AN INVESTIGATION OF AN INTERMEDIATE REPRESENTATION FOR A HIGH LEVEL LANGUAGE

By

DAVID ASA HARDEN

Bachelor of Science in Architectural Studies

Oklahoma State University

Stillwater, Oklahoma

1979

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
May, 1988

Thesis
1988
H259i
cop.2

AN INVESTIGATION OF AN INTERMEDIATE

REPRESENTATION FOR A HIGH

LEVEL LANGUAGE

Thesis Approved:

G. E. Hedrick
Thesis Adviser

M. J. Folk

Donald Grace

Norman N. Durham
Dean of the Graduate College

1302569

ACKNOWLEDGEMENTS

I wish to express appreciation to Dr. K. M. George and Dr. George E. Hedrick for their guidance and assistance throughout this entire study. Without their help, this work would never have been realized.

I am indebted to the many faculty members of Oklahoma State University and Saint Gregory's College who have provided assistance in my studies and made my graduate education a success.

Finally, a special thanks to the monks of Saint Gregory's Abbey for their unfailing support and encouragement. To these, my confrers, this work is dedicated.

TABLE OF CONTENTS

Chapter Page

I. INTRODUCTION. 1

Statement of the Problem 1
Motivation 2
Limitations. 3

II. INTERMEDIATE LANGUAGES/REPRESENTATION A SURVEY. 4

ILs or IRs 4
Types, Forms and Criteria. 5
IL´s 7
DIANA. 7
Conclusion 10

III. IMPLEMENTATION OF DIANA 11

An Overview of DIANA 11
External Form of DIANA 12
Internal Form of DIANA 13
Code Generation. 15

IV. THE MEMORY SCHEMA. 18

Memory Layout. 19
Code Layout. 22
The Initializer 22
The Code. 23
The Task Manager. 24
Conclusion 26

V. A DETAILED EXAMPLE. 27

VI. SUMMARY AND FUTURE WORK 29

Summary. 29
Suggested Future Work. 30

SELECTED BIBLIOGRAPHY. 31

Chapter Page

APPENDIX A - TOPOLOGY OF DIANA CLASSES AND NODES . . . 33

APPENDIX B - DIANA NODES 36

APPENDIX C - DIANA CLASSES 43

APPENDIX D - A ADA EXAMPLE 45

APPENDIX E - A DIANA EXAMPLE 47

APPENDIX F - A CAL EXAMPLE 60

LIST OF FIGURES

Figure		Page
1.	The Flat Form	13
2.	List of Attributes	13
3.	Node Structure	14
4.	Table Structure for Attributes	15
5.	Linked List Structure	15
6.	Tree Traversal Algorithm	17
7.	Memory Storage	18
8.	Code Storage	19
9.	Typical Task Stack	19
10.	Task Table Entry	21
11.	DIANA Topology	33
12.	An Simple Example	35

CHAPTER I

INTRODUCTION

Statement of the Problem

The problem addressed in this thesis is the investigation of an implementation scheme for DIANA (Descriptive Intermediate Attribute Notation for Ada). This implementation scheme is performed on the Perkin-Elmer model 3230 processor. The investigation also examines multi-tasking with respect to DIANA and the Perkin-Elmer. This scheme is implemented in C within the UNIX environment. The DIANA input is an ASCII representation as indicated in the DIANA Reference Manual [EVANS 83], and the output is CAL (Common Assembler Language). The Common Assembly Language assembler is licensed software, subject to restricted rights as defined in the Department of Defense, Armed Service Procurement Regulations, ASPR, paragraph 7-104.9(a): Rights in Data and Computer Software. At the current time Concurrent Computing Corporation, a Perkin-Elmer Company, is working on an Ada compiler for the Perkin-Elmer. The project consists of porting a validated Ada compiler, which does not use DIANA, written in Ada by using an Ada to Pascal translater as a bootstrap vehical [OROST 85].

Motivation

The idea of a Universal Intermediate Language (UIL) is intriguing to computer scientists as is the idea of concurrency. The question is; "Is an Universal Intermediate Language (UIL) possible?" Elsworth proposes two reasons for a UIL to be developed [ELSWORTH 78]. The first reason is to partition the job of building a compiler into logically independent parts, and the second reason is to make languages portable. Bassett in 1984 explicitly asks this question and points out opposing views [BASSETT 84]. One side says that it is impossible to have a UIL while the other side says that theoretically it is possible. A UIL does not currently exist but DIANA is a good candidate. It is beyond the scope of this thesis to prove the existence of a UIL.

Much work is being done on concurrency especially in its relationship to computer architecture. The next computer generation might exploit this area. Most of the present UIL´s do not have the capacity to handle concurrency. One approach has been to add concurrency to current UIL´s and the other approach has been to design a new UIL to include concurrency. DIANA was designed to be used with Ada including its multi-tasking features. This was one of the reasons for choosing DIANA. The main purpose of this study is to develop a tool to study concurrency. Such a tool presently does not exist for the PE-3230. This tool will allow others to experiment with various front-ends

of compilers.

Limitations

This is a study of DIANA rather than a study of Ada, therefore, some of the problems inherent in Ada implementations are not being investigated. As stated by the developers in the DIANA reference manual; "... DIANA is primarily intended as an interface between the parts of a compiler. It is also suitable for other programming support tools." Since the emphasis is on other support tools, only DIANA as stated in the DIANA Reference Manual (revision 3) is being investigated.

DIANA slowly is becoming the standard intermediate language (IL) for Ada. The first version, DIANA 81, was developed for Ada 80 but with the advent of Ada 83 many problems have developed with respect to DIANA 81. Different implementers have solved these problems in various ways and in the process destroyed the idea of a standard DIANA. To counteract these tendencies Tartan Labs Inc. in Pennsylvania under government contract began revising DIANA and in 1983 froze the specification for DIANA with revision 3.

CHAPTER II

INTERMEDIATE LANGUAGES/REPRESENTATION

A SURVEY

The goal of this survey is to examine Intermediate Languages, henceforth referred to as ILs, with an emphasis on DIANA (Descriptive Intermediate Attributed Notation for Ada). The first section of this survey discusses the differences between ILs and IRs (Intermediate Representations). The second section reviews types and forms of ILs. The third section canvasses the different ILs being used. The fourth section discusses DIANA. This includes alternatives and other uses for DIANA.

ILs or IRs

Elsworth in "Compilation via an Intermediate Language" presents a summary of the work done in this area up to 1978 [ELSWORTH 78]. Elsworth describes an IL as some intermediate representation of a program that can stand alone and has a form similar to "conventional assembly language and often being expressed in a character form." Elsworth goes on to say that an IL may be "defined in terms of operations on a simple abstract machine [ELSWORTH 78]."

Ganapathi and Fisher in their article on retargetable code (1984) go into detail on the distinction between ILs and IRs [GANAPATHI 84]. They refer to ILs as code generators which "provide dictions specially suited to describe the generation of target machine code" for example: languages like P-code and U-code. IRs on the other hand "help separate machine dependencies from the code generation algorithm" for example: representations like TCOL(tree common oriented language) and APT(abstract program tree).

Waite and Goos in their book on compiler construction (1984) defined ILs as "conceptual tools used in decomposing the task of compiling from the source language to the target language [WAIT 84]." They do not attempt to make a distinction with respect to IRs.

Aho, Sethi and Ullman in their book on compilers (1986), confuse the issue even more by their use of the same terminology [AHO 86]. To them the words IL and IR have the same meaning.

Using the Ganapathi and Fisher definition of an IR, DIANA is described in the DIANA Reference Manual (1983) as an IR; and in order for DIANA to communicate between computing systems an external ASCII form may be created [EVANS 83].

Types, Forms and Criteria

In order to use IRs, Ganapathi and Fisher have set down a list of considerations for designing IRs [GANAPATHI 84]:

1. Ease of writing a front-end translator for the IR.
2. Code generation treated as a separate package.
3. Ease of generating target code from the IR.
4. The ability to express machine-independent optimizations in the IR.
5. Storage binding front-end or back-end.

IRs may be looked at from various directions. Elsworth places IRs on a low level to high level scale according to their complexity and degree of machine or programming language orientation, and on a similar scale according to the degree of machine dependence involved [ELSWORTH 78]. There exists a tradeoff between efficiency and ease of portability corresponding to the high and low level IR techniques.

Ganapathi and Fisher are dealing with IRs on the independent level which Elsworth calls high level ILs. Ganapathi and Fisher break IRs down into three forms [GANAPATHI 84]:

1. Tuples: including quadruples, triples, indirect triples and n-tuples.
2. Abstract program trees and graph notation.
3. Linear representations such as reversed polish and standard polish notation.

Waite and Goos break IRs into four types: token sequences, structure trees, computation graphs, and target trees [WAIT 84].

Aho, Sethi and Ullman point out two important benefits of IRs which are the ease of producing IRs and the ease of

translating IRs into target code [AHO 86].

ILs

There are many ILs in existence. Elsworth presents a long list. Some of the more common program oriented ILs are CTL for Algol 60, Fortran and PL/I; P-code for Pascal; Zcode for Algol 68 [ELSWORTH 78]. TCOL (tree structured common language) is usually referred to as a family because each member is tailored to a particular source language.

MIL, a low-level IL in the image of Bliss, exists only in a graph form which is used in the Charrette Ada Compiler 1980 [ROSENBERG 80]. LOLITA, another low level IL for Ada, was developed in 1982 after DIANA, which first appeared in 1981 [ROUBINE 82]. L-code, an IL to define dynamic semantics, appeared in 1983 by Bryant and Grau [BRYANT 83]. L-code was developed for Pascal, Fortran and Ada. DAS (Delft Ada Subset) was developed at Delft Univ., Netherlands, for their Ada compiler [KATWIJK 83]. DAS is an attributed parse tree.

DIANA

One of the early experiences of writing a compiler for Ada took place at Carnegie-Mellon University. The product of this early experience was the Charrette Ada Compiler. Several articles appeared in Sigplan Notices, vol. 15, 1980, describing how this compiler was put together. The ILs used for this compiler were $TCOL_{ADA}$ for the high level and MIL

for the low level. The output from the compiler was VAX 11/780 assembler in an UNIX environment. On the other side of the Atlantic a team at the Institut Fur Informatik II, University of Karlsruhe, Germany, was working on an Ada compiler and developed an IL called AIDA. In 1981, these two Universities cooperated to produce DIANA. From 1981 until 1983, DIANA was placed under government contract to Tartan Labs. Upon completion of the last revision in 1983 DIANA was frozen by Tartan Labs [EVANS 83].

DIANA, often referred to as an attribute parse tree or an abstract syntax tree, was designed from the formal definition of ADA. One of the principal design criteria for DIANA was that the structure of the original source code was to be retained in the DIANA representation. Goos in an article "Problems in Compiling Ada" [GOOS 81] in 1981 states: "The intermediate representation of an Ada program by a DIANA tree is machine-independent only to the extent that the general structure and the attributes of the tree are machine-independent. The actual values of attribute may very well depend on the target computer." This article lays out a method to design an ADA compiler using DIANA but concentrates only on the front-end.

In 1982 Simpson [SIMPSON 82] and Taft [TAFT 82] in separate articles use DIANA as an IR in their designs and both point out some problems with the definition of DIANA. Revision 3 of DIANA 83 corrected these problems.

Taft states that the DIANA proposal "purposely avoids

specifying a single implementation strategy." Therefore, his company is looking at two specific implementation techniques to use DIANA most efficiently. The first technique represents DIANA nodes as ADA variant record types, and the second by implementing separate compilation using a software virtual memory technique. Simpson on the other hand is studying the implementation DIANA in the ALS environment. ALS stands for the Ada Language System which is the Ada support environment developed for the U.S. Army. Simpson study includes output from the front-end, the code generation, the program library manager and the KAPSE (Kernel Ada Program Support Environment).

The philosophy behind code generation for IRs is that they must be adaptable easily to any machine within a large class of conventional architectures, typically machines with directly addressable memory and a set of registers. In 1982 two different low level IRs came into existence, LOLITA and I-code. LOLITA, a low level IR for Ada, was presented by Roubin and company in 1982 in their article LOLITA: A Low Level Intermediate Language for Ada [ROUBINE 82]. Their main objection to DIANA was the tedious job of writing the code generator. LOLITA exists only as an IR for a particular source language which is not related to any abstract machine model.

One may ask the question with respect to low level IRs: How far is low? For LOLITA this was defined as low as machine independence would permit. For I-code, presented by

Appelbe [APPELBE 82], which was designed along the same lines as LOLITA is describe as similar to P-code in form.

To show the importance of a low level IR the Karlsruhe Ada compiler which uses DIANA also uses AIM (abstract intermediate machine) [PERCH 83]. The purpose of AIM is to ease the retargeting of the compiler.

DIANA was designed to be useful for the generation of other environment tools. A source oriented debugger on a minicomputer network for image sequence analysis at Faclbereich Informatik der Univ., Hamburg, Germany, uses DIANA [FAASCH 83]. Slape and Wallis in 1983 used DIANA to translate Fortran to Ada [SLAPE 83]. The main complaint of Slape and Wallis about DIANA was the lack of a rigorous standard. Rosenblum in A Method for the design of Ada transformation tools in a DIANA environment [ROSENBLUM 85] presents four tools using DIANA: an Ada source program optimizer, a robust programming transformer, a programming style transformer, and a debugger preprocessor.

Conclusion

IR´s exist only in internal form, therefore it is left up to the implementor on how faithful the implementation is. IL´s not only have an internal form but also an external ASCII form -- which gives a metric for discussion. As a result IL´s allow for machine independent front ends and the ability to transport code from one machine to another at the IL level, and to this end DIANA is well suited.

CHAPTER III

IMPLEMENTATION OF DIANA

An Overview of DIANA

DIANA as an intermediate language encodes the results of lexical, syntactic, and semantic analysis. Therefore, DIANA may be referred to as an attributed parse tree. Since DIANA was designed with Ada in mind, each entry in the Ada syntax has a corresponding node in DIANA. The definition of the nodes and attributes are in chapter 2 of the DIANA Reference Manual [EVANS 83]. Each node of a DIANA tree contains zero or more attributes which are structural attributes, semantic attributes, lexical attributes or code attributes.

The structural attribute prefixed with "as_" corresponds to the edges of the parse tree and always points to another DIANA node. The semantic attribute prefixed with "sm_" contains information about the static semantics of the source program and are used in type checking and aid in procedure overloading, when allowed. The lexical attribute prefixed with "lx_" contains the lexical information about the source program and is used in order to reproduce the source. The code attribute prefixed with "cd_" contains information found for the code generator. Currently, there

is only one code attribute (cd_impl_size) and it contains the number of bits needed to represent some object.

The DIANA Reference Manual contains a chapter on implementation options. The philosophy behind this chapter is to present suggestions on various types of options. The opening paragraph recommends that the options match the applications. As a result, the following implementation scheme was chosen. The simple flat form was used for the external form, and a node structure using pointers was used for the internal form. A bottom up parser was used to traverse the tree and produce assembler code which will then be put through the assembler to produce machine code.

External Form of DIANA

The external form of DIANA may take on three different appearances: the flat form, the nested form and a combination of the two forms. For the sake of simplicity, the flat form was chosen for use in this paper.

The flat form is an external representation of a node pointer structure (see figure 1). Each node has a label or node identification which is a sequence of upper case letters followed by a sequence of numbers. The label is terminated by a colon. The next item in the node is the node-name, a nonterminal which consists of a sequence of lower case letters and underlines (for a listing of the nodes used in this implementation see appendix B). The list of DIANA attributes follows enclosed in square brackets.

The square brackets may be omitted if the attribute list is empty.

```
label:   node-name [ list-of-attributes ]
```

Figure 1. The Flat Form

The list-of-attributes is a series of items with each one separated by a semi-colon (see figure 2). Each item in the list contains an attribute-name followed by a label, a sequence, or a string (see appendix B for the list used in this implementation). The label is used as a pointer to the next DIANA node in the structure and is followed by a caret. The sequence is a list of labels enclosed in angle brackets with each label followed by a caret. These also point to another node. The sequence may also be empty, in which case only the angle brackets appear. The string is a terminal attribute containing a list of ascii characters in cased in double quotes.

```
attribute-name label^;
attribute-name < label^ label^ ... >;
attribute-name "string";
attribute-name label^
```

Figure 2. List of Attributes

Internal Form of DIANA

The analyzer reads the internal form and converts it into the internal representation by using LEX, a regular expression based lexical analyzer generator developed by M. E. Lesk for AT&T Laboratories in 1975, to create the tokens and a parser which builds the tree. In building the tree the parser only checks the syntax for the node. It does not check for proper node classes. The representation chosen for the internal representation is a directed acyclic graph written in C. The nodes are represented by a generic C structure shown below in figure 3.

```
struct node { /* DIANA nodes */
  pointer parent; /* pointer to parent for traversal    */
  int token;      /* node type                          */
  int count;      /* number of visits during traversal  */
  int n_attribute;/* number of attributes in table      */
  struct table attribute[MAX_N_ATT]; /* attribute table */};
```

Figure 3. Node Structure

The non-leaf attributes are pointers to nodes. Since there are only a maximum of seven attributes per node, the attributes are stored in a table as seen in figure 4.

```
struct table {    /* structure for attributes */
  int token   /* type of attribute */
  int type    /* contents of union */
  union {
    pointer ptr;         /* pointer to next node */
    char leaf [BUFFER];  /* contents of leaf     */
    struct link sequence;/* sequence of pointers */
  } };
```

Figure 4. Table Structure Attributes

The union is used in order to distinguish among the three types of attributes: a pointer, a leaf, or a sequence. The sequence is a pointer to a linked list containing pointers to nodes (Figure 5).

```
struct link { /* structure of type sequence */
  struct node *ptr;  /* pointer to next node */
  struct link *next; /* pointer to next link */ };
```

Figure 5. Linked List Structure

Code Generation

The code generator is a tree traversal algorithm which is written in C and which translates the DIANA internal representation into CAL, a Common Assembler Language which is a product of the Department of Defense. CAL was chosen because it is portable over a wide range of machines (at a

minimum all Perkin Elmer machines) and will allow a later addition of an optimizer at the assembler level which will allow finer adjustment to a particular machine. The assembly stage also allows for error checking at this level. Common Assembly Language Programming is licensed software, subject to restricted rights as defined in the Department of Defense, Armed Service Procurement Regulations, ASPR, paragraph 7-104.9(a): Rights in Data and Computer Software.

The code generator uses a hybrid depth-first left to right tree traversal algorithm as shown in figure 6 below. The stack is used to control the order in which the nodes are processed. Each node-name has its own processing procedure which is invoked by the process procedure. The process procedure is a switch which contains an entry for each type of node (see Appendix A for this implementation). As the tree is traversed, the node processing procedure pushes the appropriate node-name attribute on the stack, and builds a symbol table, builds individual files for each task containing the assembler code for that task. Information is also gathered for the task manager.

Upon completion of the traversal procedure a clean-up procedure is called which builds the assembler code file. The clean-up procedure builds the initializer then concatenates all the task files to the initializer and then adds the task manager completing the file.

```
Procedure traversal (node-name)
  Process (node-name);
  Pop stack (node-name);
  Loop while stack is non-empty
    Process (node-name);
  end loop;
end traversal;

Procedure Process (node-name)
 -- Contains a switch which calls the
 -- appropriate procedure to do the
 -- actual processing.
```

Figure 6. Tree Traversal Algorithm

CHAPTER IV

THE MEMORY SCHEME

Traditionally, memory consists of two parts, the run-time storage and program-code storage. This memory scheme augments the traditional setup by modifying the run-time storage and program code. The run-time storage maintains a static table which contains information about each task and a stack area for each task managed at run time (see figure 7). The program code area contains a section called the task manager which interacts with the task table and the individual task at run time (see figure 8).

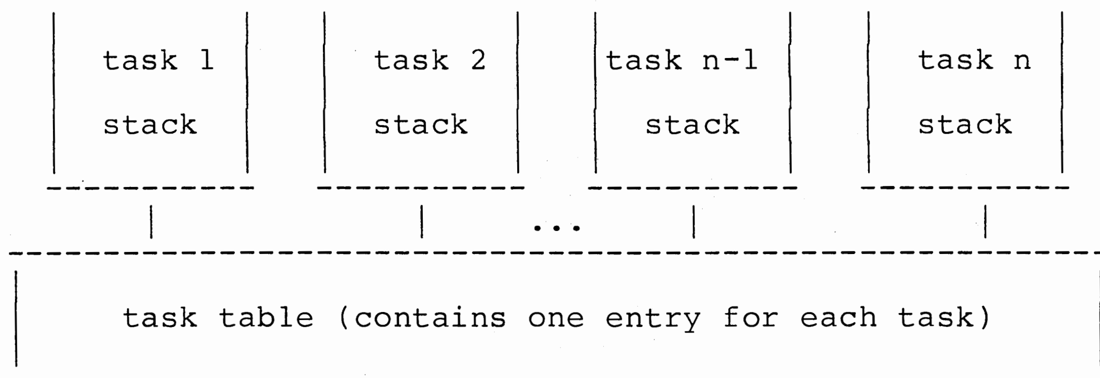

Figure 1. Memory Storage

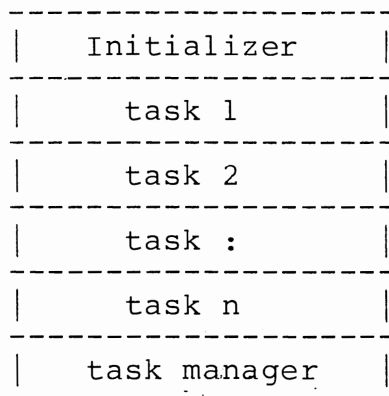

Figure 8. Code Storage

Memory Layout

The task table and the task stack area are set at compile time. Since the Perkin-Elmer model 3230 is a uni-processing machine, procedures use the stack area above the tasks where each procedure is allocated or deallocated as needed. Registers are used for pointers, parameter passing, and calculations.

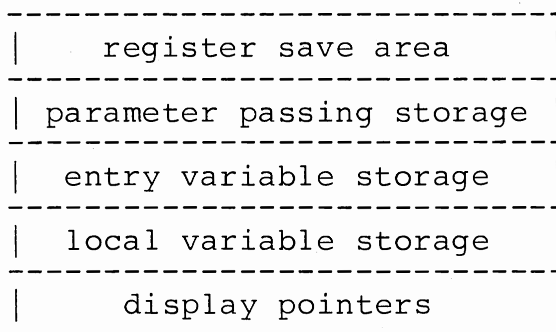

Figure 9. Typical Task Stack

The task stack area is divided up into five areas as shown in figure 9. The register save area and the parameter passing storage are used when a procedure call is invoked. The register save area is used to store the current environment and the parameter passing storage is used to implement call-by-value parameter passing mechanism. The entry variable storage is a new area in the stack model and is used to implement the rendezvous method of communication between tasks. The entry variable is used much in the same manner as the parameter passing storage but acts like a local variable storage. The local variable storage stores all variables used by the particular task. Since tasks are somewhat like procedures especially in scoping, a display area is set up in the same manner as dynamic procedure stacks.

The task table maintains an entry for each task and a typical entry may be viewed in figure 10. There are four possible states: running, ready-to-run, sleeping and terminated. A running task is currently being executed by the processor. Ready-to-run tasks are ready and waiting to be run. Sleeping tasks are tasks waiting for a rendezvous to take place. In order for a task to terminated a search for dependent tasks is made. If a dependent task does exist, then the task is left active, else it is marked as terminated. The second entry contains the priority -- the order in which the tasks are to be given attention. The third and fourth entries, next instruction and active area

pointer are used together to store the environment when a task is interrupted. The next-instruction contains the address of the next instruction for the task to execute and the active-area-pointer contains the stack location for the task. A task may not be terminated until all its child tasks have been terminated; therefore, it is necessary to know the number-of-children and the parent of a task. When a task terminates, it notifies its parent by decrementing the number of children in the parent by one. In order to implement the rendezvous mechanism, a queue is used to store information about a calling tasks wanting to make contact with the called task.

state
priority
next-instruction
active-area-pointer
number-of-children
parent
queue

Figure 10. Task Table Entry

Code Layout

The program code storage as seen in figure 8. consists of the task manager, an area for each task, and the initializer. The initializer not only builds the task table, but also builds the task stacks. The code for each task follows on a first-come first-serve basis. The task manager follows, which controls the overall running of the program.

The Initializer

The initializer creates a table entry for each task as seen in figure 10. It also builds the task stacks as seen in figure 8 then passes control to the task manager.

There are four possible states: running, ready-to-run, sleeping and terminated. Initially all tasks are set to ready-to-run state. Since this is a uni-processing system only one task may run at a time, and the tasks priorities determine which task is run first. When a task is interrupted, then the task manager assumes control, determines the problem and marks the task appropriately. Currently, there are two conditions which determine whether a task is to be placed in the sleeping state, both of which concern the rendezvous mechanism. The first one has to do with a call to another task: the one calling is put in a wait state until the called task answers. The second type of waiting occurs when a task is expecting a call: it is put into a wait state until it is called. A state is marked

done when a task and all its child tasks have terminated. As long as a child task is marked active or waiting a task may not terminate.

The priority pragma has not been implemented; however, a priority is assigned each task at compile time. The priority is determined on a first-come, first-serve basis. Each time the task manager is invoked the active task with the highest priority is run.

The initializer next initializes the next-instruction to the address of the start of the task code area and the active-area-pointer is set to the task stack address. The number-of-child tasks and the parent task address is maintained to handle the task termination. Since a task may not terminate until all its child tasks have terminated, the number-of-child tasks is decremented as each child terminates. In order to perform this operation the address of the parent task is necessary.

A queue is set up to hold the entries for a calling task. The address of the calling task and the address of the entry variable is stored in the queue. Currently the queue holds a maximum of five entries for the purpose of testing.

The Code

The code for each task follows the initializer. The code in each task is augmented to handle the rendezvous mechanism and the interaction with the task manager. The

initialization or start up for each task is not different from any other main procedure, but the termination of a task contains a control mechanism that interacts with the task manager. This termination mechanism is necessary in order for a task to remain active until all the child tasks have terminated. Code is also necessary to handle the rendezvous mechanism which is divided into two parts: the receiving mechanism and the calling mechanism. The task calling another task sends the task manager the address or the receiver, the address of the variable storage containing the information, and the address to return after completion of the rendezvous. The receive mechanism consists of two parts: a begin-accept and a end-accept. The begin-accept tells the task manager that it ready to receive a call and sends the task manager its address. The end-accept send the information back to the calling task and notifies the task manager that the rendezvous has taken place.

The Task Manager

The task manager consists of five routines: the run-next-task, task-terminator, begin-accept, end-accept and task-call. These routines store and retrieve information from the task table and interact with the individual tasks.

Run-next-task searches for the first task with the highest priority. If the task is marked active, then the environment is loaded and control is turned over to the task. If none of the tasks are active, then the program is

terminated.

The task terminator was developed to maintain an active task while child tasks exists. The mechanism is a simple call to the task manager to see whether all the child tasks have terminated. If all the child tasks have terminated then the running task is marked done but if some child tasks exist then the priority is lowered and the task is kept active and control is turned over to the run-next-task routine.

The rendezvous mechanism consists of a calling routine and a receive mechanism. The calling routine is invoked by a task attempting to make contact with another task and the receive mechanism controls the activity of the called task.

The calling routine, task-call, stores the environment of the calling task then places it in a wait state. It then places the calling task address and the address of the calling task variable in the called task queue. And it wakes up the called task if it is in a wait state.

The receive mechanism is made up of two routines: a begin-accept and end-accept. When an accept statement is encounted in the task code the task manager is invoked and the begin-task routine acquires control. This routine stores the environment of the task and then checks the queue. If the queue is empty the task is placed in the wait state; else if the queue is not empty the task is left active and control is turned over to run-next-task. When the receive mechanism is ready to terminate communication

the task manager is called and the end-accept takes control. This routine makes the calling routine active and readjusts the entry queue to the next task. And finally control is turned over to run-next-task.

Conclusion

The stack model is a very convenient tool for the implementation of DIANA since tasks and procedures act in a similar manner. The tasking convention explained above uses the stack model but augments it in two ways: by adding a task table and a stack for each task. The development of the task table came from the tasking convention that states: all tasks in a program are active upon invoking of that program. The simplicity of the task-manager and its interaction between the task stacks and the task table show the beauty of this implementation.

CHAPTER V

A DETAILED EXAMPLE

For a better understanding of DIANA, a detailed example is shown below. Ada was chosen for the high level language in this example because of its historical relationship to DIANA. This example shows an Ada program and its transformation to a CAL program through DIANA. DIANA does not have a facility to output information to a printer or monitor; therefore the following node was added to DIANA with its attributes

```
out_put => lx_symrep   : symbol_rep,
           sm_obj_type : TYPE_SPEC,
           sm_address  : EXP_VOID,
           sm_obj_def  : OBJECT_DEF;
```

which invokes the same mechanism as printf invokes in C. The output node prints the symbol representation and its contents on a single line.

The Ada program in appendix D explores the different aspects of multi-tasking. Four tasks which includes the main procedure are created. One of the tasks (t3) is embedded in another task (t2) to test task scoping. The tasks interact in various ways by passing information between them. Simple integer arithmetic and the put statement are used to explore these different aspects.

The DIANA code which was hand produced from the Ada code of appendix D is shown in appendix E. The first step in producing a DIANA code is to produce the structural tree, which is shown in appendix E for this example, and secondly decorate the tree with the other attributes. Due to the length of the DIANA code produced the semantic attributes were left out of this example.

The CAL assembler code of appendix F was produced from DIANA code of appendix E by the code generator of this implementation. To give a better understanding of appendix F, comments were placed in various places in the assembler code.

CHAPTER VI

SUMMARY AND FUTURE WORK

Summary

Intermediate languages are important to the development of language theory and DIANA has made a contribution to the development of language theory. Therefore the purpose of this study was to create a code generator for the tasking model of DIANA. To this end the study has accomplished its purpose.

In building the code generator two compiler tools were examined: the lexical analyzer LEX and the parser YACC. The lexical analyzer LEX aided greatly in this development. However, due to the nature of DIANA the parser YACC was not capable of handling the ASCII form of DIANA. Therefore a parser was developed and an internal form was created. The LEX program and the parser were written in such a way that it could be easy to extend them to include the whole of DIANA and any extension to DIANA that might be made necessary by an extension of DIANA itself.

The decision to use the assembler language CAL as an intermediate language allowed for easy error detection and correction in the development of the memory schema. By using the concepts of modular design the memory scheme

proved easy to modify and to update. More work can to be done on the priorities pragma which has not been implemented.

To further facilitate the understanding of DIANA a detailed example is given in the appendix. A work was also produced which would allow various high level languages to produce DIANA with a facility to test their code.

Suggested Future Work

This study explores only one tasking model for DIANA. There are other models in existence. One possibility is using a heap instead of a stack for memory management. Another possibility would be mapping the individual tasks to the processes of the operating system and allowing the operating system to perform inter-task communication as inter-process communication. A comparative study of different implementations on the same machine would be interesting.

SELECTED BIBLIOGRAPHY

Aho, A. V., R. Sethi and J.D. Ullman. 1986. Compilers: Principles, Techniques, and Tools, Addison Wesley, Reading Massachusetts.

Appelbe, B. and G. Dismukes. 1982. "An Operational Definition of Intermediate Code for Implementing a Portable Ada Compiler." Proceedings of the AdaTec Conference on Ada, Arlington Vigina, 266-274.

Bassett, S. 1984. "Multipass Compilers Produce Tight Code."Computer Design, vol 23, no 1, 44-47.

Brosgol, B. M. 1980. "TCOLada and the MIDDLE END of the PQCC Ada Compiler." Sigplan Notices, vol 15, no 11, 101-112.

Bryant, B. R. and A. A. Grau. 1985. "An Intermediate Language to Define Dynamic Semantics." Computer Languages, vol 9, no 2, 24-33.

Elsworth, E. F. 1978. "Compilation via an Intermediate Language." The Computer Journal, vol 22, no 3, 226-233.

Evans, A., K.J. Buther, G. Goos, W.A. Wulf. 1983. DIANA Reference Manual, Revision 3, Springer-Verlag, N.Y.

Faasch, H., V. Haarslev, H. Nagel. 1983. "Ada on a Minicomputer-Network for Image Sequence Analysis: an investigative implementation." Ada Letters, vol II, no 4, II-4.92 - II-4.96.

Ganapathi, M. and C. N. Fisher. 1984. "Attributed Linear Intermediate Representation for Retargetable Code Generation." Software-Practice and Experience, vol 14, no 4, 347-364.

Goos, G. and G. Winterstein. 1982. "Problems in Compiling Ada." Lecture Notes in Computer Science #123, 173-199.

Hisgen, A., D. A. Lamb, J. Rosenberg, M. Sherman. 1980. "A Runtime Representation for Ada Variables and Types." Sigplan Notices, vol 15, no 11, 82-90.

Katwijk, J. van and J. van Somcren. 1983. "The DAS Compiler a ststus report." Ada-Europe/AdaTec Joint Conference on Ada, Brussels, 28.1-28.2.

Lamb, D. A., A. Hisgen, M. S. Sherman, J. Rosenberg. 1980. "An Ada Code Generator for VAX/780 with UNIX." Sigplan Notices, vol 15, no 11, 91-100.

Orost, J. M. 1985. "Network News." USENET, newsgroup lang.ada Jan 24.

Perch, G., J. Uhl, H. Jansohn, W. Kirchgassner, R. Landwehr, M. Dausmann, S. Drossopouiou, G. Goos. 1983. "Ada Compiler Karlsruhe: an overview." Ada-Europe/AdaTec Joint Conference on Ada, Brussels, 2.1-2.4.

Rosenblum, D.S. 1985. "A Methodology for the Design of Ada Transformation Tools in a DIANA Environment." IEEE Software, vol 2, no 2, 24-33.

Rosenberg, J., D. A. Lamb, A. Hisgen, M. S. Sherman. 1980. "The Charrette Ada Compiler." Sigplan Notices, vol 15, no 11, 1980, 72-81.

Roubine, O., J. Teller and, O. Maurel. 1982. "Lolita: A Low Level Intermediate Language for Ada." Proceedings of the AdaTec Conference on Ada, Arlington, Vigina, 251-260.

Sherman, M. S. 1980. "A flexible Semantic Analyzer for Ada." Sigplan Notices, vol 15, no 11, 62-71.

Simpson, R. T. 1982. "The ALS Ada Compiler Front End Architecture." Proceedings of the AdaTec Conference on Ada, Arlington, Vigina, 98-106.

Slape, J. K. and P. J. L. Wallis. 1983. "Conversion of Fortran to Ada Using an Intermediate Tree Representation." The Computer Journal, vol 26, no 4, 344-353.

Taft, S. T. 1982. "Diana as an Internal Representation in an Ada-In-Ada Compiler." Proceedings of the AdaTec Conference on Ada, Arlington, Vigina, 261-265.

Waite, W. M. and G. Goos. 1984. Texts and Monographs in Computer Science: Compiler Construction, Springer-Verlag, N.Y.

APPENDIX A

TOPOLOGY OF DIANA CLASSES AND NODES

Appendix B and C contain respectively the nodes and classes of the DIANA language which are used in this implementation and have been reproduced from the DIANA reference manual [EVANS 83]. In order to gain a better understanding of convensions used in appendix B and C, figure 11 is shown below followed by an example.

```
ACTUAL      BLOCK_STUB   USED_ID  :   DIANA class names.
void        proc_id      block    :   DIANA node names.
lx_symrep   sm_value     as_name  :   DIANA attributes.

Boolean     Integer         :    Identifier defined by user.

--                     : Indicates a comment follows which
                         continues to end of line.
_s          _S         : Indicates sequence of what comes
                         before '_'.
```

Figure 11. DIANA Notation

The definition of DIANA as seen in appendix B and C is similar in form to BNF. The production rules are broken down into two parts: the terminals in appendix B and the

nonterminals in appendix C. In the following definition

```
EXP ::= leaf | tree;
```

EXP is defined as a class name or a nonterminal followed by a choice of a leaf or tree. The leaf and the tree are node names or terminals. The two nodes in this example may be expressed as follows

```
tree => as_left : EXP,
        as_op   : OPERATOR,
        as_right: EXP;

leaf => as_string : Character_S;
```

where the node name tree has three attributes as_left, as_right, and as_op. The node name leaf has one attribute as_string which is supplied by the user. The class name OPERATOR is described as

```
OPERATOR ::= Add | Subtract | Multiply | Divide;
```

where Add, Subtract, Multiply and Divide are built in operators.

Using the node names and classes described above, an algebraic expression shown in figure 12a is transformed into a graphical representation in figure 12b and finally into a pseudo-DIANA flat form in figure 12c. For more information on the DIANA flat form see page 12 of this thesis.

x + y * z

Figure 12a. Algebraic Form

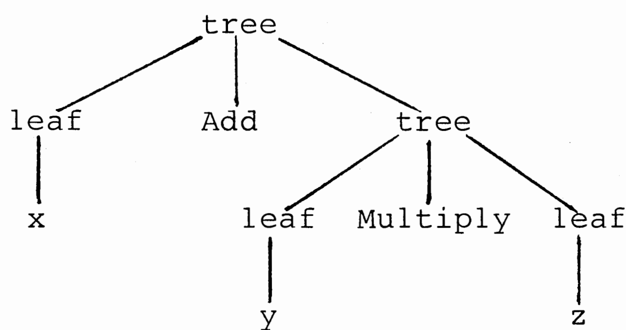

Figure 12b. The Graphic Form

```
A0: tree [ as_left A1^, as_op Add, as_right A2^; ]
A1: leaf [ as_string "x"; ]
A2: tree [ as_left A3^, as_op Multiply, as_right A4^; ]
A3: leaf [ as_string "y"; ]
A4: leaf [ as_string "z"; ]
```

Figure 12c. The Flat Form

Figure 12. An Simple Example

APPENDIX B

A LIST OF DIANA NODE NAMES AND THERE ATTRIBUTES

```
abort =>          as_name_s           : NAME_S,
                  lx_srcpos           : source_position,
                  lx_comments         : comments;

accept =>         as_name                     : NAME,
                  as_param_assoc_s            : PARAM_S,
                  as_stm_s                    : STM_S,
                  lx_srcpos                   : source_position,
                  lx_comments                 : comments;

arguement_id => lx_symrep             : symbol_rep;

assign =>         as_name             : NAME,
                  as_exp              : EXP,
                  lx_srcpos           : source_position,
                  lx_comments         : comments;

assoc =>          as_designator       : DESIGNATOR,
                  as_actual           : ACTUAL,
                  lx_srcpos           : source_position,
                  lx_comments         : comments;

attribute =>      as_id               :  ID,
                  -- always a "used_name_id" whose attributes
                  -- points to a prefined "attr_id"
                  as_name             : NAME,
                  lx_srcpos           : source_position,
                  lx_comments         : comments,
                  sm_exp_type         : TYPE_SPEC,
                  sm_value            : value;

block =>          as_item_s           : ITEM_S,
                  as_stm_s            : STM_S,
                  as_alternative_s: ALTERNATIVE_S,
                                               -- not implemented
                  lx_srcpos           : source_position,
                  lx_comments         : comments;

box =>            lx_srcpos           : source_position,
                  lx_comments         : comments;
```

```
comp_unit =>      as_context        : CONTEXT,
                                               -- not implemented
                  as_unit_body      : UNIT_BODY,
                  as_pragma_s       : PRAGMA_S,
                                               -- not implemented
                  lx_srcpos         : source_position,
                  lx_comments       : comments;

compilation =>    as_list           : Seq OF COMP_UNIT,
                  lx_srcpos         : source_position,
                  lx_comments       : comments;

cond_entry =>     as_stm_s1         : STM_S,
                                      -- first stm is entry call
                  as_stm_s2         : STM_S,
                  lx_srcpos         : source_position,
                  lx_comments       : comments;

const_id =>       lx_srcpos         : source_position,
                  lx_comments       : comments,
                  lx_symrep         : symbol_rep,
                  sm_address        : EX_VOID,
                  sm_obj_type       : TYPE_SPEC,
                  sm_obj_def        : OBJECT_DEF,
                  sm_first          : DEF_OCCURRENCE;
                                             -- used for deferred

constant =>       as_id_s           : ID_S,  -- seq of const_id
                  as_type_spec      : TYPE_SPEC,
                  as_object_def     : OBJECT_DEF,
                  lx_srcpos         : source_position,
                  lx_comments       : comments;

constrained =>    as_name           : NAME,
                  as_constraint     : CONSTRAINT, -- void
                  cd_imp_size       : integer,
                  lx_srcpos         : source_position,
                  lx_comments       : comments,
                  sm_type_struct    : TYPE_SPEC,
                  sm_base_type      : TYPE_SPEC,
                  sm_constraint     : CONSTRAINT;   -- void

decl_s =>         as_list           : seq of DECL,
                  lx_srcpos         : source_position,
                  lx_comments       : comments;

delay =>          as_exp            : EXP,
                  lx_srcpos         : source_position,
                  lx_comments       : comments;
```

```
entry_call =>   as_name                     : NAME,
                        -- indexed when entry of family
                as_param_assoc_s            : PARAM_ASSOC_S,
                lx_srcpos                   : source_position,
                lx_comments                 : comments,
                sm_normalize_param_s        : EXP_S;

exp_s =>        as_list         : seq of EXP,
                lx_srcpos       : source_position,
                lx_comments     : comments;

function_call =>as_name                     : NAME,
                as_param_assoc_s            : PARAM_ASSOC_S,
                lx_srcpos                   : source_position,
                lx_comments                 : comments,
                lx_prefix                   : Boolean,
                sm_value                    : value,
                sm_normalized_param_s       : EXP_S;

id_s =>         as_list         : seq of ID,
                lx_srcpos       : source_position,
                lx_comments     : comments;

in =>           as_id           : ID_S, -- always in_id
                as_name         : NAME,
                as_exp_void     : EXP_VOID,
                lx_srcpos       : source_position,
                lx_comments     : comments,
                lx_default      : Boolean;

in_id =>        lx_srcpos       : source_position,
                lx_comments     : comments,
                lx_symrep       : symbol_rep,
                sm_obj_type     : TYPE_SPEC,
                sm_init_exp     : EXP_VOID,
                sm_first        : DEF_OCCURRANCE;

in_out =>       as_id           : ID_S, -- always in_out_id
                as_name         : NAME,
                as_exp_void     : EXP_VOID, -- always void
                lx_srcpos       : source_position,
                lx_comments     : comments;

in_out_id =>    lx_srcpos       : source_position,
                lx_comments     : comments,
                lx_symrep       : symbol_rep,
                sm_obj_type     : TYPE_SPEC,
                sm_first        : DEF_OCCURRANCE;
```

```
integer =>        as_range           : RANGE,
                  cd_imp_size        :Integer,
                  lx_srcpos          : source_position,
                  lx_comments        : comments,
                  sm_size            : EXP_VOID,
                  sm_type_struct     : TYPE_SPEC,
                  sm_base_type       : TYPE_SPEC;

item_s =>         as_list            : seq of ITEM,
                  lx_srcpos          : source_position,
                  lx_comments        : comments;

name_s =>         as_list            : seq of NAME,
                  lx_srcpos          : source_position,
                  lx_comments        : comments;

no_default =>     lx_srcpos          : source_position,
                  lx_comments        : comments;

null_access =>    lx_srcpos          : source_position,
                  lx_comments        : comments,
                  sm_exp_type        : TYPE_SPEC,
                  sm_value           : value;

null_stm =>       lx_srcpos          : source_position,
                  lx_comments        : comments;

number =>         as_id_s            : ID_S,
                                        -- sequence of number_id
                  as_exp             : EXP,
                  lx_srcpos          : source_position,
                  lx_comments        : comments;

number_id =>      lx_srcpos          : source_position,
                  lx_comments        : comments,
                  lx_symrep          : symbol_rep,
                  sm_obj_type        : TYPE_SPEC,
                          -- always refers to a universal type
                  sm_init_exp        : EXP;

numeric_literal =>        lx_srcpos          : source_position,
                          lx_comments        : comments,
                          lx_numrep          : number_rep,
                          sm_exp_type        : TYPE_SPEC,
                                               -- universal type
                          sm_value           : value;

out =>            as_id              : ID_S,     -- always out_id
                  as_name            : NAME,
                  as_exp_void        : EXP_VOID, -- always void
                  lx_srcpos          : source_position,
                  lx_comments        : comments;
```

```
out_id =>          lx_srcpos          : source_position,
                   lx_comments        : comments,
                   lx_symrep          : symbol_rep,
                   sm_obj_type        : TYPE_SPEC,
                   sm_first           : DEF_OCCURRENCE;

param_assoc_s =>as_list               : seq of PARAM_ASSOC,
                   lx_srcpos          : source_position,
                   lx_comments        : comments;

param_s =>         as_list            : seq of PARAM,
                   lx_srcpos          : source_position,
                   lx_comments        : comments;

parenthesized =>as_exp                : EXP,
                   lx_srcpos          : source_position,
                   lx_comments        : comments,
                   sm_exp_type        : TYPE_SPEC,
                                                -- universal type
                   sm_value           : value;

proc_id =>         lx_srcpos          : source_position,
                   lx_comments        : comments,
                   lx_symrep          : symbol_rep,
                   sm_spec            : HEADER,
                   sm_body            : SUBP_BODY_DECS,
                   sm_location        : LOCATION,
                   sm_stub            : DEF_OCCURRENCE,
                   sm_first           : DEF_OCCURRANCE;

procedure =>       as_param_s         : PARAM_S,
                   lx_srcpos          : source_position,
                   lx_comments        : comments;

range =>           as_exp1            : EXP,
                   as_exp2            : EXP,
                   lx_srcpos          : source_position,
                   lx_comments        : comments,
                   sm_base_type       : TYPE_SPEC;

select =>          as_select_clause_s         : SELECT_CLAUSE_S,
                   as_stm_s                   : STM_S,
                   lx_srcpos                  : source_position,
                   lx_comments                : comments;

select_clause =>as_exp_void           : EXP_VOID,
                   as_stm_s           : STM_S,
                                   -- first stm accept or delay
                   lx_srcpos          : source_position,
                   lx_comments        : comments;
```

```
select_clause_s =>      as_list     : seq of SELECT_CLAUSE,
                        lx_srcpos   : source_position,
                        lx_comments : comments;

stm_s =>        as_list          : seq of STM,
                lx_srcpos        : source_position,
                lx_comments      : comments;

stub =>         lx_srcpos        : source_position,
                lx_comments      : comments;

subprogram_body =>      as_designator    : DESIGNATOR,
                        -- proc_id, function_id, or def_op
                        as_header        : HEADER,
                        as_block_stub    : BLOCK_STUB,
                        lx_srcpos        : source_position,
                        lx_comments      : comments;

subprogram_decl =>      as_designator      : DESIGNATOR,
                         -- proc_id, function_id, or def_op
                        as_header          : HEADER,
                        as_subprogram_def  : SUBPROGRAM_DEF,
                        lx_srcpos          : source_position,
                        lx_comments        : comments;

task_body =>    as_id            : ID, -- always task_body_id
                as_block_stub    : BLOCK_STUB,
                lx_srcpos        : source_position,
                lx_comments      : comments;

task_body_id => lx_srcpos        : source_position,
                lx_comments      : comments,
                lx_symrep        : symbol_rep,
                sm_type_spec     : TYPE_SPEC,
                sm_body          : BLOCK_STUB_VOID,
                sm_first         : DEF_OCCURRANCE,
                sm_stub          : DEF_OCCURRANCE;

task_decl =>    as_id            : ID,   -- always var_id
                as_task_def      : TASK_DEF,
                lx_srcpos        : source_position,
                lx_comments      : comments;

task_spec =>    as_decl_s        : DECL_S,
                lx_srcpos        : source_position,
                lx_comments      : comments,
                sm_body          : BLOCK_STUB_VOID,
                        -- void only in presence of seperate
                        -- compilation
                sm_address       : EXP_VOID,
                sm_storage_size  : EXP_VOID;
```

```
terminate =>      lx_srcpos         : source_position,
                  lx_comments       : comments;

timed_entry =>    as_stm_s1         : STM_S,
                                     -- first stm is entry_call
                  as_stm_s2         : STM_S,
                                     -- first stm is delay
                  lx_srcpos         : source_position,
                  lx_comments       : comments;

used_bltn_id =>   lx_srcpos         : source_position,
                  lx_comments       : comments,
                  lx_symrep         : symbol_rep,
                  sm_operator       : operator;

used_bltn_op =>   lx_srcpos         : source_position,
                  lx_comments       : comments,
                  lx_symrep         : symbol_rep,
                  sm_operator       : operator;

used_name_id =>   lx_srcpos         : source_position,
                  lx_comments       : comments,
                  lx_symrep         : symbol_rep,
                  sm_defn           : DEF_OCCURRENCE;

used_object_id =>         lx_srcpos         : source_position,
                          lx_comments       : comments,
                          lx_symrep         : symbol_rep,
                          sm_exp_type       : TYPE_SPEC,
                          sm_defn           : DEF_OCCURRENCE,
                          sm_value          : value;

used_op =>        lx_srcpos         : source_position,
                  lx_comments       : comments,
                  lx_symrep         : symbol_rep,
                  sm_defn           : DEF_OCCURRENCE;

var =>            as_id             : ID_S,
                                          -- sequence of var_id
                  as_type_spec      : TYPE_SPEC, -- constrained
                  as_object_def     : OBJECT_DEF,
                  lx_srcpos         : source_position,
                  lx_comments       : comments;

var_id =>         lx_srcpos         : source_position,
                  lx_comments       : comments,
                  lx_symrep         : symbol_rep,
                  sm_object_type    : TYPE_SPEC,  -- constrained
                  sm_address        : EXP_VOID,
                  sm_obj_def        : OBJECT_DEF;

void => ;  -- no equivalent in concrete syntax
```

APPENDIX C

DIANA CLASSES

```
BLOCK_STUB ::=  block;

CONSTRAINED ::= constrained;

CONSTRAINT ::=  void;

DECL ::=        constant |      subprogram_decl |       var |
                task_decl;

DEF_ID ::=      proc_id |       in_id |         in_out_id |
                out_id |        var_id;

DESIGNATOR ::=  ID |    OP;

EXP ::=         NAME|           null_access | numeric_literal
                parenthesized;

EXP_VOID ::=    EXP |   void;

HEADER ::=      procedure |     entry;

ID ::=          DEF_ID |        USED_ID;

ITEM ::=        subprogram_body |       task_body |     DECL;

NAME ::=        DESIGNATOR |    function_call;

OBJECT_DEF ::=  EXP_VOID;

OP ::=          USED_OP;

PARAM ::=       in |    in_out |        out;

PARAM_ASSOC ::= EXP |   assoc;

RANGE_R ::=     range | attribute;

STM ::=         null_stm |      assign |        entry_call |
                delay |         abort |         block |
                cond_entry |    timed_entry |   accept |
                select |        terminated;
```

```
SUBPROG_DEF ::= void;

TASK_DEF ::=    task_spec;

TYPE_SPEC ::=   integer |       CONSTRAINED;

UNIT_BODY ::=   subprogram_decl |       subprogram_body |
                void;

USED_ID ::=     used_object_id |        used_name_id |
                used_bltn_id;

USED_OPS ::=    used_op |               used_bltn_op;
```

Added as a result of sm attribute

```
BLOCK_STUB_VOID ::=     block |     stub |      void;

DEF_CHAR ::=            def_char;

DEF_OCCURRANCE ::=      DEF_ID |    DEF_OP |    DEF_CHAR;

DEF_OP ::=      def_op;

FORMAL_SUBPROG_DEF ::=  NAME |      box |       no_default;

LANGUAGE ::=    arguement_id;

LOCATION ::= EXP_VOID;

SUBP_BODY_DECS ::=       block | stub | FORMAL_SUBPROG_DEF |
                         void |  LANGUAGE;
```

APPENDIX D

AN ADA EXAMPLE

```
with text_io; use text_io;
procedure main is
        a,b,c,d : integer;

        package int_io is new integer_io (integer);
        use int_io;

        task tl is
                entry tla(o : out integer);
        end tl;

        task t2 is
                entry f_a(x : out integer);
                entry t2a(p : out integer);
        end t2;

        task body tl is
                f,g,h : integer;

                task t3 is
                        entry f_f(y : out integer);
                end t3;

                task body t3 is
                        i,j,m : integer;
                begin
                        i := 20;
                        j := 25;
                        b := i + j;
                        put(b);
                        accept f_f(y : out integer) do
                                y := j + b;
                                m := j + b;
                                put(m);
                        end f_f;
                end t3;
        begin
                g := 30;
                t3.f_f(f);
                h := g + f;
                put(h);
```

```
                accept tla(o : out integer) do
                        o := 200;
                end tla;
                t2.t2a(h);
                put(h);
        end tl;

        task body t2 is
                k,l,n : integer;
        begin
                k := 5;
                l := 10;
                accept f_a(x : out integer) do
                        x := k + l;
                        n := k + l;
                        put(n);
                end f_a;
                tl.tla(n);
                put(n);
                accept t2a(p : out integer) do
                        p := 300;
                end t2a;
        end t2;

begin
        t2.f_a(a);
        d := a - b;
        put(d);
        c := a + b;
        put(c);
end main;
```

APPENDIX E

AN DIANA EXAMPLE

```
A01: compilation [ as_list < A02^ > ]

A02:  comp_unit [ as_unit_body A03^ ]

A03:    subprogram_body [
           as_header     A04^;
           as_designator A06^;
           as_block_stub A07^   ]

A04:     procedure [ as_param_s A05^ ]

A05:      param_s [ as_list <> ]

A06:     proc_id [ lx_symrep "main" ]

A07:     block [
            as_item_s A08^;
            as_stm_s  A20^ ]

A08:      item_s [ as_list < A09^ B00^ C00^ B33^ C33^ >  ]

A09:       var [
              as_id_s     A11^;
              as_type_spec  A16^;
              as_object_def A10^   ]

A10:        void [ ]

A11:        id_s [ as_list < A12^ A13^ A14^ A15^ > ]

A12:         var_id [ lx_symrep "a" ]

A13:         var_id [ lx_symrep "b" ]

A14:         var_id [ lx_symrep "c" ]

A15:         var_id [ lx_symrep "d" ]

A16:        constrained [
                as_name       A17^
                as_constraint void ]
```

```
A17:          used_name_id [ lx_symrep "integer" ]


B00:        task_decl [
               as_id        B01^;
               as_task_def B02^  ]

B01:         var_id [ lx_symrep "t1" ]

B02:         task_spec [ as_decl_s B03^ ]

B03:          decl_s [ as_list < B04^ > ]

B04:           subprogram_decl [
                  as_designator     B05^;
                  as_header         B06^;
                  as_subprogram_def void;   ]

B05:            entry_id [ lx_symrep "t1a"   ]

B06:            entry [
                   as_dscrt_range_void void;
                   as_param_s          B08^   ]

B08:             param_s [ as_list < B09^ > ]

B09:              out [
                     as_id_s     B10^;
                     as_name     B12^;
                     as_exp_void void   ]

B10:               id_s [ as_list < B11^ > ]

B11:                out_id [ lx_symrep "o" ]

B12:               used_name_id [ lx_symrep "integer" ]


C00:        task_decl [
               as_id        C01^;
               as_task_def C02^  ]

C01:         var_id [ lx_symrep "t2" ]

C02:         task_spec [ as_decl_s C03^ ]

C03:          decl_s [ as_list < C04^ C14^ > ]

C04:           subprogram_decl [
                  as_designator     C05^;
                  as_header         C06^;
                  as_subprogram_def void;   ]

C05:            entry_id [ lx_symrep "f_a"   ]
```

```
C06:            entry [
                   as_dscrt_range_void void;
                   as_param_s          C08^   ]

C08:             param_s [ as_list < C09^ > ]

C09:              out [
                     as_id_s     C10^;
                     as_name     C12^;
                     as_exp_void void   ]

C10:               id_s [ as_list < C11^ > ]

C11:                out_id [ lx_symrep "x" ]

C12:               used_name_id [ lx_symrep "integer" ]

C14:           subprogram_decl [
                  as_designator     C15^;
                  as_header         C16^;
                  as_subprogram_def void;   ]

C15:            entry_id [ lx_symrep "t2a"   ]

C16:            entry [
                   as_dscrt_range_void void;
                   as_param_s          C18^   ]

C18:             param_s [ as_list < C19^ > ]

C19:              out [
                     as_id_s     C20^;
                     as_name     C22^;
                     as_exp_void void   ]

C20:               id_s [ as_list < C21^ > ]

C21:                out_id [ lx_symrep "p" ]

C22:               used_name_id [ lx_symrep "integer" ]

B33:        task_body [
               as_id         B34^;
               as_block_stub B35^   ]

B34:         task_body_id [ lx_symrep "tl" ]

B35:         block [
                as_item_s B36^;
                as_stm_s  B117^   ]

B36:          item_s [ as_list < B37^ D00^ D45^ > ]
```

```
B37:          var [
                 as_id_s B38^;
                 as_type_spec B42^;
                 as_object_def void  ]

B38:           id_s [ as_list < B39^ B40^ B41^ >  ]

B39:            var_id [ lx_symrep "f" ]

B40:            var_id [ lx_symrep "g" ]

B41:            var_id [ lx_symrep "h" ]

B42:           constrained [
                  as_name       B43^;
                  as_constraint void ]

B43:            used_name_id [ lx_symrep "integer" ]


D00:           task_decl [
                  as_id       D01^;
                  as_task_def D02^  ]

D01:            var_id [ lx_symrep "t3" ]

D02:            task_spec [ as_decl_s D03^ ]

D03:             decl_s [ as_list < D04^ > ]

D04:              subprogram_decl [
                     as_designator     D05^;
                     as_header         D06^;
                     as_subprogram_def void;  ]

D05:              entry_id [ lx_symrep "f_f"  ]

D06:               entry [
                      as_dscrt_range_void void;
                      as_param_s          D08^  ]

D08:                param_s [ as_list < D09^ > ]

D09:                 out [
                        as_id_s     D10^;
                        as_name     D12^;
                        as_exp_void void  ]

D10:                  id_s [ as_list < D11^ > ]

D11:                   out_id [ lx_symrep "y" ]

D12:                  used_name_id [ lx_symrep "integer" ]
```

```
D45:          task_body [
                as_id          D46^;
                as_block_stub D47^   ]

D46:           task_body_id [ lx_symrep "t3" ]

D47:           block [
                 as_item_s D48^;
                 as_stm_s  D56^   ]

D48:            item_s [ as_list < D49^ > ]

D49:             var [
                   as_id_s        D50^;
                   as_type_spec   D54^;
                   as_object_def void   ]

D50:              id_s [ as_list < D51^ D52^ D53^ > ]

D51:               var_id [ lx_symrep "i" ]

D52:               var_id [ lx_symrep "j" ]

D53:               var_id [ lx_symrep "m" ]

D54:              constrained [
                    as_name        D55^;
                    as_constraint void ]

D55:              used_name_id [ lx_symrep "integer" ]

D56:           stm_s [as_list < D57^ D60^ D63^ D70^ D71^ >]

D57:             assign [
                   as_name D58^;
                   as_exp  D59^   ]

D58:              used_name_id [ lx_symrep "i" ]

D59:              numeric_literal [ lx_numrep "20" ]

D60:             assign [
                   as_name D61^;
                   as_exp  D62^   ]

D61:              used_name_id [ lx_symrep "j" ]

D62:              numeric_literal [ lx_numrep "25" ]

D63:            assign [
                   as_name D64^;
                   as_exp  D65^   ]

D64:              used_name_id [ lx_symrep "b" ]
```

```
D65:                 function_call [
                        as_name            D66^;
                        as_param_assoc_s D67^   ]

D66:                  used_bltn_op [ lx_symrep "+"   ]

D67:                  param_assoc_s [ as_list < D68^ D69^ > ]

D68:                   used_name_id [ lx_symrep "i" ]

D69:                   used_name_id [ lx_symrep "j" ]

D70:                out_put [ lx_symrep "b" ]

D71:                accept [
                       as_name     D72^;
                       as_param_s  D73^;
                       as_stm_s    D77^   ]

D72:                used_name_id [ lx_symrep "f_f" ]

D73:                param_s [ as_list < D74^ > ]

D74:                 out [
                        as_id_s      D75^;
                        as_name      D76^;
                        as_exp_void void ]

D75:                  out_id [ lx_symrep "y" ]

D76:                  used_name_id [ lx_symrep "integer" ]

D77:                stm_s [ as_list < D78^ D85^ D92^ > ]

D78:                 assign [
                        as_name D79^;
                        as_exp  D80^   ]

D79:                  used_name_id [ lx_symrep "y" ]

D80:                  function_call [
                         as_name            D81^;
                         as_param_assoc_s D82^   ]

D81:                   used_bltn_op [ lx_symrep "+"   ]

D82:                   param_assoc_s [ as_list < D83^ D84^ > ]

D83:                    used_name_id [ lx_symrep "j" ]

D84:                    used_name_id [ lx_symrep "b" ]
```

```
D85:                  assign [
                         as_name D86^;
                         as_exp  D87^  ]

D86:                   used_name_id [ lx_symrep "m" ]

D87:                   function_call [
                          as_name          D88^;
                          as_param_assoc_s D89^  ]

D88:                    used_bltn_op [ lx_symrep "+"  ]

D89:                    param_assoc_s [ as_list < D90^ D91^ > ]

D90:                     used_name_id [ lx_symrep "j" ]

D91:                     used_name_id [ lx_symrep "b" ]

D92:                  out_put [ lx_symrep "m" ]

B117: stm_s [as_list< B118^B121^B127^B134^B135^B145^B151^ >]

B118:         assign [
                 as_name B119^;
                 as_exp  B120^  ]

B119:          used_name_id [ lx_symrep "g" ]

B120:          numeric_literal [ lx_numrep "30" ]

B121:         entry_call [
                 as_name          B124^;
                 as_param_assoc_s B122^  ]

B122:          param_assoc_s [ as_list < B123^ > ]

B123:           used_name_id [ lx_symrep "f" ]

B124:          selected [
                  as_name            B125^;
                  as_designator_char B126^  ]

B125:           used_name_id [ lx_symrep "t3" ]

B126:           used_name_id [ lx_symrep "f_f" ]

B127:         assign [
                 as_name B128^;
                 as_exp  B129^  ]

B128:          used_name_id [ lx_symrep "h" ]
```

```
B129:         function_call [
                 as_name           B130^;
                 as_param_assoc_s  B131^   ]

B130:          used_bltn_op [ lx_symrep "+"   ]

B131:          param_assoc_s [ as_list < B132^ B133^ > ]

B132:           used_name_id [ lx_symrep "g" ]

B133:           used_name_id [ lx_symrep "f" ]

B134:        out_put [ lx_symrep "h" ]

B135:        accept [
                as_name    B136^;
                as_param_s B137^;
                as_stm_s   B141^   ]

B136:         used_name_id [ lx_symrep "tla" ]

B137:         param_s [ as_list < B138^ > ]

B138:          out [
                  as_id_s     B139^;
                  as_name     B140^;
                  as_exp_void void ]

B139:           out_id [ lx_symrep "o" ]

B140:           used_name_id [ lx_symrep "integer" ]

B141:          stm_s [ as_list < B142^ > ]

B142:           assign [
                   as_name B143^;
                   as_exp  B144^   ]

B143:             used_name_id [ lx_symrep "o" ]

B144:             numeric_literal [ lx_numrep "200" ]

B145:        entry_call [
                as_name           B148^;
                as_param_assoc_s  B146^   ]

B146:          param_assoc_s [ as_list < B147^ > ]

B147:           used_name_id [ lx_symrep "h" ]

B148:          selected [
                  as_name              B149^;
                  as_designator_char   B150^   ]
```

```
B149:             used_name_id [ lx_symrep "t2" ]

B150:             used_name_id [ lx_symrep "t2a" ]

B151:         out_put [ lx_symrep "h" ]


C33:       task_body [
              as_id          C34^;
              as_block_stub C35^  ]

C34:        task_body_id [ lx_symrep "t2" ]

C35:        block [
               as_item_s C36^;
               as_stm_s  C45^  ]

C36:         item_s [ as_list < C37^ > ]

C37:          var [
                 as_id_s C38^;
                 as_type_spec C42^;
                 as_object_def void  ]

C38:           id_s [ as_list < C39^ C40^ C41^ >  ]

C39:            var_id [ lx_symrep "k" ]

C40:            var_id [ lx_symrep "l" ]

C41:            var_id [ lx_symrep "n" ]

C42:           constrained [
                  as_name       C43^;
                  as_constraint void ]

C43:            used_name_id [ lx_symrep "integer" ]

C45:         stm_s [as_list < C46^C49^C71^C101^C111^C125^ >]

C46:          assign [
                 as_name C47^;
                 as_exp  C48^  ]

C47:           used_name_id [ lx_symrep "k" ]

C48:           numeric_literal [ lx_numrep "5" ]

C49:          assign [
                  as_name C50^;
                  as_exp  C51^  ]

C50:            used_name_id [ lx_symrep "l" ]
```

```
C51:            numeric_literal [ lx_numrep "10" ]

C71:           accept [
                   as_name    C72^;
                   as_param_s C73^;
                   as_stm_s   C77^  ]

C72:            used_name_id [ lx_symrep "f_a" ]

C73:            param_s [ as_list < C74^ > ]

C74:             out [
                    as_id_s      C75^;
                    as_name      C76^;
                    as_exp_void void ]

C75:              out_id [ lx_symrep "x" ]

C76:              used_name_id [ lx_symrep "integer" ]

C77:            stm_s [ as_list < C78^ C85^ C92^ > ]

C78:             assign [
                    as_name C79^;
                    as_exp  C80^  ]

C79:              used_name_id [ lx_symrep "x" ]

C80:              function_call [
                     as_name           C81^;
                     as_param_assoc_s C82^  ]

C81:               used_bltn_op [ lx_symrep "+"  ]

C82:               param_assoc_s [ as_list < C83^ C84^ > ]

C83:                used_name_id [ lx_symrep "k" ]

C84:                numeric_literal [ lx_numrep "1" ]

C85:             assign [
                    as_name C86^;
                    as_exp  C87^  ]

C86:              used_name_id [ lx_symrep "n" ]

C87:              function_call [
                     as_name           C88^;
                     as_param_assoc_s C89^  ]

C88:               used_bltn_op [ lx_symrep "+"  ]

C89:               param_assoc_s [ as_list < C90^ C91^ > ]
```

```
C90:                 used_name_id [ lx_symrep "k" ]

C91:                 numeric_literal [ lx_numrep "1" ]

C92:              out_put [ lx_symrep "n" ]

C101:           entry_call [
                   as_name           C108^;
                   as_param_assoc_s C106^  ]

C106:            param_assoc_s [ as_list < C107^ > ]

C107:             used_name_id [ lx_symrep "n" ]

C108:            selected [
                    as_name             C109^;
                    as_designator_char C110^  ]

C109:             used_name_id [ lx_symrep "t1" ]

C110:             used_name_id [ lx_symrep "t1a" ]

C111:           out_put [ lx_symrep "n" ]

C125:           accept [
                   as_name    C126^;
                   as_param_s C127^;
                   as_stm_s   C131^  ]

C126:            used_name_id [ lx_symrep "t2a" ]

C127:            param_s [ as_list < C128^ > ]

C128:             out [
                     as_id_s     C129^;
                     as_name     C130^;
                     as_exp_void void ]

C129:              out_id [ lx_symrep "p" ]

C130:              used_name_id [ lx_symrep "integer" ]

C131:            stm_s [ as_list < C132^ > ]

C132:             assign [
                     as_name C132^;
                     as_exp  C133^  ]

C132:              used_name_id [ lx_symrep "p" ]

C133:              numeric_literal [ lx_numrep "300" ]
```

```
A20:     stm_s [ as_list < A21^ A31^ A38^ A41^ A48^ >  ]

A21:      entry_call [
             as_name           A28^;
             as_param_assoc_s A26^  ]

A26:       param_assoc_s [ as_list < A27^ > ]

A27:        used_name_id [ lx_symrep "a" ]

A28:       selected [
              as_name                A29^;
              as_designator_char A30^  ]

A29:        used_name_id [ lx_symrep "t2" ]

A30:        used_name_id [ lx_symrep "f_a" ]

A31:      assign [
             as_name A32^;
             as_exp  A33^  ]

A32:       used_name_id [ lx_symrep "d" ]

A33:       function_call [
              as_name           A34^;
              as_param_assoc_s A35^  ]

A34:        used_bltn_op [ lx_symrep "-"  ]

A35:        param_assoc_s [ as_list < A36^ A37^ > ]

A36:         used_name_id [ lx_symrep "a" ]

A37:         used_name_id [ lx_symrep "b" ]

A38:      out_put [ lx_symrep "d" ]

A41:      assign [
             as_name A42^;
             as_exp  A43^  ]

A42:       used_name_id [ lx_symrep "c" ]

A43:       function_call [
              as_name           A44^;
              as_param_assoc_s A45^  ]

A44:        used_bltn_op [ lx_symrep "+"  ]

A45:        param_assoc_s [ as_list < A46^ A47^ > ]

A46:         used_name_id [ lx_symrep "a" ]
```

```
A47:          used_name_id [ lx_symrep "b" ]
A48:       out_put [ lx_symrep "c" ]
```

APPENDIX F

AN CAL EXAMPLE

```
.dp     equ   9     * display pointer
.ts     equ  10     * task pointer for stack
.tt     equ  11     * task pointer for table
.sp     equ  12     * stack pointer
.fp     equ  13     * top of stack pointer
.ap     equ  14     * arguement pointer for pass by value
 pure
 align 4
 entry _main
_main   equ *
*
*  Envirement set up
*
.RS     equ 104     * table register save area
.TL     equ  96     * length of task header
.NT     equ   3     * number of task
*
*                   Main storage area
.VL1    equ  16     * length of variable storage a,b,c,d
.EL1    equ   0     * length of entry variable storage
.DL1    equ   4     * length of desplay storage
.RL1    equ  40     * register save area
.AL1    equ  8      * parameter save area
.FL1    equ  .VL1+.DL1+.RL1+.AL1+.EL1 * size of stack
*
.EP1    equ  .DL1+.VL1 location of entry storage in its stack
.AP1    equ  .EP1+.EL1 location of arguement pt in its stack
.RP1    equ  .AP1+.AL1 location of register save in its stack
.TP1    equ  .NT*.TL+.TL+.RS  stack pointer for parent
.T1     equ  .TL*0+.RS        location of task in task table
*
*                   Task 1 storage area
.VL2    equ  12               * f,g,h
.EL2    equ   4
.DL2    equ  8
.RL2    equ  40
.AL2    equ  8
.FL2    equ  .VL2+.DL2+.RL2+.AL2+.EL2
```

```
*
.EP2    equ   .DL2+.VL2
.AP2    equ   .EP2+.EL2
.RP2    equ   .AP2+.AL2
.TP2    equ   .TP1+.FL1
.T2     equ   .TL*1+.RS
*
*                   Task 3 storage area
.VL3    equ   8                   * i,j
.EL3    equ    4
.DL3    equ   12
.RL3    equ   40
.AL3    equ   8
.FL3    equ   .VL3+.DL3+.RL3+.AL3+.EL3
*
.EP3    equ   .DL3+.VL3
.AP3    equ   .EP3+.EL3
.RP3    equ   .AP3+.AL3
.TP3    equ   .TP2+.FL2
.T3     equ   .TL*2+.RS
*
*                   Task 2 storage area
.VL4    equ   8                   * k,l
.EL4    equ   8
.DL4    equ   8
.RL4    equ   40
.AL4    equ   8
.FL4    equ   .VL4+.DL4+.RL4+.AL4+.EL4
*
.EP4    equ   .DL4+.VL4
.AP4    equ   .EP4+.EL4
.RP4    equ   .AP4+.AL4
.TP4    equ   .TP3+.FL3
.T4     equ   .TL*3+.RS
*
.FS     equ   .TP1+.FL1+.FL2+.FL3+.FL4   * total size of stack
*
*    Set up task table
*
 lr     .sp,.fp
 ai     .fp,.FS
 stm    10,12(.sp)
*
*                     make all tasks active
 li     2,0
 st     2,.T1+0(.sp) * main
 st     2,.T2+0(.sp) * task 1
 st     2,.T3+0(.sp) * task 3
 st     2,.T4+0(.sp) * task 2
*
```

```
*                       set up prority
 li     2,1              * main
 st     2,.T1+4(.sp)
 li     2,3              * task 1
 st     2,.T2+4(.sp)
 li     2,0              * task 3
 st     2,.T3+4(.sp)
 li     2,2              * task 2
 st     2,.T4+4(.sp)
*
*
*                      set up # of children
 li     2,2
 st     2,.T1+16(.sp)
 li     2,1
 st     2,.T2+16(.sp)
 li     2,0
 st     2,.T3+16(.sp)
 li     2,0
 st     2,.T4+16(.sp)
*
*
*                     set up parent address
 li     2,.T1(.sp)
 st     2,.T1+20(.sp)
 li     2,.T1(.sp)
 st     2,.T2+20(.sp)
 li     2,.T2(.sp)
 st     2,.T3+20(.sp)
 li     2,.T1(.sp)
 st     2,.T4+20(.sp)
*
*   initialize task stack pointers and jump addresses
 li    .ts,.TP1(.sp)    * main
 st    .ts,.T1+12(.sp)
 li    2,_TASKM
 st    2,.T1+8(.sp)
 li    .ts,.TP2(.sp)    * task 1
 st    .ts,.T2+12(.sp)
 li    2,_TASK1
 st    2,.T2+8(.sp)
 li    .ts,.TP3(.sp)    * task 3
 st    .ts,.T3+12(.sp)
 li    2,_TASK3
 st    2,.T3+8(.sp)
 li    .ts,.TP4(.sp)    * task 2
 st    .ts,.T4+12(.sp)
 li    2,_TASK2
 st    2,.T4+8(.sp)
*
```

```
*       Set up display pointers
*
 l       .ts,.T1+12(.sp) * main
 st      .ts,0(.ts)
 l       .ts,.T2+12(.sp) * task 1
 st      .ts,0(.ts)
 l       2,.T1+12(.sp)
 st      2,4(.ts)
 l       .ts,.T3+12(.sp) * task 3
 st      .ts,0(.ts)
 l       2,.T2+12(.sp)
 st      2,4(.ts)
 l       2,.T1+12(.sp)
 st      2,8(.ts)
 l       .ts,.T4+12(.sp) * task 2
 st      .ts,0(.ts)
 l       2,.T1+12(.sp)
 st      2,4(.ts)
*
*       Set up Queue
 li      2,0
 st      2,_task_c
 li      2,.T1+0(.sp)
 st      2,_task_pt
 b       Q10
Q05     equ   *
 l       .tt,_task_pt
 li      2,36(.tt)
 st      2,28(.tt)
 st      2,32(.tt)
*
 li      3,0
 li      2,48(.tt)
 st      3,0(2)
 st      3,4(2)
 st      2,44(.tt)
 li      2,60(.tt)
 st      3,0(2)
 st      3,4(2)
 st      2,56(.tt)
 li      2,72(.tt)
 st      3,0(2)
 st      3,4(2)
 st      2,68(.tt)
 li      2,84(.tt)
 st      3,0(2)
 st      3,4(2)
 st      2,80(.tt)
 li      2,36(.tt)
 st      3,0(2)
 st      3,4(2)
 li      2,92(.tt)
 li      2,.TL
 am      2,_task_pt
```

```
 li     2,1
 am     2,_task_c
Q10     equ   *
 l      2,_task_c
 ci     2,.NT
 bnp    Q05
*
 b      _run_n_task       end of set up
*
*
*
 pure           * start chile task 2
 align 4
 entry _TASK2
_TASK2 equ     *
*
 li     2,5          * K+L -> A    5 + 10 = 15
 l      .dp,0(.ts)
 st     2,8(.dp)
 li     2,10
 l      .dp,0(.ts)
 st     2,12(.dp)
*
T21     equ    *
 li     0,1
 bal    15,_sched
_accept21  equ   *
*
 l      .dp,0(.ts)
 l      2,8(.dp)
 l      .dp,0(.ts)
 a      2,12(.dp)
 l      .dp,0(.ts)
 st     2,.EP4+0(.dp)
*                                  Print out A using printf()
 l      2,.EP4+0(.dp)
 st     2,.AP4+4(.ts)
 li     2,L20
 st     2,.AP4+0(.ts)
 stm    10,.RP4+0(.ts)
 la     .ap,.AP4+0(.ts)
 bal    15,_printf
 lm     10,.AL4+0(.ap)
*
 l      .dp,0(.ts)
 li     .ap,.EP4+0(.dp)
 li     0,3
 bal    15,_sched
_ende21  equ   *
*
```

```
*
 li     1,.T2+0(.sp) * table address for entry
 l      .dp,0(.ts)
 li     .ap,12(.dp)
 li     0,2
 bal    15,_sched
_call21  equ   *
*                                    Print out l using printf()
 l      .dp,0(.ts)
 l      2,12(.dp)
 st     2,.AP4+4(.ts)
 li     2,L70
 st     2,.AP4+0(.ts)
 stm    10,.RP4+0(.ts)
 la     .ap,.AP4+0(.ts)
 bal    15,_printf
 lm     10,.AL4+0(.ap)
*
T22     equ   *
 li     0,1
 bal    15,_sched
_accept22  equ  *
*
 li     2,300
 l      .dp,0(.ts)
 st     2,.EP4+4(.dp)
 li     .ap,.EP4+4(.dp)
 li     0,3
 bal    15,_sched
_ende22  equ   *
*
_TASK2A equ *   task termination
 li     0,0
 bal    15,_sched
 b      _TASK2A
*
*
 pure          *  start task 3
 align 4
 entry _TASK3
_TASK3 equ   *
*
 li     2,20         * I+J -> B      20 + 25 = 45
 l      .dp,0(.ts)
 st     2,12(.dp)
 li     2,25
 l      .dp,0(.ts)
 st     2,16(.dp)
 l      .dp,0(.ts)
 l      2,12(.dp)
 l      .dp,0(.ts)
 a      2,16(.dp)
 l      .dp,8(.ts)
 st     2,8(.dp)
```

```
 *                                             Print out B using printf()
  l      .dp,8(.ts)
  l      2,8(.dp)
  st     2,.AP3+4(.ts)
  li     2,L10
  st     2,.AP3+0(.ts)
  stm    10,.RP3+0(.ts)
  la     .ap,.AP3+0(.ts)
  bal    15,_printf
  lm     10,.AL3+0(.ap)
 *
 T31     equ    *
  li     0,1
  bal    15,_sched
 _accept3    equ   *
 *
  l      .dp,0(.ts) *  J+B -> F        25 + 45 = 70
  l     2,16(.dp)
  l      .dp,8(.ts)
  a     2,8(.dp)
  l     .dp,0(.ts)
  st     2,.EP3+0(.dp)
 *                                     Print out F using printf()
  st     2,.AP3+4(.ts)
  li     2,L30
  st     2,.AP3+0(.ts)
  stm    10,.RP3+0(.ts)
  la     .ap,.AP3+0(.ts)
  bal    15,_printf
  lm     10,.AL3+0(.ap)
 *
  l      .dp,0(.ts)
  li     .ap,.EP3+0(.dp)
  li     0,3
  bal    15,_sched
 _ende3  equ   *
 *
 _TASK3A equ *   task termination
  li     0,0
  bal    15,_sched
  b      _TASK3A
 *
 *
  pure        *       start task 1
  align 4
  entry _TASK1
 _TASK1 equ   *
 *
  li     2,30         * G+F -> H        30 + 70 = 100
  l      .dp,0(.ts)
  st     2,12(.dp)
*
```

```
 li     1,.T3+0(.sp)
 l      .dp,0(.ts)
 li     .ap,8(.dp)
 li     0,2
 bal    15,_sched
_call11  equ  *
*
 l      .dp,0(.ts)
 l      2,12(.dp)
 l      .dp,0(.ts)
 a      2,8(.dp)
 l      .dp,0(.ts)
 st     2,16(.dp)
*                              Print out H usin printf()
 l      .dp,0(.ts)
 l      2,16(.dp)
 st     2,.AP2+4(.ts)
 li     2,L40
 st     2,.AP2+0(.ts)
 stm    10,.RP2+0(.ts)
 la     .ap,.AP2+0(.ts)
 bal    15,_printf
 lm     10,.AL2+0(.ap)
*
T11     equ   *
 li     0,1
 bal    15,_sched
_accept1   equ  *
*
 li     2,200
 l      .dp,0(.ts)
 st     2,.EP2+0(.dp)
*
 li     .ap,.EP2+0(.dp)
 li     0,3
 bal    15,_sched
_ende1  equ  *
*
 li     1,.T4+0(.sp)
 l      .dp,0(.ts)
 li     .ap,16(.dp)
 li     0,2
 bal    15,_sched
_call12 equ  *
*                              Print out ha using printf()
 l      .dp,0(.ts)
 l      2,16(.dp)
 st     2,.AP2+4(.ts)
 li     2,L80
 st     2,.AP2+0(.ts)
 stm    10,.RP2+0(.ts)
 la     .ap,.AP2+0(.ts)
 bal    15,_printf
 lm     10,.AL2+0(.ap)
```

```
*
_TASK1A  equ  * task termination
 li    0,0
 bal   15,_sched
 b     _TASK1A
*
*
 pure       *       start main
 align 4
 entry _TASKM
_TASKM equ  *
*
 li    1,.T4+0(.sp)
 l      .dp,0(.ts)
 li    .ap,4(.dp)
 li    0,2
 bal   15,_sched
_callm equ  *
*
 l      .dp,0(.ts) *  A+B -> C      15 + 45 = 60
 l     2,4(.dp)
 l      .dp,0(.ts)
 a     2,8(.dp)
 l     .dp,0(.ts)
 st    2,12(.dp)
 l      .dp,0(.ts) *  A - B -> D    15 + 45 = -30
 l     2,4(.dp)
 l      .dp,0(.ts)
 s     2,8(.dp)
 l     .dp,0(.ts)
 st    2,16(.dp)
*                          Print out D using printf()
 l      .dp,0(.ts)
 l      2,16(.dp)
 st     2,.AP1+4(.ts)
 li     2,L50
 st     2,.AP1+0(.ts)
 stm    10,.RP1+0(.ts)
 la     .ap,.AP1+0(.ts)
 bal    15,_printf
 lm     10,.AL1+0(.ap)
*                        Print out C using printf()
 l      .dp,0(.ts)
 l      2,12(.dp)
 st     2,.AP1+4(.ts)
 li     2,L60
 st     2,.AP1+0(.ts)
 stm    10,.RP1+0(.ts)
 la     .ap,.AP1+0(.ts)
 bal    15,_printf
 lm     10,.AL1+0(.ap)
*
```

```
_TASKMA  equ  * task termination
 li    0,0
 bal   15,_sched
 b     _TASKMA
*
*
 pure
 align 4
 entry _sched
_sched  equ   *
*
 cli     0,0 *  Normal end of task routine
 bne     S050
 l       2,16(.tt)  * check for children tasks
 bp      S005
 li      2,-1       * no children make inactive
 st      2,0(.tt)
 l       2,20(.tt)  * decrease parent task
 li      3,-1
 am      3,16(2)
 b       S010
S005    equ   * keep active children exist
 st      15,8(.tt) * jump address
 li      2,1   *  lower prority
 am      2,4(.tt)
 l       2,4(.tt) * prority check for lowest
 c       2,_prority
 bnp     S010
 li      2,1 * make lower
 am      2,_prority
S010    equ  *
 st      15,8(.tt)  * save jump address for return
 b       _run_n_task
*
S050    equ  *  continue
*
 cli     0,1  * Entry begin call
 bne     S150
 st      15,8(.tt) * store jump address
 l       2,28(.tt) * check to see if queue is empty
 l       3,0(2)
 cli     3,0
 bne     S100
 li      2,1        * queue is empty put to sleep
 st      2,0(.tt)
S100    equ    *
 b       _run_n_task
*
S150    equ   *
*
```

```
 cli    0,2    * Task call to entry
 bne    S200
 st     15,8(.tt)    * store jump address
 st     .ap,24(.tt)  * store arguement pointer
 li     2,1          * put to sleep
 st     2,0(.tt)
 li     2,0          * wake up entry task
 st     2,0(1)
 l      3,32(1)      * load up queue
 cl     3,28(1)      * check fullness of queue
 bne    S160
S170    equ    *
 st     .tt,0(3)   * store calling table task address
 st     .ap,4(3)   * store arguement pointer address
 l      2,8(3)   * load up next empty location in queue tail
 st     2,32(1)
 b      S180
S160    equ   *  empty check for queue
 l      2,0(3)
 cli    2,0
 be     S170
 b      _err1  * too many tasks queued up terminate
S180    equ   *
 b      _run_n_task
*
S200    equ   *
*
 cli   0,3   * Entry end signal
 bne   S250
 li    2,0        * make calling task active
 l     3,28(.tt)
 l     4,0(3)
 st    2,0(4)
 l     2,0(.ap) * transfer call
 l     4,4(3)
 st    2,0(4)
 li    2,0
 st    2,0(3)   * zerro out queue
 l     2,8(3)   * load up next item in queue
 st    2,28(.tt)
 br    15   * return to task and continue
*
S250   equ *
*
 b    _err2
*
```

```
_run_n_task  equu  *
*
 li     2,0         *     "      prority level rotation
 st     2,_prority_c
 b      S300
S400    equ   *  set up for # of tasks
 li     2,.Tl+0(.sp)  * initialize task pointer for rotation
 st     2,_task_pt
 li     2,0         *     "      active task check
 st     2,_task_c
 b      S375
S325    equ   * check active
 l      .tt,_task_pt
 l      2,0(.tt)
 cli    2,0
 bne    S350
 l      2,4(.tt)
 cl     2,_prority_c
 bne    S350
 l      15,8(.tt)   * load up jump address
 l      .ts,12(.tt) *  "   task pointer for stack
 l      .ap,24(.tt) *  "   arguement pointer
 br     15         * run task
S350    equ  *
 li     2,.TL       * check next task
 am     2,_task_pt
 li     2,1
 am     2,_task_c
S375    equ *
 l      2,_task_c
 ci     2,.NT * check # of tasks
 bnp    S325
 li     2,1   * increment next prority
 am     2,_prority_c
S300    equ   *
 l      2,_prority_c
 c      2,_prority
 bnp    S400
 b      S500 * normal termination
*
_errl   equ   *
 li     2,ERRl
 st     2,0(.sp)
 stm    10,8(.sp)
 lr     .ap,.sp
 bal    15,_printf
 lm     10,8(.ap)
 b      S500
*
```

```
_err2   equ   *
 st     0,4(.sp)
 li     2,ERR2
 st     2,0(.sp)
 stm    10,8(.sp)
 lr     .ap,.sp
 bal    15,_printf
 lm     10,8(.ap)
*
S500    equ  *
*-----------------------------------------------------
*     Normal termination
 li    6,L99
 st    6,0(.sp)
 stm   0,36(.sp)
 lr    .ap,.sp
 bal   15,_printf
 lm    0,36(.ap)
*-----------------------------------------------------
*
 lm    10,12(.sp)
 br    15
 impur
 extrn _printf
_prority  equ  *
 dc    a(3)
_task_pt   equ  *
 dc    a(0)
_task_c    equ  *
 dc    a(0)
_prority_c equ  *
 dc    a(0)
ERR1   equ  *
 db    y'a',y'20',y'51',y'75',y'65',y'75',y'65',y'20'
 db    y'6f',y'76',y'65',y'72',y'20',y'66',y'6c',y'6f'
 db    y'77',y'20',y'65',y'72',y'72',y'6f',y'72',y'20'
 db    y'a',y'0'
ERR2   equ  *
 db    y'a',y'20',y'54',y'61',y'73',y'6b',y'20',y'73'
 db    y'63',y'68',y'65',y'64',y'75',y'6c',y'61',y'72'
 db     y'20',y'65',y'72',y'72',y'6f',y'72',y'20',y'25'
 db     y'64',y'a',y'0'
L10    equ  *
 db    y'a',y'62',y'20',y'3d',y'20',y'25',y'64',y'a'
 db    y'0'
L20    equ  *
 db    y'a',y'61',y'20',y'3d',y'20',y'25',y'64',y'a'
 db    y'0'
L30    equ  *
 db    y'a',y'66',y'20',y'3d',y'20',y'25',y'64',y'a'
 db    y'0'
L40    equ  *
 db    y'a',y'68',y'20',y'3d',y'20',y'25',y'64',y'a'
 db    y'0'
```

```
L50     equ   *
 db     y´a´,y´64´,y´20´,y´3d´,y´20´,y´25´,y´64´,y´a´
 db     y´0´
L60     equ   *
 db     y´a´,y´63´,y´20´,y´3d´,y´20´,y´25´,y´64´,y´a´
 db     y´0´
L70     equ   *
 db     y´a´,y´6c´,y´20´,y´3d´,y´20´,y´25´,y´64´,y´a´
 db     y´0´
L80     equ   *
 db     y´a´,y´68´,y´61´,y´20´,y´3d´,y´20´,y´25´,y´64´
 db     y´a´,y´0´
L99    equ   *
 db     y´a´,y´65´,y´6e´,y´64´,y´a´,y´0´
 end
```

VITA 2

DAVID ASA HARDEN

Candidate for the Degree of

Master of Science

Thesis: AN INVESTIGATION OF AN INTERMEDIATE REPRESENTATION FOR A HIGH LEVEL LANGUAGE

Major Field: Computing and Information Science

Biographical:

Personal Data: Born Flushing, New York, October 10, 1950, the son of Elmer David and Cathrine Harden. Made monastic vows in the presence of the Abbot of St. Gregory's Abbey, Shawnee, Oklahoma on August 20, 1981 and received the name Br. Isidore, O.S.B..

Education: Graduated from Walt Whitman High School, Huntingtion Station, New York, in June, 1968; received Associate Degree in Construction Technology from New York State University at Farmingdale in May, 1970; received Bachelor of Science Degree in Architecture from Oklahoma State University in May, 1979; completed requirements for the Master of Science degree at Oklahoma State University, in May, 1988.

Professional Experience: Teaching Assistant, Department of Construction Technology, Oklahoma State University, January, 1979, to May, 1979; Teaching Assistant, Department of Math/Science, St. Gregory's College, Shawnee, Oklahoma, August, 1981 to May 1982; Instructor, St. Gregory's College, August 1985 to present.