80-004

# A RENOVATION OF THE UNC CAI SYSTEM

by

Paul C. Clements

A Thesis submitted to the faculty of the University of North Carolina at Chapel Hill in partial fulfillment of the requirements for the degree of Master of Science in the Department of Computer Science.

Chapel Hill

1980

Approved by:

Peter Calingaert

Adviser

David L. Parnas

Reader

Stephen F. Weiss

Reader

PAUL CHARLES CLEMENTS. A Renovation of the UNC CAI System (Under the direction of Peter Calingaert.)

This thesis describes work undertaken to modify the Computer-Assisted Instruction system at the University of North Carolina at Chapel Hill. The modifications include enhanced internal documentation of the PL/I source code, and restructuring of local code sections to eliminate confusing branches. The architecture of the system has been changed by deleting the lightpen feature, replacing its functions with keyboard commands, color-coding of system information and warning messages, and the implementation in the author language of integer subtraction. Suggestions for future work are offered. The new System Programmer's Manual is attached as an appendix to this document.

TABLE OF CONTENTS

The host operating environment for the CAI system was developed at roughly the same time by Gary Schultz [5]. The CAI system runs under the Chapel Hill Alphanumeric Terminal (CHAT) system, a single-region resident time-sharing monitor that runs under the OS/360 MVT operating system at UNC. CHAT allows the interaction of multiple application programs with CRT display stations, and provides input/output programming support for that interaction. CAI is only one subtask (application program) that runs in the CHAT environment; others include a graphics interface package, a text editor, and a HASP monitor program. CHAT was designed and built with one kind of display terminal in mind: a Computer Communications Incorporated CC-30 Communications Station. This is a four-color CRT with a lightpen, and a forty-character by twenty-line screen. In addition, each terminal has a program-controlled random-access slide projector. CHAT software includes support for controlling the projector and lightpen as well as reading and writing the CRT screen.

Mudge paid close attention to the working environment of his users, and carefully designed special CAI workstations. Each station consists of a CC-30 CRT and its keyboard, lightpen, slide projector, and a desk-type writing surface. All of this is enclosed in an acoustically treated cubicle approximately six feet square.

The CAI system was put into test production in 1972, and was used to teach beginning PL/C to Computer Science and Library Science students of UNC until the fall of 1978. The original course material was written by Dr. Frederick P. Brooks, Jr. [2], and has served as the curriculum of the system, except for minor modification, throughout the entire life of CAI here.

During its creation and testing, the CAI system enjoyed the services of a systems programming team, whose responsibility it was to isolate and fix bugs in the new system; a group of course proctors, who oversaw the instruction process, and smoothed the way for the student users of the system; and a corps of students in several sections of programming courses, who learned from the system, and evaluated its ease of use and effectiveness for Mudge and his colleagues. CAI was a major effort of the Department.

Any large system will show signs of age as time passes, and the CAI system has been no exception. By early 1979, it was only a minor responsibility of the Department's software manager. No one used it for instructional work or research at all. The team originally responsible for its creation had long since graduated, and the workstations fell into disrepair.

# Chapter 2

# DEFICIENCIES IN THE CAI SYSTEM

Why did the CAI system fall from prominence into obscurity? For one thing, the people who built it left. For another, it was always intended to be a research tool above all; any production use it found would merely be a bonus. Moreover, one or two of the design decisions seem to have created needless restrictions in the handling of the lesson files. More important, I believe, is that Mudge's full design was never implemented, and the omission of several features from the final version of the system rendered it awkward and impractical to use. And finally, since Mudge chose to implement his own author language, rather than use a general-purpose language already existent, he automatically incurred the expense and overhead of writing what is essentially an interactive compiler for that language, and imbedding that compiler into the system. Now, writing a bad compiler is a sizable task; writing a good one is an enormous task, and in my judgment the DIAL translator lies somewhere between the two extremes. In any case, it does not give its user the kind of support to which many have become accustomed.

The unimplemented features of the CAI system were primarily features of the author language. The control structures of DIAL are basically those found in PL/I, with a few major extensions. Therefore, the DIAL architecture defines the IF-THEN, IF-THEN-ELSE, and DO-WHILE statements, all of which are semantically defined as in PL/I. None of these three was actually implemented in the system. Consider for a moment their use in a response-analysis situation, by examining the following possible uses:

```
IF answer ¬= expected_answer
     THEN SHOW 'Wrong answer; try again.';
     ELSE SHOW 'Your answer is correct.';
```

Or:

```
IF answer = wrong_answer_1
     THEN SHOW hint_1;
ELSE IF answer = wrong_answer_2
     THEN SHOW hint_2;
ELSE IF answer = wrong_answer_3
     THEN SHOW hint_3;
ELSE SHOW big_hint;
```

Or:

```
        (Read answer)
        DO WHILE (answer ¬= expected_answer);
             SHOW 'Wrong answer; try again.';
             (Read answer)
        END;
```

Or, finally:

```
        I = 0;
        (Read answer)
        DO WHILE (I<N & answer ¬= expected_answer);
             SHOW 'Not right yet; try again.';
             I = I + 1;
             (Read answer)
        END;
```

DIAL's control structure for answer analysis is the MATCH statement. Syntactically, it is of the form

MATCH expected_answer, label

and if the student response is equivalent to "expected_answer", then a branch takes place to "label". Implicit in the first MATCH statement following a SHOW statement is the command to read an input line from the display screen. Similarly, there is the UNREC statement:

UNREC label1, label2, label3, ...

where the i-th unrecognized response to the most recently executed SHOW statement will cause a branch to the i-th label in the UNREC label list. An unrecognized response is one that has not been specified in the MATCH statement(s) following the controlling SHOW statement. The UNREC statement corresponds to the OTHERWISE statement of the PL/I SELECT construction. It relieves an author from having to test (via MATCH statements) for every possible student input. Rather, it allows the author to specify an action to be taken when the student's response falls through all of the associated MATCH statements.

There are more complex forms of both the MATCH and UNREC statements, offering more flexible actions, but the fact that both rely on explicit branching to do what could be accomplished without it seems to me to be unwieldy. As a result, DIAL programs must rely heavily on branching, and this offers little chance for structured author programming, which in turn makes course writing all the more time-consuming and awkward.

For instance, consider how the first example above would be formulated with only the incomplete DIAL available:

```
READ:  MATCH expected_answer, RIGHT;
       SHOW 'Wrong answer; try again.';
       GOTO DONE;
RIGHT: SHOW 'Your answer is correct.';
DONE:   /* end of if-then-else simulation */
```

Or, for the third example:

```
        READ:  MATCH expected_answer, OUT_OF_LOOP;
               SHOW 'Wrong answer; try again.';
               GOTO READ;
        OUT_OF_LOOP: /* end of do-while simulation */
```

The case statement simulated in the second example above would offer an even more dramatic display of labels and branching. As for the fourth example, which branches conditionally after testing the value of a counter variable, it is not clear to me that in the limited version of DIAL that would be possible at all.

It is for this reason that I believe CAI has fallen dormant; no one has been willing to compose new lesson material for the system, and as a result, the PL/C course originally written for the system is now embarrassingly outdated. It was written in conjunction with a textbook no longer used, and does not reflect the doctrines of structured programming espoused by the books now in vogue.

Another major reason contributing to the overall problem is the hardware involved in the CHAT system. It is all showing its age. By today's standards, the CC-30 terminals are archaic. The display is hard on the eyes, the screen is quite small; and as time passes, they are becoming less and less mechanically reliable. Spare parts are no longer available. In fact, each component of each workstation is extremely susceptible to hardware failure. Most of the lightpens were long ago disconnected; the slide projectors and their controllers are all moth-balled.

Since the CAI system depends so heavily on CHAT, and since the CAI hardware supported by CHAT seems to be on its last leg, the project could very well have ended here. Any work on modifying and rebuilding the CAI system would have been pointless, except that under development at the time of my work was a new version of CHAT that would allow its application programs to run either on the CC-30 terminals or on the Hewlett-Packard 2621 or 2645 terminals recently acquired by the Department of Computer Science. With this new system over the horizon, a future for CAI no longer seemed implausible.

Before that work could be completed, however, the age of the system once again proved an impediment. Recall that the lessons in the CAI system were no longer useful because they were written before structured programming was generally espoused, let alone practiced. Naturally, the implementation of the system itself predates the creation of those lessons. The original code contained dozens of indiscriminate (and unnecessary) branches; following the flow of control in many of the code sections was quite difficult. The internal comments were sketchy, and not generally helpful.

Little attention was paid to indentation and layout of the statements. The file structures are quite complex, and nowhere were they explained in detail. A System Programmer's Manual did exist, but was only about half completed. Nowhere did there exist an accounting of precisely which elements of the architecture were actually implemented. More of a hindrance than any of this, however, was the fact that the seventeen modules (PL/I external procedures) of the system overlapped in terms of task and effect. Most communicated with each other through a large number of external variables, and only a few were responsible for discrete, well-defined tasks. Modifying such a system is a risk at best; at worst, it may spawn a disaster. Because each module makes assumptions about what changes others have (or have not) made to the files or data structures, a small change in one may have far-reaching and quite unforeseen side effects in any number of others. The CAI system serves as a shining tribute-by-counterexample to the doctrines of structured programming.

Chapter 3

# THE MODIFICATIONS UNDERTAKEN

My work on the CAI system was divided into two categories: architectural modifications which will be visible to the production user of the system, and modifications which will be visible only to a CAI system programmer.

Included in the latter category are the enhancement of internal documentation, restructuring of local code sections, and the completion of the System Programmer's Manual.

The internal documentation that I supplied to the modules consisted of enhanced header paragraphs, as well as in-line comments. The header documentation for each module followed the same format, and consisted of the following information:

1) Function. This is a general prose definition of what the particular module is supposed to do. Because of the nebulous nature of each module's task, this information is necessarily imprecise. It is meant to give a programmer who is searching for the cause of a problem a good idea of which module to pay special attention to. In a few cases, I was able to use descriptions that already existed, provided by the original CAI programming team.

2) Parameters. This is a list of the parameters for this module, with a brief description of the information carried by each. Included is whether the parameter is given (that is, its value is set externally, and only used here), returned (its value is set in this module to be used externally), or both.

3) Input. This is a specific description of what input this module expects. By "input", I mean input from the user or input from the CAI system files, not input through parameters from other modules. There is an exception to this distinction, however. The CAI system has a module called FILEIO, the purpose of which is to read and write one of the system files; a parameter is the information to be transmitted. A call to this procedure has the same effect as performing file input/output; therefore, in the interest of providing more complete information, I have used the "Input" paragraph in many cases to describe such calls. However, I was careful to make it clear in each case that I was in fact describing the effect of a call to FILEIO, and not "real" input.

4) Output. Similarly, this describes what output this module generates for the user, or onto the files.

5) Functions called. This is a list of all of the procedures called by this module, with a brief decription of the task of each. Internal procedures are differentiated from external procedures, but both are described.

6) References. For those modules where external sources of information may be helpful, such sources are cited.
7) Compilation information. Designed to help bring about a successful compilation and linking of this module, this section gives a complete transcript of the necessary JCL, including a list of compiler options to use.
8) Use of Branching in this Procedure. This is a list of all the relevant statement labels to be found in the procedure, with an explanation of when and how each one is used in branching.
9) Other information. Some modules may call for special documentation; for instance, in the main procedure, I included a diagram of the overall system call structure, and a paragraph explaining the changes made to the architecture and documentation of the 1980 system. In the DIAL compiler routine, the productions of the grammar are catalogued, and in the AUTHOR routine, there is a list of all of the valid author commands.

The heading descriptions for internal procedures followed the same conventions, with one extension. I included a list of all of the externally-defined variables used locally by each internal procedure. Of course, PL/I's block structure makes it perfectly legal to use (without declaration) any variable defined in a containing block, but imprudent use of such variables can very easily give rise to side effects, and tends to obscure the precise nature of the role of the internal procedure.

I followed no guide for providing in-line comments for the code, other than a loose and subjective one. First, any major segment of code which accomplished a fairly discrete task warranted a paragraph of commentary. If one was lacking, I provided it. I decided quite subjectively which were "major segments" and which were not. Second, any other statement or segment whose purpose was not clearly explained, and that seemed important enough to try to understand, received an explanatory comment. Again, the decisions here were subjective. I tried not to be terse.

Often, I was forced to understand a group of statements in order to effect my architectural modifications, and then it was a simple matter to record what I had learned about a certain section. By the same token, those parts of the system that clearly did not apply to my work may remain somewhat under-documented.

While my modified system actually comprises roughly two percent fewer executable PL/I statements than its predecessor, the source listings occupy some fifty more pages, an increase of about eighteen percent. I attribute most of that to new internal documentation. Naturally, that does not speak to the quality of the added information, but should at least offer evidence of my good intentions.

An adjunct to the internal documentation was the work I did to to improve the visual quality of the code. The rules applied were quite simple. They included uniformly indenting the bodies of loops and compound statements (including any contained comments describing parts therein), aligning each END with its associated DO, and indenting each code section beneath the comment describing it. The resulting physical structure of the code is meant to aid the understanding of its logical structure.

My restructuring of the code was fairly modest in nature, but extremely widespread. Like the internal comments, this was as much a tool for my own understanding of the system as anything else. My first strategy was to delete every GOTO statement that I could; my tactic was to identify and eliminate every statement label within each procedure. The labels that were branch targets served to point to the branches themselves. The other labels served no useful purpose, and were summarily erased. Any mnemonic significance they offered was replaced with a comment. Most of the branches that I was able to delete were used to simulate large alternative-selection (IF-THEN-ELSE) statements, by branching around a section if some condition were not true, and falling through it otherwise. I replaced those with IF-THEN-ELSE statements. Others were used to build loops (I replaced those with loop statements), and still others simulated procedures (which I replaced with real procedures).

After all of the dust has cleared, it is evident that the CAI system still suffers from a plethora of GOTO statements. Some are used merely to terminate loops, and I judged those to be innocuous; others are simply too deeply embedded in the logic to be easily replaced. For the latter group, I adopted another strategy: if it can't be eliminated, then it must be explained. Therefore, as explained above, the documentation for each module includes a paragraph entitled "Use of Branching in this Procedure", which lists every statement label not used just as a loop escape, and explains how and when it is used.

Another modification to the system code was not so modest in nature. The modules of the CAI system communicate with each other through a large aggregation of external variables. The system was originally built so that all of those variables were stored in a single large PL/I structure, and then any procedure which referenced any of those variables was passed the entire structure as a parameter [4]. This was done because the load module must be reentrant; there may be several users at once, and each invocation must have its own copy of the external variables. The super-structure approach worked well, but had drawbacks in terms of clarity. For instance, many of the modules use only two or three of the external variables, but had to declare the entire set (taking almost two pages in the listing). W. James Sneeringer [6] has proposed a more reasonable solution. By declaring each such variable with the attribute CONTROLLED EXTERNAL and then allocating them all in the main procedure, only those variables actually needed by a procedure need be declared there. Reentrancy is preserved, and Sneeringer argues that the compiled code obtained by this method in fact produces less overhead than that of the structure parameter approach. Therefore, I set out to replace the structure in each procedure with only those variables actually used, and I believe that the gains in clarity are well worth the effort. Even though the modules still communicate through external variables, each module at least contains a list explicitly defining the only variables it may change. In addition, I discovered that half of the fifty external variables did not in fact need to be known to more than one external procedure, and they were quickly demoted to local variables.

The new System Programmer's Manual is included as an appendix to this document. The sections that I supplied are listed in an appendix to the manual. It should be noted that much of the information added to the manual came directly from the internal documentation of the system. For instance, the module descriptions are from each procedure's "Function" paragraph in the header comment.

Although time has not permitted removing all of the deficiencies pointed out in Chapter 2, the architectural modifications I have made include extraction of the lightpen from the system, color-coding and re-wording of the system messages to the user, and the implementation of five new author command features.

The lightpen facility was taken from the system for three reasons. The first is that, for all practical purposes, there are none left; nearly all are out of service. The second reason is that the Hewlett-Packard terminals do not feature lightpens; hence neither will the envisioned new version of the CAI system. Finally, the deletion of the lightpen facility has served to reduce the amount of run-time memory required by the system, and to streamline the system, in the sense that input can now only be done via the terminal keyboard. This, I believe, is important from a human factors standpoint. Neither the student nor the author need worry about using the lightpen, which seems to me to be a rather unnatural input mechanism. It was the deletion of the lightpen that allowed my system to use slightly fewer PL/I statements than its predecessor, while actually doing more work.

The system diagnostic messages were modified in two ways. First, I attempted to make them more informative and more congenial. For instance, the sign-on greeting "UNC CAI SYSTEM" (displayed in bland blue) was replaced with "Welcome to the UNC CAI System" (shown in a more congenial green). I added some messages. For instance, some commands in the system ask an author whether he really wants something done; if he fails to confirm it, the command is nullified. The old system responds merely by waiting for another command. My version acknowledges the cancellation, and displays (in green) "Command canceled." Finally, I paid special attention the color of the displayed message. Previously, most diagnostics appeared in blue. I adopted a more varied scheme. If the purpose of the message is to confirm or inform, it appears in green. Special words within are highlighted by appearing in yellow. For instance, one message is "The COMPILE switch is now OFF." The message is green, except for "OFF", which is yellow. If the purpose of the message is to warn or report failure, it appears in yellow. For instance, "You already have a lesson by that name" appears in yellow when an author tries to create a lesson with the same name as another. When the system halts in case of unrecoverable error, the error code appears in red. Basically, the philosophy is this: when the message reports something the user should expect, it appears in green; when it reports something he should be concerned about, it appears in yellow; when it announces abnormal termination, it appears in red. Blue, then, is left to be the color of the text that the user himself enters.

Of the five new author features implemented, three merely replace functions lost when the lightpen was removed from the system; two are truly new.

The three replacement functions are "CLEAR", which allows an author to ask for a blank screen so he may enter a DIAL statement that was too long to be entered where the cursor was previously positioned; "EDIT", which allows an author to make changes to DIAL text by merely typing in the changes, rather than re-entering the entire statement; and "CSW", which controls the compile switch, the bit which decides whether or not to re-compile an author's entire lesson every time the author makes a change to it.

The two new features are integer subtraction and the "RESEQUENCE" command. Just as the IF-THEN and DO-WHILE statements appeared in the DIAL grammar but were never actually implemented, neither was most of the arithmetic expression evaluation mechanism. Since integer addition was already in place, it was a simple matter to mirror the process and implement subtraction.

The "RESEQUENCE" command allows an author to renumber the statements in a lesson according to a given starting value and increment. Implementing that command brought about an unexpected change to the system architecture elsewhere. It turned out that renumbering a lesson was a decidedly nontrivial task if, as was the case, each statement's number field was not of uniform length. For instance, "1 DCL J INTEGER" and "0001 DCL J INTEGER" are both equally valid representations of statement number one of some DIAL lesson. However, the renumbering became quite straightforward when one assumed numeric fields of constant length. Because the system limits statement numbers to positive integers not exceeding 9999, I adopted the policy of four-digit number fields for all statements. In addition to the gain in ease of renumbering, this policy has a valuable human engineering effect. It aids an author in preparing more readable DIAL source text by forcing all statements to begin in the same column, independent of the statement number. Thus, systematic alignment and indentation become easier. This policy is certainly not without precedent; TSO, for example, employs it. One may argue that I made this decision for the wrong reason (ease of implementation), but I submit that the final result is in fact a better architecture.

In fact, I would hope that my modifying work has produced exactly that: a better architecture throughout. I am confident that it has produced a better implementation in terms of clarity, efficiency, and ease of future change. The CAI system is still not the invaluable aid hoped for by any of its creators; it is, however, a viable tool which I have tried to make easier to use.

# Chapter 4

# SUGGESTIONS FOR FUTURE DEVELOPMENT

The event that may breathe new life into the CAI system will also precipitate the next major requirement for change. When CHAT is able to run on Hewlett-Packard terminals, a major overhaul of the CAI system will be in order. Two strategies are possible.

The first is to adapt the system to the new terminals by applying local fixes. Even though terminal input and output are technically done by only one procedure in the system, almost a dozen procedures make the assumption that the screen size of the terminal is that of a CC-30, which is quite different from that of the Hewlett-Packards. The most common example of this is creating a multi-line display message by inserting the appropriate number of blanks, rather than a new-line command, at the end of each screen line. Some modules display entire screens at one time, and the formatting for such a display depends heavily on the screen size. Rooting out all such assumptions would require careful examination of nearly all of the procedures constituting the CAI system, and a tedious modification of many.

The second strategy for adapting to the new terminals requires more initial work, but would be much more satisfying in the long run. The Hewlett-Packard terminals are eminently more sophisticated than the CC-30s. They offer features such as on-screen editing, programmable function keys, screen paging or scrolling, tabs, and blinking display; and some can read and write a cassette tape. Most importantly, all of the advanced features are programmable; that is, they can be initiated and controlled by special control characters received by the terminal from a program. The new CHAT system will take advantage of all of these features, many of which could add an exciting power to the output capability of the CAI system.

Consider the following excerpt from a hypothetical student session. The system displays a screen of new lesson material by issuing the command for the terminal to read and display one of many files on its builtin cassette tape. The student presses the RETURN key to signal that he has read the information. The system commands the terminal to move the material off-screen, and lock it into a page of its memory. The system displays a question, and the student responds. The system instructs the terminal to record the response on its other cassette tape, for later analysis. On-line response analysis takes place as usual, and the CAI system discovers that the student's answer is wrong. It orders the terminal to page the lesson material back onto the screen; this time, however, key words and lines of the screen are highlighted with an inverse video or a flashing display. The student is asked the question again, and this time he responds correctly. The system displays the congratulatory message, rings the terminal's bell, and goes on to present the next topic.

Of course, to realize such a situation would take an inordinate amount of work. But such powerful capabilities present a broad range of possible enhancements to the CAI system. It seems to me that a reasonable first step would be to build a CAI system so that only one module truly knew of the terminal characteristics. It would be responsible for fitting given text strings to the available screen size. Perhaps the calling procedures would adopt command parameters similar to what might be found in an elementary text-formatter: "center", "underscore", "newline", "table", etc.

After that, new DIAL commands may be created. For instance, one might visualize a feature that would allow an author to define a screen-full of text, specifying which words or lines should be highlighted on subsequent displays, and another command that would allow that text to be written onto a cassette tape, and referred to symbolically thereafter (much like the SLIDE data type currently implemented in DIAL). Then, there might be a SHOW_TAPE command, and a SHOW_HIGHLIGHTED_TAPE command. The possibilities are profuse.

A second major improvement to the CAI system would be the implementation of an on-line file maintenance subsystem. This was envisioned by the original team of CAI system programmers, and still seems a good idea. There are three major files used by the CAI system. One contains information about all authors known to the system, another about all students known to the system, and the third contains everything else (for example, the source and object code for all the lessons; the list of lessons constituting each course; the status of each student currently taking a course, and so forth). To add or delete or change records in these files, single-purpose batch programs must be submitted. A more elegant solution would be to invoke an on-line CHAT program which would be able accomplish any of the tasks now handled by the batch programs. The list would include adding an author or a student to the system; deleting an author or a student from the system; deleting all students enrolled in a particular course from the system; reporting on a particular student's progress; displaying a directory of all students or authors currently in the system; removing lessons from a course; creating or removing a course; and displaying formatted information about the contents or available space on the files. The directory or information requests could offer a print option, causing a print job to be submitted to the batch job stream.

Implementing such a system would not be very difficult, as programs already exist which perform each transaction listed above. The print option has been implemented in another CHAT application program, and so that logic is also available. Finally, such a system could employ some of the modules of the on-line CAI system, such as the main driver, the input/output interface module, and the module which updates the author directory, with very little (if any) modification.

## BIBLIOGRAPHY

1. Barrier, O. Jack, Clements, Paul C., and Mudge, J. Craig. UNC CAI System Programmer's Manual. University of North Carolina at Chapel Hill. 1980.

2. Brooks, Frederick P. Lessons on PLC programming, written for the UNC CAI system. 1973.

3. Mudge, J. Craig. Human Factors in the Design of a Computer-assisted Instruction System. Ph.D. Dissertation. University of North Carolina at Chapel Hill. 1973.

4. Mudge, J. Craig. On Writing Reentrant Programs in PL/I. SACM Newsletter--a Publication of the University of North Carolina Student Chapter of the Association of Computing Machinery, Chapel Hill (November 1971), 2-3.

5. Schultz, Gary D. The CHAT System: An OS/360 MVT Time-Sharing Subsystem for Displays and Teletype. Master's Thesis. University of North Carolina at Chapel Hill. 1973.

6. Sneeringer, James. More on Writing Reentrant Programs in PL/I. SACM Newsletter--a Publication of the University of North Carolina Student Chapter of the Association of Computing Machinery, Chapel Hill (December, 1971), 5-7.

# Appendix A

# CAI SYSTEM PROGRAMMER'S MANUAL

## PREFACE: USING THIS MANUAL

This manual is intended for anyone engaging in modification, documentation, enhancement, or maintenance of the UNC CAI System. A user of this manual should already have performed the following:

1. Read the CAI Operations Manual, written by Mitchell J. Bassman; this gives an overall view of the files used by CAI and the utility programs which exist to maintain them.

2. Read chapters 4 and 5 of <u>Human Factors in the Design of a Computer-Assisted Instruction System</u>, by J. Craig Mudge. These chapters explain the architecture of the DIAL programming language, and the operational environment of the CAI system.

3. Become familiar with CHAT and the CC-30 terminals.

4. Become familiar with the CAI system. Using the file procedures listed in the Operations Manual, enter yourself as a student into the system, and then go through some of the course material in student mode. Enter yourself as an author in the system, and write a small instructional lesson in DIAL. Explore the author commands. Try executing your lesson. Experiment with color displays, the DIAL pattern matching facility, the SQZ and CASE system variables.

**CONTENTS**

# Chapter 1

# OVERVIEW OF THE CAI SYSTEM

The CAI System can be thought of as consisting of two constituent sets of PL/I procedures. One set makes up the on-line system, that is, that system which appears to the user sitting down at a CC-30 terminal and signing on to a CAI program under CHAT. The other routines are off-line; executed by batch jobs, they perform file maintenance tasks and the like[1]. There are three on-line systems of CAI. They are invoked by signing on to CHAT and entering the appropriate program name.

The first is called COURSE; it is an on-line program for displaying and removing lessons from a CAI course. This is important, because a course cannot be deleted from the CAI System until all of its lessons have been removed.

The second on-line program is called CAIOLFI (for "CAI On-Line File Inquiry") and provides information about a given student's file status. There is currently no external documentation about this program, but it is well-engineered and its use should be self-explanatory. Unfortunately, it depends on user input through a lightpen, and so its use is restricted to those CC-30 terminals with a functioning lightpen.

The third on-line system is that which is described in Mudge's dissertation, and exists in two forms: CAI or CAIAUTH. The former is for student use; it presents lesson material, analyzes responses, and logs progress. The latter does all of that as well, but it also supports the author environment of the system. The source handler, compiler, and code generator for DIAL statements are contained here, as is the large module which serves as the interface between an author and the system.

Over the years, some specialized terminology has sprung up, which bears explaining. The term "CAISYSTEM" refers to all of the production programs having to do with CAI at UNC;

--------------------

[1] There is in fact a small set of hybrid routines that may be executed either from a batch program or called by an on-line program; these will be pointed out and described in Section 9.2.

this includes on-line and off-line routines. On the other hand, the "CAI System" typically refers to either the CAI or CAIAUTH on-line programs. This manual deals primarily with the CAI System. The other on-line programs and the rest of CAISYSTEM (i.e., the off-line programs) are described herein, but only briefly.

# Chapter 2

# THE EXTERNAL PROCEDURES CONSTITUTING THE CAI SYSTEM

There are seventeen distinct external procedures which, when linked together, form the on-line programs CAI or CAIAUTH. This chapter contains a description of each one. Note that the text comes directly from the header paragraphs for each module. Information about the logic of each procedure may be found by consulting the comments in the source code listing of that procedure. The overall function will be emphasized herein, not the actual implementation strategy employed to achieve that function.

## 2.1 <u>CAIMAIN</u>

CAIMAIN is the main procedure of the on-line CAI System. It contains the ON ERROR on-unit which allows the system to display fatal-error diagnostics before it dies. It also calls SNONOFF, the sign-on/off routine, to see whether an author or a student is the user. On the basis of that call, CAIMAIN either calls AUTHOR or EXECTOR. Finally, another call to SNONOFF occurs for sign-off, and the task terminates.

## 2.2 <u>SNONOFF</u>

This routine performs user sign-on and sign-off functions, depending on the value of a given bit parameter.

If the function is to sign ON a user, SNONOFF reads the user id from the screen, and accesses the AUTHREC and STUREC files to see if it belongs to a student or an author (respectively), or neither.

A sign-on attempt may fail if (1) two incorrectly-formatted ids are given; (2) two unknown ids are given; or (3) an attempt to enqueue a required resource fails.

If the sign-on is successful and the user is a student, then either a "resume" or a "recover" sequence takes place, according to the RECOVNEEDED bit in the student's SREC structure (read from file STUREC). A "resume" is what happens when the student's last session ended normally. The system restores his run-time environment by reading it into core from his "resume area" on disk, where it was stored when he signed off from his last session. This time, therefore, he will start in a lesson about where he left off last time. A "recover" is what happens when the student's last session ended abnormally (e.g., the system crashed before sign-off). For more about resuming/recovering and the file I/O involved, see the declarations for MAINFILE, SCB, and SREC; also, the in-line comments of SNONOFF attempt to give a good idea of the protocol involved.

If the user is an author, SNONOFF calls SYS_SW to make sure that this system was generated with complete author facilities. If not, the sign-on fails.

The record of the student or author (SREC or AREC) is updated, written, and freed. The structures SREC and AREC (and therefore the ISAM files STUREC and AUTHREC) will not be used again until signoff. SNONOFF then returns to CAI-MAIN, informing it whether the user is a student or an author.

If the function is to sign off a user, then SNONOFF recalls whether the user is a student or an author. If an author, his record is updated with data from this session, and written back onto AUTHREC. If a student, his SCB is written onto the SCB_PART of his SREC, for use in his next session's resume sequence; he will begin work next time at a place near where he left off this time. His SREC is then updated, and re-written onto file STUREC.

## 2.3 <u>FULLCAI</u>

The module FULLCAI consists solely of the procedure SYS_SW. All this procedure does is return a bit value of "false", to be tested by SNONOFF. When an author signs on to the system, SNONOFF calls the procedure SYS_SW to see whether this is the version of the system that supports author facilities. When that version of the system is generated, this module is included. When the student-only version of the system is generated, the STUONLY module is included instead; its version of SYS_SW returns a "true" bit value, and informs SNONOFF that only students may use that system.

See Section 3.1 of this manual for further information.

## 2.4 STUONLY

This module, like FULLCAI, consists only of a version of the procedure SYS_SW that returns a "true" bit value, informing SNONOFF that the system that has been generated does not include the author facilities. This allows SNONOFF to prevent an author from signing on to a student-only version of the system.

See Section 3.2 of this manual for further information.

## 2.5 AUTHOR

AUTHOR is the prime interface between an author and the CAI System. It is called as soon as CAIMAIN learns from SNONOFF that it is an author (not a student) who has signed on. AUTHOR converses with its user via the command language and line-numbering mechanism. It is the facility that allows an author to enter DIAL statements and have them compiled, by invoking the syntax-directed compiler for each statement received.

This procedure, not COMPLER, builds the object code and source code files for a lesson. COMPLER compiles one statement at a time; it merely returns (in the structure TEMP) the object code and literals for that one statement. AUTHOR takes the contents of TEMP and adds them to the instruction and literal files for the lesson being worked on. AUTHOR is forever updating the lesson's Lesson Control Block (LCB), to keep track of, for example, the page translation tables and CAIFILES region allocation.

Also, AUTHOR sets up and controls calls to EXECTOR, in response to an author's request to view the execution of one of his lessons.

## 2.6 ACBPROC

This procedure is called by AUTHOR to perform processing on the Author Control Block (ACB), held on Region 3 of CAIFILES. Given a lesson name, it performs according to the requested function parameter. Its functions include searching the ACB directory to find out where the given lesson's Lesson Control Block (LCB) is stored; removing the given lesson from the ACB, returning its LCB location to AUTHOR; adding a lesson, and returning to AUTHOR the region number allocated for its LCB; returning a formatted list of all of the lessons "owned" by the author now signed on; and searching the ACB directory to see whether a given author id number occurs there.

The ACB is brought into core (and hence, enqueued) for the duration of the procedure. Error conditions which ACBPROC discovers are coded and returned in a parameter.

## 2.7 COMPLER

COMPLER is called by AUTHOR with a single DIAL source statement to parse. The main compilation loop receives tokens from SCAN, the internal lexical scanner. COMPLER in turn calls CODEGEN just before a reduction takes place to emit object code instructions into the TEMP structure. Upon return to AUTHOR, those instructions and literals are then appended to the appropriate system files. The code generation phase, then, is external to COMPLER; COMPLER is essentially just a parser.

COMPLER attempts no error recovery; as soon as it discovers an error, it returns to AUTHOR with a diagnostic message stored in TXT, and LP holding the place in the statement which COMPLER thinks is in error. The emphasis was placed on diagnostic intelligence, not recovery.

## 2.8 TABLES

No computing is done in TABLES; it merely contains the parser recognition tables for COMPLER. This module merely declares and initializes them. The tables are produced by the compiler generator PLICONST. Because the tables are declared to be EXTERNAL, they are therefore known to COMPLER.

These declarations and initializations are held in a module all to themselves for convenience; they are produced by the parser generator (See Section 6.9), and it is a simple matter to store that output in a distinct module. There is really no other reason not to internalize them within COMPLER.

## 2.9 CODEGEN

CODEGEN is the semantic action routine of the compiler; it is called with a production number just before a reduction is made by the parser. Its basic task is to emit DIAL object-code instructions into the structure TEMP. There is a section in CODEGEN for each possible reduction in the parse; each such section does at least one of the following:

1. Nothing (e.g., in the case of <IDENTIFIER> ::= <LEXICAL ID> );

2. Updates the symbol table, especially the TYPE information;

3. Builds a DIAL object-code instruction and puts it into TEMP.TEMP_INSTNS by a call to its internal procedure EMIT.

DIAL forward branches are handled with a fixup chain, as described under "Labels and Branches" in Gries (1971), page 280.

Run-time storage for DIAL character variables is allocated by CODEGEN when a construction reduces to <DCL STATEMENT> unless the variable is taking the default attribute (text), in which case its storage is allocated as the semantic action for the "<VAR> ::= <IDENTIFIER> " reduction.

When an error condition is detected (almost always mismatched operand types), ERROR is set to the appropriate code, and CODEGEN returns. COMPLER uses ERROR as an index into its array of diagnostics, and causes that message to be displayed.

## 2.10 <u>SOURCE</u>

This routine is called from AUTHOR to operate on the source code file for the lesson currently loaded. It has three entry points: GSOURCE, ASOURCE, and DSOURCE.

GSOURCE finds the lowest-numbered source statement whose number is greater than or equal to the parameter S_LNO, and returns that statement's text, its statement number, and its length.

ASOURCE adds to the source file the statement whose text, length, and number are passed in as parameters. AUTHOR tells ASOURCE whether the statement goes at the end of the source file, or somewhere in the middle. If necessary, ASOURCE creates a new block of source code for the added statement.

DSOURCE deletes the statement whose number is passed in as a parameter. If that statement does not exist in the file, DSOURCE returns a 'failure' code.

## 2.11 EXECTOR

EXECTOR's prime responsibility is to execute DIAL "Delta-machine" instructions and maintain the activation records. Since the machine instructions are stored in regions of the file CAIFILES, EXECTOR is also responsible for controlling the paging of instructions and literals. That is, when it needs to access a new block of instructions, it must fetch that page into core (via a call to FILEIO).

EXECTOR is called from CAIMAIN if a student has signed on. In that case, it uses the current run-time environment for that student (which was set up by SNONOFF during the resume/recover sequence) to discern the statement in the lesson in the course to begin execution with. When the lesson is finished, EXECTOR asks the student whether he wishes to continue to the next one, and performs according to the given response.

EXECTOR is called from AUTHOR if the user is an author. In that case, there is no run-time environment for the "student"; EXECTOR creates a dummy activation record, which disappears when execution is complete.

EXECTOR knows who is calling it by checking the value of the external variable SNONCODE.

## 2.12 PATPROC

PATPROC contains EXECTOR's external subroutine for implementing the DIAL system pattern-matching function. Basically, the student's response is passed in, and checked for the occurrence of a given character-string pattern, also a parameter. PATPROC returns a bit value indicating match or no-match. The pattern may contain cent-sign ("don't-care") symbols.

For an explanation of the architecture of the DIAL system pattern-matching function, see Section 4.7 of Mudge's dissertation.

## 2.13 LOGGER

This module is called by EXECTOR while in student mode; its purpose is to record every typed response issued by a student to a question, and to record the execution of every statement of a lesson. This is a statistical and analytic tool, designed to let the author see the responses evoked by his questions.

Each student's log record consists of a chain of the structures LOGRECORD. MAINFILE keeps track of all of the chains. Each structure is stored on a region of CAIFILES. A student's LOGRECORD stays allocated throughout the execution of a lesson, and is written to disk when the execution is terminated. From time to time, all of the log records are removed from CAIFILES and written onto tape by the off-line utility program CAILOG; MAINFILE is cleared of all of the references. The tape may then be analyzed and summarized by an off-line program (which does not yet exist).

## 2.14 <u>FILEIO</u>

FILEIO is the generalized file input/output procedure which the system uses to communicate with its main file, CAIFILES. The functions which FILEIO can perform include reading, writing, opening, closing, and rewriting the file. Moreover, if a read is to take place on Regions 1 through 4 of the file, which are serially-reusable resources of the system, FILEIO makes the necessary calls to the enqueuing and dequeuing procedure, ##EQDQ.

For further information about the CAIFILES file, see Section 5.3 of this manual. For information about the parts of that file which are serially-reusable resources, see Section 8.1.

## 2.15 <u>ALLOTOR</u>

ALLOTOR is a utility routine called by other procedures of the CAI System which need to use blocks of storage ("regions") on the main system file, CAIFILES. ALLOTOR keeps track of which regions are currently in use and which are currently free by maintaining the Free-Block List. There are 1499 regions on CAIFILES; region 1 holds MAINFILE, region 2 holds the Course Control Block (CCB), and region 3 holds the Author Control Block (ACB).

The Free-Block List (FBL) itself occupies regions 4 through 10 of the file. All of the other regions (11 through 1499) are controlled by ALLOTOR. ALLOTOR can either claim a free block for user by the calling procedure (thus removing its number from the FBL) or return a no-longer-needed block to the FBL.

For an explanation about the mechanism of the FBL, see the declarations for the structures FBL_TOP and FBL_SEC, illustrated in Figures 6 and 7 of this manual.

## 2.16 **##EQDQ**

This routine issues calls to ENQ and DEQ, which are CHAT routines that provide a PL/I interface to the OS ENQ/DEQ supervisor macros. ##EQDQ also maintains the status of each of the serially-reusable resources whose use is being controlled.

For an explanation about the system's serially-reusable resources and the enqueue and dequeue protocol, see Section 8.1 of this manual.

The most heavily-used entry point of this module is the ##STOP routine. ##STOP is called when the system discovers an error condition with which it can't cope. For instance, if the CAIFILES Free-Block List empties, ALLOTOR calls ##STOP. Each caller supplies its own stop code, which ##STOP passes on to IHESARC, the OS routine which sets the task return code. Hence, the stop code is displayed to the user. For an explanation of the stop codes and the mechanism of ##STOP, see Chapter 7 of this manual.

Other entry points in the module include:

##INIT -- Called from CAIMAIN, ##INIT defines all of the system resources, and enqueues the "system" resource, to lock out off-line programs;

##ENQ -- Called from FILEIO, to enqueue the first four regions of CAIFILES;

##DEQ -- Called from FILEIO, after writing on the first four regions of CAIFILES, to dequeue;

##EQEXT (##DQEXT) -- Called from SNONOFF before (after) sign-on (sign-off) to enqueue (dequeue) the student or author resources;

##EQID (##DQID) -- Called from SNONOFF before (after) sign-on (sign-off) to enqueue (dequeue) a particular user id number.

## 2.17 **#CC30IO**

This is the interface to the CHAT system input/output routines; as such, it is the only module that needs to know about them. All other routines merely address entry points of this module, without actually talking to CHAT at all. There are several entry points, each of which serves a special purpose:

#DISP and #DSPURC display the given text on the current row (designated by the external variable ROW); the latter is also passed column information, while the former begins the display in column one of ROW.

#RD_T and #RD2_T read input from the CRT screen; the former uses column information passed in as a parameter; the latter begins the read in the column where the cursor last was.

#SETROW sets the value of ROW.

#SETCRS positions the cursor on the row of the previous display, in the column specified by the parameter.

#SLIDE displays the slide specified by the given parameter. A specially encoded parameter may also signal #SLIDE to turn the projector on or off.

#DELAY calls the CHAT system delay function, to cause a pause of the given number of seconds.

#D_DIAG displays the given text (usually a system diagnostic message for an author) in the bottom two rows of the CRT.

#CAR asks the user to mount a particular slide tray on the slide projector.

#EDIT implements the "EDIT" author command, asking the user to move the cursor to the row on which he would like to enter input. #EDIT sets ROW to that row.

# Chapter 3

# CAI SYSTEM GENERATION

Each source module defined in Chapter 2 should be compiled and linked according to the procedure given in its header paragraph. The object modules have usually been stored in the partitioned dataset UNC.CS.E557C.CAI.SYSLIB.

Remember, there are two versions of the CAI System. The first, called CAI, is for student use only; it does not contain any of the author facilities. The second, called CAIAUTH, may be used by both students and authors. However, CAIAUTH uses a large amount of core for modules that a student user will never employ. That is why the simplified student version exists.

Each system knows which version it is by calling the external procedure SYS_SW. There are two versions of SYS_SW; one returns a true bit, and is stored in the module STUONLY. The other returns a false bit, and is stored in the module FULLCAI. Naturally, the module STUONLY is linked into the student-only version, and FULLCAI is linked into the full-blown edition. When an author signs on to the system, the SNONOFF module calls SYS_SW. If '1'B is returned, a "student-only" message is displayed, and the sign-on fails.

## 3.1 GENERATING THE FULL SYSTEM--CAIAUTH

The full student/author CAIAUTH system is generated as follows:

```
//jobname JOB acct,name,parms
//*PW=password
//     EXEC    PLFLD,PARM.L='XREF,LIST,MAP,RENT'
//L.SYSLIB  DD  DISP=SHR,DSN=UNC.CS.E557C.CHAT.PL1LIB
//          DD  DISP=SHR,DSN=UNC.CS.E557C.CHAT.SYSLIB
//          DD  DISP=SHR,DSN=SYS1.PL1LIB
//L.SYSLMOD DD  DISP=SHR,DSN=UNC.CS.E557C.CHATJCL.CAI.LOADLIB,VOL=
//L.OLDMOD  DD  DISP=SHR,DSN=UNC.CS.E557C.CAI.SYSLIB
//L.SYSIN   DD  *
    INCLUDE OLDMOD(SNONOFF)
    INCLUDE OLDMOD(AUTHOR)
    INCLUDE OLDMOD(ACBPROC)
    INCLUDE OLDMOD(SOURCE)
    INCLUDE OLDMOD(TABLES)
    INCLUDE OLDMOD(COMPLER)
    INCLUDE OLDMOD(CODEGEN)
    INCLUDE OLDMOD(EXECTOR)
    INCLUDE OLDMOD(CAIMAIN)
    INCLUDE OLDMOD(#CC30IO)
    INCLUDE OLDMOD(LOGGER)
    INCLUDE OLDMOD(ALLOTOR)
    INCLUDE OLDMOD(FILEIO)
    INCLUDE OLDMOD(PATPROC)
    INCLUDE OLDMOD(##EQDQ)
    INCLUDE OLDMOD(FULLCAI)
    ENTRY IHENTRY
    NAME CAIAUTH(R)
//
```

The order of the included object modules is not important, with one exception. The module TABLES must appear in the list before the module COMPLER. The reason is that TABLES initializes certain COMPLER data structures, declaring them STATIC EXTERNAL INITIAL. The COMPLER module knows them as STATIC EXTERNAL, and so the linker must be given the initialized variable references first.

The load module created by this procedure takes about fourteen tracks of a 3330 disk volume.

## 3.2 GENERATING THE STUDENT-ONLY VERSION--CAI

The procedure for generating the student-only version of the system follows. Notice that the AUTHOR, COMPLER, CODEGEN, ACBPROC, SOURCE, and TABLES modules are not included; this results in a dramatic savings in the amount of core required. Because the system makes extensive use of dynamic storage allocation, measurements of the run-time memory requirement are necessarily imprecise. However, the author version of the system requires approximately 226K, while the student-only version uses only about 134K. The LET option and the LIBRARY statement promise the linker that AUTHOR (and hence, the other omitted procedures) will never be called.

```
//jobname JOB account,name,parms
//*PW=password
//    EXEC PLFLD,PARM.L='XREF,LIST,RENT,LET'
//L.SYSLIB   DD  DISP=SHR,DSN=UNC.CS.E557C.CHAT.PL1LIB
//           DD  DISP=SHR,DSN=UNC.CS.E557C.CHAT.SYSLIB
//           DD  DISP=SHR,DSN=SYS1.PL1LIB
//L.SYSLMOD  DD  DISP=SHR,DSN=UNC.CS.E557C.CHATJCL.CAI.LOADLIB,VOL=
//L.OLDMOD   DD  DISP=SHR,DSN=UNC.CS.E557C.CAI.SYSLIB
//L.SYSIN    DD  *
    INCLUDE OLDMOD(SNONOFF)
    INCLUDE OLDMOD(EXECTOR)
    INCLUDE OLDMOD(CAIMAIN)
    INCLUDE OLDMOD(#CC30IO)
    INCLUDE OLDMOD(LOGGER)
    INCLUDE OLDMOD(ALLOTOR)
    INCLUDE OLDMOD(FILEIO)
    INCLUDE OLDMOD(PATPROC)
    INCLUDE OLDMOD(##EQDQ)
    INCLUDE OLDMOD(STUONLY)
    LIBRARY *(AUTHOR)
    ENTRY IHENTRY
    NAME CAI(R)
//
```

## 3.3 NOTES CONCERNING SYSTEM GENERATION

The OLDMOD DD line points to the PDS where the object modules for the source routines are stored. The SYSLMOD dataset is where the load module for the generated system will go. Note that there is another dataset reserved for test versions of the CAI System; you may want the load module to go there. That dataset is UNC.CS.E557C.CHATJCL.CAI.TEST. The "NAME" command in the linker input specifies the member name given to the load module, and it is this name that must be given when signing on under CHAT in order to invoke that load module. The

"(R)" specifies that if a member with the given name already exists, it is to be written over with the new load module. If "(R)" is not specified and a member with the given name already exists, then the linker stores the load module in member TEMPNAME, unless a member already exists with <u>that</u> name, in which case the job step fails.

## 3.4 GENERATING A SYSTEM TO RUN UNDER CHATHP

A version of CHAT that will allow its application programs to run either on the CC-30 or the Hewlett-Packard terminals is now under development; it is called CHATHP. A CAI System may be generated to run under CHATHP. Since the CHAT interface appears only in the #CC30IO module, only that module will have to be changed. Re-compile that module according to the following JCL, and then generate the (full or student-only) system as usual, as shown above.

```
//jobname JOB acct,name,T=2,R=200K
//      EXEC PLFCL,PARM.C='M',PARM.L='XREF,LIST,NCAL'
//C.SYSLIB  DD DISP=SHR,DSN=UNC.CS.E557C.CHAT.SOURCE
//C.CHATSRC DD DISP=SHR,DSN=UNC.CS.E557C.CHAT.SOURCE
//C.SYSIN   DD DSN=UNC.CS.E557C.CAI.SOURCE(#CC30IO),DISP=SHR
//L.SYSLIB  DD DISP=SHR,DSN=UNC.CS.E557C.CHAT.PL1LIB
//          DD DISP=SHR,DSN=UNC.CS.E557C.CHAT.SYSLIB
//          DD DISP=SHR,DSN=SYS1.PL1LIB
//L.SYSLMOD DD DISP=SHR,DSN=UNC.CS.E557C.CAI.SYSLIB,VOL=
//L.SYSIN  DD *
      INCLUDE SYSLIB(HPNTRYS)
      NAME #CC30IO(R)
//
```

The generated system should run on the HP terminals. Now, all is not automatically well. The HP terminals have a larger screen than the CC-30's, and all of CAI expects a CC-30 screen. Messages are formatted with that in mind, and variables (such as THROW_LIM, ROW, COL, etc.) are set accordingly. Until the CAI System can be rewritten for CHATHP, try reducing the screen size on the HP's by using the margin-setting and memory-lock features. However, the CHATHP version should also run as usual on the CC-30 terminals.

Chapter 4

# THE TRANSLATION AND EXECUTION OF DIAL CODE

The main purpose of the CAISYSTEM is to provide a programming system for the DIAL language. This chapter is intended to give the internal representations of source and object code for DIAL programs, and to explain the relationships among the AUTHOR, COMPLER, and EXECTOR routines. Oversimplified, the relationship is this: COMPLER translates into object code the single DIAL source statement given to it by AUTHOR; the object code is later executed by EXECTOR.

The object code generated by COMPLER is not System/360 machine code, but an intermediate form which Mudge chose to call "Delta code". This Delta code is executed by EXECTOR. Thus, EXECTOR can be thought of as an implementation of a "Delta machine". This machine has a single-address instruction format; each instruction consists only of an eight-bit opcode, followed by a two-byte operand. Instructions are paged by the Delta machine. Character string (read-only) literals addressed by Delta machine instructions are also paged.

Thus, object code consists of a set of instruction pages, a set of literal pages, and the associated page translation tables.

For an explanation of the object code instruction set, see the CODEGEN module, especially the section where the opcode mnemonic variables are declared.

The object code is reentrant, allowing for multiple student execution of the same copy of a lesson. A student's activation record contains his current Delta machine status (including state definitions and register contents), and his storage for DIAL program integer variables. This information is is held in the student's Student Control Block (SCB). An activation record also contains of the storage necessary for DIAL character strings. This is held in two structures known as POOL1 and POOL2, each of which may be thought of as forming half of one very long PL/I character string. The SCB contains indexing and pointer information about POOL1 and POOL2. A student's activation record, then, consists of one occurrence of each of SCB, POOL1, and POOL2.

Each DIAL lesson has three main parts: a source code file, an object code file, and a symbol table. Each of these parts is stored on a region or regions of the system's main file, CAIFILES; the lesson's Lesson Control Block (LCB) contains a directory pointing to each such region. Of these three parts, all are used at compile time, whereas only object code (plus, of course, an SCB) is used at lesson-execution time.

Now for a more detailed look at the DIAL compilation process. The first step in the process is started when AUTHOR receives a )LESSON command from an author. That causes the initialization of the symbol table, the page-translation tables, the LCB, and the source and object code files.

DIAL statements may then be entered for the new lesson. AUTHOR passes each one to COMPLER for compilation and object code generation. If COMPLER finds an error, it specifies a diagnostic message to AUTHOR, and returns; AUTHOR then displays the diagnostic and the offending statement, and reads the author's correction. If COMPLER does not find an error, it reports that all is well, and AUTHOR prompts the user for another command or DIAL statement.

Now for each DIAL statement that AUTHOR receives from the user, it passes that statement to COMPLER, along with the current symbol table. COMPLER operates on that statement only, emitting its object code into TEMP_INSTNS and TEMP_LIT. On return, if no error has been detected, AUTHOR does the following:

1. adds TEMP_INSTNS to the end of the file of INSTNS pages;

2. adds TEMP_LIT to the end of the file of LIT pages;

3. adds the source statement to the end of the source code file;

4. performs housekeeping, e.g., fixing up forward-reference chains in the file of INSTNS;

5. saves all the foregoing on their respective disk files.

# Chapter 5

# CAI SYSTEM DISK INPUT/OUTPUT

There are three on-line disk files used by the CAI System during a session. Two are ISAM files, and contain student and author directories, respectively. The third is a direct-access file, containing everything else: all the LCB's, the source and object code files, the SCB's, etc.

The ISAM files are accessed only by the SNONOFF routine at sign-on or sign-off. During the rest of the session, only the direct-access file, with its much faster access method, is ever used.

## 5.1 STUREC - THE STUDENT RECORD FILE

STUREC is an ISAM dataset containing student enrollment records. For a description of the creation and maintenance of this file, consult Chapter 2 of the CAI Operations Manual. STUREC is declared thus:

```
        DECLARE STUREC FILE RECORD KEYED UPDATE DIRECT
                        ENVIRONMENT(INDEXED);
```

Input/output is by the usual PL/I statements

```
    READ FILE(STUREC) INTO(structure_name) KEY(STU_ID);

    WRITE FILE(STUREC) FROM(structure_name) KEYFROM(STU_ID);
```

The records in file STUREC are instances of the PL/I structure SREC. STUREC contains all pertinent student personal data and CAI status. See Figure 1 for a complete description of the record format of the STUREC file.

```
DECLARE
/**********************************************************************/
/* SREC is what one record on the ISAM file CAI.STUREC looks like.    */
/* It contains everything there is to know about a particular stu-    */
/* dent in the CAI System.  In particular, SREC is composed of two    */
/* major kinds of information.  The part consists of all the          */
/* bureaucratic information available (name, address, course, etc.). */
/* The second is the copy of the student's Delta machine activation   */
/* record, which is stored on SREC between sessions, and used to      */
/* create the on-line (in-core) activation record at sign-on.  SREC   */
/* is used in the on-line CAI System only by SNONOFF (since it's the */
/* only module that accesses the ISAM files).  It is also used by     */
/* some of the off-line utilities programs, and by the on-line        */
/* student-file-inspector CAIOLFI.                                     */
/*                                    %INCLUDED from DCLLIB(SRECDEF). */
/**********************************************************************/
 1 SREC  CONTROLLED,    /*STUDENT RECORD*/

   /*******************************************************************/
   /* PART I:  Bureaucratic information about the student.  The       */
   /* data that does not change from session to session (e.g., name, */
   /* id, creation date, etc.) was put here by the off-line utility   */
   /* program STUMAINT.  The rest is updated by the on-line system.   */
   /* See the Operations Manual to learn about STUMAINT.              */
   /*******************************************************************/
   2 ID             CHAR(9),          /*** KEY *** Student's id number; */
                                      /* the file's record key.         */
   2 NAME           CHAR(24),

   2 PERSONNEL,
     3 PHONE        CHAR(7),

     3 YEAR         CHAR(1),          /* '1'-'4', 'G', or 'F' (faculty) */

     3 MAJOR        CHAR(4),          /* Format: standard UNC department*/
                                      /* abbreviation; e.g.: 'COMP'.    */

     3 UNC_COURSE CHAR(9),            /* Format:  'AAAABBBBN', where    */
                                      /* 'AAAA' = standard UNC dept.    */
                                      /* abbreviation; 'BBBB' = course  */
                                      /* number (right-justified); and  */
                                      /* 'N' = the section number.  For */
                                      /* example, 'COMP  161' stands    */
                                      /* for COMP 16, Section 1.        */

   2 DATE           CHAR(6),          /* Date this record was created;  */
                                      /* format = 'YYMMDD'.             */

   2 TERMHRS        FIXED BIN(15,0), /* Number of terminal hours this  */
                                      /* student has spend on the course*/
                                      /* he's now enrolled in. Excludes */
                                      /* time from abnormally-ended     */
                                      /* sessions.                      */
```

```
2 SESSIONS,
  3 NRECOVERS   FIXED BIN(15,0), /* How many recovers have been    */
                                 /* done for this student; i.e.,   */
                                 /* how many sessions have ended   */
                                 /* abnormally so far?             */

  3 NRESUMES    FIXED BIN(15,0), /* How many resume sequences have */
                                 /* taken place for this student;  */
                                 /* i.e., how many sessions have   */
                                 /* ended normally so far?         */

2 COURSE        CHARACTER(6),    /* What CAI course is this student*/
                                 /* taking?                        */

2 RECOVNEEDED   BIT(1),          /* ON means last session ended    */
                                 /* abnormally, and the student's  */
                                 /* activation record will have to */
                                 /* be obtained from the RECOVER   */
                                 /* area at sign-on time. Can also */
                                 /* mean that this student has a   */
                                 /* session in progress right now. */
2 COURSE_END    BIT(1),          /*Has student finished the course?*/

/*****************************************************************/
/* PART II:  The copy of the student's activation record, saved  */
/* in his SREC on the ISAM file STUREC between sessions.  The    */
/* activation record is composed of three parts:  the Student    */
/* Control Block (SCB), POOL1, and POOL2.                         */
/*****************************************************************/
2 SCB_PART        LIKE SCB,      /* Copied into the in-core SCB at */
                                 /* sign-on time; SCB copied into  */
                                 /* here at sign-off time.         */

2 POOL1_PART      CHAR(2032),    /* Copied into POOL1.POOL1DATA at */
                                 /* sign-on time; copied from      */
                                 /* there at sign-off time.        */

2 POOL2_PART      CHAR(2036);    /* Copied into POOL2.POOL2DATA at */
                                 /* sign-on time; copied from      */
                                 /* there at sign-off time.        */
```

Figure 1:  Model of a STUREC Record - SREC

## 5.2 AUTHREC - THE AUTHOR RECORD FILE

AUTHREC is an ISAM dataset containing author records. For a description of the creation and maintenance of this file, consult Chapter 3 of the CAI Operations Manual. AUTHREC is declared thus:

```
      DECLARE AUTHREC FILE RECORD KEYED UPDATE DIRECT
                   ENVIRONMENT(INDEXED);
```

Input/output is by the usual PL/I statements

```
  READ FILE(AUTHREC) INTO(structure_name) KEY(AUTH_ID);

  WRITE FILE(AUTHREC) FROM(structure_name) KEYFROM(AUTH_ID);
```

The records in file AUTHREC are instances of the PL/I structure AREC. AUTHREC contains all pertinent author personal data and CAI status. See Figure 2 for a complete description of the record format of the AUTHREC file.

```
DECLARE
 1 AREC CONTROLLED,
   /*********************************************************************/
   /* Each instance of this structure is a record on the keyed ISAM  */
   /* file CAI.AUTHREC.CHATJCL. The file contains personnel and work */
   /* information about each author known to the CAISYSTEM.  The file*/
   /* is used by the on-line module SNONOFF to compare the given     */
   /* sign-on id to know author id's.  It is also used by off-line   */
   /* maintenance routines such as AUTHREPT and AUTMAINT.  It is     */
   /* initialized by AUTMAINT when an author is added to the system. */
   /*    The on-line system updates TERMHRS and NSESSIONS, and uses   */
   /* ID.  It is possible to completely remove AUTHREC from the on-  */
   /* line system and from the CHATJCL.  See the System Programmer's */
   /* Manual, under "Suggestions for Future Work".                   */
   /*                              %INCLUDEd from DCLLIB(ARECDEF). */
   /*********************************************************************/

   2 ID             CHAR(9),        /**RECORD KEY**  Author's id #. */

   2 NAME           CHAR(24),

   2 PERSONNEL,
     3 PHONE        CHAR(7),
     3 UNIV_ADDRESS CHAR(20),

   2 DATE_EST       CHAR(6),        /* Date this record added to    */
                                    /* AUTHREC.                     */

   2 TERMHRS        FIXED BIN(15,0), /* Number of terminal hours     */
                                    /* spent.  Form:  nnn.nn        */

   2 NSESSIONS      FIXED BIN(15,0); /* Number of successful sign-ons*/
                                    /* accomplished by this author. */
```

Figure 2:  Model of an AUTHREC Record - AREC

## 5.3 CAIFILES - THE HEART OF THE CAI SYSTEM

The most important of the on-line datasets is CAIFILES. The creation and maintenance of CAIFILES is described in Chapter 4 of the CAI Operations Manual. CAIFILES is declared thus:

```
DECLARE CAIFILES FILE RECORD DIRECT UPDATE KEYED
           ENVIRONMENT(REGIONAL(1));
```

Note the disparity between the DD name (CAI) and the last qualifier of the dataset name (CAIFILES); this is the only file with that unfortunate quality. In this manual, it will be referred to by the less ambiguous of the two names, "CAIFILES".

CAIFILES is a direct-access dataset divided logically into blocks or "regions". Each region is exactly 2036 bytes in length, because that was an attribute given it when it was created. In PL/I, REGIONAL(1) files have the property that each block holds exactly one record. Hence, there is exactly one key for each region, since the file is also KEYED RECORD. Keys for PL/I files are character strings; when CAIFILES was created, KEYLEN=9 was specified. So, each region has a nine-character key, and the key for each region is simply the character form of the region number. For example, Region 192 has a key of '      192'. All input and output to CAIFILES by the CAI System is done by the procedure FILEIO; it is the only routine that "knows" the file. Many routines <u>logically</u> cause I/O with that file, but only via calls to FILEIO.

There are 1499 regions on CAIFILES; it occupies about 250 tracks of a 3330 disk pack. The first ten regions are fixed in purpose; they will always contain the same kind of information. Figures 3 through 7 offer descriptions of the information contained on the first ten regions of CAIFILES.

```
DCL 1 MAINFILE BASED (MAINPTR),
    /*****************************************************************/
    /* This file is held on Region 1 of CAIFILES; it is accessed     */
    /* by on-line routines LOGGER (to find where to write new log    */
    /* information) and SNONOFF (to effect recover/resume sequence);*/
    /* by the on-line program COURSE, to check password information,*/
    /* and by off-line programs.                                     */
    /*                              %INCLUDEd from DCLLIB(MAINDEF). */
    /*****************************************************************/

    /*** LOGFILES: ***/
    2 LOGF_COUNT        FIXED BIN(15,0),  /* How full is LOGFILES?    */

    2 LOGFILES(300)     FIXED BIN(15,0),  /* Each student has his     */
                   /* own logfile.  A "logfile" is a chain of "log-   */
                   /* records"; a logrecord is defined by the DCLLIB */
                   /* member LOGRDEF.  A logfile is where the on-     */
                   /* line system (via the LOGGER module) stores all */
                   /* of the lesson-execution information:  which     */
                   /* statements were executed, and what responses    */
                   /* were given by the student.  Each logrecord is   */
                   /* held in a region on CAIFILES.  This array       */
                   /* points to the first logrecord of each logfile   */
                   /* chain.                                          */

    /*** CHECKPOINT INFORMATION: ***/
    /* When a student signs on, SNONOFF puts into MAINFILE the          */
    /* CAIFILES block number where his RESUME area SCB and his          */
    /* RECOVER area SCB are stored.  That way, if the session ends      */
    /* abnormally, they can be recovered the next time that student     */
    /* signs on.  If the session ends normally, then SNONOFF stores     */
    /* the RESUME area in the student's SREC, deletes his RECOVER       */
    /* area, and removes from MAINFILE the pointers to them. Another*/
    /* way to put it is that MAINFILE.CHECKPOINT lists all students     */
    /* who are signed on right now, plus all those whose last           */
    /* session terminated abnormally.                                   */
    2 CHP_COUNT         FIXED BIN(15,0),  /* How many students have   */
                   /* checkpoint information in MAINFILES right now? */

    2 CHECKPOINT(50),
      3 STU_ID          CHARACTER(9),     /* This student's id nbr.   */

      3 RECOV           FIXED BIN(15,0),  /* Block number of this     */
                                 /* student's RECOVER area SCB copy.  */

      3 RESUM           FIXED BIN(15,0),  /* Block number of this     */
                                 /* student's RESUME area SCB copy.   */

            /*** LOG OF PASSWORD ACCESSES  ***/
            /* This log is cleared by running the off-line program */
            /* ACCSLOG.  See CAI Operations Manual for details.     */
    2 ACC_COUNT FIXED BIN(15,0),  /* How many accesses now stored?  */

    2 ACCESS_LOG(37),
  3 ID CHAR(9),     /* Who made the restricted access?              */
  3 PGM CHAR(8),    /* What program was accessed?                   */

       /*** PASSWORD ACCESS KEYS  ***/
2 FILE_ACCESS(9),   /* Room to store 9 restricted pgm names and     */
                    /* their associated password keys.              */
  3 PGM CHAR(8),
  3 KEY CHAR(3),

       /*** DAILY-JOB COMMAND VECTOR ***/
2 DJCOMV BIT(16);   /* See CAI Operations Manual for explanation;*/
                    /* especially sections about the CAILOG and  */
                    /* DJCVMNT utility programs.                  */
```

Figure 3:  MAINFILE - Region 1 of CAIFILES

```
DCL 1 CCB  BASED(CCB_PTR),
     /*****************************************************************/
     /* This is the Course Control Block, held on Region 2 of file    */
     /* CAI (CAI.CAIFILES.CHATJCL).  Like all blocks on CAIFILES, it */
     /* is 2036 bytes long.  The CCB contains information about all   */
     /* of the production courses in the CAI System.  In particular, */
     /* it lists all the course names, the number and names of each  */
     /* course's lessons, and in which block of CAIFILES the LCB for */
     /* each lesson is stored.  The CCB can hold information for up */
     /* 40 courses, and 204 associated lessons.                       */
     /*                                  %INCLUDEd from DCLLIB(CCBDEF). */
     /*****************************************************************/

     2 COUNT         FIXED BIN(15,0), /* Number of courses currently   */
                                      /* in the CAI System.            */

     2 NEXT_FREE     FIXED BIN(15,0), /* The next free element in      */
                                      /* CCB.RECS. So, NEXT_FREE-1 is  */
                                      /* the nbr of lessons in the CCB.*/

     2 COURSES(40),
       /***************************************************************/
       /* For the i-th course, CCB.COURSE(i) holds the name of that   */
       /* course, the number of lessons comprising it, and the index */
       /* in CCB.RECS of the information about its first lesson.      */
       /***************************************************************/

       3 COURSE_ID CHAR(6),           /* The name of the course.       */

       3 REC_INDEX FIXED BIN(15,0), /* Index of this course's first  */
                                      /* lesson's data in CCB.RECS.    */

       3 REC_LEN   FIXED BIN(15,0), /* Number of lessons in this     */
                                      /* course.                       */

     2 RECS(204),
       /***************************************************************/
       /* For each lesson of each course, CCB.RECS holds the lesson  */
       /* name and the block number of CAIFILES where that lesson's  */
       /* LCB is stored.  The lessons are grouped together by course.*/
       /***************************************************************/

       3 LESSON    CHAR(6),           /* Name of this lesson.          */

       3 LCB       FIXED BIN(15,0); /*Block nbr of this lesson's LCB.*/

       /***************************************************************/
       /* For example, the CCB can hold information about 40 - COUNT */
       /* more courses, and 204 - NEXT_FREE + 1 more lessons.  The    */
       /* LCB for the first lesson of course #i is held in the block */
       /* of CAIFILES given by CCB.RECS( COURSES(i).REC_INDEX ).LCB. */
       /* The LCB block for the j-th lesson of course #i is given by */
       /* CCB.RECS( CCB.COURSES(i).REC_INDEX + j - 1 ).LCB.  The      */
       /* i-th course has CCB.COURSES(i).REC_LEN lessons attached.    */
```

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

Figure 4:  The Course Control Block - Region 2 of CAIFILES

```
DCL 1 ACB BASED(ACB_PTR),
    /*****************************************************************/
    /* AUTHOR CONTROL BLOCK:  Held on Region 3 of CAIFILES, this     */
    /* structure gives status information about all authors known    */
    /* to the system.  Basically, it is a list of all authors, with */
    /* information about where all the LCB's are for the lessons     */
    /* currently in each one's library.                              */
    /*                                %INCLUDEd from DCLLIB(ACBDEF). */
    /*****************************************************************/

    2 COUNT        FIXED BIN(15,0), /* Number of authors currently   */
                                    /* known by the system; how full */
                                    /* ACB.AUTHORS is.               */

    2 NEXT_FREE    FIXED BIN(15,0), /* Next free element in ACB.RECS.*/
                                    /* NEXT_FREE - 1 is the total    */
                                    /* number of unattached lessons  */
                                    /* in all authors' libraries.    */

    2 AUTHORS(40),
      /* Each instance of ACB.AUTHORS holds a particular author's    */
      /* id number, the index in ACB.RECS of information about that */
      /* author's first lesson, and the number of lessons that the   */
      /* currently has in his library (un-attached to courses).      */

      3 AUTH_ID    CHAR(9),

      3 REC_INDEX FIXED BIN(15,0), /*INDEX OF AUTHOR'S FIRST LESSON */
                                   /*IN RECS.                       */

      3 REC_LEN    FIXED BIN(15,0), /* How many unattached lessons   */
                                    /*this author has in his library.*/
    2 RECS(184),
      /* Each instance of ACB.RECS holds a lesson name and the       */
      /* block number in CAIFILES of that lesson's Lesson Control    */
      /* Block.                                                      */

      3 LESSON     CHAR(6),         /* The lesson name.              */

      3 LCB        FIXED BIN(15,0); /* Block number of the LCB for   */
                                    /* this lesson.                  */

    /*****************************************************************/
    /* For instance, Author #i's id is in ACB.AUTHORS(i).AUTH_ID.    */
    /* He has ACB.AUTHORS(i).REC_LEN lessons in his library.  The    */
    /* LCB for the first one is held on the CAIFILES block specified*/
    /* ACB.RECS( ACB.AUTHORS(i).REC_INDEX ).LCB.  The LCB for the    */
    /* j-th one is in ACB.RECS( ACB.AUTHORS(i).REC_INDEX + j-1).LCB.*/
    /* There is room for (184 - ACB.NEXT_FREE + 1) more lessons in   */
    /* the ACB.  There is room for (40 - COUNT) more authors in the */
    /* system.                                                       */
    /*****************************************************************/
```

Figure 5: The Author Control Block - Region 3 of CAIFILES

```
/***********************************************************************/
/* The Free-Block List (FBL) is a list of all of the CAIFILES regions*/
/* that are not currently in use.  (Regions 1 thru 10 are always in   */
/* use, containing MAINFILE, the CCB, the ACB, and the FBL.  The      */
/* other regions may, for example, contain sections of DIAL source    */
/* code or object code, an LCB, an SCB, a LOGFILE, etc.)  The FBL     */
/* is a stack, composed of eight sections.  Each section can list     */
/* 500 entries.  Sections 0 and 1 are both on Region 4 of CAIFILES;   */
/* although they are defined to be two sections, they in fact act     */
/* one double-large section of the stack.  The rest of the stack      */
/* (Sections 2 thru 7) are on Regions 5 thru 10, respectively, with   */
/* one section per region.  The top of the stack (i.e., where the     */
/* pushes/pops take place) is Section 0 or 1 (depending on how full   */
/* the 1000-element free-block list in FBL_TOP is). They are the only*/
/* sections of the stack that fills up or empties out one element at */
/* a time.  When Section 0 fills up, the whole stack is shifted down */
/* A SECTION (500 ENTRIES) AT A TIME.  Section 0 is then empty once  */
/* again, and ready to be filled up again, entry by entry.  On the   */
/* other hand, when Sections 0 and 1 are both empty because of many  */
/* allocations, the entire stack is shifted up, a section at a time; */
/* all 500 entries of Section 2 are put into Section 1, and so forth.*/
/*****************************************                            */
                                         /*                           */
DCL 1 FBL_TOP BASED (FBL_TOP_PTR),       /*                           */
                                         /*                           */
/*****************************************                            */
/* FBL_TOP holds the top of the FBL stack.  Sections 0 and 1 of the  */
/* stack reside here, and the entire structure resides on Region 4   */
/* of CAIFILES.  When Section 0 fills up, it triggers a stack shift  */
/* to move all sections down.  Similarly, when Sections 0 and 1 are  */
/* both empty, it triggers a stack shift to move all sections up one.*/
/*                                   %INCLUDEd from DCLLIB(BTOPDEF). */
/***********************************************************************/

      2 NEXT_FREE          FIXED BIN(15,0), /* Which is the next empty*/
                           /* element of FBL_TOP.FREE_BLOCKS.         */
                           /* FBL_TOP.FREE_BLOCKS( NEXT_FREE ) is the */
                           /* absolute top of the stack.  NEXT_FREE   */
                           /* varies from 1 to 1000.  When it's 1,    */
                           /* it's time to shift the whole stack down */
                           /* a section to empty out Section 0.  When */
                           /* 1000, both Section 0 and Section 1 are  */
                           /* empty, and the whole stack must be shif-*/
                           /* ted up a section to fill up Section 1.  */

      2 BOS_SECTION        FIXED BIN(15,0), /* Tells which section of */
                           /* the stack is the last one used; i.e.,   */
                           /* which section is the stack bottom.      */
                           /* Varies from 0 to 7.                     */

      2 FREE_BLOCKS(1000)  FIXED BIN(15,0), /* Sections 0 and 1 of    */
                           /* the stack.  Section 1 is defined to be  */
                           /* FREE_BLOCKS(501) thru FREE_BLOCKS(1000),*/
                           /* inclusively, while Section 0 is         */
```

```
                        /* FREE_BLOCKS(1) thru FREE_BLOCKS(500),    */
                        /* inclusively.  Section 0 fills up from    */
                        /* from the bottom; i.e., FREE_BLOCKS(500)  */
                        /* is filled before FREE_BLOCKS(499); when  */
                        /* FREE_BLOCKS(1) is filled, Section 0 is   */
                        /* full, and a section shift must occur.    */

        2 FILL          CHARACTER(32);   /* Not used; brings the    */
                        /* structure up to 2036 bytes, the size of  */
                        /* one CAIFILES region.                     */
```

Figure 6:  FBL_TOP - Region 4 of CAIFILES

```
DCL 1 FBL_SECTION BASED(FBL_SEC_PTR),
/************************************************************************/
/* FBL_SECTION is what each of the remaining sections (2 thru 7) of   */
/* the FBL stack look like.  These sections reside on Regions 5 thru */
/* 10 of CAIFILES.  None of the elements of any section is ever       */
/* handled individually; rather, an entire section may be shifted     */
/* all at once.                                                       */
/*                                  %INCLUDEd from DCLLIB(BSECDEF).   */
/************************************************************************/

      2 FREE_BLOCKS(500)   FIXED BIN(15,0), /* The list of the 500    */
                           /* free blocks held in this section.       */

      2 FILL               CHARACTER(1036); /* Not used; makes this   */
                           /* structure the exact size of one region  */
                           /* of CAIFILES:  2036 bytes.               */

/************************************************************************/
/* So, for instance, the element at the absolute bottom of the stack */
/* is FBL_SECTION.FREE_BLOCKS(500) of Section # FBL_TOP.BOS_SECTION. */
/* There are ( 7 - FBL_TOP.BOS_SECTION ) stack sections not current- */
/* ly in use, and ( 999 - FBL_TOP.NEXT_FREE ) more regions may be    */
/* allocated before a stack-shift must be done to re-fill Sections   */
/* 0 and 1.  Remember, allocating blocks empties out the FBL, and    */
/* freeing blocks fills it up.                                       */
/************************************************************************/
```

Figure 7:  FBL_SEC - Regions 5-10 of CAIFILES

The rest of CAIFILES (Regions 11 through 1499) may hold one of several kinds of information, or nothing at all. In particular, such a region may hold

1. a block of DIAL source code (see Figure 8);
2. a lesson's compile-time symbol table (see Figure 9);
3. a block of a lesson's object-code instructions (see Figure 10);
4. a block of a lesson's object-code literal pages (see Figure 11);
5. a block of storage for a lesson's character-string variables (see Figure 12);
6. a block of a student's log file (see Figure 13);
7. a Student Control Block for a particular student (see Figure 14);
8. a particular lesson's Lesson Control Block (see Figure 15).

```
DCL 1 SOURCE BASED (SRC_PTR),
/****************************************************************************/
/* SOURCE is what a CAIFILES source-code block looks like.  The          */
/* LCB for a lesson contains a list of all the blocks for that          */
/* lesson which contain its source.  Each one looks like this.  The */
/* actual source is held in SOURCE.DATA.  The index of that space       */
/* is contained in SOURCE.DIR_2.  Each block of source code can         */
/* contain forty statements, or 1792 bytes, whichever is less.          */
/*                                    %INCLUDEd from DCLLIB(SCEDEF).*/
/****************************************************************************/
      2 DIR_2(40),
        /* The directory into SOURCE.DATA.  The I-th source state-   */
        /* ment held in this block has a statement number of          */
        /* SOURCE.DLNO(I), is SOURCE.DLEN(I) bytes long, and is        */
        /* in SUBSTR( SOURCE.DATA, SOURCE.DLOC(I), SOURCE.DLEN(I) ).*/
        3 DLNO FIXED BIN(15,0),
        3 DLEN FIXED BIN(15,0),
        3 DLOC FIXED BIN(15,0),

      2 FREE_LINE FIXED BIN(15,0),  /* The next free element of       */
        /* SOURCE.DIR_2.  I.e., there are FREE_LINE - 1 statements   */
        /* currently held on this block (i.e., in this structure).   */

      2 FREE_POINT FIXED BIN(15,0), /* The next free location in      */
        /* the pool of source code, SOURCE.DATA.                      */

      2 DATA CHARACTER(1792);       /* The pool of characters which */
        /* make up the source statements held on this block.         */
```

Figure 8:  Template for a Block of DIAL Source Code

```
DECLARE
   1 C_SYM_BCD  BASED(C_SYM_B_PTR),
  /*****************************************************************/
  /* C_SYM_BCD is the first half of the compile-time symbol table  */
  /* for this lesson.  The second part is C_SYM_DOPE.  This part   */
  /* is just a list of the identifiers currrently in the table;    */
  /* all information about them is contained in the other part.    */
  /* When not in core, each instance of C_SYM_BCD resides on a     */
  /* region of CAIFILES, pointed to by the lesson's LCB.  The      */
  /* index of an identifier in C_SYM_BCD is also its index in      */
  /* C_SYM_DOPE.                                                   */
  /*                              %INCLUDEd from DCLLIB(SYMBDEF).  */
  /*****************************************************************/
    2 BCD(200)   CHARACTER(10),  /* List of variables' identifiers. */
    2 FILL       CHARACTER(36);  /* Unused; exists just to make the */
                                 /* structure the exact size of one */
                                 /* CAIFILES region:  2036 bytes.   */

DECLARE
/*********************************************************************/
/* This is the second part of the symbol table.  It holds the actual */
/* type information about each entry whose identifier appears in     */
/* C_SYM_BCD.  The index of an entry in C_SYM_BCD is also its index  */
/* in here.                                                          */
/*                                  %INCLUDEd from DCLLIB(SYMDDEF).  */
/*********************************************************************/
1 C_SYM_DOPE BASED (C_SYM_D_PTR),
                                 /***********************************/
  2 TYPE(339) FIXED BIN(15,0),   /* Explained in the following Type  */
  2 ADDR(339) FIXED BIN(15,0),   /*           Definition Table:      */
  /*******************************                                  */
  /* TYPE|     MEANING      | CONTENTS OF "ADDR"  |  PUT THERE BY   */
  /* ====+==================+=====================+================ */
  /*   0 |Undefined.        |0.                   |                 */
  /*     |                  |                     |                 */
  /*   1 |Character string  |SCB.STORAGE.S_PTR    |CODEGEN <DCL ST> */
  /*     |  variable.       |  index.             |  or default.    */
  /*     |                  |                     |                 */
  /*   2 |CCNAME            |Address in LIT (an   |CODEGEN  <DCL ST> */
  /*     |                  |  encoding of page   |  using ABSOLUTE_ */
  /*     |                  |  and offset).       |  NEXT_LIT.      */
  /*     |                  |                     |                 */
  /*   3 |CHAR STR CONST    |         --          |       --        */
  /*     |                  |                     |                 */
  /*   4 |Integer variable. |SCB.STORAGE.INT index.|CODEGEN <DCL ST> */
  /*     |                  |                     |                 */
  /*   5 |Integer constant. |         --          |       --        */
  /*     |                  |                     |                 */
  /*   6 |Slide variable.   |SCB.STORAGE.INT index.|CODEGEN <DCL ST> */
  /*     |                  |                     |                 */
  /*   7 |Slide Constant    |Actual value.        |CODEGEN <DCL ST> */
  /*     |  (SC).           |                     |                 */
  /*     |                  |                     |                 */
  /*   8 | (--not used--)   |                     |                 */
```

```
   /*      |                  |                        |                  */
   /*    9 |Label.            |Branch address in       |CODEGEN <LABEL>   */
   /*      |                  |  INSTNS (encoding of   |                  */
   /*      |                  |  page and offset).     |                  */
   /*      |                  |                        |                  */
   /*   10 |Label needing     |Pointer to last ele-    |CODEGEN, when     */
   /*      |  fixup.          |  ment added to the     |  forward branch  */
   /*      |                  |  fixup chain.          |  situation.      */
   /*      |                  |                                           */
   /*   11 |PROC NAME         |          -- not implemented --            */
   /*   12 |PLISUB            |          -- not implemented --            */
   /*   13 |GLOBAL CCNAME     |          -- not implemented --            */
   /*   14 |GLOBAL INT VAR    |          -- not implemented --            */
   /*   15 |ARRAY             |          -- not implemented --            */
   /*      |                  |                                           */
   /**********************************************************************/

/* The following is a list of information needed to be retained from */
/* one author session to another.                                    */
  2 P_COUNTER FIXED BIN(15,0), /* Next free location (page,offset) in*/
                               /*INSTN PART OF OBJCODE BEING BUILT.  */

  2 SYM_FREE  FIXED BIN(15,0), /* Next free element in this symbol   */
                               /* table.                             */

  2 INT_FREE  FIXED BIN(15,0), /* Next available element in the run- */
                               /* time integer storage area,         */
                               /* SCB.STORAGE.INT.                   */

  2 STR_FREE  FIXED BIN(15,0), /* Next free element in the list of   */
                               /* string pointers, SCB.STORAGE.S_PTR.*/

/* FOLLOWING IS NEEDED TO COMPILE "CONTROLLING SHOW" INFORMATION:    */
  2 SH_LAST   BIT(1),
  2 CTRL_SW   BIT(1), /*FOR "CONTROLLING SHOW" - SIGNALS PROC AUTHOR */
                      /*TO GO CHANGE LAST INSTRUCTION (A NOP) GENER- */
                      /*ATED BY PREVIOUS DIAL STATMT TO A ZEROUNREC. */
  2 UNREC_P           FIXED BIN(15,0),
  2 REP_STACK(10)     FIXED BIN(15,0),
  2 REPEAT_STACK_TOS  FIXED BIN(15,0),
  2 SYM_SPARE(23)     FIXED BIN(15,0),

  2 FILL      CHARACTER(600);  /* Not used; exists just to bring the */
                               /* structure up to the size of one    */
                               /* CAIFILES region:  2036 bytes.      */
```

Figure 9: A Lesson's Compile-Time Symbol Table

```
DCL 1 INSTNS BASED(INS_PTR),
/*************************************************************************/
/* This is what a page (CAIFILES region) of object-code instructions */
/* looks like.  Delta-machine instructions are single-address; each  */
/* consists of an 8-bit opcode, following by a two-byte operand.      */
/* The instruction pages for each lesson are pointed to by that       */
/* lesson's LCB.  For more information about Delta-machine code and  */
/* DIAL translation, see Chapter of the System Programmer's Manual.  */
/*                                   %INCLUDEd from DCLLIB(INSTDEF).*/
/*************************************************************************/
      2 NEXT_FREE   FIXED BIN(15,0), /* The next free instruction     */
                                     /* location in OPCODE and OPND   */
          /* below.  Will be 513 (i.e., signalling "full") for all    */
          /* but the last page of instructions filled by AUTHOR.      */

      2 OPCODE(512) BIT(8),          /* The opcode parts.  For a list*/
                                     /* of all opcodes and their      */
          /* associated mnemonics, see procedure CODEGEN.             */

      2 OPND(512)   FIXED BIN(15,0), /* The operand parts.            */

      2 FILL        CHAR(498);       /* Not used; exists only to      */
                                     /* bring the structure up to     */
          /* 2036 bytes:  the size of a CAIFILES region.              */
```

Figure 10: Template for a Block of DIAL Object Code

```
DCL 1 LIT BASED (LIT_PTR),
/***********************************************************************/
/* This is what a page (CAIFILES block) of object-code character-     */
/* string literals looks like.  The literals pages for a lesson are   */
/* pointed to by the lesson's LCB.                                     */
/*                                    %INCLUDEd from DCLLIB(LITDEF). */
/***********************************************************************/
  2 NEXT_FREE FIXED BIN(15,0), /* The next free location in LIT.DATA.*/
                               /* Used by AUTHOR when adding a       */
           /* literal to this page, from TEMP.TEMP_LIT.               */

  2 DATA       CHARACTER(2034); /* The actual character-string lit-  */
                               /* erals for this page.  Each literal */
           /* begins with two bytes of length information, and a     */
           /* two-byte header containing START_COL screen formatting */
           /* information for DIAL SHOWAS statements at execution    */
           /* time.  The length, then, is the length of the literal  */
           /* plus two.  The START_COL information is used by the    */
           /* lexical SCAN routine in COMPLER, and by EXECTOR.       */
```

Figure 11:  Template for a Block of String Literals

```
DECLARE
     1 POOL1 BASED(PL1_PTR),
     /*******************************************************************/
     /*    POOL1 is the first half of the run-time storage for DIAL     */
     /* character-string variables.  The lengths and pointers of/to    */
     /* each string are held in SCB.STORAGE.S_PTR.  The second half    */
     /* of the run-time string-variable storage is POOL2.  Each of     */
     /* POOL1 and POOL2 is the size of a region of CAIFILES. Together  */
     /* with SCB, they make up a student's entire Delta machine run-   */
     /* time activation record.  When there is not enough room left    */
     /* on POOL1/POOL2 to store another string, EXECTOR's internal     */
     /* routine COMPACTIFY operates, to compress the strings and       */
     /* the pointers in SCB.STORAGE.S_PTR.                              */
     /*    Between sessions for a particular student, POOL1 and POOL2  */
     /* data parts (as well as SCB) are copied into the student's SREC*/
     /* on file STUREC.  The reason that all three are the size of     */
     /* CAIFILES regions is the resume/recover protocol of SNONOFF.    */
     /* When a student signs on, his activation record is stored in    */
     /* CAIFILES. When he signs off, it is saved in SREC, and deleted  */
     /* from CAIFILES.  That way, if the session ends abnormally,      */
     /* before the save onto STUREC, the system will have saved a      */
     /* reasonably up-to-date activation record for him.               */
     /*    Remember, only the DATA parts are stored in SREC.  The      */
     /* whole structures POOL1 and POOL2 are used either in core, or   */
     /* in a student's RESUME and RECOVER areas on CAIFILES.           */
     /*                                   %INCLUDED from DCLLIB(PL1DEF). */
     /*******************************************************************/
       2 POOL1DATA    CHARACTER(2032),

       2 POOL2BLK#    FIXED BIN(15,0), /* Points to the CAIFILES block */
                                       /* holding this student's       */
                                       /* RESUME area POOL2 structure. */

       2 C_POOL2BLK# FIXED BIN(15,0); /* Points to the CAIFILES block */
                                       /* holding this student's       */
                                       /* RECOVER area POOL2 structure.*/

DECLARE
     1 POOL2 BASED(PL2_PTR),
     /*******************************************************************/
     /* POOL2 is the second half of the run-time storage space for     */
     /* character-string variables.  See the declaration for SCB and   */
     /* POOL1 for information.                                          */
     /*                                   %INCLUDED from DCLLIB(PL2DEF). */
     /*******************************************************************/
       2 POOL2DATA CHARACTER(2036);
```

Figure 12:  Template for a Block of String Variable Storage

```
1DCL 1 LOGRECORD BASED(LOG_PTR),
 /*********************************************************************/
 /* Each student's log file is a chain of LOGRECORDs.  The first      */
 /* LOGRECORD of each chain is pointed to by MAINFILE.LOGFILES.  Each */
 /* LOGRECORD occupies one CAIFILES region.                           */
 /*                                   %INCLUDEd from DCLLIB(LOGRDEF). */
 /*********************************************************************/

   /* LOGFILE HEADER:  the first 16 bytes of each LOGRECORD.          */
       2 COURSE      CHAR(6),  /* Name of course this student is in.  */

       2 STU_ID      CHAR(9),  /* THE FIRST BLOCK OF A GIVEN STUDENT'S */
                        /* LOG; FILLED BY 'LOGGER' DURING AN 'OPEN'   */
                        /* OPERATION.                                 */

       2 FLAGS(8)    BIT(1),   /* INDICATOR DESCRIBING THIS SESSION.  */
                        /* FLAGS(1) = '0'B IF SESSION IS A RESUME;    */
                        /*          = '1'B IF SESSION IS A RECOVER;   */
                        /* FLAGS(2) THRU FLAGS(8) ARE UNUSED.         */
   /* DATA: */
       2 ENTRIES_CT FIXED BIN(15,0), /* Number of entries in this     */
                        /* this student's LOGRECORD.DATA.             */

       2 FREE_POINT FIXED BIN(15,0), /* Next free byte in LOGRECORD   */
                        /* .DATA; USED WHEN BUILDING LOGRECORD.       */

       2 DATA        CHAR(2014),      /* FORMAT:                      */
                        /*    1. DIAL lesson line # - FIXED BIN(15,0) */
                        /*    2. Length of response - FIXED BIN(15,0) */
                        /*           0 means no response was given; just */
                        /*             a plain interrupt was received; */
                        /*       1-800 means the actual response length. */
                        /*         801 means no response at all was   */
                        /*             given (or expected); used in the */
                        /*             open/close operations.         */
                        /*    3. What time response was made - CHAR(4) */
                        /*    4. Actual student response - CHARACTER(*) */

       2 NEXT_LOG_BLOCK  FIXED BIN(15,0); /* The number of the block  */
                        /* in CAIFILES holding the continuation of this */
                        /* student's log information.  If this is zero, */
                        /* it means there is no continuation; this is  */
                        /* the last block in this student's list.     */
```

Figure 13:  A Block of a Student's Log Information

As mentioned before, each of the regions 11 through 1499 may be called upon to hold different kinds of information at different times. Consider a contrived example. Assume that region 476 currently holds the LCB for a particular lesson in some author's library. Suppose that author signs on to the system, and requests a )PURGE of that lesson. The AUTHOR procedure would call ACBPROC to remove the lesson from the author's directory. Since the lesson is to be purged, its LCB is no longer needed. Region 476 would then be released, and contain nothing.

Now further assume that the author begins inserting statements into another lesson he is working on. AUTHOR handles that by calling the SOURCE module. Suppose that the block of source code fills up, and there is no room for any more source statements on that block. SOURCE would then need to create a new block of source code, and region 476 just happens to be free at the moment. Region 476 could then find itself holding source code, when just seconds before it held an LCB.

This example illustrates the notion of a list of all of the free blocks, and some orderly mechanism for claiming blocks for use (i.e., removing them from the free-block list) and freeing blocks from use (i.e., returning them to the free-block list). Such a mechanism is embodied in the ALLOTOR routine. ALLOTOR maintains the free-block list, which is held on Regions 4 through 10 of CAIFILES; any routine which needs a new region for whatever purpose calls ALLOTOR with the request. ALLOTOR responds with the number of the block which the caller may use. Similarly, to free a block, a procedure calls ALLOTOR with the number of the block no longer used.

```
DCL 1 SCB  BASED (SCB_PTR),
/*********************************************************************/
/* A Student Control Block exists for each student in the system. It */
/* is in effect an activation record for a particular student; it    */
/* contains current Delta machine status (register contents and      */
/* state switches) and run-time storage for DIAL lesson variables    */
/* (integers (for INTEGER and SLIDE types) and pointers into POOL1   */
/* and POOL2 (for CHARACTER types) ).  An SCB also points to its     */
/* student's RESUME and RECOVER file areas.   EXECTOR's internal     */
/* procedure INIT_SCB performs all initialization of the SCB, except */
/* for GLOBALS, during its beginning-of-new-lesson sequence.  The    */
/* procedure EXECTOR is the primary user/manipulator of SCB. Between */
/* sessions, a student's SCB is stored in the SREC.SCB_PART for that */
/* student.                                                          */
/*                                --                                 */
/* Because of the re-entrancy of the system, each student must (and  */
/* does) have his own SCB, and thus has his very own private Delta   */
/* machine.  However, it is easier not to think about many students  */
/* and many SCBs and many Delta machines all operating at once.  In- */
/* stead, nothing is lost by pretending there is only one of each;   */
/* the re-entrancy takes care of itself.                             */
/*                                       %INCLUDEd from DCLLIB(SCBDEF).*/
/*********************************************************************/

    /*****************************************************************/
    /* Current register contents of this student's Delta machine.    */
    /*****************************************************************/
      2 IC          FIXED BIN(15,0),     /* The instruction counter.   */
                                         /* IC tells what statement in */
                    /* the current lesson is being executed for this   */
                    /* student.  Encoded in the IC is the instruction  */
                    /* page, and the offset within the page.  Special  */
                    /* values of IC are:                               */
                    /*   IC = 0 - Student has not yet started into a   */
                    /* course (thus, SCB.LESSON is meaningless).       */
                    /*   IC = 513 - Execution is at Instruction #1 in  */
                    /* SCB.LESSON.                                     */
                    /*   IC < 0  -  Student has finished SCB.LESSON;   */
                    /* he should begin the next session at the start   */
                    /* his course's next lesson.                       */

      2 PAUSE_LEN FIXED BIN(15,0),       /* How long (in seconds) will */
                                         /* the system pause between   */
                    /* SHOWs?  Set by the "PAUSE <- n" DIAL statement. */

      2 UNREC_CTR FIXED BIN(15,0),       /* How many unrecognized re-  */
                                         /* sponses to a SHOW have been*/
                    /* received?  This is used to process the DIAL     */
                    /* UNREC statement to control branching.           */

      2 STATES,
      /***************************************************************/
      /* Current states of this student's Delta machine.             */
      /***************************************************************/
```

```
  3 READ_ISSUED BIT(1),
  3 SHOW_UP     BIT(1),
  3 CASE        BIT(1),    /* Translate alphabetic case? Set by */
                           /*   DIAL "CASEON", "CASEOFF" stmts. */
  3 SQZ         BIT(1),    /* Squeeze blanks from a response?   */
                           /*   Set by "SQZON", "SQZOFF" stmts. */
  3 CC          BIT(1),    /* Condition code - set by some DIAL */
                           /*   instructions, tested by others. */

2 STORAGE,
/*************************************************************/
/* Run-time storage for variables.  Instructions with operand */
/* type 2(3) reference this area.  A type-2(3) operand means  */
/* that the operand is a pointer into SCB.STORAGE.INT (if the */
/* opcode is appropriate to integers) or SCB.STORAGE.S_PTR.   */
/*************************************************************/
  3 INT(190) FIXED BIN(15,0),    /* Run-time integer values are*/
                                 /* held here.  Such a value   */
        /* may belong to a DIAL variable of type INTEGER       */
        /* or of type SLIDE.  In addition, the DIAL system     */
        /* integer constants are held here:                    */
        /*   SCB.STORAGE.INT(1) holds PAUSE;                   */
        /*   SCB.STORAGE.INT(2) holds SQZ;                     */
        /*   SCB.STORAGE.INT(3) holds CASE;                    */
        /*   SCB.STORAGE.INT(4) holds QVAL;                    */
        /*   SCB.STORAGE.INT(5) holds AVAL;                    */
        /*   SCB.STORAGE.INT(6) holds RVAL.                    */

  3 S_PTR,
  /* The list of character-string value pointers, pointing     */
  /* into POOL1 and POOL2.                                     */

    4 LEN(400)    FIXED BIN(15,0),/* How long is the character- */
                                 /* string value pointed to    */
        /* by this entry of SCB.STORAGE.S_PTR?                 */

    4 ADDR(400)   FIXED BIN(15,0),/* Where in POOL1/POOL2 does  */
                                 /* it start?                  */

    4 FREE_POINT FIXED BIN(15,0),/* Which is the next free     */
                                 /* position in the character  */
        /* storage area POOL1.POOL1DATA || POOL2.POOL2DATA.*/

2 LESSON       CHARACTER(6),      /* Lesson currently being     */
                                 /* executed; held on the SCB  */
        /* instead of just this student's SREC, because it */
        /* may change during the course of the session.    */

2 GLOBALS(18) FIXED BIN(15,0),

2 POOL1BLK#    FIXED BIN(15,0),   /* Next block in student's    */
                                 /* RESUMEDUMP file.           */

2 C_POOL1BLK# FIXED BIN(15,0);   /* Next block in student's    */
```

```
                                        /* RECOVDUMP file.                */
```

Figure 14:  A Student Control Block

```
DCL 1 LCB  BASED(LCB_PTR),
/***********************************************************************/
/* There is a Lesson Control Block, or LCB, for each lesson in the    */
/* CAI System, whether it is attached to a course, or in an author's */
/* library.  The LCB for each lesson points to all the source and     */
/* object code blocks for that lesson, and contains the usual direc- */
/* tory information (like how many source code blocks there are, and */
/* does this lesson need recompiling, etc.).  If the lesson is one    */
/* that is attached to a course, then this LCB is pointed to by that */
/* course's CCB.  If it is in an author library, then this LCB is     */
/* pointed to by the ACB.                                              */
/*                                   %INCLUDEd from DCLLIB(LCBDEF). */
/***********************************************************************/

    /*******************************************************************/
    /*                         Object Code Part                        */
    /*******************************************************************/
      2 C_SYM_B_BLK# FIXED BIN(15,0), /* The CAIFILES block numbers  */
      2 C_SYM_D_BLK# FIXED BIN(15,0), /* where this lesson's compile-*/
                                      /* time symbol table (C_SYM_BCD*/
               /* and C_SYM_DOPE, respectively) are stored.  These   */
               /* blocks are loaded when AUTHOR receives a )LOAD     */
               /* command, for later use by COMPLER.                 */

      2 I_PAGES_CT    FIXED BIN(15,0), /* How many instructions pages */
                                      /* (blocks of object code in-  */
               /* structions) have been used so far; also, how many  */
               /* elements of LCB.I_PAGES are currently filled.      */

      2 L_PAGES_CT    FIXED BIN(15,0), /* How many pages (blocks) of  */
                                      /* character-string literals   */
               /* have been used so far. Also, how many elements of  */
               /* LCB.LITAREAS are currently filled.                 */

      2 I_PAGES(64),   /* This lesson's instructions-pages directory: */

        3 BLK#         FIXED BIN(15,0), /* The page translation table  */
                                      /* for th instructions pages.  */
               /* The table is in logical page order; i.e., the      */
               /* n-th logical page of instructions is held on       */
               /* CAIFILES block number LCB.I_PAGES(n).BLK#.         */

        3 LARGEST_LN FIXED BIN(15,0), /* The highest source code      */
                                      /* line number in the page.    */
               /* This is used to translate the "M" into instruction */
               /* counter (IC) form, when AUTHOR receives a          */
               /* ")XEQ M,N" command, or EXECTOR receives a          */
               /* ")PROCTOR M" command.                              */

      2 LITAREAS(32) FIXED BIN(15,0), /* The page translation table  */
                                      /* for the literal areas. The  */
               /* n-th logical literal page is on CAIFILES block     */
               /* number LCB.LITAREAS(n).  This is built by COMPLER; */
               /* it is used (read-only) by EXECTOR.                 */
```

```
  2 S_PTR_COUNT  FIXED BIN(15,0), /* The number of elements now  */
                                  /* in SCB.STORAGE.S_PTR, which */
          /* is the list of pointers into POOL1 and POOL2, the  */
          /* run-time storage pools for character variables.    */
          /* This is set from C_SYM_DOPE.STR_FREE by AUTHOR. It */
          /* seems like it ought to also be equivalent to       */
          /* SCB.STORAGE.S_PTR.FREE_POINT - 1, but I'm not sure.*/


/****************************************************************/
/*                          Source Code Part                    */
/****************************************************************/
  2 BLOCK_COUNT  FIXED BIN(15,0), /* Number of CAIFILES blocks   */
                                  /* used so far to store source */
          /* Also, the number elements in LCB.MAX_LNO and       */
          /* LCB.BLOCK that are currently in use.               */

  2 MAX_LNO(418) FIXED BIN(15,0), /* Maximum line number in each */
                                  /* block.  Used primarily to   */
          /* find on which block number a particular source     */
          /* statement resides.                                 */

  2 R_MIN_LNO    FIXED BIN(15,0), /* Lowest line number of the   */
                                  /* source code changed since   */
          /* the last compile of this lesson.  Used in AUTHOR.  */

  2 COMPILED     BIT(1),          /* Has this lesson been recom- */
                                  /* piled since the last change?*/

  2 SPARE1       CHARACTER(13),   /*    (Apparently not used)    */

  2 BLOCK(418)   FIXED BIN(15,0), /* The source code directory.  */
                                  /* The n-th logical block of   */
          /* source code resides on CAIFILES block LCB.BLOCK(n).*/

  2 CDATE        CHARACTER(6),    /* The date this lesson was    */
                                  /* last changed.               */

  2 CTIME        CHARACTER(9),    /* The time this lesson was    */
                                  /* last changed.               */

  2 SPARE2 CHAR(1);               /*    (Apparently not used)    */
```

Figure 15:  A Lesson Control Block

# Chapter 6

# OPERATING INSTRUCTIONS

This chapter deals with various operational aspects of the CAI System that may be useful for a CAI System Programmer.

## 6.1 CHATJCL

CHATJCL is that set of job control language that keeps CHAT and its application programs up and running. The CAI System files must be specified in the CHATJCL. In particular, the following two datasets must appear in CHAT's "//STEPLIB" specification:

```
 //     DD  DISP=SHR,DSN=UNC.CS.E557C.CHATJCL.CAI.TEST
 //     DD  DISP=SHR,DSN=UNC.CS.E557C.CHATJCL.CAI.LOADLIB
```

Next, the three on-line file datasets must be identified. This is done in the section of the CHATJCL entitled "CHAT DD CARDS":

```
//AUTHREC DD DISP=SHR,DSN=UNC.CS.E557C.CHATJCL.CAI.AUTHREC
//STUREC  DD DISP=SHR,DSN=UNC.CS.E557C.CHATJCL.CAI.STUREC
//CAI     DD DISP=SHR,DSN=UNC.CS.E557C.CHATJCL.CAI.CAIFILES
```

Traditionally, there has been someone in the Department responsible for maintaining the CHATJCL; a CAI System Programmer will normally not have to do it.

## 6.2 MOTH-BALLING THE CAI SYSTEM

During long periods of inactivity in the CAI System, it is a good idea to moth-ball it. This saves valuable on-line disk space, and reduces the possibility of file damage brought about by an unauthorized person manipulating the CAI system files. To moth-ball the CAI System:

1. Remove the references to CAI in the CHATJCL, described above. Traditionally, the lines have not actually been taken out, but merely transformed into JCL comments. In this way, restoration of CAI in the CHATJCL is quite simple.

2. Move the files to off-line disk packs.

3. Since CHATJCL.CAI.TEST is only for test versions of the on-line system, it may be possible to delete it completely, provided there are no load modules residing therein that warrant preservation. This dataset is often reduced in size to one track, in lieu of actual deletion.

4. Follow the wrap-up procedures outlined in Section 7.2.2 of the CAI Operations Manual.

## 6.3 AUTHOR COMMAND FACILITIES

The author command facilities are described in Section 5.5 of Mudge's dissertation. The following addenda now apply:

1. Only the first three letters of any command are significant. The command may be entered in upper and/or lower case letters.

2. The )include command was never implemented. Hence, neither was )group.

3. The )course author command was never implemented; it was instead built as an independent on-line CHAT program.

4. The )number command was renamed. It is now )line.

5. The )resequence command was renamed. It is now )renumber.

6. The )print command was never implemented, but there is an off-line PRINT program. It is described in Section 7.1 of the CAI Operations Manual.

7. There is a )csw command, which flips the COMPILE switch on and off. This bit controls whether or not the lesson is recompiled each time an out-of-sequence statement is entered. This command replaces the lightpen button '*C*' described in Mudge.

8. The *SUBST* and *THROW* lightpen commands do not exist. The *THROW* feature was replaced with a keyboard command, )cle, which gives the author a clear screen on which to enter the next (presumably long) DIAL statement.

9. There exists a )copy command, which lets one author bring a copy of another author's lesson into his own library. The syntax is:
```
)COPY <author-id> <lesson-name>
```
where <author-id> is the sign-on identification of the author whose lesson is to be copied, and <lesson-name> is the name of the lesson to be copied.

10. There is an )edit command, which allows the author to specify the row on which he wishes to enter input. This is meant to allow an author to make changes to a lesson by listing the lesson and making the changes directly to the affected lines, rather than having to re-type entire statements.

## 6.4 ADDENDA TO THE DIAL SPECIFICATIONS

The author language described in Chapter 4 of Mudge is not quite the author language that was actually implemented in the CAI System.

1. The following features are not implemented: FRAME, SUBSTR, LENGTH, INDEX, PLISUB, procedures, vectors, IF-THEN, IF-THEN-ELSE, and DO-WHILE. Default branching for a MATCH statement is not implemented.

2. All DIAL keywords and their abbreviations are reserved. They may not be used for any other purpose.

3. Setting the system variable CASE may be done only via "CASEON" and "CASEOFF" statements. Setting CASE by assignment of integer value was never supported.

4. Setting the system variable SQZ may be done either via the "SQZON" and "SQZOFF" statements, or by assignment of an integer value. Assigning zero to SQZ is equivalent to "SQZOFF"; assigning any other value is equivalent to "SQZON".

## 6.5 PROCTOR OVERRIDE FACILITY

In student mode, the proctor override facility can be used to jump to any lesson or to any statement within a lesson. There is almost no on-line diagnostic help with this facility, in order to discourage student use of the facility. The proctor enters
```
)pro
```

whereupon the system responds with a ")" prompt. The proctor enters

```
                    )ove <lesson>
or,
                    )ove <line #>
```

where the lesson name is entered to override to the beginning of that lesson, and a line number is entered to override to that line within the lesson currently being executed.

If the override statement is invalid, then the )pro command must be entered to try again.

## 6.6 OPEXEC

The CHAT program OPEXEC may be used by a proctor to discover which CHAT terminals are in use, how much core is currently available, and many other things. For more information, see the OPEXEC Description document.

## 6.7 USING THE PL/I OPTIMIZING COMPILER

When CHAT was originally built, it supported application programs written only in PL/I(F) or S/360 Assembly Language. However, it has been modified fairly recently[2] to accept programs compiled with the PL/I Optimizing Compiler. The source code accepted by the two compilers differs in a few ways. Under the Optimizer, internal procedures may not be declared, all builtin functions must be declared, and some of the system functions have different names.

It may be advisable one day to generate a CAISYSTEM under the optimizing compiler; IBM no longer offers programming support for PL/I(F). The 1980 CAI System was built with the Optimizer in mind; all builtins have been declared. Therefore, to generate a CAI System under the PL/I Optimizing Compiler, do the following:

1. Delete every declaration of an internal procedure in each of the source modules. Most of the seventeen modules do not contain any such declarations.

2. In the entry point ##STOP of module ##EQDQ, replace the call to IHESARC with a call to PLIRETC. Besides this, the source code should not have to be changed, although you may want to take advantage of the DO

---

[2] by Lee Nackman

UNTIL, REPEAT, SELECT, and LEAVE statements that are available with the Optimizer.

3. Re-compile all the modules using the Optimizer. The options for compilation may have different names; make sure the same options are in fact requested. Moreover, invoke the optimization feature of the compiler; the Optimizer does not optimize unless requested to do so.

4. When the new system is generated, it will have to be linked in with CHAT's PL/I Optimizer interface (as opposed to the PL/I(F) interface currently linked in). Consult the current CHAT expert to make sure which CHAT dataset(s) to use.

## 6.8 DEBUGGING AIDS

This section deals with methods by which you can cause the CAI System to generate output elsewhere than on the user's display screen.

The CHAT system maintains a log file where sign-on and sign-off information are stored for a limited period of time. To write to this log file, use the CHAT routine LOGIT. To use LOGIT, declare it in each external procedure you expect will invoke it.

```
DECLARE LOGIT ENTRY(CHARACTER(*) VARYING);
```

When a new CAI System is generated, LOGIT will be known because it resides in the CHAT SYSLIB. To invoke LOGIT, call it with a character string. To see the results, sign on to OPEXEC under CHAT, and enter the command "SHO LOG". Your message should appear. Be warned, however, that displaying the log file also clears it. Also, the file is not very large. Writing many messages to the file will cause earlier messages to be lost. Therefore, it is advisable to display the file frequently.

Another way to obtain output from the system is to include a print file in the system. The most convenient file to use is the system print file, SYSPRINT. However, it is more convenient to direct the output to a print dataset in the CHATJCL, as the contents may then be viewed interactively. Such a dataset must have the DCB characteristics of a print file. One such file that currently exists in the CHATJCL has the ddname FACPRINT. So, to send output to the file FACPRINT, include the following statement in the main routine of the CAI System:

```
OPEN FILE(SYSPRINT) STREAM OUTPUT PRINT
PAGESIZE(nn) LINESIZE(yy) TITLE('FACPRINT');
```

The PAGESIZE and LINESIZE parameters are optional, but specifying PAGESIZE(20) and LINESIZE(40) should fit the output to a CC-30 screen. Now, any conventional PL/I print statement (e.g., PUT LIST, PUT DATA, etc.) will direct output to this file. This method has the additional benefit that when the system ends abnormally, diagnostic information is automatically sent to the print file, and without this file specification, such information would be lost.

The easiest way to view the contents of your output file is by invoking the CHAT program DISPL. DISPL will prompt you for the ddname; respond with 'FACPRINT'. The first screen-full of the file will be displayed; by pressing INT you can page through the file. If you enter a character string, DISPL will scan the file for the next occurrence of that string, and display its location. When you reach the end of the file, however, the program ends. To look at any part of the file again, you must re-invoke DISPL. An advantage to this approach is that, unlike LOGIT, viewing your output file does not destroy it.

It may be prudent one day to create a print file just for CAI; a likely ddname would be CAIPRINT. To do that, allocate an on-line dataset with print-file DCB characteristics. Then, have the person in charge of the CHATJCL insert a DD card that looks like this:

```
//CAIPRINT DD DISP=SHR,DSN=dataset_name
```

## 6.9 GENERATING A NEW VERSION OF THE COMPILER

The steps for generating a new version of the compiler are:

1. Produce the grammar for the new version using BNF programming and the XPL grammar analyzer. See McKeeman, Horning, and Wortman, A Compiler Generator.

2. Once the grammar has been thoroughly tested, use the BNF program as input to the PL/I procedure CONSTRUCTOR. This procedure produces punched output, which replaces code sections of the compiler. (The compiler is actually split into three routines, COMPLER, CODEGEN, and TABLES.)

3. PARSER contains two procedures, PARSER and COMPILER. Once the code section changes have been made in COMPILER, then test data should be run against PARSER. Note: the listing for the previous run of PARSER

contains instructions for converting an existing version of COMPILER into a version to be run against PARSER.

4. Once the changes have been verified, remove the compiler from PARSER (see instructions in the PARSER listing). Compile the compiler routines and link them into the SYSLIB object code library.

5. Generate a new version of CAISYSTEM, using the procedures described in Chapter 3.

The datasets used are:

1. For CONSTRUCTOR, UNC.CS.E557C.CAI.CONSTR, with member XPLCONST for the XPL Constructor, and member PLICONST for the PL/I Constructor.

2. The object module library UNC.CS.E557C.CAI.SYSLIB.

3. For the XPL program, UNC.CS.E557C.CAI.XPLOBJ.

The JCL is as follows:

Using XPLCONST:

```
//        JOB
//XPL        EXEC PGM=XPLCONST,REGION=150K
//STEPLIB  DD   DSN=UNC.CS.E557C.CAI.CONSTR,DISP=SHR
//PROGRAM  DD   DSN=UNC.CS.E557C.CAI.XPLOBJ,DISP=SHR
//SYSPUNCH DD   SYSOUT=B,DCB=(RECFM=FB,LRECL=80,BLKSIZE=400)
//SYSPRINT DD   SYSOUT=A
//SYSIN    DD   *
      .
      .
      .
/*
//
```

Using PLICONST:

```
//       JOB
//          EXEC PGM=PLICONST,REGION=300K
//STEPLIB  DD   DSN=UNC.CS.E557C.CAI.CONSTR,DISP=SHR
//SYSPRINT DD   SYSOUT=A
//COMMENT  DD   SYSOUT=B,DCB=(RECFM=FB,LRECL=80,BLKSIZE=400)
//DCLOUT   DD   SYSOUT=B,DCB=(RECFM=FB,LRECL=80,BLKSIZE=400)
//INITOUT  DD   SYSOUT=B,DCB=(RECFM=FB,LRECL=80,BLKSIZE=400)
//CODEGEN  DD   SYSOUT=B,DCB=(RECFM=FB,LRECL=80,BLKSIZE=400)
//EINPUT   DD   *
      .
      .
      .
/*
//
```

# Chapter 7

# ABNORMAL TERMINATION CODES

When a CHAT program in execution raises the ERROR condition, CHAT displays a message of the form:

```
SUBTASK ENDED nnnn
PLEASE CALL PROCTOR
```

and then aborts the subtask. (The "proctor" reference is historical; CHAT's original purpose was to provide a run-time environment for CAI.) There are three forms the returned code can take:

1. If the code begins with "S", it represents an OS system completion code. Look up the three-digit code in the IBM Messages and Codes manual, under "System Completion Codes (SCC)". For instance, "S80A" means that not enough storage was available for a successful GETMAIN operation.

2. If the code lies between 1001 and 1099, inclusive, it was generated by the CAI System in response to an error condition it detected, but could not fix. The routine detecting the error issued a call to the ##STOP entry point of the ##EQDQ module, which (among other things) calls the PL/I system routine IHESARC. IHESARC generates a task return code by adding the given parameter to the return code normally generated by the system (1000). Therefore,

   ```
   CALL ##STOP(10)
   ```

   yields a return code of 1010, which means that the DIAL compiler's symbol table is full. Subtract 1000 from the returned code, and look up the error in Table 1.

3. The returned code is 2000. This signals a PL/I execution-time error. In this case, the ERROR condition is raised, and the associated ON-unit in the main procedure takes control. CAIMAIN calls two PL/I builtin functions, ONCODE and ONLOC. ONCODE returns the code of the error which raised the condition, and ONLOC returns the entry point where the condition was raised. CAIMAIN displays these two diagnostics, and then calls ##STOP to halt the system.

**TABLE 1**

**Table of STOP Codes**

| | | |
|---|---|---|
| 1 | SNONOFF | During recover phase, the student's checkpoint entry was not found on MAINFILE during signon. |
| 2 | SNONOFF | Student's checkpoint entry was not found on MAINFILE during sign off. |
| 3 | EXECTOR | During setup for student execution, course was not found on the CCB. |
| 4 | EXECTOR | During setup for student execution, lesson was not found on the CCB. |
| 5 | SNONOFF | While setting up student's checkpoint entries on MAINFILE, no room was found (CHP_COUNT > 50). |
| 6 | ACBPROC | Author's ID not found on the ACB. |
| 7 | ACBPROC | No room left in RECS (NEXT_FREE>184). |
| 8 | CODEGEN | No room left in STORAGE.S_PTR; called from internal procedure S_ALLOC. |
| 9 | CODEGEN | No room left in TEMP_INSTNS; called from internal procedure EMIT. |
| 10 | COMPLER | Symbol table full. Called from internal procedure SCAN. |
| 11 | AUTHOR | No LIT pages left; more than 16 used. |
| 12 | AUTHOR | No INSTNS pages left; over 64 used. |
| 13 | SOURCE | No room left on source code file for a split. |
| 14 | SOURCE | No room left on source code file while adding sequentially. |
| 15 | SOURCE | Statement number not found in first level directory; called from ASOURCE. |
| 16 | SOURCE | Statement number not found in second level directory; called from ASOURCE. |
| 17 | AUTHOR | No room left on CCB to do an )ATTACH. |
| 18 | -- | |
| 19 | EXECTOR | During change of lesson, lesson name not found on the CCB. |
| 20 | EXECTOR | During change of lesson, no more lessons found for this course. |
| 21 | LOGGER | During open operation, no room in logfiles on MAINFILE. |
| 22 | EXECTOR | During processing of STORECH (OP(34)) instruction, COMPACTIFY was called but did not free enough space for the next string operation. |
| 23 | -- | |
| 24 | CAIMAIN | During execution of the ON ERROR unit, the ONCODE was not found in the table of codes, ONCODES(0:99). |
| 25 | CODEGEN | Compiler Debug Stop 1 -- in PLISEG production. |
| 26 | CODEGEN | No room left in STORAGE.INT. |

| | | |
|---|---|---|
| 27 | EXECTOR | During setup for execution in the case that IC<0, no next lesson found. |
| 28 | EXECTOR | Addressing error. Page decoded from IC is greater than I_PAGES_CT. |
| 29 | EXECTOR | Addressing error. Page decoded from IC is less than 1. |
| 30 | EXECTOR | Addressing error in student-only mode. No ENDLESSON instruction at end of lesson; system tries to execute beyond the lesson's last instruction. |
| 31 | EXECTOR | Addressing error. IC was about to retrieve an instruction in the last page of the lesson but beyond INSTNS.NEXT_FREE. This is a DIAL system software error. |
| 32 | -- | |
| 33 | -- | |
| 34 | EXECTOR | In FETCH21, the page decoded is greater than L_PAGES_CT. |
| 35 | EXECTOR | In FETCH21, the pages decoded is less than 1. |
| 36 | AUTHOR | During a )PURGE, lesson name was echoed correctly, but ACBPROC failed to find the lesson on the ACB. |
| 37 | -- | |
| 38 | EXECTOR | When setting up for a student beginning a course (IC=0), no lessons were found on the CCB. |
| 39 | AUTHOR | In Stage 1 of recompiling, an error was found in a DIAL statement. |
| 40 | SOURCE | DIAL system error; S_LEN of a statement is <1 or >800. Called from GSOURCE. |
| 41 | -- | |
| 42 | ##EQDQ | When ENDing on a CAIFILES resource, the subtask already had control of it. |
| 43 | ##EQDQ | When DEQing on a CAIFILES resource, the subtask did not have control of it. |
| 44 | FILEIO | Called to read with BLK# < 1. |
| 45 | FILEIO | Called to write with BLK# < 1. |
| 46 | FILEIO | Called to rewrite with BLK# < 1. |
| 47 | FILEIO | Called to read with BLK# > 1499. |
| 48 | FILEIO | Called to write with BLK# > 1499. |
| 49 | FILEIO | Called to rewrite with BLK# > 1499. |
| 50 | ALLOTOR | Attempt to allocate block numbered higher than 1499; no more blocks on FREE_BLOCK_LIST; some must be freed. |
| 51 | AUTHOR | In Stage 1 of recompiling, GSOURCE returned a nonzero return code. |
| 52 | ALLOTOR | Stack cannot be shifted down any more. Software error caused by returning more blocks than allocated. |
| 53 | FILEIO | Bottom-of-stack on free-block list has been reached. |
| 54 | SNONOFF | Attempt to DEQ on STUDENT resource |

|  |  |  |
|---|---|---|
|  |  | not successful. |
| 55 | SNONOFF | Attempt to DEQ on AUTHOR resource not successful. |
| 56 | SNONOFF | Attempt to DEQ on a sign-on ID was not successful. |
| 60 | EXECTOR | A "read-pen" or "read-either" instruction was encountered in a system without lightpen capabilities. |
| 99 | CAIMAIN | ON CONDITION(ABNORM) signalled more than 10 times. |

# Chapter 8

# PROTECTION OF CAISYSTEM DATASETS

CAISYSTEM lives and dies with its files. Great care must be taken that all datasets are protected from system failure, CAISYSTEM bugs, and acts of God. Precautions fall into two categories. First, any access to a shared file causes that file to become enqueued; no other access may occur until a dequeue operation has been completed. This is true whether the access request comes from the on-line system or from an off-line program. Second, a general protocol for taking backups of all datasets has been established.

## 8.1 USING THE OS "ENQ" AND "DEQ" FACILITIES

The CAI System module ##EQDQ calls the CHAT system routines ENQ and DEQ, which serve as interfaces between CHAT and the OS ENQ/DEQ Supervisor Macros. ##EQDQ has several entry points[3], each of which serves a various enq/deq function.

There are eight serially-reusable resources defined in CAISYSTEM, but some do not exist at times. They are divided into three groups:

1. The first group always exists. It consists of the first four blocks of CAIFILES, which contain data shared by several of the routines.
   Resource 1: MAINFILE
   Resource 2: CCB (Course Control Block)
   Resource 3: ACB (Author Control Block)
   Resource 4: FBL_TOP (Free-Block-List top)

2. The second group of resources exists when the on-line system is running and an off-line program is in execution as well. Basically, it prevents an off-line program from changing an author-related file while an author is signed on, changing a student-related file while a student is signed on, etc.
   Resource 5: CAISYSTEM (anyone signed on)

---

[3] See the module description in Section 2.16 for a description of each.

Resource 6: STUDENT (any student signed on)
Resource 7: AUTHOR (any author signed on)

3. Finally, the third "group" is used to prevent two users from signing on to the on-line system with the same identification number. When a user signs on, whether student or author, his identification number is enqueued, and remains so until he signs off.
   Resource 8: ID#

All enqueuing and dequeuing is done by the module ##EQDQ; however, only certain other modules call ##EQDQ to request such enqueuing/dequeuing. The module FILEIO controls allocation of the CAIFILES shared blocks. A resource is enqueued when a read to that block of the file is requested. The dequeue request is issued when a REWRITE to that file is received. The enqueue request is for exclusive control and with the condition that the task wait until the resource is available.

The CAISYSTEM resource is controlled by the module CAIMAIN. The STUDENT, AUTHOR, and ID# are handled by the SNONOFF module. The enqueue request is for shared control, with the condition that the task not wait if control is not immediately available. If that is the case, SNONOFF displays a failure message to the user, returns to the main procedure, and the system terminates.

Off-line utility routines requesting any of these resources do so with exclusive control specified. This provides a lockout if the resource is already in use.

For a description of the JCL and %INCLUDEs necessary to use ##EQDQ and ENQ/DEQ facilities, consult the header paragraph in the ##EQDQ source listing. For a brief explanation of the ##EQDQ module, see its description in Section 2.16 of this manual.

## 8.2 DATASET BACKUPS AND RESTORATIONS

Regular backups should be taken of each of the files. The backup and restoration procedures are well-defined and well-documented in the CAI Operations Manual. Note that when the CAI daily jobs are running, backups are automatically made. Otherwise, they must be done by submitting batch jobs described in the Operations Manual. Of course, backups are only advisable when there is any system work (test or production) currently in progress.

# Chapter 9

# THE CAISYSTEM UTILITY PROGRAMS

In addition to the on-line routines of the CAI System, several other programs exist to maintain CAI files, to produce reports, and to serve as tools for other programs. This chapter briefly describes the function of each of these utility programs. For each program, a list of %INCLUDEd modules on the DCLLIB PDS will be listed, followed by a list of all the external procedures called. All are described more fully in the CAI Operations Manual; the appropriate section number will be given for each program.

The source code for all of the utility programs resides on the PDS

UNC.CS.E557C.CAI.UTILITY.SOURCE.

## 9.1 ON-LINE PRODUCTION UTILITIES

These programs are invoked by CHAT on-line; their load modules all reside in the PDS

UNC.CS.E557C.CHATJCL.CAI.LOADLIB

(along with the load modules for CAI and CAIAUTH).

1. COURSE - list the lessons attached to a course, and delete lessons from a course.
   %INCLUDEd members: PERNDEF, CCBDEF, LCBDEF.
   Procedures called: #DISP, #DELAY, #E_INIT, #SETROW, PASSWRD.
   Reference: Section 4.3.3.

2. CAIOLFI - view the record and current status of a particular student.
   %INCLUDEd members: CHAT.SOURCE(CCIDCL), SRECDEF. By calling various entry points of #CC30IO, CAIOLFI could no doubt effect its I/O the way CAI and CAIAUTH do; however, by %INCLUDing the CHAT I/O declarations, CAIOLFI makes the calls to the CHAT I/O routines directly.
   Reference: Chapter 8.

## 9.2 OFF-LINE PRODUCTION UTILITIES

These utilities are invoked by batch programs, although included here are some subroutines that may be called by the on-line programs. Their load modules reside on UNC.CS.E557C.CAI.UTILITY.LOAD.

1. ACCSLOG - prints/clears the log of all password accesses.
   %INCLUDEd members: MAINDEF.
   Procedures called: ##INIT, ##EQEXT, ##DQEXT, HEADER.
   Reference: Section 4.4.3.

2. AUTHREPT - prints report giving all personal information, lesson names, and work times of all authors in the System.
   %INCLUDEd members: ARECDEF, ACBDEF.
   Procedures called: HEADER.
   Reference: Section 3.4.

3. AUTMAINT - utility for adding/deleting/changing author records in the System.
   %INCLUDEd members: ARECDEF, ACBDEF, EQDQEXT.
   Procedures called: ##INIT, ##EQEXT, ##DQEXT.
   Reference: Section 3.3.

4. CAILOG - prints a log of all student activity since the last run of CAILOG; clears the logfile blocks on file CAIFILES.
   %INCLUDEd members: MAINDEF, LOGRDEF, EQDQEXT.
   Procedures called: FREEBLK, HEADER, ##INIT, ##EQEXT, ##DQEXT.
   Reference: Section 4.4.1.

5. CAIREST - restore CAIFILES recover/resume dump status from tape backup.
   %INCLUDEd members: MAINDEF, SCBDEF, PL1DEF, PL2DEF, SRECDEF, EQDQEXT.
   Procedures called: FREEBLK, ##INIT, ##EQEXT, ##DQEXT.
   Reference: Section 6.3.

6. CCBMAINT - inserts/deletes courses to/from the Course Control Block (and hence, the System).
   %INCLUDEd members: CCBDEF, EQDQEXT.
   Procedures called: ##INIT, ##EQEXT, ##DQEXT.
   Reference: Section 4.3.2.

7. CHECK - displays a "please cancel" message to the console. This is the first step of each of the CAI daily jobs, but is overridden by proper execution. This prevents unauthorized users from running the daily jobs.
   Reference: Section 7.3.5.

8. DIRECTRY - prints a directory of all students in the System, complete with personal information.
%INCLUDEd members: SRECDEF.
Procedures called: HEADER.
Reference: Section 2.4.2.

9. DJCVMNT - alters the Daily Job-Control Command Vector.
%INCLUDEd members: MAINDEF.
Procedures called: HEADER, ##INIT, ##EQEXT, ##DQEXT.
Reference: Section 4.3.1.

10. FIXCAIF - adds blocks to the free block list. This should be done by the on-line system, but it isn't always. A block may be no longer used, but somehow not returned to the list. By running MAPCAIF (see below), one can ascertain which blocks are no longer used, yet not in the free block list. Running FIXCAIF returns them to that list.
%INCLUDEd members: EQDQEXT.
Procedures called: HEADER, ##INIT, ##EQEXT, ##DQEXT.
Reference: Section 4.3.4.

11. FREEBLK - a subroutine. Given a block number, it returns that block of CAIFILES to the free block list.
%INCLUDEd members: PERMDEF.
Procedures called: ALLOTOR.
Reference: Section 4.3.5.

12. HEADER - a subroutine. Causes the printing of a CAI header line at the top of a page of output.
Reference: Section 7.3.1.

13. LOCATE - given a string of data fields delimited by '$', LOCATE picks off the leftmost field. This is useful, because the input to other utilities (such as CCBMAINT) is just such a string of fields.
Reference: Section 7.3.2.

14. MAPCAIF - prints a block accounting map of CAIFILES.
%INCLUDEd members: MAINDEF, ACBDEF, CCBDEF, LCBDEF, LOGEDEF, SCBDEF, PL1DEF, BTOPDEF, BSECDEF.
Procedures called: HEADER, ##INIT, ##EQEXT, ##DQEXT.
Reference: Section 4.4.2.

15. PASSWRD - a subroutine. Called by on-line password-protected programs. PASSWRD asks for an author id, validates it, and then moves the cursor offscreen, and requests a password. If the password is correct for the invoking program (a parameter), PASSWRD returns a '1'; otherwise it returns a '0'.
%INCLUDEd members: PERMDEF, MAINDEF, ACBDEF.

Procedures called: FILEIO, #D_DIAG, #DELAY, #DISP, #RD2_T, #RMV_DIAG, LOGIT (a CHAT routine, provided at load-module-generation time, as it is in the on-line system).
Reference: Section 7.3.4.

16. PRINT - prints an author's directory, and the object code, literals, and symbol table for a specified lesson. This is a substitute for an envisioned on-line )PRINT command.
%INCLUDEd members: PERMDEF, LCBDEF, SCEDEF, SYMBDEF, SYMDDEF, INSTDEF, LITDEF, ARECDEF, EQDQEXT.
Procedures called: ACBPROC, GSOURCE, FILEIO, ##INIT, ##EQID, ##DQID.
Reference: Section 7.1.

17. SAVINIT - clears and initializes the LOGSAVE tape.
Reference: Section 7.3.3.

18. STUDUMP - prints part of the specified students' current SREC file information.
%INCLUDEd members: SRECDEF.
Procedures called: HEADER.
Reference: Section 2.4.3.

19. STUMAINT - inserts/deletes/changes student records in the System.
%INCLUDEd members: SRECDEF, SUMMDEF, EQDQEXT.
Procedures called: ##INIT, ##EQEXT, ##DQEXT, ##EQID, ##DQID, HEADER, LOCATE.
Reference: Section 2.3.

20. STUREPT - prints a progress report for a given student. Included are the amount of time spent by the student, number of recovers and resumes, and course and lesson currently being viewed by the student.
%INCLUDEd members: SRECDEF, EQDQEXT.
Procedures called: HEADER, ##INIT, ##EQEXT, ##DQEXT.
Reference: Section 2.4.1.

21. STUREST - restores students' RECOVNEEDED bits as appropriate, after loss of file STUREC.
%INCLUDEd members: SRECDEF, MAINDEF, EQDQEXT.
Procedures called: HEADER, ##INIT, ##EQEXT, ##DQEXT.
Reference: Section 6.2.

## 9.3 <u>NON-PRODUCTION OFF-LINE UTILITIES</u>

The following utilities are not considered to be ready for production use. They are not tested, or not fully implemented; some should never in fact have to be used at all. No load modules for these programs exist.

1. ACCESS - prints all password-protected programs and their associated passwords.
   %INCLUDEd members: MAINDEF.
   Reference: Section 4.4.3.

2. AUTHINIT - a one-shot initialization program for the ISAM file AUTHREC. Should never have to be used again. The source code <u>is obsolete and should be used as a model only</u>, should the file ever need to be recreated. <u>DO NOT EXECUTE THIS PROGRAM</u>.

3. CAIINIT - a one-shot initialization program for file CAIFILES. Should never have to be used again. The version in the UTILITY.SOURCE dataset is obsolete, <u>and is retained for historical purposes only</u> in case another one should have to be written. <u>DO NOT EXECUTE THIS PROGRAM</u>.
   Reference: Section 4.2.

4. STUINIT - a one-shot initialization program for the ISAM file STUREC. Should never have to be used again; in fact, the retained source code is obsolete, serving only as a procedural model should STUREC ever have to be re-initialized. <u>DO NOT EXECUTE THIS PROGRAM</u>.
   Reference: Section 2.2.

5. SUMMARY - extracts information from file SUMMREC and prints summary statistics at the end of a semester. Must be modified each semester to work at all.
   %INCLUDEd members: SUMMDEF, SRECDEF.
   Reference: Section 2.4.4.

6. SUMMERG - merges summary information from two different records into one. Used when a student is registered for a time under a temporary id, and then later under his real one.
   %INCLUDEd members: SUMMDEF.
   Reference: Section 2.3.8.

7. SUMMNIT - a one-shot initialization program for the ISAM file SUMMREC. The source is obsolete, and this program should serve only as a model, should the file ever need to be re-created. <u>DO NOT EXECUTE THIS PROGRAM</u>.
   %INCLUDEd members: SUMMDEF.
   Reference: Section 2.2.

# Chapter 10

# FUTURE WORK

This chapter is an informal description of some projects which might be undertaken to streamline or improve the CAI System. The order in which the suggestions appear is not meant to suggest an order of importance.

There are two major projects that would greatly improve the entire CAISYSTEM: a comprehensive conversion to CHATHP, and the creation of the CAI On-Line File Maintenance (CAIOLFM) system. Both of these tasks are described in A Renovation of the UNC CAI System, in Chapter 4.

Another project would be to remedy some or all of the known problems and deficiencies of the CAI System. These are documented in a hand-written list, kept in the back of the CAI Operations Manual. This list should be kept up to date, as it serves as a valuable and convenient way to keep up with system problems.

## 10.1 REMOVING AUTHREC FROM THE ON-LINE SYSTEM

The keyed ISAM file CHATJCL.CAI.AUTHREC could be removed from the on-line system. It is composed of instances of the structure AREC, containing (for each author) his name and address, his id, when he entered the CAISYSTEM, and how many hours and sessions he has had at a terminal. The on-line system sets the last two items, and SNONOFF accesses his id as the key to the file. SNONOFF does this at sign-on time, reading the author's AREC into core. Since all the id's are also stored in the ACB, SNONOFF need not ever use AUTHREC. The off-line program AUTMAINT puts the information about a new author into AUTHREC by creating a new instance of AREC. It also must update the ACB. Now, all that SNONOFF would have to do to see whether it had an author signed on would be to make a single call to ACBPROC, asking it to verify an author id. AUTHREC need not ever appear in the on-line system, and could in fact be taken out of the CHATJCL, and put on an off-line volume somewhere. What this would mean is that the on-line system would no longer be able to record the number of hours/sessions, but no one uses that information anyway; the CAI system is no longer experimental.

Now, if CAIOLFM were ever built (or CAIOLFI enhanced to include author file inspection) then AUTHREC would have to remain online in the CHATJCL; however, it still seems like a streamlining to take it out of the CAI System.

## 10.2 REJUVENATING THE CAI OPERATIONS MANUAL

The text for the CAI Operations Manual currently resides on a disk dataset in the form of Hypertext input. Since Hypertext is no longer available at UNC, an effort should be made to convert the manual to a form acceptable to a current text-formatter. The easiest candidate might be TEXT360. However, by using a text-editor, conversion to any other formatter (e.g., FORMAT, SCRIPT) might not be too difficult. Once that is done, a comprehensive update is in order. The Operations Manual is generally complete and well-written, but many details are outdated. For instance, many dataset names have been changed over the years, and the JCL given for some jobs has changed slightly.

At this writing, there is one copy of the Operations Manual that has been updated by hand to reflect all such changes. This was done during the development of the 1980 CAI System. The Department's Facilities Manager should know where that annotated manual is; it is clearly marked as the updated version.

In the back of the updated Operations Manual is a hand-written manual of class start-up procedures. It was written (apparently hastily) by someone on the original CAI team, and should one day be enhanced and text-formatted. Such a manual could, and undoubtedly did, serve quite a useful purpose.

## 10.3 THE CAISYSTEM DAILY JOBS

The CAISYSTEM daily jobs described in Section 7.4, Appendix A, and Appendix B of the Operations Manual need re-working. When CAI was in prime production, the level of use of the system warranted high-frequency backups and progress reports. Current department policy makes it clear that the CAI System will never again support as many users as it once did. Therefore, the daily jobs will probably no longer be needed. Instead, a single weekly job to handle file backups and print reports might make better sense.

## 10.4 A ")HELP" COMMAND FOR AUTHOR

During an author session, it would be quite convenient to be able to invoke a help command, to provide command syntax information. A good first step would be merely to display a static screen-full of such information, and have the user press INT to return to the prior screen format when finished reading. Later, operands may be added to the command. For instance, entering ")HELP LIST" would display usage information about the LIST command. The information available via ")HELP" could be about author comands, DIAL syntax and semantics, and even the abnormal termination codes.

## 10.5 WRITING A CAI SYSTEM USER'S MANUAL

No concise manual exists for an author who wishes to learn to use the system. The DIAL architecture and command language description are given in Mudge's dissertation, but both are outdated and incomplete. Features are described which were never implemented, and later features have been implemented which are not described in the dissertation. A very important contribution to the CAI System would be made by the person who wrote such a manual.

## 10.6 IMPROVING LESSON RE-COMPILE TIME

In his dissertation, Mudge (Chapter 8) offers a scheme for greatly reducing the time it takes for a lesson to recompile:

> Improving the recompile time involves a major software change.... An incremental compiler would avoid producing a (completely) new object code file for each source code change by structuring the object code file as a chained list, with each node being a set of object code instructions corresponding to one source code statement. This would provide the important fast response to author changes. It should be the next task undertaken in improving the implementation of the CAI System. Note, however, that such a chained structure can, by introducing another level of indirectness, result in a slower execution. So, at say )ATTACH time, all references should be resolved to absolute ones, and the code linearized, so that the execution speed is equivalent to (that of) the directly compiled code in the current implementation.... Some reprogramming of the current implementation could result in a language

processor closer (in nature) to the incremental compiler. For example, the system could make some ad hoc determination of which parts of the object code file need not be discarded.

Improvement in run-time performance might be attained by optimizing the object code at )ATTACH time. In particular, useless NOP instructions could be deleted, and object code pages could be compressed.

## 10.7 MODIFYING DIAL

Mudge (Chapter 9) offers some view about extending the author language, and the reader is referred there. One extension which seems especially useful and promising would be the PARSE system matching function, which would take a grammatical specification of a construct, and see whether a student's response parsed correctly under that construct. Such a function, it seems, would greatly simplify answer analysis in programming-language lessons, but its use would certainly not be limited to such an application.

A more immediate task should be the creation of a new DIAL grammar, whether for an extended or completely new author language, or for the one which is now in place. The grammar is needlessly complex; there are many unnecessary productions of the form <nonterminal> ::= <terminal>. Further, the grammar blurs the distinction between arithmetic and string expressions (there is even one production of the form <string expr> ::= <arith expr>) and necessitates much run-time type checking by the EXECTOR module. Many constructions which are syntactically legal are semantically disallowed, forcing the code generator and the executor to do more work than would otherwise be necessary.

Another example is the peculiar way in which the grammar treats IF-THEN-ELSE and DO-WHILE statements. The ELSE part is considered to be a statement in itself. Thus, it is syntactically legal for an ELSE clause to appear, by itself, anywhere in a DIAL program. Similarly, the ENDDO statement may appear anywhere, and the syntax does not care whether or not there is a corresponding DO WHILE. That currently causes no problems, since neither feature of the language is actually implemented. However, a new grammar would surely be in order before their implementation is attempted.

Finally, the implementation of those two statements should be attempted. They would provide much-needed relief from the GOTO mania imposed by the rest of the DIAL language. Moreover, the IF-THEN / IF-THEN-ELSE construction should include compound statement ("do-group") capabilities.

# Appendix A

# NOTES ABOUT THE MANUAL

## A.1 HISTORY OF THIS MANUAL

The original CAI System Programmer's Manual was a hand-typed documented prepared by J. Craig Mudge in 1973. It was modified later that year by O. Jack Barrier, and again by Mitchell J. Bassman in 1975. In 1980, it was text-formatted and enhanced by Paul C. Clements, to complement the newly-renovated CAI System.

The 1975 manual exists in the 1980 version as the following parts: Chapters 4, 7, and 8; Sections 3.1, 3.2, 6.5, 6.6, and 6.9.

## A.2 GENERATING THIS MANUAL

This manual was written using the SYSPUB commands of the University of Waterloo SCRIPT text-formatter. The file descriptions in Chapter 5 are imbedded directly from the CAI System declaration library,

UNC.CS.E557C.CAI.DCLLIB,

the same dataset used by the procedures of the CAI System to retrieve their declarations from. So, that dataset must be specified to SCRIPT as the SYSLIB dataset. By using this imbedding scheme, any changes to the file structures will not make the SCRIPT text of this manual obsolete. Merely make the changes to the declaration library (which would have to be done anyway) and re-generate this manual. In particular, the following DCLLIB members are imbedded into this manual; they are listed in order of appearance:

> SRECDEF, ARECDEF, MAINDEF, CCBDEF, ACBDEF, BTOP-DEF, BSECDEF, SCEDEF, SYMBDEF, SYMDDEF, INSTDEF, LITDEF, PL1DEF, PL2DEF, LOGRDEF, SCBDEF, and LCBDEF.

To generate a copy of this manual, submit the following job:

```
//jobname JOB account,name,T=2,PAGES=100,FORMS=1411
//*PW=password
//  EXEC SCRIPT,REGION=300K,OPTIONS='SEQC=73,CO=25,FNS=1000',
```

# Appendix B

## THE 1975 CAI SYSTEM

The system that Mudge and his associates built has remained unchanged since 1975. Even though the newly-renovated (1980) version replaces that system as the production system, it is still possible to use the 1975 system. The student-only version of the 1975 system is now called "OLDCAI". The full-powered version of the 1975 system is now called "OLDCAIA". It should be pointed out that the 1975 system and the 1980 system both operate on <u>precisely the same datasets</u>.

In terms of file handling and lesson manipulation, the systems are almost completely compatible. That is, a lesson created under one system may be loaded, listed, attached, changed, purged, or renamed successfully using the other system. The resequencing command, however, is an exception. That command is not supported in the 1975 system, and expects lessons to have four-digit line numbers in the 1980 system.

In terms of DIAL programming, the two systems are also nearly compatible. That is, a lesson created using one system may be re-compiled and/or executed using the other. There are two exceptions. The 1975 system does not support integer addition. The 1980 system does not support lightpen facilities. So, any lesson containing either of those two features is bound to the system under which it was created.

The source code for the 1975 system is on the partitioned dataset UNC.CS.E557C.CAI75.SOURCE. It is also stored on a tape. The 1975 system's object modules and DCLLIB dataset are also on tape. See the Operations Manual for details.

## REFERENCES

1. Bassman, Mitchell J. UNC CAI System Operations Manual. University of North Carolina at Chapel Hill. 1975.

2. Gries, David. Compiler Construction for Digital Computers. Wiley. New York. 1971.

3. McKeeman, W. M., Horning, J. J, and Wortman, D. B. A Compiler Generator. Prentice-Hall. Englewood Cliffs, N. J. 1970.

4. Mudge, J. Craig. Human Factors in the Design of a Computer-assisted Instruction System. Ph.D. Dissertation. University of North Carolina at Chapel Hill. 1973.

5. Schultz, Gary D. The CHAT System: An OS/360 MVT Time-Sharing Subsystem for Displays and Teletype. M. S. Thesis. University of North Carolina at Chapel Hill. 1973.

## LIST OF FIGURES

## LIST OF TABLES