The Implementation of the Run-time System of A Concurrent Programming Language, Pascal-C

Yin-lam Wong

A Thesis

in

The Department

of

Computer Science

Presented in Partial Fulfilment of the Requirements
for the degree of Master of Computer Science at
Concordia University
Montréal, Québec, Canada

January 1985

© Yin-lam Wong, 1985

ABSTRACT

The Implementation of
the Run-time System of
A Concurrent Programming Language,
Pascal-C

Yin-lam Wong

This thesis describes a multiprocessor project specific to combinatorial computing. An overview of its system architecture, the software and the hardware is provided. A detailed description of the implementation of the communication subsystem of the network and the implementation of the run-time system of a concurrent programming language, Pascal-C, is given with emphasis on their programming methodologies and the problems encountered.

Acknowledgement

I would like to express my sincerest thanks to my thesis supervisor Dr.C.Lam, who guided and encouraged me throughout the preparations of this thesis. He also read the several drafts of this thesis, making numerous suggestions for its improvement. The final version of this thesis owes much to him; the mistakes are my own.

I thank Mr.S.Cabilio who modified the run-time system of Parallel Pascal. His work was very helpful for the preparations of this thesis.

I am grateful to Mr.L.Thiel for the discussion and the advices, to Mrs.P.Dubois for her technical expertise, cooperation and help.

Contents

Chapter 1. Introduction 7

Chapter 2. General description of the project 11

Chapter 3. Description of Pascal-C 15

3.1. Down Procedure 16

3.2. Wait Statement 18

3.3. Terminate statement 19

3.4. Critical Proceudure 20

3.5. Copy Section 22

Chapter 4. Implementation of the Pascal-C System 24

4.1. Brief description of each 'layer' in Pascal-C 24

4.2. Interface between Preprocessor and Run-time system 29

Chapter 5. Description of Run-time system 34

5.1. Data structures 35

5.2. Master Processor 40

5.3. Slave Processor 47

5.4. Cooperation of the master processor and the slave processor 52

Chapter 6. Implementation Techniques used in the run-time system 56

6.1. Update the actual variable parameter of a down procedure 56
6.2. Running a critical procedure requested by a slave 60
6.3. Implementation of the Terminate statement 63

Chapter 7. Description of the Communication subsystem 66
7.1. Overview of the Communication subsystem 66
7.2. User-CSS interface 68
7.3. Internal structures of the CSS 70

Chapter 8. Testing the RTS and running a Pascal-C program 82
8.1. Testing the Run-time system 82
8.2. Running program on the Pascal-C system 85

Chapter 9. Conclusion 90

References. 94

Appendix 1. Example Program 99

Appendix 2. List of Action Codes used by the Preprocessor-RTS interface 119

CHAPTER 1. INTRODUCTION.

The objective of this thesis is to describe a multiprocessor project specific to combinatorial computing, with emphasis on the implementation of the run-time system of a parallel programming language, Pascal-C.

Motivation of the project:

There are many problems of combinatorial nature for which the only known solutions require exhaustive search of many possibilities [Lam et al 83]. For example, examining all sequence S of N moves in a chess game would require operating in a search space in which the number of nodes grows exponentially with N. The critical problem of search is the amount of time and space necessary to find a solution [Feigenbaum 81].

A large main-frame computer is often used to solve these problems because of its speed. However, it also offers fast floating point calculations; it is able to handle a large volume of input/output operations and it is usually associated with a sophisiticated operating system. These features are unnecessary in solving many combinatorial problems. Thus, it is often not economical to use a large computer for solving these problems. On the other hand, a single minicomputer cannot provide the necessary speed.

However a network of mini- or micro- processors may satisfy the speed requirement. Moreover, the special nature of a combinatorial problem allows it to be partitioned into many small individual subproblems. Thus, it is not difficult to assign jobs to each processor in the network.

The system that we have designed to solve combinatorial problem is a network of computers. Each node of the network contains a mini- or a micro- processor, with its own independent memory. One or more nodes of the network also possess other peripherals such as terminals, printers and disk drives so that the system can communicate with the outside world. The fact that the problem can be partitioned into many small subproblems suggests that it is suitable to have a hierarchical (master-slave like) architecuture. The master processor takes a problem and divides it into many subproblems. It then assigns the subproblems to its slave processors. The slave processors find the solution and return the answers to the master processor.

Given the multiprocessor network, it is important to design a high level programming language which is suitable for solving combinatorial problems on such a network. "Several parallel programming languages have been intended as general purpose language. However, they only provide constructs for a low level process creation and communication suitable for real-time computation, and I/O

synchronization. It is difficult to use them for parallelism found in many high level algorithms" [Opatrny 84]. Our project team have designed a high level language, called Pascal-C which is a dialect of Pascal described in chapter 3.

This multiprocessor project was proposed and directed by Dr. Lam [Lam et al 82]. The designs of the system architecture and the concurrent programming language, Pascal-C, were done by a team consisting Dr. Atwood, Mr. Cabilio, Dr. Desai, Mr. Grogono, Dr. Lam, Dr. Opatrny and Mr. Thiel. During the first implementation, a Pascal-C preprocessor was implemented by Mr. Cabilio [Cabilio 85]. The original version of the communication subsystem was written by Dr. Lam. The top level design of the run-time system was also discussed in length by the team. My contribution to the multiprocessor project is the refinement of this top level design of the run-time system and the final implementation of the run-time system of Pascal-C as well as the implementation of the communication subsystem. My work on the implementation of the run-time system of Pascal-C was also helped by Mr. Cabilio who modified the run-time system of Parallel Pascal and Dr. Lam who wrote a program for the extended memory management.

Organization of the thesis:

Chapter 2 gives a general description of the multiprocessor project. Chapter 3 describes the Pascal-C features in detail. These are the extensions of Standard Pascal for parallel computations. Chapter 4 is an overview of the implementation of the whole system. Chapters 5 and 6 describe the run-time system in detail. Chapter 5 describes the organization of the run-time system and its handling of the distribluted features of Pascal-C. Chapter 6 describes some of the techniques used in implementing the run-time system. Chapter 7 describes the communication subsystem. Chapter 8 describes the testing of the system and running of a Pascal-C program on the system. The conclusion is given in Chapter 9. Moreover, Appendix 1 shows a sample program written in Pascal-C language while Appendix 2 shows the action codes used for the interface between the Pascal-C preprocessor and the run-time system.

CHAPTER 2. GENERAL DESCRIPTION OF THE PROJECT.

In this chapter, we describe the objective of the project, the architecture of the multiprocessor system as well as the software and hardware used in this project.

Objective.

The objective of this project is to construct a multiprocessor system for solving combinatorial problems. The system should be powerful enough to solve a problem in a reasonable amount of time and it should be easy to use.

The problems for which the system can appropriately be used have the following characteristics [Lam et al 82].

1. "They are processor-bound: the amount of calculation required is large in comparison to the amount of input and output and is also large in absolute terms. The known solutions of the problems require processing time that increases exponentially with the size of the input."

2. "The given original problems can be partitioned into many subproblems which can be solved independent of one another."

We have decided to have a network of mini- or micro-processor. These processors will co-operate to solve the problems.

Architecture of the system.

Since the given problem can be divided and subdivided into many smaller independent subproblems, we decided to have a hierarchical architecture : a problem is given to a 'master' processor and the master processor partitions the original problem into many subproblems and asks its 'slave' processors to solve these subproblems. In a multi-level system, the slave processor subdivides the given subproblem into sub-subproblems and ask its own slave processors to solve them. The processor, which is both a slave and a master processor, is called an intermediate level processor.

The processors in this system are logically connected as a 'tree' with the 'root' being the master processor to whom the user's original problem is given. The system consists of the following charateristics:

1. A processor is a master, a slave or both.

2. A processor can communicate with its master and slaves only. No communication should take place

between slaves.

3. Every processor has its own memory and no memory is shared between processors.

Software.

Our goal in this area is to provide software that

1. is easy to learn.

2. allows the user to write efficient programs to solve combinatorial problems.

3. can be implemented in a reasonable amount of time.

We have decided to extend the existing language Pascal with features for parallel processing. Pascal-C is Standard Pascal with some extended features that enable programmers to exploit a tree-structured multiprocessor system. Moreover, to write programs in Pascal-C, the programmers do not have to worry about the communication and the synchronization between processors. All the users have to know about the system is the number of levels of the system.

Hardware.

Currently, the configuration consists of a DEC PDP-11/34 and several LSI-11/23's. The PDP-11/34 is used as a master and the LSI-11/23's are used as slaves. Each processor has at least three serial lines with baud rate between 1200 and 19200.

In this first implementation, every processor has a console terminal and a random access system device. It makes debugging easier, because programs do not have to be downloaded at run-time and because the console terminal allows us to trace the actions of all the processors.

CHAPTER 3. DESCRIPTION OF PASCAl-C.

Pascal-C is a programming language specifically designed to solve combinatorial problems on a multiprocessor system with a tree structure. It contains most of the features of Standard Pascal. In addition, we have added some new features which allow programmers to exploit a tree-structured multiprocessor architecture. The new features are as follows:

1. Down Procedure <DP>.

2. Terminate statement.

3. Wait statement.

4. Critical Procedure <CP>.

5. Copy Section.

This chapter describes the new features (Pascal-C features) which are not in Standard Pascal. In each section, we describe briefly a Pascal-C feature as well as its syntax and semantics. For a more detailed description about Pascal-C, please read the Pascal-C Report [Lam et al 82].

Section 1. Down Procedure <DP>.

When a master processor is given a problem, it usually partitions the problem into many subproblems and then assigns these subproblems to its slave processors. The way that a programmer assigns subproblems to the slave processors is by using a Down Procedure (DP for short) call. A DP is like a normal procedure in Standard Pascal except that it is executed by a slave processor rather than by the master processor. Moreover, the master processor can continue with its execution right after the down procedure has been assigned to a slave, without having to wait for the completion of the execution of the DP. Thus, many processors can work in parallel.

Syntax.

```
<DP declaration>::=
    down <procedure heading> <copy section> <block>;
   |down <procedure heading> <block>;
```

In this and subsequent productions, undefined non-terminals are assumed to be those of Pascal. The <copy section> will be described later in this chapter.

Semantics.

The master processor at the 'root' of the 'processor tree' is called the level 0 processor and its slave processors are the level 1 processors. Their slaves are level 2 processors, and so on. In general, down procedures invoked by processor in level K will be executed by processors in level K+1. Calling a DP causes initiation of a process at level K+1.

Moreover, a down procedure P can be invoked as many times as the user wishes. The slave processes, which are concurrently executing the down procedure P, are said to be in the same DP class (class P).

A down procedure can access only its parameters, locally declared variables, procedures and functions as well as objects defined in its Copy Section. It also can call Critical Procedure (CP for short) mentioned in its procedure heading. These CPs will be executed in its master processor. Critical procedures are described in Section 4 in this chapter.

The new values of the actual parameters corresponding to variable formal parameters will not be updated until all the processes executing procedures of this class P have finished their tasks and a WAIT(P) statement is executed for

this DP class.

Section 2. WAIT(DP) statement.

It is obvious that there must be a mechanism by which a master process can be suspended until the outstanding slaves, who are executing DP in this class, complete their tasks. The statement WAIT(DP) is designed for this purpose.

Syntax.

<wait-statement>::= wait (<DP class name>)

Semantics.

After the master processor has distributed a down procedure class to its slave processors, it must execute a WAIT(DP) statement for this DP class before it leaves the environment where the actual parameters passed to the DP are declared. In addition, only a master can execute a WAIT(DP) statement. The effect of the execution of a WAIT(DP) statement is as follows:

1. When a WAIT(DP) statement is executed by a process,

this process will be suspended until all slave processors executing DP in this class have finished their jobs.

2. When the process resumes execution, it updates the variable parameters of this DP class if this DP class is not in the 'terminated' state. The meaning of the 'terminated' state is explained in the next section.

3. It brings this DP class out of the 'terminated' state if this DP class was there.

Section 3. TERMINATE(DP) statement.

There are occasions when the master processor needs to terminate its slave processors whether or not they have completed their current tasks. For instance, some problems are considered solved as soon as any single solution is found. The user can use the TERMINATE(DP) statement to terminate a class of DP when the problem is solved.

<u>Syntax.</u>

<terminate-statement> ::= terminate (<DP class name>)

Semantics.

When a TERMINATE(DP) statement is executed, the class of DP is terminated. All slave processors working on this DP class, are to be freed.

Moreover, it brings this DP class into a 'terminated' state. While a DP is in a 'terminated' state, all further invocations to this DP class will not be sent to a slave. However, any parameter evalution implied by the DP call will still be carried out. This DP class will not exit from the 'terminated' state until a WAIT(DP) statement is executed for this DP class.

Only a master can execute a TERMINATE(DP) statement. This is similar to the WAIT(DP) statement.

Section 4. Critical procedure <CP>.

A critical procedure is a procedure defined in the master process and which may be called by the master itself or by its slaves. However, only the master processor can execute a CP. While executing a CP, it should not be interrupted by the master process or any other CP.

The slave process can report some useful information

before the end of its execution by invoking a critical procedure. It is particularly useful for reporting intermediate results. In the case where only one single solution is required, the slave processor can invoke a critical procedure in the master process to report the answer as well as to terminate this DP class.

Syntax.

<CP declaration>::= critical <procedure heading>
<block>;

Semantics.

A critical procedure declared at process level K can be invoked by process at level K or K+1. However, in order to make a CP at level K available for the process at level K+1, the CP's name must be passed as a parameter to the DP executing at level K+1.

If a CP at level K is called by a process at level K, it will be executed immediately since no other process is in the critical section when the level K process (the master process) is active.

If a CP at level K is called by a slave process at level K+1, it has to wait until the master processsor is not executing another CP.

A CP can have only value parameters and can access objects local to the CP itself or which are declared globally.

Section 5. Copy Section.

Since a down procedure is indeed executed by a slave processor, all non-local procedures, functions, variables and types invoked in the down procedure as well as the parameters of the down procedure must be downloaded to the slave processor before execution of the down procedure. To benefit both the readers of the program and the compiler, all non-local procedures, functions and variables used in a down procedure must be mentioned in the Copy Section of the down procedure.

Syntax.

<copy section>::= copy <identifier> {,<identifier>} ;

Semantics.

All data (variables) mentioned in the copy section of a DP will be downloaded to the slave when a DP of this class is invoked. The current values of the variable will be downloaded to the slave processor while initiating a DP. A slave processor can change the values of these variables during execution but it does not affect these variables in the master side.

The functions, procedures, types declared in the copy section are used by the Pascal-C preprocessor. They are copied into the program in the slave side in order to give a complete and self-contained program. The preprocessor is described in [Cabilio 85].

CHAPTER 4. IMPLEMETATION OF THE PASCAL-C SYSTEM.

In this chapter, we will look at the whole multiprocessor system from the viewpoint of implementation. The system consists of five 'layers'. Section 1 contains the brief description of the functions of each layer. In Section 2, we describe the interface between the layers.

In this first version of implementation, the multiprocessor system contains only two levels. The first level contains only the master processor and the second level can contain an arbitary number of slave processors.

Section 1. Brief description of each 'layer'.

The five 'layers' of the system are listed below:

1. The 'Pascal-C' layer : code generated by the Pascal-C preprocessor.

2. The run-time system layer (RTS) : code from the run-time library of Pascal-C.

3. The 'CSS' layer : code for the communication subsystem.

4. The 'OS' layer : code belonging to the operating system.

5. The lowest layer is the hardware layer.

Each layer calls upon the services of the layers below it but can be defined independently of the layers above it [Lam et al 82]. Each layer is transparent to the layer above it. The function of each layer is described in this section.

Compiler of Pascal-C.

In order to execute a Pascal-C program in our target machine, we need a translator which takes a Pascal-C program as input and translates it into an assembly language program or object codes of the target machine. It is not a simple task to write a Pascal-C compiler. Since there is an existing Parallel Pascal system [PP 82] for the target machine (PDP-11), we have decided, in the first version of implementation of the Pascal-C system, to make use of the Parallel Pascal system.

We have implemented a Pascal-C preprocessor which takes a Pascal-C program as input source program and translates it into Pascal programs as output. Then, we use the Parallel Pascal compiler to compile the resulting Pascal programs.

The Pascal-C preprocessor output can be divided into two parts.

1. the Master Pascal-C: the output program for the master processor.

2. the Slave Pascal-C: the output programs for the slave processor.

In Chapter 8, we shall describe the Pascal-C preprocessor output in detail.

Constuition of the preprocessor was a master thesis project of S.Cabilio, and a detailed description of the preprocessor of Pascal-C is contained in [Cabilio 85].

Run-time system of Pascal-C.

Predefined procedures of a high level programming language (e.g. READ and WRITE in Standard Pascal) may be stored in the run-time library. The compiler will insert codes to call these predefined procedures from the run-time library while compiling. Without a run-time library, we may need a very sophiciciticated compiler to generate very complicated code. The existing Parallel Pascal run-time system supports all Standard Pascal features needed to be implemented in the run-time system. Thus, to implement the

Pascal-C run-time system, it is sufficient to implement the library functions which support the Pascal-C features.

To implement each Pascal-C feature, the Pascal-C preprocessor (or compiler) inserts one or several calls to the run-time system and then the run-time system will perform the necessary actions.

In brief, Pascal-C run-time system does the following:

1. Logical communication between processors (master processor and slave processor).

2. Allocating down procedures to slave processors.

3. Initiating and terminating processors.

4. Updating variable parameters of down procedures.

5. Implementing critical procedures called by slaves.

6. Scheduling the executions of critical procedures and protecting the critical section.

The implementation of Pascal-C run-time system will be described in detail in Chapter 5.

Communication subsystem (CSS).

An efficient implementation of the communication between processors strongly affects the efficiency of our multiprocessor system. This is because messages and data have to be sent often between master and slave.

The communication subsystem supports the physical communication between the run-time system processes. In other words, the run-time system is a user of the CSS. Messages and data communication between a master processor and a slave processor are done via the CSS. The CSS uses the physical link between two processors to provide several virtual channels which can be opened or closed individually. The CSS contains several external procedures which can be called by its users to do the following:

1. Initiate/terminate the CSS.

2. Open/close the virtual channels.

3. Send/receive message through a specific channel.

The user does not have to be concerned with the internal structure of the CSS and can consider the CSS as an error free communication system. Indeed, error recovery is done within the CSS. In Chapter 7, the CSS is described in

detail.

Operating system layer.

The operating system used is RT-11. The Parallel Pascal run-time system will call the operating system library to implement I/O operation whenever an I/O is necessary.

In this version, every processor must have its own copy of the operating system RT-11 because each of them possesses at least a terminal and a random access system device.

Section 2. Interface between Preprocessor and Run-time system.

To implement the Pascal-C features, a Pascal-C compiler or a preprocessor must insert code into its output programs to call the run-time system in order to take the appropriate actions. Thus, the interface between Pascal-C compiler/preprocessor and run-time system must be very clear. No misunderstanding or ambiguity is allowed.

Since Pascal-C is implemented as a preprocessor rather than as a compiler, calls to the run-time system cannot be

inserted in the same way as is usually done in a compiler. All the run-time system calls inserted into the preprocessor output programs are declared as external procedures. For each type of run-time system call, a separate action code is associated with it. Each run-time system call supplies the action code and the required parameters. The action codes were proposed by Mr. Cabilio.

The action codes are grouped according to the number of parameters associated with it and whether the parameter is a value parameter or a variable parameter. This idea was suggested by Dr. Lam. A list of the action codes and the meaning of their associated parameters is shown in Appendix 2.

Since all run-time system procedures are external procedure to the Pascal-C preprocessor output programs, type checking is defeated. This is an important simplification. The following example shows how the RTS exploits this fact. Here, the action code is 64 and it tells the RTS to download the value parameter P1 and the length of the parameter is P2. The RTS procedure for this action group is RTS3.

The RTS procedure declared in the preprocessor output program is as follows:

```
VAR A:type for P1;
    LENGTH:integer;
PROCEDURE ZZPRO1 (ACTION:INTEGER; VAR P1:type for P1;
                  P2:INTEGER);
  PROCEDURE RTS3 (ACTION:INTEGER; VAR P1:type of P1;
                  P2:INTEGER); EXTERNAL;
BEGIN
  RTS3(ACTION,P1,P2);
END;
  .
  .
ZZPRO1(64,A,LENGTH);
  .
  .
```

Procedure RTS3 defined in RTS is as follows:

```
PROCEDURE RTS3 (ACTION:INTEGER; ADDR:INTEGER; L:INTEGER);
```

Two points should be noted in the above example. First, the Procedure RTS3 declared in the preprocessor output program is within the Procedure ZZPRO1. Using this method we can call RTS3 to download parameters with different types. Secondly, the second parameter of the RTS3 declared in RTS is a value parameter with Integer type. Since Pascal passes the address of the variable parameter, the RTS can treat it as an integer type variable and thus

pick up the starting address of the parameter. The actual data type of the parameter P1 is not important to the RTS.

For a master processor, the preprocessor inserts RTS calls under the following conditions:

1. When the main process has to get into or to exit from the critical section.

2. When it needs to reserve a slave processor to run a down procedure.

3. When it wants to inform the reserved slave processor to execute a down procedure.

4. When it has to download data.

For a slave processor, RTS will be called under the following conditions:

1. When the slave processor is ready to receive a job (to execute a down procedure).

2. When it needs to call a critical procedure.

3. When it wants to upload variable parameters whose values have changed since downloading.

4. When it needs to receive data from the master processor.

CHAPTER 5. DESCRIPTION OF THE RUN-TIME SYSTEM.

Overview of the Run-time System.

The Pascal-C programming language is designed in such a way that the users do not have to worry about the communication between processors or the synchronization of processes.

The Pascal-C run-time system has the following two major functions.

1. Responses to the RTS calls from the Pascal-C preprocessor or compiler.
   To implement Pascal-C features, the Pascal-C compiler or preprocessor generates appropriate RTS calls. The run-time system will then take the appropriate action.

2. Communication between processors.
   Communication between the master process and the slave process takes place in the RTS level.

In this chapter, we first describe some of the important data structures used in the RTS (Section 1). We then describe the run-time system from the viewpoint of a

master (Section 2) and then from the viewpoint of a slave (Section 3). In each section, we describe what the run-time system will do for different RTS calls from the Pascal-C code; and how the RTS responds to the message received from its master or slave. Since the implementation of some Pascal-C features requires the cooperation of the master processor and the slave processor, we describe their cooperation in Section 4. Some implementation techniques will be described in Chapter 6.

Section 1. Data Structures.

In order to do its job properly, the RTS has to keep and update some information about the status of slaves, down procedures, critical procedure. More specifically, it has to keep the following information:

a. SLAVE-STATUS-RECORD.

This record keeps the current status of slave processor. The master keeps a SLAVE-STATUS-RECORD for each of its slaves. A SLAVE-STATUS-RECORD contains the following fields:

1. STATE: it indicates what the slave is currently doing. There are three possible states: a. idle. b. dead. c. busy.

2. DP-ID: if the slave is running a down procedure, this is the identification number of the down procedure it is running. If this slave is idle, DP-ID indicates the down procedure it was previously running. This implies that the code of this down procedure is still in the memory of the slave.

3. CAPACITY: this contains information about the slave, for example, whether it has its own slaves or what peripherals it has. This field will be useful for the later versions and is not used in the current version.

b. DP-STATUS-RECORD.

This record keeps the current status of a DP class. The master process keeps a DP-STATUS-RECORD for each DP class. A DP-STATUS-RECORD contains the following fields:

1. WAIT-FLAG: it is set by executing a WAIT(DP) statement. Right after execution of the WAIT(DP) statement, it is cleared by reinitializing the DP-STATUS-RECORD.

2. TERMINATE-FLAG: it is set by executing a TERMINATE(DP) statement and is cleared when the DP

class exits from the 'terminated' state.

3. ACTIVE-NO: it contains the number of slave processors currently executing DP in this DP class. It is initialized to 0. Add 1 to it when a slave starts executing a DP in this DP class and subtract 1 from it when the slave has finished execution of the DP.

4. ACTIVE-FLAG: it is set by activating a DP in this DP class and cleared when both the WAIT-FLAG is set and ACTIVE-NO is zero.

5. SLAVE-OCCUPIED: it keeps information about which slaves are running down procedures in this class. An array of bits is used. Each bit corresponds to a slave processor. For instance, bit 0 corresponds to slave number 0, bit 1 corresponds to slave number 1, and so on. The corresponding bit is set if the slave is running a DP of this class and is cleared when the slave has finished executing the DP.

6. HEAD-OF-VAR-PARA-LIST: it is the pointer to the array containing the address of actual parameter list in the extended memory. Figure 6.1 shows the data structure of this variable.

7. NUMBER-OF-VAR: indicates the number of variable parameters in the formal parameter list in this DP.

c. CP-DESCRIPTOR.

It contains part of the information required by the master processor in order to execute a CP invoked by a slave processor. It conatins only two fields:

1. ENTRY-POINT: the entry point of the CP.

2. STATIC-LINK: the static link of the CP.

d. CP-PACKAGE.

This package contains the information necessary for the master processor to execute a CP requested by a slave processor. A CP-PACKAGE contains the following fields:

1. the CP-DESCRIPTOR.

2. PARAMETERS: all value parameters of the CP.

e. UPLOAD-VAR-PACKAGE:

This package contains the information of the actual variable parameter whose value has changed during execution of a DP. It is prepared by the slave

processor executing the DP and it will be sent to the master. It contains the following fields:

1. FORMAL-VAR-NUMBER: an identification number for the formal parameter whose actual parameter is to be uploaded.

2. ACTUAL-PARAMETER-ID: the identity of the actual parameter. It is indeed the memory address of the actual parameter and its value is supplied by the master processor.

3. POSITION: the position of the element whose value has changed.

4. NEW-VALUE: the new value of the element.

5. LENGTH: the length (in bytes) of the element changed.

We will explain how and when to use or update the information in the above data structure later in this chapter and Chapter 6.

Section 2. Master Processor.

Run-time system call from the Pascal-C in-line code.

Pascal-C preprocessor generates RTS calls for each of the following conditions:

1. The master process wants to call a down procedure.

2. The master needs to send data to the slave processor.

3. The master executes a WAIT(DP) statement.

4. The master executes a TERMINATE(DP) statement.

5. A CP is going to be invoked.

6. During initialization of the Pascal-C system.

Down procedure call.

If the called DP class is not terminated, the main process is blocked until the RTS can find a free slave processor and reserve this slave processor for running this down procedure. When such a processor is available, the RTS sets the ACTIVE-FLAG and updates the SLAVE-OCCUPIED field in

the DP-STATUS-RECORD. It also updates the STATE and DP-ID fields in the SLAVE-STATUS-RECORD.

The run-time system checks if this down procedure call is done in a critical section. If so, an error is detected because Pascal-C does not allow a DP being invoked from within a critical section.

It then gets into a critical section and informs the reserved slave processor to prepare for the execution of the down procedure. The reason that it gets into a critical section is to prevent the data, which are to be sent to the slave processor, from being changed by other slave processor who may execute a critical procedure.

In the case that the DP class is in the 'terminated' state, the RTS does nothing and returns immediately.

Downloading data to slave processor.

The data needed to be downloaded to the slave are 1) the parameters of the DP, 2) the variables mentioned in the copy section of the DP and 3) CP-DESCRIPTOR(s).

Two kinds of parameters may be downloaded to the slave processor, namely, a variable parameter and a value parameter. Sending value parameter to the slave processor

is simpler because all that has to be done is to send the current value of the actual parameter to the slave processor.

For a variable parameter, there is more to be done besides sending the current value of the actual parameter to the slave processor. The language definition of Pascal-C requires the actual parameter to be updated only when a WAIT(DP) statement is executed for this DP class. Thus, we need to remember the address of the actual parameter and reserve some temporary spaces to store the returned values. We will also send the ACTUAL-PARAMETER-ID to the slave processor. In Section 4, we describe how variable parameters are updated in detail.

The downloading of the CP-DESCRIPTOR and of data mentioned in the Copy Section is the same as the downloading of value parameters.

After all the data are sent to the slave processor, the main process exits from the critical section.

Executing a WAIT(DP) statement.

The RTS will first set the WAIT-FLAG for this DP class. The main process is blocked until there is no active process executing a down procedure in this DP class. When there is

no slave processor executing a down procedure in this DP class, the RTS will update all the actual parameters if the TERMINATE-FLAG is not set. Finally, the RTS will reinitialize the DP-STATUS-RECORD for this DP class. It will also return to the space pool the temporary space used to store the returned values of the variable parameters.

In the case where the ACTIVE-FLAG is not set, the execution of a WAIT(DP) statement will have no effect except that optionally a warning message is printed.

Executing a TERMINATE(DP) statement.

The RTS will first set the TERMINATE-FLAG for this DP class and then send a 'terminate' signal to the slave processors which are executing the down procedure in this DP class. Lastly, all the spaces reserved for storing the returned values of variable parameters are released.

Similiar to the execution of a WAIT(DP) statement, the execution of a TERMINATE(DP) statement will have no effect if the ACTIVE-FLAG is not set.

Executing a CP.

The Pascal-C preprocessor inserts code to call the appropriate RTS routines for getting into a critical section

right before a CP is invoked. After the execution of a CP, it will also invoke the RTS in order to exit from the critical section. By doing so, it guarantees that the execution of the CP will not be interrupted by another CP.

Initialization of RTS.

Before the user program starts execution, the RTS first initiates the processes in both the master and the slave processors which handle communication. It also initializes the global variables such as SLAVE-STATUS-RECORD and DP-STATUS-RECORD.

Communication between the master and the slave processors.

The CSS uses the physical link between two processors to provide several virtual channels while the RTS uses these channels to send/receive messsages and data to/from its buddy. The two processors connected by a physical link are buddies of each other. We provide a physical link for each pair of master processor and slave processor. Moreover, the RTS opens two channels for each pair. One channel is used to send/receive messages while the other is used to send/receive data.

Since a processor cannot predict when a message will be

received from its buddy (master/slave), we need to have a process, which is dedicated to receive messages from its buddy. For a slave processor, a process FROM-MASTER is dedicated to receiving messages from the master. The messages received by the process FROM-MASTER will be described in the next section. For a master processor, a process FROM-SLAVE is dedicated to receiving messages from its buddy slave. We now list the type of message the process FROM-SLAVE may receive. In the rest of this section, we describe what actions the Master RTS will take after receiving these messages.

1. The slave uploads a new value of a variable parameter of a DP.

2. The slave requests to execute a CP.

3. The slave uploads the CP-PACKAGE.

4. The slave has finished its current task.

The slave uploads variable parameters.

When the slave uploads the new values of variable parameters, the Master RTS will temporarily store the new values in the extended memory. The actual parameters are not updated until a WAIT(DP) statement is executed for this

class of DP. The whole process of updating variable parameters of down procedures is described Chapter 6.

The slave request to execute a CP.

Since critical procedures must be executed in a critical section, the request should not be accepted right away if the master process is in a critical section. However, the Master RTS implements a Pending-CP-Queue to enqueue the request (in the current version, this queue can take only one element). If the Pending-CP-Queue is not full, the RTS puts the request in the queue and then sends a 'go-ahead' signal to the slave to tell it that the request is accepted. Otherwise, the master ignores this request and the slave has to keep on requesting until the request is accepted.

The slave uploads the CP-PACKAGE.

Having received the 'go-ahead' signal, the slave will send the CP-PACKAGE to the master processor. The Master RTS will then store it in the extended memory. This CP-PACKAGE will not be opened until the RTS actually schedules to run the CP.

The slave informs that it has finished the current task.

In this case, the master will update the ACTIVE-NO and the SLAVE-OCCUPIED fields in the DP-STATUS-RECORD as well as the STATE field in the SLAVE-STATUS-RECORD.

Section 3. Slave Processor.

Run-time system calls from the Pascal-C in-line code.

Pascal-C preprocessor generates RTS calls in the following situations:

1. The slave is ready to execute a new task given by the master.

2. The slave needs to receive data from the master.

3. The slave needs to upload the new values of a variable parameter.

4. The slave wants to execute a CP in its master processor.

Ready to execute a new task given by the master.

When the slave processor is ready to execute a new task, the Slave Pascal-C will get into an idle loop until the master has assigned a new task to it.

Receiving data from the master processor.

There are four kinds of data the slave may receive from its master. They are 1) variable parameters, 2) the value parameter, 3) data mentioned in the the Copy Section of this DP in the user program and 4) CP-DESCRIPTOR for those critical procedures mentioned in the heading of the DP.

The preprocessor is responsible for reserving enough space to store the received data. The starting address and the length of the data are part of the parameters in the RTS call. All the Slave RTS has to do is to call the appropriate routine of CSS to receive the data.

However, since we may need to upload a variable parameter later on, the RTS will receive the ACTUAL-PARAMETER-ID of the actual parameter so that it will be possible to upload the variable parameter. The Slave RTS will save the ACTUAL-PARAMETER-ID for each variable formal parameter. This is described in detail in Chapter 6.

Uploading variable parameters.

When the Slave Pascal-C invokes the RTS for uploading a variable parameter, it supplies the variable formal parameter's ID number, the position of the element being changed, the new value of the element and the length (in bytes) of the element. The RTS will use the given information to prepare the UPLOAD-VAR-PACKAGE.

For every variable parameter, only those elements whose values have changed, will be uploaded. When the slave processor uploads the new values of a variable parameter, it sends to its master the UPLOAD-VAR-PACKAGE.

Executing critical procedures.

When the Slave Pascal-C wants to call a CP, it supplies the CP-PACKAGE while invoking the Slave RTS.

To execute a CP in the master processor, the slave first sends a request to the master, and keeps on doing so every 2 seconds until it receives the 'go-ahead' signal from its master. Then, it sends its master the CP-PACKAGE for the CP it wants to execute. Finally, the Slave Pascal-C is resumed.

Communication between the master and the slave processors.

In order to communicate with the master processor, the slave processor will open the two communication channels on its side. One is to send/receive messages to/from its master processor while the other is to send/receive data to/from the master processor.

The process FROM-MASTER is dedicated to receive control messages in the slave side. The messages that FROM-MASTER may receive are listed below:

1. The master informs the slave to get ready for executing a certain down procedure.

2. The master informs the slave to terminate the current task.

3. The master informs the slave to close the system.

4. The master wants to know if the slave is alive.

5. The master tells the slave that the request for executing a critical procedure is accepted.

Getting ready to execute a down procedure.

The Slave RTS receives a parameter from its master that indicates which down procedure is to be executed. The Slave RTS passes control back to the Pascal-C user program with this parameter. In Chapter 8, we will describe how the Pascal-C program loads the down procedure code into memory.

Termination of the current task.

In a two-level system, the slave processor will simply terminate the current task and get back to the state in which it waits for a new task. However, in a multi-level system, the slave processor must terminate its own slave processor before it can terminate its current task.

Closing system.

The system should be closed only at the end of the execution of the user program. In other words, it closes the system when the whole problem is solved and thus no processors should be working on any subproblems.

The Slave RTS will first check whether it itself is in the idle state (waiting for new task). If so, it tells its master that it agrees to close the system. If it is running a DP, it will inform the master that it is an error to close

the system.

The master wants to know if the slave processor is alive.

In this case, all the slave processor has to do is to signal the master that it is fine.

Request to execute critical procedure is apccepted.

When the request to run a CP is accepted, the Slave RTS uploads the CP-PACKAGE of the CP it wants to execute and then resumes the Slave Pascal-C.

Section 4. Cooperation of the master and the slave processor.

This section provides a step-by-step description of a DP and a CP call. Here the term Master Pascal-C and Slave Pascal-C refers to the preprocessor output for the master program and the slave program respectively. The calls to the Master RTS and Slave RTS are embedded in-line in the program.

Down Procedure Call:

Master Pascal-C: The user program calls a DP and the parameters of the DP are evaluated. Then

it calls the RTS.

Master RTS: If the TERMINATE-FLAG is not set, it reserves a slave processor for this DP before it resumes Pascal-C. Otherwise, the Pascal-C is resumed immediately.

Master Pascal-C: It first gets into a critical section and then sees if the DP class is terminated, if not, it calls the RTS.

Master RTS: Informs the reserved slave processor to start execution of the DP called.

Pascal-C: Master Pascal-C informs the Master RTS to send data to the slave processor. Meanwhile, Slave Pascal-C informs the Slave RTS to receive data. Communication will not be done until both sides are willing to send and receive.

[Some time later... Assuming that this class of DP is not terminated]

Slave Pascal-C: Informs the Slave RTS to upload the changed values of the variable parameters, if any.

Slave RTS:	Uploads the UPLOAD-VAR-PACKAGE.
Master RTS:	Stores the uploaded new values in the extended memory.
Slave Pascal-C:	Completes the process and passes control to the Slave RTS.
Slave RTS:	Informs the Master RTS that it has finished its current task and is waiting for the next task.
Mastesr RTS:	Updates the DP-STATUS-RECORD and the SLAVE-STATUS-RECORD. Finally, sets this slave free.

Critical Procedure Call:

Slave Pascal-C:	Requests the Slave RTS to execute a CP in the master.
Slave RTS:	Signals the Master RTS that it requests to execute a CP.
Master RTS:	Puts the request in the Pending-CP-Queue if the queue is not full, it signals the Slave RTS that the request is accepted.

If the queue is full, the Slave RTS does not receive the permission and goes back to the previous step to summit the request again until the request is granted.

Slave RTS: Sends up the CP-PACKAGE. Then it can resume the Slave Pascal-C.

Master RTS: Stores the CP-PACKAGE and runs the CP as soon as no one is in the critical section.

CHAPTER 6. IMPLEMENTATION TECHNIQUES USED IN THE RUN-TIME SYSTEM.

In this chapter, we describe how the RTS is actually implemented. In particular, three items are described.

1. How the variable parameter of a down procedure is updated.

2. How a critical procedure called by a slave processor is run.

3. Implementation of the TERMINATE(DP) statement in a slave processor.

Section 1. Update the variable parameter of a down procedure.

In Pascal-C, the variable parameter of a down procedure has to be updated in the master after the down procedure has been executed. This is because we do not have shared memory in the system. Therefore, it is not sufficient to pass the memory address of the variable parameter to a slave processor when a down procedure is called. In fact, the actual parameters of a class of DP are not updated until a WAIT(DP) statement is executed for this DP class. On the

other hand, it is possible that a slave returns the new values of variable parameter before a WAIT(DP) statement is executed for the class of DP. Moreover, Pascal-C does not allow any element of an actual parameter to be changed more than once. Thus, we need to store the returned values of a variable parameter temporarily until all the slaves executing the DP in this class has finished their tasks and a WAIT(DP) statement is executed for this DP class. To implement the requirement mentioned above, we decided to do the following:

1. The ACTUAL-PARAMETER-ID of the actual parameter is downloaded to the slave during the downloading of a variable parameter. The slave can use it to identify the actual parameter when it returns the new values of a variable parameter. Since the address of an actual parameter is unique, the ACTUAL-PARAMETER-ID contains this address only.

2. Since it may take a large amount of space to store the returned value of a variable parameter, we decided that the returned value of a variable parameter would be temporarily stored in the extended memory to reduce the chance that the user will run out of memory space. (The term 'extended memory' refers to physical memory above the 64K byte boundary that can be accessed only by using special

hardware). The extended memory is treated as a heap area. Before activating a slave to execute the DP, the master RTS will first reserve some space in the extended memory for storing the returned values of the variable parameters of the DP.

3. Figure 6.1 shows how the returned value of a variable parameter of a DP is kept. The element HEAD-OF-VAR-PARA-LIST in the DP-STATUS-RECORD points to the dynamic array FORMAL-VARS where FORMAL-VARS[i] points to the ACTUAL-PARAMETER-LIST for the ith formal variable parameter. There can be several distinct actual parameters for a single formal variable parameter. All the actual parameters passed as the same formal parameter are linked together in the ACTUAL-PARAMETER-LIST. However, the same actual parameter passed to different formal parameters will be considered as a distinct acutal parameter. A BIT-MAP is used to keep track of which elements in the actual parameter have changed. Each bit corresponds to an element of the actual parameter. All bits are set to zero at the beginning. When the slave returns a new value of variable parameter, the Master RTS will first check if the value of the element has already changed by looking at the bit which corresponds to this element. If the bit is zero, change it to one

and store the new value. If the bit is already one, the RTS will report the error.

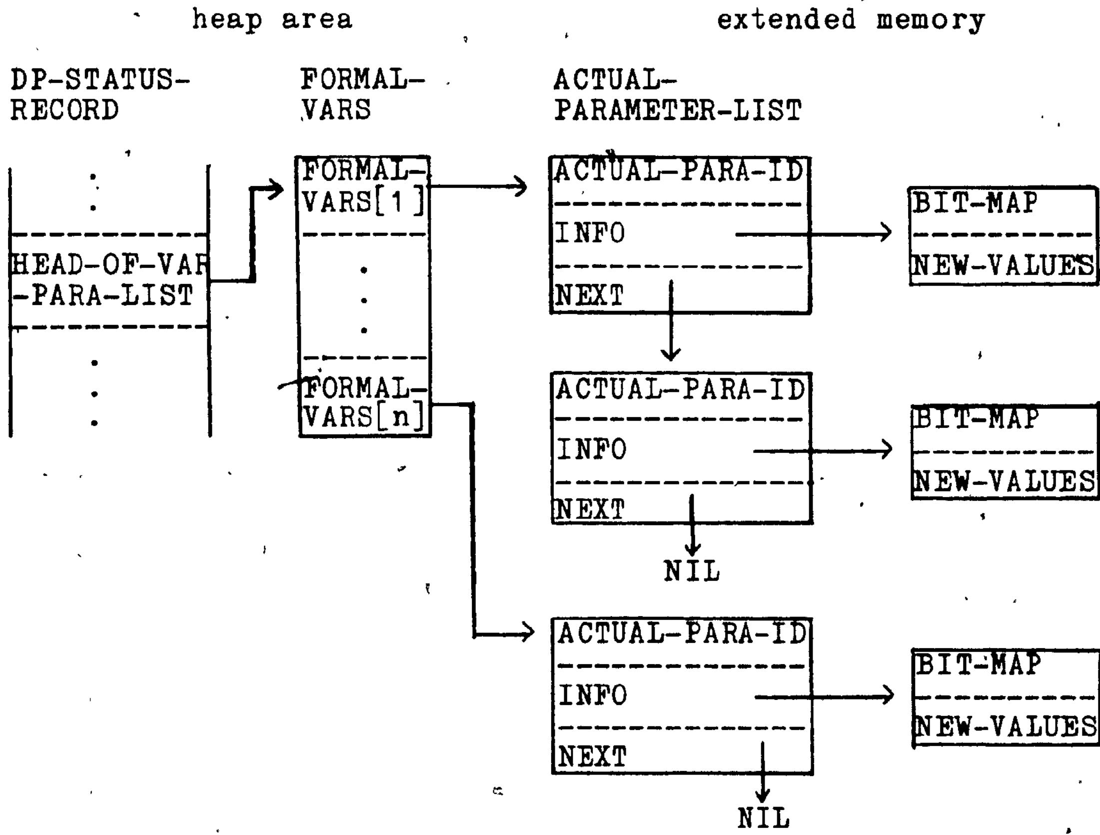

Figure 6.1. Data structure of storing variable parameter.

<u>Rationale.</u>

During the implementation of the updating of the variable parameter of a down procedure, we did consider other alternatives. For example, it is possible to upload all variable parameters regardless of whether their values have changed or not. In this case, the slave RTS would not

need to keep the information about the UPLOAD-VAR-PACKAGE. Some computation time and working space in the slave side can be saved. The original values of the actual parameters would be kept by the Master RTS rather than by the Slave Pascal-C. The comparison of the original values and the new values of the actual parameters would be done by the Master RTS rather than the Slave Pascal-C. However, the Master RTS would have to record the current values of the actual parameters during each invocation of a down procedure. This could lead to using a much larger working space in the master side.

The workload of the master is quite heavy especially when the number of slaves is large. Moreover, the memory space on the master side is limited. Thus, we rejected this alternative.

Section 2. Running a critical procedure called by a slave.

In order to run a CP called by a slave, the entry point and the static link of the CP and the parameters (if any) must be known to the master processor. Moreover, all the CPs must be executed in a critical section and use the main process stack. We describe how it is done below:

1. In order to have sufficient information to run a CP

called by a slave, the preprocessor inserts some code which sends the CP-DESCRIPTOR (containing the entry points and the static link) of all CPs which may be called from within a DP, to the slave processor, when the DP is activated. When the slave processor requests to execute a CP, it will send to the master the CP-PACKAGE (containing the CP-DESCRIPTOR and all the parameters of the CP) so that the master processor will be able to execute the desired CP.

2. Since a CP may need to use a large stack during the execution (e.g. a CP may be a recursive procedure), the CP should use the main process stack. In other words, to execute a CP called by a slave process, the Master Pascal-C has to be suspended while the main process is executing the CP.

When the process FROM-SLAVE receives the request to run a CP, it will schedule to run a CP according to the following situations:

a. The critical section is not occupied.

   In this case, it first gets into the critical section and then calls a routine (in Parallel Pascal), which changes the program counter (PC) of the main process to the entry point of 'Run-CP' (a routine which will invoke the CP

called by a slave). Thus, the main process will execute the CP called by the slave before executing the Master Pascal-C when the main process is resumed.

b. The critical section is occupied.

Since only the main process may execute a CP, the process occupies the critical section must be the main process. Thus, in this case, the Master RTS will reserve the critical section for running the CP called by the slave. When the main process tries to exit from the critical section, the Master RTS will check if the critical section is reserved for running a CP called by a slave. If so, instead of resuming the Master Pascal-C, it schedules the main process to run the CP by calling the routine 'Run-CP'.

3. Procedure Run-CP is a macro routine. When it is in control, it first picks up the information in the CP-PACKAGE and loads the entry point and the static link as well as parameters of the CP into the main stack. It then calls the CP.

   After the execution of the CP, it passes control back to the RTS. The RTS will then schedule whatever should be done next.

Rationale.

We have considered another alternative to implement the running of a CP requested by a slave. We can create a process, called 'Process-CP', which is dedicated to running the critical procedure called by a slave. The advantage of adopting this alternative is that it is not necessary to make changes, such as changing the program counter, at the very low level of the system. In other words, the coding of the RTS would be cleaner and it would make the program more transportable.

However, we would need to reserve a very large stack for the process 'Process-CP'. Otherwise we would have the risk of running out of stack space, especially when the critical procedure is a recursive procedure. Since memory space for the Pascal-C user is already very limited, we decided not to use this alternative.

Section 3. Implementation of the TERMINATE(DP) statement.

The execution of a TERMINATE(DP) statement for a class P, in fact, affects the slave processors who are executing the DP of class P. These slaves should terminate execution of the DP and return to the idle state where the slave processors are waiting for new tasks to be assigned by its

master processor. In order to do so in a proper way, without spoiling things in the RTS and the layers below RTS, we have to take care of the following things:

1. In a multi-level system, an intermediate level processor must set the TERMINATE-FLAG for all its DPs and must signal its own slaves to terminate.

2. It makes sure that no process other than the slave main process occupies any space in the heap area, and then restore the Heap Pointer.

3. Restore the program counter (PC) and the stack pointer (SP) of the main process so that the main process can get back to the idle state and wait for a new task.

4. Initialize the global variables in the RTS so that it is ready to take care of a new task.

In order to restore the PC and SP of the main process and the Heap Pointer to the point where the slave is in an idle state, we need to store their values when the system has just started up. There are two routines (written by S.Cabilio) which are used to store these values at the beginning and to restore them when needed, in the Parallel Pascal RTS.

In this version, the CSS uses the heap area to keep temporary data. Thus, before restoring the Heap Pointer, the Slave RTS has to verify that all the pending messages has been sent/received and the CSS is in a 'stable' state (No traffic is going on). Then, the RTS will raise the priority of the current process and restore the SP and PC of the main process and the Heap Pointer. Raising the priority of the current process gaurantees that the current process will not be interrupted.

CHAPTER 7. DESCRIPTION OF THE COMMUNICATION SUBSYSTEM.

In this chapter, we describe the communication subsystem (CSS). Section 1 gives an overview of the CSS. Section 2 describes the interface between the user and the CSS (User-CSS interface). Section 3 describes the internal structures of the CSS.

Section 1. Overview.

The CSS is organized into two parts. 1) the User-CSS interface and 2) the internal structures of the CSS. The internal structures of the CSS are transparent to the user. Thus, the users need to know only the User-CSS interface in order to use the CSS.

The function of the CSS is to pass messages between processors. However, it does not interpret the message itself. The basic concept of the User-CSS interface is the idea of a frame and a channel.

Channels are used to pass information between two processors. Several channels can be open on the same physical link between two processors. The two processors connected together with a physical link are said to be buddies of each other.

Frames are used to contain the messages that need to be sent between the two processors. A frame in our CSS consists of two parts: 1) the length of the message (in bytes) and 2) the message itself. A maximum limit is imposed on the length of a message so that the CSS can expect the size of the frame.

When a processor wants to send/receive a message to/from another processor, it specifies the identity of the buddy and the channel number. The message will be actually tansmitted when both the sender and the receiver has issued a send and a receive request respectively. In other words, the sender and the receiver processes are blocked until the message has transmitted.

From the users' viewpoint, the channel appears to be error free; all messages are sent correctly. In fact, it is the CSS which detects the transmission errors and tries to retransmit the message if an error has occurred. In case that it cannot correct the error after several retransmissions, it will report the error to the user and halt the system.

Section 2. User-CSS interface.

This part describes the six procedures that can be invoked by the users of the CSS. These procedures are described below.

1. Procedures to start/stop the CSS.

```
Procedure Startcss;
Procedure Stopcss;
```

Before using the CSS, the user must call 'Startcss' so that the CSS can have a chance to initialize all its processes and variables.
When the user has finished using the CSS, he/she should call 'Stopcss' so that the CSS can delete all the processes it has created and return all the memory space that it has used.

2. Procedures to open/close a virtual channel.

```
Procedure Openchannel (bid, cno : byte; var
    receiverframe : frame; var errorflag : boolean);
Procedure Closechannel (bid, cno : byte; var
    receiverframe : frame; var errorflag : boolean);
```

To open/close a bidirectional virtual channel, the

user needs to specify to which buddy (bid) and for which channel (cno) he/she wants to open/close a channel.

In order to know where to store the received frame, the user has to let the CSS know the address of the receiver frame (receiverframe). Every time the CSS receives a message for the user, it stores it in the 'receiverframe' and the user can pick it up.

In case that the CSS finds an error during the opening/closing of the specified channel, it signals the user by raising the 'errorflag'.

3. Procedures to send/receive frame.

```
Procedure Sendframe (bid,cno,trafficcode:byte;
                     var outframe:frame);
Procedure Receiveframe (bid,cno:byte);
```

To send/receive a data frame, the user must specify to/from which buddy (bid) and via which channel (cno) it wants to send/receive the message.

Since the receiver has already specified where the receiving frame is, there is no need for him to specify the receiving frame again. However, the sender has to specify where the outgoing message (outframe) is.

The value of the 'trafficcode' in procedure

Sendframe can be either 1 or 2. In case that a user keeps on sending messages to the second user and does not expect to receive any message in return, the first user should set the parameter 'trafficcode' to 1. When the CSS sees that 'trafficcode' is 1, it will inform the second user to send back the acknowledgment frame right after he has received the message without waiting for the acknowledgment-time-out to occur since there is no outgoing frame in the second user's side. This aspect of improving the communication efficiency will be explained in detail in Section 3.

Section 3. Internal structures of the CSS.

The CSS is organized in three layers. They are as follows:

1. Physical layer : it does the actual transmission based on bytes.

2. Datalink layer : it handles error detection and recovery.

3. Routing layer : it controls the traffic (decides when to send a user's frame).

Equivalent layers in two different processors may communicate with each other. Each layer will interpret only the information supplied by the same layer of its buddy processor.

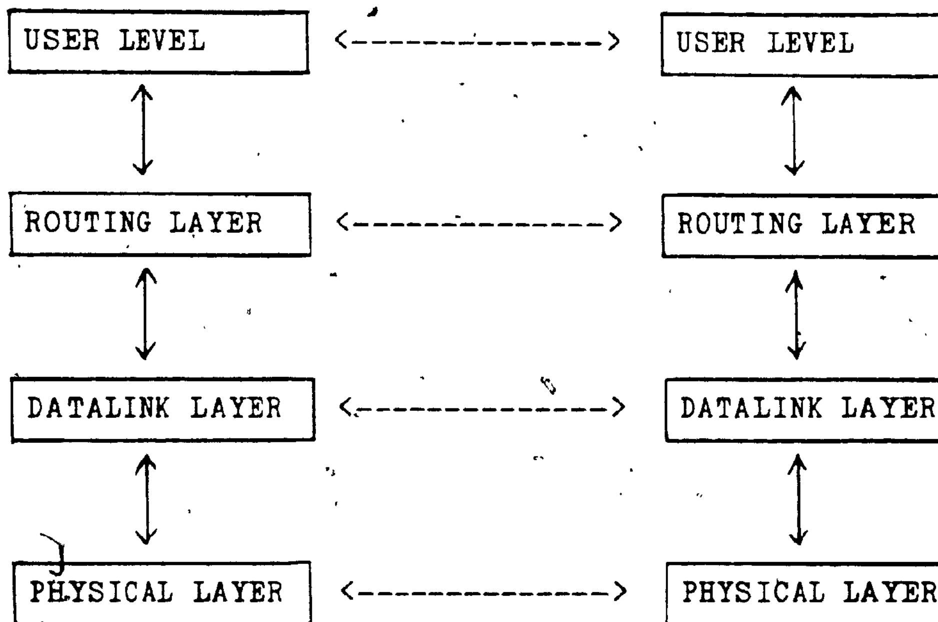

Figure 7.1. Layers and their interfaces.

Physical layer.

It consists of several interrupt processes to do the actual transmission. The major interrupt processes are the Transmit process and the Receiver process. These interrupt processes are mostly device dependent.

The actual transmission is done one byte at a time. Since they are interrupt driven, they send/receive a byte

whenever an interrupt occurs. They also counts the number of the bytes that they have sent/received for a frame. The receiver signals the datalink layer when a whole frame is received. These processes have no capabilities of doing error detection or error recovery.

Data-link layer.

In order to detect error and to do error recovery whenever possible, a header and a trailer are added to the user's message frame. The header and the trailer are used to contain information which is for the purpose of detecting errors. The datalink layer of the receiving processor will look at the header and the trailer when a frame is received. If the message is a good one, it takes away the header and the trailer and passes only the message to its host (the process that the CSS is servicing).

The header contains the length of the padded message, the receiving processor's identity, the sequence number of the frame and the acknowledgment number. The trailer contains the checksum. The use of the above information will be discussed below. There are two major error detecting and error recovery methods. They are as follows:

1. Compute the checksum by using CRC (Cyclic Redundancy

Checking) method.

2. One bit sliding window protocol.

CRC checksum.

In order for the receiver processor to see if any transmission error has occurred in a frame, we use the CRC method with the CRC-16 as the generator polynomial. Before a frame is sent, the sender calculates the checksum and puts the result in the trailer of the outgoing frame. When the receiver processor receives the frame, it will recompute the checksum. If the checksums do not agree, a transmission error has occurred.

We had tried the byte-wise CRC calculation both with and without the look-up table. With the look-up table, we need 0.5 kilobytes of memory space to store the table. But, it works 2.85 times faster than the version without the look-up table. Figure 7.2. shows their results.

Number of Clock ticks (60/sec)

Byte in message	BIT WISE	BYTE WISE (without look-up table)	BYTE WISE (with look-up table)
1200	101	57	20

Figure 7.2. Comparison of the CRC routines

Sliding window protocol.

With a sliding window protocol, every processor keeps information about which frames are permitted to be sent (Next-frame-to-send) and which frames are expected to be received (Frame-expected). Next-frame-to-send and Frame-expected are usually a sequence of numbers which range from 0 to N. The maximum number N of Next-frame-to-send implies that the processor has reserved N buffers in memory for frames being sent but not yet acknowledged.

When the sender sends out a frame, it inserts the sequence number of the outgoing frame into the header. This sequence number is the same as the Next-frame-to-send in the sender's side. After the receiver has received a frame, it checks if the sequence number in the received frame is the same as the value of Frame-expected in the receiver's side. If so, the receiver will accept this frame and pass the

message to its host. The receiver will also advance the value of Frame-expected. Finally, the receiver has to acknowledge this received frame. When the sender sees that an outgoing frame has been acknowledged, it advances the Next-frame-to-send.

A one bit sliding window protocol is, in fact, stop-and-wait. Both Next-frame-to-send and Frame-expected may only be 0 or 1. Thus, after sending out an outgoing frame, the sender has to wait until the acknowledgment is received before sending out the next frame.

Piggyback acknowledgment.

In order not to waste time to send out a frame with no message but the only acknowledgment signal, the acknowledgment is hooked onto the header of the next outgoing data frame. However, there may not be any outgoing frame when a processor has received a frame. Thus, we may need to send a frame with no message but the acknowledgment signal to inform the sender in this case.

Timers.

To be able to recover from error and to make the communication subsystem run smoothly, we make use of three timers. They are transmitter timer, receiver timer and

acknowledgment timer.

When a sender sends out an outgoing frame, it turns on the transmitter timer. The acknowledgment of this outgoing frame is expected to be received before the transmit-time-out occurs. Thus, when the transmit-time-out occurs, the sender will retransmit the previous outgoing frame again if the acknowledgment has not been received yet.

When a receiver has received the first byte of a data frame, it turns on the receiver timer. In case that the receiver cannot receive the whole data frame before the receiver-time-out occurs, the receiver will reopen the physical receiving channel and to wait for the sender to send this frame once again.

When a data frame is received correctly, the receiver will turn on the acknowledgment timer. In case that there is no outgoing frame before the acknowledgment-time-out occurs, the receiver will send out a special acknowledgment frame to the sender.

The length of time set for these timers is an important factor in keeping the CSS running smoothly. For instance, the time length of the transmitter timer should be greater than the expected amount of actual transmission time for two messages of maximum length. Thus, we can be sure that the

acknowledgment can reach the sender before the sender's transmitter timer times out.

Routing layer.

The routing layer decides when to send a user's message. Synchronization between buddies is done in this layer. In order to decide whether a user's message should be sent or not, the routing layer uses a special channel (channel 0) to communicate with the routing layer of its buddies. The messages that are sent via channel zero are listed as follows:

1. Request to send a message.

2. Request to wait before sending a message.

3. Give permission to send a message.

When a sender wants to send a message, the routing layer will first send the message 'request to send a message' to the receiver. The receiver will then return the message 'give permission to send a message' after its host has informed it to receive a message. Only when the sender has received the message 'give permission to send a message' will the routing layer of the sender decide to send the

message to the receiver.

The following example shows what the routing layer does in order to send a message from User1 to User2.

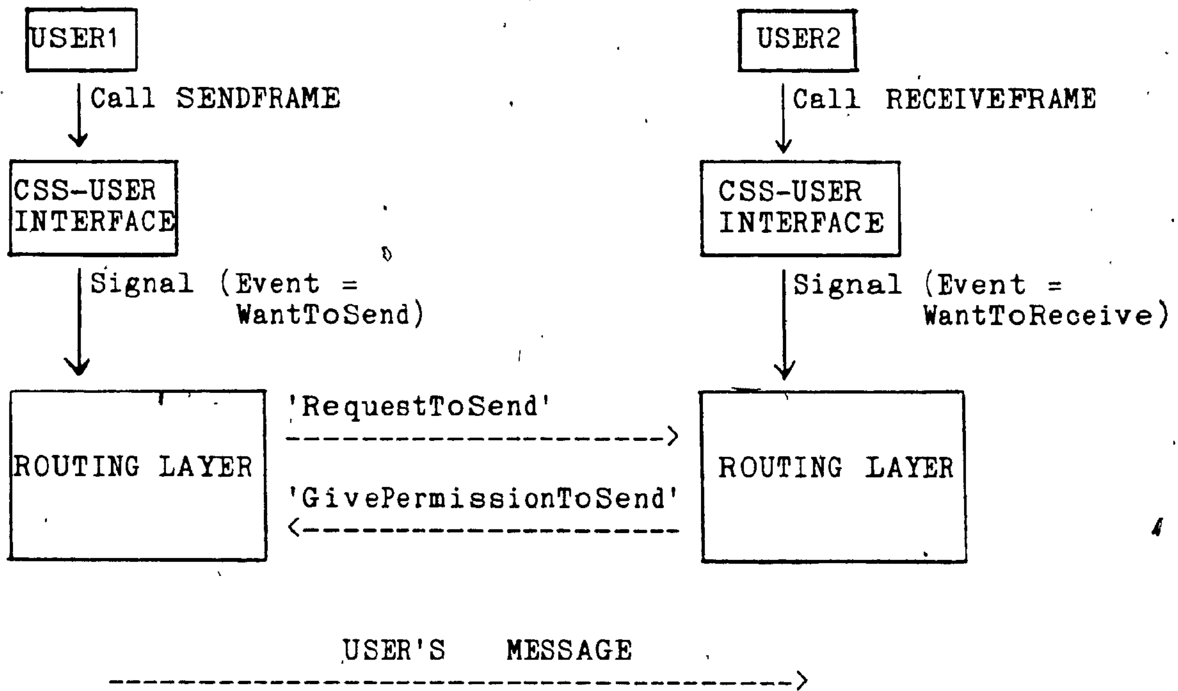

Figure 7.3. Functions of the Routing layer.

Layers Synchronization.

Integer semaphores are used to synchronize the operations of the layers in the CSS. There are four major semaphores used in the CSS.

1. Datalinksig : to wake up the datalink layer.

2. Routingsig : to wake up the routing layer.

3. Receiveready : to unblock the receiver when the message is received and thus the receiver can pick up the received message.

4. Transmitready : to unblock the sender after the message has been sent.

Neither the datalink layer nor the routing layer are active until a semaphore signal has been received. Since an integer semaphore is used, no signal will be lost even if several signals are sent to a layer before this layer gets the chance to handle them. Moreover, since both the datalink and the routing layers handle several different events, an event set is associated with the 'Datalinksig' and to the 'Routingsig'. When someone wants to signal these layers, it should also specify what the event is in the event set so that the routing layer or the datalink layer will know what event it is.

Figure 7.4. shows what each layer does in order to pass a data frame from a user to the other.

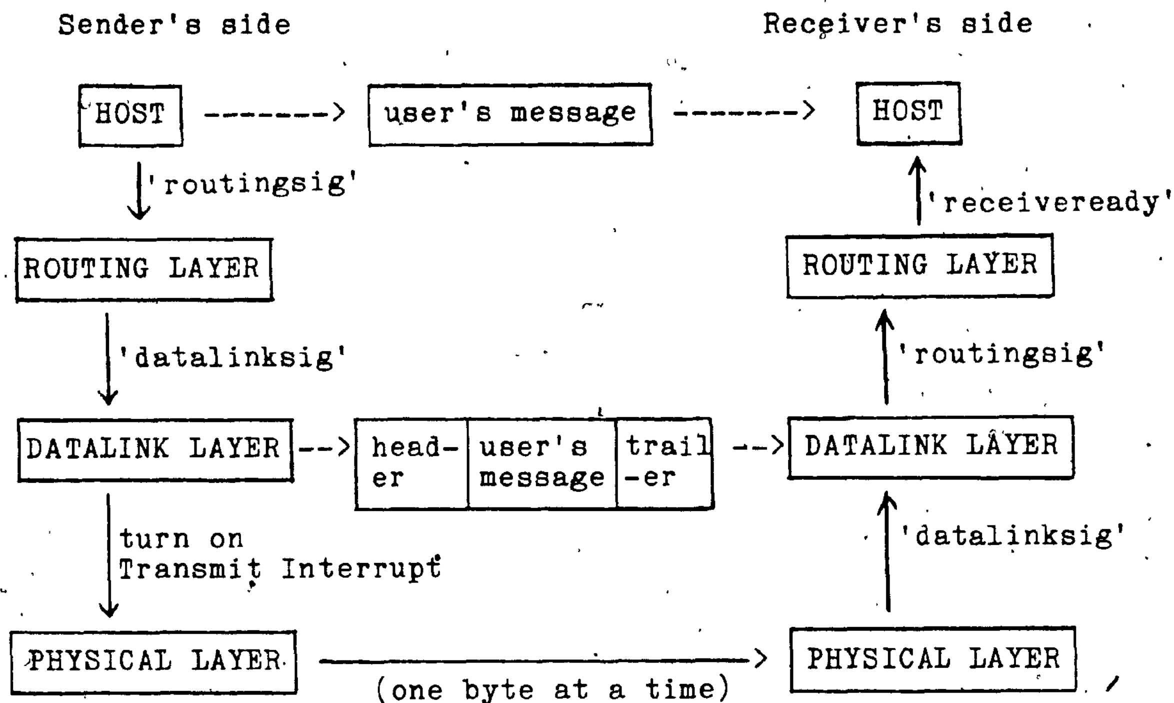

Figure 7.4. Passing a data frame from a sender to a receiver.

Performance.

We have recorded the transmission speed of the CSS by executing a program which opens two bidirectional channels. It sends and receives a certain number of messages to and from its buddy simultaneously via each channel. Connecting the processor PDP-11/34 and the processor LSI-11/73 by a serial line with 9600 baud rate, we have recorded that the transmission spped is 673 bytes/second (including the overhead). The overhead includes the header and the trailer

in each data frame as well as the messages passing through channel zero.

CHAPTER 8. TESTING THE RTS AND RUNNING A PASCAL-C PROGRAM.

In this chapter, we first describe how the run-time system was tested and then we describe how a Pascal-C program is actually run on the system.

Section 1. Testing the Run-time system.

To debug the run-time system itself, we focused on testing the correctness of the implementation of each of the Pascal-C feature to be discussed below. We wrote up a set of test programs to test the RTS. Some of these testing programs are simple and designed to test one Pascal-C feature at a time while some others are more complicated and may trigger many different events. Since Pascal-C is a concurrent programming language, the timing of the events may get the system into differnet situations. We describe the things that were tested for each Pascal-C feature.

1. Down procedures.

We tested the downloading and uploading of data (including value parameters and variable parameters) for a DP. We use different parameters which occupied one byte, one word or more than 128 bytes (since we had set the maximum length of a data frame to be 128 bytes).

To test the actual variable updating, we have done the following:

a. Invoke a DP several times with the same actual parameter.

b. Invoke a DP several times with different actual parameters.

2. Wait(DP) statement.

We have tested a Wait(DP) statement under the following situations:

a. The class of DP is not active at all.

b. At least one slave processor is executing this class of DP.

c. The class of DP is active but no slave is currently executing it.

d. The class of DP has been terminated.

3. Terminate(DP) statement.

We have tested this statement both within a CP and

within a normal procedure of the main program under the following situations:

a. The class of DP is not active at all.

b. The class of DP is active but no slave is executing it.

c. At least one slave is executing this DP.

Moreover, we have brought the system into the situation in which the master is sending a message to the slave to terminate the current job while the slave has just finished the current job and is trying to inform the master about it.

4. Critical procedures.

We have tested a CP which was invoked by both a master and a slave. In order to test the implementation of a critical section, one of the testing CP was itself a recursive procedure. We have also tested a CP invoked by a slave under the following conditions:

a. The critical section is not occupied.

b. The master is in the critical section and activating a DP.

c. A CP which is invoked by the master is being executed.

d. A CP which is invoked by another slave process is being executed.

5. Copy section.

Implementation of the copy section is done by the Pascal-C Preprocessor and is not described here.

Besides, we tested how the Pascal-C restrictions were checked during run time. For example, an attempt to download variable parameters which are in the heap area of the memory or an attempt to leave the environment where the actual parameters of an active DP is declared, will be reported as a run time error by the RTS.

Section 2. Running program on the Pascal-C system.

In this section, Part 1 describes the output of the Pascal-C Preprocessor. Part 2 describes the steps required in order to run a Pascal-C program.

Pascal-C Preprocessor Output.

Since a Pascal-C program is indeed executed by both the master processor and the slave processors, for each Pascal-C program, the Pascal-C preprocessor will generate different files needed by the master processor and the slave processors.

In order to run several different down procedures in a slave processor, we make use of the overlay handler of the RT-11 system. Since the down procedures can never be run simultaneously by a single slave processor, they can occupy the same overlayed area of memory. When a down procedure is called, the master supplies the ID of the down procedure to the slave who will execute it. The Slave Pascal-C RTS will call the desired down procedure and the overlay handler will load in the code of the desired DP. Thus, for each slave processor, we need to have a main program, which is for starting execution of the DPs only, and the code for each of the down procedures. Figure 8.1 shows an example of the slave main program. In this example, there are 2 down procedures (DP1 and DP2) and their IDs are 1 and 2 respectively. Procedures in the overlayed area are declared as external procedure to the main program in Parallel Pascal.

```
Program Slavemain;
  .
  .
var DPID:integer;
  .
  .
Procedure RTSCALL (var DPID:integr); external;
procedure DP1;external;
procedure DP2; external;
  .
  .
repeat
 (*wait for the master's instruction to run a DP*)
   RTSCALL (DPID);
 (*RTS will return the ID number of the desired DP*)
 case DPID of
  1: DP1;
  2: DP2;
 end; (*case*)
until DPID=-1;
  .
  .
```

Figure 8.1. Example of slave main program

In summary, the Pascal-C preprocessor generates the following files:

1. PCMAST : the master program which contains all the code executed by the master.

2. PCSLAV : the slave main program which is only a shell used to call the actual down procedure.

3. PCDOWN : all the modules for down procedures. For each down procedure in the Pascal-C program, the preprocessor generates an output module which is an external procedure to the main program of the slave processor and will be executed by the slave processor when needed.

Steps needed to follow when running a Pascal-C program.

1. Use the preprocessor to preprocess the Pascal-C program. If no Pascal-C syntax error is detected, the preprocessor will generate the files mentioned above.

2. Use the Parallel Pascal compiler to compile and generate object models for PCMAST (the master program), PCSLAV (the slave main program) and for each down procedure.

3. Use the linker to generate the executable files for both the master and the slave. A command file has

already been set up. Please read the instruction file 'RTS.INS' which is in the LSI-11/73 if one wants to run the Pascal-C program with the current system.

4. Start running program in both master and slave side.

CHAPTER 9. CONCLUSION.

In this thesis, I have described 1) the overview of our multi-processing project specific to solving combinatorial problems, 2) the concurrent programming language, Pascal-C and 3) the implementation details of the Pascal-C Run-time system.

There are several existing multi-processors systems and concurrent programming languages. Most of them are not specific to solving a particular type of problem. We have designed the architecture of our multi-processor system to be suitable for solving combinatorial problems. The language Pascal-C is suitable for writing programs to solve combinatorial problems in a hierarchical multi-processor system. However, more research work should be done to find out the performance of this multi-processors system.

By implementing the language Pascal-C and testing the Pascal-C system, we have strong confidence to say that it is a feasible language. Being a dialect of Pascal, Pascal-C is reasonably easy to use. From the viewpoint of the implementor, they need to implement only the Pascal-C features. The Standard Pascal features can be handled in a conventional way. Moreover, the implementation of the Pascal-C features can be done in a reasonable amount of time.

During the implementing of the first version of Pascal-C, we have concentrated on a two level multi-processors system. The charateristics (including the weaknesses) of this first version of the Pascal-C run-time system can be summarized as follows:

1. As much memory space as possible is reserved for the Pascal-C user. Since the computers we currently use have a 64k-byte memory, we have made use of the extended memory in the PDP-11/34 machine for temporarily storing the returned values of formal variable parameter of DP; we also made use of the overlay feature provided by the operating system RT-11 to keep those procedures which are not invoked frequently in the overlayed section.

2. Error diagonsis is reported as precisely as possible.

3. Each processor possesses a random access system device, the operating system RT-11 and a console terminal. This eliminates the step of downloading programs at run time and allows us to trace the actions of all the processors.

4. The run-time system is written in the language Parallel Pascal. However, most of the code is also

in Standard Pascal (e.g. the preprocessor-RTS interface is written in Standard Pascal). The code involving concurrency handling does not spread all over the modules. In the case that we implement the Pascal-C system in another dialect of Pascal with concurrency features, most of the code in this version can be kept.

5. The Master RTS may periodically poll its slaves to make sure that they are still 'alive'. No further error detection and recovery have been done. However, it does not create serious problems since we can use the consloe terminal to display error message or partial results.

In the future, error detection and recovery should be studied in depth. Some preliminary suggestions are discussed below:

1. From the experience gained from this first implementation, it is good to let each or some of the slaves possess an operating system, a console terminal and a random access system device. The user can then trace the actions of the slave processors when necessary. However, the programs should not be downloaded during run time. Transferring the programs to the slave processors

before run time simply puts the burden on the user.

2. When a slave processor detects the usual software problems such as overflow and range errors, it should report them to its master. This report should flow up to a master processor who possesses a console terminal, and this master processor should inform the user the failure.

Lastly, I wish that this first implementation of the Pascal-C run-time system is useful and the experiences gained from this implementation are helpful for the future development of this project.

Reference.

J. W. Atwood, S. Ganesan, M. Lafleur, W. Prager

'Measurement of the solution of two combinatorial Problems'

CIPS Session 84 1984

P. Brinch Hansen

The Architecture of Concurrent programming

Prentice-Hall 1977

P. Brinch Hansen

'Distributed Processes: A concurrent programming concept'

CACM, Vol.21 PP. 934-941 1978

D. R. Brownbridge

'Data Driven and Demand Driven Computer Architecture'

ACM Computing Surveys. Vol.14 PP. 93-143 1982

S. Cabilio

'M. Comp. Sci. Thesis' (in preparation)

Department of Computer Science, Concordia University 1985

R. P. Holt

'A short introduction to concurrent Euclid'

SIGPLAN Vol.17 PP. 60-79 1982

R. P. Cook

'*MOD - A language for distributed programming'

IEEE Proceeding of International Conference on Distributing Computing Systems 1979 PP. 233-241 1979

P. H. Enslow

'Multiprocessor Organization -- A Survey'

ACM Computing Survey Vol.9 PP. 103-130 1977

E. A. Feigenbaum

The Handbook of Artificial Intelligence , 1981

J. A. Feldman

'High level programming for Distributed Processing'

CACM Vol. 22 PP. 353-367 1979

C. Hewitt et al

'Security and Modularity in Message Passing'

IEEE Proceeding of International Conference on Distributing Computing System PP.347-358 1979

C. A. R. Hoare

'Communicating Sequential Processes'

CACM Vol. 21 PP. 666-677 1978

R. W. Hockney

Parallel Computers : Architecture, programming and algorithms 1981

E. Horowitz and A. Zorat

'Divide-and-conquer for parallel processing'

IEEE transactions of computers VC-32, No.6 PP. 582-584 1983

J. D. Ichbiah et al

'The Reference manual for the programming language ADA'

U. S. DOD 1980

K. Jones and P. Schwarz

'Experience using Multiprocessor System'

ACM Computing Surveys. Vol.12 PP. 121-165 1980

D. J. Kuck

'A survey of parallel machine organization and programming'

ACM Computing Surveys. Vol.9 PP. 76-87 1977

C. Lam et al

'Multiprocessor Project for Combinatorial Computing'

CIPS Session 82 PP. 325-329 1982

B. Liskov

'Primitive for distributed computing'

Proceeding of the 7th symposium on operating system principles 1979

H. H. Mashburn

'The C.mmp/Hydra Project : An architecture overview'

Computer Structures : Principles and Examples PP. 350-370 McGraw-Hill 1981

W. Morven Gentleman

'Message Passing between Sequential Processes : the Reply Primitive and the Adminstrator Concept'

Software-Pratice and Experience. Vol.11 PP. 435-466 1982

J. Opatrny

'Parallel Programming Constructures for Divide-and-Conquer Computing'

CIPS Session 84 1984

Parallel Pascal User's manual

Interactive Technology Incorporated, Portland, Oregon. 1982

P. D. Stotts Jr.

'A comparative survey of concurrent programming languages'

SIGPLAN Vol.17 No.9 PP. 76-87 1982

N. Wirth

'Design and Implementation of Modula'

Software-Pratice and Experience Vol.7 PP. 67-84 1977

Appendix 1. Example program.

This example program solves the KNAPSACK problem. It is written in Pascal-C and have been compiled and executed.

The Knapsack problem:

> Given an integer N, a finite sequence of integers A1,A2, ..., An, and an integer TARGET, find a subset CHOICE such that the sum of the A's selected by CHOICE is equal to TARGET.

```
PROGRAM KNAPSACK(INPUT,OUTPUT);
(* DEMONSTRATION PROGRAM FOR PARALLEL KNAPSACK
   STOPS AFTER FINDING MAXSOLN NUMBER OF SOLUTIONS, WHERE
   MAXSOLN IS A CONSTANT READ IN.
   ALL THE REQUIRED INPUT DATA ARE IN A FILE CWLKSK.DAT.
   THERE ARE TWO VERSIONS OF THE DOWN PROCEDURE PARKNAP1.
   VERSION 1 : RESULTS RETURNED VIA A CRITICAL PROCEDURE.
   VERSION 2 : RESULTS RETURNED VIA A VARIABLE PARAMETER.
*)

CONST
  MAXN=30;
  MAXSUBP = 32;

TYPE
  VECTOR = ARRAY [1..MAXN] OF INTEGER;
  WORD = SET OF 1..MAXN;
  SOLNVECT = ARRAY [1..MAXSUBP] OF INTEGER;

VAR
  A: VECTOR;
  CHOICE: WORD;
  I, J, N, SUM, LEVEL, TARGET, MAXSOLN :INTEGER;
  SOLCOUNT: INTEGER; (* COUNT OF NUMBER OF SOLUTION FOUND *)
  SUBPNO : INTEGER;  (* COUNT OF THE NUMBER OF SUBPROBLEM
                        GENERATED                            *)
  NOSOLNFOUND : SOLNVECT;  (* NOSOLNFOUND[I] = NUMBER OF
                              SOLUTION FOUND IN THE I-TH
                              SUBPROBLEM                     *)
  SOLNFOUND : INTEGER;    (* NUMBER OF SOLUTIONS FOUND BY
                             SLAVE                           *)
  OPTION : ARRAY[1..5] OF BOOLEAN;
         (* OPTION 1 : IF TRUE, USES VERSION 1 OF PARKNAP,
```

```
                        ELSE, USE VERSION 2                *)

PROCEDURE READINPUT;
VAR I: INTEGER;
    F: TEXT;
BEGIN
  RESET(F, 'CWLKSK.DAT');
  READ( F, N, TARGET, LEVEL, MAXSOLN,I);
  OPTION[1] := (I=1);
  FOR I:=1 TO N DO
    READ(F, A[I]);
END; (* READINPUT *)


PROCEDURE PRINTCHOICE(CHOICE: WORD);
VAR I,CNT : INTEGER;
BEGIN
  WRITE ('(');
  CNT := 0;
  FOR I:=1 TO N DO
  IF I IN CHOICE THEN
  BEGIN
    CNT := CNT + 1;
    WRITE(I:2);
    IF (CNT MOD 20) = 0 THEN WRITELN
    ELSE IF (CNT MOD 5) = 0 THEN WRITE('  ');
  END;
  WRITELN(')');
END; (* PRINTCHOICE *)


PROCEDURE PRINTSOLNVECT;
VAR I: INTEGER;
BEGIN
  FOR I:= 1 TO SUBPNO DO
  BEGIN
    WRITE (NOSOLNFOUND[I] : 5);
    IF (I MOD 10) = 0 THEN WRITELN
    ELSE IF (I MOD 5) = 0 THEN WRITE('   ');
  END;
  IF (I MOD 10)<>0 THEN WRITELN;
END;  (*  PRINTSOLNVECT  *)


PROCEDURE KNAP( I, SUM: INTEGER; CHOICE: WORD);
BEGIN
  IF (I <= N) AND (SUM < TARGET) THEN
  BEGIN
    KNAP(I+1, SUM, CHOICE);
    KNAP(I+1, SUM+A[I], CHOICE+[I]);
  END
```

```
  ELSE IF SUM =TARGET THEN
    SOLNFOUND:= SOLNFOUND+1;
END;


DOWN PROCEDURE PARKNAP1(SUM: INTEGER; CHOICE: WORD;
      CRITICAL PROCEDURE COUNTSOLUTION(SOLNFOUND: INTEGER));
COPY
  N, A, TARGET, LEVEL, SOLNFOUND,
  PRINTCHOICE , KNAP;


BEGIN (* MAIN LINE OF DOWN PROC. PARKNAP1 *)
  WRITELN(' NEW DOWN PROC, TARGET=',TARGET:5,
          ' SUM=',SUM:5,' CHOICE=');
  PRINTCHOICE(CHOICE);
  SOLNFOUND:=0;
  KNAP(LEVEL, SUM, CHOICE);
  WRITELN(' DOWN PROC FINISHED, SOLUTIONS FOUND =',
           SOLNFOUND:5);
  COUNTSOLUTION (SOLNFOUND);
END; (* DOWN PROC. PARKNAP *)


DOWN PROCEDURE PARKNAP2 (SUM : INTEGER ; CHOICE : WORD;
            SUBPID : INTEGER; VAR NOSOLNFOUND : SOLNVECT);
(* DOWN PROCEDURE TO SOLVE THE KNAPSACK PROBLEM AND RETURN
   THE NUMBER OF SOLNTIONS FOUND IN NOSOLNFOUND [SUBPID].
*)

COPY
  N, A, TARGET, LEVEL, SOLNFOUND,
  PRINTCHOICE, KNAP;

BEGIN   (* MAIN LINE OF DOWN PROCEDURE PARKNAP2  *)
  WRITELN(' NEW DOWN PROC, TARGET=',TARGET:5, ' SUM=',
          SUM:5, ' CHOICE=');
  PRINTCHOICE(CHOICE );
  SOLNFOUND := 0;
  KNAP(LEVEL, SUM, CHOICE);
  WRITELN(' DOWN PROC FINISHED, SOLUTIONS FOUND =',
          SOLNFOUND:5);
  NOSOLNFOUND [SUBPID] := SOLNFOUND;
END;    (* DOWN PROCEDURE PARKANNAP2  *)


CRITICAL PROCEDURE COUNTSOLUTION( SOLNFOUND: INTEGER);
(* CRITICAL PROCEDURE IN THE MASTER TO UPDATE THE NUMBER
   OF SOLUTIONS FOUND. THE PARAMETER SOLNFOUND IS THE
   NUMBER OF SOLUTIONS FOUND BY THE SLAVE.
*)
```

```
BEGIN
  SOLCOUNT:=SOLCOUNT+SOLNFOUND;
  WRITELN(' SLAVE REPORTED FINDING', SOLNFOUND:5,
          ' SOLUTIONS');
    IF SOLCOUNT> MAXSOLN THEN TERMINATE(PARKNAP1);

END; (* CRIT. PROC. COUNTSOLUTION *)


PROCEDURE INITIAL(I, SUM: INTEGER; CHOICE: WORD);
BEGIN
  IF I < LEVEL THEN
  BEGIN
    INITIAL(I+1, SUM, CHOICE);
    INITIAL(I+1, SUM+A[I], CHOICE+[I]);
  END
  ELSE BEGIN
    SUBPNO := SUBPNO + 1;
    IF OPTION[1] THEN PARKNAP1(SUM, CHOICE, COUNTSOLUTION)
    ELSE PARKNAP2 (SUM, CHOICE, SUBPNO, NOSOLNFOUND);
    WRITELN(' SUBPROGRAM',SUBPNO :4,' GENERATED, CHOICE:');
    PRINTCHOICE(CHOICE);
  END;
END;

BEGIN (* MASTER KNAPSACK *)
  READINPUT;
  SOLCOUNT:= 0;
  SUBPNO := 0;
  I:=1; SUM:=0; CHOICE:=[];
  INITIAL(I, SUM, CHOICE);
  IF OPTION[1] THEN BEGIN
    WAIT(PARKNAP1);
    WRITELN(' PROGRAM DONE.',SOLCOUNT:5,' SOLUTIONS FOUND');
  END ELSE BEGIN
    WAIT(PARKNAP2);
    WRITELN(' PROGRAM DONE, NUMBER OF SOLUTIONS',
            ' REPORTED BY SLAVES ARE');
    PRINTSOLNVECT ;
  END;
END.
```

Pascal-C Preprocessor Output for this esxample program

1. PCMAST : program in the master side

```
(*THIS IS A PARALLEL PASCAL MASTER PROGRAM TRANSLATED FROM A PASCAL-C SOURCE
  PROGRAM. TO CREATE A RUNNABLE MASTER PROGRAM, USE THE PARALLEL PASCAL
  COMPILER ON THIS FILE. THE SLAVE'S MAIN PROGRAM AND DOWN PROCEDURE(S)
  ARE PRODUCED AS SEPARATE FILES  BY THE PREPROCESSOR.*)

PROGRAM KNAPSACK(INPUT,OUTPUT);
VAR ZZRTSBUF:ARRAY [1..1500] OF INTEGER;(*FOR PASCAL-C RTS*)

(* DEMONSTRATION PROGRAM FOR PARALLEL KNAPSACK
   STOPS AFTER FINDING MAXSOLN NUMBER OF SOLUTIONS, WHERE
   MAXSOLN IS A CONSTANT READ IN.
   ALL THE REQUIRED INPUT DATA ARE IN A FILE CWLKSK.DAT.
   THERE ARE TWO VERSIONS OF THE DOWN PROCEDURE PARKNAP1.
   VERSION 1 : RESULTS RETURNED VIA A CRITICAL PROCEDURE.
   VERSION 2 : RESULTS RETURNED VIA A VARIABLE PARAMETER.
*)

CONST
  MAXN=30;
  MAXSUBP = 32;

TYPE
  VECTOR = ARRAY [1..MAXN] OF INTEGER;
  WORD = SET OF 1..MAXN;
  SOLNVECT = ARRAY [1..MAXSUBP] OF INTEGER;

VAR
  A: VECTOR;
  CHOICE: WORD;
  I, J, N, SUM, LEVEL, TARGET, MAXSOLN :INTEGER;
  SOLCOUNT: INTEGER; (* COUNT OF NUMBER OF SOLUTION FOUND *)
  SUBPNO : INTEGER;  (* COUNT OF THE NUMBER OF SUBPROBLEM
                        GENERATED                       *)
```

```
  NOSOLNFOUND : SOLNVECT;  (* NOSOLNFOUND[I] = NUMBER OF
                              SOLUTION FOUND IN THE I-TH
                              SUBPROBLEM  *)
  SOLNFOUND : INTEGER;     (* NUMBER OF SOLUTIONS FOUND BY
                              SLAVE                        *)
  OPTION : ARRAY[1..5] OF BOOLEAN;
          (* OPTION 1 : IF TRUE, USES VERSION 1 OF PARKNAP,
                        ELSE, USE VERSION 2                *)


PROCEDURE READINPUT;
VAR I: INTEGER;
    F: TEXT;
BEGIN
  RESET(F, 'CWLKSK.DAT');
  READ( F, N, TARGET, LEVEL, MAXSOLN,I);
  OPTION[1] := (I=1);
  FOR I:=1 TO N DO
    READ(F, A[I]);
END; (* READINPUT *)


PROCEDURE PRINTCHOICE(CHOICE: WORD);
VAR I,CNT : INTEGER;

PROCEDURE ZZPRO1(ZZACTION:INTEGER;VAR ZZDATA:WORD);
PROCEDURE RTS2(ZZACTION:INTEGER;VAR ZZDATA:WORD);EXTERNAL;
BEGIN RTS2(ZZACTION,ZZDATA)END;
BEGIN
  WRITE ('(');
  CNT := 0;
  FOR I:=1 TO N DO
  IF I IN CHOICE THEN
  BEGIN
    CNT := CNT + 1;
    WRITE(I:2);
    IF (CNT MOD 20) = 0 THEN WRITELN
```

```
    ELSE IF (CNT MOD 5) = 0 THEN WRITE(' . ');
  END;
  WRITELN(')');
;ZZPRO1(35,CHOICE); (*CHECK FOR ACTUAL VAR PARAMS TO DOWN PROCS*)
END; (* PRINTCHOICE *)


PROCEDURE PRINTSOLNVECT;
VAR I: INTEGER;
BEGIN
  FOR I:= 1 TO SUBPNO DO
  BEGIN
    WRITE (NOSOLNFOUND[I] : 5);
    IF (I MOD 10) = 0 THEN WRITELN
    ELSE IF (I MOD 5) = 0 THEN WRITE('    ');
  END;
  IF (I MOD 10)<>0 THEN WRITELN;
END;  (*  PRINTSOLNVECT  *)


PROCEDURE KNAP( I, SUM: INTEGER; CHOICE: WORD);

PROCEDURE ZZPRO1(ZZACTION:INTEGER;VAR ZZDATA:INTEGER);
PROCEDURE RTS2(ZZACTION:INTEGER;VAR ZZDATA:INTEGER);EXTERNAL;
BEGIN RTS2(ZZACTION,ZZDATA)END;
BEGIN
  IF (I <= N) AND (SUM < TARGET) THEN
  BEGIN
    KNAP(I+1, SUM, CHOICE);
  KNAP(I+1, SUM+A[I], CHOICE+[I]);
  END
  ELSE IF SUM =TARGET THEN
    SOLNFOUND:= SOLNFOUND+1;
;ZZPRO1(35,I); (*CHECK FOR ACTUAL VAR PARAMS TO DOWN PROCS*)
END;
```

```
(*PASCAL-C RTS DEFINITIONS*)
PROCEDURE RTS1 (ACTION:INTEGER); EXTERNAL;
PROCEDURE RTS2 (ACTION,DATA:INTEGER); EXTERNAL;
PROCEDURE RTS3 (ACTION,DATA1,DATA2:INTEGER); EXTERNAL;
FUNCTION RTS4 (ACTION,DATA:INTEGER):BOOLEAN; EXTERNAL;
(*END RTS DEFINITIONS*)

 PROCEDURE PARKNAP1(SUM: INTEGER; CHOICE: WORD;
       PROCEDURE COUNTSOLUTION(SOLNFOUND: INTEGER));
(*RTS REDEFINITIONS TO DEFEAT TYPE CHECKING*)
PROCEDURE ZZPRO1(ZZACTION:INTEGER;VAR ZZDATA:VECTOR;ZZSIZE:INTEGER);
PROCEDURE RTS3(ZZACTION:INTEGER;VAR ZZDATA:VECTOR;ZZSIZE:INTEGER);EXTERNAL;
BEGIN RTS3(ZZACTION,ZZDATA,ZZSIZE)END;
PROCEDURE ZZPRO2(ZZACTION:INTEGER;VAR ZZDATA:INTEGER;ZZSIZE:INTEGER);
PROCEDURE RTS3(ZZACTION:INTEGER;VAR ZZDATA:INTEGER;ZZSIZE:INTEGER);EXTERNAL;
BEGIN RTS3(ZZACTION,ZZDATA,ZZSIZE)END;
PROCEDURE ZZPRO3(ZZACTION:INTEGER;VAR ZZDATA:INTEGER;ZZSIZE:INTEGER);
PROCEDURE RTS3(ZZACTION:INTEGER;VAR ZZDATA:INTEGER;ZZSIZE:INTEGER);EXTERNAL;
BEGIN RTS3(ZZACTION,ZZDATA,ZZSIZE)END;
PROCEDURE ZZPRO4(ZZACTION:INTEGER;VAR ZZDATA:INTEGER;ZZSIZE:INTEGER);
PROCEDURE RTS3(ZZACTION:INTEGER;VAR ZZDATA:INTEGER;ZZSIZE:INTEGER);EXTERNAL;
BEGIN RTS3(ZZACTION,ZZDATA,ZZSIZE)END;
PROCEDURE ZZPRO5(ZZACTION:INTEGER;VAR ZZDATA:INTEGER;ZZSIZE:INTEGER);
PROCEDURE RTS3(ZZACTION:INTEGER;VAR ZZDATA:INTEGER;ZZSIZE:INTEGER);EXTERNAL;
BEGIN RTS3(ZZACTION,ZZDATA,ZZSIZE)END;
PROCEDURE ZZPRO6(ZZACTION:INTEGER;VAR ZZDATA:INTEGER;ZZSIZE:INTEGER);
PROCEDURE RTS3(ZZACTION:INTEGER;VAR ZZDATA:INTEGER;ZZSIZE:INTEGER);EXTERNAL;
BEGIN RTS3(ZZACTION,ZZDATA,ZZSIZE)END;
PROCEDURE ZZPRO7(ZZACTION:INTEGER;VAR ZZDATA:WORD;ZZSIZE:INTEGER);
PROCEDURE RTS3(ZZACTION:INTEGER;VAR ZZDATA:WORD;ZZSIZE:INTEGER);EXTERNAL;
BEGIN RTS3(ZZACTION,ZZDATA,ZZSIZE)END;
PROCEDURE ZZPRO8(ZZACTION:INTEGER;PROCEDURE ZZCRIT(SOLNFOUND:INTEGER));
PROCEDURE RTS5(ZZACTION:INTEGER;PROCEDURE ZZCRIT(SOLNFOUND:INTEGER));EXTERNAL;
BEGIN RTS5(ZZACTION,ZZCRIT)END;
(*END RTS REDEFINITIONS*)
```

```
BEGIN
IF RTS4(96,1)THEN BEGIN RTS3(66,1,0);(*NEW ACTIVATION UNLESS TERMINATED*)
ZZPRO1(64,A,60); (*SEND COPY OF VARIABLE TO SLAVE*)
ZZPRO2(64,N,2); (*SEND COPY OF VARIABLE TO SLAVE*)
ZZPRO3(64,LEVEL,2); (*SEND COPY OF VARIABLE TO SLAVE*)
ZZPRO4(64,TARGET,2); (*SEND COPY OF VARIABLE TO SLAVE*)
ZZPRO5(64,SOLNFOUND,2); (*SEND COPY OF VARIABLE TO SLAVE*)
ZZPRO6(64,SUM,2); (*SEND COPY OF VARIABLE TO SLAVE*)
ZZPRO7(64,CHOICE,12); (*SEND COPY OF VARIABLE TO SLAVE*)
ZZPRO8(128,COUNTSOLUT); (*SEND CRIT PROC INFO TO SLAVE*)
RTS1(16)END; (*END DOWNLOADING*)
END; (* DOWN PROC. PARKNAP *)


 PROCEDURE PARKNAP2 (SUM : INTEGER ; CHOICE : WORD;
            SUBPID : INTEGER; VAR NOSOLNFOUND : SOLNVECT);
(*RTS REDEFINITIONS TO DEFEAT TYPE CHECKING*)
PROCEDURE ZZPRO1(ZZACTION:INTEGER;VAR ZZDATA:VECTOR;ZZSIZE:INTEGER);
PROCEDURE RTS3(ZZACTION:INTEGER;VAR ZZDATA:VECTOR;ZZSIZE:INTEGER);EXTERNAL;
BEGIN RTS3(ZZACTION,ZZDATA,ZZSIZE)END;
PROCEDURE ZZPRO2(ZZACTION:INTEGER;VAR ZZDATA:INTEGER;ZZSIZE:INTEGER);
PROCEDURE RTS3(ZZACTION:INTEGER;VAR ZZDATA:INTEGER;ZZSIZE:INTEGER);EXTERNAL;
BEGIN RTS3(ZZACTION,ZZDATA,ZZSIZE)END;
PROCEDURE ZZPRO3(ZZACTION:INTEGER;VAR ZZDATA:INTEGER;ZZSIZE:INTEGER);
PROCEDURE RTS3(ZZACTION:INTEGER;VAR ZZDATA:INTEGER;ZZSIZE:INTEGER);EXTERNAL;
BEGIN RTS3(ZZACTION,ZZDATA,ZZSIZE)END;
PROCEDURE ZZPRO4(ZZACTION:INTEGER;VAR ZZDATA:INTEGER;ZZSIZE:INTEGER);
PROCEDURE RTS3(ZZACTION:INTEGER;VAR ZZDATA:INTEGER;ZZSIZE:INTEGER);EXTERNAL;
BEGIN RTS3(ZZACTION,ZZDATA,ZZSIZE)END;
PROCEDURE ZZPRO5(ZZACTION:INTEGER;VAR ZZDATA:INTEGER;ZZSIZE:INTEGER);
PROCEDURE RTS3(ZZACTION:INTEGER;VAR ZZDATA:INTEGER;ZZSIZE:INTEGER);EXTERNAL;
BEGIN RTS3(ZZACTION,ZZDATA,ZZSIZE)END;
PROCEDURE ZZPRO6(ZZACTION:INTEGER;VAR ZZDATA:INTEGER;ZZSIZE:INTEGER);
PROCEDURE RTS3(ZZACTION:INTEGER;VAR ZZDATA:INTEGER;ZZSIZE:INTEGER);EXTERNAL;
BEGIN RTS3(ZZACTION,ZZDATA,ZZSIZE)END;
PROCEDURE ZZPRO7(ZZACTION:INTEGER;VAR ZZDATA:WORD;ZZSIZE:INTEGER);
```

```
PROCEDURE RTS3(ZZACTION:INTEGER;VAR ZZDATA:WORD;ZZSIZE:INTEGER);EXTERNAL;
BEGIN RTS3(ZZACTION,ZZDATA,ZZSIZE)END;
PROCEDURE ZZPRO8(ZZACTION:INTEGER;VAR ZZDATA:INTEGER;ZZSIZE:INTEGER);
PROCEDURE RTS3(ZZACTION:INTEGER;VAR ZZDATA INTEGER;ZZSIZE:INTEGER);EXTERNAL;
BEGIN RTS3(ZZACTION,ZZDATA,ZZSIZE)END;
PROCEDURE ZZPRO9(ZZACTION:INTEGER;VAR ZZDATA:SOLNVECT;ZZSIZE:INTEGER);
PROCEDURE RTS3(ZZACTION:INTEGER;VAR ZZDATA:SOLNVECT;ZZSIZE:INTEGER);EXTERNAL;
BEGIN RTS3(ZZACTION,ZZDATA,ZZSIZE)END;
(*END RTS REDEFINITIONS*)

BEGIN
IF RTS4(96,2)THEN BEGIN RTS3(66,2,1);(*NEW ACTIVATION UNLESS TERMINATED*)
ZZPRO1(64,A,60); (*SEND COPY OF VARIABLE TO SLAVE*)
ZZPRO2(64,N,2); (*SEND COPY OF VARIABLE TO SLAVE*)
ZZPRO3(64,LEVEL,2); (*SEND COPY OF VARIABLE TO SLAVE*)
ZZPRO4(64,TARGET,2); (*SEND COPY OF VARIABLE TO SLAVE*)
ZZPRO5(64,SOLNFOUND,2); (*SEND COPY OF VARIABLE TO SLAVE*)
ZZPRO6(64,SUM,2); (*SEND COPY OF VARIABLE TO SLAVE*)
ZZPRO7(64,CHOICE,12); (*SEND COPY OF VARIABLE TO SLAVE*)
ZZPRO8(64,SUBPID,2); (*SEND COPY OF VARIABLE TO SLAVE*)
ZZPRO9(65,NOSOLNFOUN,64); (*SEND VAR PARAMETER TO SLAVE*)
RTS1(16)END; (*END DOWNLOADING*)
END;    (* DOWN PROCEDURE PARKANNAP2  *)


 PROCEDURE COUNTSOLUTION( SOLNFOUND: INTEGER);
(* CRITICAL PROCEDURE IN THE MASTER TO UPDATE THE NUMBER
   OF SOLUTIONS FOUND. THE PARAMETER SOLNFOUND IS THE
   NUMBER OF SOLUTIONS FOUND BY THE SLAVE.
*)
BEGIN
  SOLCOUNT:=SOLCOUNT+SOLNFOUND;
  WRITELN(' SLAVE REPORTED FINDING', SOLNFOUND:5,
          ' SOLUTIONS');
    IF SOLCOUNT> MAXSOLN THEN RTS2(33,1) (*TERMINATE (PARKNAP1)*)
;
```

```
END; (* CRIT. PROC. COUNTSOLUTION *)


PROCEDURE INITIAL(I, SUM: INTEGER; CHOICE: WORD);

PROCEDURE ZZPRO1(ZZACTION:INTEGER;VAR ZZDATA:INTEGER);
PROCEDURE RTS2(ZZACTION:INTEGER;VAR ZZDATA:INTEGER);EXTERNAL;
BEGIN RTS2(ZZACTION,ZZDATA)END;
BEGIN
  IF I < LEVEL THEN
  BEGIN
    INITIAL(I+1, SUM, CHOICE);
    INITIAL(I+1, SUM+A[I], CHOICE+[I]);
  END
  ELSE BEGIN
    SUBPNO := SUBPNO + 1;
    IF OPTION[1] THEN BEGIN RTS2(34,1);RTS1(Ø); (*RESERVE SLAVE & LOCK BEFORE DOWN
CALL*)
PARKNAP1(SUM, CHOICE, COUNTSOLUTION);RTS1(1)END (*UNLOCK AT END OF DOWN CALL*)

    ELSE BEGIN RTS2(34,2);RTS1(Ø); (*RESERVE SLAVE & LOCK BEFORE DOWN CALL*)
PARKNAP2 (SUM, CHOICE, SUBPNO, NOSOLNFOUND);RTS1(1)END (*UNLOCK AT END OF DOWN
CALL*)
;
    WRITELN(' SUBPROGRAM',SUBPNO :4,' GENERATED, CHOICE:');
    PRINTCHOICE(CHOICE);
  END;
;ZZPRO1(35,I); (*CHECK FOR ACTUAL VAR PARAMS TO DOWN PROCS*)
END;

BEGIN (*MASTER MAINLINE*)
RTS2(48,2); (*INITIALIZE MASTER RTS*)
 (* MASTER KNAPSACK *)
  READINPUT;
  SOLCOUNT:= Ø;
  SUBPNO := Ø;
```

```
  I:=1; SUM:=0; CHOICE:=[];
  INITIAL(I, SUM, CHOICE);
  IF OPTION[1] THEN BEGIN
    RTS2(32,1) (*WAIT (PARKNAP1)*)
;
    WRITELN(' PROGRAM DONE.',SOLCOUNT:5,' SOLUTIONS FOUND');
  END ELSE BEGIN
    RTS2(32,2) (*WAIT (PARKNAP2)*)
;
    WRITELN(' PROGRAM DONE, NUMBER OF SOLUTIONS',
           ' REPORTED BY SLAVES ARE');
    PRINTSOLNVECT ;
  END;
;RTS1(17) (* END PROGRAM *)
END.
```

2. PCSLAV : main program in slave side

```
(*MAIN PROGRAM FOR SLAVE. PASCAL-C DOWN PROCEDURES ARE LINKED AS EXTERNAL
  (POSSIBLY OVERLAID) PARALLEL PASCAL PROCEDURES RUN BY THIS PROGRAM.*)
VAR RTS:ARRAY [1..1500] OF INTEGER;(*FOR PASCAL-C RTS*)
    DOWNID:INTEGER;
PROCEDURE DP1;EXTERNAL;
PROCEDURE DP2;EXTERNAL;
PROCEDURE ZZSTAT;EXTERNAL;
PROCEDURE RTS2(X,Y:INTEGER);EXTERNAL;
PROCEDURE RTS6(X:INTEGER;VAR Y:INTEGER);EXTERNAL;
BEGIN
 RTS2(48,0); (* INITIALIZE SLAVE RTS *)
 ZZSTAT; (* SAVE STATE TO RESTORE AFTER TERMINATION *)
 REPEAT
  RTS6(160,DOWNID); (* WAIT FOR DOWN PROC REQUEST *)
  CASE DOWNID OF
   1:DP1;
   2:DP2;
   OTHERWISE
  END;
 UNTIL DOWNID = -1
END.
```

3. PCDOWN : down procedures in slave side

```
(* DOWN PROCEDURE PARKNAP1 *)

(*$E*)PROCEDURE DP1;
(*THIS IS A PARALLEL PASCAL EXTERNAL PROCEDURE, TRANSLATED FROM A DOWN
  PROCEDURE IN A PASCAL-C SOURCE PROGRAM. THIS MODULE SHOULD BE COMPILED BY
  PARALLEL PASCAL AND LINKED (POSSIBLY OVERLAYING OTHER SIMILAR MODULES) TO
  THE MAIN PROGRAM FOR THE SLAVE.*)

CONST
MAXN=30;
TYPE
VECTOR = ARRAY [1..MAXN] OF INTEGER;
TYPE
WORD = SET OF 1..MAXN;
VAR A:VECTOR;
VAR N:INTEGER;
VAR LEVEL:INTEGER;
VAR TARGET:INTEGER;
VAR SOLNFOUND:INTEGER;
PROCEDURE
PRINTCHOICE(CHOICE: WORD);
VAR I,CNT : INTEGER;
BEGIN
  WRITE ('(');
  CNT := 0;
  FOR I:=1 TO N DO
  IF I IN CHOICE THEN
  BEGIN
    CNT := CNT + 1;
    WRITE(I:2);
    IF (CNT MOD 20) = 0 THEN WRITELN
    ELSE IF (CNT MOD 5) = 0 THEN WRITE('  ');
  END;
  WRITELN(')');
END;
```

```
PROCEDURE
KNAP( I, SUN: INTEGER; CHOICE: WORD);
BEGIN
  IF (I <= N) AND (SUM < TARGET) THEN
  BEGIN
    KNAP(I+1, SUM, CHOICE);
  KNAP(I+1, SUM+A[I], CHOICE+[I]);
  END
  ELSE IF SUM =TARGET THEN
    SOLNFOUND:= SOLNFOUND+1;
END;
(*RTS REDEFINITIONS TO DEFEAT TYPE CHECKING*)
PROCEDURE ZZPRO1(ZZACTION:INTEGER;VAR ZZDATA:VECTOR;ZZSIZE:INTEGER);
PROCEDURE RTS3(ZZACTION:INTEGER;VAR ZZDATA:VECTOR;ZZSIZE:INTEGER);EXTERNAL;
BEGIN RTS3(ZZACTION,ZZDATA,ZZSIZE)END;
PROCEDURE ZZPRO2(ZZACTION:INTEGER;VAR ZZDATA:INTEGER;ZZSIZE:INTEGER);
PROCEDURE RTS3(ZZACTION:INTEGER;VAR ZZDATA:INTEGER;ZZSIZE:INTEGER);EXTERNAL;
BEGIN RTS3(ZZACTION,ZZDATA,ZZSIZE)END;
PROCEDURE ZZPRO3(ZZACTION:INTEGER;VAR ZZDATA:INTEGER;ZZSIZE:INTEGER);
PROCEDURE RTS3(ZZACTION:INTEGER;VAR ZZDATA:INTEGER;ZZSIZE:INTEGER);EXTERNAL;
BEGIN RTS3(ZZACTION,ZZDATA,ZZSIZE)END;
PROCEDURE ZZPRO4(ZZACTION:INTEGER;VAR ZZDATA:INTEGER;ZZSIZE:INTEGER);
PROCEDURE RTS3(ZZACTION:INTEGER;VAR ZZDATA:INTEGER;ZZSIZE:INTEGER);EXTERNAL;
BEGIN RTS3(ZZACTION,ZZDATA,ZZSIZE)END;
PROCEDURE ZZPRO5(ZZACTION:INTEGER;VAR ZZDATA:INTEGER;ZZSIZE:INTEGER);
PROCEDURE RTS3(ZZACTION:INTEGER;VAR ZZDATA:INTEGER;ZZSIZE:INTEGER);EXTERNAL;
BEGIN RTS3(ZZACTION,ZZDATA,ZZSIZE)END;
(*END RTS REDEFINITIONS*)

PROCEDURE DOWN;
(*BEGIN DECLARATIONS FOR DOWN PROCEDURE PARAMETERS*)
VAR SUM:INTEGER;(*VALUE PARAMETER*)
VAR CHOICE:WORD;(*VALUE PARAMETER*)
VAR ZZLIN8,ZZENT8:INTEGER; (*STAT LINK & ENTRY PT FOR CRIT PROC IN MASTER*)
(*END OF DECLARATIONS FOR DOWN PROCEDURE PARAMETERS*)
```

```
(*PASCAL-C RTS DEFINITIONS*)
PROCEDURE RTS1(ACTION:INTEGER);EXTERNAL;
PROCEDURE RTS3(ACTION:INTEGER;CHANGED:BOOLEAN;SIZE:INTEGER);EXTERNAL;
(*END RTS DEFINITIONS*)

(*RTS REDEFINITIONS TO DEFEAT TYPE CHECKING*)
PROCEDURE ZZPRO6(ZZACTION:INTEGER;VAR ZZDATA:INTEGER;ZZSIZE:INTEGER);
PROCEDURE RTS3(ZZACTION:INTEGER;VAR ZZDATA:INTEGER;ZZSIZE:INTEGER);EXTERNAL;
BEGIN RTS3(ZZACTION,ZZDATA,ZZSIZE)END;
PROCEDURE ZZPRO7(ZZACTION:INTEGER;VAR ZZDATA:WORD;ZZSIZE:INTEGER);
PROCEDURE RTS3(ZZACTION:INTEGER;VAR ZZDATA:WORD;ZZSIZE:INTEGER);EXTERNAL;
BEGIN RTS3(ZZACTION,ZZDATA,ZZSIZE)END;
PROCEDURE ZZPRO8(ZZACTION:INTEGER;VAR ZZDATA:INTEGER;ZZSIZE:INTEGER);
PROCEDURE RTS3(ZZACTION:INTEGER;VAR ZZDATA:INTEGER;ZZSIZE:INTEGER);EXTERNAL;
BEGIN RTS3(ZZACTION,ZZDATA,ZZSIZE)END;
PROCEDURE COUNTSOLUT(SOLNFOUND:INTEGER;ZZSIZE,ZZLINK,ZZENTRY:INTEGER);
PROCEDURE RTS3(ZZACTION:INTEGER;VAR ZZDATA:INTEGER;ZZSIZE:INTEGER);EXTERNAL;
BEGIN RTS3(76,ZZENTRY,ZZSIZE+6)END;
(*END RTS REDEFINITIONS*)

BEGIN (*DOWN PROCEDURE CORE*)
ZZPRO6(74,SUM,2); (*RECEIVE COPY OF VARIABLE FROM MASTER*)
ZZPRO7(74,CHOICE,12); (*RECEIVE COPY OF VARIABLE FROM MASTER*)
ZZPRO8(74,ZZLIN8,4); (*RECEIVE CRIT PROC INFO FROM MASTER*)
(* MAIN LINE OF DOWN PROC. PARKNAP1 *)
  WRITELN(' NEW DOWN PROC, TARGET=',TARGET:5,
          ' SUM=',SUM:5,' CHOICE=');
  PRINTCHOICE(CHOICE);
  SOLNFOUND:=0;
  KNAP(LEVEL, SUM, CHOICE);
  WRITELN(' DOWN PROC FINISHED, SOLUTIONS FOUND =',
          SOLNFOUND:5);
  COUNTSOLUTION (SOLNFOUND,2,ZZLIN8,ZZENT8)
;
RTS1(17)END; (* END DOWN PROCEDURE CORE *)
```

```
BEGIN (*REGION FOR COPY SECTION ITEMS*)
ZZPRO1(74,A,60); (*RECEIVE COPY OF VARIABLE FROM MASTER*)
ZZPRO2(74,N,2); (*RECEIVE COPY OF VARIABLE FROM MASTER*)
ZZPRO3(74,LEVEL,2); (*RECEIVE COPY OF VARIABLE FROM MASTER*)
ZZPRO4(74,TARGET,2); (*RECEIVE COPY OF VARIABLE FROM MASTER*)
ZZPRO5(74,SOLNFOUND,2); (*RECEIVE COPY OF VARIABLE FROM MASTER*)
DOWN END; (*END COPY SECTION REGION*)
(* DOWN PROCEDURE PARKNAP2 *)

(*$E*)PROCEDURE DP2;
(*THIS IS A PARALLEL PASCAL EXTERNAL PROCEDURE, TRANSLATED FROM A DOWN
  PROCEDURE IN A PASCAL-C SOURCE PROGRAM. THIS MODULE SHOULD BE COMPILED BY
  PARALLEL PASCAL AND LINKED (POSSIBLY OVERLAYING OTHER SIMILAR MODULES) TO
  THE MAIN PROGRAM FOR THE SLAVE.*)

CONST
MAXN=30;
CONST
MAXSUBP = 32;
TYPE
VECTOR = ARRAY [1..MAXN] OF INTEGER;
TYPE
WORD = SET OF 1..MAXN;
TYPE
SOLNVECT = ARRAY [1..MAXSUBP] OF INTEGER;
VAR A:VECTOR;
VAR N:INTEGER;
VAR LEVEL:INTEGER;
VAR TARGET:INTEGER;
VAR SOLNFOUND:INTEGER;
PROCEDURE
PRINTCHOICE(CHOICE: WORD);
VAR I,CNT : INTEGER;
BEGIN
  WRITE ('(');
  CNT := 0;
```

```
  FOR I:=1 TO N DO
  IF I IN CHOICE THEN
  BEGIN
    CNT := CNT + 1;
    WRITE(I:2);
    IF (CNT MOD 20) = 0 THEN WRITELN
    ELSE IF (CNT MOD 5) = 0 THEN WRITE('  ');
  END;
  WRITELN(')');
END;
PROCEDURE
KNAP( I, SUM: INTEGER; CHOICE: WORD);
BEGIN
  IF (I <= N) AND (SUM < TARGET) THEN
  BEGIN
    KNAP(I+1, SUM, CHOICE);
  KNAP(I+1, SUM+A[I], CHOICE+[I]);
  END
  ELSE IF SUM =TARGET THEN
    SOLNFOUND:= SOLNFOUND+1;
END;
(*RTS REDEFINITIONS TO DEFEAT TYPE CHECKING*)
PROCEDURE ZZPRO1(ZZACTION:INTEGER;VAR ZZDATA:VECTOR;ZZSIZE:INTEGER);
PROCEDURE RTS3(ZZACTION:INTEGER;VAR ZZDATA:VECTOR;ZZSIZE:INTEGER);EXTERNAL;
BEGIN RTS3(ZZACTION,ZZDATA,ZZSIZE)END;
PROCEDURE ZZPRO2(ZZACTION:INTEGER;VAR ZZDATA:INTEGER;ZZSIZE:INTEGER);
PROCEDURE RTS3(ZZACTION:INTEGER;VAR ZZDATA:INTEGER;ZZSIZE:INTEGER);EXTERNAL;
BEGIN RTS3(ZZACTION,ZZDATA,ZZSIZE)END;
PROCEDURE ZZPRO3(ZZACTION:INTEGER;VAR ZZDATA:INTEGER;ZZSIZE:INTEGER);
PROCEDURE RTS3(ZZACTION:INTEGER;VAR ZZDATA:INTEGER;ZZSIZE:INTEGER);EXTERNAL;
BEGIN RTS3(ZZACTION,ZZDATA,ZZSIZE)END;
PROCEDURE ZZPRO4(ZZACTION:INTEGER;VAR ZZDATA:INTEGER;ZZSIZE:INTEGER);
PROCEDURE RTS3(ZZACTION:INTEGER;VAR ZZDATA:INTEGER;ZZSIZE:INTEGER);EXTERNAL;
BEGIN RTS3(ZZACTION,ZZDATA,ZZSIZE)END;
PROCEDURE ZZPRO5(ZZACTION:INTEGER;VAR ZZDATA:INTEGER;ZZSIZE:INTEGER);
PROCEDURE RTS3(ZZACTION:INTEGER;VAR ZZDATA:INTEGER;ZZSIZE:INTEGER);EXTERNAL;
BEGIN RTS3(ZZACTION,ZZDATA,ZZSIZE)END;
```

```
(*END RTS REDEFINITIONS*)

PROCEDURE DOWN;
(*BEGIN DECLARATIONS FOR DOWN PROCEDURE PARAMETERS*)
VAR SUM:INTEGER;(*VALUE PARAMETER*)
VAR CHOICE:WORD;(*VALUE PARAMETER*)
VAR SUBPID:INTEGER;(*VALUE PARAMETER*)
VAR NOSOLNFOUN,ZZBUF9:SOLNVECT;(*VAR PARAM & COPY FOR UPDATE CHECK*)
ZZ9X1:1..32; (*INDEX TO SCAN FOR UPDATES IN ABOVE VAR PARAMETER*)
(*END OF DECLARATIONS FOR DOWN PROCEDURE PARAMETERS*)


(*PASCAL-C RTS DEFINITIONS*)
PROCEDURE RTS1(ACTION:INTEGER);EXTERNAL;
PROCEDURE RTS3(ACTION:INTEGER;CHANGED:BOOLEAN;SIZE:INTEGER);EXTERNAL;
(*END RTS DEFINITIONS*)

(*RTS REDEFINITIONS TO DEFEAT TYPE CHECKING*)
PROCEDURE ZZPRO6(ZZACTION:INTEGER;VAR ZZDATA:INTEGER;ZZSIZE:INTEGER);
PROCEDURE RTS3(ZZACTION:INTEGER;VAR ZZDATA:INTEGER;ZZSIZE:INTEGER);EXTERNAL;
BEGIN RTS3(ZZACTION,ZZDATA,ZZSIZE)END;
PROCEDURE ZZPRO7(ZZACTION:INTEGER;VAR ZZDATA:WORD;ZZSIZE:INTEGER);
PROCEDURE RTS3(ZZACTION:INTEGER;VAR ZZDATA:WORD;ZZSIZE:INTEGER);EXTERNAL;
BEGIN RTS3(ZZACTION,ZZDATA,ZZSIZE)END;
PROCEDURE ZZPRO8(ZZACTION:INTEGER;VAR ZZDATA:INTEGER;ZZSIZE:INTEGER);
PROCEDURE RTS3(ZZACTION:INTEGER;VAR ZZDATA:INTEGER;ZZSIZE:INTEGER);EXTERNAL;
BEGIN RTS3(ZZACTION,ZZDATA,ZZSIZE)END;
PROCEDURE ZZPRO9(ZZACTION:INTEGER;VAR ZZDATA:SOLNVECT;ZZSIZE:INTEGER);
PROCEDURE RTS3(ZZACTION:INTEGER;VAR ZZDATA:SOLNVECT;ZZSIZE:INTEGER);EXTERNAL;
BEGIN RTS3(ZZACTION,ZZDATA,ZZSIZE)END;
(*END RTS REDEFINITIONS*)

BEGIN (*DOWN PROCEDURE CORE*)
ZZPRO6(74,SUM,2); (*RECEIVE COPY OF VARIABLE FROM MASTER*)
```

```
ZZPRO7(74,CHOICE,12); (*RECEIVE COPY OF VARIABLE FROM MASTER*)
ZZPRO8(74,SUBPID,2); (*RECEIVE COPY OF VARIABLE FROM MASTER*)
ZZPRO9(75,NOSOLNFOUN,64); (*RECEIVE VAR PARAMETER FROM MASTER*)
ZZBUF9:=NOSOLNFOUN; (*SAVE COPY OF VAR PARAMETER TO DETECT CHANGES*)
   (* MAIN LINE OF DOWN PROCEDURE PARKNAP2  *)
  WRITELN(' NEW DOWN PROC, TARGET=',TARGET:5, ' SUM=',
         SUM:5, ' CHOICE=');
  PRINTCHOICE(CHOICE );
  SOLNFOUND := 0;
  KNAP(LEVEL, SUM, CHOICE);
  WRITELN(' DOWN PROC FINISHED, SOLUTIONS FOUND =',
         SOLNFOUND:5);
  NOSOLNFOUND [SUBPID] := SOLNFOUND;
;
ZZPRO9(72,NOSOLNFOUN,64); (*BEGIN UPDATE OF VAR PARAMETER*)
FOR ZZ9X1:=1 TO 32 DO BEGIN
RTS3(73,NOSOLNFOUN[ZZ9X1]<>ZZBUF9[ZZ9X1],2);(*CHECK FOR UPDATE*)
END;(*FOR ZZ9X1*)
RTS1(17)END; (* END DOWN PROCEDURE CORE *)

BEGIN (*REGION FOR COPY SECTION ITEMS*)
ZZPRO1(74,A,60); (*RECEIVE COPY OF VARIABLE FROM MASTER*)
ZZPRO2(74,N,2); (*RECEIVE COPY OF VARIABLE FROM MASTER*)
ZZPRO3(74,LEVEL,2); (*RECEIVE COPY OF VARIABLE FROM MASTER*)
ZZPRO4(74,TARGET,2); (*RECEIVE COPY OF VARIABLE FROM MASTER*)
ZZPRO5(74,SOLNFOUND,2); (*RECEIVE COPY OF VARIABLE FROM MASTER*)
DOWN END; (*END COPY SECTION REGION*)
```

Appendix 2. List of Action Codes for Preprocessor-RTS interface.

Part A lists all the action codes while Part B briefly describes each action code.

Part A. List of Action Code.

Group Name	Short Name	Code Number	Parameters (1st) meaning	(1st) type	(2nd) meaning	(2nd) type
RTS1	Lock	m 0	----		----	
RTS1	Unlock	m 1	----		----	
RTS1	Flush Buffer	m 16	----		----	
RTS1	End Check	m,s 17	----		----	
RTS2	Wait DP	m 32	DP-ID	integer	----	
RTS2	Terminate DP	m 33	DP-ID	integer	----	
RTS2	Reserve a slave	m 34	DP-ID	integer	----	
RTS2	Scope Check	m 35	Bottom of stack	integer	----	
RTS2	Initialization	m,s 48	Total # of DPs	integer	----	
RTS3	Download Value para	m 64	Addr. of given para.	integer	Length	integer
RTS3	Download Var para.	m 65	addr. of given para.	integer	Length	integer
RTS3	Activate DP	m 66	DP-ID	integer	# of var para.	integer
RTS3	Prepare for upload var para.	s 72	Addr. of given para.	integer	Length	integer

Group Name	Short Name	Code Number	Parameters (1st) meaning	type	(2nd) meaning	type
RTS3	Upload Var para.	s 73	Actual Upload	boolean	Length	integer
RTS3	Receive value para	s 74	Addr. of given para.	integer	Length	integer
RTS3	Receive var para.	s 75	Addr. of given para.	integer	Length	integer
RTS3	CP request	s 76	Addr. of CP package	integer	Length	integer
RTS4	Test Terminate	m 96	DP-ID	integer	----	
RTS5	Download CP	m 128	Entry Point	integer	Static Link	integer
RTS6	Wait for new task	s 160	DP-ID (var para)	integer	----	

Note. 1. All parameters in the above list are value parameters, or otherwise stated as variable parameter.

2. In the Code Number cloumn, 'm' and 's' means that this action code is used in the master side and the slave side respectively.

3. RTS4 is a function rather than a procedure. Its result type is boolean.

Part B. Brief description of Action Codes.

Lock

Enter the Critical Section.

Unlock

Leave the Critical Section.

Flush Buffer

Send whatever remains in the buffer to its buddy.

End Check

Check if it is save to terminate the system.

Wait DP

Perform the necessary operations as specified in the Wait statement.

Terminate DP

Perform the necessary operations as specified in the Terminate statement.

Reserve a slave

Reserve an available slave for the specified DP.

Scope Check

Check if the user attempts to leave the environment in

which the actual parameter of a DP is declared.

Initialization

Initialize all global variables of the RTS and create the necessary processes.

Download value parameter

Download the data mentioned in the copy section and value parameter of a DP.

Download variable parameter

Download the variable parameter of a DP.

Activate DP

Inform the slave to get ready for a new task.

Prepare for uploading variable parameter

Get ready to upload portions of the varialbe parameter specified.

Upload var

Upload changed portion of a variable parameter.

Receive value parameter

Receive the data mentioned in the copy section, the value parameter or the entry point and the static link

of a critical procedure.

Receive variable parameter

Receive variable parameter of a DP.

CP request

Request to run a CP in its master.

Test terminate

Test if a DP call should really be processed or not.

Download CP

Send to the slave the entry point and the static link of the CP which will be invoked later on.

Wait for a new task

Wait for its master to send it a new task.