DL/CSE/TM04

technical memorandum Daresbury Laboratory

DL/CSE/TM04

ENVIRONMENT FOR RUNNING RTL/2 PROGRAMS ON INTERDATA COMPUTERS

by

W H PURVIS, Daresbury Laboratory

MARCH, 1977

Science Research Council

Daresbury Laboratory

Daresbury, Warrington WA4 4AD

1. INTRODUCTION

This report describes the environment in which an RTL/2 program executes and how to set it up for initial entry to a user's program.

When an RTL/2 program is running, it assumes that certain conventions are obeyed, e.g. that some registers point at certain areas of memory, and that these in turn contain information which will not be violated behind the programs back. Entry to such a program will obviously have to be performed via some code sequence not written in RTL/2 and not therefore bound by these conventions. This report describes this environment for Interdata machines.

The basic aim is to provide the reader with guide lines on how to set up this environment under any operating system. Some comments on other aspects of operating system interfacing have also been included and two Appendices describe the 'CONTROL ROUTINES' and CODE section linkage conventions.

This document should contain all the information a user needs in order to run his own programs under a new operating system; it does not tackle the problem of moving RTL/2 utilities as well.

The reader is assumed to be familiar with the Interdata machines and the documents listed in the Bibliography.

The word CAL is used throughout this report to refer to the assembly language for both 16 and 32 bit Interdata computers.

2. STACKS

2.1 Stack Usage

Any correct RTL/2 program manifests itself at run-time in the form

of nested control operations. This is most obvious in the case of procedure execution; it is only possible to enter a procedure at its head and exit either by obeying a RETURN or ENDPROC statement, which returns control to the calling procedure, or to perform a GOTO to a LABEL variable residing in a data brick, or passed to the procedure, i.e. one that can only have been initialised by a procedure which has already executed in part. The procedure call/return mechanism is explicitly nested and is enforced by the syntax of the language, but the GOTO exit is only verifiable dynamically and may fail since there is no guarantee that the label has been set.

Textual nesting can occur within a procedure, for instance where variables are declared in BLOCKs or FOR loops. These are of no concern since in RTL/2 all stack manipulation is done on procedure entry and exit.

A 'STACK' (last-in first-out list) is used by the RTL/2 object code to hold this nested information. In order to start up an RTL/2 program, we must set up an embryo stack in the appropriate layout. Appendix B contains a general discussion of the object code and in particular describes the stack. Some of that information is repeated below.

Figure 1 shows the layout of a section of stack as it would be utilised by a single procedure. It contains all the regions which may or must be created on procedure entry. The first executable instruction of every procedure body is a call on a control routine which, using parameters embedded in the code, allocates space for local variables and work space on the stack, on top of similar regions already created. On procedure exit this space is de-allocated. Thus as successive nested procedure calls are obeyed, the stack grows. As exits are obeyed, the stack contracts.

The 'LINK CELL' contains all the information needed to control the un-nesting operation. It has two main elements:

(i) The address of the link call for the calling procedure, and

(ii) The program counter value for resumption in the calling procedure.

Via element (i), all current link cells are chained together. The entry to this chain, i.e. the address of the current head link cell is held permanently in register 14. This register is also used to address all local variables, parameters and temporary results used by the procedure. The compiler can calculate the total space needed for these and this is included in the object code to determine the amount of stack space to allocate at procedure entry. If the procedure calls another procedure, then the space needed for the link cell and parameters of the called procedure are also included in the space for this procedure. This is to allow the parameters to be set up before any space checking is done.

2.2 Entering the User's Program

Having established the dynamics of stack utilisation we are now in a position to describe the requirements for setting up the stack for entry to the 'first' RTL/2 procedure of the program. All we need do is generate sufficient of the standard procedure environment as is necessary to match the specification of the procedure. The simplest case is a main procedure with no parameters, where it might be adequate to simply call it by the following sequence:

```
LDAI    10,STACK            ADDRESS OF STACK AREA
BAL     15,MAINPROG         'FIRST' PROCEDURE
```

The value in register 10 is the address at which the new link cell is to be created. The called procedure then calls the 'control routine'

RR01 to save registers and establish the new link cell as the current link cell. The control routine will save registers 14 and 15 in the link cell. The first link cell will therefore have an undefined value in the field which should point to the previous link cell. This is undesirable as the control routine which interprets global GOTO statements (RR03), has to have some way of determining whether the 'target' label is in scope, which it does by scanning the chain of links backwards, looking for a match on register 14. If the label has not been set, no match will be found and it will career off all over core. Thus, a convention has been made that the first link cell will have zero in the link field and that all control routines should check for this when scanning back through the link cell chain. A more suitable entry sequence would therefore be:

```
SAR     14,14               CLEAR LINK POINTER
LDAI    10,STACK            ADDRESS OF STACK
BAL     15,MAINPROG
```

RTL/2 standards require that the main procedure be of the following specifications:

```
ENT PROC RRJOB() ;
```

2.3 Multiprogramming

The above code assumed that the start up code was part of the process which was to execute the RTL/2 program, for by definition a stack characterises a process. If some other process were to be made responsible for setting up RTL/2 stacks, for instance when creating processes dynamically, the address of the stack would have to be picked up via some external agency (e.g. a dynamic store allocator or a list of free stacks). Any parameters would then have to be inserted into the appropriate core

locations and the initial value of register 10 copied into the process register dump area, wherever that might be. This is obviously highly dependent on the operating system. It is not generally possible to do this in RTL/2 code even if the stack is declared as such in an RTL/2 module, since assignments to stacks are not defined in the language. Named RTL/2 stacks may be accessed only by machine code sections.

The method of process parameterisation suggested above is not generally satisfactory since, being local workspace, the parameters are not accessible to other procedures run as part of the process unless they are passed as parameters of each call, which is inefficient. In the next section an alternative method, using SVC DATA bricks, will be described.

2.4 SVC DATA Bricks

An ordinary DATA brick appears only once in core. It may be private off-stack workspace for a particular procedure or group of procedures, or it may act as a common communication area between several processes. It is often necessary to have the ability to create data bricks which, like stacks, are private to and referenced by the same code in each process, thereby preserving the re-entrancy of the code. In RTL/2 such areas are known as SVC DATA bricks. The compiler generates code to access SVC data which is different to that used to access ordinary data. The latter is normally accessed via a symbolic label which may be external. Whichever applies the effect is to define some address which is to be used by all tasks. In the case of SVC data areas, a symbolic label is again used, always external, but in this case the value assigned to this label is not a true address but an offset. All such references are indexed by register 8 which contains the address of the stack base for the current task. Thus the actual address used will differ between tasks. Each task has

its own SVC data areas, held within the stack area. Since the SVC data bricks are held in the stack area, the total size of these bricks will be needed, since register 14 will need to be initialised to point beyond these areas. To facilitate this, a further external symbol has been defined: SVCTOP. This symbol should be defined as the total length of the SVC area. Since there is no method for automatically defining offsets for the SVC areas, it is necessary to provide a small section of CAL code which defines the SVC brick names as being entry points, then contains EQU statements to give these symbols appropriate values. The following gives an example of such a piece of code:

```
          ENTRY RRSIO,RRERR,SVCTOP
RRSIO     EQU   0
*         DAS   1               IN
*         DAS   1               OUT
RRERR     EQU   RRSIO+ADC+ADC
*         DAS   2               ERL
*         DS    2               ERN
*         DAS   1               ERP
SVCTOP    EQU   RRERR+ADC+ADC+ADC+ADC     (ERN ROUNDS UP TO ADC)
          END
```

Care should be taken to ensure that the mappings are correct, in particular, that padding is not ignored. Under normal circumstances, a standard SVC definition module will be adequate for most purposes.

3. CONTROL ROUTINES

Control routines are CAL subroutines which support RTL/2 code at run time. Since floating point operations are provided, either in hardware

or by simulation in software, the functions to be provided are fairly simple. They can be split into four groups:

(a) Procedure entry and exit controls.

(b) Array bound checking.

(c) Variable shifts.

(d) Fixed-point to floating-point conversion and vice-versa.

Access to these routines is provided using a common base register (register 9) which points to a list of branch instructions. The compiler will generate an instruction of the following form:

```
BAL   13,x(9)
```

whenever it needs the assistance of a control routine. While the use of the base register gives no advantage on the 16-bit machines, it does allow the 32-bit machines to use the shorter RX1 format instead of the RX3 format. The saving can be significant in programs of any complexity. Register 13 is always used as the link register when calling control routines as opposed to the normal linkage register (register 15) which is used for procedure calls. Arguments to the control routines are normally passed in registers. For full details of the arguments required, see Appendix A.

The control routines are written in CAL and may be modified to suit any specific needs. The error handling mechanism attempts to follow system standards in that the procedure RRGEL is called for unrecoverable errors. To prevent stack overflow from causing a loop in error recovery the procedure RRGEL should bypass procedure entry checking by using:

```
OPTION(1) ND ;
```

This means that space must be reserved beyond the end of stack to allow RRGEL to save the linkage registers without overwriting code or

data areas. This is allowed for by the compiler which adds space for four address constants (2 byte for 16 bit, 4 byte for 32 bit) onto the end of compiler generated stacks. As an example, the following RTL/2 code:

```
STACK MYSTK 100 ;
```

would generate the following CAL code:

```
MYSTK    DAC   .E1
         DAS   100          100 address words
.E1      DAS   4            overflow area
```

4. INITIALISATION

4.1 The Base Program

The base program is responsible for establishing a suitable environment for the execution of an RTL/2 program. It has to perform three main functions;

(i) stack initialisation and entry to the users program,

(ii) the procedure RRGEL and default settings for the error label ERL and the error procedure ERP in the SVC brick RRERR,

(iii) termination on return from the user program.

4.2 Startup Code

This sets up the RTL/2 working environment and enters the user's main procedure. The stack must be initialised as described previously, along with the registers 8, 9, 10, 14 and 15. When control returns from the user procedure, the termination code should be entered.

4.3 Error Label

RTL/2 system standards require that the SVC data brick RRERR be incorporated in all systems. It is declared:

```
SVC DATA RRERR;
   LABEL ERL;
   INT ERN;
   PROC(INT) ERP;
ENDDATA;
```

In order that the user may be able to GOTO ERL without having to forego whatever system error monitoring facilities are available, the procedure:

```
PROC (INT) RRGEL;
```

will invoke the monitoring before exiting to ERL. The control routines currently call RRGEL in the same manner as any other RTL/2 procedure. RRGEL should normally be provided as part of the support package although a user may wish to provide his own version for special applications. Note that RRGEL may be called for stack overflow conditions, in which case it must not use any local variables or call other procedures unless some means of switching stacks is available. In addition, under these conditions the procedure entry control routine must be bypassed, otherwise an infinite loop will result.

The base program should also initialise ERL and ERP to suitable values. The error label can be set to pass control directly to the termination code if it is assumed that RRGEL has always output some error diagnostics first.

ERP should also output an error message, then return control to the caller.

4.4 Termination

On return from the user's main procedure, or via ERL, the base

program should take some sensible action, such as returning control to the operating system, or halting. If any resources have been allocated dynamically to this task, they should be freed at this point.

4.5 Line Number Tracing

When error messages are being generated, it is very useful if the source line number corresponding to the position of the error can be included in the message. This may be achieved provided the address of the calling routine is known. Each procedure has a prefix which contains the name of the procedure and the offset of the line number table from the start of that procedure. The format of the prefix is as follows:

```
        DC    C'procname'          8 bytes
        DC    Z(LNTAB-PROC)        offset to l.n. table
        DC    Z(STKSIZE)           size needed on stack
PROC    BALR  13,9                 procedure entry routine
```

The procedure entry routine stores the value of register 13 in the new link cell, thereby making it available to the diagnostic routines. This can then be used to access the procedure name and the line number table pointer. The line number table is in the following format:

```
LNTAB   DC    Z(LNEND-LNTAB)       length of table
        DC    Z(entry)             entries,
        ...
LNEND   EQU   *                    end of table
```

Entries in the table are of two forms: line number definitions, and statement offsets. Line number definitions consist of a negative number which corresponds to the current line number when complemented. A statement offset is the position, relative to the start of the procedure, of the first instruction corresponding to the next line number. Thus

consecutive line numbers will be represented by statement offsets. Line number definitions are only used when two adjacent statement offsets would be the same. To determine the line number which corresponds to a given address, first subtract the start address to get the offset. Then scan sequentially down the table. The first entry will always be a line number definition. In this case and whenever line number definitions are met, set that line number to that value. For each statement offset, compare the known offset with that value; if the offset is less than the current value then the search is complete and the line number is known, otherwise increment the line number and repeat the process with the next entry. Note that the table is always present but will have no entries unless the TR option was specified during compilation of that procedure. In this case the length of the table will be set to 2. If the end of the table is reached without the condition above being met then either the TR option was not specified or the error occurred in the last statement of the procedure and the line number reached may be used.

5.. STANDARDS

The implementor should always make an attempt to provide facilities in his package which conform to the RTL/2 recommended standards. The stream I/O library is written in RTL/2 and can be implemented very quickly provided procedures can be provided to match IN and OUT.

Certain systems will not require these facilities but the error handling conventions should be followed closely. Deviations from these standards may cause problems with later releases of the software.

6. BIBLIOGRAPHY

(i) RTL/2 language specification, June 1974
RTL/2 reference 1 version 2
This document is the authoritative definition of the RTL/2 language.

(ii) Standards for RTL/2 Systems, May 1973
RTL/2 reference 4 version 2
This defines non-IO RTL/2 standards.

(iii) Standard Stream IO for RTL/2 Systems, May 1973
RTL/2 reference 5 version 2
The companion of (ii).

(iv) Interdata Users Manual, February 1973
Publication number 29-261R01
Description of Interdata 16-bit machines.

(v) Interdata 32 Bit Series Reference Manual, June 1974
Publication number 29-365R01
Description of Interdata 32 bit machines.

(vi) Common Assembler Language (CAL) User's Manual
Publication number 29-375R03
Describes the Common Assembler Language used for both 16 and 32 bit machines.

APPENDIX A

CONTROL ROUTINE SPECIFICATIONS

A1 Introduction

These routines are coded using CAL and conform to the following conventions:

Entry is by means of a branch-and-link instruction using R13 as the link register. Arguments may be passed in registers or as in-line constants immediately following the call. In the latter case the control routine will return to the first executable instruction following the constant. Entry points are defined as offsets in an entry vector addressed by register 9. Register 9 must therefore be preserved intact throughout the program.

A2 RRO1 - Procedure Entry Housekeeping

Called at the head of an RTL/2 procedure, i.e. immediately on entry. Establishes the new link cell, and adjusts R14 to point at it. It is assumed that procedures are always called with the following code sequence:

```
        set up parameters (if any)
        LDAI   10,new stack frame
        BAL    15,proc     (or BALR   15,11)
```

where 'new stack frame' is the position of the new link cell on the stack and will be of the form ' n(14) ' where 'n' is some integer, and 'proc' is the name of the procedure being called. The control routine is called by the first instruction of the called procedure as follows:

```
        DC     C'procname'          8 byte name of called proc
        DC     Z(ltoff)
        DC     Z(stksz)             space needed on stack
PROC    BALR   13,9                 call rr01
        start of procedure code
```


Registers 13, 14 and 15 are saved in the new link cell addressed by register 10, then the space needed on the stack is checked. If insufficient space is available, RRGEL(1) is invoked. Otherwise register 14 is made to point at the new link cell and control is returned via register 13.

A3 RRO2 - Procedure Exit

Although the compiler generates a branch and link to this routine, the return address is ignored. Registers 13, 14 and 15 are restored from the current link cell and control is passed back to the user by means of a BR 15 instruction. Since register 15 has just been restored from the stack, control will pass to the procedure that invoked the calling procedure.

Any results to be returned will be in registers 0, 1 or F0 and none of these registers are modified in any way.

A4 RRO3 - Variable GOTO

Whenever a GOTO statement specifies a LABEL variable or expression, the LABEL value is loaded into registers 0 and 1 and RRO3 is invoked. This routine begins a search, starting with the current link cell, and terminating when the head of the link cell chain is reached. At each level, the stack level of the label is compared with the address of the current link cell. If a match is found, then it is assumed that the label was set by the corresponding procedure, register 14 is set to that stack level and a branch taken to the label address. If no match is found before the head of the chain is encountered, then an error has occurred and RRGEL(2) is invoked. An additional check which could be added is to check that the label address lies between the procedure start address (register 13 in the current link cell) and the line number table for that procedure (which may be accessed via register 13 as described in the section on line numbers).

This will prevent errors passing undetected if a procedure call results in a stack frame starting at the same address as a different procedure which set an error label.

A5 RRO4 - Switch Processing

When a switch statement is encountered, the compiler generates code to evaluate the expression then invokes RRO4 to decide where the switch is to go. The branch-and-link instruction is followed by a halfword constant whose value is the number of labels present in the switch list. This is followed by a list of offsets to the labels. The offsets are from the first constant, since the address of this is passed in register 13 by the BAL instruction. The control routine first checks that the value given is in the range 1 - N where N is the number of labels. If this is the case, then the value is doubled and used as index into the list of offsets. The offset is loaded and used in an indexed branch to the selected label. If the value is not in range, the address of the first instruction following the list is calculated and this is branched to instead. This is to conform to the RTL/2 specification whereby a switch statement with expression out of range is to be ignored.

A6 RRO5, RRO6, RRO7, RRO8, RRO9 - Array Checking

The following set of routines are very similar and will be dealt with together. Their function is to check a subscript in register 11 against the length field of an array whose address is passed in register 4. If the subscript is not positive or is greater than the length of the array, then RRGEL(3) is invoked. If not, the base address is added to the subscript and is returned in register 11 to be used to access the array element. In the case of RRO9, which is used for arrays of records, the length of the record structure is passed as a halfword constant, coded

in line following the BAL instruction. The subscript is already aligned in preparation for the accessing and so the length must also be aligned before comparison may be made.

A7 RR10 - Logical Left Shift

This instruction expects an integer in register 1 and shifts it left by the number of places specified in register 11. If the value in register 11 is greater than 15 then the result will be zero.

A8 RR11 - Logical Right Shift

This routine expects an integer in register 1 and shifts it right by the number of places specified in register 11. If the value in register 11 exceeds 15 then the result will be zero.

A9 RR12 - Logical General Shift

This routine takes the integer passed in register 1 and shifts it left if register 11 is positive and right if register 11 is negative. If the absolute value of register 11 is greater than 15, the result will be zero. This routine branches to RR10 or RR11 depending on the sign of register 11.

A10 RR13 - Arithmetic Left Shift

This routine expects a double length integer in registers 0 and 1 for 16 bit machines, or register 1 for 32 bit machines. This value is shifted left by the number of places specified in register 11. If register 11 exceeds 31 then the result will be zero.

A11 RR14 - Arithmetic Right Shift

This routine expects a double length integer as in RR13 and shifts it to the right by the number of places specified in register 11. If the value in register 11 exceeds 31, the result will be either zero if the

initial value was positive, or -1 if the value was negative.

A12 RR15 - Arithmetic General Shift

This routine tests the sign of register 11. If positive or zero, it branches to RR13, otherwise it branches to RR14.

A13 RR16 - Arithmetic General Shift - Single Length

This routine tests the sign of register 11. If it is positive, then it branches to RR10 to perform a single length left shift. Otherwise, the value in register 11 is complemented and used to shift register 1 right by up to 15 places. If the complemented value exceeds 15, register 1 is shifted right by 15 places, thereby giving a result of 0 or -1 depending on the initial sign of register 1.

A14 RR17 - Fixed-point to Floating-point Conversion

This routine is only required for the 16-bit machines since the hardware of the 32-bit machines provides an instruction for the conversion. The input to this routine is a double length integer in registers 0 and 1. This should be converted to a floating point value and returned in floating point register 0. This routine is used for single length conversions also by expanding the single length value to double length.

A15 RR18 - Floating-point to Fixed-point Conversion

This routine is needed for both 32 and 16-bit machines. The contents of floating point register 0 should be converted to an integer and returned in register 1. Scaling for fractions is performed before this routine is invoked. Note that the value should be rounded to the nearest integer value.

APPENDIX B

CODE CONVENTIONS AND CODE STATEMENTS

B1 Introduction

This section summarises some important conventions of the CAL code used on the Interdata - both compiler generated and hand-written code so that a user may understand the compiled code of his system, where necessary, and may write CODE statements in RTL/2 modules when this is necessary, in order, for example, to access peripheral devices. Within this appendix we use the term 'word' to mean either a half-word if the target machine is a 16-bit machine, or a full-word if the target is a 32-bit machine. The terms 'full-word' and 'half-word' are used to mean 32-bit and 16-bit entities respectively.

We start by describing, in some detail, the layout of the stack at run-time.

B2 The Stack Mechanism

Program workspace is allocated in the stack area starting from the low addressed end and working upwards as procedure calls are nested. The stack data for a given procedure is arranged above a LINKCELL as shown in fig. 1. The link cell contains three words which are used to save registers 13, 14 and 15 at procedure entry and from which these registers are retrieved on procedure exit. These registers contain the following values:

Register 13 contains the return address given by the BALR 13,9 instruction which invoked the control routine RR01. This can be used to determine the address of the procedure for diagnostic purposes.

Register 14 is used to point at the current linkcell. On procedure entry, however, it is still pointing at the linkcell of the calling procedure. RROI will modify it to point at the new linkcell as soon as the registers have been saved in the new linkcell and the space checking has been done. There will be, therefore, a chain of linkcells, running from the current cell, back through all nested calls, to the base program that initiated execution of the users program.

Register 15 is set by the BAL instruction in the calling procedure to the return address. Therefore, procedure exit can return to the point of invocation using this value.

Above the linkcell are stored the parameters (if any) of the current procedure. These are stored in place by the calling procedure before passing control to the current procedure. They may be accessed by the current procedure using an offset indexed by register 14.

Adjacent to the parameters and accessible in the same way are the local variables declared in this procedure. These include variables which are declared within inner blocks and FOR-loop control variables. Note that variables within disjoint blocks will share the same space in this area.

Finally, there is a general work area, containing space for temporary results generated during expression evaluation, and also space for linkcells and parameters of procedures which may be called by the current procedure. This space must be included in the current stack frame to allow procedure calls to be set up without additional checks. This area will overlap with stack frames of called procedures to utilise space efficiently.

Note that although the effect is similar to a push-down stack, the stack pointer (register 14) only changes at procedure entry and exit, all internal stacking being simulated within the compiler which produces references to locations at fixed offsets from this register.

The effects of successive procedure calls and exits is illustrated in fig. 2. Note that a procedure that calls itself recursively will behave in exactly the same fashion, each invocation acquiring the same amount of space on the stack.

It is important that the user be able to estimate stack size correctly. Too little stack space will cause task failure which may happen in unusual and untested circumstances when the procedures in the task are nested to a greater depth than usual. Too much stack is wasteful of core space.

Two techniques may be used:

(i) Initial Over-estimate.

The task may be run initially with an over-large stack. After several test runs the amount of stack space used may be taken as an estimate of the space needed. The runs should include conditions which will cause the program to reach its expected maximum stack depth. If it is not possible to ensure this condition then some excess should be allowed for such a condition to arise during normal running. Note that recursive procedures can use large amounts of stack space under certain conditions.

The amount of space used can be monitored in several ways, the best probably being to extend the procedure entry control routine to keep a check on the maximum amount of space used. Termination may print out this value before stopping. The value should be kept in an SVC brick so

that multitask systems can operate correctly.

An alternative method is to preset all of the stack to some unlikely value. At termination the stack is scanned, from the top, until a word is found that does not contain this value. This is likely to be the highest point reached on the stack. It is advisable to allow a few extra words beyond this since space may have been allocated to a procedure, but not used.

(ii) Accurate Calculation

It is possible to calculate the displacement between successive link cells by inspection of the compiled code, and thereby to calculate (by inspection of all possible routes) the longest stack usage. This is tedious and is not recommended. The method is as follows. Inspect the code generated for each procedure. Where the procedure invokes another procedure, a LHI 10,x(14) instruction will be found preceding the BAL or BALR. The value of x is the distance from the current link cell to the new link cell. These values should be added for each nesting path. In addition, the value given for the stack space needed, in the prefix of the final procedure in the path, should be added. The largest of these totals is the maximum amount of space needed. For recursive calls of procedures, the distance between link cells should be multiplied by the number of times the procedure recurses (if known).

Note that the space occupied by SVC data bricks is also in the stack area and so must be added to the final result.

B3 Register Conventions

The standard usage of the registers in compiled RTL/2 is:

RO is used to hold the most significant half of double length results (16 bit only). It is also used to hold the stack pointer in label

values. It is used in the MOD operator as the remainder of a divide instruction.

R1 is the general purpose register. Normal integer and fraction arithmetic takes place in R1. The address part of label values is normally in R1.

R2 is occasionally used as a temporary work register when non-commutative operations are to be performed.

R3 acts as the base for REF variables. The address is loaded into R3 and references are made using 0(3) as the operand.

R4 is used in array and record accessing. Record addresses are loaded into R4 and accessing is by "x(4)" where "x" is the offset within the record of the selected component. During subscripting operations the array base address is often loaded into R4. If subscript checking is done then R4 will always contain the array base address.

R5 is not used at present.

R6 is not used at present.

R7 is used as the base for EXT or SVC data. The address of the data brick is loaded into R7 so that the element within the brick may be accessed using "x(7)" as the operand. In the case of SVC data the address is set up using "LDAI 7,name(8)" since the name is only an offset within the stack.

R8 is used as stack base pointer. This value should never change during normal execution of an RTL/2 task.

R9 is used to address the entry vector for the control routines.

Again, this register should never change in value.

R10 is used to pass the address of a new stack frame to a called procedure. It may also be changed as a result of operations on the subscript register R11 (16 bit only).

R11 acts as the subscript register. Where possible the subscript value is calculated directly in R11. Complicated expressions will result in the value being calculated in R1 and then being moved to R11. R11 is also used to hold the address of a procedure in calls on procedure variables.

R12 is not used in any way.

R13 is used as link address in control routine calls.

R14 is the current stack frame base register. It is used as base address in accessing parameters, local variables and temporaries.

R15 is used as link register in procedure calls.

N.B. Procedure calls preserve only registers 8, 9, 12 and 14. All other registers can be assumed to change over a procedure call. Control routines follow the same convention. Registers 8, 9, and 12 are preserved only in the sense that they will not be explicitly changed. Register 14 is held on the stack and is restored immediately before return.

B4 RTL/2 Data Formats

Standard RTL/2 data types are implemented on an Interdata machine as follows:

BYTE An RTL/2 byte is represented by a single byte. When in a register, the remainder of the register is set to zero.

INT An RTL/2 integer is always 16 bits in length and is held in standard two's complement form.

FRAC An RTL/2 fraction is a 16-bit halfword. The binary point is assumed to be immediately to the right of the sign bit. Thus the value 0.5BO is HEX 4000 and -1.0BO if HEX 8000.

REAL RTL/2 real values are held in standard Interdata floating-point format. The 32 bits are as follows:

1	sign bit (1 if negative)
2-8	hexadecimal exponent (offset by 64)
9-32	mantissa (normalised, always positive)

Note that the mantissa must always be normalised, the hardware giving very odd results if this is not the case. The exponent represents the power of sixteen by which the mantissa; should be multiplied. Changing the sign of a floating point number is done by simply inverting the sign bit.

REF)
) are all represented by an address 'word'. Byte addressing
PROC) is used throughout.
)
STACK)

LABEL Two 'word's are used, the first containing the address of the stack frame current when the label is set, the second containing the address of the label in the code.

BIG INT In the case of the 16 bit machines, registers 0 and 1 are used to hold double length results. On the 32 bit machines, a single register is sufficient.

FINE INT)
)
BIG FRAC) as for BIG INT.
)
FINE FRAC)

RECORDS RTL/2 records are laid out as a succession of components, each with their own format as defined above and below. The record itself has no extra structure except that padding may be inserted to align the components to the boundaries required by the hardware. Records are always aligned according to the requirements of the longest components.

ARRAYS Arrays are represented by successive elements, prefixed by a halfword length field. The address of an array is the address of a fictitious element zero. This address corresponds to the address of the length field only for arrays of 16-bit elements. For longer element lengths, the address will be a number of bytes preceding the length.

Multidimensional arrays are compiled in the standard RTL/2 style, as arrays of REF arrays as many times as necessary.

B5 RTL/2 Brick Layout

RTL/2 bricks are compiled as follows:

PROC brick:

An RTL/2 PROC consists of three parts: a prefix, the main code area and the line number table. The prefix consists of an eight character field containing the name of the procedure followed by two halfword constants. These define the amount of space needed on the stack and the length of the code area. The latter value is used to calculate the position of the line number table.

The code area then follows. The entry point to the procedure is at the beginning of this area and the first instruction is a BALR 13,9 to enter the procedure entry control routine. This is then followed by code for the first executable statements of the procedure.

The line number table appears immediately following the last instruction of the procedure and consists of a halfword length field followed by a list of halfword entries. If line number trace has not been requested only the length field exists, containing the value 2. Entries in the table consist of line number definitions (values less than zero) and statement offsets (positive values). Determination of the line number is by scanning the table with the offset of the instruction (P). If a negative entry is found then the line number (L) is set to the two's complement of that entry. Otherwise, the value of the entry is compared with P. If the value is less than P the L is incremented. If not, or the end of the table is reached, then the value of L corresponds to the line number needed.

DATA brick:

An RTL/2 DATA brick is compiled with the data laid out in the same order as defined in the RTL/2 text. Bytes may be inserted to ensure correct alignment of items that require halfword or fullword alignment.

SVC and EXT data bricks do not generate any code since they are effectively definitions of data areas not contained within the current module. The positions of items within these bricks is however determined in the same manner as for other DATA bricks.

STACK bricks:

An RTL/2 STACK brick is compiled as an uninitialised area of space with a pointer in the first word to the end of the stack area. Space is

left at the end of the stack for four words to allow RRGEL to be called after a stack overflow. The number in the stack definition determines the number of 'words' of space in the stack. Thus STACK FRED 100 will create a stack of two hundred bytes for a 16 bit machine and four hundred bytes for a 32 bit machine.

SVC PROC bricks:

There is no RTL/2 means of compiling SVC procesures. The code generated by references to SVC procedures is identical to that generated for normal procedures.

B6 Code Statements

This section summarises the main features to be considered when writing code statements in RTL/2 modules designed for Interdata machines. It is assumed that the user is familiar with CAL assembly language in which the statements are written.

Code statements or "code sequence" have syntax which follows the overall standard as described in the RTL/2 Specification Manual thus:

```
codeseq     ::= codeheading codeitem
codeheading ::= digitlist , digitlist ;
codeitem    ::= ISO7-character-other-than-@-or-/
              | @ letitem / name
letitem     ::= name | number | string | comment | separator
```

Thus on the Interdata implementation the characters 'trip1' and 'trip2' of the specification manual are @ and / respectively. The trip characters are used to gain access to the RTL/2 variables of LET names etc. defined in the RTL/2 section of the program, from within the code sequence.

A code statement is, as far as the RTL/2 compiler is concerned, similar in nature to other statements. When it is entered during the execution of a procedure the registers 0 to 7 may be used freely as the compiler assumes that the values contained in these registers will no longer be valid after the code sequence has been executed. Space on the stack will have been reserved but there is no means of determining where this starts. As a result, the recommended technique is to define local variables to be used as temporary work space in the block containing the code sequence. The first number in the codeheading is ignored by the compiler and zero may be used for its value. The second number is used to specify the amount of space needed on the stack. Because of the problem mentioned above, this value will also be zero in most cases.

The code within the code sequence should consist of standard CAL statements, typed one to a line, terminated by @RTL. Access to RTL/2 names etc. may be obtained using the trip characters @ AND / as follows.

Local variables may be accessed using @name(14) since the @name is replaced by the offset of the variable from the start of the current stack frame. Register 14 always points to the start of the current frame and should normally be used as a base register.

Global variables may be accessed by @name/brick where brick is the name of the data brick containing the variable. This is replaced by a CAL expression consisting of 'brickname'+'offset' where 'brickname' is the name of the data brick and 'offset' is the position of the variable within that brick.

Elements of records may be accessed in two ways depending on the record. @name/mode will produce the offset of the element from the start

of the record and this may be used either to add to a CAL expression defining the record if directly addressable or in conjunction with a record address held in a register, e.g. @COMP/MODE(4) where COMP is the name of the element and register 4 contains the address of the record.

FIGURE CAPTIONS

Fig. 1 Layout of a section of stack.

Fig. 2 The effects of successive procedure calls and exits.

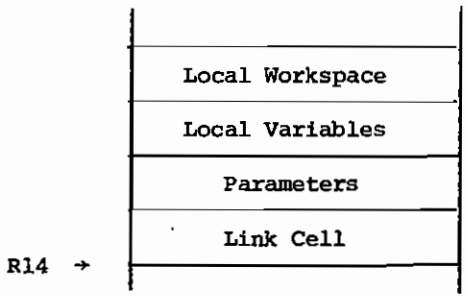
Local Workspace
Local Variables
Parameters
Link Cell
R14

Fig. 1

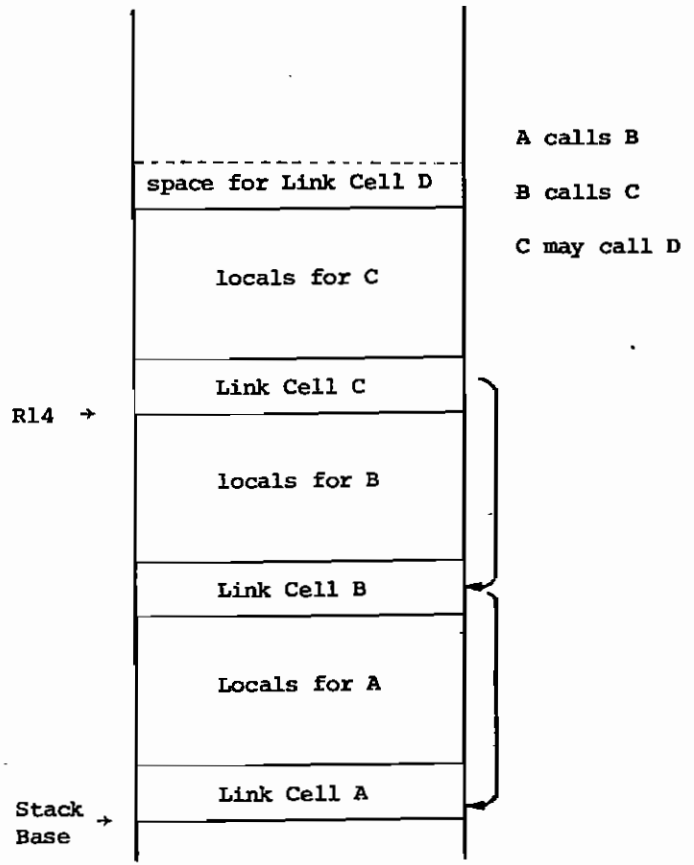
A calls B
B calls C
C may call D
space for Link Cell D
locals for C
Link Cell C
R14
locals for B
Link Cell B
Locals for A
Link Cell A
Stack Base

Fig. 2