LIFE *Line* 2

by
Carl Helmers
Editor, BYTE

LIFE Line 1 (BYTE #1) presented the general picture of the LIFE program application of your computer. That picture includes the rules of the game, methods of interactively entering graphic data, major software components in verbal description and some of the hardware requirements of the game. In this installment, the discussion turns to some of the program design for the LIFE application. The discussion starts "at the top" (overall program flow) and works down to more detailed levels of design, concentrating upon the "evolution algorithm" which generates new patterns from old patterns.

As in the previous LIFE Line, the goal of the article series is as much to explain and instruct as it is to elaborate upon this one particular system. This article concentrates on the program design process as illustrated by a real LIFE example.

What Is This Process — Designing A Program?

For the readers who are only just now beginning to learn the programming of computers, an elementary acquaintance with some machine's language, a BASIC interpreter, or high level languages would tend to give the impression that programming is fundamentally simple. It is! To write a program which fills a single page of listing — whatever the language or machine architecture involved — is not a tremendously difficult task. When it comes to more complex projects — say 1000 or more words of hand or machine-generated code on your microcomputer — the problem is how to preserve the blissful innocence of simplicity in the face of the worldly forces of complexity.

When you begin to talk about programs more complex than a one page assembly or machine code listing of some specialized service routine or simple "gimmick" program (see the Kluge Harp article in this issue), the complexities and subtleties of scale begin to enter into the programming art. For an application such as the LIFE program, proceeding from the vague notion "I want this application" to a working program can be done in innumerable ways — many of which will work quite well. This is the first ambiguity of scale — where do you head as you start programming? Unless you have a unique parallel processing mind, you can't possibly concentrate on the whole problem of programming at once.

In order to make a big application program work, you have to select "bits and pieces" of the desired result, figure out what they do and how they fit into the big picture, then program them one by one. These little pieces of the program — its "modules" — are like the multiple layers of stone blocks in a pyramid. In fact, defining what to do is very much like the tip of some Egyptian tyrant's tomb in the spring flood . . . as the murky generalities recede, more and more of the structure of the program is defined and clarified. Fig. 1 illustrates the pyramid of abstractions at the start of a program design process. The top layer is clear — a LIFE program is the desired goal. The next layer down is for the most part visible through the obscuring water. But the details of the base of the pyramid — while you know they have to be there in some form — are not at all visible at the start. The design process moves the logical "water level" surrounding the pyramid lower and lower as you figure out more and more of the detail content of the program.

Start at the Top . . .

In LIFE Line 1, I mentioned two major functions which compose a practical LIFE program — data entry and manipulation is one, the LIFE evolution algorithm is the second. Together, these functions define the "program control" layer of the LIFE pyramid. Fig. 2 is a flow chart illustrating the program control algorithm which is the top level of the program structure. Although the diagram — and the algorithm — are extremely simple, they

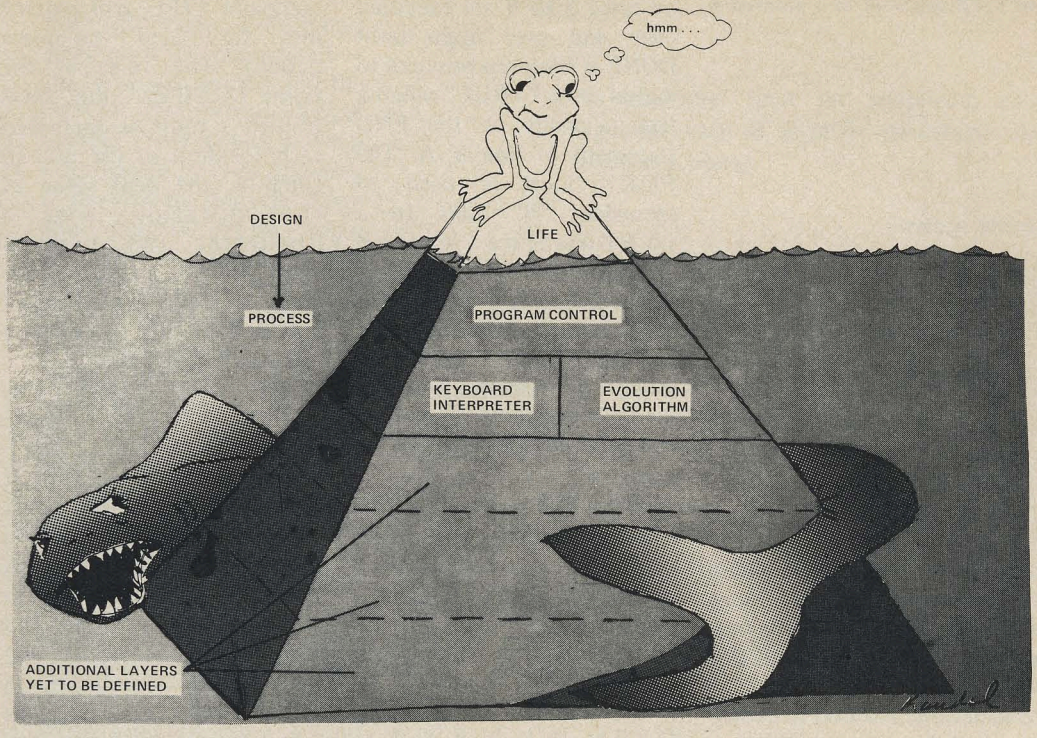


Fig. 1. Defining what to do is like the tip of some Egyptian tyrant's tomb in the spring flood . . . as the murky generalities recede more and more of the structure is defined and clarified . . .

serve a very useful purpose in the program design process: *This high level design has split most of the programming work into two moderately large segments, each of which is less complicated than the whole program.* This view of the problem now gives us two major components upon which to concentrate attention once the top level routine is completed. The program control algorithm of Fig. 2, elaborated in Fig. 3, is the "mortar" which cements together these two blocks of function.

The LIFE program is entered by one of a number of methods. Fig. 2 illustrates branch or jump possibilities from a systems program called a "monitor," "executive" or "operating system" – the preferred way once you get such a system generated. If your system runs "bare bones" with little system-resident software, you might select the starting point and activate the program through use of hardware restart mechanisms and a front panel console.

The first module of the LIFE application to be entered is the KEYBOARD_INTERPRETER, a set of routines which is used to define the initial content of the LIFE grid using interactive commands and the scope display output. The KEYBOARD_INTERPRETER eventually will receive a "GO" command or an "END" command from the user – whereupon it will return to the main routine with the parameters "DONE" and "N" defined. If "N" is greater than zero, control flows to the evolution process – and "N" generations of LIFE will be computed and displayed as they are completed. After the "N" generations have been completed, the scope display and the LIFE grid have the last completed results. If the program is not "DONE," control flows back to the KEYBOARD_INTERPRETER for modification of the data, clearing the screen and starting over, or other operations. If the program is "DONE" then the control flows back to the systems programs – or to a halt point.

This program control algorithm is elaborated in more explicit detail in Fig. 3.

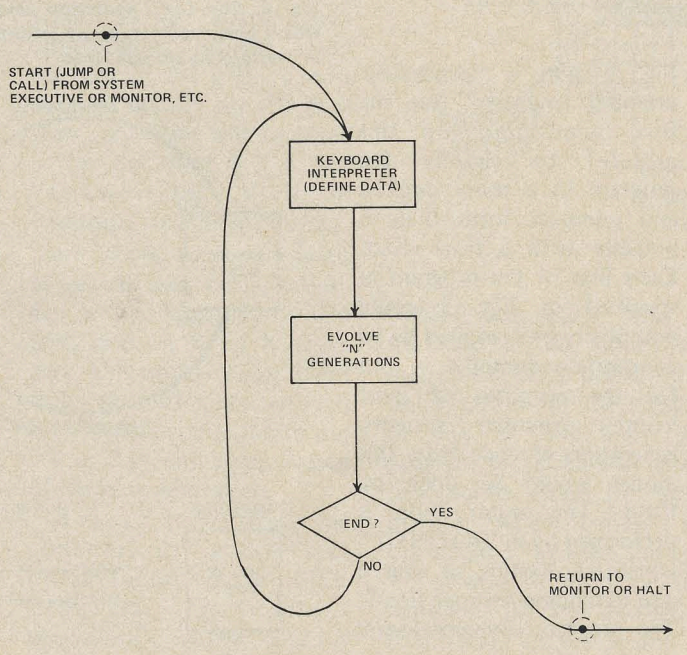


Fig. 2. LIFE program flow of control.

Fig. 3. The main control routine of LIFE specified in a procedure-oriented language . . .

```
1  LIFE:
2     PROGRAM;
3        DONE = FALSE;
4        DO UNTIL DONE = TRUE;
5           CALL KEYBOARD_INTERPRETER (N,DONE);
6           DO FOR I = 1 TO N;
7              CALL GENERATION;
8           END;
9        END;
10       RETURN; / * TO EXECUTIVE, MONITOR, OR JUST HALT */
11       CLOSE LIFE;
```

Subroutines Referenced by LIFE:

KEYBOARD_INTERPRETER . . . This is the routine which looks at the interactive keyboard and interprets user actions such as specifying initial patterns, modifying patterns, etc. N is defined by the GO command which causes return from this subroutine to LIFE.

GENERATION . . . This is the routine which is used to evolve one generation of the life matrix and display the result. Since the entire matrix is kept in software by *GENERATION* until after a new matrix has been evolved, there will never be any partially updated patterns on the scope.

Data (8 bit bytes) used by LIFE at this level:

FALSE – the value "0".

TRUE – the value "1".

DONE – variable set by *KEYBOARD_INTERPRETER* after a user command (GO) to start execution.

N – a variable set by user interaction in *KEYBOARD_INTERPRETER* giving the number of generations to evolve.

I – a temporary loop index variable.

. . . the problem is how to preserve the blissful innocence of simplicity in the face of the wordly forces of complexity.

Fig. 3 uses a "procedure-oriented language" (see the box accompanying this article) to specify the program in a more explicit and compact form than is possible with a flow chart. Each line of the program as specified in Fig. 3 could potentially be compiled by an appropriate compiler – but for the purposes of most home computer systems, generation of code from this model would be done by hand. The outer loop is performed by a "DO UNTIL" construct starting at line 4 and extending through line 9. The program elements found on lines 5 to 8 are executed over and over again until DONE is found to be equal to logical 1 or "TRUE" when a test is made at the END statement of line 9. A "DO FOR" loop is used to sequence "N" calls to a subroutine called GENERATION which does the actual work of computing the next generation content and then displaying it on the scope. The remainder of Fig. 3 summarizes the data and subroutines referenced by LIFE.

From this point, the LIFE Line can extend in two directions. In order to have a complete LIFE program, both areas have to be traversed – the KEYBOARD_INTERPRETER and the GENERATION routine . . .but the partitioning has nicely separated the two problems. The simpler and most self-contained of the two segments is the GENERATION algorithm, so I'll turn attention to it next.

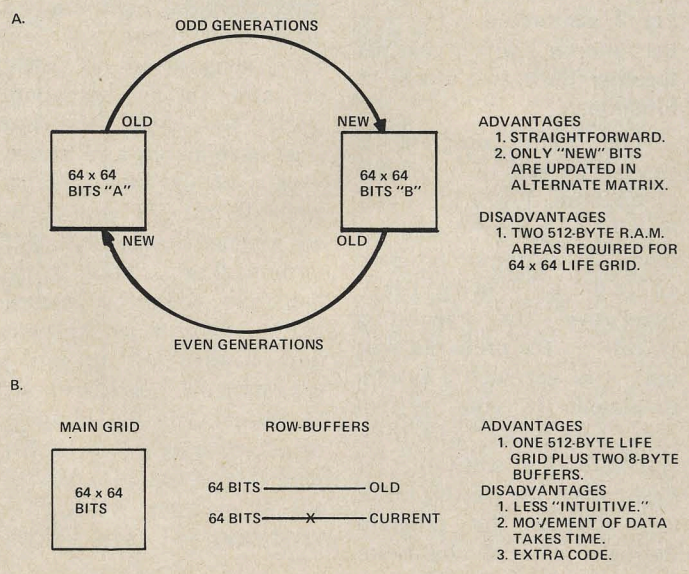
Grid Scanning Strategies

The GENERATION subroutines of the LIFE program has as its design goal the transformation of one complete LIFE grid pattern into the "next" complete pattern. The rules of the Game of LIFE – the "facts of life" – must be applied to each location in the grid to compute the next value of that location. Fig. 4 illustrates two potential strategies for computing the next generation – methods of scanning the grid to compute one location at a time.

The first strategy, Fig. 4(a), is to employ alternate copies of a complete LIFE grid of 64 by 64 points. If generations are numbered consecutively, the generation algorithm would transform copy A into a "next" copy in B on odd generations, and complete the cycle by transforming copy B into a "next" copy in the A grid on even generations. Since each grid requires 4096 bits – which can be packed into 512 bytes – a total of 1024 bytes is required for data storage if this method is used. The primary advantages of this method are its "straightforward" nature and its separation of old and new data at all times.

A second strategy is illustrated in Fig. 4(b), the strategy of using alternate row-buffers with only one

Fig. 4. The LIFE evolution algorithm – matrix scanning techniques which preserve relevant old information while creating new information in overlapping storage areas.



main grid copy. Two 64 bit rows must be maintained – the last previous row and the current row – as 8 byte copies. These copies contain information *prior to updating* in the row by row scan down the matrix. The main advantage is a saving of data areas (partially offset by more complicated software). The main disadvantages are its less "intuitive" nature, the extra time required to make data copies, and a slightly larger program.

The choice between these two methods is primarily one of the amount of storage to be devoted to data. The tradeoff is in favor of the double matrix method when very small LIFE matrix sizes are considered. The extra 8 bytes required for a second copy of an 8 by 8 grid of bits hardly compares to the programming cost of the alternate row-buffer strategy. When large matrices are considered, however, the memory requirements of an extra copy of the data are considerable, but the programming involved is no more difficult. For example, consider the limit of an 8 bit indexing method – a 256 by 256 grid. This will require a total of 8192 bytes for *each copy* of the LIFE grid. Two copies of the LIFE grid would use up 16k bytes, or one fourth of the addressing space of a typical contemporary microcomputer, and all of the addressing space of an 8008 microcomputer! At the 64 x 64 bit level, the tradeoff is much closer to the break-even point, but I expect to find at least 100 bytes saved as a result of using the row-buffer method. An assumption which is also being made when the alternate row-buffer method is used is that the scope display or TV display you use for output will have its own refresh memory so that the "old" pattern can be held during computation of the new. If this is not the case, a less desirable output in which partially updated patterns are seen will be the result. (Counting the CRT refresh, the method of Fig. 4(a) thus requires three full copies of the matrix information, and the method of Fig. 4(b) requires two full copies.)

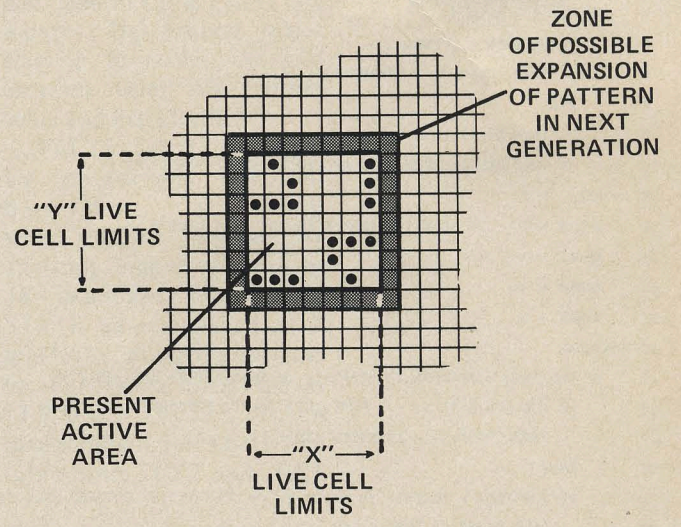
An objective: Split the processing into moderately large segments, each of which is less complicated than the program taken as a whole.

Active Area Optimization

With the choice of a matrix scanning strategy – the alternate row-buffer method – another consideration in designing the generation algorithm is a computation time optimization method. There is no real need to calculate a new value of every cell in a mostly empty LIFE grid. If I only have one glider with its corner at location (34, 27) of the grid, why should I compute any new generation information outside the area which could possibly be affected by the present pattern's evolution? Again, the savings in computation time using active area optimization depend upon the size of the grid. If most patterns occupy the full grid, then very little will be saved – for the small 8 x 8 grid "straw man" used in discussing scanning strategy, there would also be no point to active area optimization. But with a huge 256 by 256 grid, and an 8 by 8 active area, this optimization might mean the difference between a 15 minute computation and a 1 or 2 second computation of the next generation.

Fig. 5 illustrates the concept of active area optimization in a LIFE program. The current generation's active area is defined as the set of X and Y limits on the extent of live cells in the grid. In Fig. 5, the active area is the inner square of 7 x 7 = 49 grid locations. In computing the next generation, a box which is one location wider in each of the four cardinal directions is the "zone of possible expansion" for the pattern. In Fig. 5, this zone is the outer box of 9 by 9 locations. The computation of "next generation" values need only be carried out for the 81 grid locations bounded by the outer limits of the zone of possible expansion. Thus in the case of the 64 by 64 matrix of LIFE points, this optimization for the pattern of Fig. 5 will limit the program to calculation of 81 new points versus the 4096 points which would be calculated if at least one cell was found at each of the minimum (0) and maximum (63) values of the X and Y

Fig. 5. Active area optimization – never compute more than the absolute minimum if speed is at a premium.



The "facts of life" must be applied to each location in the grid to compute the next value – cell or no cell – of that location.

coordinates. This case yields a savings of 98% of the maximum generation to generation computing time.

The GENERATION Subroutine

Fig. 6 illustrates the code of the GENERATION routine, specified in a procedure-oriented language, along with notes on further subroutines and data requirements. The procedure starts by initializing the data used for the scan of the matrix, in lines 3 through 15. THIS and THAT are used to alternately reference the 0 and 1 copies of an 8 byte data item called a 2 by 8 byte data area called "TEMP". (Subscripts, like in XPL and PL/M are taken to run 0 through the dimension minus 1.) NRMIN, NRMAX, NCMIN, and NCMAX are used to keep track of the new active area limits after this generation is computed; NROWMIN, NROWMAX, NCOLMIN and NCOLMAX are originally initialized by the KEYBOARD_INTERPRETER and are updated by LIMITCHECK after each generation is calculated – using the new active area limits.

The actual scan of the grid of LIFE, stored in the data area called LIFEBITS, is

Fig. 6. The GENERATION routine specified in a procedure-oriented language . . .

```
 1 GENERATION:
 2    PROCEDURE;
 3    THIS = 0;  /* INITIALIZE POINTERS TO TEMPORARY ROW  */
 4    THAT = 1;  /*      COPY VARIABLE "TEMP"             */
 5    DO FOR I = 0 TO 7;
 6       IF NROWMIN = 0 THEN
 7          TEMP (THAT,I) = LIFEBITS(63,I);
 8       ELSE
 9          TEMP (THAT,I) = 0;
10       /* THIS ESTABLISHED WRAP-AROUND BOUNDARY CONDITION */
11    END;
12    NRMIN = 99;  /* THEN INITIALIZE ACTIVE AREA LIMITS  */
13    NRMAX = 0;
14    NCMIN = 99;
15    NCMAX = 0;
16 ROW_LOOP:
17    DO FOR IROW = NROWMIN TO NROWMAX; /* SCAN ACTIVE ROWS ONLY */
18       DO FOR I = 0 TO 7;  /* COPY THIS ROW TO TEMPORARY */
19          TEMP (THIS,I) = LIFEBITS (IROW,I);
20       END;
21       DO FOR ICOL = NCOLMIN TO NCOLMAX; /* SCAN ACTIVE COLUMNS ONLY */
22          CALL FACTS_OF_LIFE (IROW, ICOL);
23       END;
24       X = THIS;
25       THIS = THAT;
26       THAT = X; /* THIS SWITCHES BUFFERS */
27    END;
28    CALL LIMITCHECK;
29    CALL DISPLAY;
30    CLOSE GENERATION;
```

Subroutines Referenced by GENERATION:

EVOLVER . . . This is the routine used to calculate the next value of the ICOLth bit in the IROWth row of LIFEBITS using the current value of the next row, the saved value in TEMP of the previous row, and the saved value in TEMP of the current row before updating.

LIMITCHECK . . . This is the routine used to calculate the next values of NROWMIN, NROWMAX, NCOLMIN, NCOLMAX using the current values of NRMIN, NRMAX, NCMIN and NCMAX.

DISPLAY . . . This routine transfers the LIFEBITS data to the display, on whatever kind of device you have.

Data (8 bit bytes) used by GENERATION at this level:

X = TEMPORARY
I = temporary index (not the same as the I in Fig. 3)
ICOL = index for column scanning . . .
IROW = index for row scanning . . .
NCMAX = current maximum column index of live cells
NCMIN = current minimum column index of live cells
NRMAX = current maximum row index of live cells
NRMIN = current minimum row index of live cells

Data (8 bit bytes) used by GENERATION but shared with the whole program:

THIS = current line copy index into TEMP.
THAT = previous line copy index into TEMP.
TEMP = 2 by 8 array of bytes containing 2 64-bit rows.
NROWMIN = minimum row index of live cells.
NROWMAX = maximum row index of live cells.
NCOLMIN = minimum column index of live cells.
NCOLMAX = maximum column index of live cells.
LIFEBITS = 64 by 8 array of bytes containing 64 rows of 64 bits.

Assumptions:

LIFEBITS, NROWMIN, NROWMAX, NCOLMIN, NCOLMAX are initialized in *KEYBOARD_INTERPRETER* for the first time prior to entry – and retain old values across multiple executions of *GENERATION* thereafter.

Fig. 7. The LIMITCHECK routine specified in a procedure-oriented language . . .

```
1  LIMITCHECK:
2     PROCEDURE;
3     /* CALCULATE NEXT ROW LIMITS */
4     IF NRMIN-1 < NROWMIN THEN NROWMIN = NRMIN-1;
5     IF NRMAX+1 > NROWMAX THEN NROWMAX = NRMAX+1;
6     IF NROWMAX > 63 THEN NROWMAX = 63;
7     IF NROWMIN < 0 THEN NROWMIN = 0;
8     /* CALCULATE NEXT COLUMN LIMITS */
9     IF NCMIN-1 < NCOLMIN THEN NCOLMIN = NCMIN-1;
10    IF NCMAX+1 > NCOLMAX THEN NCOLMAX = NCMAX+1;
11    IF NCOLMAX > 63 THEN NCOLMAX = 63;
12    IF NCOLMIN < 0 THEN NCOLMIN = 0;
13    CLOSE LIMITCHECK:
```

Subroutines Referenced by LIMITCHECK:

None.

Data (8 bit bytes) used by LIMITCHECK but shared with the whole program:

NCOLMAX, NCOLMIN, NROWMAX, NROWMIN
NRMAX, NRMIN, NCMAX, NCMIN *(see Fig. 6)*

Assumptions:

The arithmetic of the comparisons in this routine is done using signed two's complement arithmetic – thus a negative number results if 0 - 1 is calculated . . . this is consistent with code generation on most 8 bit micros.

performed by the set of DO groups beginning with ROW_LOOP at line 16. For each row of the matrix, ROW_LOOP first copies the row into TEMP as the THIS copy (the THAT copy is left over from initialization the first time at lines 5 to 11, or from the previous ROW_LOOP iteration thereafter). Following the copying operation, another DO FOR loop goes from NCOLMIN to NCOLMAX applying the FACTS_OF_LIFE to each grid position in the current (THIS) row as saved in TEMP. New data is stored back into LIFEBITS by FACTS_OF_LIFE. At the end of the row loop, prior to reiteration, the THIS and THAT copies of temp are switched by changing the indices. What was THIS row becomes THAT row with respect to the next row to be computed.

After all the rows have been computed, line 28 is reached. Line 28 calls subroutine LIMITCHECK to compute the next generation's active area computation limits using the results of this generation. Line 29 then calls a module named DISPLAY to copy the results of GENERATION into the output display device. The LIMITCHECK routine simply performs comparisons and updating – Fig. 7 illustrates the high level language description of its logic.

Computing The Facts of LIFE . . .

Fig. 8 contains the information on implementing the Facts of LIFE in a programmed set of instructions. The computation is divided into two major parts. The first part is to determine the STATE of the bit being updated, where "STATE" is a number from 0 to 8 as described in LIFE Line 1 last month. The second major step is to evolve the grid location using its current value and the STATE.

FACTS_OF_LIFE begins by performing left and bottom boundary wrap-around checks by adjusting indices. Lines 8 to 18 determine the current STATE by referencing all 8 grid locations surrounding the location being computed at (IROW, ICOL). In determining the state, the subroutines TGET and LGET

Two copies of a 256 by 256 grid would require more memory than (for example) an 8008 can address if you want to have programs along with your data.

Why should I compute any new generation information outside the area which could possibly be affected by the present pattern's evolution?

Fig. 8. The FACTS_OF_LIFE routine specified in a procedure-oriented language. FACTS_OF_LIFE does the actual calculation of the next value for the LIFEBITS location at the IROWth row and ICOLth column based upon the previous value of the 8 neighboring locations. (The state defined in LIFE Line 1, last month.) This routine implements the rules described in BYTE #1, page 73.

```
 1  FACTS_OF_LIFE:
 2     PROCEDURE (IROW,ICOL);
 3     M = IROW + 1;
 4     IF M > 63 THEN M = 0; /* BOTTOM BOUNDARY WRAP CONDITION */
 5     N = ICOL - 1;
 6     IF N < 0 THEN N = 63; /* LEFT BOUNDARY WRAP CONDITION */
 7  DETERMINE_STATE:
 8     STATE = TGET (THAT,N);
 9     STATE = STATE + TGET (THIS,N);
10     STATE = STATE + LGET (M,N);
11     N = ICOL;
12     STATE = STATE + TGET (THAT,N);
13     STATE = STATE + LGET (M,N);
14     N = ICOL + 1;
15     IF N > 63 THEN N = 0; /* RIGHT BOUNDARY WRAP CONDITION */
16     STATE = STATE + TGET (THAT,N);
17     STATE = STATE + TGET (THIS,N);
18     STATE = STATE + LGET (M,N);
19  EVOLVEIT:
20     NEWCELL = 0; /* DEFAULT EMPTY LOCATION UNLESS OTHERWISE */
21     OLDCELL = TGET (THIS, ICOL);
22     IF OLDCELL = 1 THEN DO;
23        IF STATE = 2 OR STATE = 3 THEN NEWCELL = 1;
24     END;
25     ELSE DO;
26        IF STATE = 3 THEN NEWCELL = 1;
27     END;
28     CALL LPUT (IROW, ICOL, NEWCELL);
29     IF NEWCELL = 1 THEN CALL SETLIMIT (IROW, ICOL);
30     CLOSE   FACTS_OF_LIFE;
```

Subroutines Referenced by FACTS_OF_LIFE:

TGET . . . This is a "function" subroutine which returns an 8 bit value (for example in an accumulator when you generate code) of 00000001 or 00000000 depending upon whether or not a referenced column in one of the two temporary line copies in TEMP is 1 or 0 respectively. The first argument tells which line of the two, and the second argument tells which column (0 to 63) is to be retrieved.

LGET . . . This is a "function" subroutine which returns an 8 bit value similar to TGET, but taken instead from the bit value at a specified row/column location of LIFEBITS.

LPUT . . . This subroutine is used to set a new value into the specified row/column location of LIFEBITS.

NOTE: The routines LGET and LPUT will be referenced from the KEYBOARD_INTERPRETER *routine in the course of manipulating data when setting up a life pattern.*

SETLIMIT . . . This subroutine is used to check the current active region limits when the result of the facts of life indicate a live cell.

Data (8 bit bytes) used by FACTS_OF_LIFE at this level:

IROW = Parameter passed from *GENERATION.*
ICOL = Parameter passed from *GENERATION.*
M = temporary, row index.
N = temporary, column index.
STATE = count of "on" bits in neighborhood of *IROW, ICOL.*
OLDCELL = temporary copy of old cell at *IROW, ICOL.*
NEWCELL = new value for location *IROW, ICOL.*

Data (8 bit bytes) used by FACTS_OF_LIFE but shared with the whole program:

THAT, THIS (see Fig. 6)

What was THIS row becomes THAT row with respect to the next row to be computed. (What's in a name? A pointer of course!)

are used to reference bits in TEMP and LIFEBITS respectively, using appropriate bit location indices. The values returned by these two "function subroutines" are either 0 or 1 in all cases — thus counting the number of "on" cells consists of adding up all the TGET or LGET references required to examine neighboring grid locations.

Once the STATE of the grid location is determined, the Facts of LIFE are implemented by examining the *positive* cases of an "on" (live cell) value for the grid location. A cell will be in the grid location for the next generation in only two cases: If the old content of the location was a live cell and the STATE is 2 or 3; or if the old content of the location is 0 (no cell) and the STATE is 3. A default of NEWCELL = 0 covers all the other cases if these two do not hold. Line 28 stashes the new value away in LIFEBITS with subroutine LPUT, and if the new value of the grid location

Fig. 9. The SETLIMIT routine specified in a procedure-oriented language.

```
1 SETLIMIT:
2    PROCEDURE (IROW,ICOL);
3    IF IROW < NRMIN THEN NRMIN = IROW;
4    IF IROW > NRMAX THEN NRMAX = IROW;
5    IF ICOL < NCMIN THEN NCMIN = ICOL;
6    IF ICOL > NCMAX THEN NCMAX = ICOL;
7    CLOSE SETLIMIT;
```

Subroutines Referenced by SETLIMIT:
None.

Data (8 bit bytes) used by SETLIMIT at this level:

IROW = parameter passed from *FACTS_OF_LIFE.*
ICOL = parameter passed from *FACTS_OF_LIFE.*

Data (8 bit bytes) used by SETLIMIT but shared with the whole program:

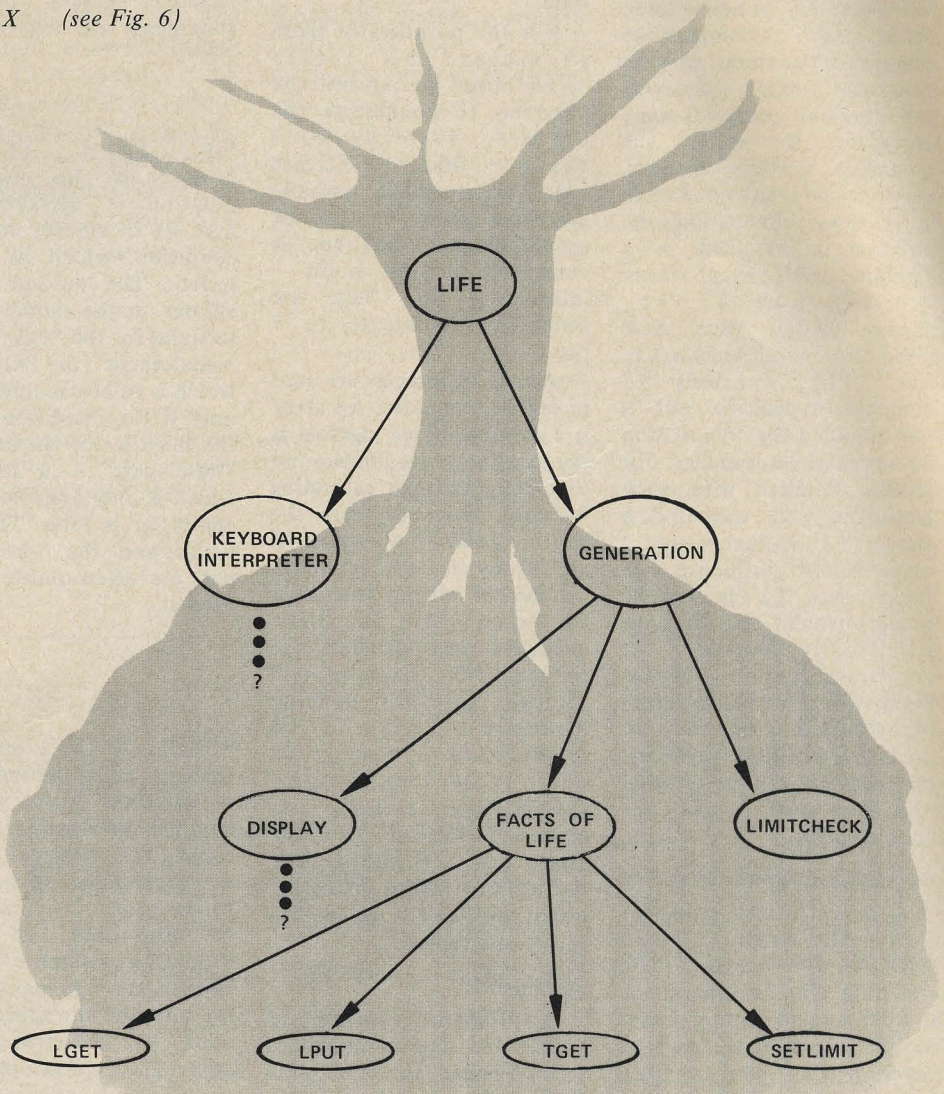
NRMIN, NRMAX, NCMIN, NCMAX *(see Fig. 6)*

is a live cell, SETLIMIT is called (see Fig. 9) in order to update the active area pointers NRMAX, NRMIN, NCMAX and NCMIN.

Where Does the LIFE Application Stand?

An alternative to the pyramid structure way of viewing programming program designs introduced at the beginning of this article is a "tree" notation showing the heirarchy of modules in the application. The "Tree of LIFE" is shown in Fig. 10 as it exists in materials printed to date. The next installment of LIFE Line will explore the left hand branch of the tree diagram by a similar presentation of a KEYBOARD_INTERPRETER algorithm.

Fig. 10. The Tree of LIFE.



LIFE Line 2 Addendum

Procedure-Oriented Computer Languages

The examples of programs accompanying two articles in this issue have been constructed in a procedure-oriented language. This method of program representation is compact and complete. In principle, one could write a compiler to automatically translate the programs written this way into machine codes for some computer. By writing the programs in this manner, more detail is provided than in a flow chart, and the program is retained in a machine independent form.

The particular representation used here resembles several languages in the "PL/1" family of computer languages, but is not intended for compilation by any existing compiler. For readers familiar with such languages, you will find a strong PL/1 influence and a moderate XPL influence. In a future issue BYTE will be running articles on a language specifically designed for microcomputer systems, PL/M, which is an adaptation of the XPL language for 8-bit machines. For the time being, this representation is used with some notes to aid your understanding.

Programs and Procedures

A *program* is a group of lines which extends from a PROGRAM statement to a matching CLOSE statement. It is intended as the "main routine" of an application. A *procedure* is a similar group of lines which extends from a PROCEDURE statement to its CLOSE statement. A procedure may have *parameters* indicated in the PROCEDURE statement, and may be *called* as a "subroutine" from a program or another procedure. A procedure may be called in a "function" sense as well, in which case a RETURN statement would be required to set a value.

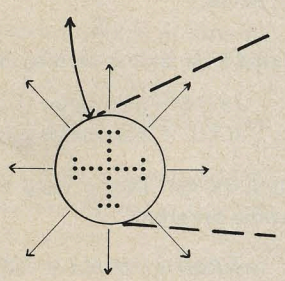
Data

For the purposes of these examples, no "data declarations" are put into the programs to complicate the picture. Instead, each example has a section following it which verbally describes each data name used. Only one "data type" is considered at this point – integers – and these are generally assumed to be 8 bits.

Arrays of integers are used in several examples. An array is a group of bytes, starting at the location of its *address* and extending through ascending memory addresses from the starting point. The purpose of an array is to reference "elements" within the array by "subscripts". For these examples, the elements are referenced by the numbers 0 through "n-1" for an array dimension of length "n". If LIFEBITS is an array of 64 by 8 bytes, then LIFEBITS(63,7) is the last element of the last row of the array, and LIFEBITS(I,J) is the byte at row I, column J provided I and J are within the proper ranges.

Statements

A program or procedure consists of statements which specify what the computer should do. In a machine language, these would correspond to the selected operation codes of the computer which is being programmed. For a high level language, one statement typically represents several machine instructions. In these the high level language statement has a "semantic intent" – a definition of its operation – which can be translated into the lower level machine language. In these examples several types of statements are employed ...



"IF ... THEN ... ELSE ..." constructs are used for notation of decisions. The first set of ellipses indicate a condition which is to be tested. The second set of ellipses in the model is used to stand for the "true part" – a statement (or DO group) which is to be executed if and only if the condition is true; the third set of ellipses is the "false part" – a statement which is only executed if the condition is false. The word ELSE and the whole false part are often omitted if not needed.

"CALL X" is a statement used to call a subroutine, in its simplest form. A more complicated form is to say CALL X(Z) where Z is a set of "arguments" to be passed to the routine. Another form of subroutine call is the "function reference" in an assignment statement, where the name of the subroutine is used as a term in an arithmetic expression.

"assignment" – a statement of the form "X = Y;" is called the *assignment statement.* Y is "evaluated" and the result is moved into X when the statement is executed. If X or Y have subscripts as in "TEMP(THIS,I) = LIFEBITS(IROW,I);" then the subscripts (such as "THIS,I" and "IROW,I" in the example) are used to reference the name as an array and pick particular bytes.

"DO groups" – a grouping of several statements beginning with a "DO" statement and running through a corresponding "END" is used to collect statements for a logical purpose. In "DO FOR I = 0 to 7;" this purpose is to execute the next few statements through the corresponding "END;" 8 times with I ranging from 0 to 7. "DO UNTIL DONE=TRUE;" is an example of a group which is repeated indefinitely until a condition is met at the END. "DO FOREVER" is a handy way of noting a group to be repeated over and over with no end test, a practice often frowned upon.