1988

Cview, a graphical program generator for the C programming language

Walter E. Martin

Rochester Institute of Technology
School of Computer Science and Technology

Cview
a
Graphical Program Generator for the C Programming Language

By
Walter E. Martin

A thesis, submitted to
The Faculty of the School of Computer Science and Technology,
in partial fulfillment of the requirements for the degree of
Master of Science in Computer Science.

Approved by:

Dr. Peter Lutz
Committee Chairperson

Dr. Andrew Kitchen
Reading Committee Member

Dr. Peter Anderson
Ex Officio Member

Title of thesis: Cview a Graphical Program Generator for the C Programming Language

I Walter E. Martin hereby grant permission to the Wallace Memorial Library, of RIT, to reproduce my thesis in whole or in part. Any reproduction will not be for commercial use or profit.

Table of Contents

Dedication

This thesis is dedicated to my mother and late father; both understood the importance and value of higher education and supported me through my academic adventures and misadventures; my late aunt, Alice McKinney, for encouraging my interests in the physical and mathematical sciences; my wife Victoria for her support, understanding and inspiration and my son Charles who during his brief five year life has contributed more to "daddy's thesis" than he will ever imagine.

This document was prepared and printed on an Apple Macintosh SE and a LaserWriter Plus using Microsoft Word and Silicon Beach Super Paint (with thanks to the folks at Publisher's Workshop). The text of the thesis is 11, 14 and 18 pt. Palantino and computer generated items are set in 11 pt. Courier

Abstract

The electronics industry has long been a user of graphical design aids. Graphical design tools for VLSI circuits are used to lay out the design, generate data structures for the simulation of the circuit and eventually produce the photomasks for the production of the chip. As such these tools form a working environment that allows an engineer to design, debug and produce a final product.

Software design, in general, is still done by writing individual lines of code. At best a macro language or a meta-language will be available to ease the pain of detail coding. While these tools are useful, they are text oriented instead of graphics oriented. It makes more sense to allow programmers to draw regular features of their programs, just as VLSI designers use libraries of common electronic devices. Source code could then be generated from the drawing. A software package, Cview, has been designed and implemented to explore the idea of computer aided software design.

1. Introduction

It is an acknowledged fact that computers have had a major impact on society at large. Software development and maintenance, a $40 billion industry in 1980, accounts for over 80% of the total computer systems costs today. Embedded computer systems in industrial settings have led to a situation where 40% of the work force uses computers without any knowledge of computer operations and a major portion of the U.S. population relies and implicitly trusts computer software in applications such as nuclear fueled power plants, automated banking tellers, air traffic control and ignition/fuel injection systems in their cars.

The decreasing cost of hardware has been driven by intense competition and the automation of both the design and manufacturing of computer components. Software design, until recently, has been considered more of an art form than an engineering endeavor and has resisted automation. The increasing software to hardware cost ratio has forced programmers to view their craft in a new light; a light that emphasizes speed of design and implementation, reliability, usability and ease of maintenance. The profession of software engineering, the application of mathematics and scientific method to the production of programs and documentation, has developed to address the changes needed to produce the complex software required by today's society **[BOEH81]**.

Work in the area of automated software engineering has been concentrated in three areas - Applications generators/program generators, programming/system development environments and individual software tools. The range of sophistication ranges from the very complex (if not impossible to implement) to much simpler programs such as context sensitive editors.

The traditional approach to system development consists of analyzing requirements, identifying alternatives, selecting the best solution, obtaining approvals, designing internal and external system specifications, writing and testing the procedural code, publishing documentation, conducting training and implementing the system. This approach is generally unsatisfactory for several reasons. Due to the long design time the system requirements usually change before the system is implemented. Changes in design have large impacts on budgets and completion dates, so this approach causes large backlogs of undeveloped systems and the systems that are difficult to maintain [CARD82]. Application generators promise to break the traditional development cycle, producing easily maintained, documented systems within the required time and budget constraints.

Applications generators, ideally, accept as input database specifications - preferably for a generalized DBMS, report formats, types of transactions and job control logic and produce an executable application. The problems associated with applications generators range from the inertia built in to the use of standard high level languages to the short comings of current application generator systems. Application generators currently are limited because of non-standard data-base implementations, limited language syntax and lack of debugging facilities. Problem specificity is another area of difficulty. A particular application generator may work well for accounting problems but performs poorly for more generalized problems. None of the currently available packages that have been billed as "applications generators" are able to meet the goals listed above and are considered to be of limited use [GROC82] [WALD82].

Program generators are programs that write source level programs given a set of input parameters. Program generators are an integral part of applications

generators and are plagued by many of the same problems. Three areas of knowledge must be built into a program generator - algorithm design (understand the problem then plan, implement, verify and evaluate the solution), detail program syntax and problem domain. None of the program generators available today meet all of these criteria **[KANT85]**. The existing program generators write programs for a very limited class of problems and adhere to both a limited design philosophy and coding style. If all of the problems to be solved belong to one problem space (formatting video screens and reports, sorting, simple file updating) a program generator can be used to produce the prototype code and then programmers could modify or reuse procedures as needed **[ROTH82]**. While this is not the complete generation of source code the amount of time saved using prototyped procedures could be significant.

CASE (Computer Aided Software Engineering) is a system similar to CAD/CAM systems in common use by electrical and mechanical engineers. CAD/CAM systems used for VLSI design typically include an editor that is used to layout the individual components, tools that use the layout file to generate a description of the circuit for testing purposes and tools that convert the layout file to the photomasks used to produce the wafers from which the individual chips will be cut. A CASE system could encompass either a system development environment or a programming environment.

System development environments are used to manage the work of multiple programmers working on complex projects. In this work environment multiple versions of one program may be in use or multiple programmers may be updating one program. The environment includes a programming environment (described below) plus the tools required to manage the complexities of the current project. As with any complex system the requirement for proper documentation is paramount

since it constitutes the backbone of communication between programmers, designers, managers and the customer. The documentation for the system would be generated directly from the use of the development environment **[CAMP83] [ELLI85] [NOTK85]**.

Programming environments are collections of "tools" (editor, compiler, loader/linker and debugger) that have a unifying command structure. These tools are generally designed to work with one programmer on one program at a time. In many cases the programmer is working with an incomplete specification and must quickly generate a prototype program for user approval. The program does not necessarily have to be a working version but just a skeleton that performs some basic functions. The tools in the programming environment allow the programmer to quickly generate a prototype program then do incremental development if the user is satisfied **[TAYL82] [CHES84]**. While these tools can be separate programs with a common command structure, the trend is toward a "seamless" interface such as Magpie **[DELI84]** and the Cornell Program Synthesizer **[TEIT81]**. These seamless systems include syntax directed editors, incremental compiling and immediate execution of program fragments. Magpie uses a windowed environment while the Cornell Program Synthesizer works on text oriented video terminals.

The editors in these two systems differ somewhat. Magpie does not prevent programmers from entering syntactically incorrect source code. The environment accepts the input and checks for errors. If errors exist the programmer is informed but no corrective action on the programmers part is required. The Cornell Program Synthesizer is a syntax directed editor for the PL/I language. The programmer inserts template statements via a predefined function key then uses the cursor keys to move to and fill in each part of the template. A parser checks the work as it is entered and

immediate corrective actions are taken as needed. Many errors are eliminated in this system since it is impossible to have an error in the template.

This thesis deals with this one aspect of the programming environment - the graphics oriented syntax directed program editor. The use of a syntax directed editor has several advantages. Since programmers tend to think of programs in terms of their structure **[NOTK85]**, editing is done in terms of the program language constructs instead of individual lines of code. This eliminates the "; in the wrong place" type syntax errors (and a few compiles). The programmer manipulates a minimum of source code since the editor can supply a template (via a function key or menu selection) for keywords or program structures. Detail source code or other templates can then be inserted in the appropriate places by the programmer **[STAN84]**. Once a template is complete the editor could then check the statement(s) for syntactic correctness marking errors for immediate correction. The data structure generated by the editor (a parse tree for example) could then be used as input to a debugger and/or object code generator **[MORR81]**.

The editors and concepts mentioned so far have all been text oriented, there are, however, graphics oriented editors. The GRASP **[WORK83]** and GRIP **[WORK85]** programming environments include graphics oriented editors using D-Chart (named in honor of E. W. Dijkstra) diagrams. D-Charts are control flow diagrams that emphasize the structure of algorithms. Within the GRASP environment there are twelve primitive commands for defining entry/exit points, blocks and abstract references, loops, selections and cases. The commands have both a text and graphical form - a D-Chart is specified as a list of commands, which are syntax checked as they are entered, and then the two dimensional graphical representation is displayed on a

CRT. GRIP, on the other hand, manipulates the D-Chart symbols directly on the screen using the recursive substitution of symbols to complete the diagram.

To construct a D-Chart the following rules could be used **[HWAN82]** :

1) A D-Chart consists of any of three control structures (sequential, selective, repetitive). Cases are considered a sequential structure.

2) There is one entrance and one exit from each control structure.

3) Each control structure entrance and exit must be easily identified.

4) There must be only one entry and exit from a repetitive control structure.

5) Each repetitive structure must be identified by a different alphabetic character (F, D, W for for, do . . while, and while).

6) The logical flow of the chart must flow from top to bottom.

7) A control structure can completely contain another control structure.

8) The GOTO can be used to implement a control structure.

The claimed advantage of this notation over another graphical representation, flowchart, is that D-Charts use only three structures, sequential, selective (IF-THEN-ELSE) and repetitive for all loops. This allows for the one-to-one translation from D-Chart to any block structured language. D-Charts are also easier to follow since the control of flow is always from top-to-bottom with no upward flow or crossing flowlines.

2. Design Background.

There exists a more or less set number of steps in program design. The designer is introduced to the problem either verbally or via written specifications, the overall design is sketched out defining the individual functions, procedures and data structures then individual modules are designed and coded. The design of the main program with its associated functions and procedures ideally is detailed using some intermediate language such as pseudocode, flowchart, dataflow diagrams, etc. This allows the designer to think through the processing and catch problems that may arise from faulty specification, faulty logic or plain oversight. Once this design document is finished it is translated into whatever programming language has been deemed suitable for the task at hand.

The problem with this system is that flowchart has been the predominant design language and programmers rarely prepare a flowchart as part of the design process. Not that programmers are to be blamed. If a programmer's understanding of the problem is complete enough to draw a flowchart, it is complete enough to write the program. Obvious problems will probably manifest themselves during the coding process. Once the program is finished any logical or operational flaws can be quickly worked out of the code during the testing phase of the development cycle. Once the program is working a flowchart can be produced *documenting* the details of the code. Flowcharts have been used as documentation aids rather than design aids because programmers naturally work with programming languages and flowcharts are difficult to draw and modify. The role of flowcharts then has been that of program documentation rather than program design aid **[BROO82]**.

Let's look in a different direction for a possible solution. The designer of VLSI circuits faces the same design problems as the software designer. If the hardware designers task is to design a simple but specialized ALU she/he will probably "know" that the design will involve two operand latches feeding an adder array which is followed by two product latches. The output from the adder array and the latches will be fed to a 4 to 1 multiplexor whose output is sent to an 8 bit I/O port. The design has mentally been sketched out in some detail only requiring the addition of signal and power busses. At this point in the design cycle the design methodology is different for hardware and software.

The software designer can sit down at a terminal and type in the skeleton of the proposed program. This much can be debugged and tested before the addition of detail code. The hardware designer does not have that luxury. The production of silicon chips is both time consuming and costly. The ALU must be laid out complete with all the interconnections, power busses and connection pads using a CAD system. This is where the hardware designer has the major advantage. The CAD system will have all of the common components (latches, I/O ports, etc.) available as predefined objects. The hardware designer needs only to insert these pretested components into the layout and connect them together, following a set of design rules determined by the particular circuit fabrication technology. The output file can be used for three purposes. It can be used by another program for the simulation of the circuit to insure that it meets all of the design criteria, the photomasks for the production of the chips can be produced from it and the file as it is displayed on the CAD workstation is a means of communication between designers (the equivalent of a flowchart). If the design does not meet specifications the CAD system can be used to change the buss structure or layout geometry until the design works properly (at least satisfies the simulator.)

At the point where the hardware designer is testing a complete circuit the software designer is probably typing in detail code and doing incremental testing. As the testing uncovers problems individual lines of detail code are modified using a text editor. Eventually the program is finished and tested but as yet there is no external documentation. An interesting comparison can be made: The hardware designer must work with an intermediate graphical representation of the final product that can be used both for testing and documentation. The final product for the most part works correctly and requires very little debugging. The software designer tends to work on different versions of the final product following a "code then test" procedure that eventually arrives at the final product. This procedure does not produce the required external documentation.

If the software designer works with a CAD system designed to layout programs in a graphical manner can this problem be eliminated? A short example, which uses flowchart as the graphical language, should suffice to demonstrate the feasibility of this idea. Let's say that the designer must write a main program that reads single keystrokes from the keyboard and accepts Load, Save, Edit, Compile, eXecute and Quit as valid commands - all other keystrokes cause the terminal to beep. Each of the above characters causes a function to be executed with control returning to the main program. The programmer "knows" from experience that the program will most likely consist of a pre-checked loop which contains a read and a multiway select. Now instead of using a terminal to type in this program skeleton, the programmer uses a CAD system to draw the program structure.

Using a mouse as a pointing device a loop structure could be selected from a menu and inserted into an empty flowchart. The multiway select would be picked

next and inserted as the body of the loop. The programmer would then indicate the number of cases in the multiway selection. At this point the skeleton of the program is defined and few if any keystrokes have occurred. Procedural code and various expressions would be inserted next using process symbols and a built-in word processor. The variable definitions before the loop, keyboard read, multiway selection expression, function calls and loop expression would all be entered. The completed flowchart (actually the data structures that represent the flowchart) and the VLSI circuit layout at this point are equivalent. Both are graphical representations of some object - a silicon chip on one hand and a program on the other. Both can be used to produce the object that they represent - a program is used to generate the photomasks for chip production and a program could generate source code from the CAD file. And both are a means of communicating the function of the object to another designer.

As mentioned the tools used for testing by each group of designers is different. Hardware designers have at their disposal a number of circuit and logic simulators for checking the validity of their designs before the actual chips are produced. In the software realm the "final" product is produced, compiled and tested with representative input data. In either case the CAD file is the input to the testing environment regardless of the tools and techniques used.

The goal of this thesis is to produce a CAD system for the design and generation of programs in the C programming language. The system will comply with the following general design criteria.

1) It should use a familiar or easily learned graphical notation.

2) The user interface should be easy to learn and use.

3) The drawing skills required of the user should be minimized.

4) The format of the CAD file should have a simple fixed format editable with a standard editor.

5) The amount of code that is generated for the predefined structures should be maximized.

6) The code should be generated in a reasonable and consistent manner.

The first three items are of paramount consideration. If the user interface for the CAD system is difficult to learn and use the tendency will be to avoid it. No assumptions can be made about the users artistic ability - the user should only have to indicate locations of objects and the system should do the placement and interconnection of the symbols. The CAD system must be designed so that the amount of code that can be generated per unit of work time exceeds the amount of code that can be typed using a standard editor. This augments the basic assumption that not only is the programmer writing the program but producing a required piece of documentation.

Requiring the file format to be of a "simple fixed format" accessible with a standard editor has several advantages. First, if the file format is relatively simple it is possible to fix files that have been, for some reason, damaged. Second, if a simple change must be made to the detail code (assignment statements, expressions, etc.) an editor can be used rather than the CAD system. While this sounds strange and possibly counter to the idea of using a CAD system, it should be remembered that the CAD system is used to lay out the structure of the program. Detail procedural code is

added after the structure is complete. Third, a simple file format makes the coding of adjunct utilities (debuggers and syntax checkers) easier.

3. Cview.

3.1 Cview Iconography

Two traditional graphical tools for detailing procedural code are flowcharts and flowgraphs. Flowcharts are not a favored tool for program design [BROO82]. Drawing a detailed picture of a program replete with on and off page connectors, predefined procedures, preparations, annotations, etc. requires time and patience. Even more time is required to update the flowchart when the program specification changes or a programming bug is corrected.

Flowgraphs, consisting of terminals, connectors, decisions and flowlines, can adequately represent the structure of a program, but no detailed information can be included since the standard flowcharting symbols which convey detailed program information are not part of the flowgraph iconography.

In Cview a middle ground has be taken, avoiding the extreme detail of flowcharting but allowing for more detail than flowgraphs. The flowchart symbols preparation, input/output and predefined process are defined strictly as a process (a sequence of procedural statements). This reduces the number of symbols required to represent programs from nine to six. The symbol for comment/annotation has been eliminated by allowing the user to annotate individual symbols using a built-in text editor. This reduces the total number of symbols to the following:

TERMINAL	Indicates the entry point to and the final exit from a flowchart.
PROCESS	Any user defined procedural code. This can include preparation, input/output and predefined process.
DECISION	Binary operation used to form IF-THEN-ELSE and all loop structures.
CONNECTOR	Merging point for two and only two logical flowlines.
FLOWLINE	Indicates the sequence of processes and decisions.

While this subset of flowchart can represent all of the standard program structures plus detail procedural code, it lacks two useful features. The C programming language supports the switch statement which allows multiway selection without using a series of nested IF-THEN-ELSE statements and parallel processing via the UNIX fork and wait system calls. With the addition of switch and parallel section symbols, a relatively simple subset (and/or extension) of the flowchart language is defined that is hopefully powerful enough to represent complex single and multi-tasking programs. This modified subset of the flowchart language will be referred to a Cgraph.

The Cgraph symbols are used either individually, in the case of process and switch, or in combination to form the Cview program structures IF-THEN-ELSE, LOOP and PARALLEL SECTION. The program structures are closely based on the three basic flowgraphs in Figure 3.1.1.

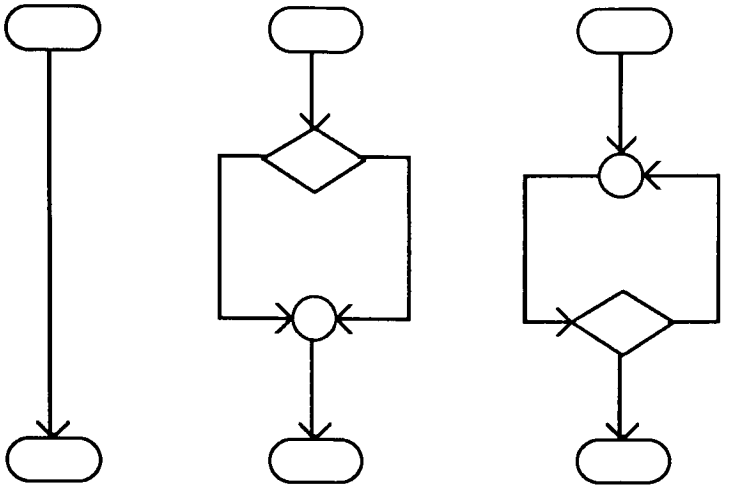


Figure 3.1.1

The null flowgraph is the starting point for building all Cview diagrams. The FORK-WAIT parallel sections are structurally identical to IF-THEN-ELSE and all the loops (WHILE, FOR and DO-WHILE) share the common loop representation. Given this the symbols or structures in Figure 3.1.2 are defined as atomic units to Cview.

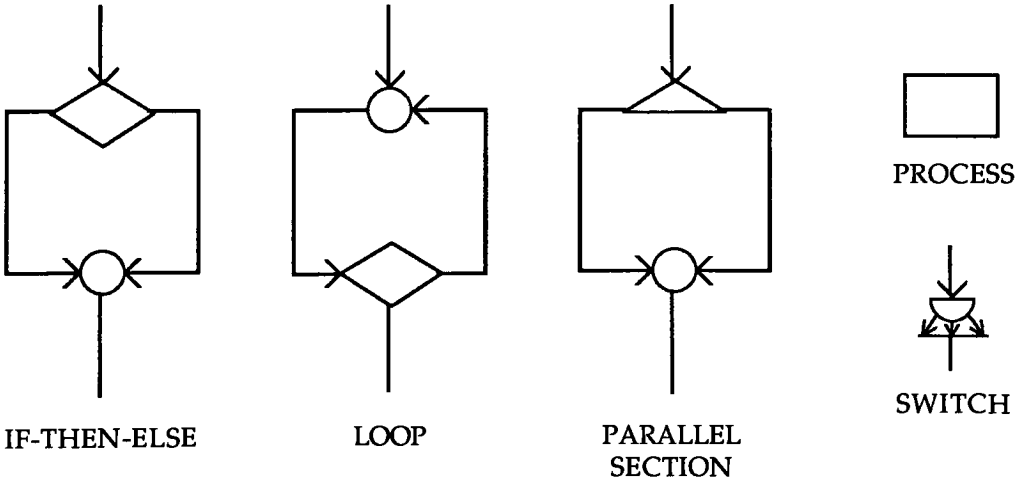


Figure 3.1.2

If flowcharts are so unpopular, why base a graphics editor on them rather than D-Charts or some other notation? The first and most basic reason is that almost every student of computer science has at some time, either from lecture or text, been exposed to the concept of flowcharting. While familiarity may breed contempt, the flowchart symbols are none the less a familiar notation. A notation, in this case, that allows a programmer to generate a required document (a graphical description of the program) and the program that corresponds to that document. The second reason is that studies indicate that the comprehension of programs and manual procedures described using flowchart and flowchart like notations compare favorably with prose descriptions **[KAMM75] [SHNE77].** The last reason is that many of the advantages claimed for D-Chart can also be claimed for Cview. Comparing a set of rules for generating a Cview diagram, listed below, with the rules listed in the introduction for D-Charts, shows a favorable comparison.

1) A Cview chart consists of any of four control structures (sequential, IF-THEN-ELSE, LOOP, FORK/WAIT). Cases are considered a sequential structure.

2) There is one entrance and one exit from each control structure.

3) Each control structure entrance and exit must be easily identified (terminated by decision, fork, connector).

4) There must be only one entry and exit from a repetitive control structure.

5) Each repetitive structure can be identified as either being pre or post checked. Extra commands can show th loop type.

6) The logical flow of the chart flows from top to bottom except inside of loops where one leg of a loop flow in the up direction.

7) A control structure can completely contain another control structure.

8) The GOTO statement is not allowed.

3.2 Design Overview

The Cview editor/program generator is based on the concepts found in the GRIP editor. The Cview diagrams for programs are drawn using a graphical interface in a top-down manner using recursive substitution. In designing the editor many decisions had to be made concerning the user interface. This interface includes the use of the video display, keyboard and mouse.

The original design concept was to make Cview as portable as possible. To accomplish this goal the screen design was based on a character oriented display assuming only cursor positioning capabilities. The "drawing" functions were isolated during the code design to one source and include file. The initial implementation of Cview was done on an IBM-XT using a 24X80 character mode display. At a later date Cview may be ported to another system supporting vector or bit-mapped graphics.

It was decided to leave the screen as uncluttered as possible. When writing applications for a full screen or windowed environment, it is very tempting to display every possible control and status item using multiple windows or dedicated sections of the display and build features in to the system just because they are easily implemented **[THIM85]**. Also if the screen cannot contain a minimum number of required windows (the window "working set") the user will spend time searching for information in hidden windows, deleting windows to make room for others or opening windows to get at required information **[MAGU85]**. This could lead to a design that is visually confusing and difficult for a new user to learn. Instead of constantly displaying status information, pop-up menus and appropriately shaped cursors were used.

"Pop-ups" are items that literally pop up on the screen to perform some specific function then disappear. Pop-ups can be used to display simple textual information, request confirmation for irreversible commands or allow the programmer to select program options via either fill-in items, command lists or buttons.

The justification for **not** using multiple windows is straight forward. The D-Chart editor which is part of the GRIP programming environment **[WORK85]** is detailed in Figure 3.2.1

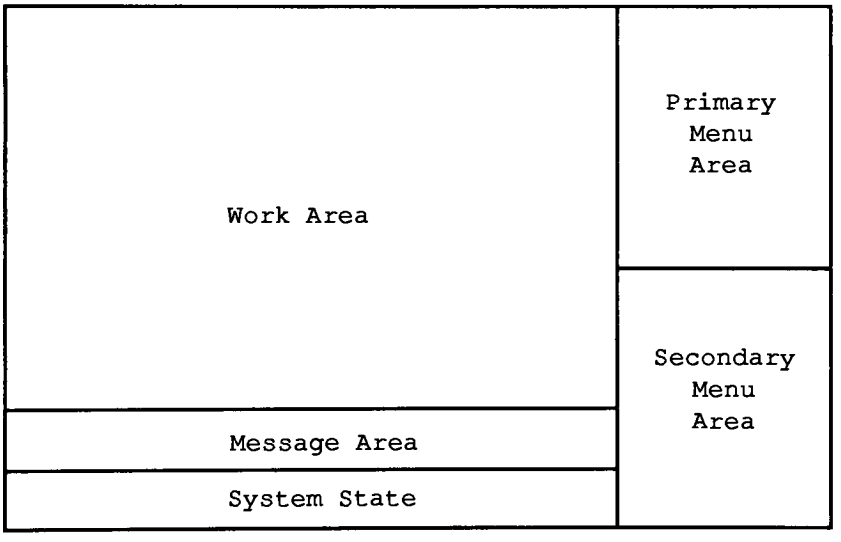


Figure 3.2.1

The Primary and Secondary menu areas are replaced with pop-up menus which contain command lists. The Message area is replaced with pop-ups, that again, only appear when needed. Finally, the System state which conveys information concerning the current command or location being edited is replaced with a pop-up

or an appropriately shaped cursor. This leaves only the Work Area where the flowgraph is composed. The biggest advantage of this is that the Work Area has use of the complete screen and does not surrender any "real estate" to informational windows.

Both the positioning of the cursor, command selection and diagram editing are controlled via a mouse. The command pop-ups are invoked by clicking the right hand button and all selections are made by positioning the cursor to the appropriate spot and clicking the left mouse button. To make a change to a diagram it is only necessary to point to the area of interest and click the left mouse button. Depending on the current command this could insert or delete a program structure, move the structure to a new location, rotate the structure or attach text to a symbol via the built-in text editor. This button usage was inspired by and is consistent with the mouse used on Sun Workstations **[SUNP85]**.

Since both text and graphical systems are to be supported the data structures representing the individual Cgraph symbols had to be designed so as to be independent of the display. Further restriction were encountered since the original implementation was done on an 8088 microprocessor based system; the code and static variables had to occupy less than 64K of memory, dynamic memory was limited to around 512K and the speed of the processor was limited. Several options were explored.

Cview could have been written as a traditional graphics oriented CAD package with drawing/editing commands, windowing, viewporting and scrolling, This would allow the designer to zoom away from the diagram to get an overview of the work then expand the section of interest and continue working. The problem with

this is two fold. First, if the display is using ASCII characters to represent the Cview diagram, zooming away to an overview produces a representational problem - the character set is not rich enough and the screen is too course grained to give a good picture of large diagrams. Second, the time required to do the calculations to clip and map the diagram to the screen could degrade the response time, especially given the target processor.

After some consideration it was decided to use the standard flowcharting form as a metaphor for the Cview system. The screen represents one page of a flowcharting form. The screen, like the paper form, is divided into boxes where symbols can be drawn. This has the advantage in that there are no clipping and mapping calculations to do and the symbols always have a constant size. This eliminates the two problems with the previous method. The disadvantage is that it is difficult to obtain an overview of the complete diagram since it may be scattered over several pages.

The data structures were designed keeping this metaphor in mind. The Cview system had to be able to manipulate symbols that were placed on a two-dimensional grid that was mapped on to the screen. The data that represented the individual Cgraph symbols had to contain the following items:

1) The symbol type (decision, process, flowline, etc.).

2) The side(s) where flowline connected to the symbol.

3) The direction of the flowline(s) connected to the symbol.

4) If the symbol is a decision that forms a loop, is this a WHILE, FOR or DO-WHILE?

5) If the symbol is a decision or a fork, on which sides are the exit indicators (T/F or P/C) to be placed?

6) If the symbol is a flowline that forms part of a loop, is recursive substitution to be allowed on that symbol? (No substitutions allowed before a decision in a prechecked loop for example.)

7) If the symbol is a process or a switch, does that symbol own a subscreen (page)?

The above listed information was assigned binary values and placed in two bytes of data. The layout of these two bytes is detailed in Figure 3.2.2. The one extra bit was used to indicate that the Cview diagram was either a main line or a function. This bit is associated with the terminal symbol.

These two bytes can be used to describe the type, the connects to neighboring symbols and the flow of control through a symbol. The extra data that must be associated with these two bytes are: a pointer to text and a pointer to an owned page (process and switch). This gives the data structure in Figure 3.2.3.

DECISION_DIRECTION:

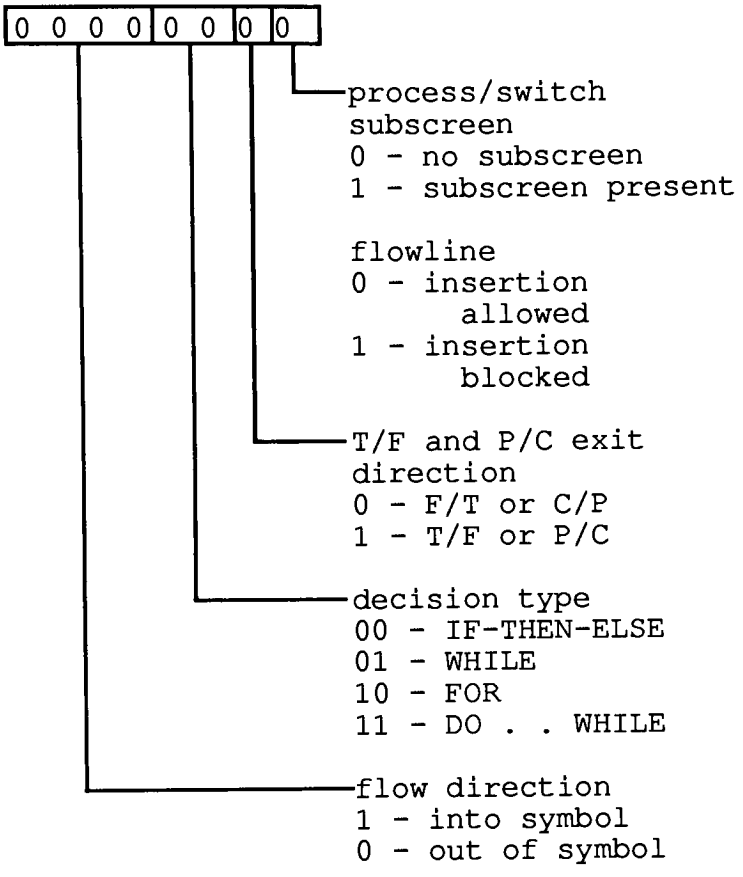


Figure 3.2.2 (cont.)

SYMBOL_CONNECTIONS:

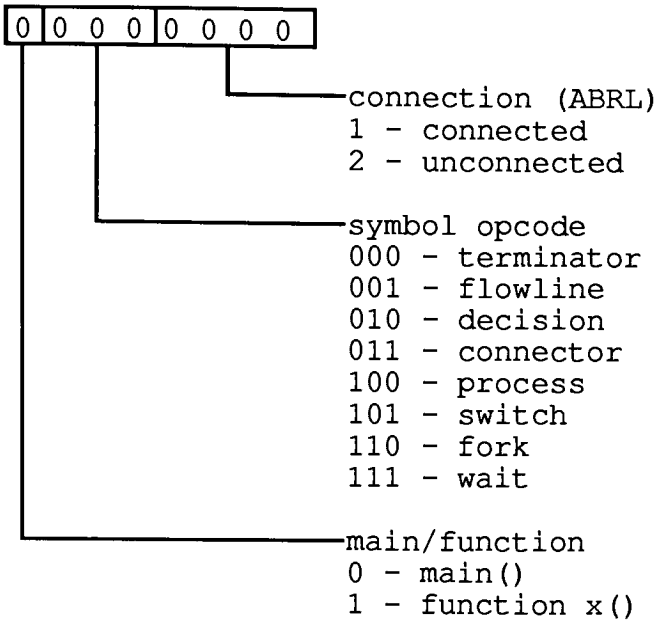


Figure 3.2.2

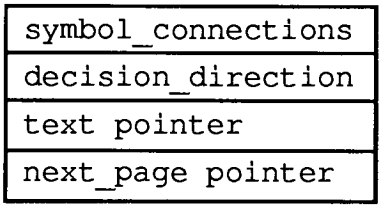


Figure 3.2.3

This data structure maps to the screen in the following way. Each symbol is represented by a 3X3 grid. The center of the grid is the symbol while the cells adjacent to it in the Manhattan directions are used for the connections. The diagonal cells are used for the T/F or P/C exit designators for decisions and forks.

Organizing these data structures so they represent a page efficiently becomes the next problem. To ignore the metaphor described above seemed foolish since the mapping from a two-dimensional array to a two-dimensional screen was so easy and

obvious (Figure 3.2.4). The first thought was to declare each page to be a two dimensional array of the structure described above. However, after drawing a few preliminary diagrams and looking at sample flowcharts (a good approximation of a Cview diagram) it was decided that too many cells would be unused and would thus waste memory. Retreating to the other extreme, dynamically allocate each symbol structure, made the data structure to screen mapping potentially difficult, but provided efficient use of memory. The middle road of using either a sparse matrix or a matrix of pointers was investigated.

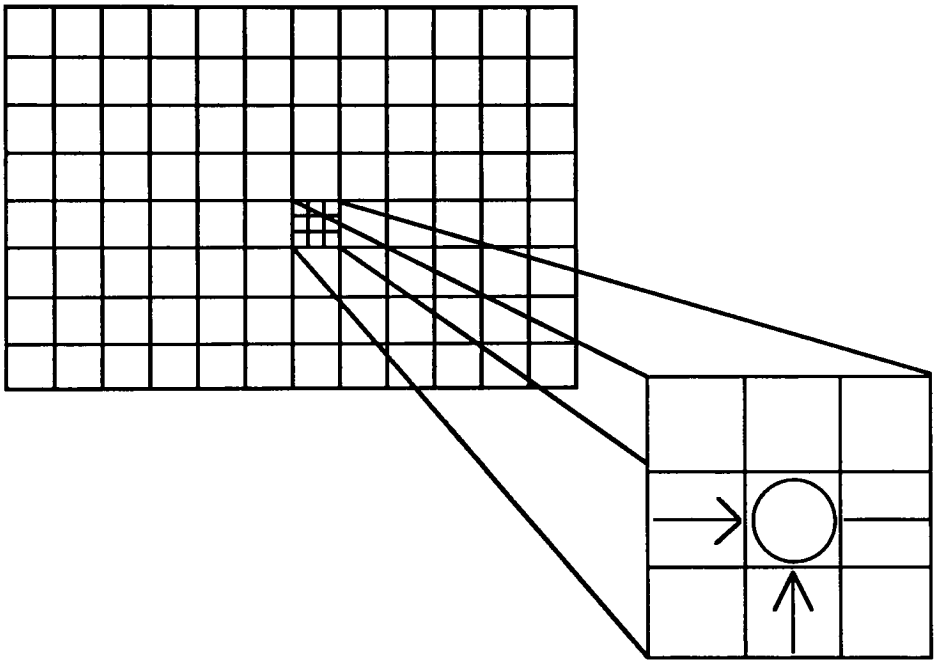
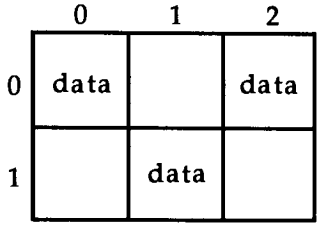


Figure 3.2.4

Given a standard text oriented video screen (24 lines X 80 characters) and a Cview symbol composed of a 3X3 character cell, the dimensions of the matrix are 8X26. Assuming that the pointers consume four bytes of storage, each symbol requires ten bytes of memory (eight bytes in pointers and two in binary coded information.) To represent a mXn sparse matrix would require $m+n+1$ row/column header nodes plus the nodes that contain the actual data. The data nodes contain not

only the data structure detailed in (Figure 3.2.4) but row,column identification numbers and pointers to the next nodes; a total of 16 bytes (Figure 3.2.5). If each header node requires twelve bytes, the storage overhead for one empty matrix is 424 bytes. Added to this is the extra 16 bytes for each array element.

Original matrix:



Sparse matrix representation:

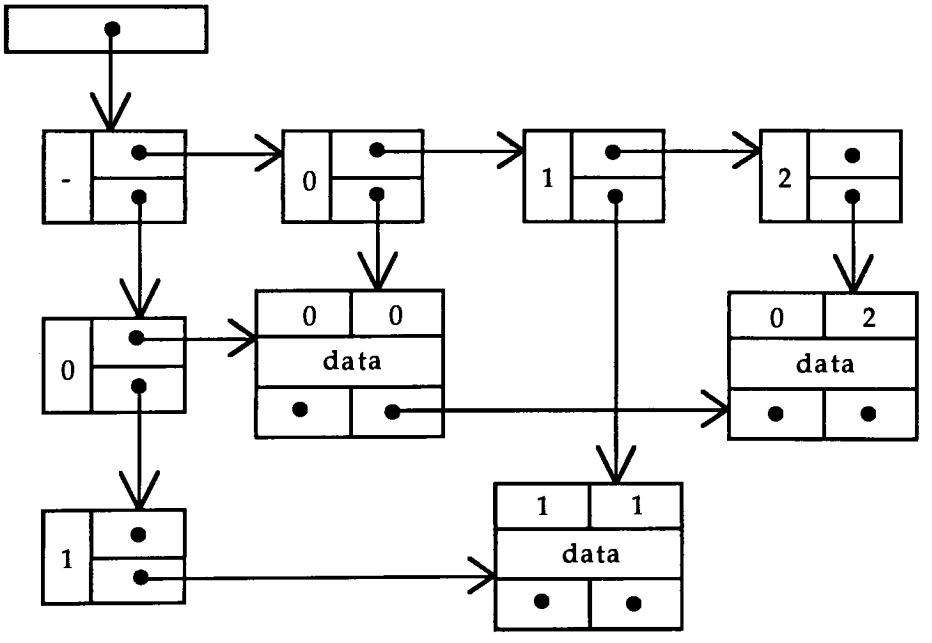


Figure 3.2.5

Using an 8X26 dense array of pointers exacts a larger initial overhead, but there is a savings of 16 bytes per element since the row,column address is known and the elements are not linked. Figure nnn indicates that if more than 12% of the

elements are occupied, the matrix of pointers uses memory more efficiently than the sparse matrix (Figure 3.2.6).

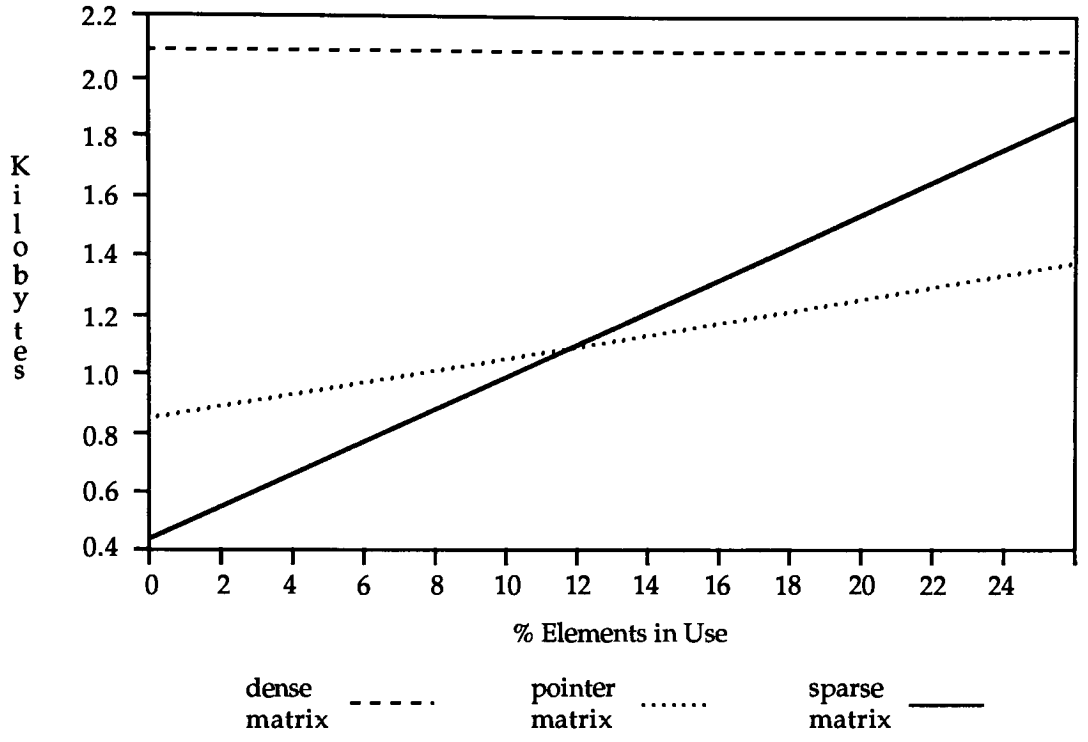


Figure 3.2.6

Having settled on the use of a dense matrix of pointers to represent a page, the question of overall structure had to be answered. Since the Cview diagram is represented positionally within the matrix the starting row,column, or entry point, must be stored with the page. A page identification number is required for the file system and a page status byte was required for the proper operation of delete command. (For a full description of how these fields are used please see the File or Edit sections.) Since pages must be linked together a next and previous pointer is included. These fields constitute the full representation of a Cview page (Figure 3.2.7.)

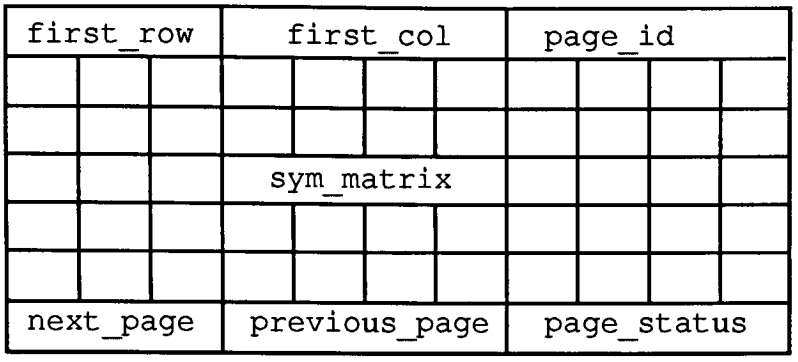


Figure 3.2.7

Pages can be forward linked in three ways. Since a process can represent an arbitrary amount of source code, the process symbol can own a subpage which contains a detailed section of the diagram. If this diagram is too large to be held on one page, a process on the subpage would own an other page with another detailed section of the diagram, ad infinitum. A switch symbol is somewhat different. It owns a linked list of pages each of which represents one of its cases. The terminal symbol on the root page which starts the diagram can own a page. This represents a separate Cview diagram. This allows one Cview diagram to represent a main program and a set of functions or a collection of functions.

Back linking is somewhat simpler since there is no reason to back link to an individual symbol. Pages back link to the owner page regardless of how the forward linking was done. The reason lies in the metaphor that the system is based upon. A process/switch/terminal can own a subpage which can be displayed on the screen. Returning from the subpage does not return to the symbol - the return is to the page, which must be redisplayed.

To summarize, the final form of the data structures that comprise a Cview diagram are best described as a linked list of mXn-ary trees of pages. Every page can theoretically (but not practically) own mXn subpages. Switch statements own a

degenerate tree of subpages unless a case owns subpages (Figure 3.2.8.) The root page is the first page in a linked list of trees if more that one function is described by the diagram (each function is a separate tree.)

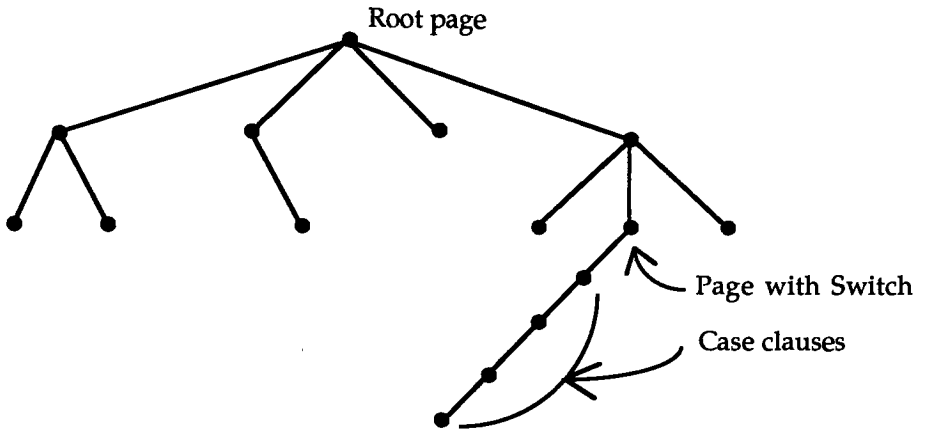


Figure 3.2.8

To allow Cview to generate C source code a simple text editor was designed that allows the user to add or attach text to symbols. The text pointer in the symbol points to the data structures in Figure 3.2.9. Since the amount of text associated with one symbol is minimal, the editor was designed to manipulate precisely one screen full of text. The exact number of lines is determined by the max_lines field of the root structure. This essentially is the maximum number of text lines on the screen. The text editor adds, deletes or modifies the contents of the linked list according to the keystrokes entered on the keyboard. (See the Symbol Command section for a full description of the editor.)

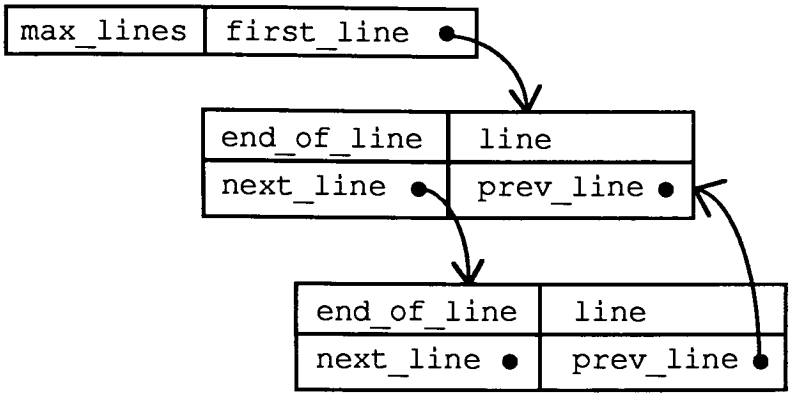


Figure 3.2.9

The last design consideration is more general. There is a fair amount of information that must be shared by many functions spread across different source files. Items in this category are: root page pointer, current page pointer, current mouse coordinates, last mouse coordinates, current file name, current command, flags which control the display, etc. To write the code with all of these fields defined as global would be both difficult and considered poor style. To alleviate this problem Cview was designed so that all of the control variables are static global in one function only. Small functions are called which either set or return the value of the variable. This allows each Cview file to access the control variables without any apparent global usage.

3.3 The Menu System

The Cview menu system is in actuality coded as two unique systems. One system is the menu handler for selecting commands and the second is a dialog manager that processes the more complex pop-up menu. These two systems were designed at different times during the development of Cview and in fact behave in very different ways despite internal similarities.

3.3.1 Menus

The command menus are displayed when the right hand mouse button is clicked. To minimize the amount of arm movement required to select a command the top-left corner (origin) of the menu is always positioned at the current cursor position. If the width or length of the menu is such that it would extend off of the screen, the origin is adjusted so that the complete menu is displayed. To select a command, the cursor is moved until it is on the same line as the command and within the bounds of the menu. Clicking the left button accepts the selected command. Clicking either button with the cursor on or outside the menu deselects it. Once a menu is deselected, either by selecting a command or clicking outside the menu it is removed from the screen. It should be noted that it is not necessary to hold down the right button to keep the menu displayed - the menu is displayed until it is deselected.

The data structure in Figure 3.3.1 describes a menu. The size of the menu includes the border drawn around the commands. Up to 20 commands can be included in a menu. Each command is a character pointer and a pointer to an integer function. The exact processing goes as follows. If the current cursor position will not allow the whole menu to fit on the screen readjust the origin of the menu. Using the

size parameters stored in the menu data structure and the current mouse position, save the contents of the video buffer where the menu is to be displayed. Display the menu and wait for a mouse button to be clicked. If the cursor is outside the menu restore the save area of the video buffer; this removes the menu from the screen. If the cursor is in the menu calculate the relative line number of the command and use this as an index to the array of commands. Execute the function that is stored in this array element passing along the current mouse position. The function either is a routine that sets the current command in the Cview environment or displays the next menu. These functions will be referred to as "call-backs". As a notation convention any command that invokes a submenu will have a "=>" next to it. This is notation used in the SunView environment for what in that system are called "walking menus" **[SUNP85]**.

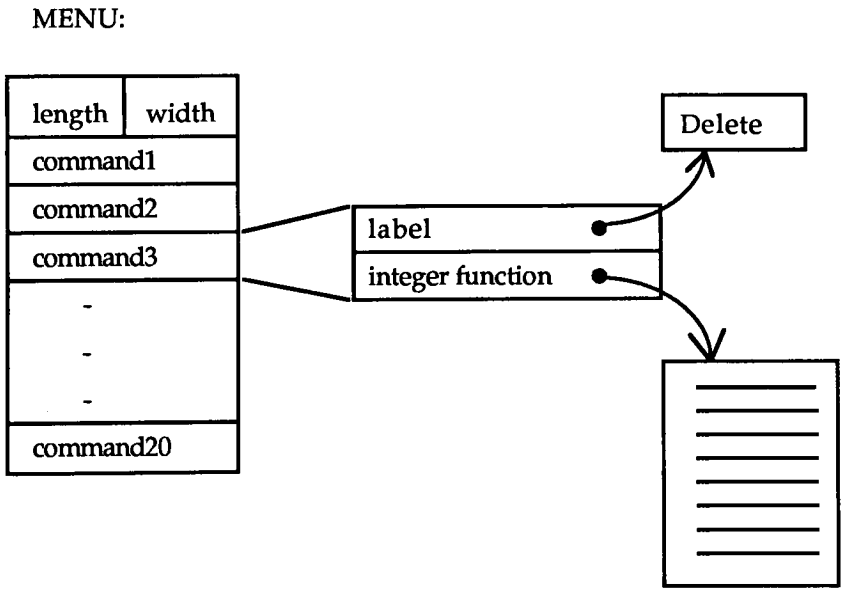


Figure 3.3.1

3.3.2 Pop-ups

Pop-ups are more complex. A pop-up box can contain three different items - text messages, fill-in the blank (or default) fields and buttons. The text messages are used for titles, labels, status information or explanations. Fill-in items are used for values that do not have a fixed value or have a range of values such as screen positions, file names, etc. Push buttons are used to toggle boolean flags and indicate the completion of a pop-ups processing by the user. The data structure and pop-up diagrammed in Figure 3.3.2 are typical for the Cview system.

A pop-up is declared as an initialized static structure in the function that uses it. The code can change the value of the various fields if the pop-up is context sensitive. An example of this is the "Save" pop-up. The format for the "Save", "Save as" and "Load" pop-ups differ only in the title. Depending on which command is selected the code places the appropriate message in the pop-up before displaying it.

The display and processing of a pop-up works as follows. The code that uses the pop-up would set any context sensitive fields then call the display manager. The display manager centers the pop-up on the screen first drawing the border then placing the messages, buttons and fill-in items. The display manager would then wait for a left-button click. When a click is detected, the position of the cursor relative to the start of the pop-up is calculated. This position is compared against the positions of the fill-ins and buttons. A button or fill-in is activated by clicking anywhere within the box that encloses it - this includes the fill-in label. If there is a match the appropriate item manager is executed.

Pop-up:

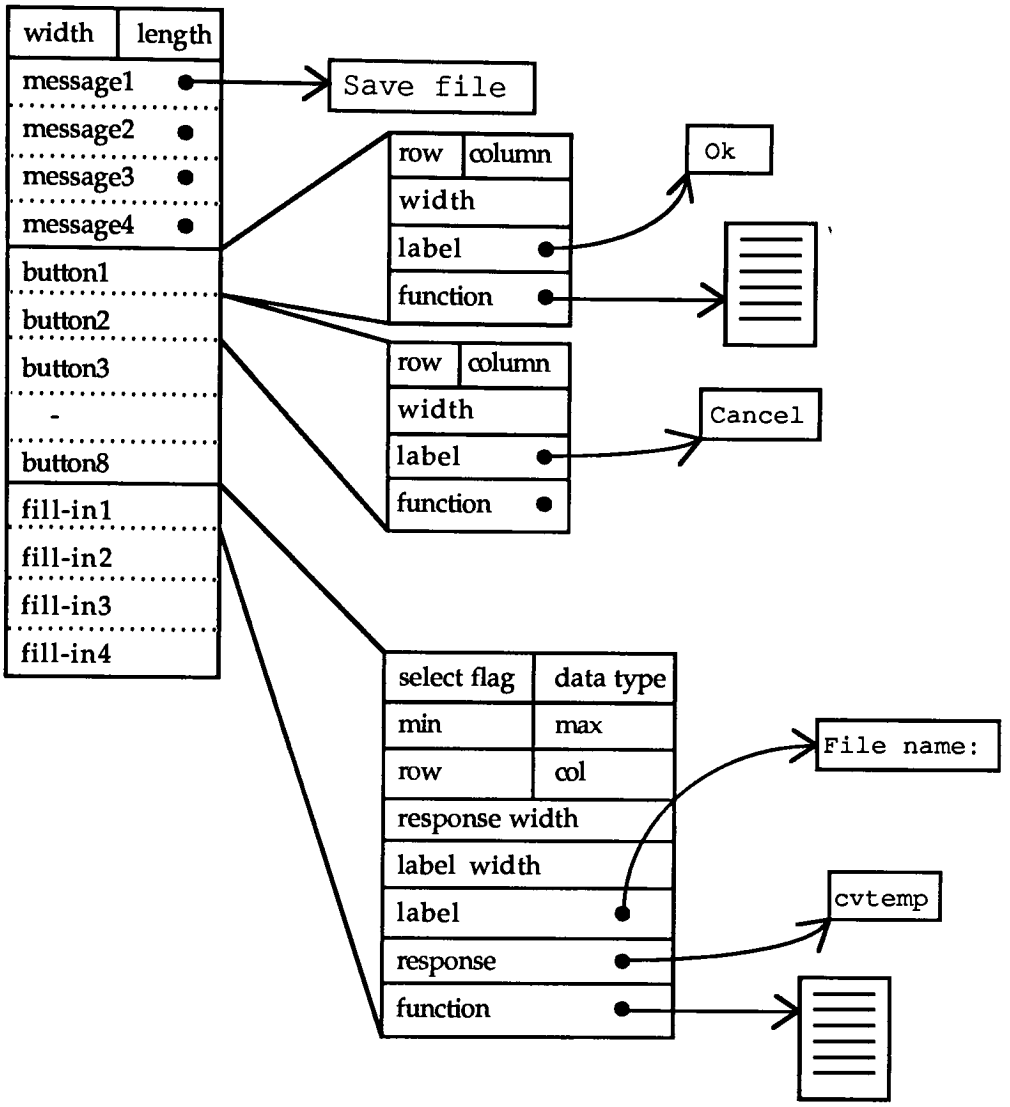


Figure 3.3.2

If the item was a fill-in the response field is cleared and the cursor changed to a text cursor for keyboard entry. After the item has been entered the manager checks the data type (either character or numeric) and if the data is numeric checks the entered value against the specified range. If the range is incorrect the field is cleared and the data must be reentered. Once the input is correctly entered the selected flag is set indicating that the fill-in item has been modified.

If a button was pushed the button manger first checks to see if the call-back pointer is NULL. If this is a NULL call-back, no action is taken - the pop-up is simply removed from the screen. This is how the cancel button or an Ok response to a message is handled. Otherwise the manager scans the array of fill-in items to see if there were any modifications. If a fill-in was modified the associated function is executed (usually changing the Cview environment) and the selection flag reset. Once the fill-in array has been scanned the button's call-back is executed. Pushing any button indicates the completion of a pop-up.

3.4 Primary Commands

The primary command menu is displayed by clicking the right mouse button. It contains the commands Symbols, Edit, Generate C, File, Options and Quit. Symbols, Edit and File have submenus while the rest of the commands execute immediately. The submenued commands and the Generate C command are described in detail in other sections - this section describes the function of the Options and Quit commands.

3.4.1 Options

Options displays the pop-up in Figure 3.4.1 which allows the user to change the behavior of the mouse and the way in which Cview diagrams are displayed and the way the abstract command selects program structures.

A clicking or beeping sound was included in the mouse functions so that the user had a audio verification of button clicks. This turned out to be important since some mice have nearly silent button operation and Cview itself does not complain visually or audibly if a particular operation is not possible. The tone and duration of the click can be varied by selecting the appropriate fill-in and changing the value. The debounce value is used as the upper limit for an empty FOR loop. This delay loop was required since a "heavy finger" on the mouse button could easily cause multiple command executions. Selecting the "Click ON" button toggles the click status to OFF for silent operation.

The titles referred to are the text that is attached to the first symbol of a Cview page. By default the text on the start or first terminal symbol is either "main()" or "function user_funciton()". If there is more than one line of text, only the first line is visible. The user can use the Text command to browse through these lines one at a time without going to the editor. The titles can be moved anywhere on the screen or turned off by selecting the appropriate pop-up item.

The show exits and grid buttons control the display of the Cview diagrams. If the user does not need to see the T/F or P/C exit indicators she/he can suppress their display. This reduces the clutter on the screen and decreases the drawing time slightly. If grid is selected the unused portions (the null array elements) of the screen are filled with characters that represent the corners of the 3X3 symbol cells. This

makes inserting or moving structures easier since the exact symbol positions can be seen. The penalty paid is much slower screen refresh since each symbol position on the screen must be drawn.

Selecting the abstract symbol button changes the behavior of the edit command abstract. For a full discussion of this command and the different processing modes please see the Edit Commands section.

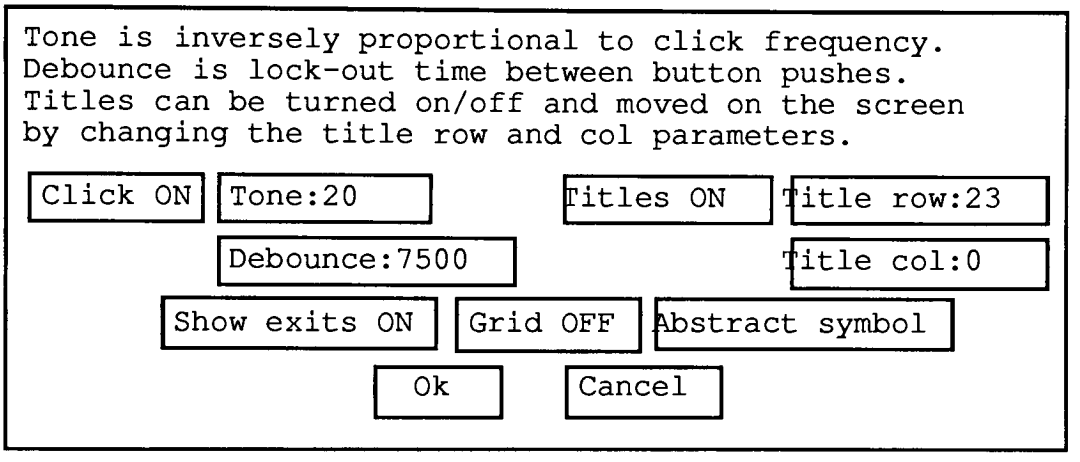


Figure 3.4.1

3.4.2 Quit

The Quit command exits form Cview. Before the program exits however it checks to see if the current Cview diagram has been modified. If no modifications have taken place the program terminates otherwise the file save command is called. Cancelling the file save pop-up causes Cview to exit without saving the changes made to the diagram.

3.5 Symbol Commands

The symbol commands, selected from the symbol submenu, are the primary Cview drawing tools. There is one command for each type of Cview structure or symbol plus the Text command which allows conditional expressions or procedural code to be attached to symbols. When one of these commands is selected the mouse cursor changes to reflect the current command. In the current text mode implementation the cursor changes to the first letter of the command except for parallel which looks like - ||. This is an extended ASCII character allowed by the MS-DOS operating system. In graphics mode it would be possible to change the cursor shape to reflect the current command.

Starting with a new null diagram, one would select the highest level structure from the menu and insert it between the terminals. This is accomplished by moving the mouse cursor to any point on the flowline between the two terminals (a logical path) and clicking the left mouse button. Cview checks the distance between the terminals to verify that there is enough space to draw the symbol/structure. Individual symbols (PROCESS and SWITCH) only take one space but IF-THEN-ELSE, LOOP and PARALLEL require three spaces vertically and three spaces horizontally. If space exists for the symbol/structure, Cview will draw it using the maximum available space. The user is not responsible for drawing individual symbols in program structures. Depending on the direction of the flowline Cview will determine where to place the DECISION/FORK and CONNECTORS then connect them with flowlines. The next symbol/structure is picked from the menu (if necessary) and inserted recursively along one of the logical paths.

3.5.1 Command Samples

While this is easy for the user to do, the processing details are complex. A short example will be used to examine the internals of this process. Suppose the user wanted to diagram the pseudocode in Figure 3.5.1.

```
c = 0;
b = 0;
WHILE NOT(eof) DO
BEGIN
      c = c + 1;
      if (x = ' ') then b = b + 1;
NIGEB;
print b/c*100;
```

Figure 3.5.1

The following sequence of events would be required.

Cview starts with a null flowgraph the length of the display. To insert a symbol/structure one clicks the right mouse button, positions the cursor anywhere on the Symbols => line then clicks the left mouse button. Since the highest level structure in the pseudocode is a WHILE loop, moving the cursor to Loop and clicking the left button again changes the cursor to "L" (Figure 3.5.2.)

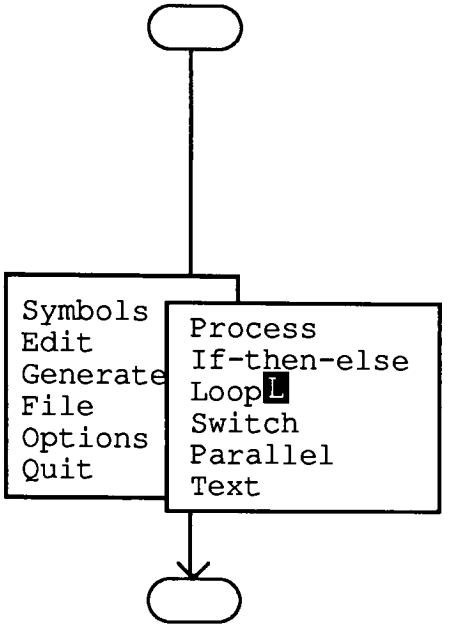


Figure 3.5.2

Positioning the cursor on the flowline between the two terminal symbols and clicking the left mouse button again causes a pop-up to be displayed. Since there are three loop structures defined in the C language the user has to pick which one is to be used. (Figure 3.5.3) In this case clicking the left button over the while causes a WHILE loop to be inserted between the terminals.

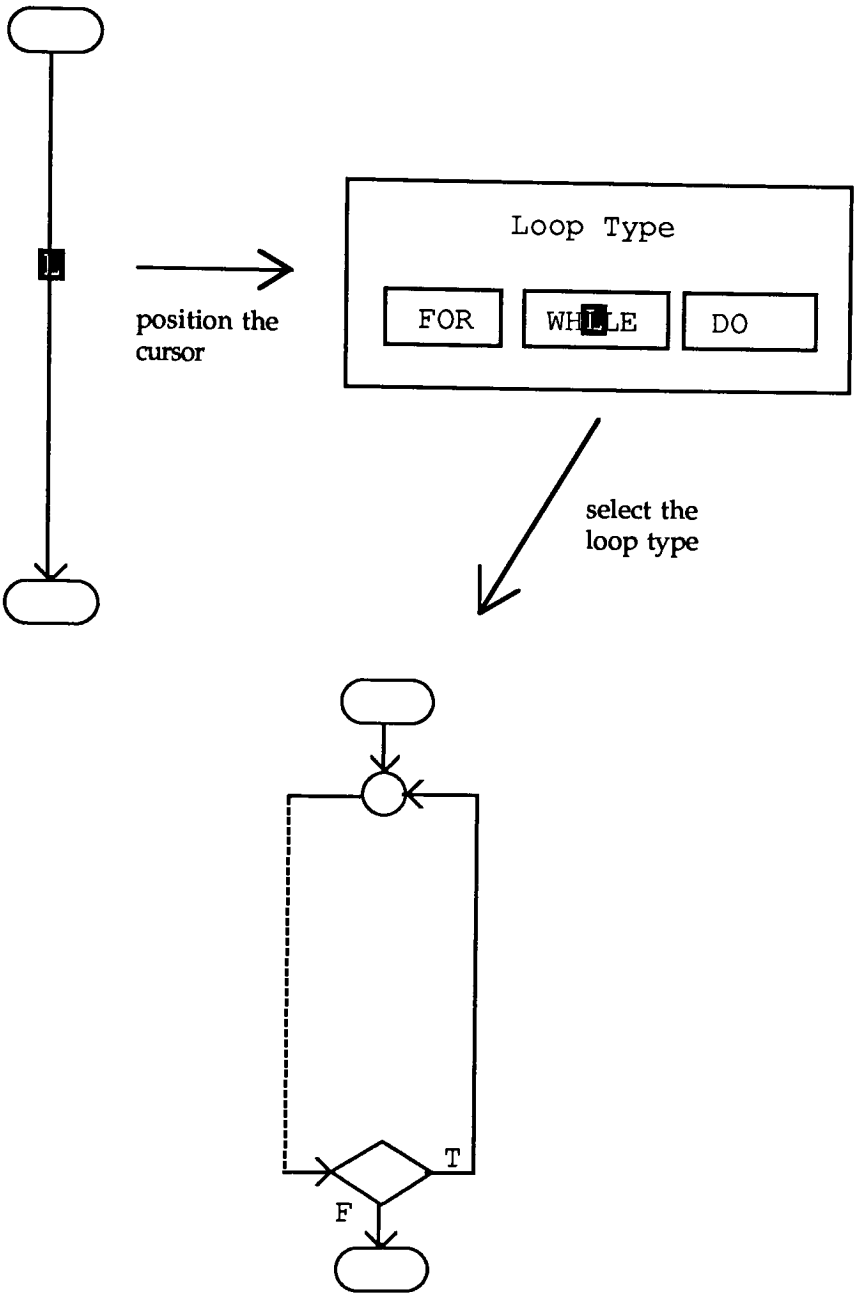


Figure 3.5.3

The insertion process has to make sure that placing a loop along this flowline will not cause a collision with an existing structure. To accomplish this, Cview checks the rectangular area (Figure 3.5.4) that will be occupied by the loop in the following ways:

1) Starting at the cursor position Cview traces up and down the flowline to the first non-flowline symbols. If the difference between the non-flowline row numbers is less than three the insertion is abandoned.

2) If there is enough space vertically, the columns on either side of the flowline are checked for collisions. Since the Cview represents the diagram as an array of pointers, this check consists of looking for non-NULL pointers in the columns. If a non-NULL pointer is found the insertion is abandoned.

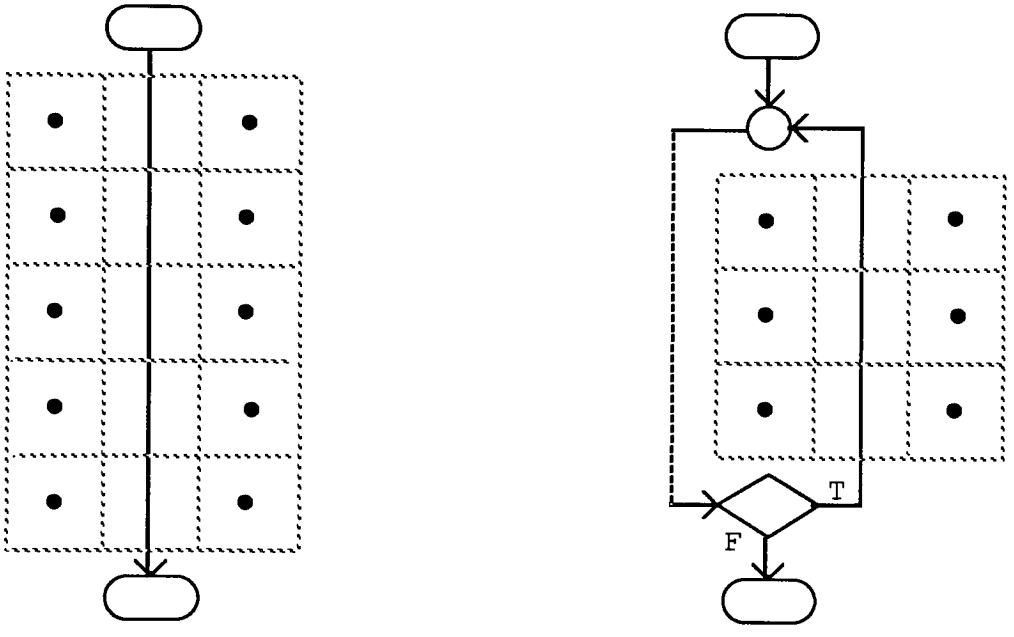


Figure 3.5.4

The direction of the flowline is determined by inspecting the direction bits in the flowline symbol that was selected. The decision and connector are placed in the rows adjacent to the terminals according to the direction of the flowlines (connector occurs first in a loop.) The original flowlines that are now between the decision and connector symbols are removed and the flowlines that constitute the loop are added. The dashed flowline represents a "blocked" or unusable logical path. Since a WHILE loop is pre-checked it is not permissible to insert symbols or structures before the

decision. This would also be true of a FOR loop - a DO-WHILE would have the opposite flowline blocked.

All of these manipulations take place on the array of pointers. The addition of the decision and connectors is accomplished by modifying the already existent flowline symbols. The flowline symbols that are apparently removed, in actuality are moved one column (either left or right depending on the direction of flow) and become part of the loop. The rest of the flowlines must have symbol structures allocated and initialized to the appropriate values. Once this is accomplished the Cview drawing routines are called to refresh the screen. In order to avoid refreshing the complete screen, the extent of the modification (starting row,column - ending row,column) is passed to the drawing routine and only those positions on the screen are redrawn.

The next structure to be inserted is the IF-THEN-ELSE. The procedure described above to select a structure is repeated. The cursor changes to "I" indicating that IF-THEN-ELSE is the current command. Clicking on the unblocked flowline (Figure 3.5.5) invokes the same checking procedure used for the WHILE loop. If space exists for the structure the matrix is modified and the corresponding area of the screen is updated.

The process that represents the THEN clause of the IF can be inserted by selecting the process command and clicking on the flowline on the TRUE side of the decision (Figure 3.5.6). The insertion of processes and switches do not require the extensive area checking required of the Cview structures. The only requirement for the insertion of either of these symbols is that they replace a flowline that is vertically oriented. Insertion on a corner or a horizontal flowline is not allowed.

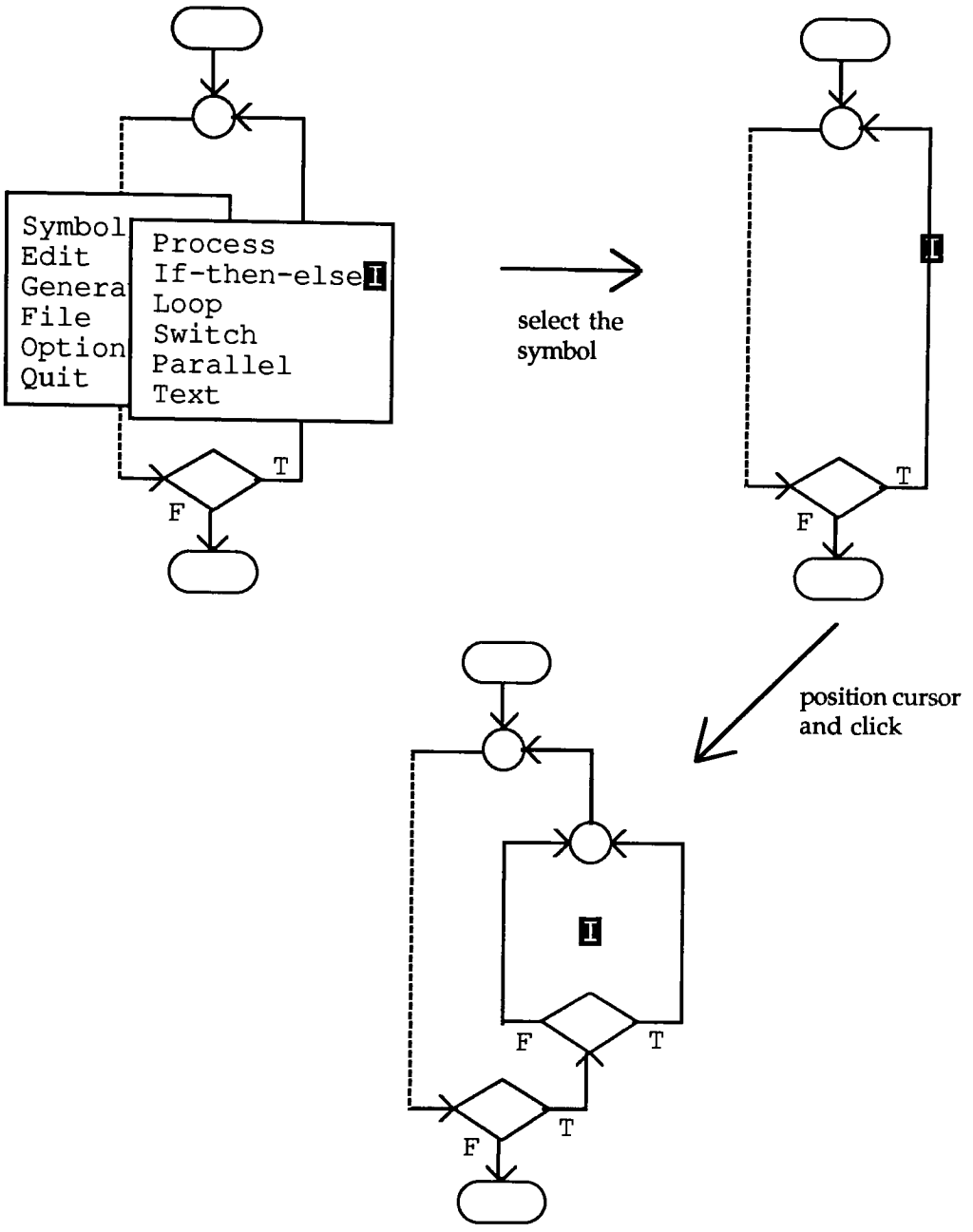


Figure 3.5.5

At this juncture both the program structures and the THEN clause are diagrammed but the full vertical extent of the page (as defined by these Figures) has been exhausted. Since Cview initializes the null flowgraph to be the full length of the matrix, and hence the screen, the processes that represent the other procedural

lines of code cannot be inserted. This problem is resolved in the next section which discusses the editing commands.

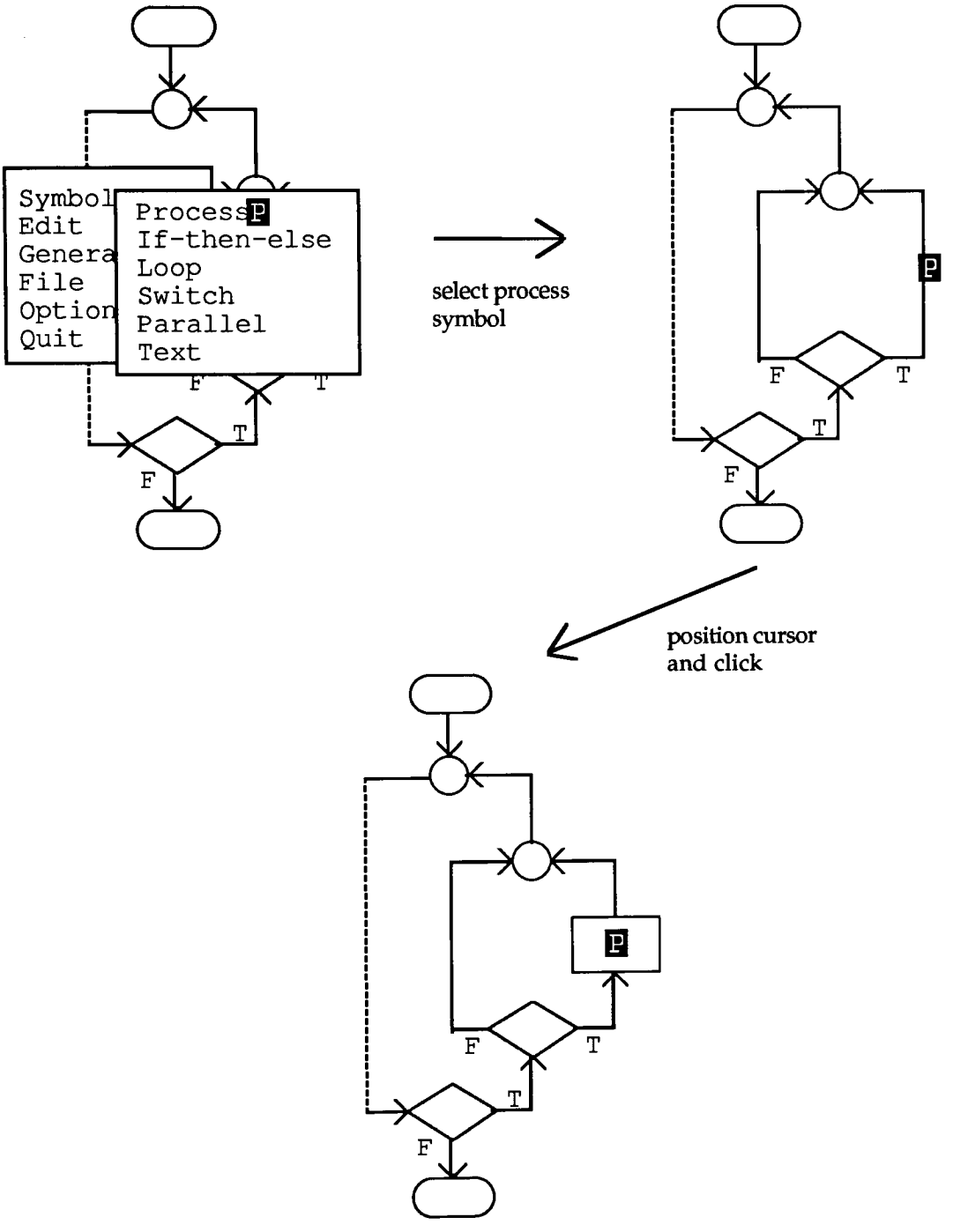


Figure 3.5.6

3.5.2 Parallel

The parallel command makes it possible to write multitasking programs for the UNIX environment. The parallel command inserts a FORK/WAIT which behaves internally like an IF-THEN-ELSE. In fact the C source code generated is actually an IF-THEN-ELSE that checks the process identification number returned by the *fork* UNIX system call. If the process identification number is not zero the THEN or PARENT logical path is executed otherwise ths copy of the task is the CHILD and the ELSE logical path is taken. For an example of a program that has parallel processing see appendix H.

3.5.3 Text

The Cview Text command invokes a simple full-screen editor that can be used to modify text that has been associated with a Cgraph symbol. To edit a symbol one chooses Text from the menu then positions the cursor and clicks the left mouse button. If the symbol is not allowed to carry text Cview does nothing otherwise the screen is cleared, existing text (if any) is displayed and the cursor is positioned at the top left corner of the screen.

The data structures used by the editor are detailed in (Figure 3.2.9). The text pointer in the Cgraph symbol points to the structure that contains the maximum number of lines to be edited (the screen length) and a pointer to a line of text. In order to conserve memory lines of text are kept as a doubly linked list. This approach was used instead of video buffer images since most of the buffer would be unused. Individual lines consist of the two pointers for the list a pointer to the text and the number of characters currently in the line.

If the text pointer in the Cgraph symbol is NULL the text editor allocates the root node and the first line in the list, setting all pointers to NULL and the end_of_line field to zero. An internal pointer which represents the current line is also initialized to the first line node. In order to maintain a reasonable response time the editor places characters directly in the video buffer instead of in the linked list. The current line pointer is used to update the end_of_line field only. When the user exits the editor the contents of the video buffer are moved into each node of the linked list using the end_of_line count as an index to the video buffer.

As the user types, the editor accepts the keystrokes and performs according to the following rules. If the keystroke is a:

HOME	Move the cursor to the top-left corner of the screen and set the internal pointer to the first node in the linked list.
END	Move the cursor to the end of the current line on the screen (controlled by the end_of_line field).
CURSORS	For left and right cursor move the cursor on the screen one character in the appropriate direction. Moving backward from column one will place the cursor at the end of the previous line, moving forward off the end of a line will place the cursor at the beginning of the same line. For cursor movement up the screen, the cursor will remain in the same column or the end of the previous line if the cursor would have been placed past the end of the previous line. The

current line pointer is updated to match the current line on the screen. No movement takes place is the current line is the first line. Cursor movement down the screen functions the same way except that if the current line pointer is positioned at the end of the list, moving the cursor down allocated and initialized a new line node. If the cursor is on the last line of the screen no action takes place.

CR — If the cursor is in column one insert a line ahead of the current line and make it the current line. If the cursor is not in column one, insert a line after the current line. The cursor is left on the new line. If the screen is full, as determined by the max_lines field, no action is taken.

BACKSPACE — If the cursor is not in column one delete the current character, collapse the line and decrease the end_of_line counter by one. If the cursor is in column one remove the line from the list and make the next line the current line. If the cursor is at the home position no action is taken.

DELETE — Delete the current character and collapse the line by one character. The cursor stays in the same column position. This is character delete as opposed to backspace erase.

CONTROL — Affects the function of the left and right cursor keys. Control-left moves the cursor to the beginning of the previous non-blank field or wraps from the beginning of a line to the end of a line. Control-right moves to the beginning of

	the next non-blank field or to the end of the line.
INSERT	Toggles the editor from "insert at current cursor position" to overstrike mode.
ESC	Exits from the editor. The user is given a choice of exiting and saving the changes, exiting without saving or returning to the editor.

All other keystrokes are accepted as is and placed in the video buffer at the current cursor position.

If text already exists when the editor is entered a copy of the list is made then the text is displayed on the screen. When the user exits the editor, if the session is to be aborted, the old list is reinstated by placing its pointer in the root structure (first_line) then the modified list is freed. Otherwise the original list is freed and the contents of the video buffer is copied to the current list.

3.6 Edit Commands

The edit commands are invoked just as the symbol are, from the primary menu. These commands are used to move, delete, rotate, change exit directions from decisions and forks, migrate portions of diagrams to new pages and navigate through pages of a diagram.

3.6.1 Move

Moving the symbols or structures of a Cview diagram requires two pieces of data. The first is the row,column extent of the items being moved along with the position of the selected symbol. The second is the target location. Once the target location is known Cview can check to see if the movement will cause any parts of the diagram to overlap. If this is detected the movement will not be allowed.

The row,column extent is determined by starting at the selected location and tracing the edge of the structures that comprise that logical path (a logical path is the body of a loop, a then clause, an else clause, a parent or a child.) This is accomplished via the state diagrams in Figure 3.6.1.

Back_track starts at the selected location and traces to the top of the logical path. The top of the path will either be the row below a terminal symbol or a flowline that changes direction from horizontal to vertical (designated as NE, NW, SE, SW corners.) Bottom_right then starts at that location and proceeds to trace the right hand side of the path. The R transition (Right, Left, Above and Below) from state 0 to state 1 is taken preferentially if the current symbol is connected in both the left and right directions. Bottom_right (and top_left) keep local variables that represent the current row and column. These are checked after each transition to see

if their values represent a new maximum value. Termination takes place when either a terminal or the opposite corner is found. In this way the right most column and bottom row (above the terminal or corner) are determined. Top_left traces the left side of the logical path keeping track of the minimum values of its row and column variables thus finding the left most column and top row (Figure 3.6.2.) These boundary values will be referred to as top_row, bottom_row, left_edge and right_edge.

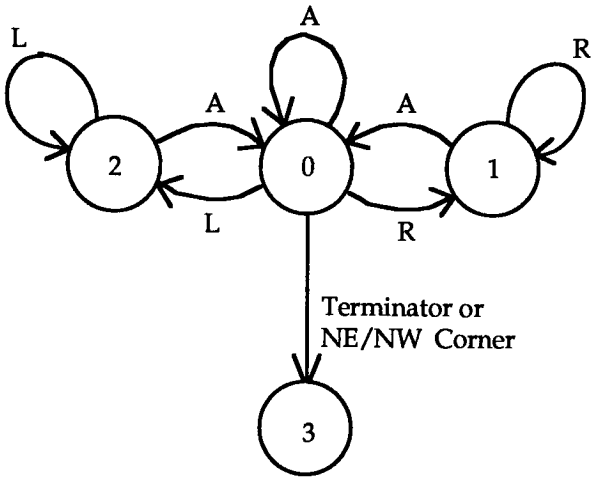
While this processing is being done, the cursor has been changed to a cross-hair indicating that the target location is to be selected. The new position is is recorded when the left button is clicked and the second part of the processing starts. The difference between the source and target columns is determined; this value will be referred to as the offset.

If the horizontal offset is not zero the area of the array to which the selected path is to be copied must be unused (all NULL pointers.) Cview checks this by inspecting each element of the array between [top_row, right_edge] to [bottom_row, right_edge+offset-1] if offset is positive and [top_row, left_edge] to [bottom_row, left_edge-offset+1] if it is negative. If this area contains non-NULL pointers the move operation terminates. If this area is clear the rows which contain the corners are checked to insure that extending the horizontal flowlines will not cause a collision with another structure. If these areas are clear, the selected area is copied to its new location. A positive offset causes the copy to take place in a right-to-left direction. Symbols, as the are copied, are replaced with NULL pointers; in this way the old location is cleared by the end of the copy operation (Figure 3.6.3). A negative offset causes the operation to take place left-to-right.

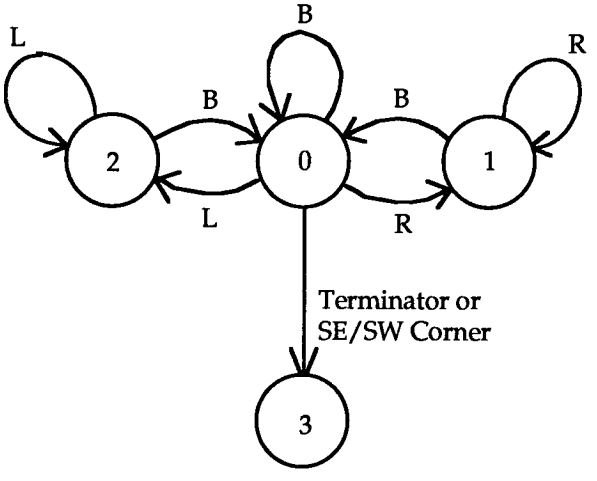
Once the area between the terminals or corners is moved, the terminals or corners themselves are moved to the target column. If the logical path was contained between corner flowlines, the flowlines are either extended or shortened as needed otherwise the terminals are moved to the target columns and processing is done.

If the vertical offset is not zero and the selected symbol is not a flowline, Cview will move the selected symbol either up or down (depending on the sign of the offset) the column. This is somewhat easier than horizontal checking since the number and configuration of symbols is limited. A terminal cannot be moved vertically - they always occupy the first and last rows of the diagram. Processes and switches can move up or down the flowline they are on either until the target row is reached or until a non-flowline or a corner symbol is found. Decision/fork and connectors can move along vertical flowline away from their matching symbols (making the structure larger) until the target row is reached or until a non-flowline or a corner is encountered. Moving toward the matching symbol (making the structure smaller) is more complex. First a search is done to determine how far each of the flowlines attached to the symbol extend horizontally. Then starting from these positions each logical path is checked for non-vertical flowlines until either a non-vertical flowline is encountered or until the target row is reached. The selected symbol along with its horizontal flowlines and corners is moved to the target row, the vertical flowline attached to the symbol is extended to the new symbol position and the excess vertical flowlines on both the logical paths are removed (Figure 3.6.4.)

back_track:



bottom_right:



top_left:

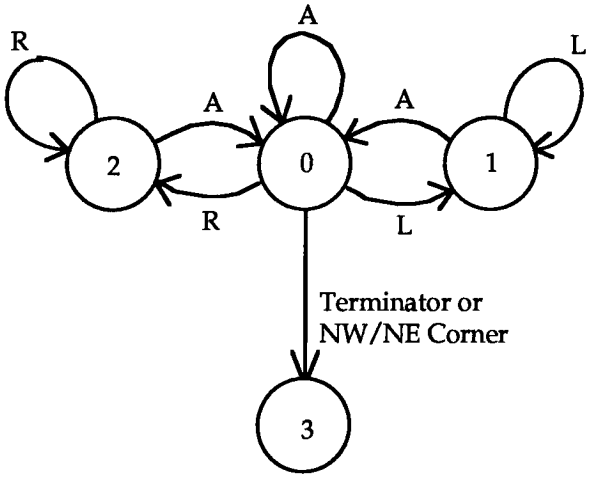
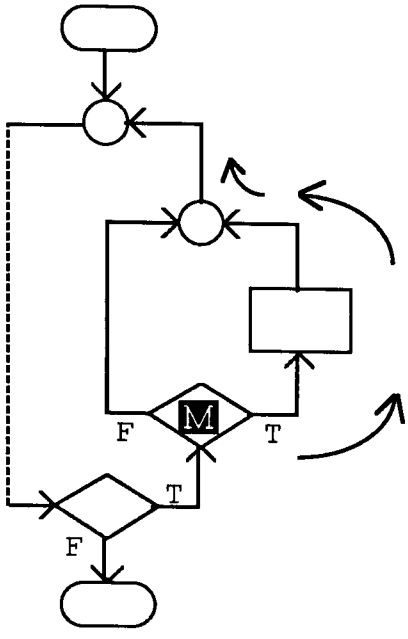
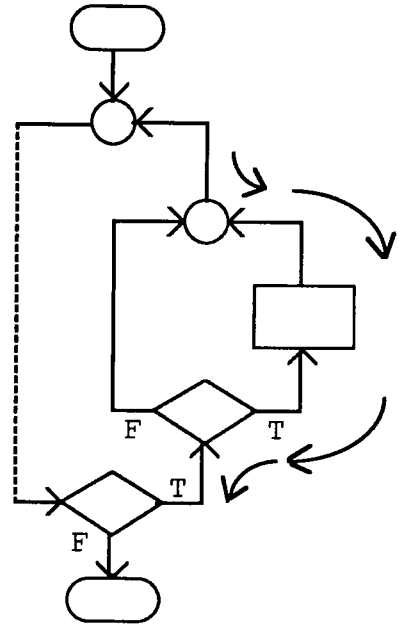
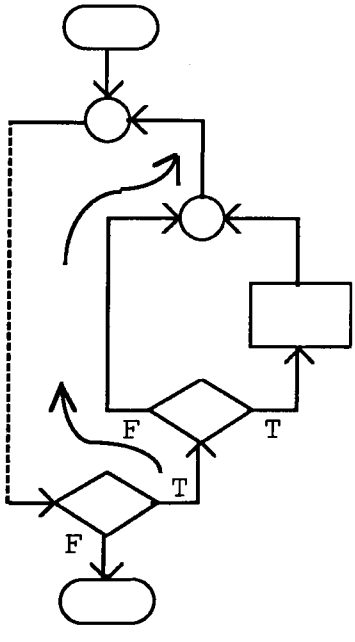


Figure 3.6.1





top_left



area to move

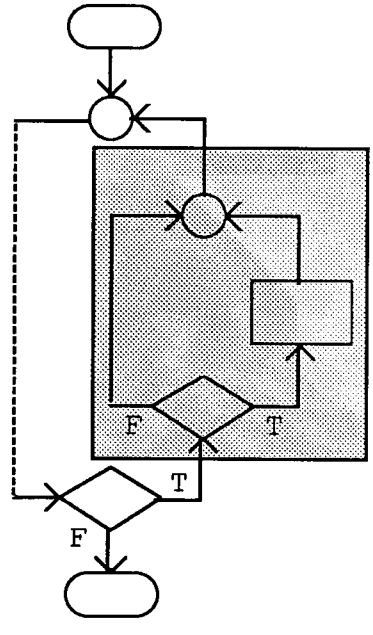


Figure 3.6.2

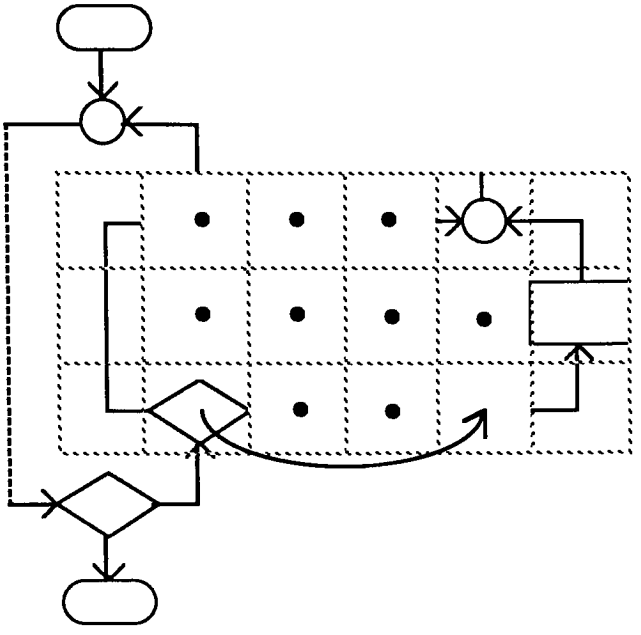


Figure 3.6.3 Copy proceeds right-to-left with symbols replaced by NULL pointers.

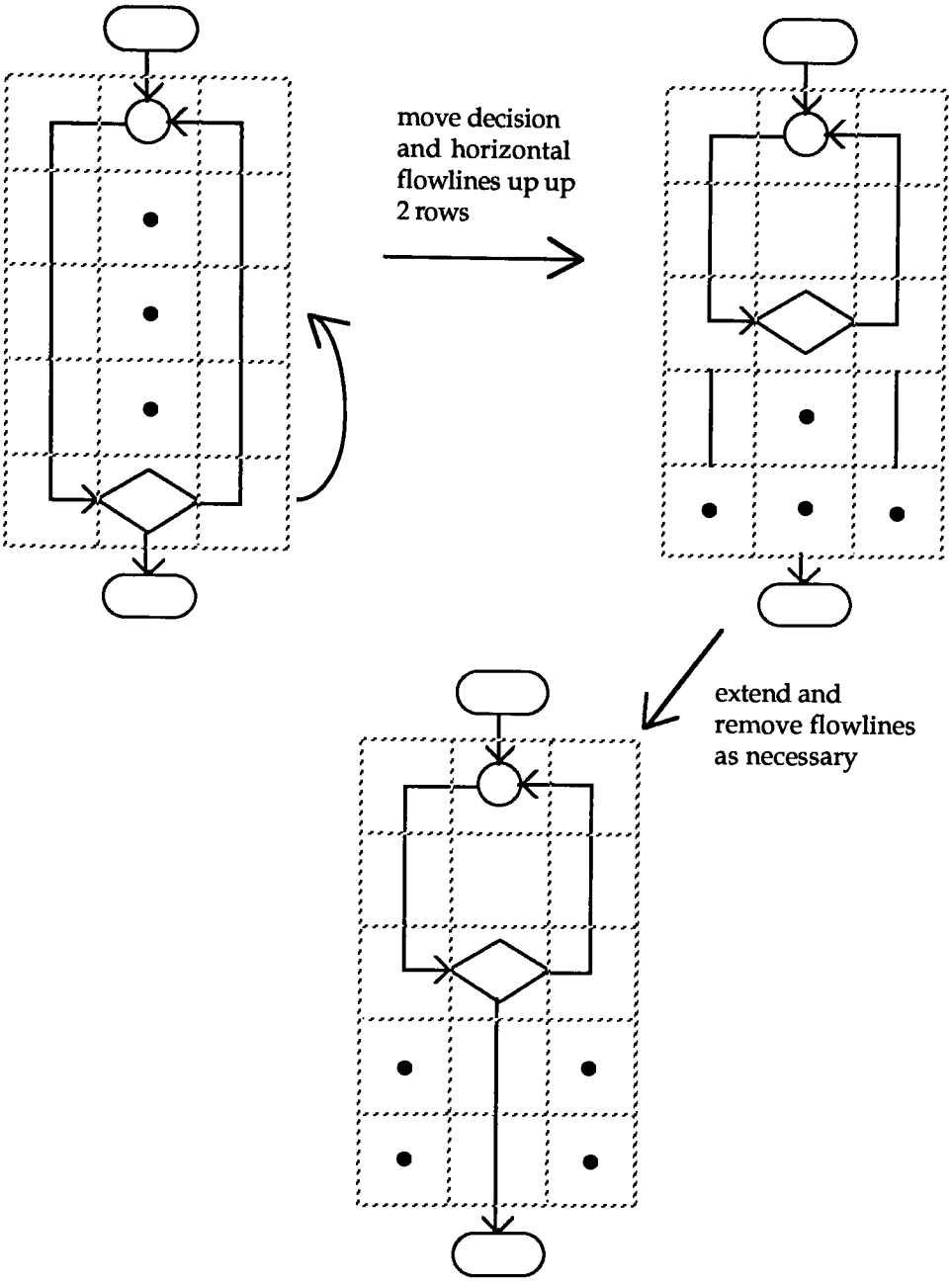


Figure 3.6.4

3.6.2 Delete

Deleting a symbol is done by selecting the delete command then pointing and clicking on a symbol. Delete is very thorough in its processing - probably too much so. When a symbol is selected, say a process, not only is that symbol deleted but any text associated with that process and any subpages as well.

Deleting a terminal can cause one of several actions. If the terminal is a "start" (the first terminal in a diagram) the whole diagram is deleted and the "main or function" pop-up displayed, giving a new diagram. If the terminal is on a subpage, Cview detects whether the subpage belongs to a process or a switch. If the owner is a process the page is deleted then reinitialized to a null diagram. Subpages owned by a switch represent cases. In this instance deleting the terminal is the same as deleting the case. The page is removed from the linked list of pages and the

Processes and switches are handled the same way. As mentioned above the deletion of a process (or switch) causes all text and subpages associated with that symbol to be freed. At this time there is no "bail-out" pop-up in the delete command to catch accidental deletions.

The selection of any part of a structure will cause the whole structure plus all sub-structures, text and subpages to be deleted and replaced with flowlines. The processing for this is similar to the move command. A function is called and passed the selected row,column position. The function traces the outline of the structure except that the top_row and bottom_row values correspond to the ends of the structures, not the ends of the logical path. All of the non-NULL symbols within this area are deleted and replaced with NULL pointers, then the flowline is extended through the area (Figure 3.6.5.)

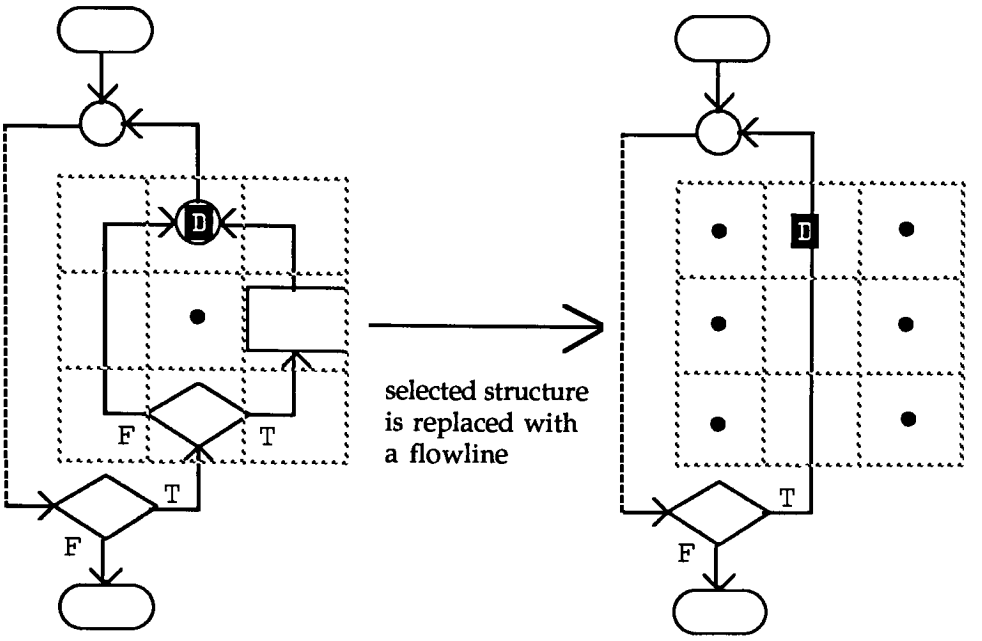


Figure 3.6.5

3.6.3 Swap

The swap command operates on two symbols, decisions and connectors. If the swap command is applied to a decision that forms either an IF-THEN-ELSE or a FORK the T/F or P/C exits are swapped. This is useful when the default position of the True or Parent exit (exit to right) might cause drawing difficulties. For instance the true exit might lead into the interior of another structure making the addition of extra symbols or structures difficult. Swapping the True/Parent exit to the opposite side of the decision generally places it on the edge of the diagram, allowing more room for expansion. Clicking on a decision that forms a loop has no effect. To swap the direction of an exit in a loop it is necessary to rotate the whole structure.

Clicking on the connector that lies at the start of the loop will cause the whole structure to be rotated around the vertical axis. This is accomplished by first finding the physical extent of the structure. The same code that finds the extent of the structure in the delete command is used for this purpose. If the structure is

symmetrical, the distance from the selected column to the right and left edges is equal, the swap is accomplished by exchanging the corresponding symbols on either side of the column (Figure 3.6.6.) The only major change that takes place to the swapped symbols is the exits from the decision must be reversed.

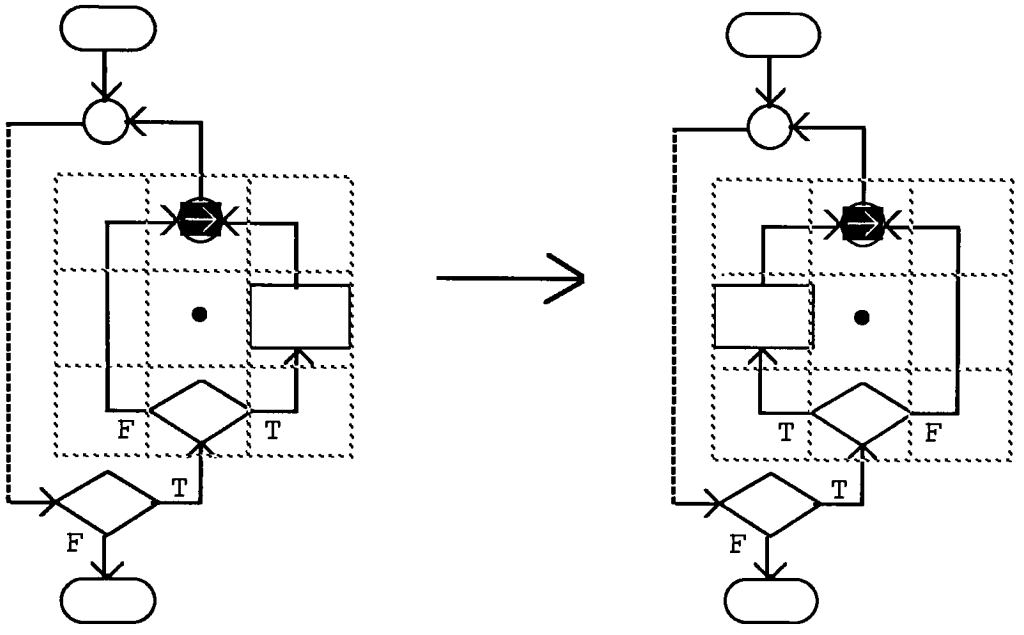


Figure 3.6.6. Symmetrical Swap Operation

If the structure is not symmetrical, a check is made to see if the swap can be made symmetrically. A symmetrical swap on an unsymmetrical structure can take place if the widest side of the structure can be rotated to the opposite side without causing a collision with another structure. Two values must be determined before this can be done: the narrow side of the structure must be identified and the difference in the widths of each side found. If the narrow side is to the left of the selected symbol the area checked for collision is [top_row,left_edge-difference] through [bottom_row,left_edge-1] otherwise it is [top_row,right_edge+1] through [bottom_row,right_edge+difference] (Figure 3.6.7). If this area of the array contains all NULL pointers the rotation will be done symmetrically.

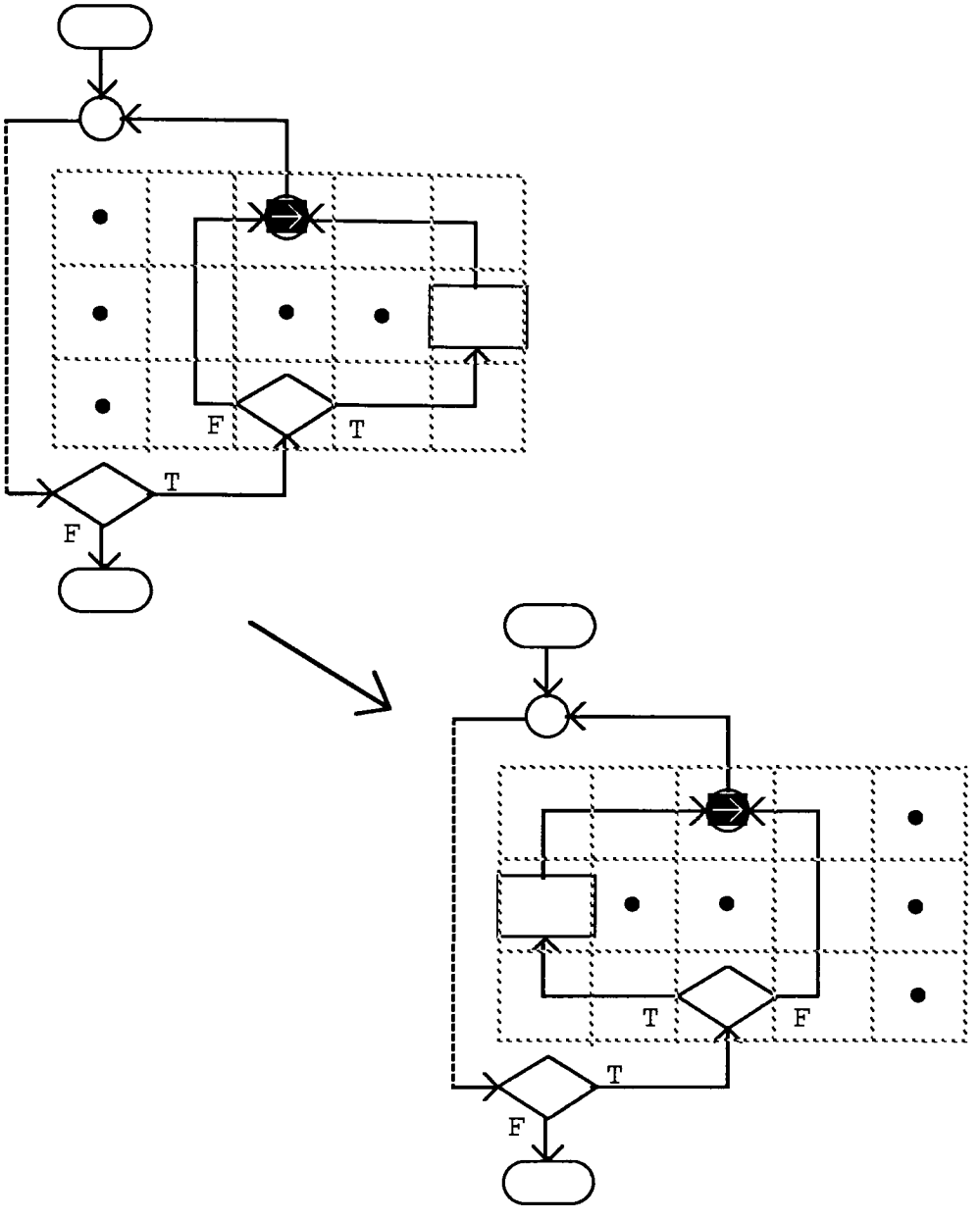


Figure 3.6.7 Symmetrical Swap of
an Unsymmetrical Structure

If a symmetrical swap cannot take place due to a collision with another structure and the structure being swapped is bounded by corners, the swap will be done around the center of the structure. The columns are swapped in exactly the same way as before except that the horizontal flowlines and the connecting corners

must be moved a distance equal to the difference in the widths of each side (Figure 3.6.8).

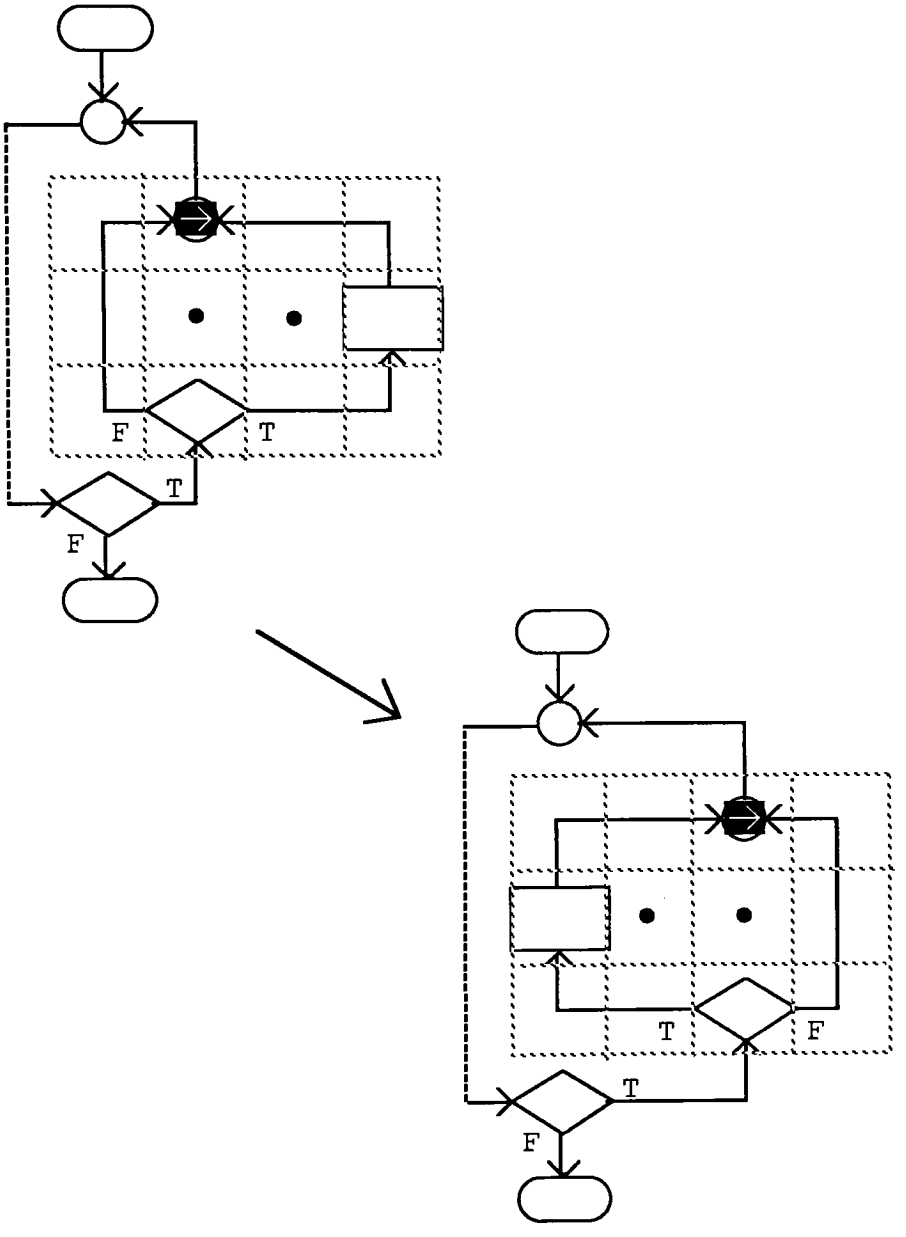


Figure 3.6.8 Unsymmetrical Swap of
an Unsymmetrical Structure

3.6.4 Inspect

During the development of Cview the inspect command was included so that the attributes of an individual Cgraph symbol could be inspected. The attributes displayed were the symbol type (process, decision, connector, etc.), the directions of the connections to the symbol, the direction of the control flow on each connection, in short a verbal description of the bit settings was given. If the symbol had text associated with it the first 20 or so characters were also displayed. This command was used extensively during the debugging phase of the system.

Eventually the command was removed from the system - and was quickly reinstated in a slightly modified form. The current implementation of Cview has no mechanism for indicating which symbols have either text or subpages associated with them; a major shortcoming that is discussed in the concluding section. If a user wanted to find a particular process or switch that had a known piece of text associated with it, he/she would have to fish around with the text command possibly editing symbols that have either the wrong text or no text associated with them. Instead of using the text editor to find the symbol of interest it turned out to be quicker to use the inspect command. The current version of inspect displays all the data the original did except for the direction and connection information.

3.6.5 Expand

During the design of a program it is usually advantageous to group lines of code together that perform one logical function. This is the concept behind functions and procedures. This idea is carried over to Cview and expanded upon slightly due to the mechanics of the system. If the expand command is applied to the "start" (first terminal) symbol on the root page, a new diagram is started. This diagram (subpage) is owned by the terminal symbol. The subsequent expansion of the "start" symbol on

the subpage adds yet another new function. In this way a Cview diagram can represent a main line and a set of one or more functions.

A switch symbol, initially, does not have any cases. Expanding a switch causes a pop-up to appear requesting that the user indicate the number of cases to generate (the default and minimum is 3.) When the user responds to this pop-up, subpages are allocated and chained off of the switch symbol. Each of these pages represents one case. The expand command pre-assigns text to the terminals on these subpages - "case NN:" on the first terminal and "break;" on the last. This supplied text can be edited by the user via the text command.

Diagrams of large programs generally will not fit on one screen. Since the diagrams are constructed in a top-down recursive manner, when the screen becomes full the last symbol inserted is a process. Since processes can represent any arbitrary amount of programming code consisting of both Cview structures and procedural code, this one symbol represents the remaining code. If the expand command is applied to this process symbol a subpage is allocated and linked to it. The insertion process is repeated until this subpage becomes full causing the insertion and expansion of yet another process.

This is not the only circumstance under which expand may be used. The user may make a conscience decision to leave the screen uncluttered and place all of the details on subpages. For instance the root page may contain only an IF-THEN-ELSE with a process on the THEN and ELSE logical paths. Each of the processes could then be expanded allowing the insertion of the detail code.

Applying the expand command to symbols that have been previously expanded causes the existing subpage to be displayed. To move from one function to

another or from one case to another the expand command is applied to the terminal that starts the diagram.

3.6.6 Return

To navigate back from a subpage to the page which contains the expanded symbol requires the use of the return command. Return will optionally return to either the page immediately previous to the current subpage or the root page. Issuing a return from the root page is a null operation. The processing required to perform this is straight forward. Each page contains a pointer to the previous page. The return command makes this pointer the current page (in the Cview environment) then redisplays the new current page. The Cview environment also keeps a pointer to the root page; so the return to the root page just consists of setting the current page pointer to the root page pointer, then redisplaying the screen.

3.6.7 Abstract

Closely related to the expand command is abstract. Abstract is used to collapse either a structure or a logical path down to a process that owns a subpage. Referring to the pseudocode in Figure 3.5.1 and the final diagram in Figure 3.5.6, it can be seen that most of the procedural code has not been inserted and there is no more room for insertions to take place. The abstract command could be used to turn the IF-THEN-ELSE into a process, which owns a subpage containing the IF-THEN-ELSE (Figure 3.6.9). With some editing here would then be enough room for two processes to be inserted on either side of the WHILE loop and one before the IF-THEN-ELSE on the subpage. These processes would represent the assignment statements and the print after the loop (Figure 3.6.10).

The "options" pop-up allows the user to select abstraction by logical path or structure individual symbols (process and switch) cannot be abstracted. This example could have been done in either mode since there was only one structure on the logical path (body of the loop.) If the current abstraction mode is "logical path", selecting an symbol within a logical path (vertical flowpath between terminals or corner flowlines) causes the whole path to be replaced with a process symbol. This is because in "logical path" mode the routine used by the move command to find the extent of a structure is called to determine what to abstract and in "structure" mode the routine used by the delete command is called.

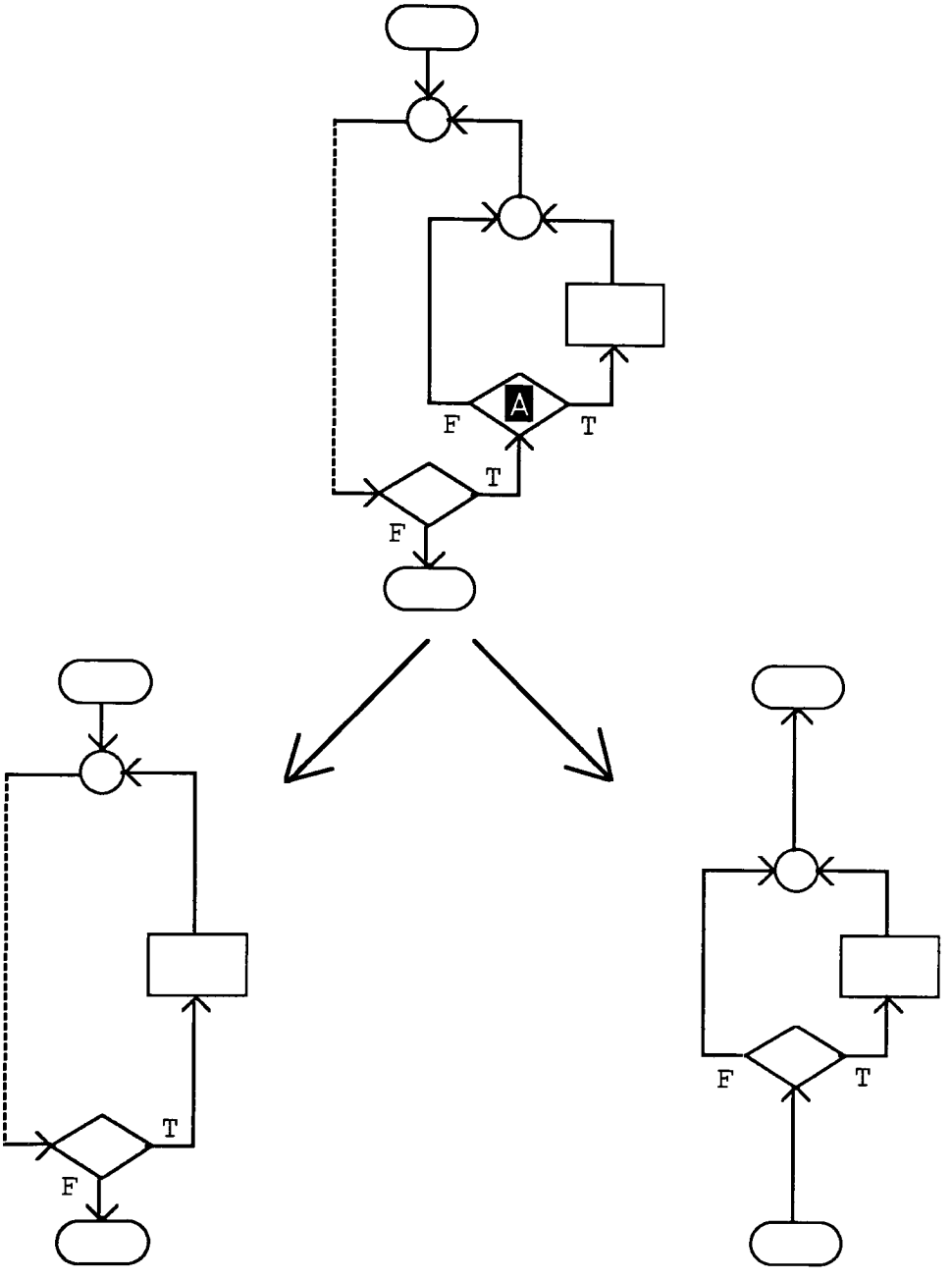
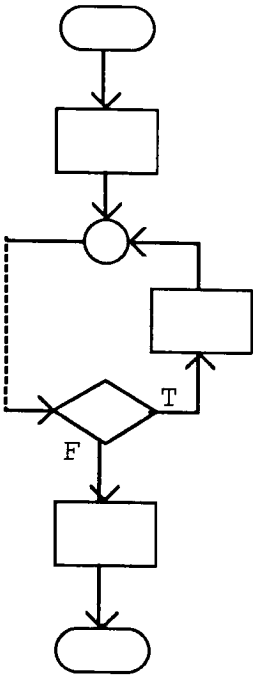


Figure 3.6.9



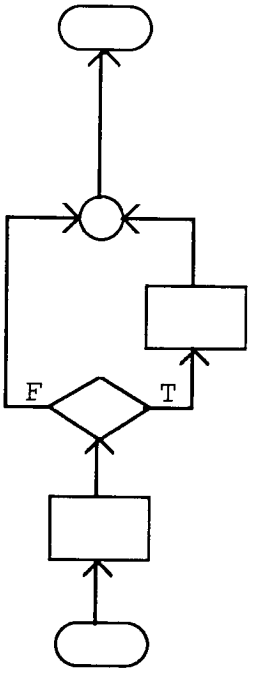


Figure 3.6.10

3.7 The File System

The Cview data structures can be saved to a disk file by issuing the "save" and "save as. ." commands. Likewise, an existing diagram can be loaded from disk via the "load" command. The data structures are saved as a text file and can be modified using any text editor. This decision was made early on in the design of Cview for two reasons. First, saving the file in "human consumable" form allowed for easier debugging and verification of the system. Second, a standard editor can be used to inspect the procedural code and possibly make minor changes. This can be dangerous though. Changing the saved file does not generate new code; the user still must run Cview to generate the new code.

3.7.1 Save

When the "save" or "save as" commands are issued a pop-up is displayed showing the current file name in the Cview environment. The user can click anywhere inside the fill-in and change the name of the file. The file name (or path and file name) does not have a default or assumed file extension; the user is free to use any extension or no extension for Cview files. If the file name is changed, the "save" command will change the file name in the Cview environment. "Save as", however, does not modify the environment and the original file name is left intact. This is done so that the user can save multiple versions of a program without disturbing the original file name.

The first items saved in the file are the Cview environment variables. The click tone, debounce delay, page title row and column indexes and the indent increment and limit are written as the first line. Next, the T/F or P/C exit status, click ON/OFF, grid ON/OFF, show page titles and abstract command mode flags are

written. A simple set of nested FOR loops scans the root page in column order looking for non-NULL elements. When the first used element is found the negative of the page number, the current (row,column) and owned page number are written. "Load" uses the sign of the page number for two purposes: to detect the start of a new page and read an extra line -the page entry point. and page status indicator Next, the two status bytes are written in hexadecimal followed by any text associated with the symbol. The end of the text is marked with a hexadecimal 'FE' (Figure 3.7.1.)

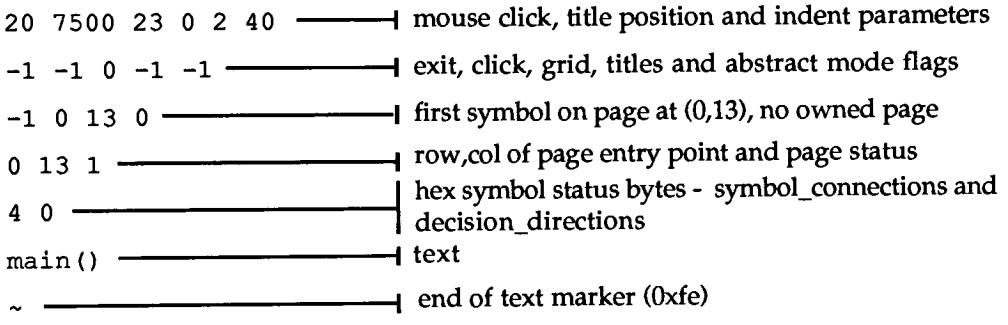


Figure 3.7.1

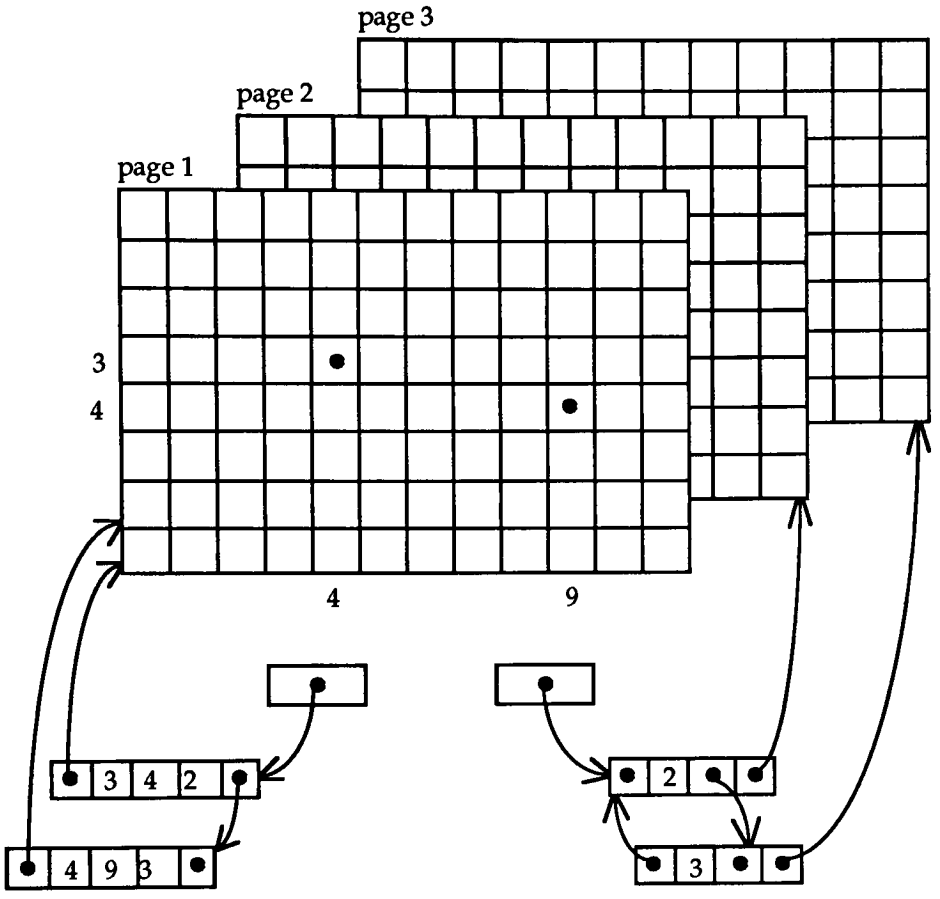
When a symbol owns a page the pointer to the owned page is added to a linked list of page pointers. This linked list is processed as a stack. When the current page has been saved the top of the stack is popped and that page is saved. If symbols on that page own subpages those addresses are pushed on the stack and are next in line for processing. This causes groups of associated pages to be written next to each other in the file.

3.7.2 Load

To load a file it is necessary to read the header information (environment variables), allocate the root page then allocate and initialize symbols as the data is read from the file. Every time a symbol has a non-zero owned page number the

current page address, the current row and column and the owned page identification number are placed in a linked list (the page connection list). When a new page is found (negative page number) the page number and the page address are placed in a linked list called the page address list. After all of the pages have been loaded from the file the two lists are used to link the pages to the symbols that own them.

The first entry is removed from the page connection list and the page address list is searched for the owned page number. Once the page number has been found the entry in the page address list is removed and the page address assigned to the next page field in the symbol (Figure 3.7.2 and Figure 3.7.3). The next entry is removed from the page connection list, the page address list searched and the page and symbol linked until the page connection list is exhausted.



To link a page:

1) Remove the first node of the page connect list and search the page address list for the owned page number.

2) Remove the page address list node and assign the page pointer from that node to the pointer at symbol (3,4) on page 1.

Figure 3.7.2

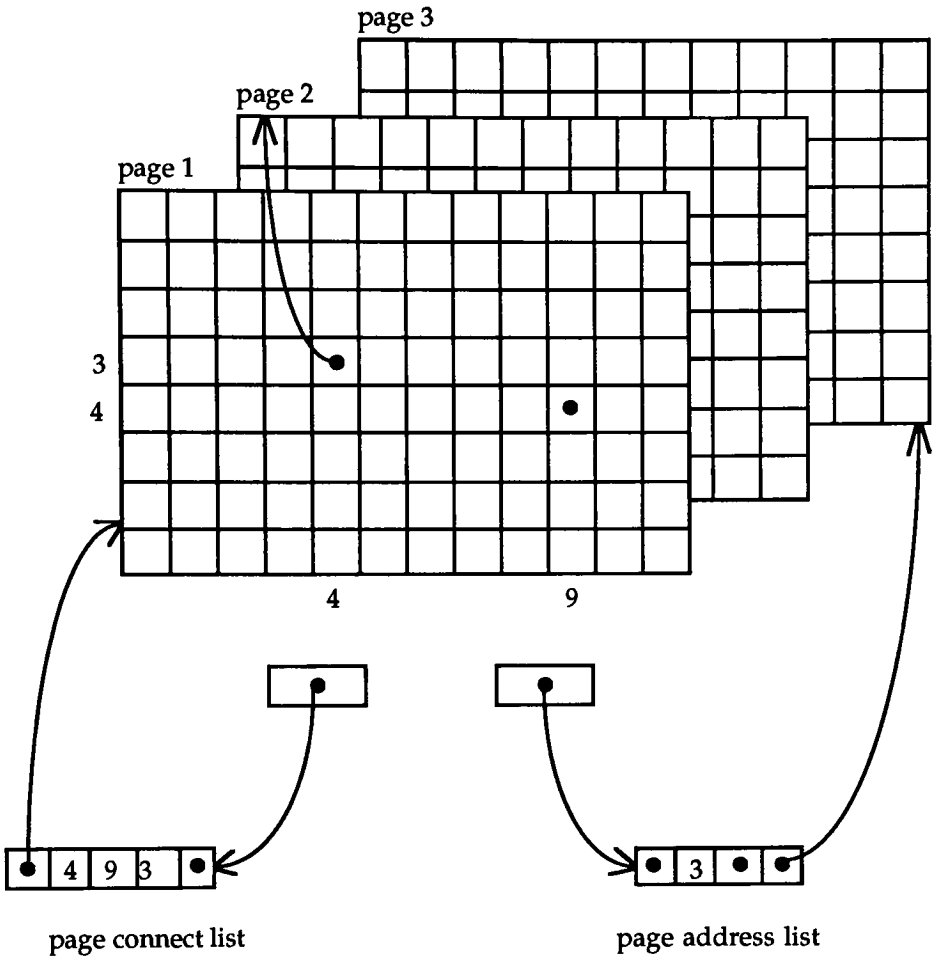

page 3
page 2
page 1
3
4
4
9
4 9 3
3
page connect list
page address list

Figure 3.7.3

3.8 The C Source Code Generator

Once a Cview diagram is developed to a user's satisfaction, the "Generate C" command can be used to produce the C source code. This can be done at any time during the development of the diagram, giving the user the capability to generate different versions of a single program. Like most of the Cview commands a pop-up allows the user to select a file name, indent increment and indentation limit. The file name defaults to the current name in the Cview environment appended with ".c". Two things should be noted about changing the file name: this pop-up does not change the name in the Cview environment and the file name should not include the extension; it will be generated. The indent increment is the number of spaces that loop bodies, then or else blocks and parents or children will be indented. The indentation will proceed until the limit is reached at which point no further indenting takes place.

If a Cview diagram is viewed as one or more logical paths delimited by terminal symbols the following grammar **[HOPC79]** describes the production of a graph:

G = (V,T,P,S)

where: V is the set of variables, {L,P,S}.
T is the set of terminators, {t,d,c,f,w}
S is the starting symbol tLt
and P is the following set of production rules:

tLt => tLPLt | tLSLt | tLdLLcLt | tLcLdLt | tLfLLwLt
L => LPL | LSL | LdLLcL | LcLdL | LfLLwL
P => PtLt
S => StLt

The variables and terminators are:

L - Logical path (flowlines)
P - Process
S - Switch
t - terminator
d - decision
c - connector
f - fork
w - wait

The following symbol sequences have special meaning:

tLt - NULL or initial flowgraph
dLLc - IF-THEN-ELSE
cLd - LOOP
fLLw - FORK/WAIT with parent/child paths
PtLt - Process that owns a subpage.
StLt - Switch that owns one case.

Repetitively applying the forth rule:

S => StLt => StLt => StLttLt => StLttLttLt

produces a switch statement with multiple flowgraphs which represent the case clauses.

The pseudocode in Figure 3.5.1. would be represented by starting with a null flowgraph and applying the following productions:

tLt	=> tLcLcLt	by the first rule.
tLcLdLt	=> tLcLdLLcLdLt	by the second rule.
tLcLdLLcLdLt	=> tLPLcLdLLcLdL	by the second rule.
tLPLcLdLLcLdLt	=>tLPLcLdLLcLdLPLt	by the second rule.
tLPLcLdLLcLdLPLt	=>tLPLcLPLdLLcLdLPLt	by the second rule.
tLPLcLPLdLLcLdLPLt	=>tLPLcLPLdLPLLcLdLPLt	by the second rule.

While this notation may be accurate and concise it has a couple of shortcomings. The first is the difficulty in interpreting the outcomes of the productions; it is not obvious that cLPLdLLcLd is a loop whose body consists of a process followed by an IF-THEN-ELSE. The second problem is that this notation describes only one flowgraph with the subpages (if any) associated with the processes and switches. Cview allows the user to link multiple flowgraphs together from the root page representing a main line routine and multiple functions. By adding one production:

tLt => tLttLt

multiple functions could be allowed, but this would also allow the following illegal (to Cview) production:

PtLt => PtLttLt by the above stated rule.

Processes do not have multiple pages linked to them.

This grammar forms the basis of the syntax diagram in Appendix C. Appendix C describes, in graphical terms, the relationship between the grammar and the Cgraph symbols, how the grammar is represented (as a data structure or fields within a data structure) and the generated C source code. This diagram addresses the second problem noted with the grammar in that multiple flowgraphs are accounted for.

The code generator starts at the entry point on the root page and follows the diagram according to the direction and connection bits set in each symbol. Individual symbols are processed according to the the following rules.

Terminals. Write any text associated with the symbol then if this is a "start" symbol write a "{", increment the indent level and proceed to the next symbol via the

direction and connection bits. If it is a "stop", write any associated text, decrement the indent level, write a "}" and return.

Flowlines. Using the current direction and connection bits move to the next symbol.

Decisions. Text associated with a decision is assumed to be either the condition expression or, if the decision is part of a FOR loop, the FOR expression. If the decision forms an IF, an "if ({associated text})" is written followed by a "{". The indent level is incremented then the horizontal flowlines on the TRUE exit are traversed until a corner flowline is found. This corner delimits the start of a new logical path and the code generator is recursively called with this new starting location. When the TRUE traversal is finished the indent level is decremented and a "}" is written. The code generator then checks the FALSE exit to determine if that logical path is empty. If there are non-flowline symbols on this path an "else {" is written, the indent level incremented and the FALSE exit is processed in the same manner as the TRUE exit.. If the decision forms a pre-checked loop, depending on the loop type either a "while ({associated text})" or a "for ({associated text})" is written followed by a "{". The indent level is incremented then only the TRUE exit is traversed. Finally if the decision is part of a post-checked loop the indent level is decremented, a "}" is written followed by "while({associated text});". No traversal takes place since the logical path has already been processed starting at the connector. In all cases processing continues according to the direction and connection bits.

Connectors. If the connector forms an IF-THEN-ELSE it is strictly treated as a termination symbol for the two logical paths. It will cause the indent level to be decremented and a closing "}" to be written. If a connector is found on a logical path

before a decision, it is the connector in a LOOP. The flowline that exits the connector is traced until the matching decision is found. This is done so the the loop type can be determined. If the LOOP is pre-checked the code generator picks up processing at the decision. When this traversal is done processing continues at the symbol after the decision. If the loop is post-checked a "do" followed by a "{" is written, the indent level incremented and the logical path that leads to the decision is traversed. No text can be associated with this symbol

Fork. The processing for this command is almost identical to that done for an IF-THEN-ELSE. Since the syntax for a fork is embedded in the code generator, as opposed to text attached to the fork symbol, the fork generates its own local variable and if statement; there is no text to be processed. The parent and child exits are processed just like the TRUE and FALSE exits on an IF-THEN-ELSE.

Wait. A wait is treated like a connector in an IF_THEN_ELSE. The syntax of the complete fork wait and exit from the child process is embedded in the code generator and associated with the fork symbol.

Process. If there is any text associated with a process symbol it is written at the current indentation level. If the process owns a subpage the code generator is called recursively using the subpage address and the row,column coordinates of the entry point to the subpage.

Switch. The string "switch({associated text})" is written followed by a "{". The indent level is incremented, then if the symbol owns a subpage, the code generator is called to generate the cases. Upon returning from generating the cases the indent level is decremented and the matching "}". is written.

4. Conclusions

Several issues, divided between two major areas, will be discusses in this chapter. One area is the use of the Cview system both from the stand point of the user interface, the construction of diagrams and the generation of C source code, the second is efficiency of the selected data structures.

In general the use of the command menus, pop-ups and variable cursor "shapes" provides a smooth, fast and easy to use user interface. Response to button clicks for command selection are processed as quickly as the user can move the mouse. The response to mouse commands was so quick that the debounce delay in the options menu had to be added to prevent accidental commands from entering the system. The physical aspects of using the mouse also worked out well. Because the command menus stay on the screen until a button is clicked (either selecting a command or deselecting the menu) no buttons have to remain in the pushed position. Also the mouse movement to reach a command is minimized since the menu appears at or as close as possible to the cursor position

The pop-up portion of the menu system is adequate as it is implemented; it is not perfect however. One major deficiency is that the selection of a button causes the pop-up manager to execute all the call-back routines associated with the fill-in fields, execute the call-back for the button then remove the pop-up from the screen. In hindsight it would be more reasonable to design the system so that an "Ok" or "Cancel" button causes the above processing and all other buttons are toggles; execute the call-back if the button has been pushed an odd number of times. Another problem resides in the fact that the data for the fill-in fields is read by the standard

input function scanf. This allows the user to type fields that are longer than the field on the screen.

From the standpoint of drawing diagrams, is the system easy to use? From this users point of view the answer is, no. The system is severely limited in its present implementation due to the size of the screen - there is only enough room for eight rows of Cgraph symbols on a 24X80 character oriented screen. For problems of reasonable size (see Appendix G) that have many levels of nested program structures this causes problems. If the program has structures nested more than two levels deep, the third substitution must be a process that is expanded to hold the balance of the structures. This assumes that there are no processes or switches inserted in the diagram. The insertion of two processes in the original NULL flowgraph only leaves enough room for one structure with a process inserted. The process in the structure is then expanded to a new page where the same sequence is repeated - after processes are inserted enough room is left for only one or two structures, forcing all subsequent structures on another page.

This leads to what could be called "low density" diagrams, an analog to the window working set discussed in **[MAGU85]**. The number of symbols on a page is relatively low since structures cannot be nested more than a couple of levels deep. This forces the user to place nested structures on different pages (or windows). The addition of each page adds a window to the working set and during diagram editing more windows must be inspected or navigated through, thus slowing down the editing process.

The data structure that represents a page, a dense array of pointers, was selected because it used memory more efficiently if the percentage of used elements

was more than 12%. The number of used elements on the pages that represent the programs in Appendix G averages out to 15 per page or only 7.1% (Table 4.1).

Percentage Utilization Data from Appendix G. Diagrams																			
																			average
SYM./PAGE	15	21	14	13	13	15	13	15	13	15	15	21	14	13	15	13	15	13	15
% UTIL.	7.2	10.	6.7	6.3	6.3	7.2	6.3	7.2	6.3	7.2	7.2	10.	6.7	6.2	7.2	6.2	7.2	6.2	7.1

Table 4.1

To determine how many rows would have to be added to achieve 12% utilization, the formula in Figure 4.1 was developed. This formula assumes that the only operation is the recursive substitution of IF-THEN-ELSE, LOOP or FORK/WAIT. The variable **d** is the maximum number of used elements, **m** is the number of rows of Cgraph symbols on the screen and n the even integers. The formula follows from the argument: If the number of rows **m** is four, there is enough room for one structure requiring eight Cgraph symbols (Figure 4.2). If one more row is allowed the number of symbols increases by two due to the extra flow-lines, the third position is a "hole" in the structure held by a NULL pointer. Incrementing **m** to six allows for the full nesting of two structures with no "hole".

$$d = (m - (2 + m \bmod 2)) + \sum_{\substack{n = 0, 2, 4, \ldots \\ m-n \geq 2 + m \bmod 2}} m - n$$

Figure 4.1

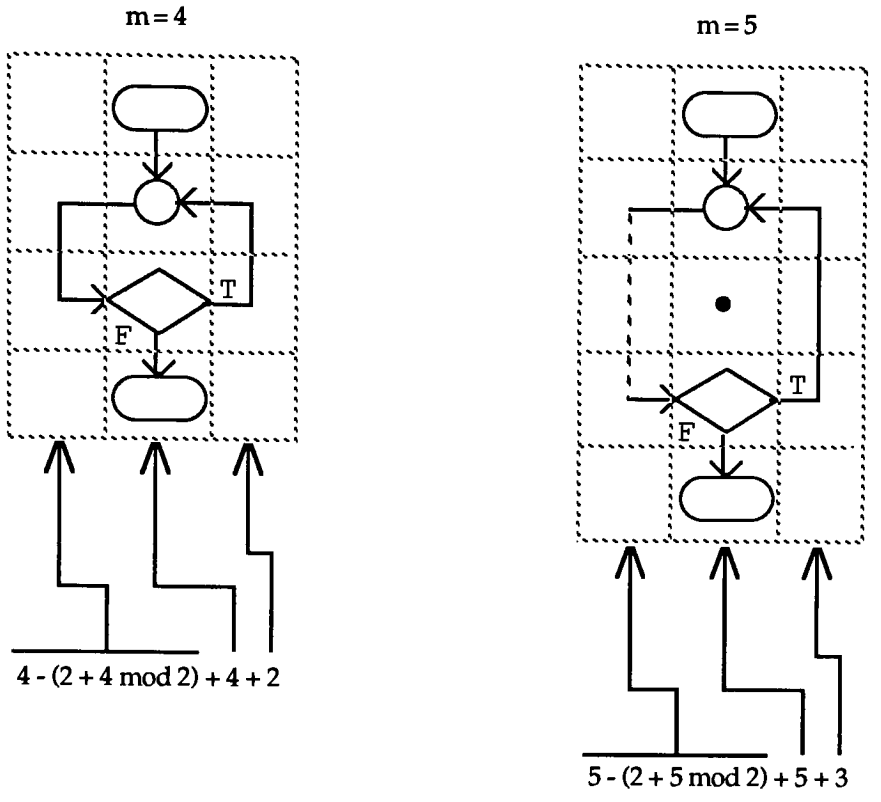


Figure 4.2

The results of calculating **d** for **m** = 4, 5, 6, . . .,13 then determining the percentage utilization for an array with 26 columns are graphed in Figure 4.3. The original estimate of eight rows producing about a 12% utilization was correct. Because the insertion of processes and switches lowers the number of levels that can be nested on one page, the utilization dropped by 5%. This implies that if the display can support 13 rows of Cgraph symbols, a maximum utilization of about 17%, then actual usage should be about 12%. Display adaptors that support 44 display rows are available. This would allow for 14 rows on a Cview diagram and minimally efficient use of the data structures.

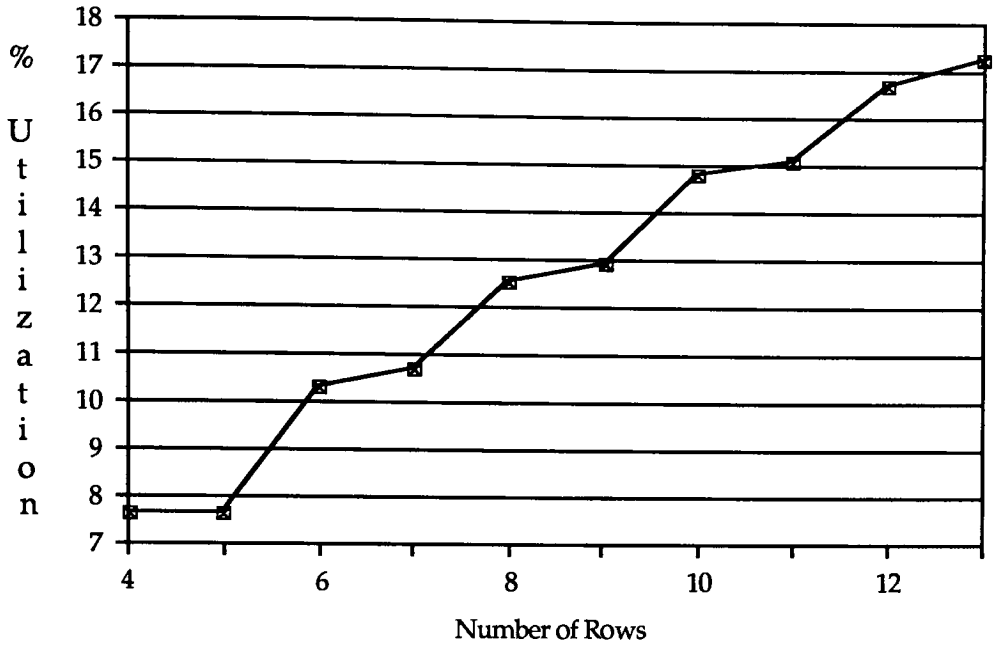


Figure 4.3

This is not to say that a 14 row display would be sufficient to make working with the system easy. This number of rows only allows for efficient use of memory. To make the system easy to use it would be preferable to keep the complete diagram on one screen or as much of the diagram as the designer wants to work with at one time. In this case the more rows the display can support the better.

Another major problem with the system is determining which process and switch symbols have subpages and which symbols have text associated with them. It was mentioned in the edit command section that the inspect command had to be added back to the system because the current implementation of Cview has no way of communicating which symbols own subpages or text. A reasonable modification to the character mode system would be to establish a convention such as: a highlighted symbol owns text and an underlined symbol owns a subpage. On a

device the text could be "greeked" in the symbol and a bold or double border could represent a symbol with a subpage.

The design goal of having the generated source code meet one standard format was easily met since the indentation is all done in a consistent manner. Every THEN/ELSE block, PARENT/CHILD and LOOP body start and end with brackets, even if there is only one statement in the block. The code within a block is always indented by a standard amount (assuming the programmer entered the detail code without indenting individual lines) and the start of a program structure always includes one line of white space to set it off. The indentation can be set via the options pop-up to suit various tastes but the two column default produces easy to read output.

In summation the system in its present form is minimally useful due to the display limitations. If these limitations were removed and the above mentioned difficulties resolved, Cview could potentially be quite useful for quickly generating program skeletons, generating reasonably sized programs and automatically providing necessary system documentation.

5. REFERENCES:

[BOEH81] Barry W. Boehm, Software Engineering Economics New Jersey: Prentice-Hall, Inc. 1981

[BROO82] F. P. Brooks, The Mythical Man-month, Essays on Software Engineering, Massachusetts: Addison-Wesley Publishing Company 1982, The Flow Charting Curse - pp167-9

[CAMP83] M. J. Campo, "A case in point: Industry needs documents," Mini/Micro NE. Electronics Show and Convention, 3/1/1-5 1983

[CARD82] A. F. Cardenas, W. P. Grafton, "Challenges and Requirements for new Application Generators," AFIPS Conference Proceedings. vol. 51, 1982 National Computer Conference, 341-9 1982

[CHES84] M. Chesi, E. Dameri, M. P. Franceschi, "ISDE: An interactive software development environment," SIGPLAN Notices (USA), Vol.19, No.5 81-8 May 1984 - Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments 23-25 April

[DELI84] N. M. Delisle, D. E. Menicosy, "Viewing a programming environment as a single tool," SIGPLAN Notices (USA), vol.19, no.5 49-56 May 1984, Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments 23-25 April 1984 Pittsburg, Pa

[ELLI85] R. J. Ellison, B. J. Staudt, "The evolution of the GANDALF system," J. Syst. & Software (USA), vol. 5, no.2 107-19 May 1985

[GROC82] J. M. Grochow, "Application Generators: An Introduction," AFIPS Conference Proceeding. VOL.51 1982 National Computer Conference 389-92 1982 7-10 June 1982 Houston, TX, USA

[HOPC79] John E. Hopcroft, Jeffrey D. Ullman, Introduction to automata theory, languages and computation. Addison-Wesley Publishing Company Inc. 1979.

[HWAN82] C. J. Hwang, "Structured D-Charts: A diagramatic Methodology in Structured Programming." AFIPS Conference Proceedings, vol.51, 735-748, Jun. 1982

[KAMM75] R. Kammann, "The Comprehensibility of Printed Instructions and the Flowchart Alternative." Human Factors, 1975 17(2), 183-191

[KANT85] E. Kant, "Understanding and automating algorithm design." IEEE Trans. Software Eng. (USA) vol. SE-11, no.11 1361-74 Nov. 1985

[MEDI81] R. Medina-Mora, P. H. Feiler, "An incremental programming environment." IEEE Trans. Software Engineering, vol. SE-7, 5 (Sep. 1981) 472-482.

[MAGU85] M. C. Maguire, "A review of human factors guidelines and techniques for the design of graphical human-computer interfaces", Comput. & Graphics (GB) vol. 9, no. 3, 221-35, 1985

[MORR81] J. M. Morris, M. D. Schwartz, "The design of a language-directed editor for block-structured languages." ACM SIGPLAN Notices 16, 6 (June 1981), 28-33

[NOTK85] D. Notkin, "The GANDALF project," J. Syst. & Software (USA), vol. 5, no. 2 91-105 May 1985

[ROTH82] R. L. Roth, "Program Generators and Their Effect on Programmer Productivity", AFIPS Conference Proceedings. vol.51. 1982 National Computer Conference, 381-8 1982 7-10 June 1982 Houston, TX, USA

[SHNE77] B. Shneiderman, R. Mayer, D. McKay, "Experimental Investigations of the Utility of Detailed Flowcharts in Programming," CACM, June 1977, vol.20, no.6

[STAN84] T. A. Standish, R. N. Taylor, "ARCTURUS: A prototype advanced Ada programming environment," SIGPLAN Notices (USA), Vol.19, No.5 57-64 May 1984 - Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering

[SUNP85] SunView Systems Programmers Guide, Sun Microsystems Inc., 1985

[TAYL82] T. Taylor, T. A. Standish, "Initial thoughts on rapid prototyping techniques." Software Engineering Notes, 7,5, 160-66 (Mar. 1982)

[TEIT81] R. Teitelbaum, T. Reps, "The Cornell Program Synthesizer: A Syntax Directed Programming Environment," CACM, vol.24 no.9, 563-573 1981

[THIM85] H. Thimbleby, "Failure in the technical user-interface design process," Comput. & Graphics (GB), vol.9, no.3 187-93 1985

[WALD82] J. H. Waldrop, "Application Generators: A Case Study," AFIPS Conference Proceeding. vol.51, 1982 National Computer Conference, 363-8 1982 7-10 June 1982 Houston, TX, USA

[WORK83] D. A. Workman, "GRASP: A Software Development System using D-Charts." Software Pract. and Exper., vol.13, no.1, 17-32, 1983

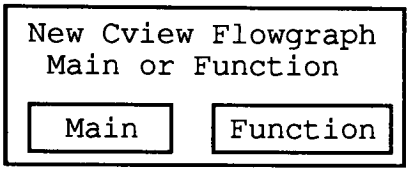
[WORK85] D. A. Workman, F. Arefi, M. Dodani, "GRIP: A formal framework for developing a support environment for graphical interactive programming." Conference on Software Tools, p. 138-53 1985, 15-17 April 1985 NY City

Appendix A.

This appendix is a summary of the Cview commands and functions. Each pop-up menu item is discussed and examples of the command actions are presented.

Starting up:

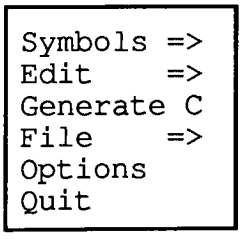
After Cview is started by typing "cview" at the system prompt, a pop-up is displayed:



which allows the user to either generate a main program with optional functions or only functions. Once this selection has been made a null flowgraph is displayed on the screen.

Root Menu:

When the right-hand mouse button is depressed the following menu will appear:

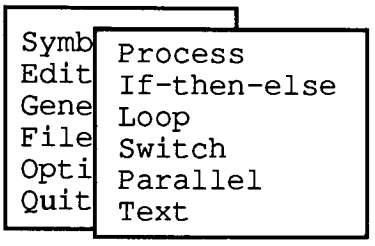


If the mouse cursor was too close to the edge of the screen the position of the menu is adjusted so that it appears on the screen in its entirety. Moving the cursor to a line (exclusive of the menu border) and clicking the left button selects that command or option. If the command or option has a "=>" next to it, that selection has a submenu

with more selections. Clicking the left button with the mouse cursor on the menu border or outside the box deselects the menu.

Symbol Menu:

Pointing to the "Symbol" selection in the main menu and clicking causes the symbol submenu to be displayed:



Selection of an item from the list causes the cursor to change to the current symbol. If the selection was not Text then clicking the left mouse button when the mouse cursor is over a vertical flowline causes the current symbol to be inserted; provided there is enough room. If the Text command was selected, clicking on a non-flowline symbol causes the text editor to be invoked. All text entered is associated with the selected symbol and will be processed according to the following rules: Text attached to a(n) -

1) decision is either a boolean expression or a FOR expression list.

2) switch is the switch expression.

3) process is considered to be procedural code and will be used "as is."

4) terminal will be used "as-is". This text is initialized to either a "main" or "function" line if the terminal is the start of the flowgraph, "case nn:" if it is the start of a case, "return" if it is the end of the flowgraph or "break;" if the end of a case. These default settings can be edited by using the Text command.

All other text is treated as comments.

The editor uses the following special keys:

1) Cursor keys - move the text cursor within the screen bounds.

2) ALT-left arrow - move from the current position to the beginning of the previous string of non-blank characters.

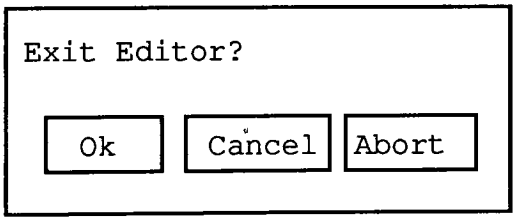
3) ALT-right arrow - move from the current position to the end of the next string of non-blank characters.

4) Home - move the cursor to the first character of the first line.

5) End - move the cursor to the end of the current line.

6) Return - If the cursor is at the first character position, hitting return (enter) inserts a line before the current line, otherwise the cursor moves to the first column of the next line.

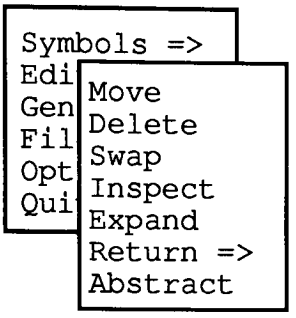
7) Esc - causes the following pop-up to appear:



Selecting Ok saves the changes and returns to the graph, Cancel returns to the editor and Abort ends the editor without saving the changed text.

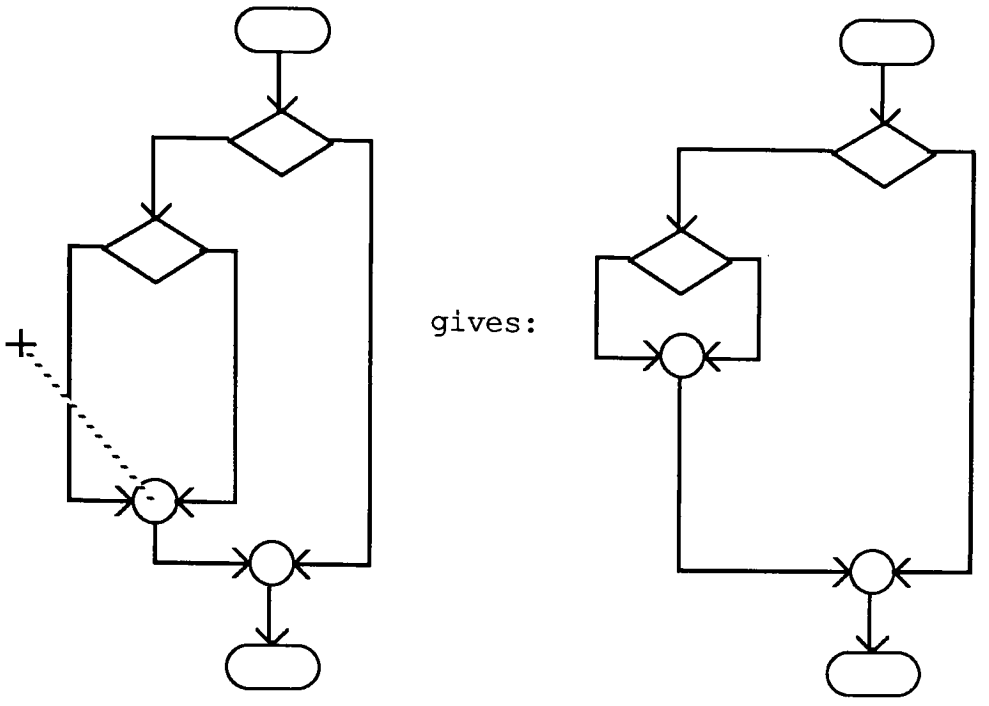
Edit Menu:

The commands that can be selected from the edit menu:



are used to manipulate program structures, sections of program structures (connectors, decisions and flowlines), create new Cview flowgraph pages and navigate through a flowgraph.

Move: Point to any non-horizontal, non-corner flowline symbol and click the left button. The cursor will change to a cross-hair. Point to the new symbol position and click the left button again. The cursor will change back and all the structures on the selected flowline will be moved. If the move would cause symbols to overlap, the cursor simply reverts to its original form and the move is not done. Example: Select the connector then click at the new location -



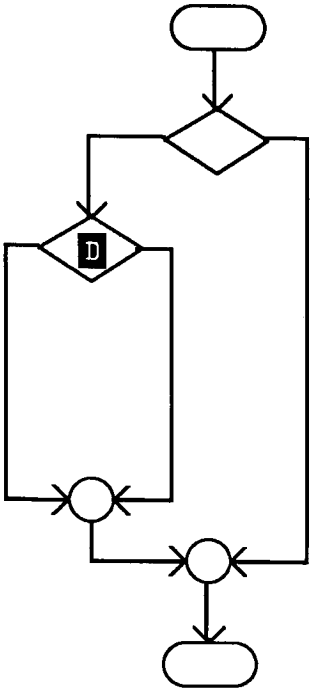
Delete: Point to any non-flowline symbol and click the left button. Structures/symbols are deleted according to the following rules:

1) Process and Switch - The symbol plus any associated text and subpages are deleted.

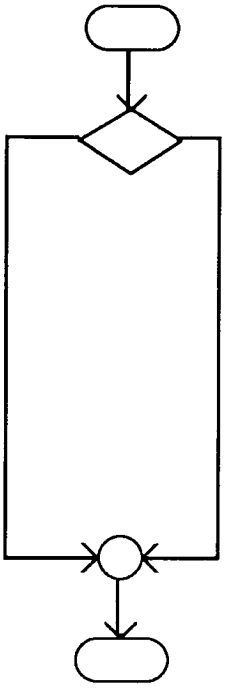
2) Terminal - If this is the root terminal this deletes the whole flowgraph. The startup menu is displayed asking if you want to generate a main program or a function. If this is not the root page, delete all the structures and symbols on this page and any subpages then reinitialize the page to a null flowgraph. If this is a terminal for a case clause then delete this case.

3) All others - The program structure (if, loop or parallel section) is deleted including all substructures, text and subpages.

Example: Clicking either the decision or the connector on the nested if-then-else causes the structure to be deleted -

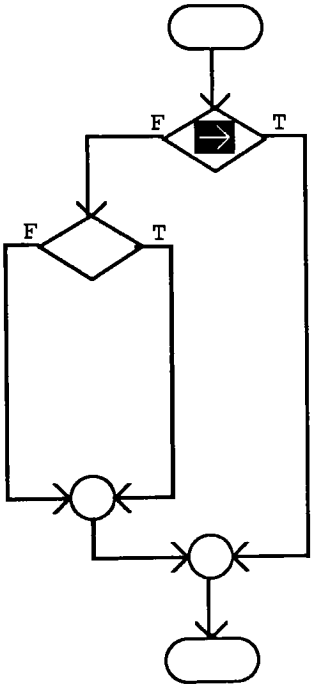


Gives:

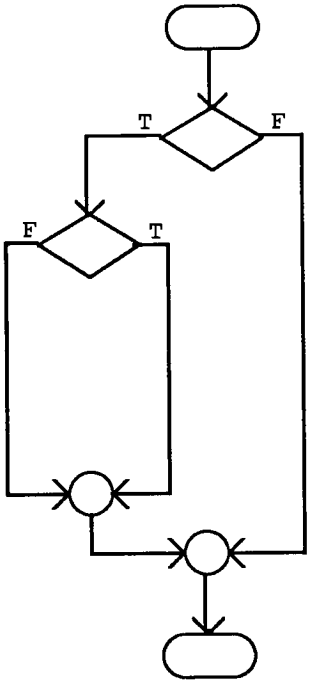


Swap: Clicking on an "if" decision or fork symbol in this mode will cause the true/false or parent/child exits to switch places. Clicking on a "loop" decision has no effect. Clicking on the connector of a program structure rotates the structure about its vertical axis. Swap will not rotate a structure if it will cause parts of the graph to overlap or extend off the edge of the screen.

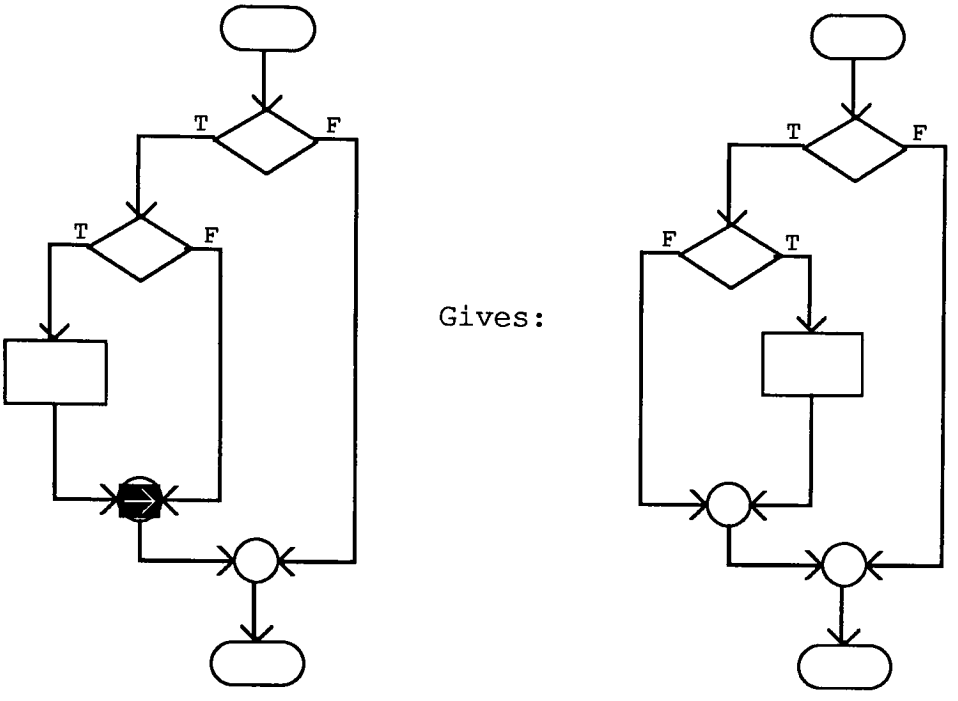
To swap exits on a decision or fork -



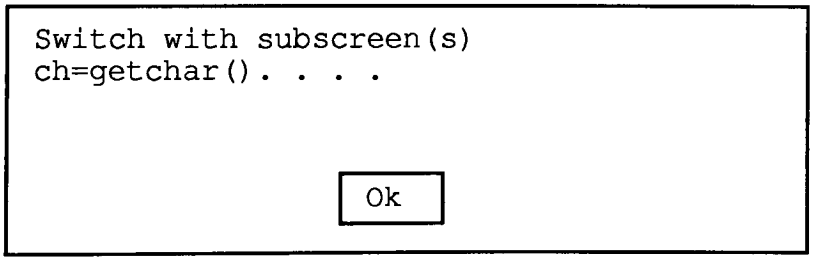
Gives:



To rotate a program structure on its vertical axis -



Inspect: Displays the symbol type, indicates if the symbol has any subscreens (pages) and shows the first twenty characters of text, if any exits, using the following pop-up:



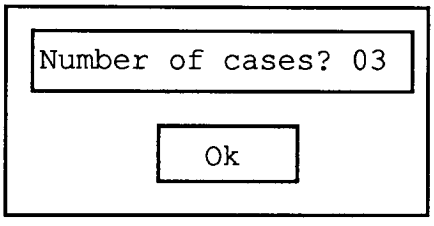
Originally used for debugging, this command was left so that it would be easier to find which symbols have either text or subpages associated with them.

Expand: Either goes to the next subpage or allocates a new subpage. Specifically:

1) If the symbol is a process, expand to a new page or allocate a new page.

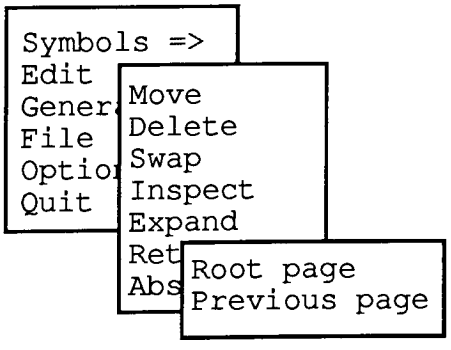
2) If the symbol is the root terminal, expand to the next page which is a function or allocate a new page and set the null flowgraph to an empty function. If the terminal is for a case clause, expand to the next case. If the current page is the last case, allocate a new case clause and initialize the constant to -1.

3) If the symbol is a switch that has never been expanded display the following pop-up:



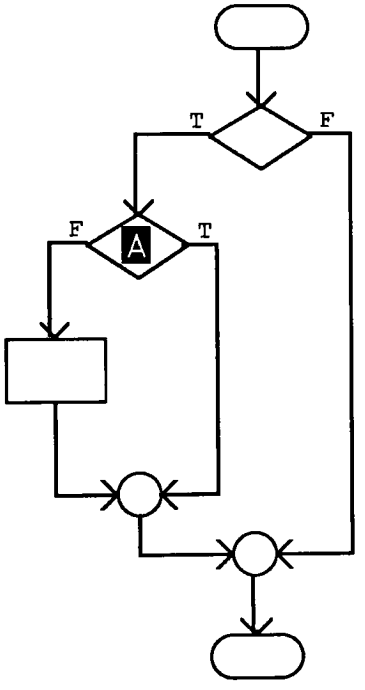
then allocate the number of pages requested as case clauses whose values are the integers 0..(number of clauses). Otherwise, expand to the first case clause.

Return: Displays the following pop-up:

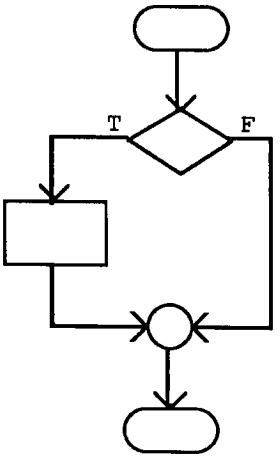


This will display either the root page or the page that owns this subpage. Returning from the root page has no effect.

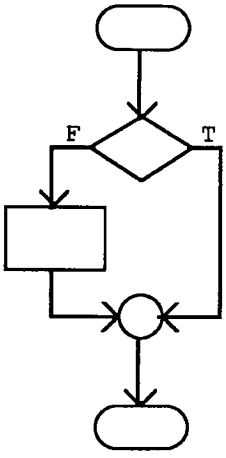
Abstract: If the screen becomes too cluttered, a structure or symbol can be cut from the page, placed on a subpage and be replaced by a process symbol. At the end of the operation the screen will show the structures/symbols on the subpage. Issuing a "return to the previous page" command will return to the page to which the abstract command was applied. Example:



Will produce two pages - the second of which will be the displayed page.

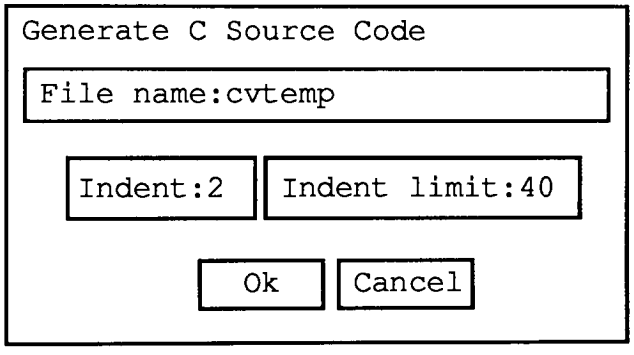


And:



Generate C:

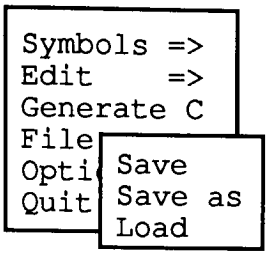
The following pop-up is displayed:



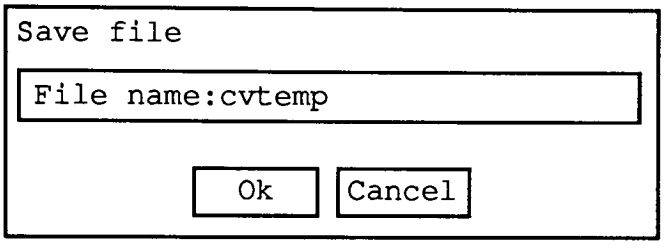
The default file name is cvtemp. If a different file name is supplied do not include the ".c" extension, it will be supplied. The Cview internal representation is converted to C source code using the indent value and indent limit. Each block of code within "{}"'s is indented by the value of indent until the limit is reached.

File:

Selecting the file option causes a submenu to be displayed:



The pop-up associated with each of these commands:



has the same format; the only difference being the pop-up title, indicating the current operation. The file names are accepted as typed with no default extensions. The commands behave according to the following rules:

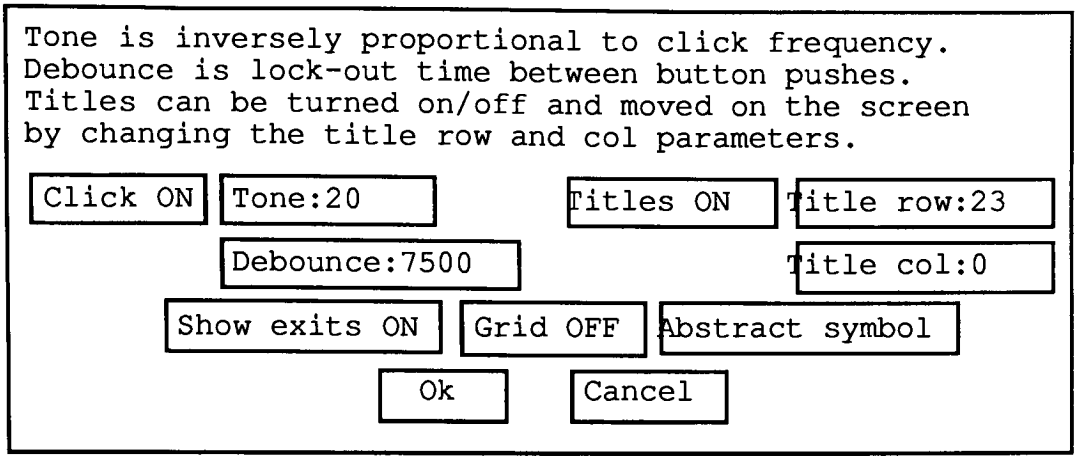
1) Save - Saves the file using the current file name (the name displayed in the pop-up.)

2) Save as - Saves the file using the displayed file name but does not save the file name in the Cview environment.

3) Load - Loads the displayed file name and saves the name in the Cview environment as the current file name. If the current graph has been modified and not saved, the load command will display a "save file" pop-up before executing the load.

Options:

The options command allows various parameters that are kept as part of the Cview environment to be modified. This is done via the following pop-up:



The items that are controlled are:

Mouse: The click tone for the mouse buttons can be enabled/disabled, the tone can be changed and the amount of time that Cview locks the mouse out after a button is pushed can be modified. Smaller debounce values allow for quicker button response.

Titles: The first line of text associated with the pages start symbol can be removed from the screen or have its row, column position changed.

Exits: The true/false or parent/child exits can be displayed or hidden.

Grid: An alignment grid can be displayed to aid in drawing or moving symbols or structures. *This option can caused degraded response due to the amount of screen output generated.*

Abstract: Causes the abstract command to either process the symbol/structure that was selected or every symbol/structure on the selected logical path.

Quit:

If the current graph has not been modified this command will return to the system. Otherwise, a "save file" pop-up is displayed (see the File command) allowing the user a last chance to retain the current file.

Appendix B.

NAME

cview - graphical design aid for the C programming language

SYNOPSIS

cview

DESCRIPTION

Cview allows a designer to generate part or all of a C program using a mouse based graphics editor. A modified version of flowchart is used to represent:

1) IF-THEN-ELSE
2) FOR
3) WHILE
4) DO . . . WHILE
5) SWITCH
6) FORK/WAIT (parallel processing)

The designer can use cview to quickly layout and generate the skeleton of a main program and one or more functions. Detail code (comments, conditional statements, and sequential code) can be added to the generated code using a standard text editor or can be inserted in the flowgraph from cview. For more details on how to use cview please see the user manual.

FILES

Cview file names are accepted as typed by the user; no extensions are supplied or required. Generated C source code is written to a file whose name is formed from the cview file with a ".c" extension. EXAMPLE: If the cview file is k&rp18 the C source will be written to k&rp18.c.

DIAGNOSTICS

The diagnostics produced by cview are intended to be self-explanatory.

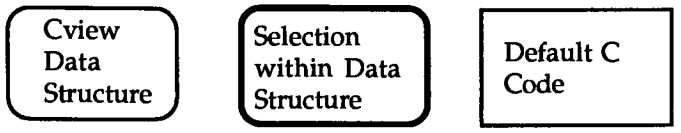
BUGS

Abstracting symbols that have logical paths that overlap along the vertical axis can cause symbols to be mis-selected.

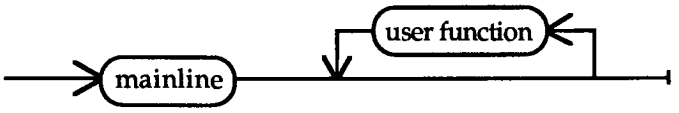
Appendix C.

Cview Syntax and Generated Code

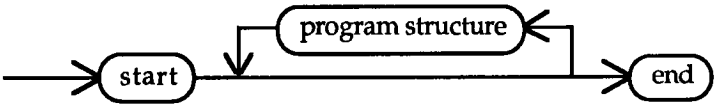
Key to Symbols:



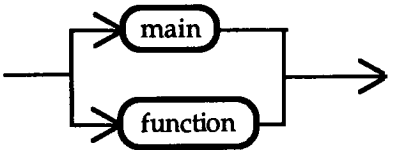
Cview Diagram



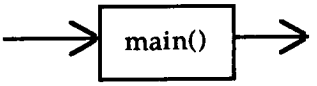
Mainline or User Function



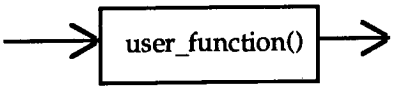
Start



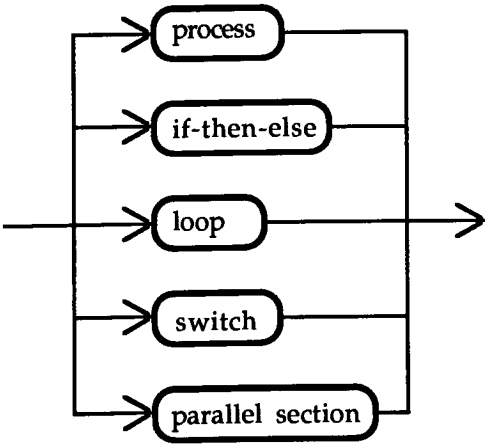
Main

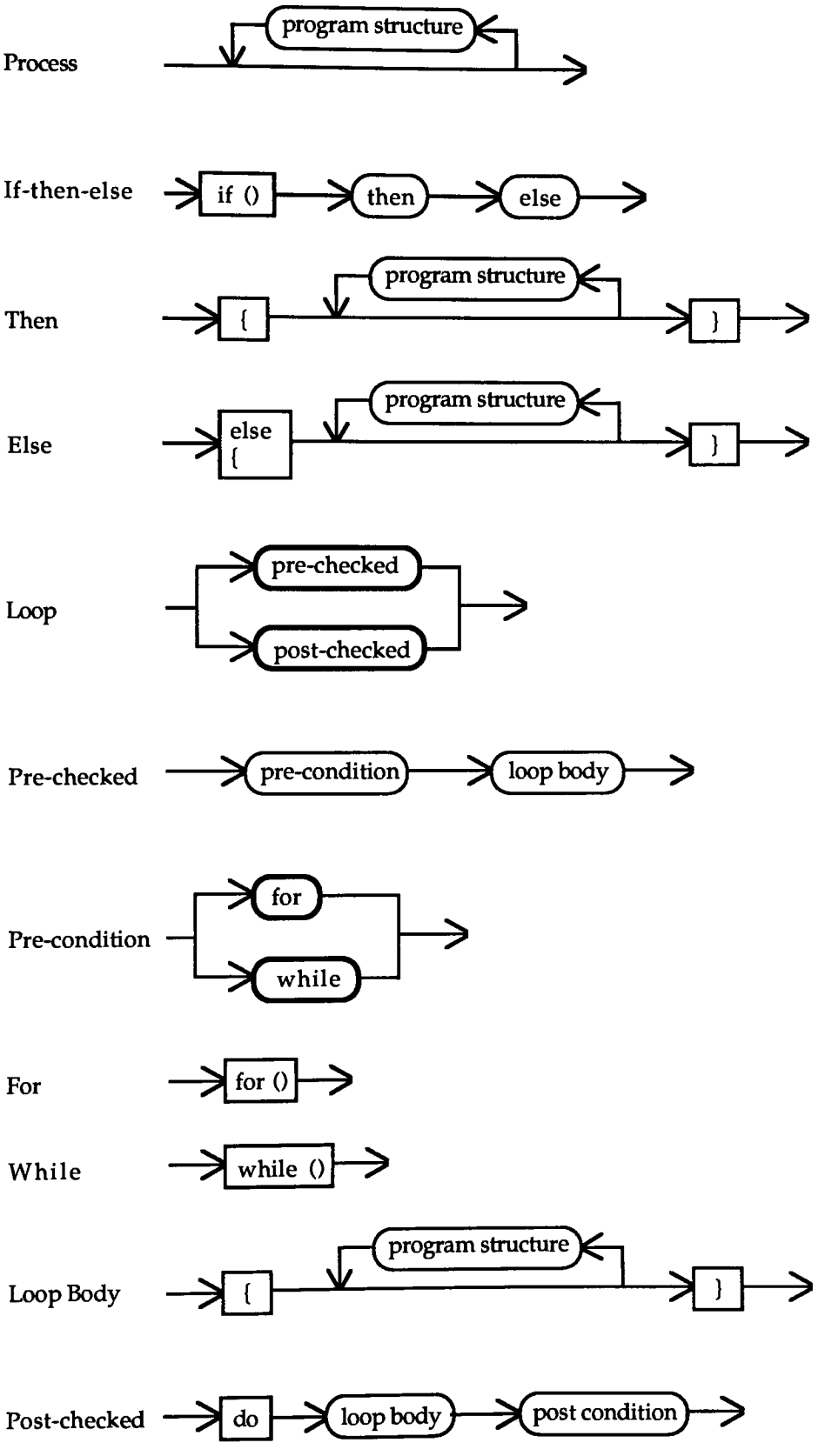


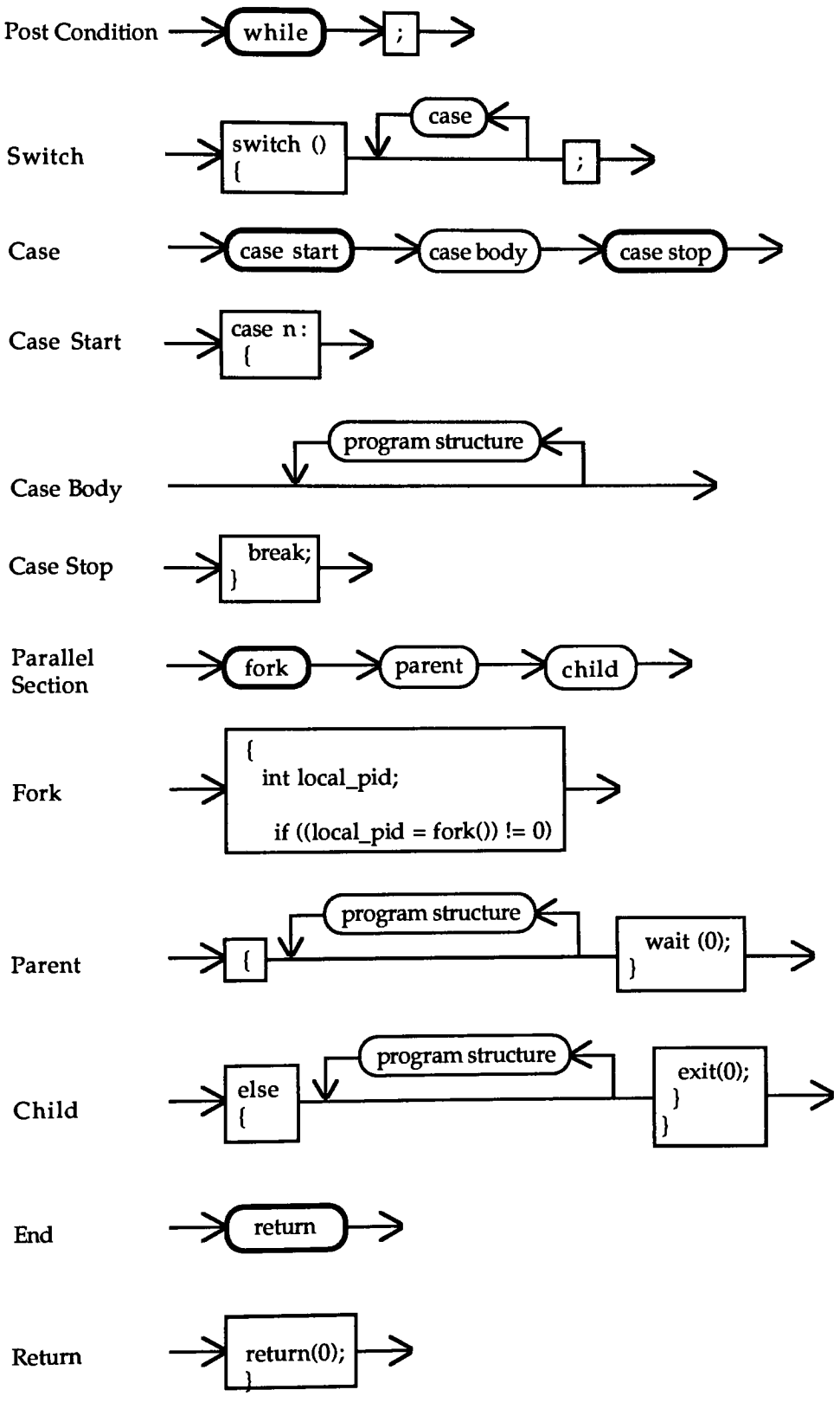
Function



Program Structure




Process
program structure
If-then-else
if ()
then
else
Then
{
program structure
}
Else
else
{
program structure
}
Loop
pre-checked
post-checked
Pre-checked
pre-condition
loop body
Pre-condition
for
while
For
for ()
While
while ()
Loop Body
{
program structure
}
Post-checked
do
loop body
post condition


Post Condition
while
;
Switch
switch ()
{
case
;
Case
case start
case body
case stop
Case Start
case n :
{
Case Body
program structure
Case Stop
break;
}
Parallel Section
fork
parent
child
Fork
{
int local_pid;
if ((local_pid = fork()) != 0)
Parent
{
program structure
wait (0);
}
Child
else
{
program structure
exit(0);
}
}
End
return
Return
return(0);
}

Appendix D.

Function to file cross reference.

cvmouse.c:

_get_mouse_position	_mouse_buttons	_mouse_click
_mouse_hide	_mouse_installed	_mouse_show
_set_mouse_cursor	_set_mouse_position	_wait_for_button_one

cvinspec.c:

_inspect_symbol

cvdelete.c:

_delete_extent	_delete_symbol	_delete_text_list
_fill_in_line	_find_structure_extent	

cvswap.c:

_flip_area	_swap_corners	_swap_inout_exits
_swap_structure		

cvmod.c:

_exec_command

cvexpand.c:

_case_number	_case_ok	_expand_symbol
_insert_page		

cvdialog.c:

_button_manager	_button_pushed	_dialog_manager
_fill_in_manager	_fill_in_selected	_place_buttons
_place_fillins	_place_messages	

cvmenu.c:

_display_menu	_display_return_menu	_disp_edit_menu
_disp_file_menu	_disp_symbols_menu	_return_to_prev
_return_to_root	_set_abstract	_set_delete
_set_expand	_set_if	_set_inspect
_set_loop	_set_move	_set_parallel
_set_process	_set_quit	_set_swap
_set_switch	_set_text	_show_menus

cvoption.c:

_option_abstract	_option_click	_option_exits
_option_grid	_option_ok	_option_page_title
_option_set_col	_option_set_debounce	_option_set_row
_option_set_tone	_set_options	

cvdraw.c:

_box	_clear_screen	_clear_window
_cv_draw_symbol	_draw_connections	_draw_tfpc_exits
_hide_cursor	_locate	_redraw_page
_restore_scr	_save_scr	_scrgetc
_scroll_down	_scroll_up	_scrputc
_set_mode	_show_cursor	

cvabstr.c:

_abstract_area

cvabort.c:

_abort_ok _abort_process

cvfile.c:

_add_to_page_address_list	_connect_all_pages	_file_ok
_file_set_file_name	_get_line	_get_page_address
_load_file	_load_file_from_disk	_pop_page_address
_pop_page_connection	_push_page_address	
_push_page_connection		
_save_as	_save_file	_save_file_to_disk
_write_page		

cvtext.c:

_beep	_copy_text	_copy_text_chain
_delete_space	_display_text	_edit_abort
_edit_exit	_edit_manager	_edit_stay
_edit_text	_free_text_chain	_init_text_blocks
_init_text_line	_insert_line_after	_insert_line_prev
_insert_space	_quick_write	

cview.c:

_main

cvmove.c:

_add_horizontal_lines	_back_track	_bottom_right
_clear_to_nulls	_delete_horizontal_lines	_find_corners
_find_move_extent	_lengthen_structure	_move_horizontal
_move_symbol	_move_vertical	_no_corner_collision
_shorten_structure	_top_left	

cvsymbol.c:

_area_is_clear
_column_is_clear
_find_bottom
_find_top
_init_cview_symbol
_init_page
_make_decision_structure
_make_if_para
_make_loop
_make_null_flowgraph
_make_pre_post_bodies
_make_process_switch
_make_then_else
_set_do
_set_for
_set_function_graph
_set_main_graph
_set_while

cvcgen.c:

_cgen
_cvcgen_ok
_cvcgen_set_file_name
_cvcgen_set_indent
_cvcgen_set_indent_limit
_cv_indent
_else_not_empty
_gen_case_func
_gen_code
_gen_prechecked_loop
_gen_then_else
_skip_to_corner
_write_text

cvenv.c:

_get_abstract_status
_get_click_status
_get_current_command
_get_current_expansion
_get_current_file
_get_current_page
_get_debounce_interval
_get_dialog_message
_get_exit_status
_get_graph_dirty
_get_grid_status
_get_indent
_get_indent_inc
_get_indent_limit
_get_mouse_coord
_get_next_page_number
_get_page_title_col
_get_page_title_row
_get_pal_root
_get_pcs_root
_get_root_page
_get_speaker_tone
_get_sps_root
_reset_graph_dirty
_set_current_command
_set_current_expansion
_set_current_file
_set_current_page
_set_debounce_interval
_set_dialog_message
_set_graph_dirty
_set_indent
_set_indent_inc
_set_indent_limit
_set_mouse_coord
_set_next_page_number
_set_page_title_col
_set_page_title_row
_set_pal_root
_set_pcs_root
_set_root_page
_set_speaker_tone
_set_sps_root
_show_page_title
_toggle_abstract
_toggle_click
_toggle_exits
_toggle_grid
_toggle_page_title

Appendix E.

External function to file cross reference. A file name of "clib" indicates a supplied library function.

cvabort.c:

external function	file
dialog_manager	cvdialog.c
int86	clib
sprintf	clib

cvabstr.c:

external function	file
abort_process	cvabort.c
clear_window	cvdraw.c
fill_in_line	cvdelete.c
find_move_extent	cvmove.c
find_structure_extent	cvdelete.c
free	clib
get_abstract_status	cvenv.c
get_current_page	cvenv.c
get_mouse_coord	cvenv.c
init_page	cvsymbol.c
init_text_blocks	cvtext.c
make_null_flowgraph	cvsymbol.c
malloc	clib
redraw_page	cvdraw.c
set_current_page	cvenv.c
sprintf	clib

cvcgen.c:

external function	file
abort_process	cvabort.c
atoi	clib
dialog_manager	cvdialog.c
fcloseall	clib
find_structure_extent	cvdelete.c
fopen	clib
fprintf	clib
fputc	clib
free	clib
get_current_file	cvenv.c
get_dialog_message	cvenv.c
get_indent	cvenv.c
get_indent_inc	cvenv.c
get_indent_limit	cvenv.c
get_root_page	cvenv.c

malloc	clib
set_dialog_message	cvenv.c
set_indent	cvenv.c
set_indent_inc	cvenv.c
set_indent_limit	cvenv.c
sprintf	clib

cvdelete.c:

external function	file
abort_process	cvabort.c
clear_screen	cvdraw.c
clear_window	cvdraw.c
free	clib
free_text_chain	cvtext.c
get_current_expansion	cvenv.c
get_current_page	cvenv.c
get_mouse_coord	cvenv.c
init_cview_symbol	cvsymbol.c
make_null_flowgraph	cvsymbol.c
malloc	clib
redraw_page	cvdraw.c
set_current_page	cvenv.c

cvdialog.c:

external function	file
abort_process	cvabort.c
atoi	clib
box	cvdraw.c
clear_window	cvdraw.c
free	clib
get_mouse_coord	cvmouse.c
hide_cursor	cvmouse.c
locate	cvdraw.c
malloc	clib
mouse_hide	cvmouse.c
mouse_show	cvmouse.c
printf	clib
restore_scr	cvdraw.c
save_scr	cvdraw.c
scanf	clib
show_cursor	cvmouse.c
wait_for_button_one	cvmouse.c

cvdraw.c:

external function	file
get_current_page	cvenv.c
get_exit_status	cvenv.c
get_grid_status	cvenv.c
get_page_title_col	cvenv.c

get_page_title_row	cvenv.c
int86	clib
mouse_hide	cvmouse.c
mouse_show	cvmouse.c
printf	clib
show_page_title	cvenv.c

cvenv.c:

no external functions

cvexpand.c:

external function	file
abort_process	cvabort.c
atoi	clib
clear_window	cvdraw.c
dialog_manager	cvdialog.c
get_current_expansion	cvenv.c
get_current_page	cvenv.c
get_dialog_message	cvenv.c
get_mouse_coord	cvenv.c
init_page	cvsymbol.c
init_text_blocks	cvtext.c
make_null_flowgraph	cvsymbol.c
malloc	clib
redraw_page	cvdraw.c
set_current_expansion	cvenv.c
set_current_page	cvenv.c
set_dialog_message	cvenv.c
sprintf	clib

cvfile.c:

external function	file
abort_process	cvabort.c
dialog_manager	cvdialog.c
fclose	clib
fcloseall	clib
fgetc	clib
fopen	clib
fprintf	clib
free	clib
get_abstract_status	cvenv.c
get_click_status	cvenv.c
get_current_file	cvenv.c
get_debounce_interval	cvenv.c
get_dialog_message	cvenv.c
get_exit_status	cvenv.c
get_graph_dirty	cvenv.c
get_grid_status	cvenv.c
get_indent_inc	cvenv.c
get_indent_limit	cvenv.c
get_page_title_col	cvenv.c
get_page_title_row	cvenv.c
get_pal_root	cvenv.c

get_pcs_root	cvenv.c
get_root_page	cvenv.c
get_speaker_tone	cvenv.c
get_sps_root	cvenv.c
init_cview_symbol	cvsymbol.c
init_page	cvsymbol.c
init_text_blocks	cvtext.c
insert_line_after	cvtext.c
malloc	clib
reset_graph_dirty	cvenv.c
set_current_page	cvenv.c
set_debounce_interval	cvenv.c
set_dialog_message	cvenv.c
set_indent_inc	cvenv.c
set_indent_limit	cvenv.c
set_page_title_col	cvenv.c
set_page_title_row	cvenv.c
set_pal_root	cvenv.c
set_pcs_root	cvenv.c
set_root_page	cvenv.c
set_speaker_tone	cvenv.c
set_sps_root	cvenv.c
show_page_title	cvenv.c
sprintf	clib
sscanf	clib
toggle_abstract	cvenv.c
toggle_click	cvenv.c
toggle_exits	cvenv.c
toggle_grid	cvenv.c
toggle_page_title	cvenv.c

cview.c:

external function	file
abort_process	cvabort.c
clear_screen	cvdraw.c
clear_window	cvdraw.c
exec_command	cvmod.c
get_current_command	cvenv.c
get_current_page	cvenv.c
get_dialog_message	cvenv.c
get_graph_dirty	cvenv.c
hide_cursor	cvmouse.c
init_page	cvsymbol.c
locate	cvdraw.c
make_null_flowgraph	cvsymbol.c
malloc	clib
mouse_buttons	cvmouse.c
mouse_hide	cvmouse.c
mouse_show	cvmouse.c
printf	clib
redraw_page	cvdraw.c
save_file	cvfile.c
set_current_command	cvenv.c
set_current_page	cvenv.c

set_dialog_message	cvenv.c
set_mode	cvenv.c
set_mouse_cursor	cvenv.c
set_mouse_position	cvenv.c
set_root_page	cvenv.c
show_cursor	cvdraw.c
show_menus	cvmenu.c

cvinspec.c:

external function	file
dialog_manager	cvdialog.c
free	clib
get_current_page	cvenv.c
get_mouse_coord	cvenv.c
malloc	clib
sprintf	clib

cvmenu.c:

external function	file
free	clib
get_current_page	cvenv.c
get_mouse_coord	cvenv.c
get_root_page	cvenv.c
load_file	cvfile.c
locate	cvdraw.c
malloc	clib
mouse_buttons	cvmouse.c
mouse_hide	cvmouse.c
mouse_show	cvmouse.c
printf	clib
restore_scr	cvdraw.c
save_as	cvfile.c
save_file	cvfile.c
save_scr	cvdraw.c
set_current_command	cvenv.c
set_current_expansion	cvenv.c
set_current_page	cvenv.c
set_dialog_message	cvenv.c
set_mouse_cursor	cvenv.c
set_mouse_position	cvnev.c
set_options	cvnev.c

cvmod.c:

external function	file
abstract_area	cvabstr.c
delete_symbol	cvdelete.c
edit_text	cvtext.c
expand_symbol	cvexpand.c
get_current_command	cvenv.c

inspect_symbol	cvinspec.c
make_decision_structure	cvsymbol.c
make_process_switch	cvsymbol.c
move_symbol	cvmove.c
set_graph_dirty	cvenv.c
swap_structure	cvswap.c

cvmouse.c:

external function	file
get_click_status	cvenv.c
get_current_command	cvenv.c
get_debounce_interval	cvenv.c
get_speaker_tone	cvenv.c
inp	clib
int86	clib
outp	clib
set_mouse_coord	cvenv.c

cvmove.c:

external function	file
abort_process	cvabort.c
area_is_clear	cvsymbol.c
clear_window	cvdraw.c
free	clib
get_current_page	cvenv.c
get_mouse_coord	cvenv.c
get_mouse_position	cvenv.c
init_cview_symbol	cvsymbol.c
malloc	clib
redraw_page	cvdraw.c
set_current_command	cvenv.c
set_mouse_cursor	cvenv.c
wait_for_button_one	cvmouse.c

cvoption.c:

external function	file
atoi	clib
dialog_manager	cvdialog.c
get_abstract_status	cvenv.c
get_click_status	cvenv.c
get_debounce_interval	cvenv.c
get_exit_status	cvenv.c
get_grid_status	cvenv.c
get_page_title_col	cvenv.c
get_page_title_row	cvenv.c
get_speaker_tone	cvenv.c
itoa	clib
set_debounce_interval	cvenv.c
set_dialog_message	cvenv.c
set_page_title_col	cvenv.c
set_page_title_row	cvenv.c
set_speaker_tone	cvenv.c
show_page_title	cvenv.c

toggle_abstract	cvenv.c
toggle_click	cvenv.c
toggle_exits	cvenv.c
toggle_grid	cvenv.c
toggle_page_title	cvenv.c

cvswap.c:

external function	file
get_exit_status	cvenv.c
abort_process	cvabort.c
add_horizontal_lines	cvmove.c
area_is_clear	cvsymbol.c
clear_window	cvdraw.c
delete_horizontal_lines	cvmove.c
find_structure_extent	cvdelete.c
free	clib
get_current_page	cvenv.c
get_mouse_coord	cvenv.c
malloc	clib
no_corner_collision	cvmove.c
redraw_page	cvdraw.c

cvsymbol.c:

external function	file
abort_process	cvabort.c
clear_window	cvdraw.c
cv_draw_symbol	cvdraw.c
dialog_manager	cvdialog.c
free	clib
get_current_command	cvenv.c
get_current_page	cvenv.c
get_dialog_message	cvenv.c
get_mouse_coord	cvenv.c
get_next_page_number	cvenv.c
init_text_blocks	cvtext.c
malloc	clib
redraw_page	cvdraw.c
set_dialog_message	cvenv.c
sprintf	clib

cvtext.c:

external function	file
abort_process	cvabort.c
clear_screen	cvdraw.c
clear_window	cvdraw.c
dialog_manager	cvdialog.c
free	clib
get_current_page	cvenv.c
get_dialog_message	cvenv.c
get_mouse_coord	cvenv.c
get_page_title_col	cvenv.c
get_page_title_row	cvenv.c
getch	clib

hide_cursor	cvmouse.c
kbhit	clib
locate	cvdraw.c
malloc	clib
mouse_hide	cvmouse.c
mouse_show	cvmouse.c
printf	clib
redraw_page	cvdraw.c
restore_scr	cvdraw.c
save_scr	cvdraw.c
scrgetc	cvdraw.c
scroll_down	cvdraw.c
scroll_up	cvdraw.c
scrputc	cvdraw.c
set_dialog_message	cvenv.c
show_cursor	cvmouse.c

Appendix F.

Sample programs from *The C Programming Language* by Brian Kernighan and Dennis Ritchie. The Cview internal file is listed first followed by the generated C source code.

Cview file for the original program that generates a temperature conversion table.

```
20 7500 23 0 2 40
-1 -1 0 -1 -1
-1 2 12 0
0 13
16 21
~
1 3 12 0
1c ff81
~
1 4 12 0
1c ff81
~
1 5 12 0
1c ff81
~
1 6 12 0
1a ff81
~
1 0 13 0
4 0
/* Fahrenheit-Celsius table
** K&R p. 8
*/
   \
#include "stdio.h"

main()
~
1 1 13 0
4c ff80
int lower, upper, step;
float fahr, celsius;

lower = 0;
upper = 300;
step = 20;

fahr = lower;
~
1 2 13 0
3b ffa0
~
1 6 13 0
27 14
fahr <= upper
~
1 7 13 0
8 ff80
return(0);
```

```
~
1 2 14 0
15 40
~
1 3 14 0
1c 40
~
1 4 14 0
4c 40
celsius = (5.0/9.0) * (fahr-32.0);
printf("%4.0f %6.1f\n", fahr, celsius);
fahr = fahr + step;
~
1 5 14 0
1c 40
~
1 6 14 0
19 10
~
0 0 0 0
```

Generated source code:

```
/* line word and char count
** K&R p.18
*/

#include "stdio.h"

#define YES 1
#define NO  0

main()
{
  int c, nl, nw, nc, inword;

  inword = NO;
  nl = nw = nc = 0;
  /* read data and keep appropriate counts */

  while ((c = getchar()) != EOF)
  {
    ++nc;
    /* count individual words */

    if (c == ' ' || c == '\n' || c == '\t')
    {
      inword = NO;
    }
    else
    {

      if (inword == NO)
      {
        inword = YES;
        ++nw;
      }
    }
```

```
    if (c == '\n')
    {
      ++nl;
    }
  }
  printf("%d %d %d\n", nl, nw, nc);

  return(0);
}
```

Cview file for K&R word, character and line count program.

```
20 7500 23 0 2 40
-1 -1 0 -1 -1
-1 0 13 0
0 13
4 0
/* line word and char count
** K&R p.18
*/

#include "stdio.h"

#define YES 1
#define NO  0

main()
~
1 1 13 0
4c ff80
int c, nl, nw, nc, inword;

inword = NO;
nl = nw = nc = 0;
~
1 2 13 0
1c ff80
~
1 3 13 4
4c ff81
~
1 4 13 0
4c ff80
printf("%d %d %d\n", nl, nw, nc);
~
1 5 13 0
1c ff80
~
1 6 13 0
1c ff80
~
1 7 13 0
8 ff80
return(0);
~
-4 1 12 0
0 13
```

```
16 21
~
4 2 12 0
1c ff81
~
4 3 12 0
1c ff81
~
4 4 12 0
1c ff81
~
4 5 12 0
1c ff81
~
4 6 12 0
1a ff81
~
4 0 13 0
4 0
/* read data and keep appropriate counts */
~
4 1 13 0
3b ffa0
~
4 6 13 0
27 14
(c = getchar()) != EOF
~
4 7 13 0
8 ff80
~
4 1 14 0
15 40
~
4 2 14 5
4c 41
~
4 3 14 5
4c 41
~
4 4 14 0
4c 40
++nc;
~
4 5 14 0
1c 40
~
4 6 14 0
19 10
~
-5 3 12 0
7 13
16 40
~
5 4 12 0
1c 40
~
5 5 12 0
1a 20
```

```
~
5 0 13 0
4 40
~
5 1 13 0
1c 0
~
5 2 13 0
1c 0
~
5 3 13 0
3b 30
~
5 5 13 0
27 40
c == '\n'
~
5 6 13 0
1c 0
~
5 7 13 0
8 0
~
5 3 14 0
15 40
~
5 4 14 0
4c 40
++nl;
~
5 5 14 0
19 10
~
-5 2 11 0
0 13
16 20
~
5 3 11 0
1c ff80
~
5 4 11 0
1c ff80
~
5 5 11 0
1a ff80
~
5 1 12 0
16 20
~
5 2 12 0
2b ff80
inword == NO
~
5 5 12 0
37 30
~
5 6 12 0
1a ff80
~
```

```
5 0 13 0
4 0
/* count individual words */
~
5 1 13 0
2b ff80
c == ' ' || c == '\n' || c == '\t'
~
5 2 13 0
15 10
~
5 3 13 0
4c ff80
inword = YES;
++nw;
~
5 4 13 0
1c ff80
~
5 5 13 0
19 ff80
~
5 6 13 0
37 30
~
5 7 13 0
8 ff80
~
5 1 14 0
15 10
~
5 2 14 0
1c ff80
~
5 3 14 0
4c ff80
inword = NO;
~
5 4 14 0
1c ff80
~
5 5 14 0
1c ff80
~
5 6 14 0
19 ff80
~
0 0 0 0
```

Generated source code:

```
/* line word and char count
** K&R p.18
*/

#include "stdio.h"

#define YES 1
```

```
#define NO  0

main()
{
  int c, nl, nw, nc, inword;

  inword = NO;
  nl = nw = nc = 0;
  /* read data and keep appropriate counts */

  while ((c = getchar()) != EOF)
  {
    ++nc;
    /* count individual words */

    if (c == ' ' || c == '\n' || c == '\t')
    {
      inword = NO;
    }
    else
    {

      if (inword == NO)
      {
        inword = YES;
        ++nw;
      }
    }

    if (c == '\n')
    {
      ++nl;
    }
  }
  printf("%d %d %d\n", nl, nw, nc);

  return(0);
}
```

Appendix G.

Samples of original code written by the author for a class in Programming Language Theory vs. the same code generated from Cview.

```
/* ========================================================== */
/*                                                            */
/* author      : walter e. martin                             */
/* date        : 07/08/84                                     */
/* description: this is the "missing UNIX tool", field.       */
/*              the user can specify up to 256 field num-     */
/*              bers (unordered with repeats) that will       */
/*              be extracted from the input and printed.      */
/*                                                            */
/* ========================================================== */
#include <stdio.h>
#define MAXLINE 4096
#define MAXOPS  256
/* ========================================================== */
/*                                                            */
/* main does the following:                                   */
/*                            1) if no arguments are present  */
/*                               print a "usage" message.     */
/*                            2) convert arguments to ints    */
/*                               and store in array "ops"     */
/*                            3) read lines and call item-    */
/*                               scan to extract fields       */
/*                            4) print fields as the are ex-  */
/*                               tracted                      */
/*                                                            */
/* ========================================================== */
main(argc,argv)
int argc;
char *argv[];.
{
        int ops[MAXOPS];
        int count,len;
        char line[MAXLINE],item[MAXLINE];
        if (argc == 1)
        {

                /* ==================== */
                /* print "usage" messge */
                /* ==================== */
                printf("Usage: field argl arg2 . . . argn\n");
                printf("Args are field numbers in the input\n");
                Printf("record.  Fields are delimited
                byspaces.\n");
                printf("Up to 256 fields may be specified.\n");
        }
        else
        {

                /* ================================= */
                /* convert input arguments to integers */
                /* ================================= */
                for (count = 0; count<argc-1; count++)
```

```
                {
                        ops[count] = stoi(*++argv);
                        ops[(count+1)] = -1;
                }

                /*=============================================*/
                /* while not end of file read and process
                records*/
                /*=============================================*/
                while ((len = getline(line,MAXLINE))>0)
                {
                        count = 0;

                      /*
                      ==========================================*/
                      /* for each argument extract the
                      associated*/
                      /* field and print it
                      */
                      /*
                      ==========================================*/
                        do
                          {
                                  itemscan(line,ops[count],item);
                                  printf((count<argc-2) ? "%s " :
                                  "%s\n",item);
                           }
                           while (ops[++count] != -1);
                }
        }
}
/* ============================================================ */
/*                                                              */
/* author       : kernighan & ritchie (modified by w. martin)   */
/* date         : 07/06/84                                      */
/* description : this is a modified version of getline found    */
/*                  on page 26 of "The C Programming Language.  */
/*                                                              */
/* ============================================================ */
getline(s, lim)
char s[];
int  lim;
{
        int c, i;
        for (i = 0; i<lim-1 && (c=getchar())  != EOF && c != '\n';
        ++i)
                s[i] = c;
        s[i] = '\0';
        return(i);
}
/* ============================================================ */
/*                                                              */
/* author       : kernighan & ritchie (modified by w. martin)   */
/* date         : 7/6/84                                        */
/* description : stoi (string to integer) is a version of       */
/*                 atoi from page 39 of "The C Programming      */
/*                 Language."                                   */
/*                                                              */
/* ============================================================ */
```

```
stoi(s)
char s[];
{
            int i, n;
            n = 0;
            for (i = 0; s[i] >= '0' && s[i] <= '9'; ++i)
                  n = 10 * n + s[i] - '0';

            /* ======================================== */
            /*  if a string did not totally process or  */
            /*  if an input parameter was zero abort    */
            /* ======================================== */
            if (s[i] != '\0'  || n == 0)
            {
                  fprintf(stderr,"illegal field number!\n");
                  exit(1);
            }
            return(n);
}
/* ============================================================== */
/*                                                                */
/* author      : walter e. martin                                 */
/* date        : 7/7/84                                           */
/* description : see below for detailed comments                  */
/*                 This is at best hacker code.  The input        */
/*                 string is rescanned for every argument which   */
/*                 is at best inefficient, but it is compact      */
/*                 and relatively easy to understand.             */
/*                                                                */
/* ============================================================== */
itemscan(lne,num,itm)
char *lne, *itm;
int num;
{
            char *titm;
            int fieldcount = 0;

            /* ====================================== */
            /* save the pointer to the returned field */
            /* find the beginning of the first field  */
            /* ====================================== */
            titm = itm;
            while (*lne == ' ')
            ++lne;

            /* ======================================== */
            /* do until the end of string is found or   */
            /* until the desired string is extracted    */
            /* 1) point itm to the output string        */
            /* 2) move the current field to itm         */
            /* one character at a time                  */
            /* 3) increment the field count             */
            /* 4) find the beginning of the next        */
            /* field                                    */
            /* ======================================== */
            do
                {
                  itm = titm;
                  while(*lne != ' ' && *lne != '\0')
```

```
                    *itm++ = *lne++;
               fieldcount += 1;
               while(*lne == ' ')
                    lne++;
          }
          while(*lne != '\0' && fieldcount != num);

          /* ======================================== */
          /* if the end of string was found before    */
          /* the requested field was found set the    */
          /* returned field to a null string.         */
          /* ======================================== */
          if (fieldcount != num)
               itm = titm;
          *itm = '\0';
          return;
}
```

This is the FIELD program generated from Cview using an indent of two and an indent limit of forty.

```
/* author      : Walter E. Martin
** date        : 8/30/87
** description: Redo of the "missing UNIX command", field.
*/

#include <stdio.h>
#define  MAXLINE 4096
#define  MAXOPS  256

main(argc, argv)
int argc;
char *argv[];

/* main does the following:
**
**  1) if no arguments are present print a "usage message."
**  2) convert arguments to ints and store in array "ops."
**  3) read lines and call itmescan to extract fields.
**  4) print fields as they are extracted.
**
*/
{
  int ops[MAXOPS];
  int count, len;
  char line[MAXLINE], item[MAXLINE];

  if (argc == 1)
  {
    /*
    ** print "usage message"
    */

    printf("Usage: field arg1 arg2 . . .argn\n");
    printf("Args are field numbers in the input\n");
    printf("record.  Fields are delimited by spaces.\n");
    printf("Up to 256 fields may be specified.\n");
  }
```

```
  else
  {
    /*
    ** convert input arguments to integers
    */

    for (count = 0; count<argc-1; count++)
    {
      ops[count] = stoi(*++argv);
      ops[count+1] = -1;
    }
    /*
    ** read and process lines of input until EOF is reached
    */

    while ((len = getline(line, MAXLINE)) > 0)
    {
      count = 0;

      do
      {
        itemscan(line, ops[count], item);
        printf((count<argc-2) ? "%s " : "%s\n", item);
      }
      while (ops[++count] != -1);
    }
  }

  return(0);
}

getline(s, lim)
char s[];
int  lim;
/*
** author      : K&R (modified by W.M.)
** date        : 8/30/87
** description: Modified version of getline found on p. 26 of K&R.
*/
{
  int c, i;

  for (i = 0; i<lim-1 && (c = getchar()) != EOF && c != '\n'; ++i)
  {
    s[i] = c;
  }
  s[i] = '\0';
  return(i);
}

stoi(s)
char s[];
/*
** author      : K&R (modified by W.M.)
** date        : 8/30/87
** description: stoi is a version of atoi from p. 39 of K&R
*/
{
  int i, n;
```

```
  n = 0;

  for (i = 0; s[i] >= '0' && s[i] <= '9'; ++i)
  {
    n = 10 * n + s[i] - '0';
  }
  /* string did not process */
  /*
  ** if a string did not totally process or if an
  ** input parameter was zero, abort
  */

  if (s[i] != '\0' || n == 0)
  {
    fprintf(stderr, "illegal field number!\n");
    exit(1);
  }
  return(n);
}

itemscan(lne, num, itm)
char *lne, *itm;
int num;
/*
** author      : Walter E. Martin
** date        : 8/30/87
** description: See below for detailed comments.  This is at best
**              hacker code.  The input string is rescanned
**              for every argument which is at best inefficient,
**              but is compact and relatively easy to understand.
*/
{
  char *titm;
  int fieldcount = 0;

  /*
  ** save the pointer to the returned field
  ** find the beginning of the first field
  */

  titm = itm;

  while (*lne == ' ')
  {
    ++lne;
  }
  /*
  ** do until the end of string is found or until
  ** the desired string is extracted
  **   1) point itm to the output string
  **   2) move the current field to itm one
  **      character at a time
  **   3) increment the field count
  **   4) find the beginning of the next field
  */

  do
  {
```

```
    itm = titm;

    while (*lne != ' ' && *lne != '\0')
    {
      *itm++ = *lne++;
    }
    fieldcount += 1;

    while (*lne == ' ')
    {
      lne++;
    }
  }
  while (*lne != '\0' && fieldcount != num);
  /*
  ** if the end of string was found before the
  ** requested field was found, set the
  ** returned field to a null string.
  */

  if (fieldcount != num)
  {
    itm = titm;
  }
  *itm = '\0';
  /* expanded process */
  return(0);
}
```

Original source code written by the author:

```
/* ========================================================== */
/*  name        : Walter E. Martin                            */
/*  date        : 07/07/84                                    */
/*  description : FMTX is a simple word processor that reads  */
/*                input and reformates it into columns whose  */
/*                width is in the range 30 to 72 (default is  */
/*                72.)  As an option the output can be cen-   */
/*                tered on the page.                          */
/*                                                            */
/* ========================================================== */
#include <stdio.h>
#define MAXLINE     4096
#define MAXCOL        72
#define MINCOL        30
main(argc,argv)
int argc;
char *argv[];
{
        int colwidth = MAXCOL, centered = 0, rightjust = 0;
        char *option;

/* ========================================================== */
/* if options have been supplied, pull them out, setting the  */
/* the appropriate flags.  if an illegal option has been      */
/* found, issue a message to the terminal.                    */
/* ========================================================== */
        if (argc != 1)
```

```
        {
           if ((*++argv)[0] == '-')
           {
                     for (option = argv[0]+1; *option != '\0';
                                  option++)
                            switch(*option)
                             {
                            case 'c' :
                                    centered = 1;
                                    break;
                            case 'r' :
                                    fprintf(stderr,"r - not
                                    implemented\n"
                                    rightjust = 1;
                                    break;
                             default
                               fprintf(stderr,"illegal option ");
                               fprintf(stderr,"'%c'found\n",*option
                               );
                               fprintf(stderr,"usage:  fmtx -cr
                               width\n");

                               exit(1);
                             }
              ++argv;
              --argc;
           }

/* ========================================================== */
/* if a column width has been supplied convert it from a      */
/* string to an integer.  if a non-numeric was detected or    */
/* if the number is out of range, issue a message to the      */
/* terminal and exit.                                         */
/* ========================================================== */
           if (argc != 1)
           {
           colwidth = 0;
           for (option = argv[0]; *option >= '0' && *option <= '9';
           option++)
                 colwidth = 10 * colwidth + *option - '0';
           if (*option != '\0' || colwidth < MINCOL || colwidth
           >MAXCOL)
           {
                 fprintf(stderr,"illegal column width\n");
                 fprintf(stderr,"30 <= width <= 72\n");
                 exit(2);
           }
          }
         }


/* ========================================================== */
/* the following code is loosely based on a state diagram     */
/* which consists of seven states.  each state is des-        */
/* cribed below.                                              */
/* ========================================================== */
{
           int len, spf = 0, count = 0, state = 1;
           char line[MAXLINE+1], fmtline[MAXCOL+1];
```

```
            register char *input, *output;
            output = &fmtline[0];

/* ========================================================== */
/* repeat the execution of the switch until state is set      */
/* to zero.                                                   */
/* ========================================================== */
            do
                  switch (state)
            {
```

```
            case 1 :
/* ========================================================== */
/* if not end of file                                         */
/*          if the input is a blank line                      */
/*                    if output buffer not empty              */
/*                              state = 7                     */
/*                    else                                    */
/*                              state = 1                     */
/*          else                                              */
/*                    state = 2                               */
/*else                                                        */
/*          state = 3                                         */
/* ========================================================== */
                    if ((len=getline(line,MAXLINE)) != 0)
                    {
                    input = &line[0];
                    spf = 0;
                    while (*input != '\0')
                    {
                         if (*input != ' ')
                              spf = 1;
                          input++;
                    }
                    if (spf)
                    {
                          input = &line[0];
                              state = 2;

                    }
                    else
                             if (output != &fmtline[0])
                             state = 7;
                    }
                    else
                              state = 3;
                    break;

            case 2 :
/* ========================================================== */
/* while the input string is not exhausted and the output     */
/* string is not filled move characters.                      */
/*                                                            */
/* if the input was exhausted before the column filled        */
/* change to state = 1 (get another line.)                    */
/* else                                                       */
/*          state = 3                                         */
/* ========================================================== */
                  while (*input != '\0' && count++ < colwidth)
                        *output++ = *input++;
                  if (*input == '\0')
                  {
                              state = 1;
                              *output++ = ' ';
                               count++;
                  }
                  else
                            {
                                  *output = '\0';
                                  state = 3;
```

```
                        }
                break;


          case 3 :
/* ========================================================== */
/* if not end of file                                         */
/*          then if the output buffer contains no spaces      */
/*                  and the next char from input is not       */
/*                  null or space, issue word too long        */
/*                  message and change to state = 6           */
/*                                                            */
/* change state according to the appropriate flag             */
/* ========================================================== */
          if (len == 0)
                *output = '\0';
          else
          {
                spf = 0;
                for (output = &fmtline[0]; *output != '\0';
                output++)
                if (*output == ' ') spf = 1;
                if (!spf && *input != '\0' && *input != ' ')
                {
                        fprintf(stderr,"word longer than
                        colum\n");
                        fprintf(stderr,"splitting word across
                        lines!\n");
                        state = 6;

                else
                while (*--output != ' ')
                {
                        --input;
                        *output = ' ';
                }
          }
          if (rightjust)
                state = 4;
          else
                if (centered)
                      state = 5;
                else
                      state = 6;
          break;
```

```
            case 4 :
/* ============================================================ */
/* right justify the output buffer then if centered output      */
/* has been requested, state = 5                                */
/* ============================================================ */
/* place call to right justify here */
                    if (centered)
                        state = 5;
                    else
                         state = 6;
                    break;


            case 5 :
/* ============================================================ */
/* write leading spaces to center the column then state = 6     */
/* ============================================================ */

                    center(colwidth);
                    state = 6;
                    break;


            case 6 :
/* ============================================================ */
/* print the column, reset the column count and buffer          */
/* pointer.                                                     */
/*                                                              */
/* if end of file exit via state = 0 else continue moving       */
/* characters from the input buffer to the column               */
/* ============================================================*/
                    printf("%s\n",fmtline);
                    output = &fmtline[0];
                    count = 0;
                    if (len == 0)
                             state = 0;
                    else
                            state = 2;
                    break;


            case 7 :
/* ============================================================ */
/* special case - flush a partial column because a blank line   */
/* was read.                                                    */
/*                                                              */
/* center the output if needed and print the partial column.    */
/* reset the counts and pointers then set state = 1.  this will */
/* get the next record from stdin.                              */
/* ============================================================ */
                    *output = '\0';
                /* place call to right justify here */
                    if (centered)
                    center(colwidth);
                    printf("%s\n",fmtline);
                    printf(" \n");
                    output = &fmtline[0];
                    count = 0;
                    state = 1;
```

```
                break;
            }
            while (state != 0);
        }
}
/* ============================================================== */
/*                                                                */
/* author      : kernighan & ritchie (modified by w. martin)      */
/* date        : 07/06/84                                         */
/* description : this is a modified version of getline found      */
/*                  on page 26 of "The C Programming Language.    */
/*                                                                */
/* ============================================================== */
getline(s, lim)
char s[];
int  lim;
{
        int c, i;
        for (i = 0; i<lim-1 && (c=getchar())  != EOF && c != '\n';
        ++i)
        s[i] = c;
        s[i] = '\0';
        return(i);
}
/* ============================================================== */
/*                                                                */
/* author      : walter e. martin                                 */
/* date        : 7/8/84                                           */
/* description : this routine writed spaces to the output de-     */
/*                vice equal to one half the difference between   */
/*                the maximum column width and the formatted      */
/*                width.used to center output on a printer.       */
/*                                                                */
/* ============================================================== */
center(cw)
int cw;
{
int lsp;
for (lsp = 1; lsp <= (MAXCOL - cw) / 2; lsp++)
 printf(" ");
}
```

The following is the FMTX program as it was generated from Cview using two character indentation and an indent limit of fourty characters.

```
/* FMTX:  A simple word processor that reads
**        input and reformats it into columns
**        whose width is in the range 30 to 72
**        (default is 72.)  As an option the output
**        can be centered on the page.
** author: Walter E. Martin
** date  : 7/7/84
*/

#include <stdio.h>
#define  MAXLINE  4096
#define  MAXCOL     72
```

```
#define  MINCOL     30

main(argc, argv)
int argc;
char *argv[];
{
  int  colwidth  = MAXCOL;
  int  centered  = 0;
  int  rightjust = 0;
  int  len;
  int  spf = 0;
  int  count = 0;
  int  state = 1;
  char line[MAXLINE+1];
  char fmtline[MAXCOL+1];
  char *option;
  register char *input;
  register char *output;
  output = &fmtline[0];

  /* if options have been supplied, pull them out, setting the
  ** appropriate flags. If an illeagal option has been found,
  ** issue a message to the terminal.
  */

  if (argc != 1)
  {

    if ((*++argv)[0] == '-')
    {


      for (option = argv[0]+1; *option != '\0'; option++)
      {

        switch(*option)
        {

          case  'c':
          {
            centered = 1;
            break;
          }

          case  'r':
          {
            fprintf(stderr,"r - not implemented\n");
            rightjust = 1;
            break;
          }

          default:
          {
            fprintf(stderr,"illegal option ");
            fprintf(stderr,"'%c' found\n", *option);
            fprintf(stderr,"usage:  fmtx -cr width\n");
            exit(1);
          }
        }
```

```
      }
      ++argv;
      --argc;
    }
    /* if a column width has been supplied convert it from a
    ** string to an integer.  If a non-numeric was detected or
    ** if the number is out of range, issue a message to the
    ** terminal and exit.
    */

    if (argc != 1)
    {
      colwidth = 0;

      for (option = argv[0]; *option >= '0' && *option <= '9';
option++)
      {
        colwidth = 10 * colwidth + *option - '0';
      }

      if (*option != '\0' || colwidth < MINCOL || colwidth >MAXCOL)
      {
        fprintf(stderr,"illegal column width\n");
        fprintf(stderr,"30 <= width <= 72\n");
        exit(2);
      }
    }
  }

  /* repeat the execution of the switch until state is set
  ** to zero.
  */

  do
  {

    switch(state)
    {

      case  1:
      {
        /* if !EOF
        **   if input == ' '
        **     if output !empty
        **       state = 7
        **     else
        **       state = 1
        **   else
        **     state = 2
        ** else
        **   state = 3
        */


        if ((len = getline(line, MAXLINE)) != 0)
        {
          input = &line[0];
          spf = 0;
```

```
      while (*input != '\0')
      {

        if (*input != ' ')
        {
          spf = 1;
        }
        input++;
      }

      if (spf)
      {
        input = &line[0];
        state = 2;
      }
      else
      {

        if (output != &fmtline[0])
        {
          state = 7;
        }
      }

      if (spf)
      {
        input = &line[0];
        state = 2;
      }
      else
      {

        if (output != &fmtline[0])
        {
          state = 7;
        }
      }
    }
    else
    {
      state = 3;
    }
    break;
}

case  2:
/* while the input string is not exhausted and the output
** string is not filled move characters.
**
** if the input was exhausted before the column filled
** change to state = 1 (get another line) else state = 3
*/
{

  while (*input != '\0' && count++ < colwidth)
  {
    *output++ = *input++;
  }
```

```
  if (*input == '\0')
  {
    state = 1;
    *output++ = ' ';
    count++;
  }
  else
  {
    *output = '\0';
    state = 3;
  }
  break;
}

case  3:
/* if !EOF then
**   if the ouput buffer contains no spaces and the next char
**     from input is not null or space, issue word too long
**     message and change state to 6
*/
{

  if (len == 0)
  {
    *output = '\0';
  }
  else
  {
    spf = 0;

    for (output = &fmtline[0]; *output != '\0'; output++)
    {

      if (*output == ' ')
      {
        spf = 1;
      }
    }

    if (!spf && *input != '\0' && *input != ' ')
    {
      fprintf(stderr,"word longer than column!\n");
      fprintf(stderr,"splitting word across lines!\n");
      state = 6;
    }
    else
    {

      while (*--output != ' ')
      {
        --input;
        *output = ' ';
      }
    }
  }

  if (rightjust)
  {
    state = 4;
```

```
    }
    else
    {

      if (centered)
      {
        state = 5;
      }
      else
      {
        state = 6;
      }
    }
    break;
}

case  4:

/* right justify the output buffer then if centered
** state = 5
*/
{

    if (centered)
    {
      state = 5;
    }
    else
    {
      state = 6;
    }
    break;
}

case  5:
/* write leading spaces to center the col
** then change to state 6
*/
{
    center(colwidth);
    state = 6;
    break;
}

case  6:
/* print the column, reset the column count and buffer
** pointer
**
** if EOF exit via state 0 else continue moving
** characters from the input buffer to the column
*/
{
    printf("%s\n",fmtline);
    output = &fmtline[0];
    count = 0;

    if (len == 0)
    {
      state = 0;
```

```
        }
        else
        {
          state = 2;
        }
        break;
      }

      case  7:
      /* special case - flush a partial column because a blank line
      ** was read center the output if needed and print the
      ** partial column
      **
      ** reset the counts and pointers then set state = 1; this
will
      ** get the next record from stdin.
      */
      {
        *output = '\0';
        /* place call to right justify here */


        if (centered)
        {
          center(colwidth);
        }
        printf("%s\n",fmtline);
        printf(" \n");
        output = &fmtline[0];
        count = 0;
        state = 1;
        break;
      }
    }
  }
  while (state != 0);

  return(0);
}

getline(s, lim)
char s[];
int  lim;
/* author      : k&r (modified by w. martin)
** date        : 7/6/84
** description : modified version of getline found on
**               page 26 of "The C Programming Language"
*/
{
  int c;
  int i;

  for (i = 0; i<lim-1 && (c=getchar()) != EOF && c != '\n'; ++i)
  {
    s[i] = c;
  }
  s[i] = '\0';
  return(i);
}
```

```
center(cw)
int cw;
/* author       : walter e. martin
** date         : 7/8/84
** description  : this routine writes spaces to the output device
**                equal to half the difference between the maximum
**                column width and the formatted width.
*/
{
  int lsp;

  for (lsp = 1; lsp <= (MAXCOL - cw) >> 1; lsp++)
  {
    printf(" ");
  }
  return(0);
}
```

Appendix H.

The this sample divides into a parent and a child process then the child process divides into a parent and a grandchild. Each process prints an indicative message until interrupted. The same process or one of the other two processes may get control.

The Cview data file looks like the following:

```
20 7500 23 0 2 40
-1 -1 0 -1 -1
-1 3 6 0
0 13
16 20

i 4 6 6
4c ff81

i 5 6 0
1a ff80

i 3 7 0
13 20

i 5 7 0
13 10

i 3 8 0
13 20

i 5 8 0
13 10

i 2 9 0
16 20

i 3 9 0
6b ff80

i 5 9 0
77 30

i 6 9 0
1a ff80

i 2 10 0
13 20

i 3 10 0
13 10

i 5 10 0
13 20

i 6 10 0
13 10
```

```
1 2 11 0
13 20

i 3 11 0
15 10

i 4 11 7
4c ff81

i 5 11 0
19 ff80

i 6 11 0
13 10

i 2 12 0
13 20

i 6 12 0
13 10

i 0 13 0
4 0
#include <stdio.h>

main()

i 1 13 0
1c ff80

i 2 13 0
6b ff80

i 6 13 0
77 30

i 7 13 0
8 ff80
return(0);

i 2 14 0
13 10

i 6 14 0
13 20

i 2 15 0
13 10

i 6 15 0
13 20

i 2 16 0
13 10

i 6 16 0
13 20

i 2 17 0
```

```
15 10

i 3 17 0
1c ff80

i 4 17 8
4c ff81

i 5 17 0
1c ff80

i 6 17 0
19 ff80

-8 2 12 0
0 13
16 21

8 3 12 0
1c ff81

8 4 12 0
1c ff81

8 5 12 0
1c ff81

8 6 12 0
1a ff81

8 0 13 0
4 0

8 1 13 0
4c ff80
int i;

8 2 13 0
3b ffa0

8 6 13 0
27 18
i = 0; i < 25; i++

8 7 13 0
8 ff80

8 2 14 0
15 40

8 3 14 0
1c 40

8 4 14 0
4c 40
printf("Grandparent processing.\n");

8 5 14 0
1c 40
```

```
8 6 14 0
19 10

-7 2 12 0
0 13
16 21

7 3 12 0
1c ff81

7 4 12 0
1c ff81

7 5 12 0
1c ff81

7 6 12 0
1a ff81

7 0 13 0
4 0

7 1 13 0
4c ff80
int i;

7 2 13 0
3b ffa0

7 6 13 0
27 18
i = 0; i < 25; i++

7 7 13 0
8 ff80

7 2 14 0
15 40

7 3 14 0
1c 40

7 4 14 0
4c 40
printf("Parent processing.\n");

7 5 14 0
1c 40

7 6 14 0
19 10

-6 2 12 0
0 13
16 21

6 3 12 0
1c ff81
```

```
.
6 4 12 0
1c ff81

.
6 5 12 0
1c ff81

.
6 6 12 0
1a ff81

.
6 0 13 0
4 0

.
6 1 13 0
4c ff80
int i;

.
6 2 13 0
3b ffa0

.
6 6 13 0
27 18
i = 0; i < 25; i++

.
6 7 13 0
8 ff80

.
6 2 14 0
15 40

.
6 3 14 0
1c 40

.
6 4 14 0
4c 40
printf("Grandchild processing.\n");

.
6 5 14 0
1c 40

.
6 6 14 0
19 10

.
0 0 0 0
```

The code generated from the above Cview data file is:

```
#include <stdio.h>

main()
{
  {
    int local_pid;

    if ((local_pid = fork()) != 0)
    {
      int i;
```

```
      for (i = 0; i < 25; i++)
      {
        printf("Grandparent processing.\n");
      }
      wait(0);
    }
    else
    {
      {
        int local_pid;

        if ((local_pid = fork()) != 0)
        {
          int i;

          for (i = 0; i < 25; i++)
          {
            printf("Parent processing.\n");
          }
          wait(0);
        }
        else
        {
          int i;

          for (i = 0; i < 25; i++)
          {
            printf("Grandchild processing.\n");
          }
          exit(0);
        }
      }
      exit(0);
    }
  }

return(0);
}
```