RTL/2

Run time environment on P800

THE RTL/2 RUN TIME ENVIRONMENT ON THE PHILIPS P800 SERIES

RTL/2 REFERENCE:69 VERSION:1

Authors: D. Webster, G.C. Stevenson, C.I. Dimmer

Date: 14th April 1976.

Related Documents: Listed in Section 8.

Purpose: To describe the operation of the object code generated by the P856/7 RTL/2 compiler. The body of the document is concerned with establishing a suitable environment. Appendices include specifications of the control routines and code conventions.

Philips Data Systems Publ. Nr.: 5122 991 28131

C O N T E N T S

1. INTRODUCTION

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

1.2. When an RTL/2 program is running it assumes that certain conventions are obeyed, e.g. that some registers point at specific areas of core, and that these in turn contain information which will not be violated behind the program's 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. Obviously this is heavily machine dependent. This document deals with the P800 environment.

1.3. The basic aim is to describe those aspects of RTL/2 program execution common to all implementations on P800 computers. A discussion of operating system interfacing has also been included and two appendices describe the 'control routines' and the code section linkage conventions.

1.4. 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 attempt to tackle the problem of moving the RTL/2 utilities as well.

1.5. The reader is assumed to be familiar with the P800 and the documents listed in the references section.

2. STACKS

2.1. Stack Usage

2.1.1. 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 by performing a GOTO to a label variable residing in a data brick, or passed as a parameter 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.

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

2.1.3. 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 II contains a general discussion of the object code and describes the stack in detail. Some of this information is repeated here.

2.2. Stack Format on P800

2.2.1. Fig.2.1. 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 procedure entry 'control routine' which, using a parameter embedded in the code, allocates space for local variables on the stack, beyond any similar regions already created. On procedure exit this space is de-allocated. Thus, as successive, nested procedure calls are obeyed, the stack expands. As exits are obeyed the stack contracts.

2.2.2. The 'Link-Cell' contains all the information needed to control the un-nesting operation. It has two elements:

i) the address of the link cell for the procedure which called this procedure, and

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

Via element (i), all 'live' link cells are chained together. The entry to this chain, i.e. the address of the currently executing procedure's link cell is held permanently in register A12.

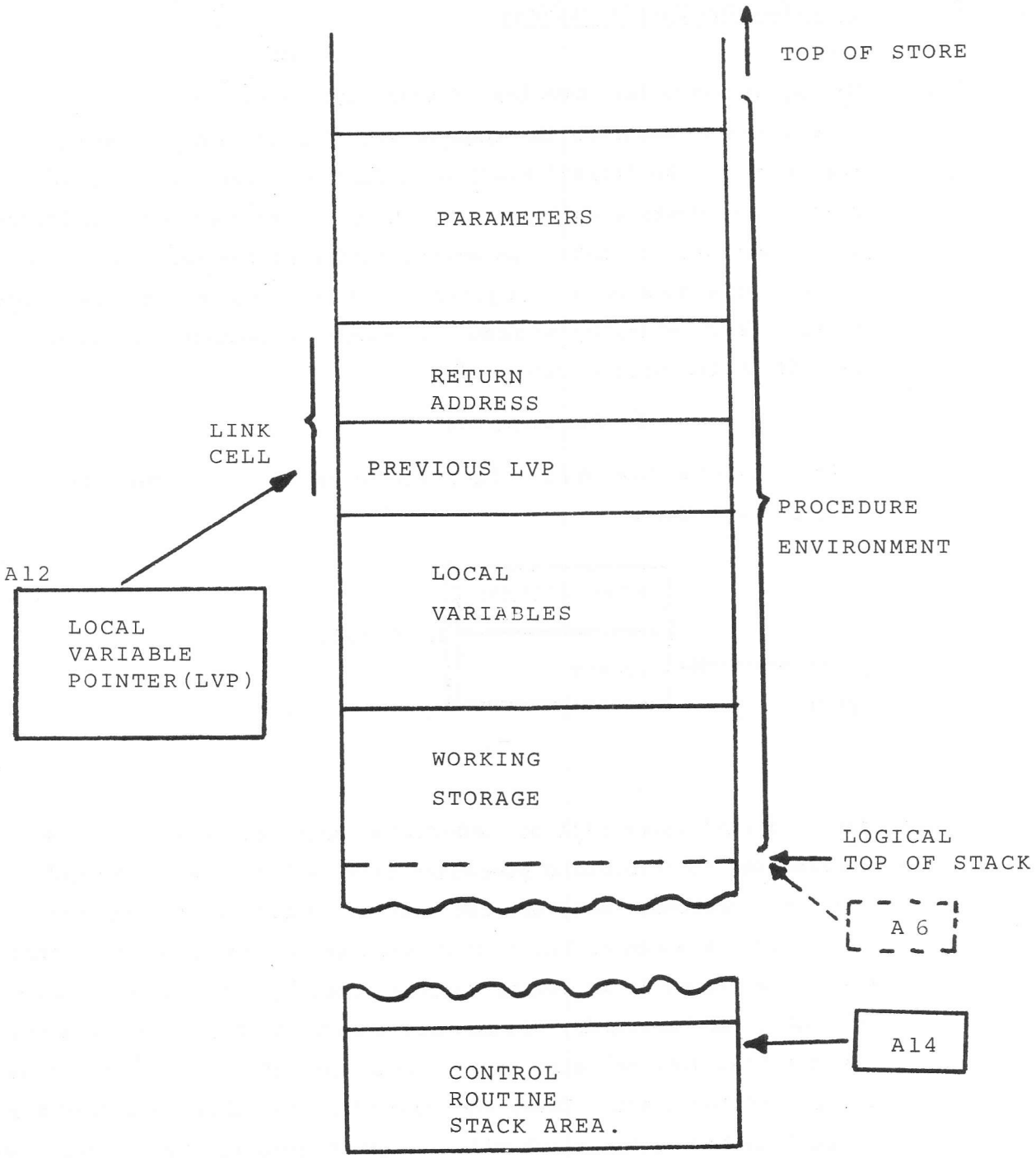


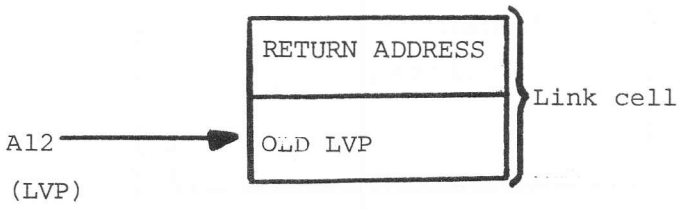
Fig. 2.1. - STACK LAYOUT OF A SINGLE PROC.

2.3. Entering The User's Program

2.3.1. Having examined 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 a program. All we need do is generate sufficient of the standard procedure environment as is necessary to match the specification of the procedure. In the simplest case of a main procedure with no parameters, no local storage, and returning no result it might be adequate to simply call it by the single instruction:

```
CF  A6, MAINPROG
```

which, assuming that A6 has been initialised, leaves the stack looking like this:

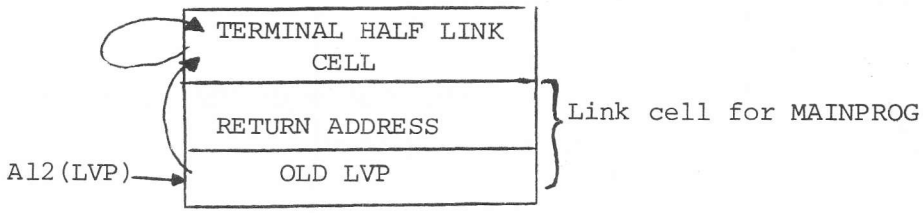


2.3.2. The "Old LVP" value will be undefined, which is fine as far as obeying the corresponding procedure exit back to the 'start-up' code is concerned. However, the control routine which interprets global GOTO statements has to have some way of determining whether the target label is in scope, which it does by scanning the chain of link cells backwards, looking for a match on A12. If the label has not been set, no match will be found and it will not recognise the end of the chain. Thus, the convention has been made that the first location on the stack will always contain its own address, and that A12 will point to it just before entry to the main procedure, thereby positively terminating the chain.

2.3.3. A suitable entry sequence might be:

```
LDR     A12,A6
STR     A12,A6
SUK     A6,2
CF      A6,MAINPROG
```

which, after procedure entry housekeeping, would leave the stack like this:



2.3.4. Currently, all run-time support packages have start-up routines which use this convention. They are independent of the characteristics of the user's programs which must always be entered by a procedure of the same name. Every user's main program would therefore have the following RTL/2 specification:

```
ENT PROC () MAINPROCEDURENAME;
```

Conventionally, the main procedure is called RRJOB in accordance with RTL/2 recommended standards.

2.3.5. Entry With Parameters

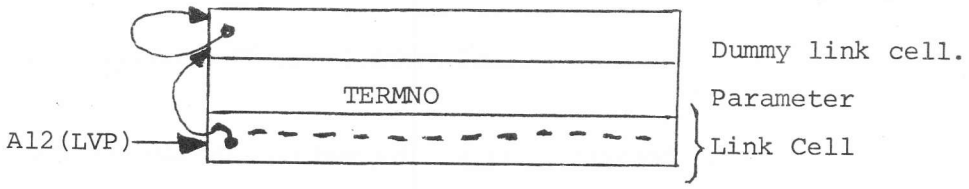
This arrangement is fine for entry to a single program environment but when a system is constructed from several processes it may be desirable to supply some sort of parameter to each. Consider, an interactive system written in RTL/2 supporting several terminals. Since the code is re-entrant one possible (not necessarily the best) method of making the particular terminal number available to the program would be to specify this number as a parameter, thus:

```
ENT PROC MAIN (INT TERMNO);
```

and to enter it via:

```
LDR A12,A6
STR A12,A6
SUK A6,4
ST  A5,2,A6
CF  A6,MAIN
```

(assuming the number to be in A5) leaving the stack so:



Obviously any number of parameters may be treated in this way.

2.4. Multiprogramming

2.4.1. In the above example it has been assumed that the start up code has been run as part of the process which was to execute the RTL/2 program, for by definition a stack characterises a process. If another process were to be made responsible for setting RTL/2 stacks, for instance when creating processes dynamically, the initial values, of A12,A14 would have to be picked up via some external agency (e.g. a dynamic store allocator or a list of spare stacks). Any parameter would then have to be inserted into the appropriate core locations and the initial values of A12,A14 copied into the task's register dump area, wherever that might be. How this is done is obviously system dependent in the extreme. It is not generally possible to do so entirely in RTL/2 code, even if the stack is declared as a STACK brick in an RTL/2 source module, since assignments to stacksare not defined in the language. Named RTL/2 stacks may be accessed (manipulated internally) only by machine code sections.

2.4.2. 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 that process unless they are passed as parameters of each call, which is inefficient. In section 3 an alternative method using SVC data bricks will be described.

2.5. Use of A6,A14

2.5.1. NOTE that A6 is not used as a 'stack pointer' except during procedure entry. All access to parameters, local variables and workspace items is by indexed addressing using register A12, the compiler performing calculation of all offsets from the current link cell.

2.5.2. During procedure entry A6 points to the logical top of stack and part of the workspace area of the calling procedure becomes the parameter area of the called procedure.

2.5.3. A14 points to a logically distinct portion of the stack and is used when calling control routines, and any FORTRAN subroutines used.

3. SVC DATA BRICKS

3.1. An ordinary data brick appears only once in core. It may be private off-stack workspace of 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 stack, are private to and referenced by the same name in each process, thereby preserving the re-entrancy of the code. In RTL/2 such areas are known as SVC DATA bricks. The compiler accesses SVC and ordinary DATA bricks differently. The latter are always addressable via some internally or externally defined symbolic label; the former, of which there may be many instances in a multiprogramming system, have to be addressed via a register, A13 on P800, the administration of which is outside the scope of RTL/2. A13, like A12,A14 has to be set up in assembler for each task in the system.

3.2. The RTL/2 object code references items in SVC data bricks via symbolic offsets with respect to A13.

For example:

```
SVC DATA FRED;
    INT I,J,K;
    REAL R;
ENDDATA;

SVC DATA RRSIO;
    PROC () BYTE IN;
    PROC (BYTE) OUT;
ENDDATA-

PROC JOE ();
    PROC (BYTE) P:=OUT;
END PROC;
```

might lead to the generation of:

```
LD  A4,RRSIO+2,A13
ST  A4,-2,A12     for the assignment.
```

3.3. It is the system constructor's responsibility to specify the mapping of all SVC data bricks by means of assembly language equivalences. For instance, the bricks defined above would be declared:

```
FRED   EQU  0
RRSIO  EQU  FRED+12
```

in some module, which definitions would be used in each program or task.

3.4. To guarantee re-entrancy in all the systems constituent procedures, the SVC DATA area mapping should be the same for any program or task. If the designer can group procedures by process it may be that some core can be saved by 'overlaying' some SVC data; however, this sort of technique is exceedingly dangerous and should be avoided.

3.5. Some SVC DATA bricks are mandatory if the target system is to comply with RTL/2 standards. These are:

RRERR - error handling
RRSIO } stream I/O
RRSED } stream I/O

3.6. The control routines and start-up code released to the user also assume the presence of other bricks, which include the bottom address of the stack for procedure entry checking purposes, and INT items which are used to record the low limit of the stack and the current line number for code compiled with the TR option. Full details of these SVC DATA bricks appear in Appendix IV.

4. CONTROL ROUTINES

4.1. Control routines are assembler subroutines which support RTL/2 generated code at run time. They fall into four main categories:

(i) Procedure entry/exit, GOTO Label variable housekeeping
(ii) Array bound checks and shift count checking.
(iii) REAL arithmetic.
(iv) Type conversions.

4.2. Some of these control routines can be omitted in certain circumstances (refer to Appendix II for details).

4.3. The control routines detect certain unrecoverable errors. In all cases control is transferred to an error handling routine (R:ROO) which simulates an RTL/2 procedure call to RRGEL, which procedure is defined in the RTL/2 System Standards.

4.4. The implementor may vary these error handling conventions if he wishes and is at liberty to 'optimise' procedure entry by removing any or all of three current facilities which are

(i) the stack limit check.
(ii) the stack usage recording.
(iii) workspace base recording.

4.5. The specifications of all the control routines may be found in Appendix II.

4.6. The control routines, although specific to the P800 computers are independent of the operating system.

5. THE BASE PROGRAM

5.1. Base Program Functions

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) initialise the stack and registers for entry to the user's RTL/2 program.

(ii) allocate space for SVC DATA bricks and set SVC DATA items to suitable default values.

(iii) provide the standard error procedure, RRGEL.

The base program is written in assembler and is specific to both the machine and operating system.

5.2. Start-up Code

5.2.1. This sets up the RTL/2 working environment and enters the user's main procedure. The stack must be initialised as described in Section 2, together with A12,A14. A13 must be set up to point at the beginning of the SVC data area as described in Section 3.

5.2.2. When control is returned from the user program some sensible action, such as returning control to the monitor, must be taken.

5.3. SVC DATA bricks

The base program will normally allocate space from the stack area for SVC DATA. Since the base program must initialize the items in this area, it must 'know' the shape and size of all SVC DATA bricks.
The definitions described in Section 3 will usually appear in the base program together with ENTRY directives for SVC DATA brick names.

5.4. RRGEL

5.4.1. 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;
```

5.4.2. In order that the user 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, on detecting an error, call RRGEL in exactly the same way as would an RTL/2 program.

5.4.3. The user must code RRGEL in accordance with his requirements for monitoring. Normally the contents of all the registers will be dumped to a suitable device, along with the error number (the parameter of RRGEL) and possibly the current source code line number if this option is in use. Having done this the base program should GOTO ERL, if ERL is in scope. Since a control routine may fail before the user's RTL/2 program has attempted to set ERL, a default should be inserted by the start-up code. Beware of failures in RRGEL which could cause indefinite looping. Correct monitoring of stack overflow (standard error 1) demands that the limit against which the stack check is made allows an emergency margin always available for the execution of RRGEL.

5.5. Miscellaneous

In single program systems, it may be convenient to include a stack and SVC DATA bricks in the base program, in a ready--initialized state. This may also be possible in a multiprog--ramming environment but care must be taken over the restartability of the code and the re-entrancy of any code to be shared.

6. OPERATING SYSTEM INTERFACE.

It is conventional in P800 series operating systems to use the LKM instruction to enter operating system functions, with parameters in registers A7,A8 and in some cases a result in in A7. An in-line parameter following the LKM defines which function is to be performed.

The RTL/2 compiler for P800 computers compiles an SVC PROC call into such a sequence, and so long as the convention is adhered to then no low-level coding is required to access operating system functions.

For any operating system facility not conforming to this pattern the recommended approach is to write an RTL/2 callable procedure, the body of which presents a completely RTL/2 compatible parameter and result specification. Further run-time support software can then be written using such procedures, and SVC PROC calls, with little recourse to low-level code.

7. STANDARDS

All standard commercial compiler packages attempt to provide facilities according to the RTL/2 recommended standards. The stream I/O formatting procedures are written in applications RTL/2 and can be used immediately providing that suitable procedures to match IN and OUT in RRSIO have been coded (usually in RTL/2 and using the operating system's I/O facilities).

Since IN and OUT imply a 'current stream' in each direction, further procedures are required if programs are to establish associations of streams with peripheral devices or files, and to allow routing of data to and from a number of streams. These procedures and the actual IN and OUT procedures are collectively known as a 'stream I/O support' package, and they are worth providing as a basis for writing portable - or at least transportable - programs.

Certain systems will no doubt not need this I/O capability, but the error handling conventions should always be followed unless the user wishes to modify the control routines in a fundamental way. Too much modification in this area can lead to trouble when subsequent software releases are made.

8. REFERENCES

(i) RTL/2 Language Specification, June 1974

RTL/2 Reference 1, Version 2. (5122 011 28951)
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. (5122 011 28961)
This defines RTL/2 standards.

(iii) Standard Stream I/O for RTL/2 Systems, May 1973.

RTL/2 Reference 5, Version 2. (5122 011 28971)
This expands the definition of the standards for stream I/O.

(iv) P8OOM Programmer's Guide

I.1. INTRODUCTION

This appendix contains a list of features of standard software that the user may wish to modify or re-write from scratch.

I.2. BASE PROGRAM

(i) Start-up.
Initialise stack, A12,A13,A14.
Size of A14 area if to be used for other purposes (e.g. FORTRAN)
Call user's main procedure.
Handle the return and exit to monitor.

(ii) Default ERP and ERL.

(iii) Procedure RRGEL.

I.3. CONTROL ROUTINES

(i) R:R01 - procedure entry
Stack check
Usage recording

(ii) Calls to RRGEL (all made via R:R00)

(iii) SVC DATA definition for access to stack usage monitoring and control items.

(iv) More elaborate debugging aids can be conveniently added to the control routines.
e.g. monitoring of procedure entry/exit, general GOTO's.

I.4. SVC DATA BRICKS

(i) Decide mapping of SVC DATA bricks

(ii) Define this mapping. Usually by global definitions of brick names in the base program but not necessarily so.

I.5. OTHER TOPICS

(i) Having defaulted ERL and ERP in the base program the user may wish to set up these variables to point at RTL/2 quantities.

(ii) The implementor is at liberty to code the procedure RRNUL, RRIPF and RROPF in RTL/2. He may not, normally, do this with RRGEL without dropping into code.

II.1. INTRODUCTION

These routines are assembler-coded subroutines which support the RTL/2 compiled code in a running system.

Some of them may be excluded from an RTL/2 program or system. Section II.6. deals with optional exclusion of various routines.

All the control routines are re-entrant. One copy may be shared among any number of concurrent RTL/2 programs or tasks.

II.2 CONTROL ROUTINE ENTRY

The Control Routines differ from an RTL/2 procedure in the method by which they are called. They cannot be explicitly called in RTL/2 text, although they can of course be called explicitly in CODE sections.

Most of the control routines are entered by:-

```
CF  A14,R:Rnn
```

and return to the compiled code by:-

```
RTN A14
```

Refer to the detailed specifications for exceptions to the above.

II.3. PARAMETERS AND RESULTS

Parameters and results of control routines are generally passed in registers. Some constant parameters are planted in-line in the compiled code as DATA directives.

The control routines operate on various types of data, including the transient double length fixed point forms ("Big" and "fine" forms). Appendix III deals with all data formats.

Byte variables, as in all stack operations, occupy the least significant half of a whole word on the stack, the most significant half being undefined. Variables of more than one word (REAL, big, fine or LABEL) are arranged on the stack in the same address order as they would be in a data brick.

II.4 DETAILED SPECIFICATIONS

In the descriptions which follow the control routines are grouped as they are grouped into separate assembler modules.

The "registers used" are in addition to any for parameters and results, and are stated so that CODE section authors are aware of which registers will be modified when calling control routines from CODE. The control routines assume that A12,A13 and A14 have not been corrupted. For further information on writing CODE sections refer to Appendix III.

Errors are mentioned only briefly in II.4 Full information is given in II.5.

II.4.1. CONTROL ROUTINE ERRORS

Module IDENT: - RTLROO

Control Routine:- R:ROO

II.4.1.1. R:ROO Error handling, control routine errors

Called by:-

(CODE sections or other control routines only - never from compiled code.)

```
CF A14,R:ROO
DATA <error number>
```

Action:-

R:ROO simulates an RTL/2 call on the standard error procedure RRGEL.

Registers Used:-

A6, A9; the former contents which may be of value in understanding the error are saved in the A14 area of the stack.

Exit:-

Via RRGEL to ERL, no direct return to calling sequence.

II.4.2. Procedure Entry and Exit

Module IDENT:- RTLRO1

Control Routines:- R:RO1, R:RO2

II.4.2.1. R:RO1 - Procedure Entry Housekeeping

Procedure calls are compiled into:-

```
⟨parameters into stack⟩
⟨set A6 pointing below parameters⟩

 CF A6,⟨proc⟩
```

Procedures begin:-

```
 CF  A6,  R:RO1
 DATA ⟨n⟩
```

where ⟨n⟩ is the size (in words) of the local variables and the maximum workspace required by the new procedure.

Exit:- to instruction following DATA ⟨n⟩ .

Errors Detected:- Stack overflow - error no.1.

Registers Used:- A5.

II.4.2.2. R:RO2 - Procedure Exit

Called by:-

⟨result if any to A1 ⟶⟩
ABL R:RO2

Procedure results are always passed back in A1 (plus A2, A3 if result is of more than one word.)

Action:- R:RO2 unwinds the stack to the link cell of the calling procedure.

Exit:- to the calling procedure at the instruction following the procedure call.

Registers Used:- A4.

II.4.3. Array Bound Checks

Module IDENT:- RTLRO3

Control Routines:- R:RO3, R:RO4,
R:RO5, R:RO6,
R:RO7.

Function:-

These control routines compute the address of an array element from the array base address and a subscript value. The element address is checked against the bounds of the array and an unrecoverable error results if the address is found to be outside.

Errors Detected:- Array bound check - error no.4.

Result:- (array element address) in A9. Neither A3 nor A10 are modified.

II.4.3.1. R:RO3 - Arrays of BYTEs

R:RO4 - Arrays of INT, FRAC, PROC, REF, and STACK.

R:RO5- Arrays of REALs

R:RO6 - Arrays of LABELs

Called by:-

⟨array base address to A10⟩
⟨array subscript value to A3⟩

CF A14, R:Rnn

Exit:- to instruction following,
CF A14, R:Rnn

Registers Used:- All.

II.4.3.2. R:RO7 - Arrays of records.

Called by:-

⟨array base address to A10⟩
⟨array subscript value to A3⟩

CF A14, R:RO7
DATA ⟨record length (in bytes)⟩

Exit:- to instruction following
DATA ⟨record length⟩

Registers Used:- A1, A2, A11.

II.4.4. Check Shift Count for SHA, SHL

Module IDENT:- RTLRO8

Control Routine:- R:RO8

Function:- Checks shift counts for SHA, SHL instructions.
A maximum count is substituted for any value in excess of the maximum.
Negative shift counts are biassed in readiness for instruction manufacture.

Called by:-

⟨shift count value to A3⟩
CF A14, R:RO8

Exit:- to instruction following the
CF A14, R:RO8
where the following code appears:-

```
ADKL  A3,/3sss
EXR   A3.
```

"sss" depends on the type of shift and the register(s) to be shifted.

II.4.5. General GOTO

Module IDENT:- RTLRO9

Control Routines:- R:RO9,
R:R1O.

II.4.5.1. R:RO9 - Stack Unwind

R:RO9 is only called from R;R1O and other assembler-written code.

Function:-

R;RO9 takes a LABEL value, checks whether the value is in scope. If so it unwinds the stack to the link cell of the procedure activation where the LABEL was set, and goes to the address held in the label.

Called by:-

Label into A1, A2, A3

A1 = target address
A2 = LVP (A12 value)
A3 = Link cell return address

CF A14, R:RO9

Exit:-

to instruction following the,
CF A14, R:RO9 only
if the label is out of scope.

Registers Used:- All.

II.4.5.2. R:R10 - General GOTO.

Function:-

R:R10 uses R:R09 to check the LABEL value and to GOTO it if it is in scope.

Called by:-

⟨Label into A1, A2, A3⟩
(see R:R09 spec)
CF A14, R:R10

Exit:-

Never returns to calling sequence. Either proceeds to LABEL setting or unrecoverable error procedure.

Errors Detected:- LABEL out of scope-
error no. 2.

Registers Used:- See R:R09 spec.

II.4.6. Line Number Trace

Module IDENT:- RTLR11

Function:-

Monitors RTL/2 line numbers when OPTION ()TR in use.

Called by:- CF A14, R:R11
DATA ⟨line number⟩

Exit:- to instuction following the
DATA ⟨line number⟩

Registers Used:- All

R:R11 may be recoded to monitor the execution of an RTL/2 program, printing out line numbers as they are executed, or to provide greater information on error conditions,
e.g. the last 10 lines executed.

II.4.7. Compare LABELs and REALs

Module IDENT:- RTLR12

Control Routines:- R:R12, R:R13

II.4.7.1. R:R12 Compare LABELS

Called by:-

```
⟨label-1 target address to A1⟩
⟨label-1 LVP (A12 value) to A2⟩
⟨label-1 link cell return address to A3⟩
⟨pointer to label-2 to A8⟩
 CF  A14,  R:R12
```

Result:-

A10 zero if labels equal,
non-zero if labels unequal.

Exit:- to instruction following CF A14, R:R12.

Registers Used:- A11
(A1, A2, A3, A8 are preserved.)

II.4.7.2. R:R13 - Compare REALs

Called by:-

〈REAL-1 mantissa, more significant half, to A1〉
〈REAL-1 mantissa, less significant half, to A2.〉
〈REAL -1 exponent to A3.〉
〈Pointer to REAL-2 to A8〉
CF A14, R:R13

Result:-

A10 zero if REAL-1 = REAL-2
A10 +ve if REAL-1 > REAL-2
A10 -ve if REAL-2 > REAL-1

Exit:- to instruction following CF A14, R:R13.

Registers Used:- A11
(A1, A2, A3, A8 are preserved)

II.4.8. REAL Add and Subtract

Module IDENT:- RTLR14

Control Routines:- R:R14, R:R15

Functions:-

R:R14 adds REAL numbers

R:R15 subtracts REAL numbers

Called by:-

⟨operand-1 into A1, A2, A3⟩

⟨Pointer to operand-2 to A8⟩

CF A14, R:Rnn

Result:-

In A1, A2, A3. For R:R14 this is (operand-1 + operand-2). For R:R15 it is (operand-1 - operand-2).

Exit:- To instruction following the,

CF A14, R:Rnn.

Errors Detected:- Floating point overflow-

- error no. 6.

Registers Used:- A7, A9, A10, A11

(A8 preserved).

II.4.9. REAL Multiply

Module IDENT: RTLR16

Control Routine:- R:R16

Function:- Multiplies REAL numbers.

Called by:-

⟨operand-1 into A1, A2, A3⟩
⟨pointer to operand-2 to A8⟩
CF A14, R:R16

Result:- (Operand-1 * Operand-2)
In A1, A2, A3.

Exit:- to instruction following the,
CF A14, R:R16.

Errors Detected:- Floating point overflow
- error no. 6.

Registers Used:- A4, A5, A6, A7, A9, A10, A11
(A8 preserved).

II.4.10 REAL Divide

Module IDENT:- RTL217

Control Routine:- R:R17

Function:- Divides REAL numbers.

Called by:-

⟨operand-1 into A1, A2, A3⟩
⟨pointer to operand-2 into A8⟩
CF A14, R:R17

Result:- (Operand-1/Operand-2)
In A1, A2, A3.

Exit:- to instruction following the,
CF A14, R:R17.

Errors Detected: Floating Point overflow
- error no. 6.

Registers Used:- A4, A5, A6, A7, A9, A10, A11
(A8 preserved)

II.4.11. Fixed Point to REAL Conversions

Module IDENT:- RTLR18

Control Routines:- R:R18, R:R19, R:R20, R:R21, R:R22.

Functions:-

Convert fixed point forms to REAL.

R:R18 converts INT

R:R19 converts FRAC

R:R20 converts Big INT

R:R21 converts Big FRAC or Fine INT.

R:R22 converts Fine FRAC.

Called by:-

⟨operand to A1, and A2 if double length⟩
CF A14, R:Rnn

Where the operand is double length the more significant half is in A1.

Result:- (REAL (operand))
In A1, A2, A3.
Mantissa more significant part A1.
Mantissa less significant part A2.
Exponent in A3.

Exit:- to instruction following,
CF A14, R:Rnn

Registers Used:- None.

II.4.12. REAL to Fixed Point Conversions

Module IDENT:- RTLR23

Control Routines:- R:R23, R;R24, R;R25

Functions:-

Conversions from REAL to fixed point forms
R:R23 converts to FRAC
R:R24 converts to INT
R:R25 converts to BYTE

Called by:-

⟨operand to A1, A2, A3⟩
CF A14, R:Rnn

Result:- In A1. For R:R25 the BYTE is in the less significant half of A1 and the more significant half is zero.

Exit:- to instruction following the
CF A14, R:Rnn

Errors Detected:- Fixed point overflow on conversion - error no. 7.

Registers Used:- None.

II.4.13. ABS REAL and Negate REAL

Module IDENT:- RTLR26

Control Routines:- R:R26, R:R27

Functions:-

R:R26 Negates a REAL number
R:R27 performs ABS (REAL) number
i.e. negate if negative.

Called by:-

⟨operand into A1, A2, A3⟩
CF A14, R:Rnn

Result:- In A1, A2, A3.

Exit:- to instruction following the,
CF A14, R:Rnn.

Errors Detected:- Floating point overflow
- error no. 6.

Overflow is only possible when ABS or negate operates on the maximum negative REAL value, when it arises because one more negative value than positive can be held in a REAL.

Registers Used:- A7.

II.4.14 Overflow checking

Module IDENT: RTLR28

Control Routines:- R:R28

Functions:- Checks for overflow when OPTION () OV in use.

Called by:- CF A14, R:R28

Exit:- to instruction following the
CF A14, R:R28

Errors detected:- fixed-point overflow
- error no. 5

Registers used:- none

Called after any assembler instruction which may set overflow: add, subtract, multiply, divide, two's complement, increment and left-arithmetic-shift (possibly due to narrowing).
May be re-coded to take different action if overflow occurs.

II.5. OPTIONAL EXCLUSION

II.5.1. Introduction

When the control routines are link edited into a program or task from an object library selection of only those modules which are required will be automatic.

This section is provided as a guide to selecting a subset of the control routines when they are to be built as a shared area for use by tasks to be built later.

TABLE II.5.1. - Optional Requirement

Of Control Routines

MODULE IDENT	CONTROL ROUTINES	WHEN REQUIRED	NOTES
RTLROO	R:ROO	ALWAYS	
RTLRO1	R:RO1,R:RO2		
RTLRO3	R:RO3,R:RO4, R:RO5,R:RO6, R:RO7.	Whenever array bound checks apply.	(a)
RTLRO8	R:RO8	Whenever SHA,SHL operators used.	
RTLRO9	R:RO9 R:R1O	ALWAYS	
RTLR11	R:R11	If OPTION () TR used	
RTLR12	R:R12,R:R13	If REAL or LABEL comparisons made	
RTLR14	R:R14,R:R15	If REAL addition or subtraction used	(b) (c)
RTLR16	R:R16	If REAL multiply used	(b) (c)
RTLR17	R:R17	If REAL divide used	(b) (c)
RTLR18	R:R18,R:R19, R:R2O,R:R21, R:R22	If any conversions from fixed point to REAL.	(c)
RTLR23	R:R23,R:R24, R:R25	If any REAL to INT, FRAC or FRAC conversions.	(c)
RTLR26	R:R26,R:R27	If ABS or negate operators used on REAL.	(b) (c)
RTLR28	R:R28	If OPTION() or used.	

TABLE II.5.1. - Notes

Note:

(a) Array bound checks apply if,

(i) OPTION ()BS or OPTION ()BC used

or (ii) Arrays used in applications RTL/2 where any other action than reading from ARRAYs of INT, BYTE, FRAC or REAL is involved.

(b) R:R15,R:R16 and R:R17 all call R:R26 and/or R:R27 so RTLR26 must be included if any of RTLR214, RTLR216 or RTLR217 are.

(c) On machines with floating point hardware RTLR14, RTLR16, RTLR17, RTLR18, RTLR23, may all be omitted if OPTION () FP is used at all times.

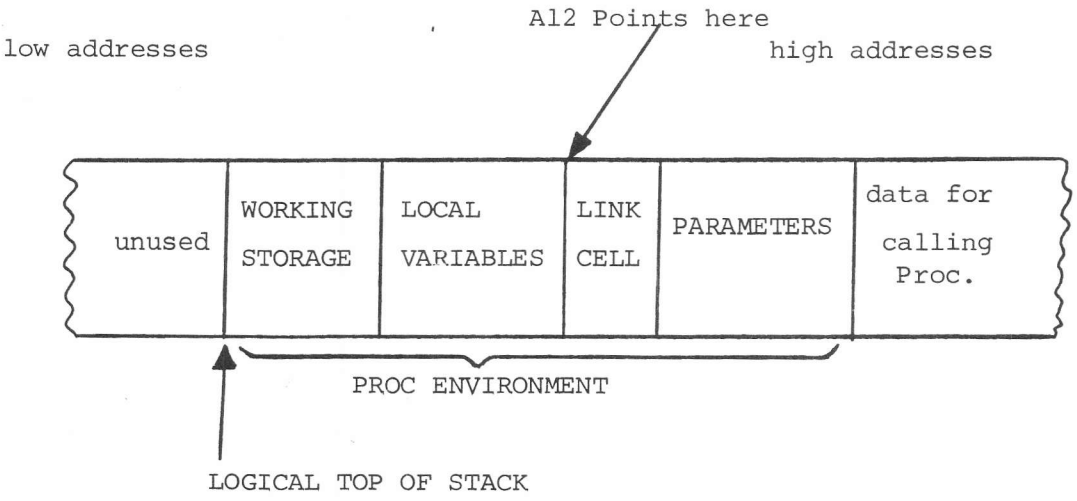
III.1. INTRODUCTION

This appendix discusses the conventions of the assembler code used on the P800 series, both compiler generated code and hand-written code - so that a user may understand the compiled code of his system. It also describes how to write code statements in RTL/2 modules when this is necessary, in order, for example, to drive peripheral devices. We start by describing in detail, the utilisation of the stack at run-time.

III.2 THE STACK MECHANISM

It is assumed that the reader is familiar with the general notion of a run-time stack which mirrors at any one time the dynamic state of a task, i.e. the nested structure of procedure calls which exist at that time - together with the local data those procedures are using.

Program workspace grows by working downwards from the high addressed end of the stack with procedure data for successively nested procedure calls.
The stack data for each procedure is arranged on each side of a LINK CELL as shown:



III.2.1. The LINK CELL

The link cell contains 2 words:

The lower addressed word contains the address of the link cell (word 0) of the procedure which called this procedure. It is pointed at by register A12 and thus contains the value A12 is to take on procedure exit.
The higher addressed word contains the link address, i.e. the value register P is to take on procedure exit.

III.2.2. LOCAL VARIABLES

Local Variables are those declared within the current RTL/2 procedure and for loop control data. This data is stored as it would be in a DATA brick except that BYTE variables occupy whole words - the more significant half of the word is unused. Local variables are addressed using indexed mode on A12. They may be overwritten after the procedure has completed.

III.2.3. PARAMETERS

These are the RTL/2 parameters, stored in the stack before calling the current procedure. Within the procedure they behave as do the local variables.

III.2.4. WORKING STORAGE

The amount of working storage in use is variable in size, being zero on entry to a procedure and accommodating temporary data (e.g. partially calculated expressions, temporary dumps of registers) as required. All data is added or removed from this area by use of the local variable pointer, A12.
A12 is set up to point to the current link cell on procedure entry. An important case of working storage use is when one procedure calls another. Parameters for the called procedure are placed in the working storage of the calling procedure, which then enters the called procedure. The latter uses control routine R:RO1 which creates the new environment and sets A12 accordingly, the parameters in working store becoming identical with the parameter area of the new procedure. Any procedure result is left in the registers A1 upwards on exit from a procedure. The procedure exit control routine, R:RO2, adjusts A12 and returns to the point of the procedure call.

III.2.5. Dummy Link Cell

The high-addressed end of the stack region contains a single word pointing to itself, which serves as a dummy (half) link cell to end the chain of link cells pointed to by A12.

III.2.6. Example of Stack Usage

In an RTL/2 task where:-

(a) the most basic procedure is PROC A();

(b) A calls PROC B(INT X, REAL Y);

(c) B calls PROC C(INT P, BYTE Q, LABEL R) INT;

to compute an integer result,

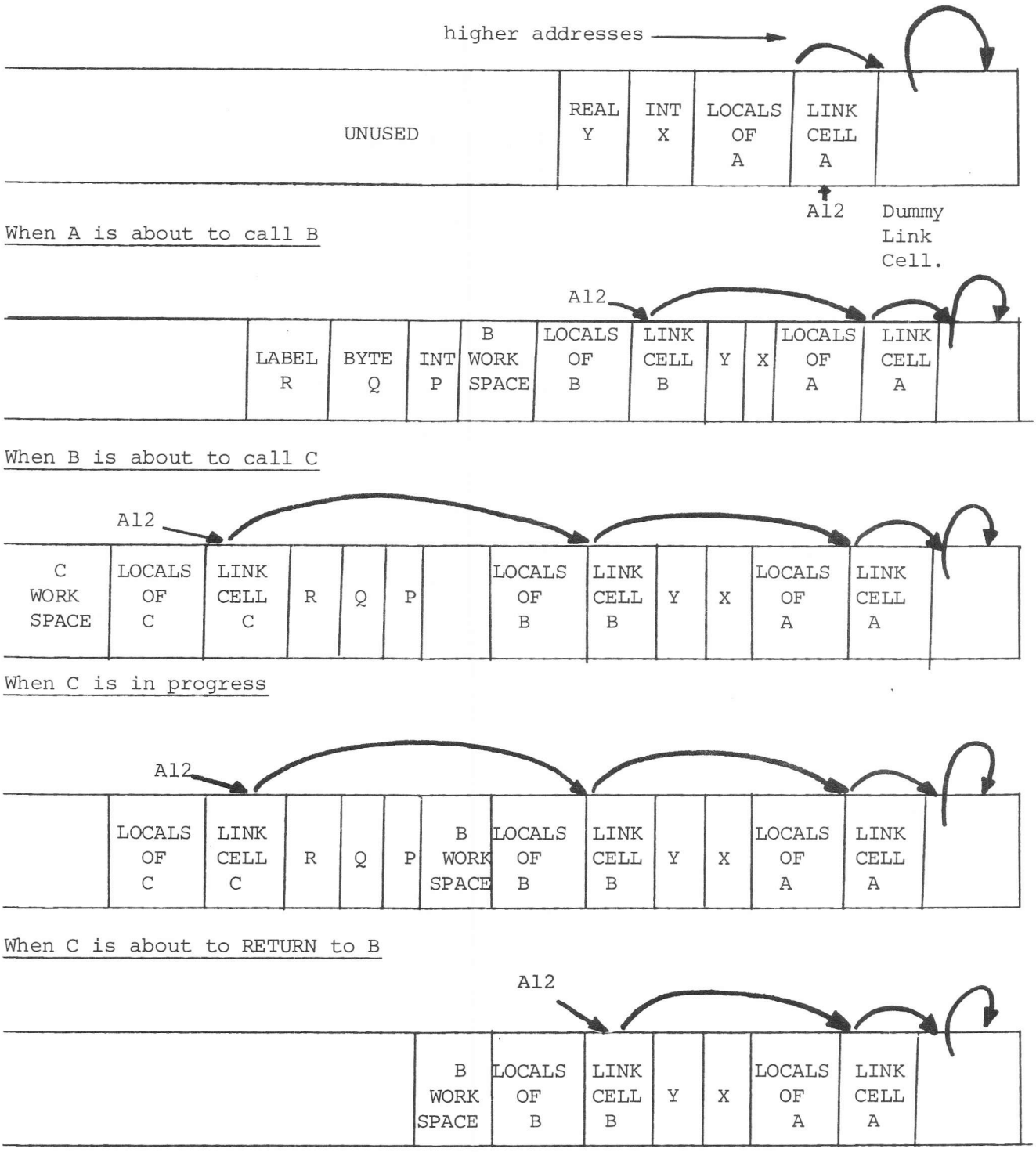
the stack would look as shown in FIG.III.2.1.

The arrows show the pointers used to unwind the stack when procedures return (by RETURN ENDPROC), or GOTO statements with a LABEL variable are executed.

The rightmost dummy link cell terminates the chain of pointers.

Note that the same principles apply if a procedure is recursive, i.e. calls itself directly or indirectly. A new link cell, local variables and workspace will be added, regardless of whether the called procedure is already active or not.

FIG. III.2.1. - STACK Mechanism Example



III.2.7. A14 Area

A14 is used when calling the control routines and any FORTRAN subroutines. It points to a logically distinct stack area which does not have to be part of the stack used by the RTL/2 compiled code.

III.3. STACK SIZE ESTIMATION

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

Two techniques may be used:

(i) Initial over-estimate. The task may be run initially with an over large stack. An integer in SVC DATA is used by the procedure entry control routine(R:RO1) and is updated to the lowest point in the stack reached by the linkcell/working storage used. After the task has run for long enough for the user to be confident that all possible routes have been taken through the various procedures, (including error procedures), this location may be inspected. 200 bytes is adequate for tasks of straight-forward type. Note that recursive procedures can use more stack than you expect. If IWRT or IWRTF is used for example it should be tested with the maximum number of significant digits that the task can produce, e.g. 5.

(ii) Accurate calculation. It is possible to calculate the displacement between successive link cells by inspection of compiled (assembly) code, and thereby to calculate (by inspection of all possible routes) the longest stack usage. This is tedious in complex programs. The method is as follows:

Add up the ⟨n⟩ values (see II.4.2.1.) for all procedures in a calling chain - allow for recursion by multiplying by the maximum depth of recursion - and find the maximum value this total can have, i.e. the stack usage of the most demanding calling chain. Remember that this figure is the no. of words of stack, and that it must be added to the allocations made by the base program for SVC DATA bricks and the A14 area used by the control routines.

III.4 REGISTER CONVENTIONS

The usage of the P800 registers in compiled RTL/2 code is as follows. Register usage within particular control routines is specified in detail in II.4.

III.4.1 Register 0(P) is the program counter.
Register A15 is the hardware stack pointer. P and A15 are not referred to explicity in the compiled code.

III.4.2 Registers A1,A2,A3 are used together for:-

(a) Passing back procedure results of all types.
(b) Parameters, particularly REAL or LABEL, to control routines.
(c) Transiently to move a REAL or LABEL to or from the stack.

Registers A1,A2 are also used for:-

(d) Double length operations, whether involving control routines or not.

Register A2 is also used for:-

(e) Holding subscripts and addresses during array addressing without bound checking.

Register A3 is also used for:-

(f) Generation of variable length shift instructions which are executed by EXR A3.
(g) Holding subscripts during array addressing with bound checking.

III.4.3. Registers A4,A5,A6 and A7 are used together:-

(a) As an extension of the workspace part of the stack. The simplest expressions involving no procedure calls and no conditions do not require any workspace in store. Any registers in use as a "stack extension" must be dumped into the stack before procedure calls or branches (in conditional expressions) or control routines which use A4,A5,A6 or A7. This dumping is performed by instructions inserted by the compiler.

Register A6 is used:-

(b) During procedure entry as a pointer to the base of the parameters, and thus to where the new link cell should be planted. A6 is used in the CF instructions used to enter RTL/2 procedures and the procedure entry control routine, R:RO1.

III.4.4 Registers A7 and A8 are used:-

(a) To hold parameters of SVC PROC calls. (see III.7.6.).

Register A8 is also used:-

(b) As a pointer to the second operand when diadic operations are performed on data types that occupy more than one word.

(c) Transiently to hold a byte during byte arithmetic.

III.4.5 Registers A9,A10,A11 are used:-

(a) Transiently within the compiled code

and

(b) By the control routines, also transiently.

Register A9 is also used:-

(c) To hold an address, either an intermediate address in a complicated expression involving records, or as the result of an array-bound checking control-routine.

Register A10 is also used:-

(d) To hold an address, either an intermediate address in a complicated expression involving variable subscripts, or as a parameter to an array-bound checking control-routine.

III.4.6 Register A12 is dedicated to being the local variable pointer. It points to the link cell of the currently executing procedure and is used to access parameters, local variables and stack workspace.

III.4.7 Register A13 is dedicated to being the SVC DATA pointer. It is used for all access to SVC DATA brick items.

III.4.8 Register A14 is dedicated to pointing to the control routine area of the stack. Control routines (except R:RO1 and R:RO2) are entered by CF A14,... instructions. The size of the control routine area of the stack includes an allowance for the standard error procedure, RRGEL, (which may use the A14 part of the stack to successfully monitor an overflow of the A12 part).

If routines in other languages are to be called from RTL/2 by CF A14,... (a bridging procedure will be necessary) then the stack requirements of using such routines must be added to the "A14 allocation".

III.5. RTL/2 DATA FORMATS

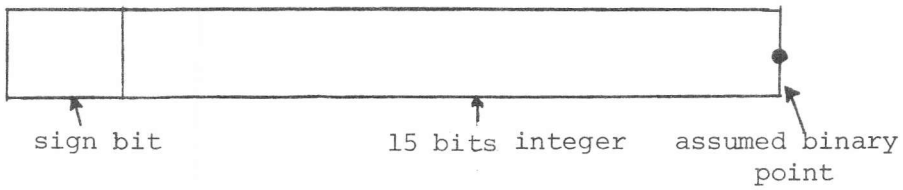
The standard RTL/2 data types are implemented on P800 as follows:

III.5.1. BYTE

An RTL/2 BYTE is a byte of P800 storage. Note that RTL/2 treats all BYTE variables as unsigned which is usually the case in P800 character instructions. Bytes within a word addressed left (even address) to right (odd address). RTL/2 BYTE storage follows this convention.

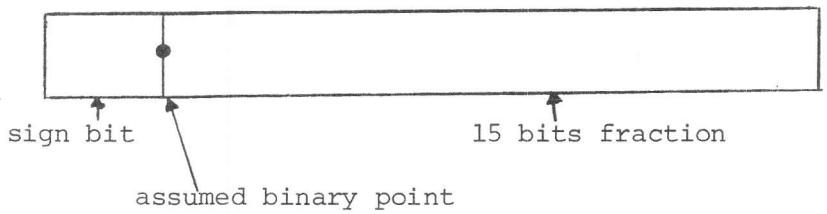
III.5.2. INT

An RTL/2 INT is represented by a P800 word in the standard 2's complement form:



III.5.3. FRAC

An RTL/2 FRAC is represented by a 2's complement 16-bit word thus:

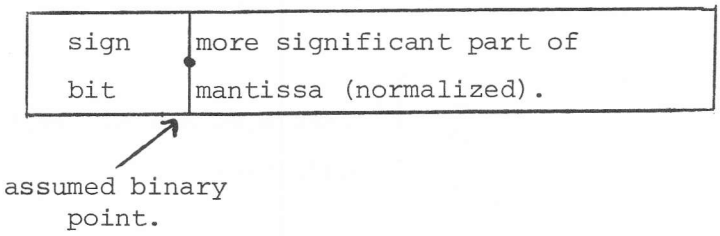


Thus 0.5BO is represented by Hexadecimal 4000
-1.0BO is represented by Hexadecimal 8000

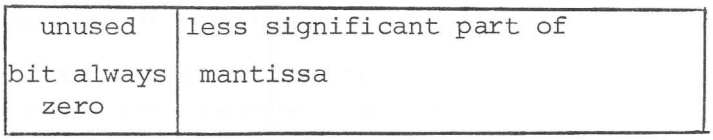
III.5.4. REAL

The standard 3-word format of P800 is used, conforming to the floating point hardware.

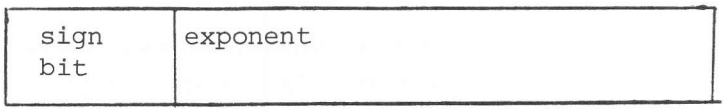
LOWEST ADDRESSED WORD.



NEXT ADDRESSED WORD



HIGHEST ADDRESSED WORD



exponent as 2's complement integer.

III.5.4.1. Negative Real Numbers are held with a negative (2's complement of double length) mantissa, and a positive or negative exponent as appropriate.

III.5.4.2. Zero is represented as 3 words of zero (i.e. 0.0*(2**0)).

III.5.5. REF, PROC, STACK variables are all represented as 16-bit addresses.

III.5.6. LABEL

Three words are used:-

Lowest addressed word:

address to go to (new value of 'p')

Next addressed word:

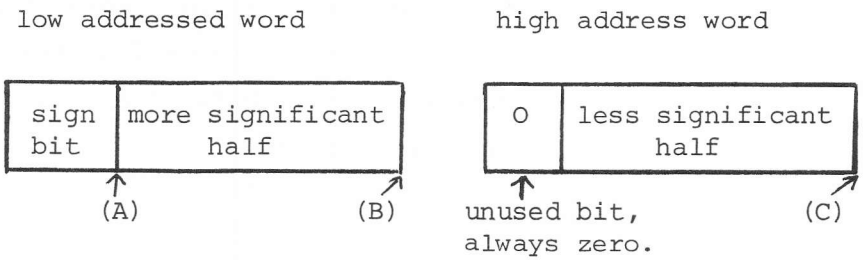
link cell address (new value of A12)

Highest addressed word:

return address of the procedure in which the LABEL was set.

III.5.7. Intermediate Modes

All the transient, double-length, forms use the standard double length arithmetic format of P800.



Only the position of the assumed binary point differs:-

For fine FRAC it is at (A).

For fine INT and big FRAC it is at (B).

For big INT it is at (C).

III.5.8. RECORDS

RTL/2 records are laid out as a succession of components each with its own format as described in III.5.1.-7 and III.5.9. The record has no extra data structure, except that padding bytes are inserted, if necessary, to ensure that all non-byte components start at an even address. Padding is also inserted so that all records, except those containing only bytes (not arrays of bytes) always start at an even address.

III.5.9. ARRAYS

Arrays are represented as follows:

where A(1),A(2), etc. are the elements and have their type and structure as already defined. L is a single word defining the length of the array as the number of elements. L is a 16-bit integer and is stored at an even address. Array elements are all stored at even addresses except when the array is an array of BYTEs or an array of records which only contain BYTEs; other arrays of records may thus contain padding bytes. The address of an array (i.e. a REF ARRAY) is the theoretical address of the zeroth element. Only in arrays of one-word elements is this the same as the address of the length word.

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

III.6. BRICK LAYOUT

RTL/2 bricks are compiled as follows.

III.6.1. PROC bricks (other than SVC PROC)

An RTL/2 PROC is compiled with a call to control routine R:RO1 at the head. This routine sets up register A12 as appropriate. RETURN statements, wherever they occur are compiled as calls to control routine R:RO2 whether or not a result is returned. Details of control routines are given in Appendix II.

Local variables used by the procedure are implemented in the stack. They are thus addressed using register A12, in indexed mode. Data brick variables may be addressed directly, unless the data brick is an SVC DATA brick; in the latter case variables must be addressed using register A13 in indexed mode.

III.6.2. DATA bricks

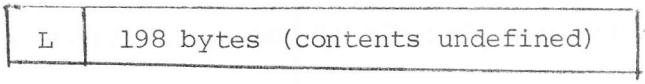
An RTL/2 DATA brick is compiled with the data laid out in the order defined in the RTL/2 text. A padding byte is inserted if necessary before any item which must start at an even address.

SVC DATA bricks are considered to be mapped in the same way, although SVC DATA bricks cannot be addressed other than via register A13 (to ensure that the appropriate task's copy is used), and cannot be allocated storage by an RTL/2 definition of them.

III.6.3. STACK Bricks

An RTL/2 stack brick is compiled as an uninitialised block of core, with the length of the remaining space, in bytes, in the first word.

N.B. the RTL/2 statement STACK FRED 2ØØ
produces the following:



length word,
containing the
number 198.

The stack, if referred to by a STACK variable, is addressed by the length word address.

III.6.4. SVC PROC bricks.

There is no RTL/2 means of compiling an SVC PROC brick.

See section III.7.6. for details of the code compiled for SVC PROC calls.

III.7. CODE STATEMENTS

This section summarises the main features of writing CODE statements in RTL/2 modules. Familiarity with the assembler language of the P800 series is assumed.

III.7.1. General Principles

CODE statements or 'code sequences' have a syntax which follows the overall standard as described in the RTL/2 Language Specification Manual thus:

```
codeseq ::= codeheading codeitem
codeheading ::= digitlist, digitlist;
codeitem ::= ISO7-character-other-than-trip-1-or-trip-2|
             trip-1 letitem|name.
letitem ::= name|number|string|comment|separator.
```

In the P800 implementation the characters 'trip1' and 'trip2' of the specification manual are & and @ respectively.
The trip characters are used to access RTL/2 defined variables or LET names, etc. from within a CODE statement, as described below. In addtion to the two trip characters, the presence of a pair of dollar signs ("$$") has special significance in this implementation. (See III.7.7.)

III.7.2. CODE body

A code statement is, as far as an RTL/2 compiler is concerned similar in nature to other statements. When it is entered during the execution of a procedure there will be no working storage in use on the stack (i.e. the workspace area and registers A4, A5,A6 and A7 will be free) and none should remain in use when it is completed. The compiler will make no assumptions about the contents of registers except A12,A13,A14 on exit from a code statement.

III.7.2.1. CODE heading information

The programmer must tell the compiler how many bytes of storage to reserve in-line for the assembler code between "CODE" and "&RTL". This length must be exact.

Secondly he must state how much workspace in the RTL/2 stack is to be used by the CODE statement in the worst case. This is so that this requirement will be checked (when the procedure containing the CODE statement is entered) by control routine R:RO1.

These two figures appear after the keyword, CODE , thus:-

```
CODE 42,2;
     .
     .
     .
&RTL;
```

is a code sequence occupying 42(decimal) bytes, and using, at most, 2 bytes of stack workspace.

III.7.2.2. Use of RTL/2 Items

In between CODE ....; and &RTL; the programmer may write any legal assembler statements. Care should be taken to conform to the assembler statements layout requirements. No editing, other than that described here in response to trip characters is performed by the compiler.

Trip-1 ('&'), trip-2 ('@') and "$$" are recognized by the compiler and displacements, values or symbols are substituted for RTL/2 items. The following sections describe this in detail. Note that it is the programmer's responsibility to use RTL/2 items in a correct manner; for example no automatic dereferencing will be added by the compiler.

III.7.3. Local Variables and Parameters

Any RTL/2 defined local variable or parameter of the current procedure (including FOR loop control variables, and those declared in inner BLOCKs or FOR loops, where in scope) are accessible in CODE sections.

The compiler substitutes in response to:-

```
& variable-name
```

the displacement of the variable from the current link cell. The displacement will be positive for parameters, negative for other locals.

Thus to access a local INT x we can use:-

(i) `LD A4,&X,A12`

since A12 points to the link cell.

The address of the variable may be calculated, if required, by:-

```
(ii) LDKL  A4,&X
     ADR   A4,A12
```

leaving A4 pointing to X.

In both examples the compiler will substitute a numeric displacement, so that if X were -2 bytes from the link cell:-

```
     LD    A4,-2,A12 is produced for (i),
```

and

```
     LDKL  A4,-2
     ADR   A4,A12    is produced for (ii).
```

III.7.4. DATA Brick Variables (not SVC DATA)

Data brick variables may be addressed directly from within CODE statements by:-

& variable name @ data brick name

The compiler substitutes the offset within the brick for the variable. The data brick name is not modified if EXT or ENT , otherwise a compiler-generated symbol (by which the data brick is known in the compiled code) is substituted.

For example, given:-

```
EXT DATA DBX;
   :
   REF INT B;
   :
ENDDATA;
DATA DBLOCAL;
   :
   INT C;
   :
ENDDATA;
```

the sequence:-

```
CODE 8,0;
   LDKL  A4,&C@DBLOCAL
   ST    A4,&B@DBX
&RTL;
```

might compile into:-

```
   LDKL  A4,R:342+/10
   ST    A4,DBX+/4
```

III.7.5. SVC DATA Items

Variables in SVC DATA bricks are denoted in CODE sequences as for DATA bricks:-

```
& variable name @ brick name.
```

and a similar symbolic expression will be substituted. However, the brick address will be the offset of the brick from the start of the SVC DATA area, register A13 must be used in indexed mode to access the variable.

To access ERN in the standard SVC DATA brick:-

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

the following might be used:-

```
LD A4,&ERN@RRERR,A13
```

giving in the compiled code:-

```
LD A4,RRERR+/6,A13
```

RRERR is defined at link edit time.

III.7.6. SVC PROC calls

Calls to SVC PROCedures are compiled as:

```
1st parameter (if two) to A7
2nd or only parameter to  A8
LKM
DATA PRNAME
```

where PRNAME is the name given to the RTL/2 SVC PROC, and is resolved at link edit time to be the correct LKM number.

If the procedure is declared as returning an integer result, the result will be assumed to be in A7.

III.7.7. Workspace

Workspace on the current RTL/2 stack may be used as required in a code section, as long as the worst-case usage is recorded as the second number after the key-word CODE (see II.7.2.)

The special trip character sequence of "$$" may be used to access workspace locations, as follows:-

```
ST A1,$$,A12
ST A2,$$+2,A12
```

"$$" is translated by the compiler to the (negative) offset of the lowest addressed word of workspace available to the CODE section. This is calculated as minus (local space + CODE workspace).

Failure to declare such workspace in the CODE heading may result in the assembler written statements corrupting the Local variables or link cells of the current or other active procedures.

III.7.8. Record Components

An RTL/2 component selector may be used to address a record component. If we have:

```
MODE COMPLEX (INT RL, IM)
```

and we have in a data brick, DATATHREE:

```
COMPLEX C
```

then C. IM may be moved into register A1, in a code section, thus:

```
LD A1,&C@DATATHREE + &IM@COMPLEX
```

Note then &C@DATATHREE gives the address of the first byte in C. &IM@COMPLEX gives the displacement within any record of mode COMPLEX (2 bytes in this simple case).

Record structure layouts need not, therefore, be known explicity by the CODE section author.

The length of a record may also be retrieved symbolically, which can be useful in addressing arrays of records. The length of any record of mode COMPLEX will be inserted in a code statement in place of:

```
&COMPLEX
```

Note that record lengths will always be padded out to a whole number of bytes unless the record only contains bytes.

III.7.9. Array Element and LENGTH.

Array elements are always stored at an address:

Array address + (element number * element size)

Array addresses, i.e. the address the compiler always substitutes for an array identifier, are adjusted so that this is so. An array address is thus the byte address of a non-existent zeroth element of the array.

III.7.9.1. For example, to load the BYTE, AB(I), given an ARRAY () BYTE AB in a DATA brick, DATAFOUR, and a local INT I, we might use:-

```
LD  A9,&I,A12
LC  A1,&AB@DATAFOUR,A9
```

For arrays of other types the subscript would need multiplication by the element size before use as an address index.

III.7.9.2. In a more elaborate case of a record array:

```
MODE COMPLEX (INT RL, IM)
ARRAY (10) COMPLEX CC in DATATHREE then
```

CC(2).IM could be accessed thus:

```
LD A1,&CC@DATATHREE+&COMPLEX+&COMPLEX+&IM@COMPLEX
```

the 2nd element being 2 record lengths on from the array address.

III.7.9.3. Array LENGTH words may be accessed at an address two bytes lower than that of the first array element. Only in the case of INT, FRAC,REF, PROC or STACK arrays is this the same as the array address.

III.7.10. Bricks

The starting address of a DATA or PROC brick may be accessed simply by trip-1 then RTL/2 identifier, eg.

```
LDKL   A1,& BRICKNAME
```

III.7.11. Control Routines

These may be used without restriction. External references should not be written to satisfy the linkage editor since they are output by the RTL/2 compiler.

Refer to Appendix II for detailed specifications of the control routines. When called from CODE sections particular attention must be paid to register usage.

III.7.12. Procedure Calls

III.7.12.1. Procedure bricks may be called from CODE sections by their RTL/2 names e.g.

```
CF  A6,& PROCNAME
```

III.7.12.2. A6 should be first initialized to point to the "$$+6" word and 8 bytes of stack workspace should be included in the CODE heading for the CF instructions to PROCNAME and from PROCNAME to R:RO1.

III.7.12.3. Parameters must be stored in the stack workspace and Parameter space should also be added to the workspace requirement so that, for example, a call to

```
PROC PQR (INT I,J);
```

from a CODE section might look like the following.

```
CODE ...,...;
     1st parameter to A1     (say)
     2nd parameter to A2     (say)
     ST   A1,$$+10,A12
     ST   A2,$$+8,A12
     LDKL A6,$$+6
     ADR  A6,A12
     CF   A6,&PQR
&RTL;
```

III.7.12.4. Procedure results may be accessed directly from A1 (plus A2,A3 if multiple word results). The parameters will be intact after return and can be accessed at the same offsets from A12, although their values may have been modified by the called procedure.

III.7.12.5. Procedure variables are used in the same way as any other data variables. When used to call procedures in CODE sections the indirection must be added (Perhaps by use of CFI rather than CF).

III.7.13 RTL/2 Labels

Label variables are accessed in the same way as other variables.

Literal labels outside the code statement may be accessed using the usual '&' facility, e.g.

```
ABL &FAIL
```

causes a jump to the RTL/2 label FAIL, which must be in the same procedure brick. Literal labels may also be defined inside code statements so as to be rendered accessible to RTL/2 statements within the same brick, thus:

```
&LI&:
```

may appear in a CODE section allowing one to write:

```
GOTO L1 or LABELVARIABLE: = L1
```

elsewhere in the same procedure.

III.7.14 Constants

III.7.14.1 Constants may be referred to in their RTL/2 form within a code statement. This has two advantages:

(i) Constants named with RTL/2 LET statements may be used under their LET names - thus ensuring that code sections are automatically edited if LET statements are changed.

(ii) The user can be saved the tedium of calculating internal representation of real or fractional constants and can use, for example, octal and binary forms of integers not available in the assembly language.

REAL, INT and FRAC constants may be written as 'trip-l' 'constant' where 'constant' has its usual RTL/2 syntax.

III.7.14.2. A string is translated into the symbolic address in the string pool of the (theoretical) zeroth element of the string (which is stored as a BYTE array). The string will be added to the string pool if it is not already present.

III.7.14.3. Example of Constants in CODE

RTL/2 text:-

```
LET   NTASK=12;

CODE  ...,...;

  LDKL      A1,&NTASK
  LDKL      A2,&"STRING"

&RTL;
```

Output from Compiler

```
LDKL        A1,/C
LDKL        A2,R:POOL+,,,,
```

where the string pool, R:POOL, contains

```
DATA        'STRING'.
```

III.7.15. Summary

The transformations which the compiler makes to RTL/2 items in code statements are as below. Programmers may use them in any way which is valid assembler, bearing in mind the meanings which the table below implies.

RTL/2 Text	Corresponding Assembler
&integer	literal value in octal
&fraction	
&string	symbolic address of conceptual zero element of string in pool
&name&:	label followed by : (colon)
&&,&@	&,@ respectively
&identifier	depends on use of identifier as below
&modename	literal value of length of mode in decimal
&brickname	symbolic address of start of brick
&literallabelname	symbolic address of label
&localname	displacement of variable from current linkcell (i.e. from contents of register 5)
&component@mode	displacement of component from start of record
&globalname @databrick	symbolic address of variable
$$	offset from A12 of the lowest addressed workspace word available to CODE section.

IV.1. INTRODUCTION

This appendix gives details of SVC DATA bricks expected by the control routines and RTL/2 base program, as supplied for Philips P800.

IV.2. STANDARD RTL/2 BRICKS

These bricks are described in RTL/2 Standards.

IV.2.1. RRSIO

Initial values of IN and OUT are RRIPF, RROPF respectively which procedures call RRGEL with error numbers of 98 and 99 respectively.

IV.2.2. RRSED

Initial values in RRSED are:-

IOFLAG : = 0

TERMCH : = HEX 80 (end-of-stream)

IV.2.3. RRERR

Initial values in RRERR are:-

ERL, points to exit from program

ERN : = 0

ERP : = RRGEL.

IV.3. NON-STANDARD BRICKS

IV.3.1. RRSTK

This has the specification:-

```
SVC DATA RRSTK;
    INT  STKLØ,
         STKLIM
         WSPBAS;
ENDDATA;
```

STKLO is the lowest reached stack addressed since the program began. Its initial value is the highest address in the stack. It is used to monitor maximum stack usage.

STKLIM is the low address limit of the stack. It is used to check for stack overflow.

WSPBAS is the address of the low end of the current procedure workspace. It is used to provide a dump of the procedure workspace.

The addresses stored in STKLO, STKLIM, and WSPBAS are all word addresses and point to one word below the lowest used, lowest allowed and lowest workspace word respectively.

IV.3.2. RRERRX

This has the specification:-

```
SVC DATA RRERRX;
    INT LINENO;
ENDDATA;
```

LINENO is the latest RTL/2 line number when the TR option is in use. Its initial value is zero.