

WORKSHOP ON ALTERNATIVES IN THE DESIGN AND IMPLEMENTATION OF A COMMON PROGRAMMING LANGUAGE FOR EMBEDDED COMPUTER APPLICATIONS

TITLE: Parallel Processing and Modular Software Construction

AUTHOR: K. Jackson, Mindef, Malvern, UK.

INTRODUCTION

The one common denominator in the software for all real time computer systems is that it consists of a set of co-operating parallel processes. This is true whether the parallelism is achieved by having many hardware processors or by multiplexing a single processor among many competing processes (pseudo-parallelism); it is also independent of the way the pseudo parallelism is achieved eg interrupts, scheduler etc. It is therefore regrettable that in the TINMAN document this topic receives only scant treatment. In fact only two sub sections (approx. $2\frac{1}{2}$ per cent) of the document are devoted to this topic.

This paper presents an opposite point of view. Starting from the parallel processing common denominator, the advantages of bringing this out into the open at as early as possible a stage in the software design are considered. The consequences of this approach are that the concept of 'the program' disappears and is replaced by a network of parallel processes and data areas. This enables a different and more flexible approach to software construction to be adopted. The language implications of this approach are quite minor but the advantages in terms of software quality and true modularity are very significant as also are the consequences on ease of management.

PARALLEL PROCESSING

Parallel processing implies that the many processes are running in parallel and that they are as far as possible unrelated in time. In other words they run asynchronously. However, since in embedded computer applications all the processes do co-operate to perform a single overall task, they must, of necessity, intercommunicate.

There are two facets to this intercommunication. Firstly two communicating processes must be able to access the data area(s) (although not necessarily at the same time); secondly a synchronisation mechanism must be provided to allow processes to mutually exclude and/or cross stimulate each other to ensure orderly intercommunication.

Before considering process intercommunication in a language context it is necessary to consider how processes are identified within the language. One approach, often used, is to allow any procedure to become a process. Data communication is then possible via any data areas that are in scope for any two communicating procedures. This approach obviously works (more accurately - systems designed by using this approach can be made to work) but it gives rise to many problems of integration and testing because of poor modularity. This poor modularity arises because data access far in excess of the designer's intention is permitted. In normal block structured languages all global data is available to all processes. In other words the question "how is the access limited to only that set intended by the designer?" is either (a) never asked or (b) answered in the negative.

In the view of the author a different approach which rejects the hierarchical block structured approach at the parallel process network level is preferable. It is necessary to clearly identify the statements that will be used for a process by some special syntactic device eg the language word PROCESS. It is also necessary (and perhaps more important) to force the process to declare explicitly a list of data areas through which it will communicate with other processes. The compiler can then check that no access is made to data areas (external to the process) that are not present on the list. In this way the designer's intentions can be checked by the compiler. However the consequence of this approach is that the concept of a program as a single

block structured hierarchy containing (at various levels as necessary) many co-operating parallel processes is destroyed together with the concept of (universally accessible) global data. Instead we are left with the program consisting of a network of co-operating parallel processes together with the data areas through which they communicate.

ADVANTAGES OF THE NETWORK APPROACH

The network, being static (on a statement-statement timescale), can then be used as the basis for a modular approach to design implementation and testing. The design is conceived as a network from the earliest possible moment. The network can be expressed in diagrammatical notation (eg Phillips network diagram ref 1) to give good design visibility. It is also easy to represent such a network in a data base form for management and project control (eg SDS ref 2). The network clearly identifies the first level of design in terms of modules that can remain visible right through the software production process.

Each module of the network must be described in terms of its purpose (in its context) related to the overall requirement (specification) of the software. In a strongly typed language each data area will have a distinct type and each process will state its communication requirements in terms of these types. Thus the major interfaces of the system are clearly visible and can be expressed:

- (a) in the network diagram
- (b) in the data base
- (c) in the actual program.

Further advantages of this clear modularity become evident in the testing phase. Each process can be individually tested by setting it up with a suitable set of data areas (ie exactly that set which it specifies internally). The process is then unable to distinguish this test environment from any operational environment. Thus it is easier to organise the testing (test processes communicate with the process under test only through the same well defined interface) and it is easier to specify the test and to check for both compliance (when running the test) and exhaustiveness (when vetting the specification).

The aspect of process synchronisation in a system can also be handled in a tidy and modular manner using this approach. The problem of real time interactions and the consequent need for a synchronisation mechanism is concentrated on the access to the intercommunicating data areas made by the parallel asynchronous processes of the network. By adopting the concept of the access procedure, all direct access to the data areas and consequently all use of synchronisation mechanisms can be encapsulated. The access procedures for the data areas thus form a convenient and sensible unit of design, implementation and testing, which collectively provide a level of abstraction that provides a reliable means of communication between processes. This leaves the processes themselves as single sequential processes that need take no special precaution to handle parallelism and asynchronism.

The final advantage of this approach that is worth mentioning here is the promise of re-usable software modules. Because of the well defined external data interface of each process, it follows that any process will work in any context so long as its data environment is set up correctly. This fact has already been used in the testing strategy outlined above but it does mean that processes can be put into a library and fetched for any number of different systems. This re-usability will be further enhanced if, from the inception of the use of the approach suggested here, designers are encouraged to achieve modularity of function within a process.

LANGUAGE AND IMPLEMENTATION IMPLICATIONS

The major implication on language design brought about by the adoption of this approach is the need to reject conventional notions of "the program" and of "global data". The compiler is never concerned with complete programs but always with modules which can be