LITTLE Newsletter # 30

J.Schwartz
December 28, 1973

# Interrupt Handling Facilities in LITTLE

## 1. Introduction

Edi Franceschini has started to explore the use of LITTLE as a language for writing the scheduling and communications packages typical of some of the main CIMS mini-resident systems routines. Of course, this requires interrupt handling facilities. The proposals which follow are suggested by a recent discussion with him.

As the correct basic notion for handling interrupts in an orderly way we suggest the notion 'process'. This notion will be given more formal definition in what follows; for the moment, let us only remark that a process is a program embedded in a complete data environment and thus ready to run. We think of a total system as comprising several processes (which cooperate harmoniously.) Some of the variables(and even arrays) to which processes have access are *private* to the process; storage for these variables is reserved when the process is created (perhaps dynamically). Other variables and arrays are *public*, and may be modified by any process.

The code required to define a process ( but note that a data environment is also required) consists of a 'main program' and the subprograms (subroutines and functions) which the process calls. We take this code to be arranged as follows

(1)
```
PROCESS processname; /* a 'process header line' */
(declarations of all namesets global to the process,
        with sizes, dimensions, and initialisation of
                                        global variables)
(code for the main program)
(code for all necessary subprocedures)
END processname;
```

(See Dave Shields' Newsletter 25 for a discussion of namesets.) The variables in a nameset may be made public by using the declaration

(2) PUBLIC NAMESET *name* ($var_1$,...,$var_n$)

instead of the alternative

(3) NAMESET *name* ($var_1$,..., $var_n$)

which declares the members of a nameset to be global to a process but private.

2. Setup and execution of processes. Access to process attributes.

The first process declared is the first to be given a data environment and the first to begin executing. Additional processes may be created dynamically by using the SETUP statement, which has the form

(4) SETUP *processname (array,index)* $var_1$ = $expn_1$,...,$var_k$=$expn_k$;

when this statement is executed, space for all the private variables of a process of the type named by *processname* (cf.(1)) is reserved in the designated *array*, starting at a location specified by the *index* appearing in (4). Every item needed to store a complete data environment will be held in this space: this includes register contents package, instruction location, procedure return addresses, etc. Execution of (4) involves the following steps:

(a) Variables are initialised as indicated by the text (1) of the process being set up.

(b) The variables $var_1$,...,$var_k$ appearing in (4) (these names refer to variables private to the process being set up) recceive the values specified by the expressions $expn_1$,...,$expn_k$.

The *array* in (4) must be of some fixed, implementation dependent, SIZE. The number of words of storage needed to hold the complete environment of a process is given by a pseudo-function

(5) PSIZE *(processname)*.

In addition to its explicitly declared variables, each process has an additional global variable called INTMASK, of an implementation determined SIZE IMASKSIZE. This mask is used, in a manner to be explained later, to allow and to repress the ability of a process to respond to external interrupts.

Once a process is set up, control may be passed to it, either explicitly by executing a RECALL statement, or automatically in response to an externally generated interrupt. RECALL statements have one of two possible forms

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

(6b) RECALL *processname (array, index)* FROM *label*

The statement (6a) resumes execution of the process *processname*, using the data environment stored in *array* beginning at the location *index*. Execution resumes from the instruction location recorded in that environment; this location is presumably that at which execution of the process with this environment was last suspended, either by the execution of a prior RECALL statement or by some earlier external interrupt. Form (6b) of the RECALL statement has much the same effect as (6a) except that execution resumes with a forced transfer to label, which must label some point in the main program of *processname*; in addition, register reload is ommitted.

For uniformity, we must make available the environment storage location of the first declared process, which sets up all the others. This is done by a statement

(7a) INITIAL SETUP *processname (array,index)* $var_1 = const_1, \ldots,$ $var_n = const_k$ BEGIN

which must be prefixed to all other executable statements defining a set of processes, and which acts much like a SETUP followed by a RECALL. Of course, in (7a) *index* must be a constant. The same rule applies to the similar statement form

(7b) INITAL SETUP *processname (array, index)* $var_1 = const_1, \ldots,$ $var_n = const_n,$

which can be used to ensure that a number of processes are properly set up prior to execution of the first statement of the process entered first.

We allows one process to access (and to modify) private global variables of another. Such quantities may be referenced as

(8) *processname.quantityname (array,index).*

Here, *processname* is the name with which (the text of) a process has been declared, as in (1); *quantityname* designates a quantity declared global in the text(1); *array* and *index* serve to locate the place where the data environment of a particular process instance is stored. A private quantity's size (and dimension , if any) is taken from the defining text of the process to which the quantity is private. If a private quantity is dimensioned, its components should be accessed in the form.

(9) *processname.quantityname (array,index, extraindex).*

As compared to (8), (9) contains an *extraindex* which defines the particular component of a private array to which reference is being made.

## 3. External Interrupts, Enable, Disable, Mask . Representation of Peripheral Operations

External interrupts are treated as autonomously forced RECALL operations of one of the two forms(6a) and (6b). Some finite set of external interrupt sources is assumed to exist, and we number these sources 1,2,3,... . Then what we need is a way of associating an interrupt coming from a particular source with a process to be RECALLED when the interrupt occurs.

For this purpose, the statement forms

(9a) ATTACH *n* To *processname (array,index)*
(9b) ATTACH *n* To *processname (array,index)* FROM *label*

are provided. Here *n* is an integer denoting an interrupt source (actually, there is no reason why *n* should not be an integer-valued expression). When an interrupt with source *n* occurs, the system behaves as if a statement of form (6) corresponding to one of the statements of form (9) has been executed.

It is well known that to manage interrupts one needs some way of disabling interrupts temporarily in 'critical' program sections. This possibility is provided by a statement

DISABLE.

The converse operation is effected by a statement

ENABLE

we assume that each time a DISABLE is executed a global public quantity called WAS_ENABLED is set to 1 if interrupts were previously enabled, and to zero otherwise. This bit may be unpredictably modified by any other of the LITTLE interrupt handling statements, and the programmer should if necessary save the bit around any invocation of such a statement.

When a process is recalled, one may wish to enter it with the ENABLE bit either on or off. The default setting is off; to make it possible to enter a process with the ENABLE bit on, we provide an instruction

PRENABLE.

If this instruction is executed before a RECALL, the RECALL will leave interrupts enabled.

Similarly, if a process is recalled in response to an external interrupt, one may wish further interrupts to be disabled temporarily on entry to the process. Disabled process entry after an interrupt will be made if we use one of the following optional forms of the ATTACH statements (9a) and (9b):

(10a) ATTACH *n* To *processname(array,index)* DISABLE
(10b) ATTACH *n* To *processname (array,index)* FROM *label* DISABLE.

An enable-disable condition is completely global.

Finer screening of interrupts than is provided by a single enable/disable bit will sometimes be necessary. We provide for this by agreeing that in addition to its explicitly declared variables each process has an additional global variable called INTMASK, of a size IMASIZE equal to the number of external interrupt sources recognised in a given implementation. The j-th external interrupt source corresponds to the j-th bit in INTMASK. If this bit is 1 in the INTMASK of a process, the interrupt source is (individually) enabled and can interrupt the process; otherwise it cannot. Note that a different INTMASK is associated with each process. This mask is saved when control exits from the process (either in response to an external interrupt or by an explicit RECALL) and is restored when control returns to the process.

We shall not attempt to invent statements for managing the wide variety of I/O devices that might be encountered. We simply assume that each one of them is represented by one or more primitive subroutine calls, written in machine language, but obeying the linkage conventions of LITTLE. All that is important is that external devices not generate interrupts or require polling too frequently. Therefore in some cases buffering actions may be associated with a routine appearing at the LITTLE level as a device-control primitive.

When necessary, trivial high frequency actions associated with certain special high frequency interrupts may be handled by short assembly code sequences hidden within the physical interrupt handler which is presupposed by the LITTLE interrupt conventions we have described. In this way, simple but frequent actions can be hidden, leaving only residual interrupts to be managed at the LITTLE level. Interrupts which have to be handled within fewer than a few dozen machine cycles probably call for assembly code; less urgent or less frequent interrupts can be handled using LITTLE.

## 4. Implementation, Robustness, Examples, Overlays.

A main charateristic of the scheme proposed is that it suppresses the concept of 'status package' that would appear at a semantic lower level, absorbing this into the more general notion of 'process data environment'. Putting this more crudely, when an interrupt occurs we store the 'status package' of the interrupted process with the other data constituting its environment. The justification for this lies in the fact that the private variables of a process, as well as subroutine return addresses and other normally 'implicit' data items, will ordinarily have to be treated as an extension of the status package, e.g. saved before the code blocks belonging to the process are re-entered for another purpose. Keeping the status package with this other data should avoid complications that would surface if these two classes of logically related data were treated seperately.

To execute an interrupt, we proceed as follows:

(a) First, we store all registers in a designated group of words within the data area associated with the interrupted process. This data environment is located by a "current data environment" pointer that is always maintained.

(b) Then we set up a new current data environment pointer.

(c) Finally, we transfer to a label associated with the process attached to the class of the current interrupt, restoring registers if necessary. Immediately prior to this jump, an ENABLE operation will be executed unless the jump has been set up by an ATTACH...DISABLE statement. If there is a hardware mask register, it will be located with the INTMASK value corresponding to the process being entered. However, this mask register reload will be suppressed if the preliminary ENABLE is suppressed; in such cases, the process being entered is responsible for loading its own mask register. The mask register will be reloaded automatically when an assignment is made to the private variable INTMASK.

The interrupt-handling scheme which has been described is *robust*, in that it gives representation to all the important interrupt-management actions which we expect conventional interrupt-management hardware to execute. Note in particular that likely hardware extensions, such as the introduction of new classes of external interrupts, can be mimicked by software. The following example will illustrate this point. Suppose that on some particular machine all interrupts trap to a fixed location (so that only one'interrupt class' exists); but that immediately after an interrupt has occured one can use an (assembly language) routine CAUSE(J) to set a quantity J equal to an integer representing the detailed interrupt cause Then, by using the following code as an innermost interrupt handler, we can make it appear at all 'outer' software levels that a more sophisticated hardware interrupt mechanism, capable of distinguishing between many different interrupt sources, is at work:

```
PROCESS HANDLE_ALL_INTERRUPTS;
/* MAIN INTERRUPT HANDLER, WHICH WE ASSUME TO BE ENTERED */
/* WITH ADDITIONAL INTERRUPTS DISABLED */

NAMESET ATTACHMENTS (ATTACHED, ATTINDICES, ENABLES)
SIZE ATTACHED (WS); /* CODES FOR PROCESSES (AND ARRAYS)
                              ATTACHED TO VARIOUS   INTERRUPT SOURCES */
SIZE ATTINDICES(WS); /* VALUES OF ATTACHED PROCESS INDICES */
SIZE ENABLES (1); /* FLAGS DISTINGUISTING 'ATTACH...DISABLE'CASES */
DIMS ATTACHED (NSOURCES), ATTINDICES(NSOURCES), ENABLES(NSOURCES);
/* 'NSOURCES' REPRESENTS THE NUMBER OF INTERRUPT SOURCES RECOGNISED */
SIZE J(WS); /* QUANTITY GIVING INTERRUPT CAUSE */
SIZE INDEX (WS); /* INDEX LOCATING ENVIRONMENT OF PROCESS
                                          TO BE RECALLED */
SIZE PROCANDARRAY(WS); /* AUXILIARY VARIABLE */

CAUSE(J); /* GET CAUSE OF INTERRUP */
PROCANDARRAY = ATTACHED(J); /* GET DESIGNATORS OF PROCESS
                                       AND ARRAY ATTACHED TO INTERRUPT */
/* NOW ESTABLISH ENABLE BIT CONDITION TO FOLLOW RECALL */
IF (ENABLES(J). EQ 1) THEN PRENABLE ELSE PREDISABLE;
INDEX = ATTINDICES(J);
GOBY PROCANDARRAY (CASE_1, CASE_2,....)

/* HERE MUST FOLLOW A LIST OF ALL THE PROCESS_NAME/ARRAY_NAME/
                                                    LABEL_NAME */
/* COMBINATIONS THAT OCCUR IN 'ATTACH' STATEMENTS */
/CASE_1/  RECALL PROC_1(ARRAY_1, INDEX);
/CASE_2/  RECALL PROC_2(ARRAY_2, INDEX) FROM LABEL_2; /* IF A LABEL IS REQUIRED */
...       (etc.)
END      /* HANDLE_ALL_INTERRUPTS */;
```

When the process shown above is used as a master interrupt handler, the following routine should be used to simulate 'ATTACH' operations.

Note that we assume this routine to be called using the sequence

```
DISABLE;
CALL SIMATTACH (SOURCENO, PROCANDARRAYCODE, INDEX,DISABLE_BIT);
```

```
SUBR SIMATTACH(SOURCEND, PROCANDARRAYCODE, INDEX, DISABLE_BIT);
/* THIS SIMULATES THE ACTION OF ONE OF THE STATEMENTS */
/* ATTACH N TO PROC(ARRAY, INDEX),  OR                */
/* ATTACH N TO PROC(ARRAY, INDEX) FROM LABEL,  OR     */
/* ATTACH N TO PROC(ARRAY, INDEX) DISABLE,     OR     */
/* ATTACH N TO PROC(ARRAY, INDEX) FROM LABEL DISABLE  */
/* THE PROC, THE ARRAY, AND THE LABEL ARE ALL TRANSMITTED */
/* IN CODED FORM AS THE PARAMETER 'PROCANDARRAYCODE' */
/* IN PRACTICE, IT WOULD BE CONVENIENT TO REPLACE THE */
/* ATTACH STATEMENT BY A SET OF MACROS                */
/*             ATTACH_PROC_ARRAY(N, INDEX),
/*             ATTACH_PROC_ARRAY_FROM_LABEL (N,INDEX), ETC. */
/*   TO GENERATE 'SIMATTACH'  CALLS                   */
SIZE SOURCENO(WS); /* PARAMETER: SOURCE-NUMBER        */
SIZE PROCANDARRAYCODE(WS); /* PARAMETER: PROCEDURE/ARRAY CODE */
SIZE INDEX(WS); /*PARAMETER: PROCEDURE ENVIRONMENT INDEX */
SIZE DISABLE_BIT(1); /* PARAMETER: POST_RECALL INTERRUPT SETTING */
ACCESS (ATTACHMENTS); /* SEE DEFINITION OF THIS NAMESET
                         IN 'HANDLE_ALL_INTERRUPTS' */
ENABLES(SOURCENO)=DISABLE_BIT; /* NEW'ENABLE_AFTER INTERRUPT'SETTING */
ATTACHED(SOURCENO) = PROCANDARRAYCODE; /* NEW PROCEDURE,
                                          ARRAY, AND LABEL(ONE SETTING)*/
ATTINDICES(SOURCENO) = INDEX; /* NEW DATA ENVIRONMENT INDEX */
IF (WAS_ENABLED) THEN ENABLE;
RETURN;
END;
```

Note that the routines shown above give a very good indication of what actual assembly-language code supporting the proposed ATTACH and recall statements would be like. Of course, at the assembly language level some additional actions

(such as register save and restore sequences) would have to be made explicit; also, some things could be done more simply, e.g. the combination ARRAY, INDEX can be condensed to a single address.

To the reader actively concerned with implementation, the dictions which we have proposed will suggest that individual items belonging to the data environment of a process are accessed by indexing to the item's address. This will be an adequate approach to implementation on large computers, and on minis with at least one (still better, several) index registers. However, on a mini without any index register, it may be important to keep the data environment of a process in a known physical relationship to its code. In such cases, the following variant technique might yield acceptable efficiency:

i) With each process type (as defined by a declaration (1) ) associate a data area in fixed physical relationship to its code. When control first passes to a process of this type (either via a RECALL or an interrupt), load, i.e. move, the data environment of the process into this fixed data area (using a fast loop.) Note in a word associated with the process type, the environment that is, in this sense, loaded.

ii) Each time control re-enters a process of given type, check whether it is currently loaded. If so, then control may enter it simply by a jump. If not, then the environment must be loaded (by two fast loops.) Note that in many of the situations which will be encountered in practice, only one instance of each process type will exist. Thus neither loading nor unloading will ever prove necessary.

iii) In using the technique just suggested, it will generally be mandatory to create several copies of routines shared between several processes, since each copy will have to stand in a fixed physical relationship to some designated data area.

iv) The type of 'jump-and-store' instruction commonly used at the hardware level in minis to support subroutine calls has the property, undesirable for our purpose, of scattering data items all of which belong to the data environment of a process. To avoid this, one may want to handle subroutine calls in a special way, e.g. by a jump and store to a location within a process environment block, followed immediately by an ordinary jump to the subroutine entry point.

## 5.Debugging, Polling, Higher and Lower Levels of Multiprocess Semantics.

One will often wish to exploit the machine independence of LITTLE by using a large machine with no convenient source of external interrupts to debug a system intended to execute on a mini with an elaborate interrupt system. A related possibility is that of using a computer with no interrupt system to control external devices which require fairly frequent polling. Note that the advantage of an interrupt-driven system over a programmed system which polls is its greater stability: in an interrupt-driven system, an external device can be sure of receiving service within some relatively short time after a service request is posted; by contrast, a programmer constrained to use only polling can find it very difficult to ensure that his programs contain no loops which run long enough to miss a poll deadline. The following easy technique for mimicking clock interrupts on a machine with no actual interrupt source is therefore of interest.

a) In compiling LITTLE source text, compile each backwards branch, i.e. each jump which can have a target address smaller than its physical address, and also each subroutine call, as follows:

ai) decrement a global counter EXEC_COUNT by an amount equal to a (compile-time) estimate of the number of instructions executed before the last proceeding point of backward branch. If this quentity remains positive, take the branch (or execute the call). If it is negative, call a (parameterless) routine to simulate an interrupt.

aii) This interrupt simulation routine will restore EXEC_COUNT to some reasonable positive value,assign an interrupt type (perhaps using a random number generator to do so), call additional subprocedures to mimic any changes in core locations or accessible registers which may be associated with an interrupt, and then RECALL whatever process has been ATTACHED to an interrupt of the class which it has decided to simulate. Note that in such a software interrupt simulation, the ENABLE flag is represented by a 1-bit global quantity, ENABLE and DISABLE being compiled merely as instructions which modify this quantity. Note also that this technique of interrupt simulation will generally slow execution only moderately.

To debug interrupt-driven families of processes, it will be convenient to supplement the presently planned LITTLE execution-time trace facilities with a facility which either builds up an interrupt history file and prints it out on demand, or prints such a file out piecemeal. This file should contain at least the following information: interrupt class, process to which control is transfered by interrupt, manner of re-entry (restart at *label* or process resume, state of ENABLE bit), process interrupted, and execution location at moment of interrupt. Also: interrupts ignored by virtue of ENABLE bit or MASK setting; process in control when interrupt ignored, with ENABLE bit, mask setting, and instruction location.

LITTLE, with the extension we have described, can also be used in a related but somewhat more sophisticated way to allow the development, within a single computer, of programs ultimately intended to run in several physically seperate computers, eventually as part of a system which can involve various peripheral devices and communications channels.

We are suggesting that the whole system can conveniently be simulated during development. Note that here the development of an entire system on a single machine can have really big practical advantages, not the least of which is the elimination of the severe logistic problems that associate themselves with work on pieces of hardware located remotely from each other. For the development of multicomputer and communications software, we propose the following approach:

i) A total family of processes, which in sum represent all parts of the proposed system, will be written in LITTLE. These routines should be grouped into several process subfamilies, each such subfamily representing the software for one physical device. Note that even passive devices such as disc memories, etc. are to be simulated by appropriate subfamilies of processes.

ii) A single global EXEC_COUNT, serving for interrupt simulation in the manner described earlier in the present section,will be maintained. When EXEC_COUNT becomes zero, interrupt action will be taken, and a central process, which we shall call SUBFAMILY_SCHEDULER, will begin to execute.

iii) Within each (device- or) computer-representing subfamily of processes, there will exist one, designated as the (interrupt) entry process for the subfamily. Within this process a private quantity called INTERRUPT_STATE, whose bits correspond to the interrupt sources physically recognised by the computer represented by the subfamily, will be declared. SUBFAMILY_SCHEDULER will always pass control back into a subfamily of processes by RECALLING the interrupt entry process of the subfamily, which will examine its INTERRUPT_STATE, will assume that any '1' bits in this quantity represent external interrupts just received and still pending, and will either RECALL a process still noted as current within the subfamily, or (if an interrupt was found pending) simulate the RECALL action ATTACHED to this interrupt.

vi) SUBFAMILY_SCHEDULER merely passes control, round-robin fashion, among all the subfamily interrupt entry processes of a total system.

v) To transmit an interrupt to a device, a process only needs to set an appropriate bit of the INTERRUPT_STATE quantity belonging to the entry process of the subfamily which represents the device.

vi) The central SUBFAMILY_SCHEDULER maintains a global clock. clock is readable by all processes supposed to have access to a real-time clock; it is advanced by an appropriate, slightly randomised amount each time SUBFAMILY_SCHEDULER gains control.

vii) It is appropriate to provide some standardised means whereby a subfamily entry process can signal the central SUBFAMILY_SCHEDULER that it has no work to do and is willing to give up the remainder of its turn to execute. This can be done by reserving a global flag location IDLE_FLAG, always set to 1 by SUBFAMILY_SCHEDULER. Then if an entry process sets IDLE_FLAG to zero and RECALLs SUBFAMILY_SCHEDULER, control will pass into the next process-subfamily at once.

viii) Each interrupt ENABLE bit is of course private to some particular device and should be represented by a quantity private to the entry process of the subfamily representing the device. Call this quantity ENABLEBIT_J (where J varies, and designates some particular device). Then a LITTLE macro can be used to redefine the ENABLE and DISABLE statement as

+ *ENABLE = ENABLEBIT_J = 1**; *DISABLE = ENABLEBIT_J = 0**

within the code representing the activity of the J-th device.

To realise the possibilities described in the present section, only one feature needs to be added to the group of LITTLE interrupt handling statements which have already been described. This is a declaration

INTERRUPT TO *processname (array, index)*

which, prefixed to the text of a complete family of processes, modifies the compilation of all backward branches in the manner explained in the proceeding pages. Note that this declaration tells the system which process is to be RECALLED when EXEC_COUNT goes to zero. Of course, branches in the routine *processname* are exempt from modification.

There exists (see SETL Newsletter 110) a literature on multiprocess semantics in which process-handling facilities more advanced than those discussed in the present newsletter are described. It might be asked if these ought not be provided in LITTLE. On the other hand, it might also be asked if it would not be better to provide only more primitive interrupt handling facilities, thereby avoiding some of the overhead implied by the scheme which we have suggested. At a more primitive semantic level, the explicit notion of process would vanish, and an explicit 'status package' concept would appear. The RECALL statement we have proposed would then be replaced with a statement

RELOAD *(array, index)*

which reloaded a status package stored at a specified address in an array. The ATTACH statement could be replaced by two statements. The first, which might be

DUMP *(array, index)*,

could store a status package (including pre-interrupt instruction location, which could probably be fetched from core) in a specified location. The second could control the location to which transfer was made when an interrupt with given source was executed. The statements DISABLE and ENABLE would remain available.

The LITTLE multiprocessing primitives which we have described include facilities for process initiation, resumption, termination (by re-using the area in which a data environment is stored) and suspension (e.g., by detaching one process from a given interrupt source and attaching another.) The most important additional concepts which would appear at a higher semantic level are the notions of process priority and priority scheduling, event variables and 'non-busy' event variable monitoring, and process coordination by the use of semaphores. We omit to provide these notions because to provide them in full generality would probably imply the use of a standard system space-management routine, which we wish to avoid as being inconsonant with the 'low level' semantic spirit of LITTLE. In general, these relatively advanced multiprocess semantic notions relate to environments in which, in addition to the processes directly attached to interrupt sources, there exists a pool of other processes attempting to execute. These processes will generally fall into several classes, represented at the implementation level by several lists. One group of processes will be *active*, i.e., asking for immediate execution. These processes may further be subdivided according to their execution priorities. Other processes may be in *monitoring* state, i.e., momentarily dormant but attached to lists associated with one or more 'evant' variables; when new values are assigned to an event variable, all the processes attached to it are detached, and, provided that they are not monitoring any other variables, transferred to the active list. Semaphores are special event variables, always accessed while all interrupts are DISABLED. It is not hard to represent semantic facilities of this type using the proposed LITTLE interrupt-handling features. Note that the details of an implementation of these facilities will vary widely, depending on such system-specific facts as whether a fixed number, a varying but small number or a varying and possibly large number of processes and/or priority levels are possible, etc.