A PL/I program is a set of one or more procedures. The execution of these procedures constitutes one or more *tasks,* each associated with a different task variable. A task is dynamic; it exists only while the procedure is being executed. The distinction between the procedure and its execution is essential to the discussion of multitasking. A procedure can be executed concurrently in different tasks. A PL/I procedure cannot invoke a COBOL routine as a task.

Only one task of a PL/I program can have active COBOL routine at any one time. If a PL/I program has more than one task active at the same time, and if one of these tasks has invoked a COBOL routine, you must ensure that other tasks wait until control has returned to the PL/I program before another non-PL/I routine is invoked.

In this discussion, the task-reference used to associate a task variable with a task is used to identify the task. For example, after

CALL INPUT TASK(A2);

we can write ".....the task A2....."

When the multitasking facilities are not used, the execution of a program comprising of one or more procedures constitutes a single task, with a single flow of control. When a procedure invokes another procedure, control is passed to the invoked procedure, and execution of the invoking procedure is suspended until the invoked procedure passes control back to it. This execution with a single flow of control is called *synchronous*. When you are concerned only with synchronous operations, the distinction between program and task is unimportant.

When multitasking, the invoking procedure does not relinquish control to the invoked procedure. Instead, an additional flow of control is established so that both procedures are executed concurrently. This process is known as *attaching* a task. The attached task is a *subtask* of the attaching task. Any task can attach one or more subtasks. The task that has control at the outset is called the *major task*. The execution of such concurrent procedures is called *asynchronous*.

When several procedures are executed asynchronously, it might be necessary for the system to select its next action from a number of different tasks. Processor operations can be carried out in one task while an input/output operation is carried out concurrently in another.

It might be that one task is to run independently of other concurrent tasks for some time, but then become dependent on some other task. Provision has been made for one task to await the completion of an operation of another task before proceeding. This process is known as *task synchronization.* Information about the the state of an operation can be held in an event-variable. Execution of a WAIT statement causes the task to wait for completion of the operation associated with the event-variable. You can set the value of the event-variable explicitly, or you can use the EVENT option when attaching tasks, in which case the value of the event-variable is set as a result of of the operation concerned.

In general, the rules associated with the synchronous invocation of procedures apply equally to the asynchronous attachment of tasks. Under LE, each task operates within its own enclave, and

the rules for operation within an LE enclave including the establishment of ON-units are no different under multitasking to what they would be in a non-multitasking environment. However, asynchronous operation introduces some special considerations, such as the fact that a number of different tasks can independently refer to one variable. This necessitates some extra rules discussed later.

Multitasking can allow you to use the processor and input/output channels more efficiently by reducing the amount of waiting time. It does not necessarily follow that an asynchronous program will be more efficient than an equivalent synchronous program (although the latter might be easier to write). Efficiency depends on the amount of overlap possible between operations with varying amounts of input/output. If overlap is slight, multitasking will be the less efficient method, because of the increased system overheads. The compiler and run-time utilize the operating system tasking facilities. If one procedure is dependent on the values of variables in another procedure that is executing asynchronously, the use of variables in separate tasks must be synchronized by the programmer.

Creation of Tasks

You can specify the creation of an individual task by using one or more of the multitasking options of a CALL statement. The PL/1 “main” must be compiled with the “task” option for the multitasking environment to be established. After a procedure is activated by execution of such a CALL statement, all blocks synchronously activated as a result of its execution become part of the created task. All tasks attached as a result of its execution become subtasks of the created task. The created task itself is a sub task of the task executing the CALL statement. All programmer-created tasks are subtasks of the major task.

It is necessary to specify the REENTRANT option for procedures that are attached as more than one to be executed concurrently. When REENTRANT is specified, the compiler generates code that is reentrant as far as machine instructions and compiler-created storage are concerned. However, you must ensure that the logic of your PL/I source code is such that the procedure remains reentrant. In particular, you must not overwrite static storage.

A task cannot be attached by a procedure entered as a result of a function reference in a PUT statement for the file SYSPRINT.

CALL Statement

The CALL statement for asynchronous operation has the same syntax as that described for a synchronous operation, with the addition of one or more of the multitasking options: TASK, EVENT, or PRIORITY.

TASK and PRIORITY option syntax:

```
>>---------TASK (task reference)----------><
>>---------PRIORITY (expression)---------><
```

Expression is evaluated to a real fixed-point binary value, m, of precision (15,0).

The TASK and PRIORITY options are intended to be used to identify a task by name, and control its relative priority. Under VSE, tasks operate under a fixed priority when they are attached. It is therefore not possible to increase or lower the priority of tasks in relation to other tasks. These options are supported for compatibility reasons only, and currently have no effect.

The event-variable is associated with the completion of the task created by the CALL statement. Another task can then wait for completion of this task by specifying the event-variable in a WAIT statement. The syntax for the EVENT option is:

EVENT option

>>---------EVENT (event-reference)------><

When the CALL statement is executed, the completion value of the event-variable is set to '0'B (for *incomplete*) and the status value to zero (for *normal status*). The sequence of these two assignments is un-interruptible, and is completed before control passes to the named entry point.

On termination of the created task, the completion value is set to '1'B. In the case of abnormal termination, the status value is set to 1 (if it is still zero). The sequence of the two assignments to the event-variable values is un-interruptible.

Examples:

Each of the statements below represents a multitasking CALL, with PROCA being attached as a sub task.

1. CALL PROCA TASK(T1);

2. CALL PROCA TASK(T2) EVENT(ET2);

3. CALL PROCA TASK(T3) EVENT(ET3) PRIORITY(-2);

4. CALL PROCA PRIORITY(1);

Coordination and Synchronization of Tasks

The rules for scope of names apply to blocks in the same way whether or not they are invoked as, or by, subtasks. Thus, data and files can be shared between asynchronously executing tasks. As a result a high degree of cooperation is possible between tasks, but this necessitates some coordination. Additional rules are introduced to deal with sharing of data and files between tasks, and the WAIT statement is provided to allow task synchronization.

An example of task synchronization is:

```
P1:  PROCEDURE;
     CALL P2 EVENT(EP2);
     CALL P3 EVENT(EP3);
     WAIT (EP2);
     WAIT (EP3);
END P1;
```

In the above example, the task executing P1 proceeds until it reaches the first WAIT statement; it then awaits the completion of the task executing P2, and then completion of the task executing P3, before continuing.

Sharing Data Between Tasks

It is the programmer's responsibility to ensure that two references to the same variable cannot be in effect at one time if either reference changes the value of the variable. You can do so by including an appropriate WAIT statement at a suitable point in your source program to force synchronization of the tasks involved. Subject to this qualification, and the normal rules of scope, the following additional rules apply:

1. Static variables can be referred to in any block in which they are known, regardless of task Boundaries.

2. Automatic and defined variables can be referred to in any block in which they are known, and which is a dynamic descendant of the block that declares them, regardless of task boundaries. The declaration that is known in the sub task is the declaration that is current in the attaching task at the point of attachment.

3. Controlled variables can be referred to in any task in which they are known. However, not all generations are known in each task. When a task is initiated, only the latest generation, if any, of each controlled variable known in the attaching task is known to the attached task. Both tasks can refer to the latest generation. Subsequent generations in the attached task are known only within the attached task; subsequent generations within the attaching task are known only within the attaching task. A task can free only its own allocations; an attempt to free allocations made by another task will have no effect. No generations of the controlled variable need exist at the time of attaching. A task must not free a controlled generation shared with a sub task if the sub task will later refer to the generation. When a task is terminated, all generations of controlled storage made within that task are freed.

4. A task can allocate and free based variables in any area to which it can refer. A task can free an allocation of a based variable not allocated in an area only if the based variable was allocated by that task. Unless contained in an area allocated in another task, all based variables allocations (including areas) are freed on termination of the task in which they were allocated.

5. Any generation of a variable of any storage class can be referred to in any task by means of an appropriate based variable reference. You must ensure that the required variable is

allocated at the time of reference.

Sharing Files Between Tasks

A file is *shared* between a task and its sub task if the file is open at the time the sub task is attached. If a sub task shares a file with its attaching task, the sub task must not attempt to close the file. A sub task must not access a shared file while its attaching task is closing the file. The sub task can re-open a file closed by the attaching task, but it will not be shared.

If a file is known to a task and its sub task and the file was not open when the sub task was attached, the file is not shared. The effect is as if the task and its sub task were separate tasks to which the file was known. That is, each task can separately open, access, and close the file. This type of open is guaranteed only for files that are DIRECT in both tasks. If one task opens a file, no other task can provide the corresponding close operation.

With respect to file sharing and input/output operations being performed by concurrent tasks, it is the application's responsibility to ensure that I/O operations on the same file are synchronized between tasks. There is no locking of records, or automatic synchronization of I/O, done by the multitasking control program.

Note: SYSPRINT will not be used for diagnostic messages in subtasks unless it was already open when the sub task was attached.

Testing and setting Event Variables

The COMPLETION built-in function returns the current completion value of the event-reference. This value is '0'B if the event is incomplete, or '1'B is the event is complete.

The STATUS built-in in function returns the current status of the event-reference. This value is non-zero if the event-variable is abnormal, or zero if it is normal.

The COMPLETION and STATUS pseudovariables can be used to set the two values of an event-variable. Alternatively, it is possible to assign the values of one event-variable to another in an assignment statement. By these means, you can mark the stages of a task; and, by using a WAIT statement in one task, you can synchronize any stage of one task with any other.

You should not attempt to assign a completion value to an event-variable currently associated with an active task or with an input/output event. The event can also be waited for in any other task. However, in this case the task waits until the event is set to complete by a WAIT statement in the initiating task.

COMPLETION - Event

COMPLETION returns a bit string of length 1, specifying the completion value of x; the event can be active or inactive. If the completion value of the event is incomplete, '0'B is returned; if complete, '1'B is returned. The syntax for COMPLETION is:

>>>>>>----------COMPLETION(x)----------<<

X event reference.

COMPLETION Pseudovariable

The pseudovariable sets the completion of value x. No interrupt can occur during the assignment to the pseudovariable. For the compiler, the COMPLETION pseudovariable cannot be used as the the control variable in a do-specification. The syntax for COMPLETION pseudovariable is:

>>>>>>----------COMPLETION(x)----------<<

X event reference. X must be inactive.

STATUS - Event

STATUS returns a FIXED BINARY (15,0) value specifying the status value of an event reference x. If the event-variable is normal, zero is returned; if abnormal, nonzero is returned.

STATUS (x)

X event-reference. If x is omitted, the event-variable associated with the current task is the default.

STATUS Pseudovariable

The pseudovariable sets the status value of an event-reference x. The variable can be active or inactive and complete or incomplete. The value assigned to the pseudovariable is converted to FIXED BINARY (15,0), if necessary. No interrupt can occur during each assignment tp the pseudovariable. The systax for STATUS is:

STATUS (x)

X event-reference. If x is omitted, the event-variable associated with the current task is the default.

PRIORITY(x) - is supported for compatibility reasons only. It has no effect, so is not discussed here.

Interlocking Tasks

There is a possibility that two tasks could interlock and enter a permanent wait state. You must ensure that this cannot happen in a program. For example:

Task T1	**Task T2 (Event E2)**
.	COMPLETION(EV)='0'B;
.	.
.	.
WAIT (E2);	.
.	WAIT(EV);
.	.
.	.
COMPLETION(EV)='1'B;	.
.	.
.	RETURN;

Task T1 waits for the completion of task T2, and task T2 waits for task T1 to execute the assignment to the COMPLETION pseudo variable, to set the event variable EV to complete.

Under the compiler, the program waits until cancelled by the operating system of the operator.

Termination of Tasks

A task terminates when one of the following occurs:

1. Control for the task reaches a RETURN or END statement for the initial procedure of the task.

2. The block in which the task was attached terminates (either normally or abnormally)

3. The attaching task itself terminates

4. Implicit action for the ERROR condition or the action on normal return from an ERROR ON-unit is carried out.

Termination is normal only if the first item of the above list applies. In all other cases, termination is abnormal.

To avoid unintentional abnormal termination of a sub task, an attaching task should always wait for completion of the sub task it attached, before the attaching task itself is allowed to terminate.

When a task is terminated, the following actions are performed:

1. All files that were opened during the task and are still open are closed.

2. All allocations of controlled variables made by the task are freed, except those it has allocated within an area allocated by another task (these are freed when the area is freed).

3. All active blocks (including all active subtasks) in the task are terminated.

4. If the EVENT option was specified when the task was attached, the completion value of the associated eve-variable is set to ‘1’B.

If a task is terminated while it is assigning a value to a variable, the value of the variable is undefined after termination. Similarly, if a task is terminated while it is creating or updating an OUTPUT or UPDATE file, the effect on the associated data set is undefined after termination. It is your responsibility to ensure that assignment and transmission are properly completed before termination of the task performing these operations.

VSE/LE specific concepts:

Please refer to LE/VSE Programming Guide for information on PL/I Multitasking run-time concepts.