OVLBI-ES MEMO NO. 24

IBM PC VLBA recording terminal control software working under MS-DOS.

Alexander Novikov and Vitaly Promislov
February 1992
Astro Space Center

Profsoyuznaya st., 84/32
Moscow, 117810
Russia, USSR

FAX: 095-310-7023
PHONE: 095-333-21-89
E-MAIL: ikimail@esoc1.bitnet

INTRODUCTION.

This memo is concerned with description of the original NRAO VLBA station control software to MS DOS working on IBM PC. This work is a part of preparation of the space OBLVI Radioastron project. In practice it was initiated to carry out the satellite decoder/VLBA recording terminal interface test and make some VLBI ground test observations at Ussuriisk station. For this purpose we need only Data Acquisition Rack (DAS) and Recorder Rack since as a first step now VLBA PC software controls these two racks. However developing the VLBA PC software for this particular purpose we overcame all principal problems such as: multitasking inside the MS-DOS, screen package adaptation and system's DOS/VXWorks differences. It can guarantee that the increasing of devices on MCB doesn't occur any DOS compatibility problems. VLBA PC software provides all features of the original NRAO VLBA station control software including the loading the observation parameters from standard VLBA schedule file. In addition likely INTERFEROMETRICS version the VLBA PC software produces MKIII log-file.
The main instrument to develop VLBA PC software under MS-DOS is program

language C++ and in following section it is presented more detail consideration the software problems that were appeared during this work.

I. MULTITASKING INSIDE MS-DOS.

At first to discard the question why some of PC multitasking operational systems (OS2, Xenix, Venix etc) didn't use it should emphasize that the next reasons pushed to use MS-DOS:
- in case using "non-MS-DOS" operational systems it was needed to spent additional money and time to buy and learn it;
- more simple task than controlling all radio telescope permitted to expect a success for less powerful MS-DOS software and IBM PC hardware than VXWorks and Motorola 68020 CPU;
- first steps on MS-DOS VLBA PC software implementation way were made by several teams in INTERFERROMETRICS company, Jodrell Bank, MPIfR Bonn and Socorro NRAO AOC.

All above mentioned teams except INTERFERROMETRICS were working with a single screen menu approach when you can't watch more than one a screen at the same time. The multiscreen approach was selected as main goal in developing VLBA PC software under MS-DOS in order do not intersect with another teams.

I.1. Why C++ ?

The original VLBA station control software was written and is being written now on Unix C language by Barry Clark's team in NRAO AOC Socorro. It consists of about more a half million lines of C code. The main problem of adaptation the existing software is to control the change in the line in the one module wouldn't occur any crashing in any far modules. The second problem concerns with the developing the original VLBA software is not completed else and in addition to own MS-DOS adaptation changes it will be necessary insert the last B.Clark's team changes to completed VLBA PC software in future.

This situation forced to keep line to line compatibility and design the multitasking kernel as a separate independent part that have to work in VXWorks style formally. In developing the multitasking kernel we followed to the tendency to define all of the access functions in single module as an attempt to achieve data encapsulation. Access from other modules to object of type Task in our case should be through one of the access functions (initTask(), currentTask(), killTask(), taskStatus(), unsuspendTask(), startScheduler(), stopScheduler(), killSelf(), setReturnValue(), getReturnValue(), etc). Concentration of access functions narrows the area in which the programmer must look if a problem occurs in the handling of Task objects. Once the program is completed, if the structure of a Task changes, the programmer will know the location of the functions that will have to be changed and retested. However, since C doesn't limit access to the internal of an object of type Task the programmer cannot be certain that other modules will not be affected as well. While it is possible to "teach" any of existing functions or operators how to operate on this new type. Each new type structure must have a different set of access functions with unique names. In C++ the access functions can be defined as part of the structure it self. These member functions become a part of the structure like member data elements. Not only is the association between the structure and its access functions more explicit, but parts of the structure can be defined to be accessible only to the member functions. Access to the member of the structure is strictly controlled in new type of structure called class in C++. Use of the access functions is no longer voluntary.

1.2. Task Class.

It is naturally to assume for multitasking developing that object task is declared as Task Class in C++ terms. Each object of class Task is an independent execution thread capable of receiving control CPU. The task comes into existence when the object is created and ceases to exist when

the object goes out of scope. Task could be initialized in C by defining a function TaskInit(), and then invoking the function after the object is declared but before it is used. A similar approach was used in C++ since initialization can be a member function with access to the private members.
To facilitate this process, C++ allows the programmer to define a special method known as constructor. A constructor is a special member function that is automatically invoked whenever an object of that type is created.
C++ also allows the programmer to define a member function that is called by name distructor when an object is destroyed. So the constructor and distructor initiates and terminate tasks. The constructor automatically allocates the memory that the task will require. It then adds the object to the scheduling linked list and prepares the object to run when the rescheduler reaches it. The distructor removes the task from the scheduler list and returns any dynamically allocated memory to the heap. The Task Class is a sub class of linked list. The task objects reside in a circular linked list. The currently active task is the object pointed at by current Task-ptr. Rescheduling always begins with the next object in the list after *current Task-ptr. The following status bits of object Task are declared in Task Class: READY, SUSPENDED, DEAD, RUNNING.
So you can see that the multitasking kernel development has a great support from such special C++ features as: class, constructor, destructor, class inheritance (see below) and more object oriented style of syntax. We made some overview of modern C++-books and found very nice example of Task Class implementation in book "Hands on Turbo C++" by Stefen R.Davis. We used code from this book as a base for our multitasking implementation.

I.3. The main restriction of the multitasking inside the DOS.

The biggest limitation of our multitasking approach is that implements

voluntary or non preemptive scheduling. That is once a task gains control of the CPU it retains that control until the rescheduler function is called.
DOS was written as a single-tasking operational system and is inherently non - reentrant. When an application task is in the middle of a DOS system call, another task cannot start a DOS call without crashing the first task. It is the main reason that doesn't develop complex preemptive task scheduler based on DOS with great work at precautions to prevent one task from making a DOS call when another DOS call is being executed. In addition a number of problems such as data collision would be required efforts compared to develop new multitasking operational system.
So one of the key of adaptation the single task written originally for multitasking VXWorks OS to non preemptive multitasking application inside the MS-DOS is to determine point for task switching. The task switching mechanism is naturally realized by setjmp() and longjmp() functions from Turbo C++ run time library. To save task context at task creation, a small area of memory is reserved and assigned for use as the task stack. The address of the stack is retained in the task object. Automatic and register variables are not shared. Task scheduling path is shown in Fig.1.

I.4. Task Initialization.

When any task's object is being created it will be included in linked-list.
EXAMPLE 1:

```
        Task task(fun,0x200,"FUN_NAME"); // create object type of task
where
                              fun is C function type of void fun(void)

                               0x200 is size of the new stack
```

"FUN_NAME" is task's name (uniq)

EXAMPLE 2:

```
Task task(fun_param,argc,argv,0x200,"FUN_NAME");
                                   where
                              fun is C function with parameters
                              type of void fun(int argc,char *argv[])

                               0x200 is size of the new stack

                               "FUN_NAME" is task's name (uniq)
```

Task's function should have the general structure shown in Fig.2.

I.5. The main two rules to adopt the NRAO C-code.

When we compiled the original NRAO C-code by Borland Turbo C++ v.2.0 compiler we made some obvious changes concerned with more strong syntax of C++ and difference between 32 bit Motorola integer word and 16 bit Intel word. Short overview the rules of these changes presents below.

I.5.1. Function definition.

The "C++" programming languages has more strong restriction in function definition than "C" and we have to make corresponding changes in VLBA software.

EXAMPLE: We have C function definition as:

```
    void fun( intarg,chararg)
    int intarg;
    char chararg;
    {
        /* some code here */
    ..................
    }
```

In the "C" language this function can have the following prototype:

```
  void fun(); /* OK in C */
```
in the C++ programing language it will produce a compiler error.
and we replaced such lines as:
```
  void fun(int,char); // OK in C and C++
```

I.5.2. Length integer variables.
Some changes are connected with different length of word on VME and IBM PC computer.

EXAMPLE:

Let's consider the following description:
```
  int int_var;
  int_var <<=20;
```
If you run it on the VME computer the result will always be OK, because the type int has 32 bit length. But the int has only 16 bit length on the IBM PC computer and it causes known problems. We have to rewrite such description as:
```
  long int_var; /* 32 bit length on the IBM PC */
  int_var <<=20; /* OK as on the VME as on the IBM PC */
```

II. NRAO Screen package implementation for MS-DOS version.

Every screen's program consists from two task running as parent task with its children task. The parent task is responsible for operator random key-input and sending MCB command/monitor messages to corresponding equipment. The children task updates the device screen information. One task pass control to another by calling rescheduler function as shown in Fig.3.
For multiscreen implementation all the single screen task are contained in circular linked-list of rescheduler. Every time the rescheduler (task's switcher) loads the next task from linked-list and passes it the control.

III. Implementing the VXWorks task's variables

VXWorks has special type of the variables - the task's variables. It means that a value of the each such variable depends from which task now is running. VXWorks switches the content of this variable every time when it pass control to the new task.

We provide the full emulation of this feature without noticeable difference for user. The special class task's variables provides it. Every task may has more than one task's variable at the same time. The task's variable creation is connected with the initialization of special taskVar class. Let us examine the task's variable implementation in the screen package. There is the common pointer pmcbmsg to the structure mcbmsg. Each task contained in scheduler list can initialize itself the own structure.

```
        extern struct mcbmsg *pmcbmsg; /* pointer definition */
                /* some code here */
                ....................
  struct mcbmsg mcbmsg;
                                             /* structure belongs some task */
  pmcbmsg=&mcbmsg;
                                             /* initialize extern pointer */
  taskVar((void*)(&pmcbmsg),(void*)(pmcbmsg));
                                             /* create task's variable */
```

IV. Formatter/PC time tagging.

We changed the main time tagging logic when control computer is a master for Formatter time setting. Previously operator should set the Formatter time by Formatter screen seeing on station timer. After that Formatter is became the master timer for PC DOS time. To tag the PC DOS time to Formatter time we designed the special single screen program STIME, which catch the Formatter 1 sec transition asking Formatter time through the MCB bus. You should run STIME program before VLBA PC software running called PCASTR. Each formatter setting during observation the modified NEWD module check the alignment between Formatter and DOS 1 sec tick and in negative case NEWD will be realign the DOS time Formatter time.

V. Shorten Log-file creation in MS-DOS version.

Our PC software support the binary log file creation. But the PC interpretation package has not been completed to emulate the same functions likely in B.Grechky package. We inserted to NEWD module the generation special MKIII-log file by copying F.Ghigo's C-code from Green Bank which is used for NRL correlator.

VI. Conclusion.

We try to demonstrate how we designed our PC VLBA software and hope that our approach can be useful not only for Radioastron project and may be for some VLBI station where will want to replace the expensive Motorola control computer to common PC. In addition we thing that our software may be powerful instrument for VLBA terminal engineering staff.

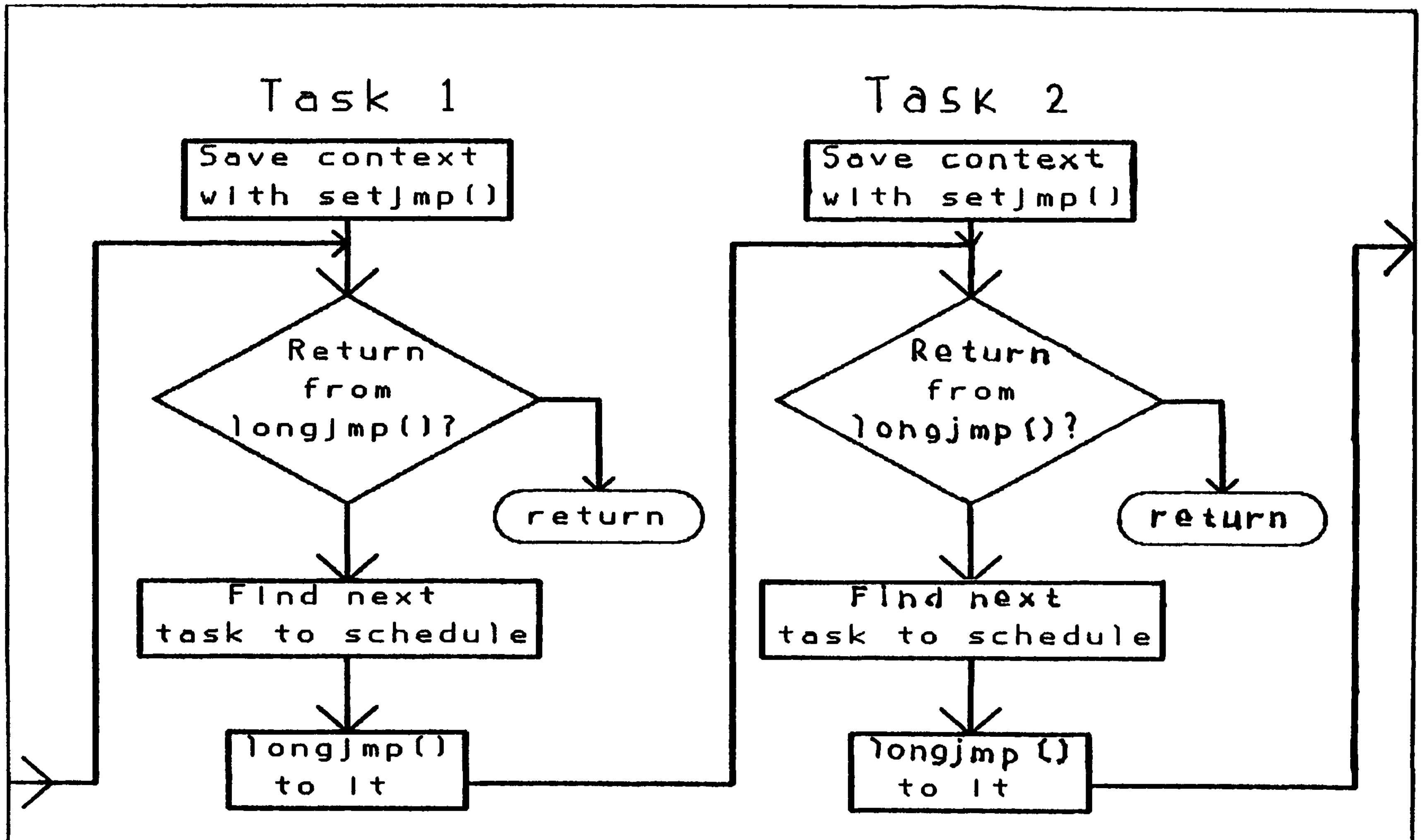
Task 1
Save context with setjmp()
Return from longjmp()?
return
Find next task to schedule
longjmp() to it
Task 2
Save context with setjmp()
Return from longjmp()?
return
Find next task to schedule
longjmp() to it

Fig.1. Task Switching with scheduler()

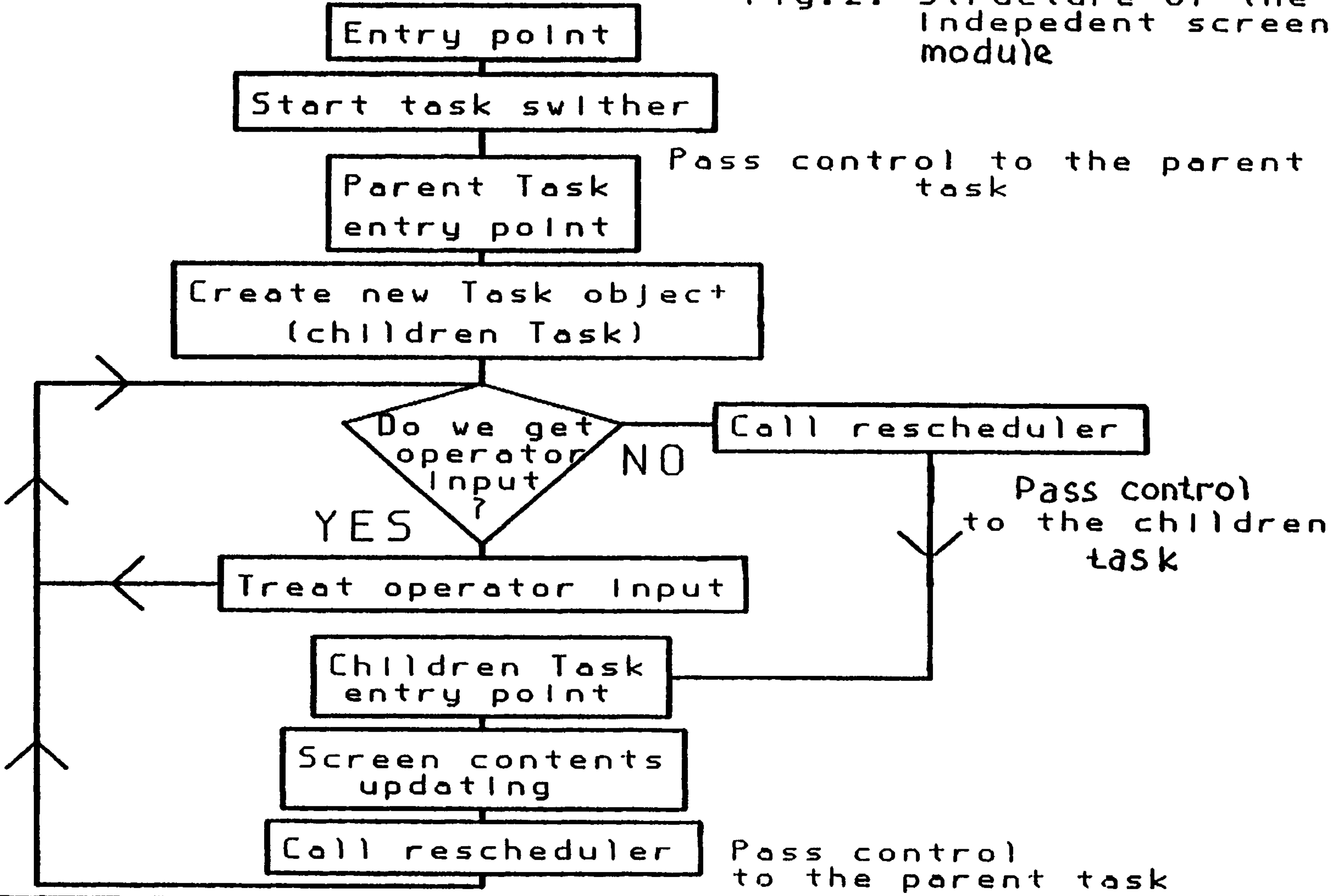
Entry point
Start task swither
Pass control to the parent task
Parent Task entry point
Create new Task object (children Task)
Do we get operator input ?
NO
Call rescheduler
Pass control to the children task
YES
Treat operator input
Children Task entry point
Screen contents updating
Call rescheduler
Pass control to the parent task

Fig.2. Structure of the Indepedent screen module

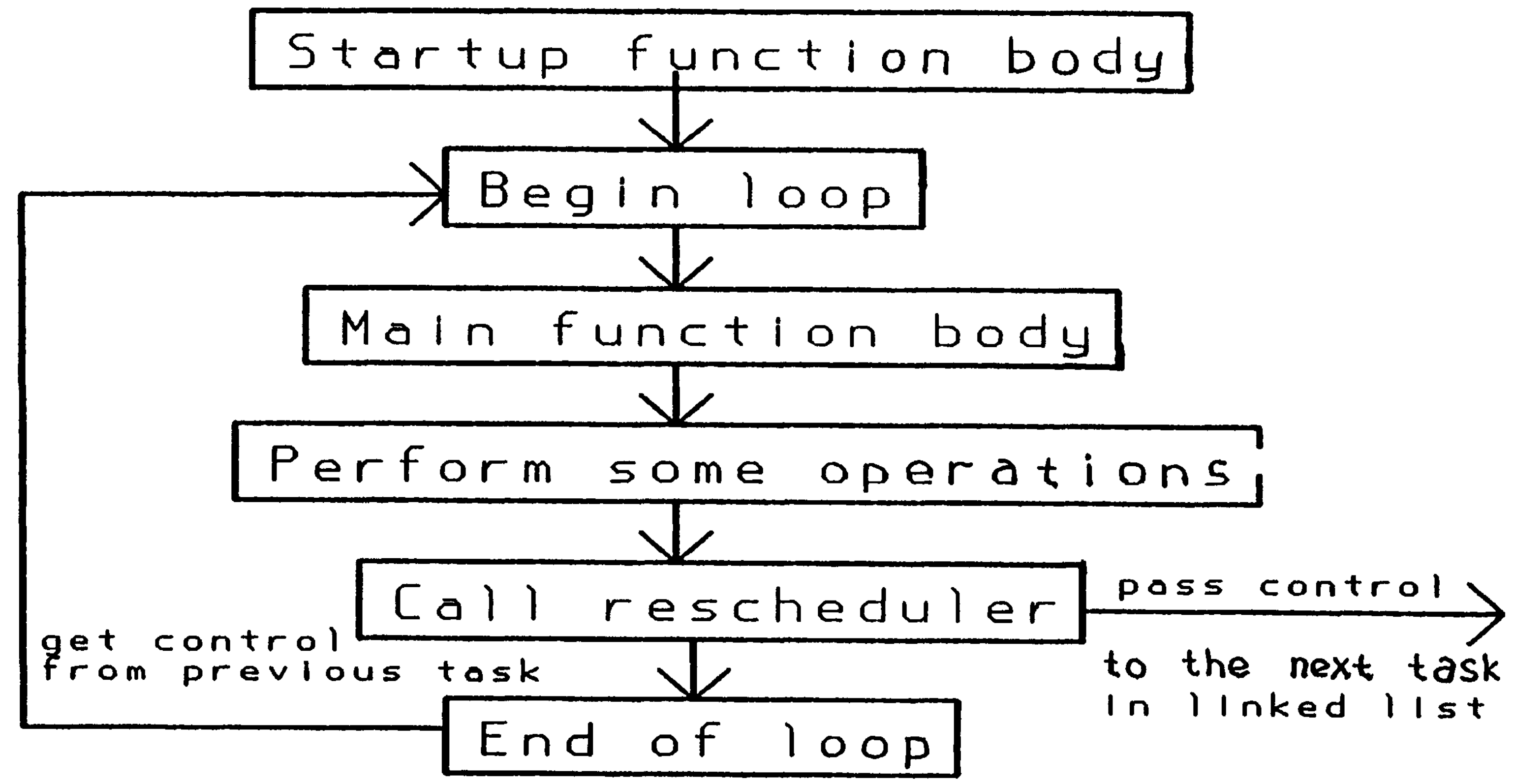

Fig.3. The general structure of the task body