents of variables can be placed in the datafile by using the embedding feature. This is much more restricted in the case of the –output– instruction than in the –write– or –writec– instructions but functions in an analogous manner. The embedded portion is set off by the same symbols (≺, ≻), but the contents between these symbols is different.

Only two arguments are used when embedding in an –output– instruction. The first argument specifies the format in which the variable is to be stored in the datafile and can be one of four possible arguments: a (alphanumeric), n (integer), o (octal), or v (floating-point). The second argument names the variable to be written in the datafile. The name can be either the primitive name (v23) or an assigned name (luft).

Information written into the datafile also has overhead information associated with it. This information consists of the student, lesson, and area names, plus the time elapsed since the student entered the lesson. The tag of the instruction can be longer than one line, but each line has all of this information written into the datafile with it, so a three-line tag would have three copies of the student, lesson, area, and time information.

The –outputl– instruction is used to place consecutive variable values in the datafile, with a label to identify the part of the lesson containing the –outputl– instruction.

The label is the first argument of the tag. This label is placed, together with the other overhead information, into the datafile each time the instruction is executed. Also placed into the datafile are the variables specified in the second and third arguments. The second argument gives the starting variable, and the third gives the number of variables. No more than 20 variables can be stored by an –outputl– instruction, and the variables stored must be contiguous. If the –outputl– instruction is used without a label, no overhead information is stored.

When the author is in the datafile editor looking at the datafile, the –outputl– instruction shows the variable contents in integer, floating-point, octal, and alphanumeric formats.

READING DATA INTO A LESSON

If the author desires to manipulate data in a manner other than what is available from the data editor, some

information can be read into a lesson from the datafile. The information that can be accessed in this manner is that from area summaries, from –outputl– instructions, and from signoff data.

However, before information can be read into the lesson, a –readset– instruction must be executed. The tag of the –readset– instruction consists of one to three arguments. The first argument is the name of the datafile or the name of the course file. The optional second argument is the code word of the datafile or course. The code word is enclosed in single quotes or it may be a variable, and it must be included in the tag when the inspect or change code words of the receiving lesson do not match the code words of the datafile or course. The optional third argument returns the number of students in the course when the first argument is a course name.

If the first argument is a datafile name, the third argument returns the number of unused records remaining in the datafile. This is set to -1 when the datafile is full, to 0 when storage into the last record begins, to 1 when storage into the next-to-last record begins, and so on.

The system-reserved word zreturn can be used to check if there is information on the datafile. If an attempt is made to read data from an empty file, an execution error results.

The system-reserved word zreturn is also used to check for the existence of readable data. The –readd– instruction reads data from a datafile into an existing lesson and causes an execution error if there is no appropriate data to be read. The value of zreturn is -1 if there is more data, and it is 0 if the end of the datafile is reached. The author must include a check of reserved-word zreturn in the lesson.

If the author desires the most complete information possible, he should wait until all students have stopped executing the lesson. The data is not complete if students are still executing one or more lessons that send information to the datafile. However, datafiles are automatically checkpointed about every 8 minutes in the same manner as common and student records.

The –readd– instruction reads one of three types of data from the datafile: –area– summaries, –outputl– information, and signoff information. The first argument of the tag specifies which type of data to read by one of the keywords: area, outputl, or signoff. A –readset– instruction must be successfully executed before a –readd– is attempted. An end of file check must be done using zreturn. The –readd– reads the datafile sequentially.

The –readd area– instruction reads area summary data from the datafile to student or common variables. The second argument of the tag is the beginning of the variables in which the data is to be stored, and the third argument is the number of variables. If the entire area summary data is to be stored, 15 variables are needed in the block of variables. This instruction, when executed, reads the next area summary block of data from the datafile. If there is no such block of data, an execution error results. The contents of the 15 words from the area summary is given in table 13-2.

TABLE 13-2. AREA SUMMARY DATA STORAGE

Location†	Contents
0	First part of student name
1	Second part of student name
2	Lesson name
3	Area name
4	Elapsed time in the area (in milliseconds)
5	Number of –arrow– instructions in the area
6	Number of ok judgments in the area
7	Number of ok judgments on the first attempt
8	Number of anticipated no judgments (matched by –wrong– or –wrongv–)
9	Number of unanticipated no judgments
10	Number of help requests satisfied
11	Number of help requests not satisfied
12	Number of term requests satisfied
13	Number of term requests not satisfied
14	Completion/noncompletion of area (-1 if completed, 0 if not completed)

† Locations are relative to the starting variable in the block of variables containing the area summary. Thus, if nc300 is the first (starting) variable, the lesson name is in nc302, the area name is in nc303, and so on.

The –readd outputl– instruction reads data placed in the datafile by an –outputl– instruction, from the datafile to the lesson. The number of words (variables) necessary to store this information can vary between 7 and 27. The variability occurs because the author specifies the number of variables whose contents is to be stored in the –outputl– instruction. Seven variables are required as a minimum for the overhead information from the corresponding –outputl– instruction. The storage of the information is given in table 13-3.

The second and third arguments of the –readd outputl– instruction are precisely like those of the –readd area– instruction. The only difference is that the number of variables required can be as many as 27 or as low as 7. If an attempt is made to store the information from an –outputl– instruction in fewer words than are necessary for storage of all the information, part of the tag of the

TABLE 13-3. –outputl– DATA STORAGE

Location†	Contents
0	Number of variables saved
1	First word of student's name
2	Second word of student's name
3	Lesson name
4	Area name
5	Execution time (from –dataon–) of –outputl– (in milliseconds)
6	–outputl– label
7-16	Tag of –outputl–

† Locations are relative to the starting variable in the block of variables containing the data from the –outputl– instruction. Thus, if nc300 is the first (starting) variable, the lesson name is in nc303, the area name is in nc304, and so on.

–outputl– instruction is lost. Conversely, if more space is allocated for storage than is required, the remainder is filled with zeros. If the information in the datafile is the result of an –outputl– instruction without a label so that there is no overhead information, the first seven variables of the block specified by –readd outputl– instruction are set to zero.

The –readd signoff– instruction functions in much the same manner as the two previous forms of the –readd– instruction; however, data is entered as a result of executing a –dataoff– instruction or as a result of exiting from the lesson. Seven words are necessary to store all the data read by this instruction. The information transferred is given in table 13-4.

Student variables are read from the course records by using the –readr– instruction. The –readr– instruction can either be set to read only one record, or the –readr– can continue to execute and read the next record in sequence. A –readset– instruction (naming the course) must be executed successfully before executing a –readr– instruction.

The –readr– instruction has three basic forms, with the keyword in the tag indicating the type of –readr– instruction to be executed: name, sequential, or roster. The name and sequential forms have the same continuation tag lines available for specifying student data options: student statistics, student variables, router variables, ldone information, and lscore information. At least one and as many as five tag lines may accompany each –readr– instruction. The roster form of the instruction does not have continuation lines.

The author must do an end-of-file check with the system-reserved word zreturn. The values of zreturn are:

TABLE 13-4. SIGNOFF DATA STORAGE

Location†	Contents
0	First word of student name
1	Second word of student name
2	Lesson name
3	Elapsed time for this session (in minutes)
4	Total time to complete the lesson (in minutes) if the lesson was completed this session; if the lesson was not completed this session, this word contains –1
5	Date
6	Time

† Locations are relative to the starting variable in the block of variables containing the signoff information. Thus, if nc300 is the first (starting) variable, the date is in nc305, the time is in nc306, and so on.

Value	Meaning
-1	More information is in the file
0	When last record of course is read
1	Name is not in the course
2	Attempted to read ldone or lscore information when "mrouter" not in use

Continuing to read the course file after all records are read (that is, zreturn equal to 0) causes an execution error.

The –readr name– instruction reads the specified student information (student statistics, student variables, router variables, ldone information, and lscore information) into a work space (student variable area or common area) for inspection by the author or instructor. As many named records as desired can be read, but a –readr– instruction must be provided for each record. Additionally, the –readr name– instruction sets a pointer to that name so that a subsequent –readr sequential– instruction starts with the next name in the roster.

The first –readr sequential– instruction reads the record of the first student in the course file (desired student data is specified in the tag lines) from the roster into a specified work space for inspection by the instructor. Each –readr sequential– instruction reads the next student's record; this process continues until the roster is exhausted.

The –readr name– and –readr sequential– instructions read student statistics from the course records, using eleven variables to store all the student statistics. These variables are v-type variables. The contents of the eleven words of student statistics is given in table 13-5. To display these variables, use the –showa– instruction for the first six and the –show– instruction for the last five.

TABLE 13-5. STUDENT STATISTICS STORAGE

Location †	Contents
0	First word of student name
1	Second word of student name
2	User type
3	Date created
4	Last day on
5	Last time on
6	Total hours on system
7	CPU usage in TIPS
8	Total days on system
9	Number of sessions on
10	Cumulative DAPM

† Locations are relative to the starting variable in the block of variables containing the student statistics. Thus, if nc300 is the first (starting) variable, the first word of the student's name is in nc300, the second word of the student's name is in nc301, and so on.

The –readr name– and –readr sequential– instructions can provide information from the system-reserved words ldone and lscore only when "mrouter" is in use. This information is stored in student variables or common. The ldone information is in 3-bit signed segments and the lscore information is in 8-bit signed segments. Therefore, one computer word stores ldone information for 20 lessons, or one computer word stores lscore information for seven lessons. Refer to appendix B for values of ldone and lscore. If the –readr name– or –reader sequential– instruction attempts to read ldone or lscore information when "mrouter" is not in use, system-reserved word zreturn is set to 2.

The –readr roster– instruction provides a list of student names. Each student name takes two words; therefore, 30 words are needed for a list of 15 names. The total number of students in a course is returned in the optional third argument of the –readset– instruction.

Once the student record data is in the lesson, it can be sorted to eliminate unwanted information, it can be packed, or it can be manipulated in any manner.

NOTES

The author can collect student comments during a lesson with the –notes– instruction. The system sends these notes to the student note file named on the course information page. If this is NONE or left blank, the notes go to the lesson note file named on the lesson information page. If the author specifies this as NONE or leaves it blank, the note is not sent anywhere.

The –notes– instruction has three forms. Upon execution of the –notes– instruction with a blank tag, the system initiates a TERM-comments (erases lines 30 and 31 and gives help on line 32). The student can enter a comment of up to 20 lines or 120 words and after completing it, can give it an identifying title. The system stores this note with the usual heading consisting of date, time, and sign-on and adds a subheading consisting of lesson, unit, and site information. (The student can also initiate a TERM-comments from anywhere in the lesson by pressing the TERM key and entering the word comments.)

The second form of the –notes– instruction has two arguments. The first argument is an n- or v-type variable containing author-supplied heading information. This information must be in a form suitable for display by the –text– instruction. The second argument is the word length of the heading information. Upon execution, the system allows the student to enter a comment and an identifying title. The system stores this note with the usual heading and adds a subheading consisting of the name of the lesson and the author-supplied information. The instructions

```
pack    n12,n11, ⋖ s,ntries ⋗ attempts at arrow in
        unit ⋖ a,zunit ⋗.
calc    n10 ⇐ int(n11/10)+1
notes   n12,n10
```

provide student feedback to the author if the student enters a comment. The student can choose to continue with the lesson without entering a comment by pressing the SHIFT BACK keys.

The third form of the –notes– instruction has three arguments. The first two arguments are the same as in the previous form of the –notes– instruction. The third argument is the keyword send. Upon execution, the system stores this note with the usual heading and a subheading consisting of the name of the lesson. The body of the note is the author-supplied information. The system titles the note with the name of the lesson. The student cannot enter a comment or the title with this type of note and is not aware that a note has been sent.

After execution of the –notes– instruction, execution resumes on the same display with the next author language instruction, and the system-reserved word zreturn is set to the following values.

Value	Meaning
-1	Note sent
0	Student did not send note (pressed SHIFT BACK)
1	TERM-comments not allowed
2	Error in format of heading information
3	Error in connecting note file

Site directors can manage terminal and ECS resources at a logical site with a site lesson. A logical site is a group of terminals which share ECS. A site lesson is a lesson specified by the site director who uses it to control terminal use and ECS use. A lesson must be specified as a site lesson to enable execution of the –site– and –station– instructions.

-SITE- INSTRUCTION

The –site– instruction returns information about terminals and ECS at a logical site. The instruction has four forms, each with its own keyword in the first argument of the tag and each returning a value in system-reserved word zreturn.

The –site set– instruction enables the other –site– instructions for the logical site named in the second argument of the tag. The first argument of the tag is the keyword set. The system-reserved word sitenam can be used to determine the name of the user's logical site. Succeeding –site set– instructions cancel previous ones. The system-reserved word zreturn is set to -1 if the –site set– instruction executes correctly and to 0 if the lesson containing this instruction is not a site lesson for the site named.

The –site info– instruction gets the current site ECS information for the site named in the –site set– instruction and stores the information in four words, starting at the variable named in the second argument. The contents of these words follows.

Word	Contents
1	ECS base allotment
2	ECS current allotment
3	Amount of ECS in use
4	Number of active terminals

The system-reserved word zreturn is set to -1 if the –site info– instruction executes correctly and to 0 if no site is set.

The –site active– instruction finds the station numbers of the active terminals for the site named in the –site set– instruction. The first argument in the tag is the keyword active. The second argument in the tag is the number of the station at which checking begins. The numbers of the active stations are stored in variables beginning with the variable in the third argument. The fourth argument determines the number of variables needed to store the numbers of the active stations. This argument specifies the number of active stations to be found. If this number is greater than the actual number of active stations, the final variable is set to -1. The system-reserved word zreturn is set to -1 if the instruction executes correctly, to 0 if no site is set, and to 1 if the starting station number is incorrect.

The –site stations– instruction finds the station numbers permanently on the site named in the –site set– instruction. The first argument in the tag is the keyword stations. The second, third, and fourth arguments are the same as the corresponding arguments in the –site active– instruction except that they look for permanent stations instead of active stations. The system-reserved word zreturn is set to the same values as with the –site active– instruction.

-STATION- INSTRUCTION

The –station– instruction returns information about individual stations (terminals) in the site named in the –site set– instruction and controls the use of those stations. The instruction has seven forms, each with its own keyword in the first argument of the tag and each returning a value in system-reserved word zreturn. The system-reserved word station can be used to determine the number of the user's station. The system stores the station number as a single number, obtained by multiplying the site number by 32 and adding the station number within that site. For example, if the user's site and station number is 4 - 13, the system stores the station number as (4X32)+13=141.

The –station info– instruction obtains information for the station specified in the second argument of the tag and stores the information in 10 words, starting at the variable named in the third argument. The contents of these words follows.

Word	Contents
1	First word of user name
2	Second word of user name
3	User's course
4	Type of user record
5	User's account
6	Session statistics in three 20-bit fields (number of disk accesses, number of CPU seconds used, and elapsed time)
7	Name of lesson or system lesson
8	ECS usage in four 15-bit fields (first field empty, ECS storage, ECS common, and lesson ECS)
9	Name of router
10	Router ECS usage in four 15-bit fields (same as word 8)

After execution of the –station info– instruction, the system-reserved word zreturn is set to the following values.

Value	Meaning
-1	Execution successful
0	No site set
1	Station number error
2	Station not in site
3	Station inactive

The –station status– instruction gets the current status of the station specified in the second argument of the tag and returns the status in system-reserved word zreturn. Values of zreturn are as follows.

Value	Meaning
-2	Station is signing on
-1	Station active
0	No site set
1	Station number error
2	Station not in site
3	Station inactive
4	Station is signing off
5	Station locked out

The –station send– instruction sends a message to another terminal. The first argument in the tag is the keyword send, the second argument is the number of the station to which the message goes, the third argument is the coarse-grid coordinates of the screen location, the fourth argument is the message, and the fifth argument is the length of the message in characters. The instruction sends the message in mode rewrite and resets the mode and screen position of the receiving terminal to its previous state. After execution of –station send–, zreturn is set to the following values.

Value	Meaning
-1	Message sent
0	No site set
1	Station number error
2	Station not in site
3	Station inactive or tried to send to own station

The –station stop1– instruction presses the SHIFT STOP keys at the station specified in the second argument of the tag. The system-reserved word backout is set to 1 at that station until the station enters a non-router lesson, allowing finish units and routers to distinguish between a SHIFT STOP pressed by the system and one pressed by the user. After execution of –station stop1–, zreturn is set to the following values.

Value	Meaning
-1	SHIFT STOP pressed
0	No site set
1	Station number error
2	Station not in site
3	Cannot press SHIFT STOP on this station

The –station logout– instruction signs off the user at the station specified in the second argument of the tag. The system-reserved word backout is set to -2 at that station until it is completely signed off. It then displays the message, Press NEXT to begin. After execution of –station logout–, zreturn is set to the following values.

Value	Meaning
-1	Station signed off
0	No site set
1	Station number error
2	Station not in site
3	Cannot sign off this station

The –station off– instruction turns off the station specified in the second argument of the tag, locking out use of the terminal. The system-reserved word backout is set to -2 at that station and the terminal displays the message - -terminal not available- -. After execution of –station off–, zreturn is set to the following values.

Value	Meaning
-1	Station turned off
0	No site set
1	Station number error
2	Station not in site
3	Cannot turn off this station

The –station on– instruction turns on the station specified in the second argument of the tag, clearing the off status for a locked out station. The terminal displays the message, Press NEXT to begin. After execution of –station on–, zreturn is set to the following values.

Value	Meaning
-1	Station turned on
0	No site set
1	Station number error
2	Station not in site
3	Station already active

When an author wants a hardcopy printout of a lesson, he can specify to the printer exactly what he wants printed using the –*list– instruction. This instruction, along with its various tags, is inserted in a lesson and is noticed by the printer when it makes its pass through the lesson but is ignored when a lesson is condensed in the system (because of the *). Usually, the safest place to put –*list– options in a lesson is at the beginning of block B.

An error directory is printed toward the end of the listing of the lesson. It lists the line numbers on which errors occurred. The errors flagged are –*list– option errors and duplicate unit names. The printer does not flag regular condense-type errors.

-*LIST- INSTRUCTION

The –*list label,**string**– instruction labels small sections of the lesson. Because this instruction puts blank lines above and below the label with no line numbers, it stands out better than a comment. The position in the printout of the label given in **string** is the same as where it is placed in the instruction in the lesson. To center the label, **string** must be centered in the instruction. The –*list label– is blanked out. The –*list title,**string**– instruction ejects a page and prints the title given in **string**.

To eject a page from anywhere in the source of a lesson, use –*list eject–. This instruction ejects a page and begins printing again with the line immediately following the instruction. The standard heading is printed.

The –*list text– instruction tells the printer to list only the tags in –write– and –writec– instructions. The conditional expression of the –writec– instruction is blanked out, and anything embedded in these instructions is also blanked out.

The –*list ignore– instruction specifies to the printer that all –*list– instructions after this one be ignored. This can be helpful when an author wants to put all the –*list– options at the beginning of a lesson where they can be found. If the options are ended with –*list ignore–, the author is assured that no other –*list– instructions hidden later in the lesson will be executed.

Lesson information from the lesson directory display is listed toward the end of the printout when the –*list info– option is used.

The –*list symbols– instruction causes a cross-reference table of symbols to be printed at the end of the listing. The symbols included are variables, defined names, and system-reserved words. The dummy arguments in function definitions are also listed. The reference table includes only those symbols used in the printed portions of the lesson.

To produce a table listing all the places where certain instructions are used in a lesson, the –*list commands,**list**– option is used. Up to 10 instructions can be specified in **list**. This instruction does not continue; to specify more than one line of instructions in **list**, the –*list commands– must be repeated.

The –*list off– instruction stops the source listing and proceeds to print any tables previously selected. To prevent printing certain blocks, use –*list off,**blocks**–. An example is:

```
*list      off,triangle-square,circle,rectangle–
```

This prevents printing blocks triangle through square, block circle, and all blocks from rectangle on. These blocks can be common, micro, charset, or normal blocks.

The –*list parts– instruction allows only condensed blocks to be printed. Blocks marked by the parts option in a lesson are not printed since they are marked to not be condensed.

The –*list charset(db)– option causes any charset in the lesson to be printed. The on dots are marked with the first character in the parenthesis (d), and the background dots are marked with the second character (b). A more readable charset listing may be obtained using –*list charset(*)–, which marks each dot with * and leaves the background blank. The default case, –*list charset–, marks on dots with 0 and background dots with –. This option must precede any –*list off– instruction in order for the charset to be printed.

A normal print of a lesson does not list leslist, micro, or vocab blocks. To print a block of this kind, the appropriate –*list leslist–, –*list micro–, or *list vocabs– instruction must precede the block to be printed.

When the mod words option for the lesson editor is turned on, the –*list mods– instruction prints the mod word and the corresponding line of code that was changed. Mod words are an option in the editor to record line-by-line changes in a lesson. They show the first 5 characters of the user's name and the date of the change. The –*list deleted– instruction prints deleted lines with an asterisk at the beginning of each deleted line.

PRINTING COMMONS AND DATASETS

Print commons using instruction –*list common,**comname**, **words,format**–. When printing a common, if a preceding source block has the same name as the common blocks, the printer does not see the common, and it is not printed. Datasets do not use a –*list– instruction in the lesson. They are printed according to the print directives given on the data page of the dataset. The formats that commons and datasets use are similar. These formats may be one of the following.

Integer or i (nc variables)

Exponential or e (vc variables)

Octal or o

Alpha or a

Hexadecimal or h

x (all of the above)

(Your own format)

Special or s

Alpha (a), integer (i), and exponential (e) print 10 words per line, octal (o) prints 5 words per line, and x prints 2 words per line. An example of the instructions used to print a common is:

```
*list      common,septvar,320,o
```

This prints the first 320 words of the common named septvar in octal format.

To print a dataset, insert the appropriate directives on the data page of the dataset. These directives include page eject information if desired, starting record number, number of records to print, and format. An example is:

```
2,7,a;2,7,e
```

This instruction prints records 2 through 8 in the dataset, first in alpha format and then in exponential format.

If the directive is omitted on the data page, the entire dataset is printed in special format.

To print a nameset (a type of dataset), insert the appropriate directives on the data page of the nameset. These directives include name, page eject information if desired, starting record number, number of records to print, and format. Some examples are:

```
name1;1,3,o
name1-name10;2,4,h
```

The first instruction prints records 1 through 3 of name1 in octal. The second instruction prints records 2 through 4 for names name1 through name10 in hexadecimal.

If the directive is omitted on the data page, the entire nameset is printed in special format. The directive can specify more records than a name has, but only existing records are printed. A name with no records is not printed.

An automatic page eject occurs after each name in a nameset for all formats. The following comments on datasets refer also to namesets.

An author may design his own format for a line of print. The format information is enclosed in parentheses.

```
*list      common,comname,words,(your format)
```

(your format) may consist of characters, numbers, and spaces or commas or both.

Characters:

- a Alpha (10 characters)
- e Exponential (vc variables, 10 characters)
- i Integer (nc variables, 10 characters)
- o Octal (20 characters)
- h Hexadecimal (15 characters)
- x Number of spaces between words (the number preceding the x is the number of spaces; default is 1)
- p Number of words to go forward before printing (the number preceding the p is the number of words to go forward; default is 1)
- l Common or dataset location (four characters)

Numbers:

0 through 9

An example of three equivalently designed formats is:

```
(a 4x i 4x o 4x e)
(a,4x,i,4x,o,4x,e)
(a4xi4xo4xe)
```

These formats print each word in four formats with four spaces between words, taking 62 characters. Each line can have up to 135 characters. The letter p may be used to print more than one word per line.

```
(o 5x p o 5x p o 5x p o 5x p o)
```

This format uses 120 characters per line.

Special format ignores **words**, but the field must be there. An example of the acceptable form is:

```
*list      common,janvar,,s
```

When printing a dataset in special format, the format is specified on the data page. The number of records is ignored, but the field must be there. To print the entire dataset in special format, one of the equivalent instructions, -l,,s- or -special-, may be used.

Special or s format prints the common or dataset in alpha format. Therefore, all numerical information must be converted to alpha strings before storing in the common or dataset. To put spaces between information on the print, use the -pack- instruction to pack the common or dataset words with blanks (code is 055) and then use the -move- instruction to insert the information to be printed.

In special format, lines are limited to 120 characters, and words in which all bits are 0 are ignored unless they are preceded by at least one nonzero word on the same line. The last word on a line must end with 12 bits set to 0. If you cannot alter the last word so that the final 12 bits are 0, add a word after it with at least the last 12 bits set to

0. Do this by adding a word with all 0 bits or a word with eight blank characters and the final 12 bits set to 0: o55555555555555550000. A space code (055) may be inserted between characters within a word for printed blanks. For blanks between words, insert a space code for each space.

The special or s format is the only format which uses page eject information. (Page ejects are automatic with the other formats, printing about 56 lines of the specified format per page.)

Page ejects are specified with –*format eject–, –*format blocks–, –*format records–, or –*format pages–. These directives must be contained in the words of the common. Each takes two words. These directives may also be in the words of a dataset, or they may be included in the specifications on the data page of a dataset.

For a page eject after each block or record, insert –*format blocks– or –*format records– at the location in the common or dataset where it is to take effect, often at the beginning. The last word of each block or record must end with 12 bits set to 0. These directives may be specified on the data page of the dataset by –blocks–, –records–, or –blocks;records–. Another acceptable form is –blocks;3,,s–.

For automatic page ejects, insert –*format pages– in the common or dataset at the location where it is to take effect, often at the beginning. This directive provides uniform margins at the top and bottom of each printed page. On the data page of the dataset, it is indicated by –pages– or by –pages;2,,s–.

The –*format eject– at the beginning of a dataset or common starts the print on a new page. If –*format eject– is embedded in a common or dataset, the new page begins after the instruction. The word preceding –*format eject– must end with 12 bits of 0's. To insert –*format eject– in the two words it needs, use the –pack– instruction.

To end the print before the end of the file, insert –*format end– with the –pack– instruction into the next word following the last one to be printed, which must end in 12 bits of 0's. The special print ends at –*format end–, end of file, or the first noncommon block.

Character codes and key codes are not the same on the PLATO system. Character codes are the 6-bit codes stored in variables and used for display. Key codes are numbers associated with each key. The character codes of a response are put in the judging copy of the student response. In contrast, the system-reserved word key contains the key code of the last key pressed (refer to table A-1).

Some keys have names associated with them, as well as key codes. Many of the function keys have a name and a key code but no character code, as they do not display a character. An example of this is the NEXT key.

Because there are 126 characters available (alternate characters use the same character codes, with a preceding FONT character code), and the 6-bit character codes can only specify 63 codes, some keys have two or three character codes rather than one character code associated with them.

ACCESS CHARACTERS

These characters (table A-2) are obtained by pressing the ACCESS (shifted □ key) key and then an associated key. Refer to appendix D for a description of this process.

TABLE A-1. CHARACTER AND KEY CODES

Character	Character Code (octal)†	Key Code (octal)††	Character	Character Code (octal)†	Key Code (octal)††
a	01	001	⇐	65	065
b	02	002	(SUB)	66	066
c	03	003	(SUP)	67	067
d	04	004	(SHIFT ⇐)	70	070
e	05	005	(CR)	71	071
f	06	006	<	72	072
g	07	007	>	73	073
h	10	010	(bksp)	74	074
i	11	011	(FONT)	75	075
j	12	012	(ACCESS)	76	076
k	13	013	;	77	077
l	14	014			
m	15	015	A	7001	101
n	16	016	B	7002	102
o	17	017	C	7003	103
p	20	020	D	7004	104
q	21	021	E	7005	105
r	22	022	F	7006	106
s	23	023	G	7007	107
t	24	024	H	7010	110
u	25	025	I	7011	111
v	26	026	J	7012	112
w	27	027	K	7013	113
x	30	030	L	7014	114
y	31	031	M	7015	115
z	32	032	N	7016	116
0	33	033	O	7017	117
1	34	034	P	7020	120
2	35	035	Q	7021	121
3	36	036	R	7022	122
4	37	037	S	7023	123
5	40	040	T	7024	124
6	41	041	U	7025	125
7	42	042	V	7026	126
8	43	043	W	7027	127
9	44	044	X	7030	130
+	45	045	Y	7031	131
-	46	046	Z	7032	132
*	47	047			
/	50	050	–	7041	141
(	51	051	'	7042	142
)	52	052	Σ (SHIFT+)	7045	145
$	53	053	Δ (SHIFT-)	7046	146
=	54	054	?	7050	150
(space)	55	055	"	7056	156
,	56	056	!	7057	157
.	57	057	∩ (SHIFT ÷)	7060	160
÷	60	060	∪ (SHIFT X)	7064	164
[	61	061	(locksub)	7066	166
]	62	062	(locksup)	7067	167
%	63	063	:	7077	177
X	64	064			

†The character code is the actual internal code used in processing information. (Value of codes in literal strings.)

††The key code identifies terminal key(s) pressed for use relative to checking.

TABLE A-2. ACCESS CHARACTERS

Lowercase			Uppercase		
Character	Character Code (octal)	Actual Key (with ACCESS)	Character	Character Code (octal)	Actual Key (with ACCESS)
(erase mode)†	767035	SHIFT ⇐ then 2	≡ (identity)	7652	)
(write mode)†	767036	SHIFT ⇐ then 3	#	7653	$
(rewrite mode)†	767037	SHIFT ⇐ then 4	{	7661	[
α	7601	a	}	7662	]
β	7602	b	(form feed)	7671	CR
¸ (cedilla)	7603	c	≤	7672	<
δ	7604	d	≥	7673	>
´ (acute accent)	7605	e	←	767001	A
λ	7614	l	©	767003	C
μ	7615	m	→	767004	D
~ (tilde)	7616	n	|	767011	I
° (degree)	7617	o	↑	767027	W
π	7620	p	↓	767030	X
` (grave accent)	7621	q	×	767060	SHIFT X
ρ	7622	r	(lock down one line)	767066	SUB1††
σ	7623	s	(lock up one line)	767067	SUP1††
θ	7624	t			
¨ (umlaut)	7625	u			
v (hacek)	7626	v			
ω	7627	w			
^ (circumflex)	7630	x			
(half space)	7631	y			
(half backspace)	7632	z			
≮	7633	0			
≯	7634	1			
@	7640	5			
≫	7641	6			
≪	7645	+			
\	7650	/			
≠	7654	=			
(half space)	7655	(space)			
↕	7656	,			
∘	7664	X			
(line feed)	7665	⇐			
(down one line)	7666	SUB			
(up one line)	7667	SUP			
(half backspace)	7674	(bksp)			
~ (equivalence)	7677	;			

† Used in animation displays.

†† The number 1 after the key name indicates that the SHIFT key must also be pressed.

FUNCTION KEYS

These keys do not have an associated character code, as they are never displayed. For clarity, the key name should be used (as when testing a variable key) rather than the key code (refer to table A-3).

TABLE A-3. FUNCTION KEYS

Key/System-Defined Key Name (if different)	Key Code (octal)	Key/System-Defined Key Name (if different)	Key Code (octal)
FUNKEY†	200	TERM	216
NEXT	202	ANS	217
NEXT1††	203	COPY	220
ERASE	204	COPY1††	221
ERASE1††	205	EDIT	222
HELP	206	EDIT1††	223
HELP1††	207	MICRO	224
BACK	210	(square)	225
BACK1††	211	STOP	226
LAB	212	TAB	230
LAB1††	213	TIMEUP function†††	233
DATA	214	CATCHUP function††††	235
DATA1††	215		

†Special-purpose key which is not associated with any key on the keyboard.

††The number 1 after the key name indicates that the SHIFT key must also be pressed.

†††The key returned at the end of a time period specified in a -time- or a timed -pause- instruction.

††††The key returned whan all output is displayed following a -catchup- instruction.

SYSTEM FUNCTIONS AND RESERVED WORDS B

The PLATO author language predefines several functions for the author. These can be used in mathematical expressions or in the definition of other functions. Additionally, there are several reserved words, which are also useable in mathematical expressions and definitions. The reserved words furnish the author with information otherwise unobtainable or obtainable only after extensive manipulation by the author. Additionally, some key codes are named. These are given in appendix A.

System functions, together with a brief description of each, are given in table B-1. Reserved words are given in table B-2.

All of these names can be redefined by the author using a –define– instruction. In such a case, the author's specification has priority, and the reserved words become unavailable to the author. Hence, the names of reserved words should be redefined only if the author is certain that the information contained in those words is not necessary.

TABLE B-1. SYSTEM FUNCTIONS

Function†	Description
abs(x)	Absolute value
arctan(x)	Arctangent (value in radians)
bitcnt(x)	Returns the number of bits set to 1 in the argument
comp(x)	Bit complement of x
cos(x)	Cosine (x in radians)
exp(x)	Returns value of e (2.713+) raised to the power of the argument
frac(x)	Fractional part of the argument (x-int(x))
int(x)	Integer part (largest integer with a value $\leq$ x)
log(x)	Common logarithm (base 10)
lmask(x)	Left-justified mask of x bits
ln(x)	Natural logarithm (base e)
not(x)	Boolean (logical) negation
rmask(x)	Right-justified mask of x bits
round(x)	Value of the argument rounded to the nearest integer
sign(x)	Returns $\pm$1 corresponding to sign of x, or 0 for x equal to 0
sin(x)	Sine (x in radians)
sqrt(x)	Square root **NOTE** Attempting to find sqrt(x) if x is negative results in an execution error, and execution of the lesson is stopped.
varloc(x)	Returns the number of user variable x

TABLE B-1. SYSTEM FUNCTIONS (Contd)

Function†	Description
And(A)	-1 if every element of array A is true
Max(A)	Maximum value in array A
Min(A)	Minimum value in array A
Or(A)	-1 if any element of array A is true
Prod(A)	Product of elements in array A
Rev(A)	Reverse of array A
Sum(A)	Sum of elements in array A
Transp(A)	Transposition of array A

†In these definitions, x is a dummy argument that can be replaced by any permissible mathematical expression, and A is a dummy argument that can be replaced by any system-defined array.

TABLE B-2. SYSTEM-RESERVED WORDS

Name	Description
	Words for General Use
args	Last number of arguments transferred to an argumented unit
backout	Distinguishes between student sign-off (SHIFT STOP) and system backout: 0 Not system backout not 0 System backout
baseu	Name of current base unit, 0 if no base unit
clock	Value of system clock in seconds (nearest millisecond)
error	Returns values for specific instructions; being phased out by zreturn
key	Contains 10-bit key code of last key pressed
mainu	Name of current main unit
mallot	ECS allotment for the logical site in which the user is registered
muse	Total ECS usage by users registered at same logical site
nhelpop	Number of successful requests for help on the same page (–helpop– type branch) cleared on entry to a new main unit and every time an –arrow– instruction is found while not in a –helpop– state
proctim	CPU usage in seconds (floating-point number); updated only at the beginning of a timeslice
ptime	-1 if current time is prime time, and 0 if not
sitenam	Name of user's logical site
station	Number assigned to terminal by system

TABLE B-2. SYSTEM-RESERVED WORDS (Contd)

Name	Description
tactive	Total number of active terminals currently on the system
user	Type of user: student, author, multiple, instructor, sabort (student records are aborted), or snockpt (autocheck is aborted)
usersin	Number of users currently in lesson (routed students are counted as being in the router and in the routed lesson)
zaccnam	Name of account containing user's course, left justified
zcondok	0 if condense errors or warning messages, and -1 if none
zcusers	Number of users signed into current common
zdsname	Name of current dataset or nameset
zdusers	Number of users currently connected to dataset 0 No dataset connection 1 User requesting value is the only user currently connected
zfroml	Name of –from– lesson
zfromu	Name of –from– unit
zgroup	Name of user's group
zinfo	Contains 24 bits of additional associated information for the current name in a nameset
zlesson	Name of current lesson
zpnfile	-1 if course is prepared for personal notes, and 0 if user has no personal notes file
zpnotes	-1 if user has personal notes not yet read, and 0 if user has read all personal notes
zretrnu	Name of the unit user returns to after a –jumpout return,return–
zreturn	Status returned from specific instructions
zsessda	Disk accesses this session
zsesset	Elapsed time this session
zsesspt	Processing time this session
zsnfile	Used with student notes: -1 Student note file is attached to course 0 No student note file specified 1 Lesson and student notes disabled
zsnotes	-1 if student has student notes not yet read, and 0 if student has read all student notes
zterm	Last term requested
zunit	Name of current unit
	Words for Displays
mode	Current text display mode: -1 Erase 0 Rewrite 1 Write

TABLE B-2. SYSTEM-RESERVED WORDS (Contd)

Name	Description
size	Current –size–
sizex	Current horizontal –size–
sizey	Current vertical –size–
where	Coarse-grid coordinates for next screen output
wherex	Fine-grid horizontal coordinates for next screen output
wherey	Fine-grid vertical coordinates for next screen output
ztouchx	Fine-grid horizontal coordinates of center of last touched square, -1 if last input was not a touch
ztouchy	Fine-grid vertical coordinates of center of last touched square, -1 if last input was not a touch
	Words for Lengths
lcommon	Length in words of common currently in use
llesson	Length in words of current lesson
lstorag	Length in words of storage currently in use
zbpc	Bits per character
zbpw	Bits per word
zcpw	Characters per word
zdsrecs	Same as zrecs; being phased out by zrecs
zdswpr	Same as zwpr; being phased out by zwpr
znscpn	Number of characters per nameset name
znsmaxn	Maximum number of names per nameset
znsmaxr	Maximum number of records per nameset
znsnams	Current number of names in use in the nameset
znsrecs	Current number of records in entire nameset
zrecs	Number of records in dataset if connected to a dataset, or number of records in current name if connected to a nameset, 0 if no dataset or nameset is connected
zwpb	Number of words per block
zwpr	Number of words per record in dataset or nameset, 0 if no dataset or nameset is connected
	Words for Answer Judging (0 is condition not true)
ansent	Number of answer-judging instructions encountered before match is made: 0 –store– failure -1 Nothing matches -2 Response is over 40 words
ansok	-1 if all specifications ok
capital	-1 if no capitalization errors

TABLE B-2. SYSTEM-RESERVED WORDS (Contd)

Name	Description
entire	-1 if all required words are present
extra	-1 if no extra (unspecified) words are in response
jcount	Number of internal 6-bit character codes in student response
judged	-1 for any ok judgment, 0 for any anticipated no judgment, and 1 for any unanticipated no judgment
ntries	Number of attempts at arrow
order	-1 if word order ok
phrase	-1 if no errors in phrases
spell	-1 if spelling of response ok
vocab	-1 if all words in student response are in vocabulary
wcount	Number of words in student response (set after –match–, –answer– or –concept–)
Words for Formula Judging	
formok	Error diagnostic information from expression compilation: -1 Formula ok 0 Bad function argument or variable index [for example, sqrt(-3) or n (i+70) where i is 90] 1 Illegal character 2 Unbalanced parentheses 3 Too many decimal points (for example, 34...243) 4 Undefined variable name 5 Logical operator does not have two $ signs (for example, $and or and$ instead of and) 6 Expression has bad form (for example, 3//4 or (/-55) 7 Value assigned to nonvariable (for example, 5 ⇐ 3) 8 Octal constant contains 8 or 9 9 Err in alphanumeric string, such as (3+"ab) 10 A number has too many digits 11 Array index is out of bounds 12 Variables are used with –specs novars– 13 Operations are used with –specs noops– 14 There are assignments without –specs okassign– 15 Improper use of units, such as 5kg+3sec 16 Expression takes too long to compile 17 Too many nested functions 18 Wrong number of arguments in a function 60 Too many temporary variables needed 62 Expanded version too long 63 Too many literals 66 Too many indexed assigns
opcnt	Number of arithmetic operations in response; counts arithmetic operators (+, –, ÷ , x), logical operators (and), bit operators (cls), and full functions (system- or user-defined)
varcnt	Number of variables or functions (user-defined only) in response

TABLE B-2. SYSTEM-RESERVED WORDS (Contd)

Name	Description
	Words for Routers and Leslists
errtype	Used with router lessons: 0 Unknown error type 1 Execution error 2 Fatal condense error 3 Specific fatal condense error: No ECS available Lesson deleted No room in ECS for common Lesson tables full ECS allocation exceeded 4 Error in finish unit of routed lesson 5 SHIFT STOP exit from condense queue
fromnum	Leslist number of lesson the current lesson was entered from; -1 if lesson is not in the leslist or if no leslist is in use
ldone	Set by last –lesson– instruction -1 –end lesson– or –lesson completed– 0 Student has not entered lesson or –lesson incomplete– 1 –lesson no end– 2 Student has entered lesson but not completed it
lessnum	Leslist number of current lesson; -1 if lesson is not in the leslist or if no leslist is in use
lleslst	Length of the leslist
lscore	Score for current lesson; set by last –score– instruction
lstatus	Status of current lesson; set by last –status– instruction
rcallow	Router common variable access, set by last –allow– instruction: 0 No access (–allow–) 1 Read-only (–allow read–) 2 Read and write (–allow write–)
rstartl	Name of current –restart– lesson
rstartu	Name of current –restart– unit
router	Name of router lesson assigned to course
rvallow	Router student variable access, set by last –allow– instruction: 0 No access (–allow–) 1 Read-only (–allow read rvars–)
	Words for Data Keeping (refer to current area only; can use even if data off, if –area– has been executed)
aarea	Name of current area
atime	Elapsed time in area (milliseconds)

TABLE B-2. SYSTEM-RESERVED WORDS (Contd)

Name	Description
aarrows	Number of –arrow– instructions encountered
aok	Number of answers judged ok
aokist	Number of answers judged ok on first try
asno	Number of anticipated no judgments
auno	Number of unanticipated no judgments
ahelp	Number of successful help requests
ahelpn	Number of unsuccessful help requests
aterm	Number of TERM requests found
atermn	Number of TERM requests not found
dataon	-1 if data is being collected; 0 if otherwise

NUMBER SYSTEMS

Any number system may be defined by two characteristics, the radix or base and the modulus. The radix or base is the number of unique symbols used in the system. The decimal system has 10 symbols, 0 through 9. Modulus is the number of unique quantities or magnitudes a given system can distinguish. For example, an adding machine with 10 digits (or counting wheels) would have a modulus of $10^{10}-1$. The decimal system has no modulus because an infinite number of digits can be written, but the adding machine has a modulus because the highest number which can be expressed is 9,999,999,999.

Most number systems are positional; that is, the relative position of a symbol determines its magnitude. In the decimal system, a 5 in the units column represents a different quantity than a 5 in the tens column. Quantities equal to or greater than 1 may be represented by using the 10 symbols as coefficients of ascending powers of the base 10. The number 984_{10} is represented as:

$$9 \times 10^2 = 9 \times 100 = 900$$
$$+8 \times 10^1 = 8 \times 10 = 80$$
$$+4 \times 10^0 = 4 \times 4 = 4$$
$$984_{10}$$

Quantities less than 1 may be represented by using the 10 symbols as coefficients of ascending negative powers of the base 10. The number 0.593_{10} is represented as:

$$5 \times 10^{-1} = 5 \times .1 = .5$$
$$+9 \times 10^{-2} = 9 \times .01 = .09$$
$$+3 \times 10^{-3} = 3 \times .001 = .003$$
$$0.593_{10}$$

BINARY NUMBER SYSTEM

Computers operate faster and more efficiently by using the binary number system. There are only two symbols, 0 and 1; the base equals 2. The following shows the positional value.

$$\ldots 2^5 \quad 2^4 \quad 2^3 \quad 2^2 \quad 2^1 \quad 2^0$$
$$32 \quad 16 \quad 8 \quad 4 \quad 2 \quad 1 \qquad \text{Binary point}$$

The binary number 0 1 1 0 1 0 represents:

$$0 \times 2^5 = 0 \times 32 = 0$$
$$+1 \times 2^4 = 1 \times 16 = 16$$
$$+1 \times 2^3 = 1 \times 8 = 8$$
$$+0 \times 2^2 = 0 \times 4 = 0$$
$$+1 \times 2^1 = 1 \times 2 = 2$$
$$+0 \times 2^0 = 0 \times 1 = 0$$
$$26_{10}$$

Fractional binary numbers may be represented by using the symbols as coefficients of ascending negative powers of the base.

$$2^{-1} \quad 2^{-2} \quad 2^{-3} \quad 2^{-4} \quad 2^{-5} \ldots$$
Binary point 1/2 1/4 1/8 1/16 1/32

The binary number 0.10110 is represented as:

$$1 \times 2^{-1} = 1 \times 1/2 = 1/2 = 8/16$$
$$+0 \times 2^{-2} = 0 \times 1/4 = 0 = 0$$
$$+1 \times 2^{-3} = 1 \times 1/8 = 1/8 = 2/16$$
$$+1 \times 2^{-4} = 1 \times 1/16 = 1/16 = 1/16$$
$$+0 \times 2^{-5} = 0 \times 1/32 = 0 = 0$$
$$11/16_{10}$$

OCTAL NUMBER SYSTEM

The octal number system uses eight discrete symbols, 0 through 7. With base 8, the positional value is:

$$8^5 \quad 8^4 \quad 8^3 \quad 8^2 \quad 8^1 \quad 8^0$$
$$32{,}768 \quad 4{,}096 \quad 512 \quad 64 \quad 8 \quad 1$$

The octal number 513_8 represents:

$$5 \times 8^2 = 5 \times 64 = 320$$
$$+1 \times 8^1 = 1 \times 8 = 8$$
$$+3 \times 8^0 = 3 \times 1 = 3$$
$$331_{10}$$

Fractional octal numbers may be represented by using the symbols as coefficients of ascending negative powers of the base.

$$8^{-1} \quad 8^{-2} \quad 8^{-3} \quad 8^{-4}$$
$$1/8 \quad 1/64 \quad 1/512 \quad 1/4096\ldots$$

The octal number 0.4520 represents:

$4 \times 8^{-1} = 4 \times 1/8 = 4/8 = 256/512$

$+5 \times 8^{-2} = 5 \times 1/64 = 5/64 = 40/512$

$+2 \times 8^{-3} = 2 \times 1/512 = 2/512 = 2/512$

$+0 \times 8^{-4} = 0 \times 1/4096 = 0/4096 = 0/512$

$298/512$

$149/256_{10}$

ARITHMETIC

ADDITION AND SUBTRACTION

Binary numbers are added according to the following rules.

0 + 0 = 0
0 + 1 = 1
1 + 0 = 1
1 + 1 = 0 with a carry of 1

The addition of two binary numbers proceeds as follows (the decimal equivalents verify the result):

Augend	0111	(7)
Addend	+0100	+(4)
Partial Sum	0011	
Carry	1	
Sum	1011	(11)

Subtraction may be performed as an addition.

8 (minuend)	8 (minuend)
-6 (subtrahend) or	+4 (tens complement of subtrahend)
2 (difference)	2 (difference; omit carry)

The second method shows subtraction performed by the adding the complement method. The omission of the carry in the illustration has the effect of reducing the result by 10.

ONE'S COMPLEMENT

The computer performs all arithmetic and counting operations in the binary one's complement mode. In this system, positive numbers are represented by the binary equivalent and negative numbers in one's complement notation.

The one's complement representation of a number is found by subtracting each bit of the number from 1. For example:

1111	
-1001	9
0110	(one's complement of 9)

This representation of a negative binary quantity may also be obtained by substituting 1's for 0's and 0's for 1's.

The value zero can be represented in one's complement notation in two ways.

$0000 \rightarrow 00_2$ Positive (+) zero
$1111 \rightarrow 01_2$ Negative (-) zero

The rules regarding the use of these two forms for computation are:

- Both positive and negative zero are acceptable as arithmetic operands.
- If the result of an arithmetic operation is zero, it is expressed as positive zero.

One's complement notation applies not only to arithmetic operations performed in A but also to the modification of execution addresses in the F register. During address modification, the modified address equals 77777_8 only if the unmodified execution address equals 77777_8 and b equals 0 or (B^b) equals 77777_8.

MULTIPLICATION

Binary multiplication proceeds according to the following rules.

0 x 0 = 0
0 x 1 = 0
1 x 0 = 0
1 x 1 = 1

Multiplication is always performed on a bit-by-bit basis. Carries do not result from multiplication, since the product of any two bits is always a single bit.

Decimal example:

multiplicand	14	
multiplier	12	
partial products	28	
	14	(shifted one place left)
product	168_{10}	

The shift of the second partial product is a shorthand method of writing the true value 140.

Binary example:

multiplicand	(14)	1110	
multiplier	(12)	1100	
partial products		0000	
		0000	shift to place digits in proper columns
		1110	
		1110	
product (168_{10})		10101000_2	

The computer determines the running subtotal of the partial products. Rather than shifting the partial product to the left to position it correctly, the computer right-shifts the summation of the partial products one place before the next addition is made. When the multiplier bit is 1, the multiplicand is added to the running total, and the results are shifted to the right one place. When the multiplier bit is 0, the partial product subtotal is shifted to the right (in effect, the quantity has been multiplied by 10_2).

DIVISION

The following examples show the familiar method of decimal division.

```
                14      quotient
divisor  13 | 185      dividend
              13
              --
              55        partial dividend
              52
              --
               3        remainder
```

The computer performs division in a similar manner (using binary equivalents).

```
                    1110      quotient (14)
divisor  1101 | 10111001      dividend
                  1101
                  ----
                  10100
                   1101
                   ----
                   1110       partial dividends
                   1101
                   ----
                     11       remainder (3)
```

However, instead of shifting the divisor right to position it for subtraction from the partial dividend (shown above), the computer shifts the partial dividend left, accomplishing the same purpose and permitting the arithmetic to be performed in the A register. The computer counts the number of shifts, which is the number of quotient digits to be obtained; after the correct number of counts, the routine is terminated.

CONVERSION

The procedures that may be used when converting from one number system to another are power addition, radix arithmetic, and substitution.

POWER ADDITION

To convert a number from r_i to r_f ($r_i < r_f$), write the number in its expanded r_i polynomial form and simplify using r_f arithmetic.

Example 1: Binary to Decimal (Integer)

$$10111_2 = 1\,(2^4)+0(2^3)+1(2^2)+1(2^1)+1(2^0)$$
$$= 1\,(16)+0(8)\ +1(4)\ +1(2)\ +1(1)$$
$$= 16\ \ +0\ \ +4\ \ +2\ \ +1$$
$$= 23_{10}$$

Example 2: Binary to Decimal (Fractional)

$$.0101_2 = 0(2^{-1})+1(2^{-2})+0(2^{-3})+1(2^{-4})$$
$$= 0\ \ +1/4\ \ +0\ \ +1/16$$
$$= 5/16_{10}$$

TABLE C-1. RECOMMENDED CONVERSION PROCEDURES (INTEGER AND FRACTIONAL)

Conversion	Recommended Method
Binary to decimal	Power addition
Octal to decimal	Power addition
Decimal to binary	Radix arithmetic
Decimal to octal	Radix arithmetic
Binary to octal	Substitution
Octal to binary	Substitution
General Rules	
$r_i > r_f$: use radix arithmetic, substitution	
$r_i < r_f$: use power addition, substitution	
r_i Radix of initial system	
r_f Radix of final system	

Example 3: Octal to Decimal (Integer)

$$324_8 = 3(8^2)+2(8^1)+4(8^0)$$
$$= 3(64)+2(8)\ +4(1)$$
$$= 192\ +16\ \ +4$$
$$= 212_{10}$$

Example 4: Octal to Decimal (Fractional)

$$.44_8 = 4(8^{-1})+4(8^{-2})$$
$$= 4/8\ \ +4/64$$
$$= 36/64$$
$$= 9/16_{10}$$

RADIX ARITHMETIC

To convert a whole number from r_i to r_f ($r_i > r_f$):

1. Divide r_i by r_f using r_i arithmetic
2. The remainder is the lowest order bit in the new expression
3. Divide the integral part from the previous operation by r_f
4. The remainder is the next higher order bit in the new expression
5. The process continues until the division produces only a remainder which will be the highest order bit in the r_f expression.

To convert a fractional number from r_i to r_f:

1. Multiply r_i by r_f using r_i arithmetic
2. The integral part is the highest order bit in the new expression
3. Multiply the fractional part from the previous operation by r_f
4. The integral part is the next lower order bit in the new expression
5. The process continues until sufficient precision is achieved or the process terminates

Example 1: Decimal to Binary (Integer)

45 ÷ 2 = 22 remainder 1; record 1

22 ÷ 2 = 11 remainder 0; record 0

11 ÷ 2 = 5 remainder 1; record 1

5 ÷ 2 = 2 remainder 1; record 1

2 ÷ 2 = 1 remainder 0; record 0

1 ÷ 2 = 0 remainder 1; record 1

101101

Thus: $45_{10} = 101101_2$

Example 2: Decimal to Binary (Fractional)

.25 x 2 = 0.5; record 0

.5 x 2 = 1.0; record 1

.0 x 2 = 0.0; record 0

.010

Thus: $.25_{10} = .010_2$

Example 3: Decimal to Octal (Integer)

273 ÷ 8 = 34 remainder 1; record 1

34 ÷ 8 = 4 remainder 2; record 2

4 ÷ 8 = 0 remainder 4; record 4

421

Thus: $273_{10} = 421_8$

Example 4: Decimal to Octal (Fractional)

.55 x 8 = 4.4; record 4

.4 x 8 = 3.2; record 3

.2 x 8 = 1.6; record 1

.431...

Thus: $.55_{10} = .431..._8$

SUBSTITUTION

This method permits easy conversion between octal and binary representations of a number. If a number in binary notation is partitioned into triplets to the right and left of the binary point, each triplet may be converted into an octal digit. Similarly, each octal digit may be converted into a triplet of binary digits.

Example 1: Binary to Octal

Binary = 110 000 . 001 010

Octal = 6 0 . 1 2

Example 2: Octal to Binary

Octal = 6 5 0 . 2 2 7

Binary = 110 101 000 . 010 010 111

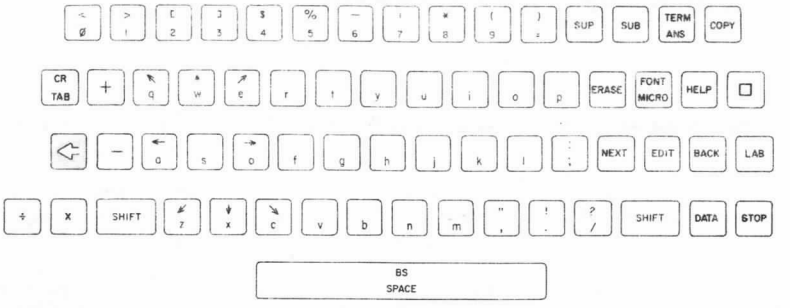
TERMINAL KEYBOARDS

The PLATO terminal consists of 64 keys similar to a standard keyboard, with some additions. The additions are for the function keys and some special characters as in A.

The lowercase keyboard (B) contains the lowercase alphabetic keys, some of the function keys, numerals, and special characters.

The uppercase keyboard (C) is accessed by holding down the SHIFT key while pressing the desired key. The uppercase keyboard contains the capital letters, the remaining function keys, and special characters.

In addition, there are hidden characters. These are accessed by pressing and releasing the ACCESS key and then pressing the desired key. There are both lowercase and uppercase access characters, shown in D and E, respectively.

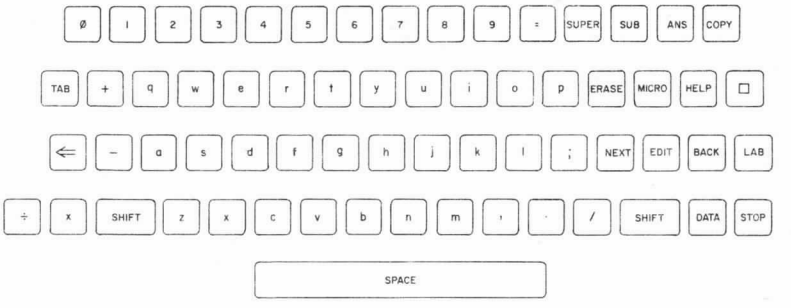


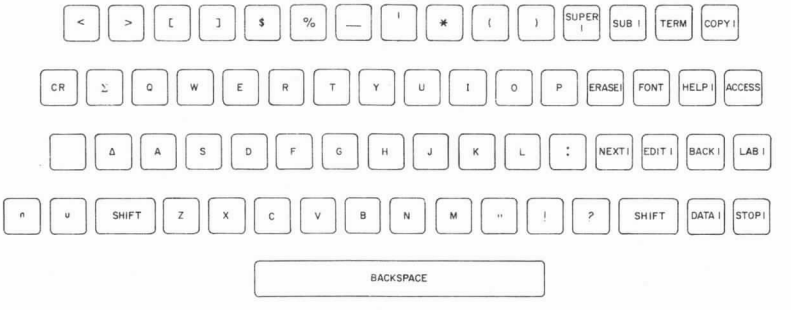
A

ACTUAL KEYBOARD

B

LOWERCASE KEYBOARD



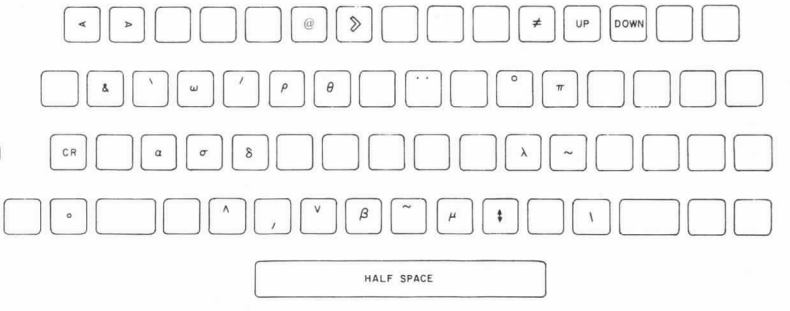


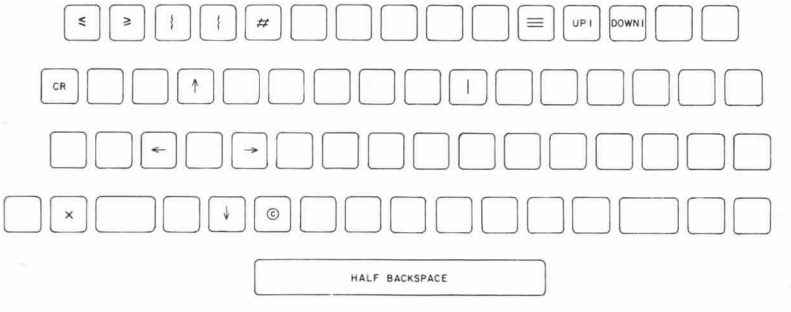
C

UPPERCASE KEYBOARD

D

LOWERCASE ACCESS KEYBOARD





E

UPPERCASE ACCESS KEYBOARD

INDEX

COMMENT SHEET

MANUAL TITLE: CDC PLATO Author Language Reference Manual

PUBLICATION NO.: 97405100 **REVISION:** C

NAME:_____

COMPANY:_____

STREET ADDRESS:_____

CITY:_____ **STATE:**_____ **ZIP CODE:**_____

This form is not intended to be used as an order blank. Control Data Corporation welcomes your evaluation of this manual. Please indicate any errors, suggested additions or deletions, or general comments below (please include page number references).

CUT ALONG LINE

AA3419 REV. 4/79 PRINTED IN U.S.A.

NO POSTAGE STAMP NECESSARY IF MAILED IN U.S.A.

FOLD ON DOTTED LINES AND STAPLE

STAPLE

STAPLE

FOLD

FOLD

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 8241 MINNEAPOLIS, MINN.

POSTAGE WILL BE PAID BY

CONTROL DATA CORPORATION

Publications and Graphics Division
ARH219
4201 North Lexington Avenue
Saint Paul, Minnesota 55112

CUT ALONG LINE

FOLD

FOLD

CDC®

PLATO

CONTROL DATA CORPORATION

CONTROL DATA EDUCATION COMPANY, P.O. BOX O, MINNEAPOLIS, MINNESOTA 55440
SALES OFFICES AND SERVICE CENTERS IN MAJOR CITIES THROUGHOUT THE WORLD

LITHO IN U.S.A.