LITTLE Newsletter # 32A

Claudette McDonald
January 9, 1975

Realisation of the LITTLE Interrupt System Described in Newsletters #30 and 32

1. Introduction.

This newsletter, essentially identical to the author's master thesis, describes a realisation of the LITTLE interrupt system specified in LITTLE Newsletter # 30. (A few restrictions, of relatively minor importance, are imposed on the general scheme described in Newsletter 30). The text of this newsletter omits three appendices of the full thesis: these appendices are respectively: the corrections to the LITTLE grammar and intermediate code-generator routine package needed to define the interrupt-handling extensions; corresponding corrections to the machine-code generating routines; and a sample program written in the extended language.

2.0 Precis

The interrupt facilities center around the notion of a 'process'. Processes consist of program text and a data environment. They can communicate with each other through 'public' variables (there is no notion of parameter passing between processes). Once a process is initialized, control may be passed to it either by execution of a RECALL statement or automatically in response to an externally generated interrupt.

The semantic primitives implemented during this project appear in Table 1. These include declarations, executable statements, compiler-directed statements, a builtin function, several builtin global variables, and a name qualification for 'private' global variables. Optional clauses are enclosed in brackets.

The clause form

processname(array,index)

which appears in many of the primitives described below implies that *processname* is a process, *array* is a previously defined array, and that *index* is an integer valued expression.

TABLE 1. Interrupt Handling Primitives.

Primitive	Function
PROCESS processname DISABLED/ENABLED; (declarations and code for main routine) [(code for all subprograms in process)] END [processname];	Definition of a process with its variables, main routine, subprograms, and its initial enable status.
INITIAL SETUP processname (array, index) [var_1 = $datalist_1$: var_2 = $datalist_2$: ... var_n = $datalist_n$];	Compile time process initialization. Space is reserved in the designated array beginning at location index for the process' data environment (status package). Process' private global variables are initialized.
INITIAL SETUP processname(array,index) FROM label [var_1 = $datalist_1$: var_2 = $datalist_2$: ... var_n = $datalist_n$];	Compile time process initialization for the process entered first. Space is reserved in the designated array, beginning at location index,

	for the process data environment (status package). Process' private global variables are initialized. After processes are compiled and loaded, control is passed to the named process, with a forced transfer to the specified label, a label in the process' main routine.
SETUP processname (array,index);	Runtime process initialization. Not completely implemented since reentrant routines are not supported in LITTLE.
RECALL processname (array,index);	Resumption of a process, at location recorded in the process data environment. Register reload performed.
RECALL processname (array,index) FROM label;	Passes control to a process, with a forced transfer to a label in the main routine of the process. Register reload is omitted.
ATTACH n TO processname(array,index);	Attaches interrupt source to a process. When interrupt n occurs, the process is resumed beginning at the location recorded in the process data environment. Register reload is performed.

ATTACH n TO processname(array,index) FROM label;	Attaches interrupt source to a process. When interrupt n occurs, control passes to the process with a forced transfer to a label in the process' main routine. Register reload is omitted.
ENABLE;	Enables all interrupts.
DISABLE;	Disables all interrupts.
PUBLIC NAMESET name; SIZE var_1(siz_1); SIZE var_2(siz_2); ⋮ SIZE var_n(siz_n); END [PUBLIC NAMESET];	Declaration of public global variables, which may be accessed by any process.
quantityname[(extraindex)] .PN. processname (array,index)	A name qualification to allow one process to reference private global variables in another process. The named quantity is declared as a private global variable in the named process. Array and index locate the place where the process data environment is stored. 'extra index' is used to reference a component of a dimensioned variable.

PSIZE(processname)	Builtin function that returns the number of words of storage needed for the process' data environment.
INTMASK	Builtin private global variable used to repress and allow specific external interrupt sources in a process. Assignment to variable causes hardware interrupt mask register reload.
WAS→ENABLED	Builtin public global variable, indicates previous enable status when a DISABLE statement is executed.
SIMULATE IN processname;	Compiler-directed statement prefixed to a family of processes modifies compilation of LITTLE source text to include code to simulate interrupts. Interrupts are simulated at the points of subroutine calls and user defined labels. The specified process is a user-provided process which assigns an interrupt type, mimics core locations or register changes associated with

| an interrupt, and recalls whatever process is attached to the interrupt type being simulated. E.g. CALL SUBK;

```
/* FROM LITTLE SOURCE TEXT */
is compiled as:
EXEC→COUNT = EXEC→COUNT -
 (compile-time estimate of number
  of instructions since last
  simulated interrupt);
IF EXEC→COUNT < 0 THEN
   CALL processname(INTMASK of
          current process);
   END IF;
CALL SUBK;
```

`INTERRUPT n;` Used in simulation process to recall whatever process is attached to interrupt n.

`EXEC→COUNT` Built-in public global variable used to control frequency | of simulated interrupts.

3.0 Description of Facilities

The description includes references to semantic primitives found in Table 1 and to LITTLE statements described in the LITTLE language user's manual, LITTLE Newsletter Number 33.

3.1 Processes

Our language extension involves the addition of 'processes' to LITTLE. Abstractly a process may be thought of as a complete program, text plus data environment ready to run. It consists of a 'main routine' plus any subroutines or functions which the process calls. It also includes declarations of all variables and constants (i.e. the complete data environment) needed for this program to run.

The process is defined as follows:

```
PROCESS processname ENABLED;  /* PROCESS HEADER LINE */   or
PROCESS processname DISABLED; /* ALTERNATE PROCESS
                                 HEADER LINE */
    [declarations for main routine]
    [global declarations for process]
    code for main routine
    [code for all necessary subprocedures]
END[processname];
```

The main routine must include at least one label. The labels designate entry points for the process. These

are points at which the process will start to perform some function. The first SUBR or FNCT statement in a process or an END statement which matches the PROCESS statement terminates the main routine. Any other LITTLE statement or declaration including those in the interrupt facilities may occur in the main routine.

Any well-defined subroutine or function as currently defined in the LITTLE user's manual can be used as part of a process. No code or declarations may occur between subprocedures. Processes need not have subprocedures. A total program, compiled in a single run, consists of one or more processes. No code, declarations, or subprograms may occur between processes.

3.2 Namescoping and Processes

An extension was made to the existing LITTLE namescoping scheme. In the existing scheme, local and global variables are defined. Global variables are global to a compilation. Our extended scheme is described as follows. Variables not declared within the scope of a NAMESET statement are local to the subprocedure or the main routine in which they are defined. Variables declared within the scope of a NAMESET (between the NAMESET declaration and the next END statement) are private global variables, available to all the routines of the process. Any subprocedure in the process may refer to these variables by using an

ACCESS *namesetname*;

statement, Routines in one process may not access namesets which are private to another process. However a new declaration

PUBLIC NAMESET *namesetname*;

is provided. Variables declared within the scope of such a nameset may be accessed by any routine in any process by providing an ACCESS statement in the routine.

To ease the use of global variables, a default first routine nameset and default access options are currently provided as control card options. The monoprocess convention is modifed for multiple processes as follows:

-- first routine nameset (GS parameter); if this option is used, all variables declared in the main routine of a process which would normally be local to the main routine are gathered into a private nameset given the name of the process

-- default access (DA parameter); if this option is used, each subprocedure in the process is granted access to all namesets defined or accessed in the main routine of a process. The effect is as though the compiler inserted an access statement, with names of namesets defined in the main routine (including the first routine nameset) and the names of all public namesets accessed in the main routine, after each SUBR or FNCT statement in the process.

Variables declared in ordinary namesets are known as *private global variables* and those in public namesets are known as *public global variables*. Processes may access or modify private global variables of another process or an individual basis by using the following syntactic forms:

(a) *quantityname* .PN. *processname(array,index)*

(b) *quantityname(extraindex)* .PN. *processname(array,index)*

The *quantityname* occurring in (a) or (b) must be defined within the text of the named process as a private global variable. Form (a) is used for unindexed variables and form (b) is used for indexed variables. In the second form, *extraindex* is the one defining the particular component of a private array in process *processname*. The *array* and *index* parameters serve to locate the process' data environment and are described in the next section.

3.3 Process Data Environment and Status Package

Every process has a data environment consisting of all declared private global or local variables and arrays, routine return addresses, and routine parameters. In addition, the compiler allocates space for a status package area in which registers and other machine dependent status words needed to implement the interrupt facilities are saved. The status package always contains space for an instruction location, an enable status bit, and an interrupt mask; aside from this, the exact size of the status package is implementation dependent. The exact amount of space needed to store the status package is made available to the user through a pseudo-function of the form

(2) PSIZE(processname)

Note that the value of this function is now provided as a constant equal to the number of words needed to hold the status package.

The original design of the interrupt facilities, as given in LITTLE Newsletter Number 30, included features needing the support of reentrant routines, not currently provided in the LITTLE compiler. If reentrancy is provided in a future LITTLE compiler, the PSIZE pseudo-function should return a value indicating the storage space needed for the 'complete' data environment of a process.

One element of the status package, the process' interrupt mask, is available to the user through a private global variable named INTMASK. This variable has an implementation dependent size equal to the number of interrupt sources recognized. The j-th external interrupt source corresponds to the j-th bit in INTMASK. If the bit is 1, the interrupt source is enabled. The use of this mask is explained in Section 3.5.

Each process also has an enable status. The mask and the status are saved whenever an explicit RECALL statement is executed (this must be done by an assembly language interrupt handler invoked whenever control exits from a process in response to an external interrupt). The interrupt mask is in the second location in the process' data environment, and the third location is for the process' enable status which the interrupt handler sets to the value '1'. The compiler provides a pointer named CDP which points to the first address of the data environment for the process currently running.

3.4 Statements

Our language extension includes statements to perform interrupt-management and multi-processing related actions. The INITIAL SETUP statement, which has one of two forms,

(3) INITIAL SETUP *processname(array,index)*
[var$_1$ = datalist$_1$: var$_2$ = datalist$_2$: ...
var$_n$ = datalist$_n$];

(4) INITIAL SETUP *processname(array,index)*
FROM *label*
[var$_1$ = datalist$_1$: var$_2$ = datalist$_2$: ...
var$_n$ = datalist$_n$];

directs the compiler to reserve storage for a process' 'status package' in a designated array starting at a specified index. The array must be previously defined and of some fixed, implementation dependent size, normally the size of the longest register or status word. The specified *index* must be a constant, and *processname* a process. The statement may include data initialization clauses for private global variables in the process named *processname*.

Such load-time assignments, made within the scope of an INITIAL SETUP statement, perform the same function as a DATA statement occurring within the process. An INITIAL SETUP statement must appear after the PROCESS header line of the process to which it refers, and after the declaration

of any global variables it names, with the exception of INTMASK.

The statement form (4) is used to set up the first process to be entered at execution time. The statement specifies a label in the main routine of the process named *processname*. The compiler, after encountering a statement of form (4), generates a routine named START containing instructions to recall the process specified in statement (4) from the designated label. Execution, in the CDC 6600 implementation for LITTLE, begins with a routine named START. Of course, the statement form (4) may be used for only one process in a single compilation run. Every other process in a single compilation run is to be set up using form (3). The statement form,

(5) SETUP *processname(array,index);*

specified in LITTLE Newsletter 30, is not implemented since it requires reentrancy not currently supported by LITTLE.

Control can be passed to a process by an explicit RECALL statement. Two RECALL statement forms are provided.

(6) RECALL *processname(array,index);*

(7) RECALL *processname(array,index)* FROM *label;*

When such a statement is executed, the currently executing process' status package is saved in the exiting process' data environment area and a transfer is made to the named process. Specifically the program location counter is saved

in the first location of the status package area, the interrupt mask register is saved in the second location, and the process' enable status is saved in the third location. Any other implementation dependent registers or status words are saved in following consecutive locations. The simple RECALL statement (6) effects a jump to the location saved in the data environment of the named process. The use of the simple RECALL assumes that the named process has been entered at some previous time and execution was suspended either by the execution of a RECALL statement within the text of the named process or by some earlier external interrupt. Within the text of the new process is the code needed to restore its process-state.

The alternate RECALL form (7) forces a transfer to a particular label in the main routine of the specified process. Register reload is omitted in this type of process switching with the exception of the interrupt mask register.

When a RECALL statement is the last statement of a routine in some process and such a statement is executed, the process should subsequently be restarted by a recall of the form (7). If the process is restarted by a simple recall, an error condition occurs.

External interrupts are treated essentially as autonomously forced RECALL operations of one of two forms corresponding to the two types of explicit recalls. Accordingly the ATTACH statement is provided to connect a particular process-state to the specified interrupt source.

(8) ATTACH *n* TO *processname(array,index)*;

(9) ATTACH *n* TO *processname(array,index)* FROM *label*;

In treating these statements, the compiler assumes some non-empty implementation defined set of interrupt sources to exist; these interrupt sources are numbered 1,2,3,...j. In the ATTACH statement, *n* is an integer-valued expression representing the source of the interrupt. When an ATTACH statement is executed, information needed at interrupt time to handle an interrupt is stored in two builtin public global arrays: the nth location in the array ATTACHMENTS stores the process entry point and the nth location in the array ATTINDICES stores the first address of the process data environment. The execution of an ATTACH statement automatically detaches an interrupt source from any other process to which it may have been attached.

When an interrupt occurs, the system behaves as if one of the two forms of explicit recalls has been executed. An assembly-language interrupt handler is assumed to 'trap' all interrupts in the system; using the builtin public global pointer CDP, this routine saves the process-state of the interrupted process in the area beginning at the location pointed to by CDP. The interrupt handler always saves the program location counter at the time of interruption in the first location in this data area, the interrupt mask value at the time of interruption in the second location of this area, and always sets the third location to the value '1'.

It changes CDP to point to the data environment of the process attached to the specific interrupt source before transferring to the entry of that process. On the occurrence of interrupt j, the addresses necessary to start the appropriate interrupt handling process are found in the jth locations of the ATTINDICES and ATTACHMENTS arrays. The body of the new process contains the code to restore its process-state if such code is necessary. The interrupt handler need not know whether a 'simple' or a 'FROM label' attach was made.

The variable names CDP, ATTACHMENTS, and ATTINDICES are defined within a nameset named PUBLOCK and are available to an assembly-language program by using a COMMON block of the same name.

The ENABLE statement is provided to enable all interrupts and the DISABLE statement is provided to disable all interrupts. The DISABLE operation does not change the value of the process interrupt mask. These two statements respectively set and clear a builtin 1-bit public global flag called ENABLE representing the enable state of the system. This flag may be accessed but not modified. Its use in interrupt simulation is described in Section 3.6. When a DISABLE statement is executed, another 1-bit public global flag called WAS→ENABLED, available to the user, is set to '1' if interrupts were previously enabled and to '0' otherwise.

Since the LITTLE compiler does not at present support reentrant routines, the user must use caution in recalling a given process to perform different functions 'concurrently'. If necessary, multiple copies of such a process should be provided, each with a different process name.

A user's reference to the interrupt handling statements is provided in Appendix A.

3.5 Enable Status and Interrupt Mask

During the execution of a recall statement and during the execution of an interrupt handler routine, all interrupts are disabled. When a new process is entered as a result of an explicit recall or an implicit (interrupt actuated) recall, the system enable status either remains disabled or changes to enabled depending upon the following rules. When a process is restarted by a simple recall or by action taken as a result of an external interrupt and a 'simple' attach, the process previous enable status is restored. That is, if the process' previous enable status was enabled, then all interrupts are enabled after reloading registers and jumping to the location stored in the process data environment, otherwise interrupts are left disabled.

When a process is restarted from a label, interrupts are left disabled if the process header statement is:

(10) PROCESS *processname* DISABLED;

Otherwise, if the process header statement is

(11) PROCESS *processname* ENABLED;

then all interrupts are enabled at every label in the main routine of the process. To selectively enable interrupts at some of the process entry points (i.e. labels in the main routine), a process header of form (10) is used and the user is responsible for providing an ENABLE statement at the appropriate entry points. The example in Section 3.7

demonstrates these cases.

The interrupt mask register is reloaded whenever a process is restarted. It is reloaded from the INTMASK value found in the process data environment. If the process has never been entered before, its interrupt mask value is taken from an initializing statement, i.e., a DATA statement or an INITIAL SETUP statement. Thus the interrupt mask register reloading is associated with process switching. Of course, an explicit assignment made to the variable INTMASK also causes reload of the interrupt mask register.

3.6 Interrupt Simulation and Simulation Processes

When the SIMULATE statement (12)

```
(12)          SIMULATE IN processname;
```

is prefixed to a family of processes, it causes the compilation of simulated interrupts within the source code. Interrupts can be simulated at subroutine calls and at user-defined labels, both of which end basic blocks. At these points the following interpolation is made into the generated code:

```
(13) EXEC→COUNT = EXEC→COUNT - (compile time estimate of number
                 of instructions since last preceding simulated
                 interrupt) /* EXEC→COUNT is a public global counter*/
     IF EXEC→COUNT < 0 THEN
```

```
save process-state /* a new VOA operation */
CALL simulation-process (INTMASK);
/* interrupt mask of current process */
END IF;
```

The simulation process shown above is named on the SIMULATE statement. This routine restores EXEC→COUNT to some reasonable positive value, assigns an interrupt type to be simulated, may call subprocedures to mimic any changes in core locations or accessible registers which may be associated with the interrupt, and then recalls whatever process has been attached to the particular interrupt. Of course, the simulation process source code is exempt from modification. That is, no simulated interrupts are compiled within the simulation process.

The simulation process uses the public global ENABLE flag and the variable name INTMASK, which gives the value of the interrupt mask in the interrupted process, to assist in determining if a particular simulated interrupt source is enabled. The process-state of the interrupted process is saved by the compiler before transferring to the simulation process. To assist in simulating the rest of the recall, an additional statement

(14) INTERRUPT *n*;

is provided. When executed, this statement imitates the

action of an interrupt handler, as described in Section 3.4, by recalling whatever process is attached to interrupt source *n*. In statement (14), *n* may be an integer-valued expression.

The simulation process is compiled differently than other processes. It does not have a 'status package' associated with it. The simulation process name, consequently, should not appear on an ATTACH, RECALL, or any SETUP statement form. The text of such a process, however, is defined as shown in (1), in the same manner as other processes. A RETURN statement is allowed in the simulation process' main routine to permit a return to the interrupted process when a particular simulated interrupt source is not enabled within the interrupted process.

3.7 Example of Interrupt Handling Facilities

In this section is an example illustrating the use of the interrupt handling facilities. Appendix E gives an example of a program intended to be compiled with simulated interrupts.

```
PROCESS T DISABLED;
+* ACTIVE = 1 **  +* INACTIVE = 2 **
/* DECLARE ARRAY USED FOR DATA ENVIRONMENT */
PUBLIC NAMESET PUB;
SIZE STPACK(WS); DIMS STPACK(5 * PSIZE(T));
     END PUBLIC;
PUBLIC NAMESET TPUB; /* INTERRUPT PROCESS COMMUNICATION */
SIZE GLOBA(1);  DATA GLOBA = 0;
SIZE GLOBC(1);  DATA GLOBC = 0;
     END PUBLIC;
NAMESET OWNT;  /* EXECUTIVE-PROCESS */
SIZE STATUS(2);              /* COMMUNICATION */
DATA STATUS = INACTIVE;
     END NAMESET;
SIZE W(WS);  DATA W = 0;    /* INTERRUPT 4 COUNTER */
SIZE Y(WS);  DATA Y = 0;    /* INTERRUPT 2 COUNTER */
INITIAL SETUP T(STPACK, 1) INTMASK = 1111L;
/* PROCESS IS ENTERED AT  'TLAB1' DISABLED  */
/TLAB1/   STATUS = ACTIVE; ENABLE;
     /* INTERRUPT SOURCES  1,2,3,4 ARE ENABLED  */
     /* SUSPEND PROCESS-TRANSFER TO EXECUTIVE */
     RECALL EXEC(STPACK, 2 * PSIZE(T)+1) FROM EXCONT;
     INTMASK = 0101L;       /* INTERRUPT SOURCES 1,3 ENABLED*/

     IF GLOBA CALL BUMPT(W);
     IF GLOBC CALL BUMPT(Y);
     INTMASK = 1111L; /* REENABLE 4 INTERRUPT SOURCES */
```

```
        STATUS = INACTIVE;
        /* TRANSFER TO EXECUTIVE-TERMINATE PROCESS */
        RECALL EXEC(STPACK, 2 *PSIZE(T)+1) FROM EXCONT;
        SUBR BUMPT(VAR);
        SIZE VAR(WS);           /* COUNTS INTERRUPTS */
        IF GLOBA THEN GLOBA = 0;  ELSE GLOBC = 0;
        VAR = VAR + 1;
        RETURN;
        END SUBR;  END PROCESS T;

        PROCESS X DISABLED;
             SIZE X(WS); DATA=0; /* INTERRUPT 3 COUNTER */
             SIZE Z(WS); DATA=0; /* INTERRUPT 1 COUNTER */
             SIZE SAVE(1); DATA SAVE=0;  /* ENABLE STATUS */
        ACCESS PUB;             /* DATA ENVIRONMENT */
        PUBLIC NAMESET XPUB;  /* INTERRUPT PROCESS COMMUNICATION*/
        SIZE GLOBB(1);  DATA GLOBB = 0;
        SIZE GLOBD(1);  DATA GLOBD = 0;
             END PUBLIC;
        NAMESET OWNX;           /* EXECUTIVE-PROCESS */
        SIZE STATUS(2);         /* COMMUNICATION */
        DATA STATUS = INACTIVE;
             END NAMESET;
        INITIAL SETUP(STPACK, PSIZE(T)+1) INTMASK = 1111L;
    /XLAB1/  STATUS = ACTIVE; ENABLE;
        /* INTERRUPT SOURCES 1,2,3,4 ARE ENABLED */
        /* SUSPEND PROCESS-TRANSFER TO EXECUTIVE */
```

```
RECALL EXEC(STPACK, 2*PSIZE(X)+1) FROM EXCONT;
/* DISABLE INTERRUPTS - SAVE ENABLE STATUS */
DISABLE;  SAVE = WAS→ENABLED;
IF GLOBB CALL BUMPX(X);
IF GLOBD CALL BUMPX(Z);
IF (SAVE) ENABLE;      /* RE-ENABLE IF NECESSARY */
STATUS = INACTIVE;
/* TRANSFER TO EXECUTIVE-TERMINATE PROCESS */
RECALL EXEC(STPACK, 2*PSIZE(X)+1) FROM EXCONT;
SUBR BUMPX(VAR);
SIZE VAR(WS);          /* COUNTS INTERRUPTS */
IF GLOBB THEN GLOBB = 0; ELSE GLOBD = 0;
VAR = VAR+1;
RETURN;
END SUBR;  END PROCESS X;

PROCESS EXEC DISABLED;
/* EXECUTVE PROGRAM */
ACCESS PUB; /* DATA ENVIRONMENT AND PROCESS-EXECUTIVE
                                        COMMUNICATION  */
PUBLIC NAMESET EXPUB; /* INTERRUPT PROCESS-EXECUTIVE
                                        COMMUNICATION  */
SIZE SWITCH(1);  DATA SWITCH = 1;
END PUBLIC;
/* FIRST PROCESS TO EXECUTE */
INITIAL SETUP EXEC(STPACK, 2*PSIZE(EXEC)+1)
```

```
          FROM EXECSTART;
     DATA INTMASK = 1111L;
/EXECSTART/  /* ALL INTERRUPTS DISABLED */
     ATTACH 1 TO V(STPACK,3*PSIZE(EXEC)+1) FROM VLAB1;
     ATTACH 2 TO K(STPACK,4*PSIZE(EXEC)+1) FROM KLAB1;
     ATTACH 3 TO V(STPACK,3*PSIZE(EXEC)+1) FROM VLAB2;
     ATTACH 4 TO K(STPACK,4*PSIZE(EXEC)+1);
/EXCONT/  ENABLE;  /* INTERRUPT SOURCES 1,2,3,4 ENABLED */
     IF SWITCH THEN
          PUBPARAM = -PUBPARAM+1;
     IF STATUS .PN. T(STPACK, 1) = INACTIVE THEN
            RECALL T(STPACK,1) FROM TLAB1;
            ELSE RECALL T(STPACK,1);  END IF;
          ELSE PUBPARAM = -PUBPARAM+1;
            IF STATUS .PN. X(STPACK,PSIZE(EXEC)+1) =
              INACTIVE THEN
              RECALL X(STPACK,PSIZE(EXEC)+1) FROM XLAB1;
              ELSE RECALL X(STPACK,PSIZE(EXEC)+1);
            END IF;  END IF;
     END PROCESS EXEC;

     PROCESS V ENABLED;
     /* INTERRUPT PROCESS FOR SOURCES 1,3 */
     ACCESS PUB, XPUB, EXPUB;
     INITIAL SETUP V(STPACK, 3*PSIZE(V)+1);
     INTMASK = 1010L;     /* INTERRUPTS 2,4 ENABLED */
/VLAB1/  GLOBA = 1;          /* PROCESS INTERRUPT 1 */
     CALL WRITB;             /* LIBRARY ROUTINE */
```

```
      SWITCH = 1;
      RECALL EXEC(STPACK,2*PSIZE(V)+1) FROM EXCONT;
/VLAB2/  GLOBC = 1;          /* PROCESS INTERRUPT 3 */
      CALL WSTATCHK;         /* LIBRARY ROUTINE */
      SWITCH = 1;
      RECALL EXEC(STPACK, 2*PSIZE(V) +1) FROM EXCONT;
     END PROCESS V;

      PROCESS K ENABLED;
      /* INTERRUPT PROCESS FOR SOURCES 2,4 */
      ACCESS PUB, TPUB, EXPUB;
      INITIAL SETUP K(STPACK,4*PSIZE(K)+1)
           INTMASK = 101L;
      /* INTERRUPTS 1,3 ENABLED */
/KLAB1/  GLOBB = 1;
      CALL READ;  /* LIBRARY ROUTINE */
      SWITCH = 0;
      RECALL EXEC(STPACK, 2*PSIZE(K) +1) FROM EXCONT;
      GLOBD = 1;              SWITCH = 0;
      CALL RSTATCHK;         /* LIBRARY ROUTINE */
      RECALL EXEC(STPACK, 2*PSIZE(K)+1) FROM EXCONT;
     END PROCESS K;
```

4.0 Implementation

Implementation of interrupt handling facilities and interrupt simulation involves three levels of modification to the LITTLE compiler. An addition to the LITTLE grammar is made; this is shown in Appendix B. Additions to the parser-generator (GEN) are made; these additions accomplish semantic checking, and build preliminary interrupt-related machine-independent macro-code and auxiliary tables as shown in Appendix C. Finally a set of modifications is made to the LITTLE assembler phase (ASM), to produce interrupt simulation code for the CDC 6600; these additions are shown in Appendix D. The macro-code produced by the GEN phase is designed to be used for either simulation mode or code generation mode. Only the simulation mode is implemented in the ASM phase for the CDC 6600 since this machine does not have hard interrupts. This Section 4 is intended as a supplement to the LITTLE System Manual and references data structures and routines described therein.

4.1 Grammar Modification

ROUTINES IN GEN: DEFLIT, ERMET

DATA STRUCTURES IN GEN: LITTAB

LITTLE is defined syntactically by a 'statement' grammar written in top-down metalanguage. New statements and primitives were added by modifying this 'statement' grammar as shown in Appendix B. Corresponding syntax error messages are

added to the ERMET routine and additional entries for new key words were made to the LITTAB table in the DEFLIT routine which implements the 'branch-on-literal' speedup of the parsing algorithm.

4.2 Parser-Generator (GEN) Modification

4.2.A Program Level Syntax

ROUTINES: GENPROC, GENSUB, GENEND, GENRET

DATA STRUCTURES: VOA, COSA

LITTLE's program level syntax is not represented in the formal metalanguage but rather it is encoded in GEN. The program syntax for the basic LITTLE language is a series of well formed subroutines and/or functions. The interrupt handling facilities modify the program syntax as follows:

<program> => <process> <process * 0>

<process> => <procheader> <main routine>
<subprogram * 0> END;

<procheader> => PROCESS <*name> ENABLED;
=> PROCESS <*name> DISABLED;

<subprogram> => [any well-formed subroutine or function]

<mainroutine> => [any LITTLE statement including interrupting handling statementsbut excluding a SUBR, FNCT, RETURN, or END statement.]

Within GENPROC, GENEND, and GENSUB are syntax checks to prohibit code or subroutines between processes; to prohibit code between subprograms, and to check for a single END card in processes that have no subprograms. The COSA stack is used to keep track of compound statement openers and a 'process-type' entry for the current process is the first entry in the COSA stack.

Since the basic unit of compilation for the LITTLE compiler is a subprogram and some compiler tables are cleared at the end of each routine, the code and declarations of the main routine are made into a special routine with no parameters, entered by a transfer instruction. The name of the process becomes the routine name. GENPROC calls GENSUB to build a VOA entry for a subroutine; then GENPROC builds a VOA 'process-operation' entry. This first subroutine is ended upon encountering a SUBR, FNCT, or END statement. If the process has other subprograms, then the GENSUB routine flags the end of the main routine and calls GENEND to do the standard 'end-of-routine' processing. If the process has no other subprograms, GENEND flags the end of the main routine, performs the 'end-of-routine' processing and then performs any 'end-of-process' processing needed. As a result of the above actions, all intermediate code is sent to the assembler phase in the form of subprograms, which minimizes the amount of changes needed in the assembler phase for 'processes'. The detailed changes needed

in target code compilation are directed by new VOA operation types within the bounds of a subroutine. The generator GENRET checks that no RETURN statements occur in the source code for the main routine.

4.2.B Namescoping

ROUTINES: GENNS, GENPNS, GENACC, GENPROC, GENSUB, GENEND, GENQNAM, INSGLOR, IFAGLOR, INSNAMR

DATA STRUCTURES: NBLOCKTAB, PPNSETS, DEFACCESSTAB, ACCESSTAB, HA, XHA, VOA, NL

Public namesets are processed by the GENPNS routine in much the same way as private namesets are processed in GENNS. The nameset name is saved for use by the ACCESS statement generator. An entry is made in NBLOCKTAB for the new nameset and a corresponding entry is flagged in an auxiliary table, PPNSETS. GENNS flags a nameset's PPNSETS entry as private and GENPNS flags a nameset's PPNSETS entry as public. These two routines also set bits in the ACCESSTAB making the nameset accessible to the current routine. At the end of a process, GENEND flags all private nameset entries in PPNSETS as nonaccessible to other processes. The GENACC routine prevents accessing of namesets private to another process.

In handling the first routine nameset control card option, GENSUB sets bits in ACCESSTAB making the new nameset accessible to the first routine. No other modification is

needed since GENPROC reinitializes the subroutine count for each new process. The default access option, when requested on a control card, is implemented by clearing DEFACCESSTAB at the beginning of each process. In GENEND at the end of the first routine in the process, ACCESSTAB is paged into the DEFACCESSTAB table. Then in each subsequent routine GENSUB pages DEFACCESSTAB back into ACCESSTAB making all the namesets available to the first routine of a process accessible to subsequent routines.

GENPROC saves process names and their sequential numbers by calling INSGLOR to hash the name into the XHA global table, which now includes a new field for the process number. No entry is needed for process names in the NL array of global attributes.

The INSGLOR routine hashes new private global operands into the XHA, also using the new field for the process number. The IFAGLOR routine, when requested to check if a private global operand has been previously defined, checks the process number field for a match. The process number field is set to the sequential index number of the process in which the operand is declared.

The GENQNAM routine intercepts references to private global variables from another process by calling INSGLOR to hash the processname provided into the XHA and using the process number found from that hash probe as the process number for the private quantity name-XHA hash probe.

GENQNAM flags an error if the variable name was not previously defined in the given process. GENQNAM then calls INSNAMR to hash the private quantity into the HA symbol table with a new HA field for the process number. GENQNAM pages the appropriate attributes of the operand from the global NL table into the VOA. Of course, a new VOA entry is made for a particular operand only on its first occurrence in a subprogram. The EMIT2 routine is called to build a fetch operation VOA entry for the accessing of an indexed private global variable.

4.2.C Statement Processors

ROUTINES: GENPROC, GENSETUP, GENGOL, GENEND, GENATCH, GENDATEN, GENSIML, GENINTR, GENIMASK, GENCALL, GENENAB, GENSAVSTAT

DATA STRUCTURES: VOA, HA, NL, ARGSTACK

The new interrupt handling statements are processed by a group of new generator routines. These routines build VOA entries using previously defined operation codes where possible and new VOA operation codes where necessary. Eleven new operation types are added. The statement generator routines are invoked by ACTGEN, the action routine in the parser. This section also discusses the processing of dictions that form parts of the interrupt handling statements, the processing of the PSIZE builtin function, the processing of labels as entry points, and assignments made to INTMASK.

GENPROC, the process declaration generator, generates the public global arrays and variables used for the execution time implementation of the interrupt handling facilities. These are the ATTACHMENTS array, the ATTINDICES array, the pointer CDP, the WAS→ENABLED flag, the ENABLE flag, and the counter EXEC→COUNT. They are declared in a nameset named PUBLOCK. (A LITTLE nameset may be accessed in an assembly language routine by a COMMON block declaration of the same name.)

In simulation mode GENPROC checks if the current process is the simulation process. For the first routine of a simulation process, GENPROC generates a parameter named INTMASK. For other processes, GENPROC builds VOA entries to restore the state of the process when reentered by a simple recall. It also generates a private nameset for such a process, containing variables named INTMASK and PROCENABLE, representing the interrupt mask value and the enable state of the process. The INPROC flag, which is used throughout GEN for conditional code compilation for processes, is set.

The SETUP statement generator, GENSETUP, when processing statement forms (3) and (4) explicitly assigns a new machine address to the variables INTMASK and PROCENABLE. Using the named *array* and specified *index* given in the statement as a base address (*array*(*index*)) for the status package storage area. GENSETUP assigns the address of the second location in the storage area to the variable INTMASK,

and the address of the third location in the storage area to PROCENABLE. The VOA and NL entries of these names are modified to reflect these new addresses. GENDAT is called to build 'data' VOA entries for operands initialized on an INITIAL SETUP statement after checking the global names tables to verify that they were previously defined as private variables within the named process. In processing a statement form (4), GENSETUP saves the processname, array name, index value, and label name from the statement and sets a flag to direct the GENEND routine to build a subprogram named START. After the current routine is processed, GENEND uses the information saved above to generate a routine containing a

RECALL *processname(array,index)* FROM *label*;

instruction.

Labels in the main routine of a process are intercepted by the GENGOL routine. This routine builds an 'entry point' VOA entry and a 'load interrupt mask' VOA entry at each label. Then it calls GENENAB to emit an 'enable interrupt' operation for processes that are entered at labels with interrupts enabled.

GENENAB implements the code expansion for the ENABLE and DISABLE statements by calling the GENIF and GENASIN routines to emit the following code:

for ENABLE:

```
ENABLE = 1;
PROCENABLE = 1;
```

for DISABLE:

```
IF ENABLE THEN
 WAS→ENABLED = 1;
  ENABLE = 0;
  PROCENABLE = 0;
  ELSE WAS→ENABLED = 0;
  END IF;
```

GENENAB builds VOA entries for the primitive enable and disable operations to be used in code generation mode by the assembler.

The GENATCH routine processes all recall and attach statements. In processing the attach operation it calls GENENAB to generate code to disable interrupts for this critical section and to reenable them if necessary after the attach, which itself is emitted as a new VOA entry. In processing a recall statement, GENATCH calls GENENAB to disable interrupts, calls GENSAVSTAT to build VOA entries for the save process-state operation, and finally builds a recall VOA entry. The basic block is ended here in simulation mode for consistency with the processing of simulated interrupts. No block is ended in code generation mode for consistency with the action of external interrupts. In either situation, however, action is taken to save the enable state of the process before the preliminary disable takes place.

GENDATEN processes the data environment diction

(14) *processname(array,index)*

for the RECALL and ATTACH statements. If reentrancy is provided in a future compiler, GENDATEN is to be used to process this diction for the run time SETUP statement (5) and for qualified name references (a) and (b). GENDATEN checks that the named *array* has been correctly sized and dimensioned. It builds a VOA entry which calculates the

data area base address.

The GENSIML routine processes a SIMULATE statement, saves the process name, and sets flags to indicate simulation mode. It initializes the variable LASTINTR used to estimate at compile time the number of instructions since the previous simulated interrupt.

The routine GENINTR processes the INTERRUPT statement by verifying that the statement occurs inside a simulation process. It calls GENASIN to generate the assignment

CDP = ATTINDICES(interrupt source number)

and builds an interrupt-VOA-entry which effects an indirect jump according to the address in the array element

ATTACHMENTS(interrupt source number)

The 'interrupt source number' is the value of the *n*-expression given on the interrupt statement (14).

An assignment to the variable INTMASK is handled by the GENIMASK routine. Interrupts are disabled during this critical code section by a call to GENENAB. The parser calls GENASIN to emit code for the global variable assignment. GENIMASK subsequently builds the VOA entry needed in code generation mode to load the interrupt mask register. GENIF and GENENAB are called to generate code which reenables interrupts if necessary.

The PSIZE builtin function is intercepted in the GENCALL routine. An implementation dependent constant value is pushed

on the ARGSTACK to be used in whatever expression is being parsed.

4.2.D Interrupt Simulation

ROUTINES: GENLABL, GENCALL, GENSAVSTAT,GENSIM,EMCALL

DATA STRUCTURES: VOA

Interrupt simulation is handled by the GENCALL routine for subroutine calls and by the GENLABL routine for user-defined labels. GENSIM is called to estimate the number of instructions since the last simulated interrupt and to call other generators to emit preliminary code from the sequence specified in Section 3.6. GENCALL and GENLABL call the GENSAVSTAT routine to build a 'save process-state' VOA entry, and use the EMCALL routine to generate a call to the simulation process.

4.2.E Miscellaneous

ROUTINES: BLOCKEN, COSADMP, VOAJUNK

DATA STRUCTURES: KIND, TYPES, JOBLAB

Entries are made for the new VOA operations in the KIND array, for use by the BLOCKEN routine, in processing basic blocks.

New entries are made in the COSADMP routine's TYPES array and in the VOAJUNK routine's JOBLAB table and GOBY statements for printing the VOA and COSA tables during compiler debugging.

The following table summarizes VOA fields of all new, interrupt-related VOA entries. The operations and fields are used to generate target code for both simulation and code generation modes unless specified otherwise.

Opcode	Operation	Field	Use
66	data area	INP1	VOA ptr to array
	base addr	INP2	VOA ptr to index
		OUP	VOA ptr to result
67	save	NAYM	HA ptr to label
	process-state	INP1	VOA ptr to CDP
		INP2	LABLIST ptr to label
68	attach	INP1	HA ptr to process entry point
		INP2	VOA ptr to data area base address
		INP3	VOA ptr to interrupt source number
		ARGBEG	XARG ptr to builtin arrays
		ARGLEN	number of arrays (= 2)
			(first XARG - VOA ptr to ATTACHMENTS)
			(second XARG- VOA ptr to ATTINDICES)
69	recall	INP1	HA ptr to process entry point
		INP2	VOA ptr to data area base address
		INP3	VOA ptr to CDP
*70	interrupt	INP1	VOA ptr to ATTACHMENTS array
		INP2	VOA ptr to interrupt source number
71	process opener	no fields	

[continued]

Opcode	Operation	Field	Use
72	entry point	NAYM	HA ptr to entry point name
**73	enable	no fields	
**74	disable	no fields	
**75	load interrupt mask	INP2	VOA ptr to INTMASK
76	restore process state	INP1	VOA ptr to CDP
		INP2***	switch for enable or disable interrupts

* operation used in simulation mode only

** operation used in code generation mode only.

*** field used in code generation mode only.

4.2.F Variable Dictionary (Generator Phase)

BEGARRAY

Saves array name, as a self-defined string, for process to be invoked at first. Saved in GENSETUP, used in GENEND.

BEGINDEX

Saves index value, as a constant, for process to be invoked at first. Saved in GENSETUP, used in GENEND.

BEGLABL

Saves label, as a self-defined string, for process to be invoked at first. Saved in GENSETUP, used in GENEND.

BEGPROC

Saves process name, as a self-defined string, for process to be invoked at first. Saved in GENSETUP, used in GENEND.

DOUBLENDFLG

If OFF, checking for a process without subprograms.

ENDFLG

Flag to prevent interrupt simulation at point of call to the 'abort system' routine.

GENRECALL

Flag set if generating a RECALL operation, used in GENENAB to prevent changing value of process enable status.

HA

New field used for private global variables.

PROCNUM

gives process number (NUMPROC) in which variable was defined.

IMASKFLG

Set if assignment to INTMASK being processed.

IMPLPSIZE

Implementation dependent dimension of status package area for a single process.

INITBEGFLAG

Set if 'INITIAL SETUP ... FROM label' statement encountered. Used in GENEND to initiate generation of START routine.

INPROC

Flag set if compiler is processing a 'process'. Flag set in GENPROC.

INPROCHDR

Flag set if processing the main routine of a process.

INSIMLPROC

Flag set if processing the simulation process.

INTHNDFCL

Flag set if interrupt handling facilities are being used. Used in GENSUB to prohibit singular subprograms outside the bounds of a process.

INTRSTAT

Set if process' initial enable status is 'enabled', OFF if 'disabled'.

LASTINTR

VOA pointer to beginning of preceding interrupt simulation code sequence. Used to estimate the number of instructions since last simulated interrupt.

MDATENVSZ

Implementation dependent size needed for arrays, used to hold process' data environment.

NEVERINPROC

Flag set at beginning of a compilation. Used to perform process initializations needed only once, such as the generation of a public nameset used for the interrupt handling facilities.

NUMINSTR

Counter for number of VOA operation entries. Used in simulating interrupts.

NUMPROC

Number of processes encountered in compilation.

NUMSOURCES

Implementation dependent value giving the number of interrupt sources recognized in the system.

PPNSETS

Table used to indicate type of nameset. Values may indicate public, private to process being compiled, or private to another process.

PNTOUSE

Process number to use in HA, and XHA hash probes for private global variables. Value used to set a new field in the HA and XHA.

PRCHECK

Flag set when hashing a process name into the global names table.

PROCBLOCK

Machine block number (nameset number) of compiler generated private nameset.

PROCNAME

Name of process being compiled, as a self-defined string.

PUBPROCBLOCK

Machine block number (nameset number) of compiler generated public nameset.

QUALNAME

Set if processing private global variables from another process. Used in IFAGLOR to omit checking the ACCESSTAB and in INSNAMR to include checking the 'process number' field for an HA probe.

SIMLFLAG

Set if in simulation mode. Used to generate simulated interrupts.

SIMLPROC

Name of the simulation process, as a self-defined string.

VOA

New operations for interrupt-handling are:

OPCODE	Operation Code
66	DATENV, calculates data area base address
67	SAVSTATE, save process-state
68	ATTACH
69	RECALL

[continued]

OPCODE operation code

70 INTERRUPT

71 PROCESS, opener

72 ENTRY POINT, multiple entry point for entering from label

73 ENABLE

74 DISABLE

75 INTERRUPT MASK LOAD

76 RESTORSTAT, restore process-state

XHA

New field used for private global variables

XPROCNUM

gives process number (NUMPROC) in which variable was defined.

4.2.G Macro Dictionary (Generator Phase)

GETNAME(HC, R)

Gets HA pointer of a previously defined variable. R is a self-defining string and HC is its HA index returned.

GETPROCNUM(XGLHAP, XHAP)

Hashes process name into global hash table XHA by using IFAGLOB macro. XHAP is the HA index of the process name. If name found, XGLHAP is the XHA index. Otherwise, XGLHAP is 0.

LASTVOA

Last VOA entry built.

PROCINVOA(XHAP)

Builds a 'SUBR' type VOA entry for a process, XHAP is its HA pointer.

4.2.H Routine Dictionary (Generator Phase)

GENLABL

Called from ACTGEN, uses GENGOL to generate user-defined labels.

GENATCH(OPTYPE,LTYPE)

Generator for ATTACH and RECALL statements. Builds new VOA entries. Called from ACTGEN and other generators.

OPTYPE = 1, generate attach sequence

OPTYPE = 2, generate recall sequence

LTYPE = 1, simple recall or attach

LTYPE = 2, recall or attach ... from label

GENDATEN

Generator for data environment base address. Builds VOA for address calculation. Called from ACTGEN and other generators.

GENENAB(EDW)

Generator for ENABLE (EDW = 1) and DISABLE (EDW = 2) statements. Called from ACTGEN and other generator

routines. Calls GENASIN and GENIF for part of implementation. Builds 'enable' and 'disable' VOA entries for code generation mode.

GENIFEND

Implements an 'END IF' statement, closing a compiler generated 'IF' statement.

GENIMASK(CASE)

Called from ACTGEN to process an assignment to INTMASK, builds a VOA entry for the register load.

CASE = 1, calls GENENAB to disable interrupts before the assignment.

CASE = 2, calls GENENAB to reenable interrupts after building 'load-interrupt-mask' VOA.

GENINTR

Generator for interrupt statement. Called from ACTGEN. Builds VOA entry. Calls GENASIN for assignment to CDP variable.

GENPNS

Generator for public nameset declarations. Called from ACTGEN.

GENPROC(STAT)

Generator for process declaration statement.

STAT = 1, process entered enabled;

STAT = 0, process entered disabled.

Generates necessary public and private global variables for interrupt handling. Calls GENSUB to initialize a new routine. Does program level syntax

checking and builds 'process entry' and 'restore process state' VOA entries.

GENQNAM(OPTYPE,INDXD)

Generator for reference to private global variables. Called from ACTGEN. Checks if quantity named is an appropriately predefined private global variable.
OPTYPE = 0, access variable; INDXD = 0, simple variable.
OPTYPE = 1, modify variable; INDXD = 1, indexed variable.
Calls ARITH to generate fetch operation for indexed variable.

GENSAVSTAT(LABPTR)

Generator for save process-state operation. Called by GENATCH for recall statement and by GENCALL and GENLABL for interrupt simulation. Builds VOA entry.

GENSETUP(CASE, SUBCASE)

Processes the various SETUP statements. Called by ACTGEN.
CASE = 1, INITIAL SETUP ... FROM LABEL ...
CASE = 2, INITIAL SETUP ...
CASE = 3, SETUP ...
CASE = 4, data initializations.
SUBCASEs 1-4, same as four cases of GENDAT routine.
GENSETUP calls GENDAT to build VOA entries for data initializations.

GENSIM

Implements preliminary code for interrupt simulation. Called by GENCALL and GENLABL.

GENSIML

Processes simulate statement. Saves process name and sets appropriate compiler flags. Called from ACTGEN.

4.3 Assembler (ASM) Modification

Our addition to the assembler (ASM) program consists of new code generator routines and an addition to the loader tables produced by the compiler for the CDC 6600.

4.3.A Code Generation

ROUTINES: ASSEMBL, ASMPROC, ASMDATENV, ASMRCLL, ASMATCH, ASMSVSTAT, ASMINTRPT

DATA STRUCTURES: OPKIND, CB, VOA

The new code generation routines are invoked from ASSEMBL, the main routine of ASM, using the VOA operation description from the OPKIND table. The ASMPROC routine invoked by a 'process-operation' VOA entry reinitializes the buffer, CB, to begin a code block that is entered by a transfer instruction and without parameters.

The code generator routines ASMPROC, ASMRCLL, ASMATCH, ASMINTRPT, ASMSVSTAT, and ASMDATENV, generate in-line code for the 'restore process-state', 'recall', 'attach', 'interrupt', 'save process-state', and 'calculate data environment base address' VOA operations respectively. Constant folding code optimization is performed in the ASMATCH and ASMINTRPT routines.

4.3.B Loader Table Addition

ROUTINES: ASSEMBL, LDRENTR, LDROUT

DATA STRUCTURES: LGOEXENTPNTS, VOA, ENTR

The LDRENTR routine is invoked by the ASSEMBL routine upon encountering an 'entry point' VOA entry. LDRENTR saves the name and address of the process' multiple entry points in the LGOEXENTPNTS table, to be written out as part of the ENTR loader table by the LDROUT routine.

4.3.C Variable Dictionary (Assembler Phase)

LGOEXENTPNTS

Table for extra entry points. To be written out as part of the ENTR loader table.

4.3.D Routine Dictionary (Assembler Phase)

ASMATCH

Code generator for attach operation, called from ASSEMBL. Emits code to save process entry point and process data area address in run-time tables.

ASMDATENV

Code generator for calculating base address of a process data environment area. Called from ASSEMBL.

ASMINTRPT

Code generator for interrupt operation. Emits code to transfer to process attached to a particular interrupt source. Called from ASSEMBL.

ASMPROC(OPSWITCH)

Initializes a process' main routine (OPSWITCH = 0) and generates code to restore the process-state (OPSWITCH = 1). Called from ASSEMBL.

ASMRCLL

Code generator for recall operation. Emits code to save data area base address of next process and to transfer to next process. Called from ASSEMBL.

ASMSVSTAT

Code generator to save process-state. Saves program location counter by generating code to save a label address. Called from ASSEMBL.

LDRENTR(ENTNAME)

Routine to save name and relocation address of an entry point in the table LGOEXENTPNTS. ENTNAME is the entry point name-string. Called from ASSEMBL.

APPENDIX A. Guide to the LITTLE Interrupt Handling Statements

In this appendix, we present the statements of the Interrupt Facilities extension to the LITTLE language. We begin with an index of the statement names and formats:

PAGE	NAME	FORMAT
	ACCESS	ACCESS ID1,ID2,...;
	ATTACH	
	A. SIMPLE	ATTACH E1 TO ID(A,E2);
	B. FROM LABEL	ATTACH E1 TO ID(A,E2) FROM L;
	DISABLE	DISABLE;
	ENABLE	ENABLE;
	INTERRUPT	INTERRUPT E;
	NAMESET	NAMESET ID;
	PROCESS	
	A. ENABLED	PROCESS ID ENABLED;
	B. DISABLED	PROCESS ID DISABLED;
	PSIZE	PSIZE(ID)
	PUBLIC NAMESET	PUBLIC NAMESET ID;
	RECALL	
	A. SIMPLE	RECALL ID(A,E);
	B. FROM LABEL	RECALL ID(A,E) FROM L;

[continued]

PAGE	NAME	FORMAT
	SETUP	
	A. INITIAL	INITIAL SETUP ID(A,E1) V = VAL: V = E2,E3,...;
	B. FIRST TO BEGIN	INITIAL SETUP ID(A,E1) FROM L V = VAL: V = E2,E3,...;
	SIMULATE	SIMULATE IN ID;

ACCESS statement

PURPOSE

To allow a subprogram to refer to public or private global variables, previously defined in a NAMESET or PUBLIC NAMESET statement, and to indicate the namesets to be accessed.

FORMAT

```
ACCESS NAMESET1, NAMESET2, ..., NAMESETK;
ACCESS NAMESET1;
```

ADDITIONAL RULES:

1. Except as noted below, all rules given for the ACCESS statement in the User's Guide for the basic LITTLE language are applicable.
2. Routines (main routine or subprogram) may access any previously defined public nameset.
3. Routines may access any previously defined ordinary (private) nameset, defined within the same process. Routines may not access private namesets defined in other processes.

EXAMPLES:

```
ACCESS INTRVARS, PROCTABLE;
```

ATTACH (simple) statement

PURPOSE

To associate an interrupt coming from a particular source with a process to be recalled when that interrupt occurs. To indicate the location in the process where execution is to begin, to name the process and to locate the process' data environment.

FORMAT

ATTACH C TO ID(A,E);

RULES

1. C specifies the specific interrupt source and must be a positive integer-valued expression. We assume the set of external interrupt sources recognized by the system is numbered 1,2,3,... .
2. ID must be a process name, A must be a previously defined array, and E must be an integer-valued expression. ID identifies the process to be recalled and array A at location E specifies the beginning of the process data area.
3. The execution of an attach statement automatically detaches an interrupt source from any other process to which it may have been attached.
4. When external interrupt C occurs, the process-state of the current process is saved in its data environment and control transfers to process ID.

[continued]

Execution begins from the instruction recorded in process ID's data environment after registers and status indicators are restored from values recorded in the same data area. The process' enable status is also restored.

EXAMPLES

```
ATTACH 1   TO PROC1(ARRAY,1);
ATTACH N-1 TO PROC2(ARRAY2,M);
```

ATTACH (from label) statement

PURPOSE

To associate an interrupt coming from a particular source with a process to be recalled when that interrupt occurs. To name the process, to locate the process' data environment, and to indicate the location in the process where execution is to begin.

FORMAT

ATTACH C TO ID(A,E) FROM L;

RULES

1. C specifies the specific interrupt source and must be a positive integer-valued expression. We assume the set of external interrupt sources recognized by the system is numbered 1,2,3,... .
2. ID must be a process name, A must be a previously defined array, E must be an integer-valued expression and L must be a label in the main routine of the process. ID identifies the process to be recalled, and array A at location E specifies the beginning of the process data area.
3. The execution of an attach statement automatically detaches an interrupt source from any other process to which it may have been attached.
4. When external interrupt C occurs, the process-state of the current process is saved in its data environment, and control transfers to process ID. Execution

begins there at label L in the main routine after reloading the interrupt mask register from the value recorded in the process' data environment.

EXAMPLES

```
ATTACH 1   TO PROC1(ARRAY1,1) FROM L1;
ATTACH N-1 TO PROC2(ARRAY2,M) FROM L2;
```

DISABLE statement

PURPOSE

To disable all interrupts; to clear a public global flag 'ENABLE' indicating the system enable status; to save the previous system enable status.

FORMAT

```
DISABLE;
```

RULES

1. When the disable statement is executed, the previous enable status is saved in the 1-bit global public variable WAS→ENABLED. This variable is set to 1 if interrupts were enabled, and is set to 0 if interrupts were disabled.
2. Execution of the disable statement clears the 1-bit public global variable ENABLE. This variable may be accessed but not modified.
3. The user is advised to save the value of WAS→ENABLED after any invocation of the disable statement since this variable's value may unpredictably be modified by other interrupt handling statements.

EXAMPLES

```
DISABLE;
PREVENAB = WAS→ENABLED; $SAVE PREVIOUS STATUS
IF PREVENAB CALL BACKUP;
```

ENABLE statement

PURPOSE

To enable all interrupts; to set a public global flag indicating the system enable status.

FORMAT

```
ENABLE;
```

RULES

1. When the enable statement is executed, a one-bit public global flag, ENABLE, is set. This variable may be accessed but not modified.
2. All interrupts not masked out by the interrupt mask are enabled when an enable statement is executed. The interrupt mask is represented by a private global variable. INTMASK, of a size equal to the number of external interrupt sources recognized in the system. The jth bit in INTMASK corresponds to the jth interrupt source. When the jth bit is 1, the interrupt source is individually enabled. An assignment to the variable INTMASK causes a reload of the interrupt mask register.

EXAMPLES

```
INTMASK = 7;  $ BECOMES CURRENT MASK VALUE
ENABLE;       $ INTERRUPTS 1,2,3 ARE ENABLED
INTMASK = 3;  $ INTERRUPTS 1,2 ARE ENABLED
```

INTERRUPT statement

PURPOSE

To recall a process attached to a particular interrupt source; to specify the interrupt source.

FORMAT

```
INTERRUPT NUM;
```

RULES

1. NUM must be a positive integer-valued expression. The interrupt sources in a particular system are numbered 1,2,3,... . NUM refers to a particular interrupt source.
2. Some process must have previously been attached to the specified source.
3. An interrupt statement may be used only within a simulation process, i.e., a process named on a SIMULATE statement.
4. Upon execution of an INTERRUPT statement, control passes to the process currently attached to interrupt source NUM. Execution begins in that process either at a location recorded in that process' data environment or from a label in the process' main routine, depending on which type of ATTACH statement was used.

EXAMPLES

```
INTERRUPT 6;
INTERRUPT NEXTOEXEC;
```

NAMESET statement

PURPOSE

To indicate the name of a set of private global variables, to begin definition of a set of private global variables.

FORMAT

```
NAMESET ID;
```

RULES

The rules are the same as those for the NAMESET statement in the LITTLE User's Manual except rule #5, which is replaced by:

5a. Variables in a nameset may be referred to in other subprograms in the same process, by using the ACCESS statement.

5b. No variable may belong to more than one nameset defined within a single process.

5c. Nameset names must be unique within a single compilation run.

EXAMPLES

```
PROCESS A DISABLED;
NAMESET B;
SIZE SYMBA(WS);
SIZE SYMBB(WS);  DIMS SYMBB(100);
END NAMESET;
END PROCESS;
PROCESS X DISABLED;
NAMESET Y;
```

[continued]

```
SIZE SYMBA(WS);   $ THESE VARIABLES ARE
SIZE SYMBB(WS);   $ DIFFERENT FROM THOSE
DIMS SYMBB(100); $ IN PROCESS A
END NAMESET;
END PROCESS;
```

PROCESS statement

PURPOSE

To give the name of a process; to indicate its initial interrupt enable status; to initiate definition of a process.

FORMAT

PROCESS ID ENABLED;

PROCESS ID DISABLED;

RULES

1. The PROCESS statement is an opener. The body of a process consists of all the following statements up to the matching END statement.
2. The body of a process may not contain a PROCESS statement.
3. The main routine of a process consists of all the declarations and statements following the PROCESS statement up to the first SUBR, FNCT, or END statement. RETURN statements are not permitted in the main routine.
4. The body of a process consists of a main routine followed by any number of well-formed subprograms. Processes need not have subprograms.
5. No code may appear between subprograms. No code or subprograms may appear between processes.
6. If execution is attempted past the last statement of the main routine, an error condition occurs. Control does not return to a process entry point or to any

previously executing process.

7. Routines in one process may not CALL routines in another process.

8. Execution of a process may begin with the first executable statement after one of the labels in the main routine. At this point, the process has the enable status given on the process statement. The interrupt mask register has been reloaded from the value recorded in the process' data environment. If the process has never been entered before the value is taken from an initializing statement, i.e., either an INITIAL SETUP or a DATA statement.

9. The execution of a RECALL statement or the occurrence of an external interrupt suspends control in the process. Control may return by an explicit recall of the process or by the occurrence of an external interrupt attached to the process.

10. A process may be resumed after being suspended. Execution begins from a location recorded in the process' data environment after restoring registers from the values recorded in this data area. The process' enable status is restored; i.e., if interrupts were previously enabled when the process was last executing, they are reenabled.

EXAMPLES

```
 PROCESS T ENABLED;
 ACCESS PUBLICSET;
 INITIAL SETUP T(ARRAY,1);
 DATA INTMASK = 7;
 /ENTT1/   CALL BUMP1(Y); $ INTERRUPT SOURCES 1,2,3 ARE ENABLED
 RECALL EXEC(ARRAYB,1) FROM EXIT;
 SUBR BUMP1(X);
 SIZE X(WS);
 X = X+1;
 RETURN;
 END SUBR;
 END PROCESS;
PROCESS X DISABLED;
ACCESS PUBLICSET;
INITIAL SETUP V(D,1) FROM ENTV2 INTMASK = 3;
/ENTV1/ A = 2;  $ ALL INTERRUPTS DISABLED
RECALL V(E,1);
/ENTV2/ ENABLE;    $ INTERRUPT SOURCES 1,2 ARE ENABLED
RECALL K(F,1);     $ AT INITIAL ENTRY
END PROCESS X;
```

PSIZE builtin function

PURPOSE

To return a value equal to the number of words of storage needed for a process' data environment.

FORMAT

```
PSIZE(ID)
```

RULES

1. When this function call occurs in an expression, an implementation dependent constant, equal to the number of words of storage needed for a process' data environment, replaces the sequence of tokens which invoked the function, and evaluation of the original expression continues.
2. The function argument is a process name.

EXAMPLES

```
SIZE STORARRAY(WS);
DIMS STORARRAY(6 * PSIZE(A));
```

PUBLIC NAMESET statement

PURPOSE

To indicate the name of a set of public global variables; to begin definition of a set of public global variables.

FORMAT

PUBLIC NAMESET ID;

RULES

1. The public nameset is an opener. The body consists of all the following statements up to the END statement which terminates the public nameset group.
2. Any size statement within the body of the nameset statement defines a public global variable, which is a member of the public nameset.
3. The member variables of a public nameset must have distinct names.
4. No variable may belong to more than one nameset defined within a single process.
5. Variables in a public nameset may be referred to in other subprograms or the main routine in any process, using the access statement.
6. A public nameset body may not contain a public nameset or ordinary nameset statement.
7. Variables in a public nameset may be referred to within the routine in which they are defined. No separate access statement is needed.

8. Nameset names must be unique within a single compilation run.

EXAMPLES

```
PUBLIC NAMESET SYMTAB;
SIZE SYMTABPTR(PS);    $ TOP OF SYMBOL TABLE
SIZE SYMTAB(WS);  DIMS SYMTAB(100);
END PUBLIC NAMESET;
```

RECALL (simple) statement

PURPOSE

To suspend execution in the current process and give control to another process; to specify the name of the new process, to specify the first location of the data environment of the new process; to specify the manner in which the new process is given control.

FORMAT

```
RECALL ID(A,INDEX);
```

RULES

1. ID is the name of the next process to execute. The data environment of that process must be allocated to array A beginning at the location specified by INDEX. A must be a previously defined array and INDEX an integer-valued expression.
2. When a simple recall statement is executed, the process-state of the currently executing process is saved in its data environment, the process-state of the new process is restored from the values saved in its data environment and execution begins at the location stored in that data environment.
   (See PROCESS statement for a description of the restoring of the process state.)
3. Execution of a simple recall assumes that the new process has been entered before and that execution was suspended there by the action of an external

interrupt or by the execution of a prior RECALL statement in process ID.\

EXAMPLES

```
RECALL PROCA(STORARRAY,1);
RECALL PROCB(STORARRAY,N);
```

RECALL (from label) statement

PURPOSE

To suspend execution in the current process and give control to another process; to specify the name of the new process; to specify the first location of the data environment of the new process; to specify the manner in which the new process is given control.

FORMAT

```
RECALL ID(A,INDEX) FROM L;
```

RULES

1. ID is the name of the next process to execute. The data environment of that process must be allocated to array A beginning at the location specified by INDEX. A must be a previously defined array and INDEX an integer-valued expression. L must be a label in the main routine of ID.
2. When a 'RECALL...FROM label' statement is executed, the process-state of the currently executing process is saved in its data environment, and execution in the new process begins at the specified label L. Other details of restarting a process from a label in the main routine are given in the PROCESS statement description.

EXAMPLES

```
RECALL PROCA(STORARRAY,1) FROM L1;
RECALL PROCB(STORARRAY,N) FROM L2;
```

INITIAL SETUP (simple) statement

PURPOSE

To reserve storage for a process data environment; to initiate a status package for the process; to initiate private global variables in the process.

FORMAT

```
INITIAL SETUP ID(A,INDEX);
INITIAL SETUP ID(A,INDEX)
 INITVAR1 = INITDATALIST1: INITVAR2 = INITDATALIST2...;
```

RULES

1. Storage is reserved for process ID in array A, beginning at the location specified by INDEX, for process ID's data environment area.
2. ID must be a process name, A must be a previously defined array, and INDEX an integer-valued constant expression. Array A must be defined with a fixed implementation size. The number of array elements in A beginning with location INDEX must be sufficient to hold the process status package.
3. The variables to be initialized must be private global variables declared in process ID.
4. If the data variable is an array element, the subscript must be a compile-time constant.
5. The data list consists of a single compile-time constant expression, unless the data variable is an array element, in which case several values, separated by commas, may occur.

6. Repeated instances of the same value in a data list may be written by writing the value followed by a repetition factor in parentheses. For example:

```
ARRAY = 0,0,0,0,0;
```

is equivalent to

```
ARRAY = 0(5);
```

7. The initial setup statement must occur after the named process declaration statement and after any named variables are declared.

EXAMPLES

```
INITIAL SETUP PROC1(C,1);
INITIAL SETUP PROC2(C,11)
  I = 1: J(3) = 2: ARRAY  = 1,2,6,28:
  BARRY = 1, 0(10), 3(5);
```

INITIAL SETUP (first process to execute) statement

PURPOSE

To reserve storage for a process data environment; to initiate a status package for the process; to initiate private global variables in the process; to name the first process to begin executing; to specify the location in the process where execution begins.

FORMAT

```
INITIAL SETUP ID(A,INDEX) FROM L;
INITIAL SETUP ID (A, INDEX) FROM L
  INITDATAVAR1 = INITDATALIST1: INITDATAVAR2=INITDATAVAR2...;
```

RULES

1. All of the rules for the simple INITIAL SETUP statement form apply to this form.
2. L must be a label in process ID's main routine.
3. At execution time, process ID is the first to begin. Execution begins with the first executable statement after label L in the main routine. Interrupts are enabled or disabled according to the rules given in the process statement description for restarting a process from a label in the main routine.

EXAMPLES

```
INITIAL SETUP PROC1(C,1) FROM L1;
INITIAL SETUP PROC2(C,11) FROM L2
 I = 1 : J(3) = 2 : ARRAY = 1,2,3,6,8 :
 BARRAY = 1, 0(10), 3(5);
```

SIMULATE statement

PURPOSE

To initiate the compilation of simulated interrupts within the source code; to name the process given control when an interrupt is simulated in the source code during execution.

FORMAT

SIMULATE IN ID;

RULES

1. ID must be the process name of a user-supplied simulation process.
2. The compilation of LITTLE source text, following the simulation statement, is modified to include code to simulate interrupts.
3. The frequency of simulated interrupts is controlled by resetting the builtin public global variable EXEC→COUNT, within the simulation process, to some positive value. This variable should be initialized on a DATA statement.
4. The simulate statement must appear before the process header declaration of the named simulation process.
5. The simulation process name may not appear on an ATTACH, RECALL, or any SETUP statement. RECALL statements may not appear within a simulation process.
6. RETURN statements may appear in the simulation process' main routine. They will cause a return to the interrupted process.

7. A reference to the variable INTMASK, within the text of a simulation process gives the value of the interrupt mask in the interrupted process. No assignment should be made to this variable within the simulated process.

8. Execution in a simulation process begins with the first executable statement after the simulation process declaration statement. A simulation process need not have a label in the main routine.

EXAMPLES

```
SIMULATE IN SIMPROC;
PROCESS SIMPROC DISABLED;
IF ENABLE .AND. .F. 1,1,INTMASK THEN
  INTERRUPT 1;
  ELSE RETURN;
END SIMPROC;
```